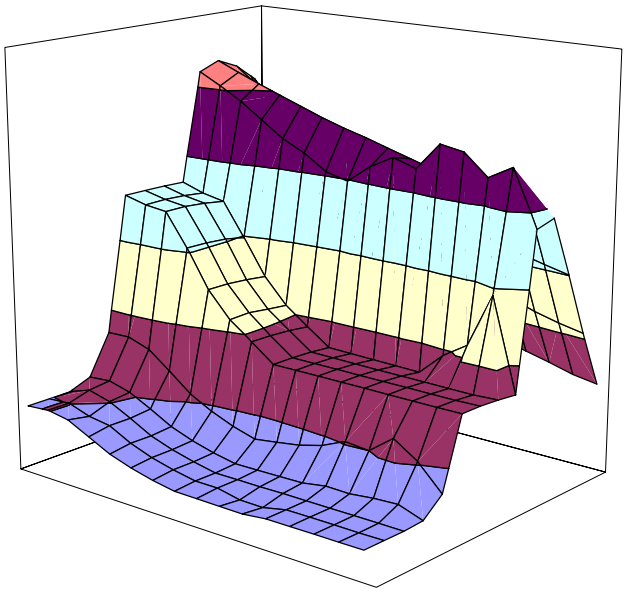
Computer Systems

A Programmer's Perspective [1]

(*Beta Draft*)



Randal E. Bryant
David R. O'Hallaron

November 16, 2001

Contents

Preface

This book is for programmers who want to improve their skills by learning about what is going on "under the hood" of a computer system. Our aim is to explain the important and enduring concepts underlying all computer systems, and to show you the concrete ways that these ideas affect the correctness, performance, and utility of your application programs. By studying this book, you will gain some insights that have immediate value to you as a programmer, and others that will prepare you for advanced courses in compilers, computer architecture, operating systems, and networking.

The book owes its origins to an introductory course that we developed at Carnegie Mellon in the Fall of 1998, called *15-213: Introduction to Computer Systems*. The course has been taught every semester since then, each time to about 150 students, mostly sophomores in computer science and computer engineering. It has become a prerequisite for all upper-level systems courses. The approach is concrete and hands-on. Because of this, we are able to couple the lectures with programming labs and assignments that are fun and exciting.

The response from our students and faculty colleagues was so overwhelming that we decided that others might benefit from our approach. Hence the book. This is the Beta draft of the manuscript. The final hard-cover version will be available from the publisher in Summer, 2002, for adoption in the Fall, 2002 term.

Assumptions About the Reader's Background

This course is based on Intel-compatible processors (called "IA32" by Intel and "x86" colloquially) running C programs on the Unix operating system. The text contains numerous programming examples that have been compiled and run under Unix. We assume that you have access to such a machine, and are able to log in and do simple things such as changing directories. Even if you don't use Linux, much of the material applies to other systems as well. Intel-compatible processors running one of the Windows operating systems use the same instruction set, and support many of the same programming libraries. By getting a copy of the Cygwin tools (`http://cygwin.com/`), you can set up a Unix-like shell under Windows and have an environment very close to that provided by Unix.

We also assume that you have some familiarity with C or C++. If your only prior experience is with Java, the transition will require more effort on your part, but we will help you. Java and C share similar syntax and control statements. However, there are aspects of C, particularly pointers, explicit dynamic memory allocation, and formatted I/O, that do not exist in Java. The good news is that C is a small language, and it

is clearly and beautifully described in the classic "K&R" text by Brian Kernighan and Dennis Ritchie [37]. Regardless of your programming background, consider K&R an essential part of your personal library.

> **New to C?**
> To help readers whose background in C programming is weak (or nonexistent), we have included these special notes to highlight features that are especially important in C. We assume you are familiar with C++ or Java. **End**

Several of the early chapters in our book explore the interactions between C programs and their machine-language counterparts. The machine language examples were all generated by the GNU GCC compiler running on an Intel IA32 processor. We do not assume any prior experience with hardware, machine language, or assembly-language programming.

How to Read This Book

Learning how computer systems work from a programmer's perspective is great fun, mainly because it can be done so actively. Whenever you learn some new thing, you can try it out right away and see the result first hand. In fact, we believe that the only way to learn systems is to *do* systems, either working concrete problems, or writing and running programs on real systems.

This theme pervades the entire book. When a new concept is introduced, it is followed in the text by one or more *Practice Problems* that you should work immediately to test your understanding. Solutions to the Practice Problems are at the back of the book. As you read, try to solve each problem on your own, and then check the solution to make sure you're on the right track. Each chapter is followed by a set of *Homework Problems* of varying difficulty. Your instructor has the solutions to the Homework Problems in an Instructor's Manual. Each Homework Problem is classified according to how much work it will be:

Category 1: Simple, quick problem to try out some idea in the book.

Category 2: Requires 5–15 minutes to complete, perhaps involving writing or running programs.

Category 3: A sustained problem that might require hours to complete.

Category 4: A laboratory assignment that might take one or two weeks to complete.

Each code example in the text was formatted directly, without any manual intervention, from a C program compiled with GCC version 2.95.3, and tested on a Linux system with a 2.2.16 kernel. The programs are available from our Web page at `www.cs.cmu.edu/~ics`.

The file names of the larger programs are documented in horizontal bars that surround the formatted code. For example, the program

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

code/intro/hello.c

can be found in the file `hello.c` in directory `code/intro/`. We strongly encourage you to try running the example programs on your system as you encounter them.

There are various places in the book where we show you how to run programs on Unix systems:

```
unix> ./hello
hello, world
unix>
```

In all of our examples, the output is displayed in a roman font, and the input that you type is displayed in an italicized font. In this particular example, the Unix shell program prints a command-line prompt and waits for you to type something. After you type the string "`./hello`" and hit the `return` or `enter` key, the shell loads and runs the `hello` program from the current directory. The program prints the string "`hello, world\n`" and terminates. Afterwards, the shell prints another prompt and waits for the next command. The vast majority of our examples do not depend on any particular version of Unix, and we indicate this independence with the generic "`unix>`" prompt. In the rare cases where we need to make a point about a particular version of Unix such as Linux or Solaris, we include its name in the command-line prompt.

Finally, some sections (denoted by a "*") contain material that you might find interesting, but that can be skipped without any loss of continuity.

Acknowledgements

We are deeply indebted to many friends and colleagues for their thoughtful criticisms and encouragement. A special thanks to our 15-213 students, whose infectious energy and enthusiasm spurred us on. Nick Carter and Vinny Furia generously provided their malloc package. Chris Lee, Mathilde Pignol, and Zia Khan identified typos in early drafts.

Guy Blelloch, Bruce Maggs, and Todd Mowry taught the course over multiple semesters, gave us encouragement, and helped improve the course material. Herb Derby provided early spiritual guidance and encouragement. Allan Fisher, Garth Gibson, Thomas Gross, Satya, Peter Steenkiste, and Hui Zhang encouraged us to develop the course from the start. A suggestion from Garth early on got the whole ball rolling, and this was picked up and refined with the help of a group led by Allan Fisher. Mark Stehlik and Peter Lee have been very supportive about building this material into the undergraduate curriculum. Greg Kesden provided

helpful feedback. Greg Ganger and Jiri Schindler graciously provided some disk drive characterizations and answered our questions on modern disks. Tom Stricker showed us the memory mountain.

A special group of students, Khalil Amiri, Angela Demke Brown, Chris Colohan, Jason Crawford, Peter Dinda, Julio Lopez, Bruce Lowekamp, Jeff Pierce, Sanjay Rao, Blake Scholl, Greg Steffan, Tiankai Tu, and Kip Walker, were instrumental in helping us develop the content of the course.

In particular, Chris Colohan established a fun (and funny) tone that persists to this day, and invented the legendary "binary bomb" that has proven to be a great tool for teaching machine code and debugging concepts.

Chris Bauer, Alan Cox, David Daugherty, Peter Dinda, Sandhya Dwarkadis, John Greiner, Bruce Jacob, Barry Johnson, Don Heller, Bruce Lowekamp, Greg Morrisett, Brian Noble, Bobbie Othmer, Bill Pugh, Michael Scott, Mark Smotherman, Greg Steffan, and Bob Wier took time that they didn't have to read and advise us on early drafts of the book. A very special thanks to Peter Dinda (Northwestern University), John Greiner (Rice University), Bruce Lowekamp (William & Mary), Bobbie Othmer (University of Minnesota), Michael Scott (University of Rochester), and Bob Wier (Rocky Mountain College) for class testing the Beta version. A special thanks to their students as well!

Finally, we would like to thank our colleagues at Prentice Hall. Eric Frank (Editor) and Harold Stone (Consulting Editor) have been unflagging in their support and vision. Jerry Ralya (Development Editor) has provided sharp insights.

Thank you all.

Randy Bryant
Dave O'Hallaron

Pittsburgh, PA
Aug 1, 2001

Chapter 1

Introduction

A *computer system* is a collection of hardware and software components that work together to run computer programs. Specific implementations of systems change over time, but the underlying concepts do not. All systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to improve at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

In their classic text on the C programming language [37], Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.1.

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

code/intro/hello.c

Figure 1.1: **The `hello` program.**

Although `hello` is a very simple program, every major part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why, when you run `hello` on your system.

We will begin our study of systems by tracing the lifetime of the `hello` program, from the time it is created by a programmer, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

1.1 Information is Bits in Context

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most modern systems represent text characters using the ASCII standard that represents each character with a unique byte-sized integer value. For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

```
 #   i    n     c   l    u    d    e   <sp>   <   s    t    d    i    o     .
 35  105  110   99  108  117  100  101   32   60  115  116  100  105  111   46

h     >    \n   \n  i    n    t   <sp>  m     a   i    n     (    )    \n  {
104   62   10   10  105  110  116   32  109   97  105  110   40   41   10  123

 \n <sp> <sp> <sp> <sp>  p    r    i    n    t    f     (    "   h    e    l
 10   32   32   32   32  112  114  105  110  116  102   40   34  104  101  108

l    o     ,  <sp>  w    o    r    l    d     \   n     "    )    ;    \n  }
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10  125
```

Figure 1.2: **The ASCII text representation of `hello.c`.**

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '#'. The second byte has the integer value 105, which corresponds to the character '`i`', and so on. Notice that each text line is terminated by the invisible *newline* character '`\n`', which is represented by the integer value 10. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating point number, character string, or machine instruction. This idea is explored in detail in Chapter 2.

Aside: The C programming language.
C was developed in 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989. The standard defines the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as "K&R" [37].

In Ritchie's words [60], C is "quirky, flawed, and an enormous success." So why the success?

- *C was closely tied with the Unix operating system.* C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel, and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were

exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.

- *C is a small, simple language.* The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
- *C was designed for a practical purpose.* C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed based of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes and objects. Newer languages such as C++ and Java address these issues for application-level programs. **End Aside.**

1.2 Programs are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understand by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program*, and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

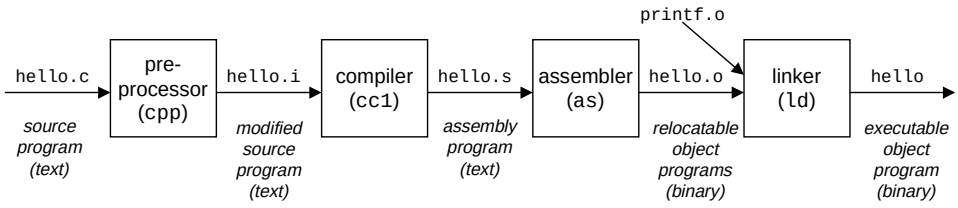


Figure 1.3: **The compilation system.**

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the # character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.

- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

Aside: The GNU project.
GCC is one of many useful tools developed by the GNU (GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. As of 2002, the GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and many others.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open source movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software*. Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel. **End Aside.**

1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a `switch` statement always more efficient than a sequence of `if-then-else` statements? Just how expensive is a function call? Is a `while` loop more efficient than a `do` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times?

In Chapter 3, we will introduce the Intel IA32 machine language and describe how compilers translate different C constructs into that language. In Chapter 5 we will learn how to tune the performance of our C programs by making simple transformations to the C code that help the compiler do its job. And in Chapter 6 we will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how our C programs can exploit this knowledge to run more efficiently.

- *Understanding link-time errors.* In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if we define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run-time? We will learn the answers to these kinds of questions in Chapter 7

- *Avoiding security holes.* For many years now, *buffer overflow bugs* have accounted for the majority of security holes in network and Internet servers. These bugs exist because too many programmers are ignorant of the stack discipline that compilers use to generate code for functions. We will describe the stack discipline and buffer overflow bugs in Chapter 3 as part of our study of assembly language.

1.4 Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable on a Unix system, we type its name to an application program known as a *shell*:

```
unix> ./hello
hello, world
unix>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

1.4.1 Hardware Organization of a System

At a high level, here is what happened in the system after you typed `hello` to the shell. Figure 1.4 shows the hardware organization of a typical system. This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel.

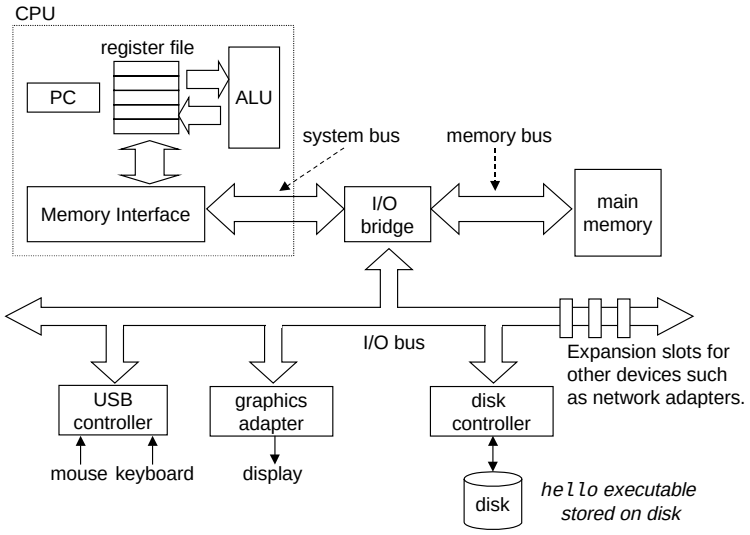


Figure 1.4: **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-sized chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. For example, Intel Pentium systems have a word size of 4 bytes, while server-class systems such as Intel Itaniums and Sun SPARCS have word sizes of 8 bytes. Smaller systems that are used as embedded controllers in automobiles and factories can have word sizes of 1 or 2 bytes. For simplicity, we will assume a word size of 4 bytes, and we will assume that buses transfer only one word at a time.

I/O devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. And in Chapter 12, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially

interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Main memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *Dynamic Random Access Memory (DRAM)* chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an Intel machine running Linux, data of type `short` requires two bytes, types `int`, `float`, and `long` four bytes, and type `double` eight bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory. [1]

From the time that power is applied to the system, until the time that the power is shut off, the processor blindly and repeatedly performs the same basic task, over and over and over: It reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple *operation* dictated by the instruction, and then updates the PC to point to the *next* instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-sized registers, each with its own unique name. The ALU computes new data and address values. Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

- *Load:* Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
- *Store:* Copy the a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
- *Update:* Copy the contents of two registers to the ALU, which adds the two words together and stores the result in a register, overwriting the previous contents of that register.
- *I/O Read:* Copy a byte or a word from an I/O device into a register.

[1] PC is also a commonly-used acronym for "Personal Computer". However, the distinction between the two is always clear from the context.

- *I/O Write:* Copy a byte or a word from a register to an I/O device.
- *Jump:* Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

Chapter 4 has much more to say about how processors work.

1.4.2 Running the `hello` Program

Given this simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters `hello` at the keyboard, the shell program reads each one into a register, and then stores it in memory, as shown in Figure 1.5.

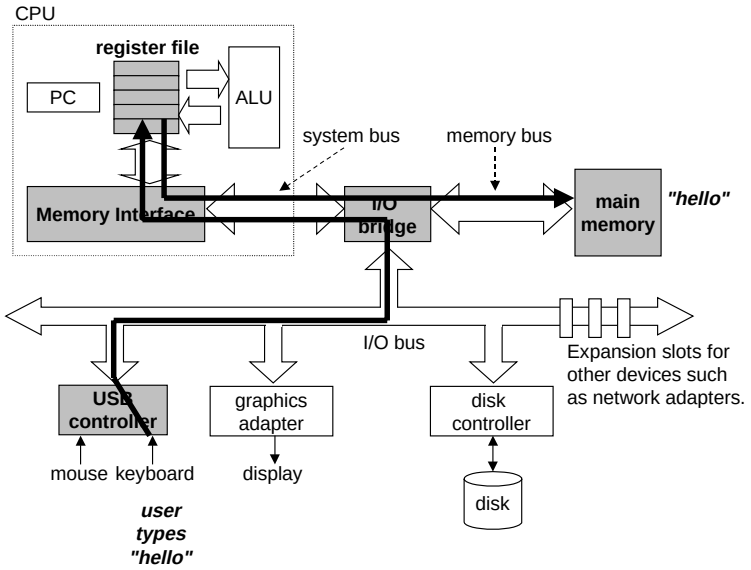


Figure 1.5: **Reading the `hello` command from the keyboard.**

When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the `hello` object file from disk to main memory. The data include the string of characters "`hello, world\n`" that will eventually be printed out.

Using a technique known as *direct memory access* (DMA) (discussed in Chapter 6), the data travels directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's `main` routine. These instruction copy the bytes

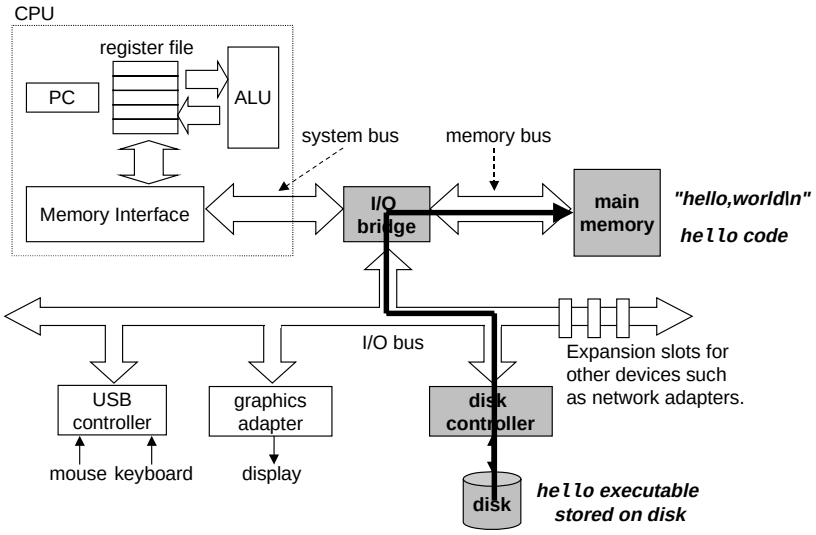


Figure 1.6: **Loading the executable from disk into main memory.**

in the "`hello, world\n`" string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

1.5 Caches Matter

An important lesson from this simple example is that a system spends a lot time moving information from one place to another. The machine instructions in the `hello` program are originally stored on disk. When the program is loaded, they are copied to main memory. When the processor runs the programs, they are copied from main memory into the processor. Similarly, the data string "`hello,world\n`", originally on disk, is copied to main memory, and then copied from main memory to the display device. From a programmer's perspective, much of this copying is overhead that slows down the "real work" of the program. Thus, a major goal for system designers is make these copy operations run as fast as possible.

Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower counterparts. For example, the disk drive on a typical system might be 100 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred of bytes of information, as opposed to millions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this *processor-memory gap* continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

To deal with the processor-memory gap, system designers include smaller faster storage devices called *caches* that serve as temporary staging areas for information that the processor is likely to need in the near

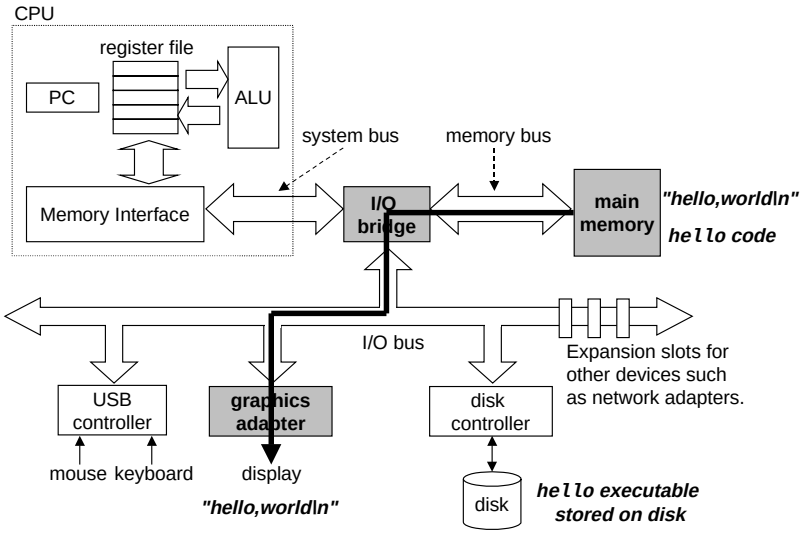


Figure 1.7: **Writing the output string from memory to the display.**

future. Figure 1.8 shows the caches in a typical system. An *L1 cache* on the processor chip holds tens of

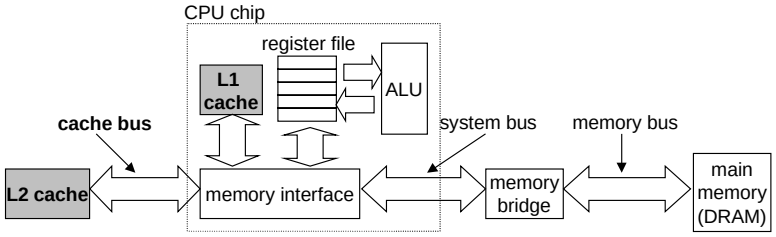


Figure 1.8: **Caches.**

thousands of bytes and can be accessed nearly as fast as the register file. A larger *L2 cache* with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as *Static Random Access Memory* (SRAM).

One of the most important lessons in this book is that application programmers who are aware of caches can exploit them to improve the performance of their programs by an order of magnitude. We will learn more about these important devices and how to exploit them in Chapter 6.

1.6 Storage Devices Form a Hierarchy

This notion of inserting a smaller, faster storage device (e.g. an SRAM cache) between the processor and a larger slower device (e.g., main memory) turns out to be a general idea. In fact, the storage devices in

every computer system are organized as the *memory hierarchy* shown in Figure 1.9. As we move from the

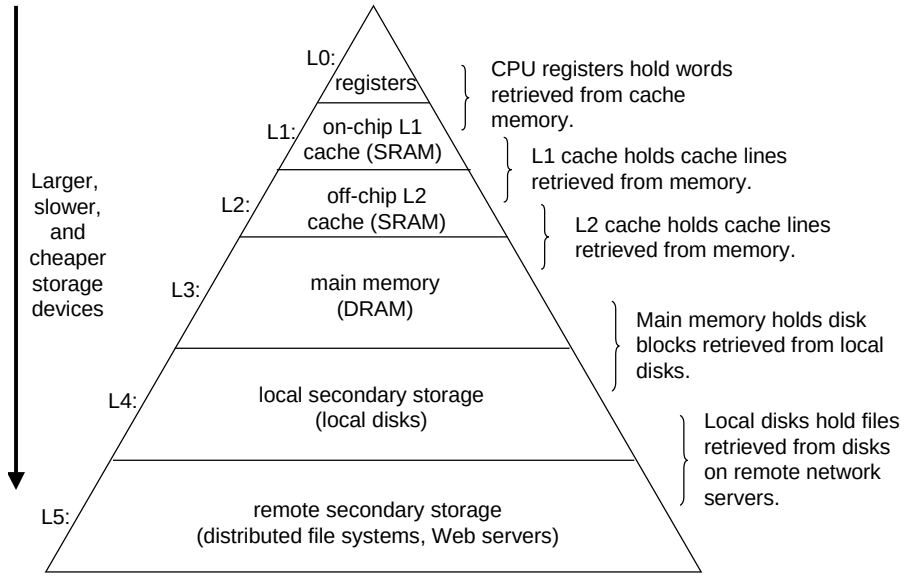


Figure 1.9: **The memory hierarchy.**

top of the hierarchy to the bottom, the devices become slower, larger, and less costly per byte. The register file occupies the top level in the hierarchy, which is known as level 0 or L0. The L1 cache occupies level 1 (hence the term L1). The L2 cache occupies level 2. Main memory occupies level 3, and so on.

The main idea of a memory hierarchy is that storage at one level serves as a cache for storage at the next lower level. Thus, the register file is a cache for the L1 cache, which is a cache for the L2 cache, which is a cache for the main memory, which is a cache for the disk. On some networked system with distributed file systems, the local disk serves as a cache for data stored on the disks of other systems.

Just as programmers can exploit knowledge of the L1 and L2 caches to improve performance, programmers can exploit their understanding of the entire memory hierarchy. Chapter 6 will have much more to say about this.

1.7 The Operating System Manages the Hardware

Back to our `hello` example. When the shell loaded and ran the `hello` program, and when the `hello` program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the *operating system*. We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10. All attempts by an application program to manipulate the hardware must go through the operating system.

The operating system has two primary purposes: (1) To protect the hardware from misuse by runaway applications, and (2) To provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices. The operating system achieves both goals

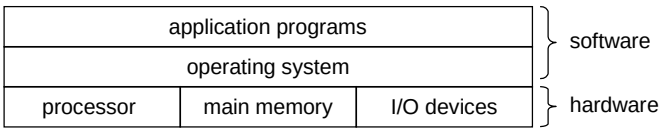


Figure 1.10: **Layered view of a computer system.**

via the fundamental abstractions shown in Figure 1.11: *processes*, *virtual memory*, and *files*. As this figure

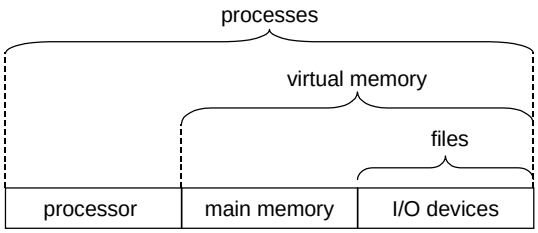


Figure 1.11: **Abstractions provided by an operating system.**

suggests, files are abstractions for I/O devices. Virtual memory is an abstraction for both the main memory and disk I/O devices. And processes are abstractions for the processor, main memory, and I/O devices. We will discuss each in turn.

Aside: Unix and Posix.
The 1960s was an era of huge, complex operating systems, such as IBM's OS/360 and Honeywell's Multics systems. While OS/360 was one of the most successful software projects in history, Multics dragged on for years and never achieved wide-scale use. Bell Laboratories was an original partner in the Multics project, but dropped out in 1969 because of concern over the complexity of the project and the lack of progress. In reaction to their unpleasant Multics experience, a group of Bell Labs researchers — Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna — began work in 1969 on a simpler operating system for a DEC PDP-7 computer, written entirely in machine language. Many of the ideas in the new system, such as the hierarchical file system and the notion of a shell as a user-level process, were borrowed from Multics, but implemented in a smaller, simpler package. In 1970, Brian Kernighan dubbed the new system "Unix" as a pun on the complexity of "Multics." The kernel was rewritten in C in 1973, and Unix was announced to the outside world in 1974 [61].

Because Bell Labs made the source code available to schools with generous terms, Unix developed a large following at universities. The most influential work was done at the University of California at Berkeley in the late 1970s and early 1980s, with Berkeley researchers adding virtual memory and the Internet protocols in a series of releases called Unix 4.xBSD (Berkeley Software Distribution). Concurrently, Bell Labs was releasing their own versions, which become known as System V Unix. Versions from other vendors, such as the Sun Microsystems Solaris system, were derived from these original BSD and System V versions.

Trouble arose in the mid 1980s as Unix vendors tried to differentiate themselves by adding new and often incompatible features. To combat this trend, IEEE (Institute for Electrical and Electronics Engineers) sponsored an effort to standardize Unix, later dubbed "Posix" by Richard Stallman. The result was a family of standards, known as the Posix standards, that cover such issues as the C language interface for Unix system calls, shell programs and utilities, threads, and network programming. As more systems comply more fully with the Posix standards, the differences between Unix version are gradually disappearing. **End Aside.**

1.7.1 Processes

When a program such as `hello` runs on a modern system, the operating system provides the illusion that the program is the only one running on the system. The program appears to have exclusive use of both the processor, main memory, and I/O devices. The processor appears to execute the instructions in the program, one after the other, without interruption. And the code and data of the program appear to be the only objects in the system's memory. These illusions are provided by the notion of a process, one of the most important and successful ideas in computer science.

A *process* is the operating system's abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware. By *concurrently*, we mean that the instructions of one process are interleaved with the instructions of another process. The operating system performs this interleaving with a mechanism known as *context switching*.

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the *context*, includes information such as the current values of the PC, the register file, and the contents of main memory. At any point in time, exactly one process is running on the system. When the operating system decides to transfer control from the current process to a some new process, it performs a *context switch* by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example `hello` scenario.

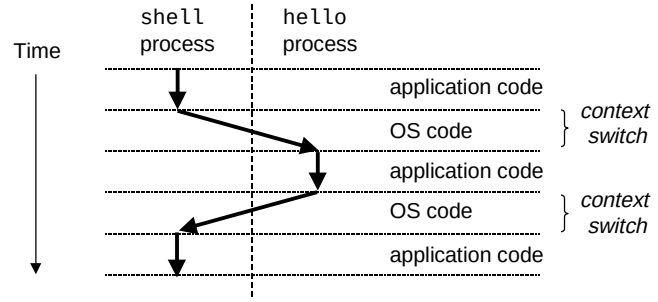


Figure 1.12: **Process context switching.**

There are two concurrent processes in our example scenario: the shell process and the `hello` process. Initially, the shell process is running alone, waiting for input on the command line. When we ask it to run the `hello` program, the shell carries out our request by invoking a special function known as a *system call* that pass control to the operating system. The operating system saves the shell's context, creates a new `hello` process and its context, and then passes control to the new `hello` process. After `hello` terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command line input.

Implementing the process abstraction requires close cooperation between both the low-level hardware and the operating system software. We will explore how this works, and how applications can create and control their own processes, in Chapter 8.

One of the implications of the process abstraction is that by interleaving different processes, it distorts

the notion of time, making it difficult for programmers to obtain accurate and repeatable measurements of running time. Chapter 9 discusses the various notions of time in a modern system and describes techniques for obtaining accurate measurements.

1.7.2 Threads

Although we normally think of a process as having a single control flow, in modern system a process can actually consist of multiple execution units, called *threads*, each running in the context of the process and sharing the same code and global data.

Threads are an increasingly important programming model because of the requirement for concurrency in network servers, because it is easier to share data between multiple threads than between multiple processes, and because threads are typically more efficient than processes. We will learn the basic concepts of threaded programs in Chapter 11, and we will learn how to build concurrent network servers with threads in Chapter 12.

1.7.3 Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its *virtual address space*. The virtual address space for Linux processes is shown in Figure 1.13 (Other Unix systems use a similar layout). In Linux, the topmost 1/4 of the address space is reserved for code and data in the operating system that is common to all processes. The bottommost 3/4 of the address space holds the code and data defined by the user's process. Note that addresses in the figure increase from bottom to the top.

The virtual address space seen by each process consists of a number of well-defined areas, each with a specific purpose. We will learn more about these areas later in the book, but it will be helpful to look briefly at each, starting with the lowest addresses and working our way up:

- *Program code and data.* Code begins at the same fixed address, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file, in our case the `hello` executable. We will learn more about this part of the address space when we study linking and loading in Chapter 7.

- *Heap.* The code and data areas are followed immediately by the run-time *heap*. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at runtime as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in Chapter 10.

- *Shared libraries.* Near the middle of the address space is an area that holds the code and data for *shared libraries* such as the C standard library and the math library. The notion of a shared library is a powerful, but somewhat difficult concept. We will learn how they work when we study dynamic linking in Chapter 7.

- *Stack.* At the top of the user's virtual address space is the *user stack* that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the

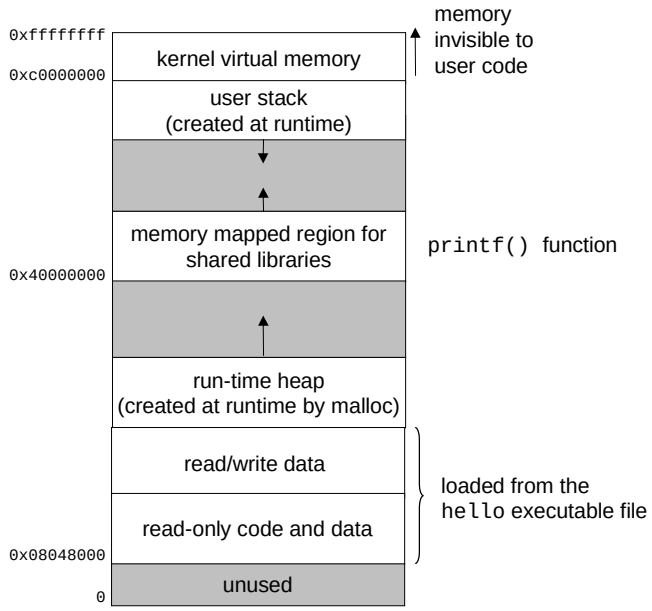


Figure 1.13: **Linux process virtual address space.**

execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. We will learn how the compiler uses the stack in Chapter 3.

- Kernel virtual memory. The *kernel* is the part of the operating system that is always resident in memory. The top 1/4 of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code.

For virtual memory to work, a sophisticated interaction is required between the hardware and the operating system software, including a hardware translation of every address generated by the processor. The basic idea is to store the contents of a process's virtual memory on disk, and then use the main memory as a cache for the disk. Chapter 10 explains how this works and why it is so important to the operation of modern systems.

1.7.4 Files

A Unix file is a sequence of bytes, nothing more and nothing less. Every I/O device, including disks, keyboards, displays, and even networks, is modeled as a file. All input and output in the system is performed by reading and writing files, using a set of operating system functions known as *system calls*.

This simple and elegant notion of a file is nonetheless very powerful because it provides applications with a uniform view of all of the varied I/O devices that might be contained in the system. For example, application programmers who manipulate the contents of a disk file are blissfully unaware of the specific disk technology. Further, the same program will run on different systems that use different disk technologies.

Aside: The Linux project.
In August, 1991, a Finnish graduate student named Linus Torvalds made a modest posting announcing a new Unix-like operating system kernel:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since April, and is starting to get ready. I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want. Any suggestions
are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
```

The rest, as they say, is history. Linux has evolved into a technical and cultural phenomenon. By combining forces with the GNU project, the Linux project has developed a complete, Posix-compliant version of the Unix operating system, including the kernel and all of the supporting infrastructure. Linux is available on a wide array of computers, from hand-held devices to mainframe computers. And it has renewed interest in the idea of open source software pioneered by the GNU project in the 1980s. We believe that a number of factors have contributed to the popularity of GNU/Linux systems:

- *Linux is relatively small.* With about one million (10^6) lines of source code, the Linux kernel is significantly smaller than comparable commercial operating systems. We recently saw a version of Linux running on a wristwatch!
- *Linux is robust.* The code development model for Linux is unique, and has resulted in a surprisingly robust system. The model consists of (1) a large set of programmers distributed around the world who update their local copies of the kernel source code, and (2) a system integrator (Linus) who decides which of these updates will become part of the official release. The model works because quality control is maintained by a talented programmer who understands everything about the system. It also results in quicker bug fixes because the pool of distributed programmers is so large.
- *Linux is portable.* Since Linux and the GNU tools are written in C, Linux can be ported to new systems without extensive code modifications.
- *Linux is open-source.* Linux is *open source*, which means that it can be down-loaded, modified, repackaged, and redistributed without restriction, gratis or for a fee, as long as the new sources are included with the distribution. This is different from other Unix versions, which are encumbered with software licenses that restrict software redistributions that might add value and make the system easier to use and install.

End Aside.

1.8 Systems Communicate With Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks. From the point of

view of an individual system, the network can be viewed as just another I/O device, as shown in Figure 1.14. When the system copies a sequence of bytes from main memory to the network adapter, the data flows across

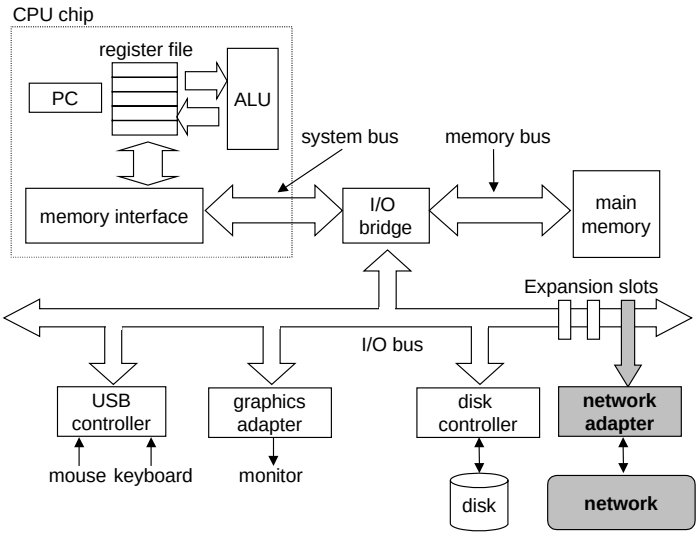


Figure 1.14: **A network is another I/O device.**

the network to another machine, instead of say, to a local disk drive. Similarly, the system can read data sent from other machines and copy this data to its main memory.

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems. For example, applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

Returning to our `hello` example, we could use the familiar telnet application to run `hello` on a remote machine. Suppose we use a telnet *client* running on our local machine to connect to a telnet *server* on a remote machine. After we log in to the remote machine and run a shell, the remote shell is waiting to receive an input command. From this point, running the `hello` program remotely involves the five basic steps shown in Figure 1.15.

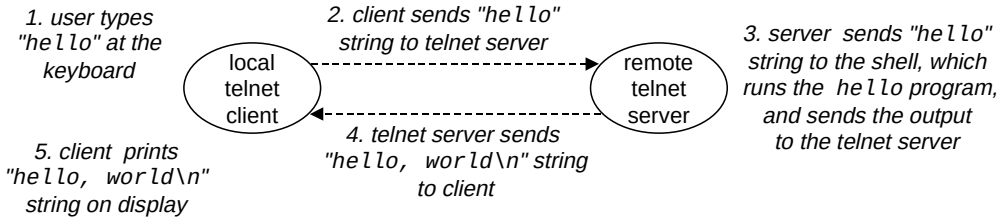


Figure 1.15: **Using `telnet` to run `hello` remotely over a network.**

After we type the "`hello`" string to the telnet client and hit the `enter` key, the client sends the string to

the telnet server. After the telnet server receives the string from the network, it passes it along to the remote shell program. Next, the remote shell runs the `hello` program, and passes the output line back to the telnet server. Finally, the telnet server forwards the output string across the network to the telnet client, which prints the output string on our local terminal.

This type of exchange between clients and servers is typical of all network applications. In Chapter 12 we will learn how to build network applications, and apply this knowledge to build a simple Web server.

1.9 Summary

This concludes our initial whirlwind tour of systems. An important idea to take away from this discussion is that a system is more than just hardware. It is a collection of intertwined hardware and software components that must work cooperate in order to achieve the ultimate goal of running application programs. The rest of this book will expand on this theme.

Bibliographic Notes

Ritchie has written interesting first-hand accounts of the early days of C and Unix [59, 60]. Ritchie and Thompson presented the first published account of Unix [61]. Silberschatz and Gavin [66] provide a comprehensive history of the different flavors of Unix. The GNU (`www.gnu.org`) and Linux (`www.linux.org`) Web pages have loads of current and historical information. Unfortunately, the Posix standards are not available online. They must be ordered for a fee from IEEE (`standards.ieee.org`).

Part I

Program Structure and Execution

Chapter 2

Representing and Manipulating Information

Modern computers store and process information represented as two-valued signals. These lowly binary digits, or *bits*, form the basis of the digital revolution. The familiar decimal, or base-10, representation has been in use for over 1000 years, having been developed in India, improved by Arab mathematicians in the 12th century, and brought to the West in the 13th century by the Italian mathematician Leonardo Pisano, better known as Fibonacci. Using decimal notation is natural for ten-fingered humans, but binary values work better when building machines that store and process information. Two-valued signals can readily be represented, stored, and transmitted, for example, as the presence or absence of a hole in a punched card, as a high or low voltage on a wire, or as a magnetic domain oriented clockwise or counterclockwise. The electronic circuitry for storing and performing computations on two-valued signals is very simple and reliable, enabling manufacturers to integrate millions of such circuits on a single silicon chip.

In isolation, a single bit is not very useful. When we group bits together and apply some *interpretation* that gives meaning to the different possible bit patterns, however, we can represent the elements of any finite set. For example, using a binary number system, we can use groups of bits to encode nonnegative numbers. By using a standard character code, we can encode the letters and symbols in a document. We cover both of these encodings in this chapter, as well as encodings to represent negative numbers and to approximate real numbers.

We consider the three most important encodings of numbers. *Unsigned* encodings are based on traditional binary notation, representing numbers greater than or equal to 0. *Two's complement* encodings are the most common way to represent signed integers, that is, numbers that may be either positive or negative. *Floating-point* encodings are a base-two version of scientific notation for representing real numbers. Computers implement arithmetic operations such as addition and multiplication, with these different representations similar to the corresponding operations on integers and real numbers.

Computer representations use a limited number of bits to encode a number, and hence some operations can *overflow* when the results are too large to be represented. This can lead to some surprising results. For example, on most of today's computers, computing the expression

```
200 * 300 * 400 * 500
```

yields −884,901,888. This runs counter to the properties of integer arithmetic—computing the product of a set of positive numbers has yielded a negative result.

On the other hand, integer computer arithmetic satisfies many of the familiar properties of true integer arithmetic. For example, multiplication is associative and commutative, so that computing all of the following C expressions yields −884,901,888:

```
(500  *  400) * (300 * 200)
((500 *  400) * 300) * 200
((200 *  500) * 300) * 400
400   * (200 * (300 * 500))
```

The computer might not generate the expected result, but at least it is consistent!

Floating point arithmetic has altogether different mathematical properties. The product of a set of positive numbers will always be positive, although overflow will yield the special value $+\infty$. On the other hand, floating point arithmetic is not associative due to the finite precision of the representation. For example, the C expression `(3.14+1e20)-1e20` will evaluate to `0.0` on most machines, while `3.14+(1e20-1e20)` will evaluate to `3.14`.

By studying the actual number representations, we can understand the ranges of values that can be represented and the properties of the different arithmetic operations. This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler. Our treatment of this material is very mathematical. We start with the basic definitions of the encodings and then derive such properties as the range of representable numbers, their bit-level representations, and the properties of the arithmetic operations. We believe it is important to examine this material from such an abstract viewpoint, because programmers need to have a solid understanding of how computer arithmetic relates to the more familiar integer and real arithmetic. Although it may appear intimidating, the mathematical treatment requires just an understanding of basic algebra. We recommend working the practice problems as a way to solidify the connection between the formal treatment and some real-life examples.

We derive several ways to perform arithmetic operations by directly manipulating the bit-level representations of numbers. Understanding these techniques will be important for understanding the machine-level code generated when compiling arithmetic expressions.

The C++ programming language is built upon C, using the exact same numeric representations and operations. Everything said in this chapter about C also holds for C++. The Java language definition, on the other hand, created a new set of standards for numeric representations and operations. Whereas the C standard is designed to allow a wide range of implementations, the Java standard is quite specific on the formats and encodings of data. We highlight the representations and operations supported by Java at several places in the chapter.

2.1 Information Storage

Rather than accessing individual bits in a memory, most computers use blocks of eight bits, or *bytes* as the smallest addressable unit of memory. A machine-level program views memory as a very large array of

Hex digit	0	1	2	3	4	5	6	7
Decimal Value	0	1	2	3	4	5	6	7
Binary Value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal Value	8	9	10	11	12	13	14	15
Binary Value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.1: **Hexadecimal Notation** Each Hex digit encodes one of 16 values.

bytes, referred to as *virtual memory*. Every byte of memory is identified by a unique number, known as its *address*, and the set of all possible addresses is known as the *virtual address space*. As indicated by its name, this virtual address space is just a conceptual image presented to the machine-level program. The actual implementation (presented in Chapter 10) uses a combination of random-access memory (RAM), disk storage, special hardware, and operating system software to provide the program with what appears to be a monolithic byte array.

One task of a compiler and the run-time system is to subdivide this memory space into more manageable units to store the different *program objects*, that is, program data, instructions, and control information. Various mechanisms are used to allocate and manage the storage for different parts of the program. This management is all performed within the virtual address space. For example, the value of a pointer in C—whether it points to an integer, a structure, or some other program unit—is the virtual address of the first byte of some block of storage. The C compiler also associates *type* information with each pointer, so that it can generate different machine-level code to access the value stored at the location designated by the pointer depending on the type of that value. Although the C compiler maintains this type information, the actual machine-level program it generates has no information about data types. It simply treats each program object as a block of bytes, and the program itself as a sequence of bytes.

New to C?
Pointers are a central feature of C. They provide the mechanism for referencing elements of data structures, including arrays. Just like a variable, a pointer has two aspects: its *value* and its *type*. The value indicates the location of some object, while its type indicates what kind (e.g., integer or floating-point number) of object is stored at that location. **End**

2.1.1 Hexadecimal Notation

A single byte consists of eight bits. In binary notation, its value ranges from 00000000_2 to 11111111_2. When viewed as a decimal integer, its value ranges from 0_{10} to 255_{10}. Neither notation is very convenient for describing bit patterns. Binary notation is too verbose, while with decimal notation, it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or *hexadecimal* numbers. Hexadecimal (or simply "Hex") uses digits '0' through '9', along with characters 'A' through 'F' to represent 16 possible values. Figure 2.1 shows the decimal and binary values associated with the 16 hexadecimal digits. Written in hexadecimal, the value of a single byte can range from 00_{16} to FF_{16}.

In C, numeric constants starting with `0x` or `0X` are interpreted as being in hexadecimal. The characters

'A' through 'F' may be written in either upper or lower case. For example, we could write the number $FA1D37B_{16}$ as `0xFA1D37B`, as `0xfa1d37b`, or even mixing upper and lower case, e.g., `0xFa1D37b`. We will use the C notation for representing hexadecimal values in this book.

A common task in working with machine-level programs is to manually convert between decimal, binary, and hexadecimal representations of bit patterns. A starting point is to be able to convert, in both directions, between a single hexadecimal digit and a four-bit binary pattern. This can always be done by referring to a chart such as that shown in Figure 2.1. When doing the conversion manually, one simple trick is to memorize the decimal equivalents of hex digits `A`, `C`, and `F`. The hex values `B`, `D`, and `E` can be translated to decimal by computing their values relative to the first three.

Practice Problem 2.1:

Fill in the missing entries in the following figure, giving the decimal, binary, and hexadecimal values of different byte patterns.

Decimal	Binary	Hexadecimal
0	00000000	`00`
55		
136		
243		
	01010010	
	10101100	
	11100111	
		`A7`
		`3E`
		`BC`

Aside: Converting between decimal and hexadecimal.
For converting larger values between decimal and hexadecimal, it is best to let a computer or calculator do the work. For example, the following script in the Perl language converts a list of numbers from decimal to hexadecimal:

bin/d2h

```
1 #!/usr/local/bin/perl
2 # Convert list of decimal numbers into hex
3 for ($i = 0; $i < @ARGV; $i++) {
4     printf("%d = 0x%x\n", $ARGV[$i], $ARGV[$i]);
5 }
```

bin/d2h

Once this file has been set to be executable, the command:

```
unix> ./d2h 100 500 751
```

will yield output:

```
100 = 0x64
500 = 0x1f4
751 = 0x2ef
```

Similarly, the following script converts from hexadecimal to decimal:

bin/h2d

```
1 #!/usr/local/bin/perl
2 # Convert list of decimal numbers into hex
3 for ($i = 0; $i < @ARGV; $i++) {
4     $val = hex($ARGV[$i]);
5     printf("0x%x = %d\n", $val, $val);
6 }
```

bin/h2d

End Aside.

2.1.2 Words

Every computer has a *word size*, indicating the nominal size of integer and pointer data. Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is the maximum size of the virtual address space. That is, for a machine with an n-bit word size, the virtual addresses can range from 0 to $2^n - 1$, giving the program access to at most 2^n bytes.

Most computers today have a 32-bit word size. This limits the virtual address space to 4 gigabytes (written 4 GB), that is, just over 4×10^9 bytes. Although this is ample space for most applications, we have reached the point where many large-scale scientific and database applications require larger amounts of storage. Consequently, high-end machines with 64-bit word sizes are becoming increasingly commonplace as storage costs decrease.

2.1.3 Data Sizes

Computers and compilers support multiple data formats using different ways to encode data, such as integers and floating point, as well as different lengths. For example, many machines have instructions for manipulating single bytes, as well as integers represented as two, four, and eight-byte quantities. They also support floating-point numbers represented as four and eight-byte quantities.

The C language supports multiple data formats for both integer and floating-point data. The C data type `char` represents a single byte. Although the name "char" derives from the fact that it is used to store a single character in a text string, it can also be used to store integer values. The C data type `int` can also be prefixed by the qualifiers `long` and `short`, providing integer representations of various sizes. Figure 2.2 shows the number of bytes allocated for various C data types. The exact number depends on both the machine and the compiler. We show two representative cases: a typical 32-bit machine, and the Compaq Alpha architecture, a 64-bit machine targeting high end applications. Most 32-bit machines use the allocations indicated as "typical." Observe that "short" integers have two-byte allocations, while an unqualified `int` is 4 bytes. A "long" integer uses the full word size of the machine.

C Declaration	Typical 32-bit	Compaq Alpha
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`char *`	4	8
`float`	4	4
`double`	8	8

Figure 2.2: **Sizes (in Bytes) of C Numeric Data Types.** The number of bytes allocated varies with machine and compiler.

Figure 2.2 also shows that a pointer (e.g., a variable declared as being of type "`char *`") uses the full word size of the machine. Most machines also support two different floating-point formats: single precision, declared in C as `float`, and double precision, declared in C as `double`. These formats use four and eight bytes, respectively.

New to C?
For any data type T, the declaration

```
T *p;
```

indicates that `p` is a pointer variable, pointing to an object of type T. For example

```
char *p;
```

is the declaration of a pointer to an object of type `char`. **End**

Programmers should strive to make their programs portable across different machines and compilers. One aspect of portability is to make the program insensitive to the exact sizes of the different data types. The C standard sets lower bounds on the numeric ranges of the different data types, as will be covered later, but there are no upper bounds. Since 32-bit machines have been the standard for the last 20 years, many programs have been written assuming the allocations listed as "typical 32-bit" in Figure 2.2. Given the increasing prominence of 64-bit machines in the near future, many hidden word size dependencies will show up as bugs in migrating these programs to new machines. For example, many programmers assume that a program object declared as type `int` can be used to store a pointer. This works fine for most 32-bit machines but leads to problems on an Alpha.

2.1.4 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what will be the address of the object, and how will we order the bytes in memory. In virtually all machines, a multibyte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the

bytes used. For example, suppose a variable `x` of type `int` has address `0x100`, that is, the value of the address expression `&x` is `0x100`. Then the four bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.

For ordering the bytes representing an object, there are two common conventions. Consider a w-bit integer having a bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit, and x_0 is the least. Assuming w is a multiple of eight, these bits can be grouped as bytes, with the most significant byte having bits $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$, the least significant byte having bits $[x_7, x_6, \ldots, x_0]$, and the other bytes having bits from the middle. Some machines choose to store the object in memory ordered from least significant byte to most, while other machines store them from most to least. The former convention—where the least significant byte comes first—is referred to as *little endian*. This convention is followed by most machines from the former Digital Equipment Corporation (now part of Compaq Corporation), as well as by Intel. The latter convention—where the most significant byte comes first—is referred to as *big endian*. This convention is followed by most machines from IBM, Motorola, and Sun Microsystems. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, personal computers manufactured by IBM use Intel-compatible processors and hence are little endian. Many microprocessor chips, including Alpha and the PowerPC by Motorola can be run in either mode, with the byte ordering convention determined when the chip is powered up.

Continuing our earlier example, suppose the variable `x` of type `int` and at address `0x100` has a hexadecimal value of `0x01234567`. The ordering of the bytes within the address range `0x100` through `0x103` depends on the type of machine:

Big endian

	0x100	0x101	0x102	0x103	
···	01	23	45	67	···

Little endian

	0x100	0x101	0x102	0x103	
···	67	45	23	01	···

Note that in the word `0x01234567` the high-order byte has hexadecimal value `0x01`, while the low-order byte has value `0x67`.

People get surprisingly emotional about which byte ordering is the proper one. In fact, the terms "little endian" and "big endian" come from the book *Gulliver's Travels* by Jonathan Swift, where two warring factions could not agree by which end a soft-boiled egg should be opened—the little end or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, and hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

Aside: Origin of "Endian."
Here is how Jonathan Swift, writing in 1726, described the history of the controversy between big and little endians:

> ...the two great empires of Lilliput and Blefuscu. Which two mighty powers have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. It began upon the following occasion. It is allowed on all hands, that the primitive way of breaking eggs, before we eat

> them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account; wherein one emperor lost his life, and another his crown. These civil commotions were constantly fomented by the monarchs of Blefuscu; and when they were quelled, the exiles always fled for refuge to that empire. It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this controversy: but the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments.
>
> In his day, Swift was satirizing the continued conflicts between England (Lilliput) and France (Blefuscu). Danny Cohen, an early pioneer in networking protocols, first applied these terms to refer to byte ordering [16], and the terminology has been widely adopted. **End Aside.**

For most application programmers, the byte orderings used by their machines are totally invisible. Programs compiled for either class of machine give identical results. At times, however, byte ordering becomes an issue. The first is when binary data is communicated over a network between different machines. A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice-versa, leading to the bytes within the words being in reverse order for the receiving program. To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation. We will see examples of these conversions in Chapter 12.

A second case is when programs are written that circumvent the normal type system. In the C language, this can be done using a *cast* to allow an object to be referenced according to a different data type from which it was created. Such coding tricks are strongly discouraged for most application programming, but they can be quite useful and even necessary for system-level programming.

Figure 2.3 shows C code that uses casting to access and print the byte representations of different program objects. We use `typedef` to define data type `byte_pointer` as a pointer to an object of type "`unsigned char`." Such a byte pointer references a sequence of bytes where each byte is considered to be a nonnegative integer. The first routine `show_bytes` is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count. It prints the individual bytes in hexadecimal. The C formatting directive "`%.2x`" indicates that an integer should be printed in hexadecimal with at least two digits.

> **New to C?**
> The `typedef` declaration in C provides a way of giving a name to a data type. This can be a great help in improving code readability, since deeply nested type declarations can be difficult to decipher.
>
> The syntax for `typedef` is exactly like that of declaring a variable, except that it uses a type name rather than a variable name. Thus, the declaration of `byte_pointer` in Figure 2.3 has the same form as would the declaration of a variable to type "`unsigned char`."
>
> For example, the declaration:
>
> ```
> typedef int *int_pointer;
> int_pointer ip;
> ```
>
> defines type "`int_pointer`" to be a pointer to an `int`, and declares a variable `ip` of this type. Alternatively, we could declare this variable directly as:

code/data/show-bytes.c

```
1 #include <stdio.h>
2
3 typedef unsigned char *byte_pointer;
4
5 void show_bytes(byte_pointer start, int len)
6 {
7     int i;
8     for (i = 0; i < len; i++)
9         printf(" %.2x", start[i]);
10    printf("\n");
11 }
12
13 void show_int(int x)
14 {
15    show_bytes((byte_pointer) &x, sizeof(int));
16 }
17
18 void show_float(float x)
19 {
20    show_bytes((byte_pointer) &x, sizeof(float));
21 }
22
23 void show_pointer(void *x)
24 {
25    show_bytes((byte_pointer) &x, sizeof(void *));
26 }
```

code/data/show-bytes.c

Figure 2.3: **Code to Print the Byte Representation of Program Objects.** This code uses casting to circumvent the type system. Similar functions are easily defined for other data types.

```
int *ip;
```

End

New to C?
The `printf` function (along with its cousins `fprintf` and `sprintf`) provides a way to print information with considerable control over the formatting details. The first argument is a `format string`, while any remaining arguments are values to be printed. Within the formatting string, each character sequence starting with '%' indicates how to format the next argument. Typical examples include '`%d`' to print a decimal integer and '`%f`' to print a floating-point number, and '`%c`' to print a character having the character code given by the argument. **End**

New to C?
In function `show_bytes` (Figure 2.3) we see the close connection between pointers and arrays, as will be discussed in detail in Section 3.8. We see that this function has an argument `start` of type `byte_pointer` (which has been defined to be a pointer to `unsigned char`,) but we see the array reference `start[i]` on line 9. In C, we can use reference a pointer with array notation, and we can reference arrays with pointer notation. In this example, the reference `start[i]` indicates that we want to read the byte that is `i` positions beyond the location pointed to by `start`. **End**

Procedures `show_int`, `show_float`, and `show_pointer` demonstrate how to use procedure `show_bytes` to print the byte representations of C program objects of type `int`, `float`, and `void *`, respectively. Observe that they simply pass `show_bytes` a pointer `&x` to their argument `x`, casting the pointer to be of type "`unsigned char *`." This cast indicates to the compiler that the program should consider the pointer to be to a sequence of bytes rather than to an object of the original data type. This pointer will then be to the lowest byte address used by the object.

New to C?
In lines 15, 20, and 24 of Figure 2.3 we see uses of two operations that are unique to C and C++. The C "address of" operator & creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding variable `x`. The type of this pointer depends on the type of `x`, and hence these three pointers are of type `int *`, `float *`, and `void **`, respectively. (Data type `void *` is a special kind of pointer with no associated type information.)

The cast operator converts from one data type to another. Thus, the cast `(byte_pointer) &x` indicates that whatever type the pointer `&x` had before, it now is a pointer to data of type `unsigned char`. **End**

These procedures use the C operator `sizeof` to determine the number of bytes used by the object. In general, the expression `sizeof(`T`)` returns the number of bytes required to store an object of type T. Using `sizeof`, rather than a fixed value, is one step toward writing code that is portable across different machine types.

We ran the code shown in Figure 2.4 on several different machines, giving the results shown in Figure 2.5. The machines used were:

Linux: Intel Pentium II running Linux.

NT: Intel Pentium II running Windows-NT.

Sun: Sun Microsystems UltraSPARC running Solaris.

Alpha: Compaq Alpha 21164 running Tru64 Unix.

code/data/show-bytes.c

```
1 void test_show_bytes(int val)
2 {
3     int ival = val;
4     float fval = (float) ival;
5     int *pval = &ival;
6     show_int(ival);
7     show_float(fval);
8     show_pointer(pval);
9 }
```

code/data/show-bytes.c

Figure 2.4: **Byte Representation Examples.** This code prints the byte representations of sample data objects.

Machine	Value	Type	Bytes (Hex)
Linux	12,345	`int`	`39 30 00 00`
NT	12,345	`int`	`39 30 00 00`
Sun	12,345	`int`	`00 00 30 39`
Alpha	12,345	`int`	`39 30 00 00`
Linux	$12,345.0$	`float`	`00 e4 40 46`
NT	$12,345.0$	`float`	`00 e4 40 46`
Sun	$12,345.0$	`float`	`46 40 e4 00`
Alpha	$12,345.0$	`float`	`00 e4 40 46`
Linux	`&ival`	`int *`	`3c fa ff bf`
NT	`&ival`	`int *`	`1c ff 44 02`
Sun	`&ival`	`int *`	`ef ff fc e4`
Alpha	`&ival`	`int *`	`80 fc ff 1f 01 00 00 00`

Figure 2.5: **Byte Representations of Different Data Values.** Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine-dependent.

Our sample integer argument 12,345 has hexadecimal representation `0x00003039`. For the `int` data, we get identical results for all machines, except for the byte ordering. In particular, we can see that the least significant byte value of `0x39` is printed first for Linux, NT, and Alpha, indicating little-endian machines, and last for Sun, indicating a big-endian machine. Similarly, the bytes of the `float` data are identical, except for the byte ordering. On the other hand, the pointer values are completely different. The different machine/operating system configurations use different conventions for storage allocation. One feature to note is that the Linux and Sun machines use four-byte addresses, while the Alpha uses eight-byte addresses.

Observe that although the floating point and the integer data both encode the numeric value 12,345, they have very different byte patterns: `0x00003039` for the integer, and `0x4640E400` for floating point. In general, these two formats use different encoding schemes. If we expand these hexadecimal patterns into binary and shift them appropriately, we find a sequence of 13 matching bits, indicated below by a sequence of asterisks:

```
    0   0   0   0   3   0   3   9
00000000000000000011000000111001
                  *************
             4   6   4   0   E   4   0   0
01000110010000001110010000000000
```

This is not coincidental. We will return to this example when we study floating-point formats.

Practice Problem 2.2:

Consider the following 3 calls to `show_bytes`:

```
int val = 0x12345678;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate below the values that would be printed by each call on a little-endian machine and on a big-endian machine.

A. Little endian: Big endian:
B. Little endian: Big endian:
C. Little endian: Big endian:

Practice Problem 2.3:

Using `show_int` and `show_float`, we determine that the integer 3490593 has hexadecimal representation `0x00354321`, while the floating-point number 3490593.0 has hexadecimal representation representation `0x4A550C84`.

A. Write the binary representations of these two hexadecimal values.
B. Shift these two strings relative to one another to maximize the number of matching bits.
C. How many bits match? What parts of the strings do not match?

2.1.5 Representing Strings

A string in C is encoded by an array of characters terminated by the null (having value 0) character. Each character is represented by some standard encoding, with the most common being the ASCII character code. Thus, if we run our routine `show_bytes` with arguments `"12345"` and `6` (to include the terminating character), we get the result `31 32 33 34 35 00`. Observe that the ASCII code for decimal digit x happens to be `0x3`x, and that the terminating byte has the hex representation `0x00`. This same result would be obtained on any system using ASCII as its character code, independent of the byte ordering and word size conventions. As a consequence, text data is more platform-independent than binary data.

Aside: Generating an ASCII table.
You can display a table showing the ASCII character code by executing the command `man ascii`. **End Aside.**

Practice Problem 2.4:

What would be printed as a result of the following call to `show_bytes`:

```
char *s = "ABCDEF";
show_bytes(s, strlen(s));
```

Note that letters '`A`' through '`Z`' have ASCII codes `0x41` through `0x5A`.

Aside: The Unicode character set.
The ASCII character set is suitable for encoding English language documents, but it does not have much in the way of special characters, such as the French 'ç.' It is wholly unsuited for encoding documents in languages such as Greek, Russian, and Chinese. Recently, the 16-bit *Unicode* character set has been adopted to support documents in all languages. This doubling of the character set representation enables a very large number of different characters to be represented. The Java programming language uses Unicode when representing character strings. Program libraries are also available for C that provide Unicode versions of the standard string functions such as `strlen` and `strcpy`. **End Aside.**

2.1.6 Representing Code

Consider the following C function:

```
1 int sum(int x, int y)
2 {
3     return x + y;
4 }
```

When compiled on our sample machines, we generate machine code having the following byte representations:

Linux: `55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`

NT: `55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Figure 2.6: **Operations of Boolean Algebra.** Binary values 1 and 0 encode logic values TRUE and FALSE, while operations ~, &, |, and ^ encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

Sun: `81 C3 E0 08 90 02 00 09`

Alpha: `00 00 30 42 01 80 FA 6B`

Here we find that the instruction codings are different, except for the NT and Linux machines. Different machine types use different and incompatible instructions and encodings. The NT and Linux machines both have Intel processors and hence support the same machine-level instructions. In general, however, the structure of an executable NT program differs from a Linux program, and hence the machines are not fully binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that a program, from the perspective of the machine, is simply sequences of bytes. The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging. We will see this more clearly when we study machine-level programming in Chapter 3.

2.1.7 Boolean Algebras and Rings

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole around 1850, and hence goes under the heading of *Boolean algebra*. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the properties of propositional logic.

There is an infinite number of different Boolean algebras, where the simplest is defined over the two-element set $\{0, 1\}$. Figure 2.6 defines several operations in this Boolean algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations, as will be discussed later. The Boolean operation ~ corresponds to the logical operation NOT, denoted in propositional logic as $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice-versa. Correspondingly, $\sim p$ equals 1 when p equals 0, and vice-versa. Boolean operation & corresponds to the logical operation AND, denoted in propositional logic as $\wedge$. We say that $P \wedge Q$ holds when both P and Q are true. Correspondingly, $p \,\&\, q$ equals 1 only when $p = 1$ and $q = 1$. Boolean operation | corresponds to the logical operation OR, denoted in propositional logic as $\vee$. We say that $P \vee Q$ holds when either P or Q are true. Correspondingly, $p \mid q$ equals 1 when either $p = 1$ or $q = 1$. Boolean operation ^ corresponds to the logical operation EXCLUSIVE-OR, denoted in propositional logic as $\oplus$. We say that $P \oplus Q$ holds when either P or Q are true, but not both.

Shared Properties

Property	Integer Ring	Boolean Algebra
Commutativity	$a + b = b + a$ $a \times b = b \times a$	$a \mid b = b \mid a$ $a \,\&\, b = b \,\&\, a$
Associativity	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a \mid b) \mid c = a \mid (b \mid c)$ $(a \,\&\, b) \,\&\, c = a \,\&\, (b \,\&\, c)$
Distributivity	$a \times (b + c) = (a \times b) + (a \times c)$	$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$
Identities	$a + 0 = a$ $a \times 1 = a$	$a \mid 0 = a$ $a \,\&\, 1 = a$
Annihilator	$a \times 0 = 0$	$a \,\&\, 0 = 0$
Cancellation	$-(-a) = a$	$\tilde{}(\tilde{}a) = a$

Unique to Rings

Inverse	$a + -a = 0$	—

Unique to Boolean Algebras

Distributivity	—	$a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$
Complement	— —	$a \mid \tilde{}a = 1$ $a \,\&\, \tilde{}a = 0$
Idempotency	— —	$a \,\&\, a = a$ $a \mid a = a$
Absorption	— —	$a \mid (a \,\&\, b) = a$ $a \,\&\, (a \mid b) = a$
DeMorgan's laws	— —	$\tilde{}(a \,\&\, b) = \tilde{}a \mid \tilde{}b$ $\tilde{}(a \mid b) = \tilde{}a \,\&\, \tilde{}b$

Figure 2.7: **Comparison of Integer Ring and Boolean Algebra.** The two mathematical structures share many properties, but there are key differences, particularly between $-$ and $\tilde{}$.

Correspondingly, $p \hat{} q$ equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon, who would later found the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 master's thesis, he showed that Boolean algebra could be applied to the design and analysis of networks of electromechanical relays. Although computer technology has advanced considerably since that time, Boolean algebra still plays a central role in digital systems design and analysis.

There are many parallels between integer arithmetic and Boolean algebra, as well as several important differences. In particular, the set of integers, denoted $\mathcal{Z}$, forms a mathematical structure known as a *ring*, denoted $\langle \mathcal{Z}, +, \times, -, 0, 1 \rangle$, with addition serving as the *sum* operation, multiplication as the *product* operation, negation as the additive inverse, and elements 0 and 1 serving as the additive and multiplicative identities. The Boolean algebra $\langle \{0, 1\}, \mid, \&, \tilde{}, 0, 1 \rangle$ has similar properties. Figure 2.7 highlights properties of these two structures, showing the properties that are common to both and those that are unique to one or the other. One important difference is that $\tilde{}a$ is not an inverse for a under $\mid$.

Aside: What good is abstract algebra?
Abstract algebra involves identifying and analyzing the common properties of mathematical operations in different domains. Typically, an algebra is characterized by a set of elements, some of its key operations, and some important elements. As an example, modular arithmetic also forms a ring. For modulus n, the algebra is denoted $\langle \mathcal{Z}_n, +_n, \times_n, -_n, 0, 1\rangle$, with components defined as follows:

$$\begin{aligned} \mathcal{Z}_n &= \{0, 1, \ldots, n-1\} \\ a +_n b &= a + b \bmod n \\ a \times_n b &= a \times b \bmod n \\ -_n a &= \begin{cases} 0, & a = 0 \\ n - a, & a > 0 \end{cases} \end{aligned}$$

Even though modular arithmetic yields different results from integer arithmetic, it has many of the same mathematical properties. Other well-known rings include rational and real numbers. **End Aside.**

If we replace the OR operation of Boolean algebra by the EXCLUSIVE-OR operation, and the complement operation ~ with the identity operation I—where $I(a) = a$ for all a—we have a structure $\langle \{0, 1\}, \hat{}, \&, I, 0, 1\rangle$. This structure is no longer a Boolean algebra—in fact it's a ring. It can be seen to be a particularly simple form of the ring consisting of all integers $\{0, 1, \ldots, n-1\}$ with both addition and multiplication performed modulo n. In this case, we have $n = 2$. That is, the Boolean AND and EXCLUSIVE-OR operations correspond to multiplication and addition modulo 2, respectively. One curious property of this algebra is that every element is its own additive inverse: $a \hat{} I(a) = a \hat{} a = 0$.

Aside: Who, besides mathematicians, care about Boolean rings?
Every time you enjoy the clarity of music recorded on a CD or the quality of video recorded on a DVD, you are taking advantage of Boolean rings. These technologies rely on *error-correcting codes* to reliably retrieve the bits from a disk even when dirt and scratches are present. The mathematical basis for these error-correcting codes is a linear algebra based on Boolean rings. **End Aside.**

We can extend the four Boolean operations to also operate on bit vectors, i.e., strings of 0s and 1s of some fixed length w. We define the operations over bit vectors according their applications to the matching elements of the arguments. For example, we define $[a_{w-1}, a_{w-2}, \ldots, a_0]$&$[b_{w-1}, b_{w-2}, \ldots, b_0]$ to be $[a_{w-1}$& b_{w-1}, a_{w-2} & $b_{w-2}, \ldots, a_0$ & $b_0]$, and similarly for operations ~, |, and ^. Letting $\{0, 1\}^w$ denote the set of all strings of 0s and 1s having length w, and a^w denote the string consisting of w repetitions of symbol a, then one can see that the resulting algebras: $\langle \{0, 1\}^w, |, \&, \tilde{}, 0^w, 1^w\rangle$ and $\langle \{0, 1\}^w, \hat{}, \&, I, 0^w, 1^w\rangle$ form Boolean algebras and rings, respectively. Each value of w defines a different Boolean algebra and a different Boolean ring.

Aside: Are Boolean rings the same as modular arithmetic?
The two-element Boolean ring $\langle \{0, 1\}, \hat{}, \&, I, 0, 1\rangle$ is identical to the ring of integers modulo two $\langle \mathcal{Z}_2, +_2, \times_2, -_2, 0, 1\rangle$. The generalization to bit vectors of length w, however, however, yields a very different ring from modular arithmetic. **End Aside.**

Practice Problem 2.5:

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

Operation	Result
a	$[01101001]$
b	$[01010101]$
~a	
~b	
a & b	
a | b	
a ^ b	

One useful application of bit vectors is to represent finite sets. For example, we can denote any subset $A \subseteq \{0, 1, \ldots, w-1\}$ as a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where $a_i = 1$ if and only if $i \in A$. For example, (recalling that we write a_{w-1} on the left and a_0 on the right), we have $a = [01101001]$ representing the set $A = \{0, 3, 5, 6\}$, and $b = [01010101]$ representing the set $B = \{0, 2, 4, 6\}$. Under this interpretation, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. For example, the operation a & b yields bit vector $[01000001]$, while $A \cap B = \{0, 6\}$.

In fact, for any set S, the structure $\langle \mathcal{P}(S), \cup, \cap, \overline{\ }, \emptyset, S \rangle$ forms a Boolean algebra, where $\mathcal{P}(S)$ denotes the set of all subsets of S, and $\overline{\ }$ denotes the set complement operator. That is, for any set A, its complement is the set $\overline{A} = \{a \in S | a \notin A\}$. The ability to represent and manipulate finite sets using bit vector operations is a practical outcome of a deep mathematical principle.

2.1.8 Bit-Level Operations in C

One useful feature of C is that it supports bit-wise Boolean operations. In fact, the symbols we have used for the Boolean operations are exactly those used by C: | for OR, & for AND, ~ for NOT, and ^ for EXCLUSIVE-OR. These can be applied to any "integral" data type, that is, one declared as type `char` or `int`, with or without qualifiers such as `short`, `long`, or `unsigned`. Here are some example expression evaluations:

C Expression	Binary Expression	Binary Result	C Result
`~0x41`	~$[01000001]$	$[10111110]$	`0xBE`
`~0x00`	~$[00000000]$	$[11111111]$	`0xFF`
`0x69 & 0x55`	$[01101001]$ & $[01010101]$	$[01000001]$	`0x41`
`0x69 | 0x55`	$[01101001]$ | $[01010101]$	$[01111101]$	`0x7D`

As our examples show, the best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

Practice Problem 2.6:

To show how the ring properties of ^ can be useful, consider the following program:

```
1 void inplace_swap(int *x, int *y)
2 {
3     *x = *x ^ *y;  /* Step 1 */
```

```
4     *y = *x ^ *y;  /* Step 2 */
5     *x = *x ^ *y;  /* Step 3 */
6 }
```

As the name implies, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables `x` and `y`. Note that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is no performance advantage to this way of swapping. It is merely an intellectual amusement.

Starting with values a and b in the locations pointed to by `x` and `y`, respectively, fill in the following table giving the values stored at the two locations after each step of the procedure. Use the ring properties to show that the desired effect is achieved. Recall that every element is its own additive inverse, that is, $a \hat{} a = 0$.

Step	`*x`	`*y`
Initially	a	b
Step 1		
Step 2		
Step 3		

One common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that indicates a selected set of bits within a word. As an example, the mask `0xFF` (having 1s for the least significant eight bits) indicates the low-order byte of a word. The bit-level operation `x & 0xFF` yields a value consisting of the least significant byte of `x`, but with all other bytes set to 0. For example, with `x` = `0x89ABCDEF`, the expression would yield `0x000000EF`. The expression `~0` will yield a mask of all 1s, regardless of the word size of the machine. Although the same mask can be written `0xFFFFFFFF` for a 32-bit machine, such code is not as portable.

Practice Problem 2.7:

Write C expressions for the following values, with the results for `x` = `0x98FDECBA` and a 32-bit word size shown in square brackets:

A. The least significant byte of `x`, with all other bits set to 1 [`0xFFFFFFBA`].

B. The complement of the least significant byte of `x`, with all other bytes left unchanged [`0x98FDEC45`].

C. All but the least significant byte of `x`, with the least significant byte set to 0 [`0x98FDEC00`].

Although our examples assume a 32-bit word size, your code should work for any word size $w \geq 8$.

Practice Problem 2.8:

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations AND and OR, it had instructions `bis` (bit set) and `bic` (bit clear). Both instructions take a data word `x` and a mask word `m`. They generate a result `z` consisting of the bits of `x` modified according to the bits of `m`. With `bis`, the modification involves setting `z` to 1 at each bit position where `m` is 1. With `bic`, the modification involves setting `z` to 0 at each bit position where `m` is 1.

We would like to write C functions `bis` and `bic` to compute the effect of these two instructions. Fill in the missing expressions in the code below using the bit-level operations of C.

```
/* Bit Set */
int bis(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ____________;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ____________;
  return result;
}
```

2.1.9 Logical Operations in C

C also provides a set of *logical* operators `||`, `&&`, and `!`, which correspond to the OR, AND, and NOT operations of propositional logic. These can easily be confused with the bit-level operations, but their function is quite different. The logical operations treat any nonzero argument as representing TRUE and argument 0 as representing FALSE. They return either 1 or 0 indicating a result of either TRUE or FALSE, respectively. Here are some example expression evaluations:

Expression	Result
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x41`	`0x01`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Observe that a bit-wise operation will have behavior matching that of its logical counterpart only in the special case where the arguments are restricted to be either 0 or 1.

A second important distinction between the logical operators && and `||`, versus their bit-level counterparts & and `|` is that the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument. Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Practice Problem 2.9:

Suppose that `x` and `y` have byte values `0x66` and `0x93`, respectively. Fill in the following table indicating the byte values of the different C expressions

Expression	Value	Expression	Value
`x & y`		`x && y`	
`x | y`		`x || y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

Practice Problem 2.10:

Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. That is, it will return 1 when `x` and `y` are equal and 0 otherwise.

2.1.10 Shift Operations in C

C also provides a set of *shift* operations for shifting bit patterns to the left and to the right. For an operand `x` having bit representation $[x_{n-1}, x_{n-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{n-k-1}, x_{n-k-2}, \ldots, x_0, 0, \ldots 0]$. That is, `x` is shifted k bits to the left, dropping off the k most significant bits and filling the left end with k 0s. The shift amount should be a value between 0 and $n-1$. Shift operations group from left to right, so `x << j << k` is equivalent to `(x << j) << k`. Be careful about operator precedence: `1<<5 - 1` is evaluated as `1 << (5-1)`, not as `(1<<5) - 1`.

There is a corresponding right shift operation `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift: *logical* and *arithmetic*. A logical right shift fills the left end with k 0s, giving a result $[0, \ldots, 0, x_{n-1}, x_{n-2}, \ldots x_k]$. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{n-1}, \ldots, x_{n-1}, x_{n-1}, x_{n-2}, \ldots x_k]$. This convention might seem peculiar, but as we will see it is useful for operating on signed integer data.

The C standard does not precisely define which type of right shift should be used. For unsigned data (i.e., integral objects declared with the qualifier `unsigned`), right shifts must be logical. For signed data (the default), either arithmetic or logical shifts may be used. This unfortunately means that any code assuming one form or the other will potentially encounter portability problems. In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case.

Practice Problem 2.11:

Fill in the table below showing the effects of the different shift operations on single-byte quantities. Write each answer as two hexadecimal digits.

`x`	`x << 3`	`x >> 2` (Logical)	`x >> 2` (Arithmetic)
`0xF0`			
`0x0F`			
`0xCC`			
`0x55`			

C Declaration	Guaranteed		Typical 32-bit	
	Minimum	Maximum	Minimum	Maximum
`char`	−127	127	−128	127
`unsigned char`	0	255	0	255
`short[int]`	−32,767	32,767	−32,768	32,767
`unsigned short[int]`	0	63,535	0	63,535
`int`	−32,767	32,767	−2,147,483,648	2,147,483,647
`unsigned[int]`	0	65,535	0	4,294,967,295
`long[int]`	−2,147,483,647	2,147,483,647	−2,147,483,648	2,147,483,647
`unsigned long[int]`	0	4,294,967,295	0	4,294,967,295

Figure 2.8: **C Integral Data types.** Text in square brackets is optional.

2.2 Integer Representations

In this section we describe two different ways bits can be used to encode integers—one that can only represent nonnegative numbers, and one that can represent negative, zero, and positive numbers. We will see later that they are strongly related both in their mathematical properties and their machine-level implementations. We also investigate the effect of expanding or shrinking an encoded integer to fit a representation with a different length.

2.2.1 Integral Data Types

C supports a variety of *integral* data types—ones that represent a finite range of integers. These are shown in Figure 2.8. Each type has a size designator: `char`, `short`, `int`, and `long`, as well as an indication of whether the represented number is nonnegative (declared as `unsigned`), or possibly negative (the default). The typical allocations for these different sizes were given in Figure 2.2. As indicated in Figure 2.8, these different sizes allow different ranges of values to be represented. The C standard defines a minimum range of values each data type must be able to represent. As shown in the figure, a typical 32-bit machine uses a 32-bit representation for data types `int` and `unsigned`, even though the C standard allows 16-bit representations. As described in Figure 2.2, the Compaq Alpha uses a 64-bit word to represent `long` integers, giving an upper limit of over 1.84×10^{19} for unsigned values, and a range of over $\pm 9.22 \times 10^{18}$ for signed values.

New to C?
Both C and C++ support signed (the default) and unsigned numbers. Java supports only signed numbers. **End**

2.2.2 Unsigned and Two's Complement Encodings

Assume we have an integer data type of w bits. We write a bit vector as either $\vec{x}$, to denote the entire vector, or as $[x_{w-1}, x_{w-2}, \ldots, x_0]$ to denote the individual bits within the vector. Treating $\vec{x}$ as a number written in binary notation, we obtain the *unsigned* interpretation of $\vec{x}$. We express this interpretation as a function

Quantity	Word Size w			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65,535	0xFFFFFFFF 4,294,967,295	0xFFFFFFFFFFFFFFFF 18,446,744,073,709,551,615
$TMax_w$	0x7F 127	0x7FFF 32,767	0x7FFFFFFF 2,147,483,647	0x7FFFFFFFFFFFFFFF 9,223,372,036,854,775,807
$TMin_w$	0x80 -128	0x8000 $-32,768$	0x80000000 $-2,147,483,648$	0x8000000000000000 $-9,223,372,036,854,775,808$
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Figure 2.9: **"Interesting" Numbers.** Both numeric values and hexadecimal representations are shown.

$B2U_w$ (for "binary to unsigned," length w):

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

(In this equation, the notation "$\doteq$" means that the left hand side is defined to equal to the right hand side). That is, function $B2U_w$ maps length w strings of 0s and 1s to nonnegative integers. The least value is given by bit vector $[00\cdots0]$ having integer value 0, and the greatest value is given by bit vector $[11\cdots1]$ having integer value $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$. Thus, the function $B2U_w$ can be defined as a mapping $B2U_w\colon \{0,1\}^w \rightarrow \{0,\ldots,2^w-1\}$. Note that $B2U_w$ is a *bijection*—it associates a unique value to each bit vector of length w, and conversely each integer between 0 and $2^w - 1$ has a unique binary representation as a bit vector of length w.

For many applications, we wish to represent negative values as well. The most common computer representation of signed numbers is known as *two's complement* form. This is defined by interpreting the most significant bit of the word to have negative weight. We express this interpretation as a function $B2T_w$ (for "binary to two's complement" length w):

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.2}$$

The most significant bit is also called the *sign bit*. When set to 1, the represented value is negative, and when set to 0 the value is nonnegative. The least representable value is given by bit vector $[10\cdots0]$ (i.e., set the bit with negative weight but clear all others) having integer value $TMin_w \doteq -2^{w-1}$. The greatest value is given by bit vector $[01\cdots1]$, having integer value $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. Again, one can see that $B2T_w$ is a bijection $B2T_w\colon \{0,1\}^w \rightarrow \{-2^{w-1},\ldots,2^{w-1}-1\}$, associating a unique integer in the representable range for each bit pattern.

Figure 2.9 shows the bit patterns and numeric values for several "interesting" numbers for different word sizes. The first three give the ranges of representable integers. A few points are worth highlighting. First, the two's complement range is asymmetric: $|TMin_w| = |TMax_w| + 1$, that is, there is no positive counterpart to $TMin_w$. As we shall see, this leads to some peculiar properties of two's complement arithmetic and can

be the source of subtle program bugs. Second, the maximum unsigned value is nearly twice the maximum two's complement value: $UMax_w = 2\,TMax_w + 1$. This follows from the fact that two's complement notation reserves half of the bit patterns to represent negative values. The other cases are the constants -1 and 0. Note that -1 has the same bit representation as $UMax_w$—a string of all 1s. Numeric value 0 is represented as a string of all 0s in both representations.

The C standard does not require signed integers to be represented in two's complement form, but nearly all machines do so. To keep code portable, one should not assume any particular range of representable values or how they are represented, beyond the ranges indicated in Figure 2.2. The C library file `<limits.h>` defines a set of constants delimiting the ranges of the different integer data types for the particular machine on which the compiler is running. For example, it defines constants `INT_MAX`, `INT_MIN`, and `UINT_MAX` describing the ranges of signed and unsigned integers. For a two's complement machine where data type `int` has w bits, these constants correspond to the values of $TMax_w$, $TMin_w$, and $UMax_w$.

Practice Problem 2.12:

Assuming $w = 4$, we can assign a numeric value to each possible hex digit, assuming either an unsigned or two's complement interpretation. Fill in the following table according to these interpretations

$\vec{x}$ (Hex)	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
0		
3		
8		
A		
F		

Aside: Alternative represenations of signed numbers
There are two other standard representations for signed numbers:

One's Complement: Same as two's complement, except that the most significant bit has weight $-(2^{w-1}-1)$ rather than -2^{w-1}:

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1}-1) + \sum_{i=0}^{w-2} x_i 2^i$$

Sign-Magnitude: The most significant bit is a sign bit that determines whether the remaining bits should be given negative or positive weight:

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i \right)$$

Both of these representations have the curious property that there are two different encodings of the number 0. For both representations, $[00\cdots0]$ is interpreted as $+0$. The value -0 can be represented in sign-magnitude as $[10\cdots0]$ and in one's complement as $[11\cdots1]$. Although machines based on one's complement representations were built in the past, almost all modern machines use two's complement. We will see that sign-magnitude encoding is used with floating-point numbers. **End Aside.**

As an example, consider the following code:

Weight	12,345		−12,345		53,191	
	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4096	0	0	0	0
8,192	1	8192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
$\pm 32,768$	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

Figure 2.10: **Two's Complement Representations of 12,345 and −12,345, and Unsigned Representation of 53,191.** Note that the latter two have identical bit representations.

```
1     short int x = 12345;
2     short int mx = -x;
3
4     show_bytes((byte_pointer) &x, sizeof(short int));
5     show_bytes((byte_pointer) &mx, sizeof(short int));
```

When run on a big-endian machine, this code prints `30 39` and `cf c7`, indicating that `x` has hexadecimal representation `0x3039`, while `mx` has hexadecimal representation `0xCFC7`. Expanding these into binary we get bit patterns $[0011000000111001]$ for `x` and $[1100111111000111]$ for `mx`. As Figure 2.10 shows, Equation 2.2 yields values 12,345 and $-12{,}345$ for these two bit patterns.

2.2.3 Conversions Between Signed and Unsigned

Since both $B2U_w$ and $B2T_w$ are bijections, they have well-defined inverses. Define $U2B_w$ to be $B2U_w^{-1}$, and $T2B_w$ to be $B2T_2^{-1}$. These functions give the unsigned or two's complement bit patterns for a numeric value. Given an integer x in the range $0 \leq x < 2^w$, the function $U2B_w(x)$ gives the unique w-bit unsigned representation of x. Similarly, when x is in the range $-2^{w-1} \leq x < 2^{w-1}$, the function $T2B_w(x)$ gives the unique w-bit two's complement representation of x. Observe that for values in the range $0 \leq x < 2^{w-1}$, both of these functions will yield the same bit representation—the most significant bit will be 0, and hence it does not matter whether this bit has positive or negative weight.

Consider the following function: $U2T_w(x) \doteq B2T_w(U2B_w(x))$. This function takes a number between 0 and $2^{w-1} - 1$ and yields a number between -2^{w-1} and $2^{w-1} - 1$, where the two numbers have identical bit representations, except that the argument is unsigned, while the result has a two's complement representation. Conversely, the function $T2U_w(x) \doteq B2U_w(T2B_w(x))$ yields the unsigned number having the same bit representation as the two's complement value of `x`. For example, as Figure 2.10 indicates, the 16-bit, two's complement representation of $-12{,}345$ is identical to the 16-bit, unsigned representation of 53,191. Therefore $T2U_{16}(-12,345) = 53,191$, and $U2T_{16}(53,191) = -12,345$.

These two functions might seem purely of academic interest, but they actually have great practical importance. They formally define the effect of casting between signed and unsigned values in C. For example, consider executing the following code on a two's complement machine:

```
1     int x = -1;
2     unsigned ux = (unsigned) x;
```

This code will set `ux` to $UMax_w$, where w is the number of bits in data type `int`, since by Figure 2.9 we can see that the w-bit two's complement representation of -1 has the same bit representation as $UMax_w$. In general, casting from a signed value `x` to unsigned value `(unsigned) x` is equivalent to applying function $T2U$. The cast does not change the bit representation of the argument, just how these bits are interpreted as a number. Similarly, casting from unsigned value `u` to signed value `(int) u` is equivalent to applying function $U2T$.

Practice Problem 2.13:

Using the table you filled in when solving Problem 2.12, fill in the following table describing the function $T2U_4$:

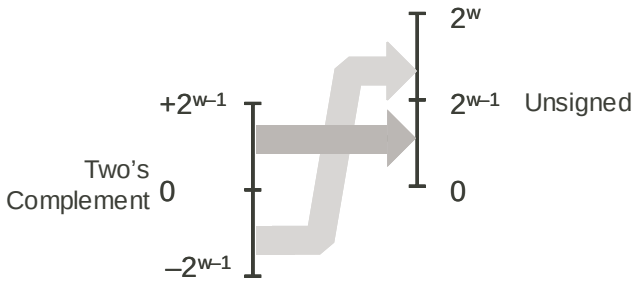


Figure 2.11: **Conversion From Two's Complement to Unsigned.** Function $T2U$ converts negative numbers to large positive numbers.

x	$T2U_4(x)$
-8	
-6	
-1	
0	
3	

To get a better understanding of the relation between a signed number x and its unsigned counterpart $T2U_w(x)$, we can use the fact that they have identical bit representations to derive a numerical relationship. Comparing Equations 2.1 and 2.2, we can see that for bit pattern $\vec{x}$, if we compute the difference $B2U_w(\vec{x}) - B2T_w(\vec{x})$, the weighted sums for bits from 0 to $w-2$ will cancel each other, leaving a value: $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$. This gives a relationship $B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$. If we let $x = B2T_w(\vec{x})$, we then have

$$B2U_w(T2B_w(x)) \quad = \quad T2U_w(x) \quad = \quad x_{w-1}2^w + x \tag{2.3}$$

This relationship is useful for proving relationships between unsigned and two's complement arithmetic. In the two's complement representation of x, bit x_{w-1} determines whether or not x is negative, giving

$$T2U_w(x) \quad = \quad \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.4}$$

Figure 2.11 illustrates the behavior of function $T2U$. As it illustrates, when mapping a signed number to its unsigned counterpart, negative numbers are converted to large positive numbers, while nonnegative numbers remain unchanged.

Practice Problem 2.14:

Explain how Equation 2.4 applies to the entries in the table you generated when solving Problem 2.13.

Going in the other direction, we wish to derive the relationship between an unsigned number x and its signed counterpart $U2T_w(x)$. If we let $x = B2U_w(\vec{x})$, we have

$$B2T_w(U2B_w(x)) \quad = \quad U2T_w(x) \quad = \quad -x_{w-1}2^w + x \tag{2.5}$$

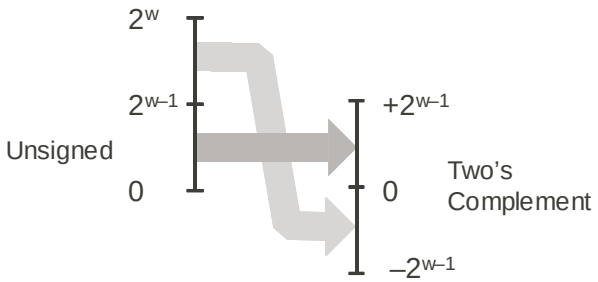


Figure 2.12: **Conversion From Unsigned to Two's Complement.** Function $U2T$ converts numbers greater than $2^{w-1} - 1$ to negative values.

In the unsigned representation of x, bit x_{w-1} determines whether or not x is greater than or equal to 2^{w-1}, giving

$$U2T_w(x) \quad = \quad \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases} \tag{2.6}$$

This behavior is illustrated in Figure 2.12. For small ($< 2^{w-1}$) numbers, the conversion from unsigned to signed preserves the numeric value. For large ($\geq 2^{w-1}$) the number is converted to a negative value.

To summarize, we can consider the effects of converting in both directions between unsigned and two's complement representations. For values in the range $0 \leq x < 2^{w-1}$ we have $T2U_w(x) = x$ and $U2T_w(x) = x$. That is, numbers in this range have identical unsigned and two's complement representations. For values outside of this range, the conversions either add or subtract 2^w. For example, we have $T2U_w(-1) = -1+2^w = UMax_w$—the negative number closest to 0 maps to the largest unsigned number. At the other extreme, one can see that $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$—the most negative number maps to an unsigned number just outside the range of positive, two's complement numbers. Using the example of Figure 2.10, we can see that $T2U_{16}(-12,345) = 65,536 + -12,345 = 53,191$.

2.2.4 Signed vs. Unsigned in C

As indicated in Figure 2.8, C supports both signed and unsigned arithmetic for all of its integer data types. Although the C standard does not specify a particular representation of signed numbers, almost all machines use two's complement. Generally, most numbers are signed by default. For example, when declaring a constant such as `12345` or `0x1A2B`, the value is considered signed. To create an unsigned constant, the character '`U`' or '`u`' must be added as suffix, e.g., `12345U` or `0x1A2Bu`.

C allows conversion between unsigned and signed. The rule is that the underlying bit representation is not changed. Thus, on a two's complement machine, the effect is to apply the function $U2T_w$ when converting from unsigned to signed, and $T2U_w$ when converting from signed to unsigned, where w is the number of bits for the data type.

Conversions can happen due to explicit casting, such as in the code:

```
1     int tx, ty;
```

```
2     unsigned ux, uy;
3
4     tx = (int) ux;
5     uy = (unsigned) ty;
```

or implicitly when an expression of one type is assigned to a variable of another, as in the code:

```
1     int tx, ty;
2     unsigned ux, uy;
3
4     tx = ux; /* Cast to signed */
5     uy = ty; /* Cast to unsigned */
```

When printing numeric values with `printf`, the directives `%d`, `%u`, and `%x` should be used to print a number as a signed decimal, an unsigned decimal, and in hexadecimal format, respectively. Note that `printf` does not make use of any type information, and so it is possible to print a value of type `int` with directive `%u` and a value of type `unsigned` with directive `%d`. For example, consider the following code:

```
1     int x = -1;
2     unsigned u = 2147483648; /* 2 to the 31st */
3
4     printf("x = %u = %d\n", x, x);
5     printf("u = %u = %d\n", u, u);
```

When run on a 32-bit machine it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

In both cases, `printf` prints the word first as if it represented an unsigned number and second as if it represented a signed number. We can see the conversion routines in action: $T2U_{32}(-1) = UMax_{32} = 4,294,967,295$ and $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$.

Some peculiar behavior arises due to C's handling of expressions containing combinations of signed and unsigned quantities. When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative. As we will see, this convention makes little difference for standard arithmetic operations, but it leads to nonintuitive results for relational operators such as `<` and `>`. Figure 2.13 shows some sample relational expressions and their resulting evaluations, assuming a 32-bit machine using two's complement representation. The nonintuitive cases are marked by '*'. Consider the comparison `-1 < 0U`. Since the second operand is unsigned, the first one is implicitly cast to unsigned, and hence the expression is equivalent to the comparison `4294967295U < 0U` (recall that $T2U_w(-1) = UMax_w$), which of course is false. The other cases can be understood by similar analyses.

Expression	Type	Evaluation
`0 == 0U`	unsigned	`1`
`-1 < 0`	signed	`1`
`-1 < 0U`	unsigned	`0 *`
`2147483647 > -2147483648`	signed	`1`
`2147483647U > -2147483648`	unsigned	`0 *`
`2147483647 > (int) 2147483648U`	signed	`1 *`
`-1 > -2`	signed	`1`
`(unsigned) -1 > -2`	unsigned	`0 *`

Figure 2.13: **Effects of C Promotion Rules on 32-Bit Machine.** Nonintuitive cases marked by '*'. When either operand of a comparison is unsigned, the other operand is implicitly cast to unsigned.

2.2.5 Expanding the Bit Representation of a Number

One common operation is to convert between integers having different word sizes, while retaining the same numeric value. Of course, this may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to a larger data type, however, should always be possible. To convert an unsigned number to a larger data type, we can simply add leading 0s to the representation. this operation is known as *zero extension*. For converting a two's complement number to a larger data type, the rule is to perform a *sign extension*, adding copies of the most significant bit to the representation. Thus, if our original value has bit representation $[x_{w-1}, x_{w-2}, \ldots, x_0]$, the expanded representation would be $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$.

As an example, consider the following code:

```
1     short sx = val;          /* -12345 */
2     unsigned short usx = sx; /* 53191 */
3     int   x = sx;            /* -12345 */
4     unsigned  ux = usx;      /* 53191 */
5
6     printf("sx  = %d:\t", sx);
7     show_bytes((byte_pointer) &sx, sizeof(short));
8     printf("usx = %u:\t", usx);
9     show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10    printf("x   = %d:\t", x);
11    show_bytes((byte_pointer) &x, sizeof(int));
12    printf("ux  = %u:\t", ux);
13    show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run on a 32-bit, big-endian machine using two's complement representations this code prints:

```
sx  = -12345:  cf c7
usx = 53191:   cf c7
x   = -12345:  ff ff cf c7
ux  = 53191:   00 00 cf c7
```

We see that although the two's complement representation of $-12{,}345$ and the unsigned representation of 53,191 are identical for a 16-bit word size, they differ for a 32-bit word size. In particular, $-12{,}345$ has hexadecimal representation `0xFFFFCFC7`, while 53,191 has hexadecimal representation `0x0000CFC7`. The former has been sign-extended—16 copies of the most significant bit 1, having hexadecimal representation `0xFFFF`, have been added as leading bits. The latter has been extended with 16 leading 0s, having hexadecimal representation `0x0000`.

Can we justify that sign extension works? What we want to prove is that

$$B2T_{w+k}([x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) \quad = \quad B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

where in the expression on the left-hand side, we have made k additional copies of bit x_{w-1}. The proof follows by induction on k. That is, if we can prove that sign-extending by one bit preserves the numeric value, then this property will hold when sign-extending by an arbitrary number of bits. Thus, the task reduces to proving that

$$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) \quad = \quad B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

Expanding the left-hand expression with Equation 2.2 gives

$$\begin{aligned}
B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\
&= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])
\end{aligned}$$

The key property we exploit is that $-2^w + 2^{w-1} = -2^{w-1}$. Thus, the combined effect of adding a bit of weight -2^w and of converting the bit having weight -2^{w-1} to be one with weight 2^{w-1} is to preserve the original numeric value.

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program. Consider the following additional code for our previous example:

```
1     unsigned  uy = x;        /* Mystery! */
2
3     printf("uy  = %u:\t", uy);
4     show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

This portion of the code causes the following to be printed:

```
uy = 4294954951:  ff ff cf c7
```

This shows that the expressions:

```
(unsigned) (int) sx           /* 4294954951 */
```

and

```
(unsigned) (unsigned short) sx     /* 53191 */
```

produce different values, even though the original and the final data types are the same. In the former expression, we first sign extend the 16-bit `short` to a 32-bit `int`, whereas zero extension is performed in the latter expression.

2.2.6 Truncating Numbers

Suppose that rather than extending a value with extra bits, we reduce the number of bits representing a number. This occurs, for example, in the code:

```
1     int   x = 53191;
2     short sx = (short) x;  /* -12345 */
3     int   y = sx;          /* -12345 */
```

On a typical 32-bit machine, when we cast `x` to be `short`, we truncate the 32-bit `int` to be a 16-bit `short int`. As we saw before, this 16-bit pattern is the two's complement representation of $-12{,}345$. When we cast this back to `int`, sign extension will set the high-order 16 bits to 1s, yielding the 32-bit two's complement representation of $-12{,}345$.

When truncating a w-bit number $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ to a k-bit number, we drop the high-order $w-k$ bits, giving a bit vector $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Truncating a number can alter its value—a form of overflow. We now investigate what numeric value will result. For an unsigned number x, the result of truncating it to k bits is equivalent to computing $x \bmod 2^k$. This can be seen by applying the modulus operation to Equation 2.1:

$$\begin{aligned} B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\ &= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\ &= \sum_{i=0}^{k-1} x_i 2^i \\ &= B2U_k([x_k, x_{k-1}, \ldots, x_0]) \end{aligned}$$

In the above derivation we make use of the property that $2^i \bmod 2^k = 0$ for any $i \geq k$, and that $\sum_{i=0}^{k-1} x_i 2^i \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$.

For a two's complement number x, a similar argument shows that $B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k = B2U_k([x_k, x_{k-1}, \ldots, x_0])$. That is, $x \bmod 2^k$ can be represented by an unsigned number having bit-level representation $[x_{k-1}, \ldots, x_0]$. In general, however, we treat the truncated number as being signed. This will have numeric value $U2T_k(x \bmod 2^k)$.

Summarizing, the effects of truncation are:

$$B2U_k([x_k, x_{k-1}, \ldots, x_0]) = B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k \quad (2.7)$$

$$B2T_k([x_k, x_{k-1}, \ldots, x_0]) = U2T_k(B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k) \quad (2.8)$$

Practice Problem 2.15:

Suppose we truncate a four-bit value (represented by hex digits `0` through `F`) to a three-bit value (represented as hex digits `0` through `7`). Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's complement interpretations of those bit patterns.

Hex		Unsigned		Two's Complement	
Original	Truncated	Original	Truncated	Original	Truncated
`0`	`0`	0		0	
`3`	`3`	3		3	
`8`	`0`	8		−8	
`A`	`2`	10		−6	
`F`	`7`	15		−1	

Explain how Equations 2.7 and 2.8 apply to these cases.

2.2.7 Advice on Signed vs. Unsigned

As we have seen, the implicit casting of signed to unsigned leads to some nonintuitive behavior. Nonintuitive features often lead to program bugs, and ones involving the nuances of implicit casting can be especially difficult to see. Since the casting is invisible, we can often overlook its effects.

Practice Problem 2.16:

Consider the following code that attempts to sum the elements of an array `a`, where the number of elements is given by parameter `length`:

```
1 /* WARNING: This is buggy code */
2 float sum_elements(float a[], unsigned length)
3 {
4     int i;
5     float result = 0;
6
7     for (i = 0; i <= length-1; i++)
8         result += a[i];
9     return result;
10 }
```

When run with argument `length` equal to 0, this code should return 0.0. Instead it encounters a memory error. Explain why this happens. Show how this code can be corrected.

One way to avoid such bugs is to never use unsigned numbers. In fact, few languages other than C support unsigned integers. Apparently these other language designers viewed them as more trouble than they are worth. For example, Java supports only signed integers, and it requires that they be implemented with two's complement arithmetic. The normal right shift operator `>>` is guaranteed to perform an arithmetic shift. The special operator `>>>` is defined to perform a logical right shift.

Unsigned values are very useful when we want to think of words as just collections of bits with no numeric interpretation. This occurs, for example, when packing a word with *flags* describing various Boolean conditions. Addresses are naturally unsigned, so systems programmers find unsigned types to be helpful. Unsigned values are also useful when implementing mathematical packages for modular arithmetic and for multiprecision arithmetic, in which numbers are represented by arrays of words.

2.3 Integer Arithmetic

Many beginning programmers are surprised to find that adding two positive numbers can yield a negative result, and that the comparison `x < y` can yield a different result than the comparison `x-y < 0`. These properties are artifacts of the finite nature of computer arithmetic. Understanding the nuances of computer arithmetic can help programmers write more reliable code.

2.3.1 Unsigned Addition

Consider two nonnegative integers x and y, such that $0 \leq x, y \leq 2^w - 1$. Each of these numbers can be represented by w-bit unsigned numbers. If we compute their sum, however, we have a possible range $0 \leq x+y \leq 2^{w+1}-2$. Representing this sum could require $w+1$ bits. For example, Figure 2.14 shows a plot of the function $x + y$ when x and y have four-bit representations. The arguments (shown on the horizontal axes) range from 0 to 15, but the sum ranges from 0 to 30. The shape of the function is a sloping plane. If we were to maintain the sum as a $w + 1$ bit number and add it to another value, we may require $w + 2$ bits, and so on. This continued "word size inflation" means we cannot place any bound on the word size required to fully represent the results of arithmetic operations. Some programming languages, such as Lisp, actually support *infinite precision* arithmetic to allow arbitrary (within the memory limits of the machine, of course) integer arithmetic. More commonly, programming languages support fixed-precision arithmetic, and hence operations such as "addition" and "multiplication" differ from their counterpart operations over integers.

Unsigned arithmetic can be viewed as a form of modular arithmetic. Unsigned addition is equivalent to computing the sum modulo 2^w. This value can be computed by simply discarding the high-order bit in the $w + 1$-bit representation of $x + y$. For example, consider a four-bit number representation with $x = 9$ and $y = 12$, having bit representations $[1001]$ and $[1100]$, respectively. Their sum is 21, having a 5-bit representation $[10101]$. But if we discard the high-order bit, we get $[0101]$, that is, decimal value 5. This matches the value $21 \bmod 16 = 5$.

In general, we can see that if $x+y < 2^w$, the leading bit in the $w+1$-bit representation of the sum will equal

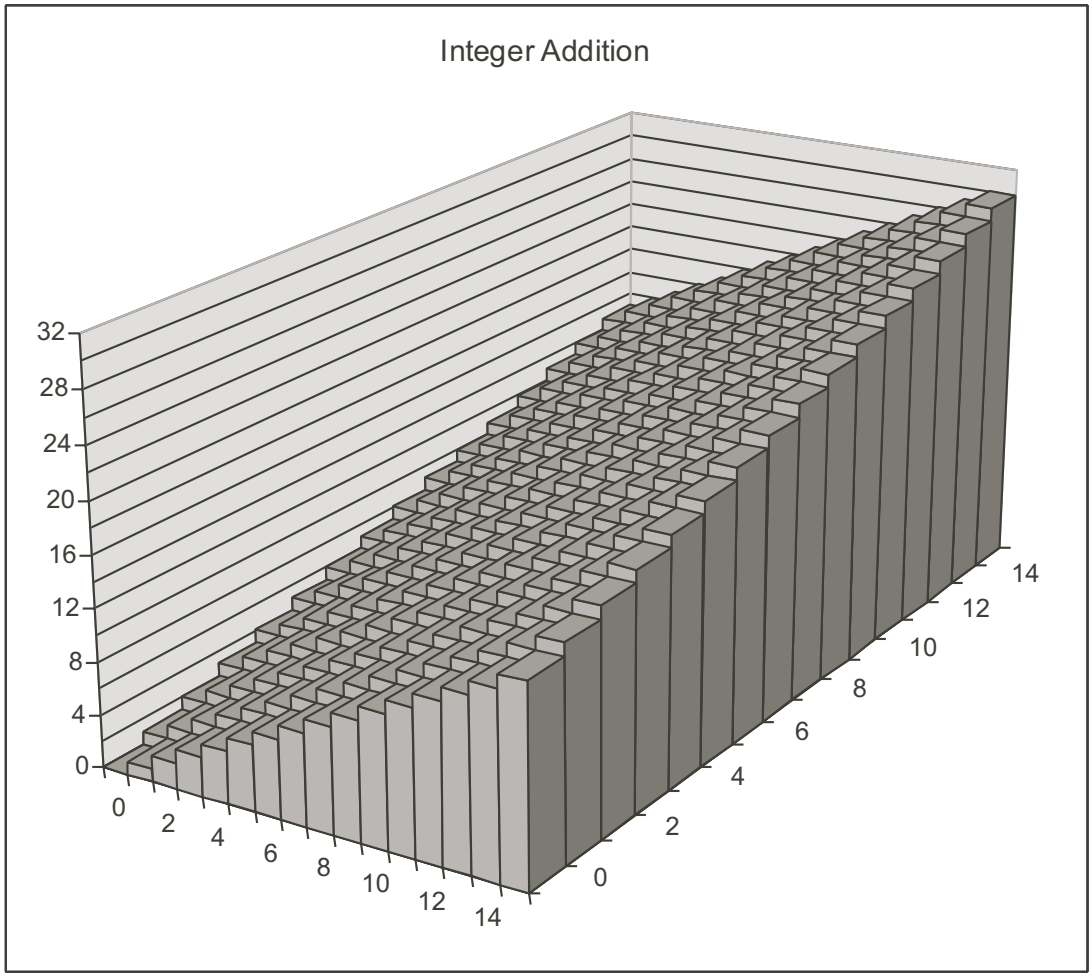


Figure 2.14: **Integer Addition.** With a four-bit word size, the sum could require 5 bits.

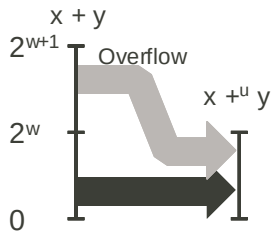


Figure 2.15: **Relation Between Integer Addition and Unsigned Addition.** When $x + y$ is greater than $2^w - 1$, the sum overflows.

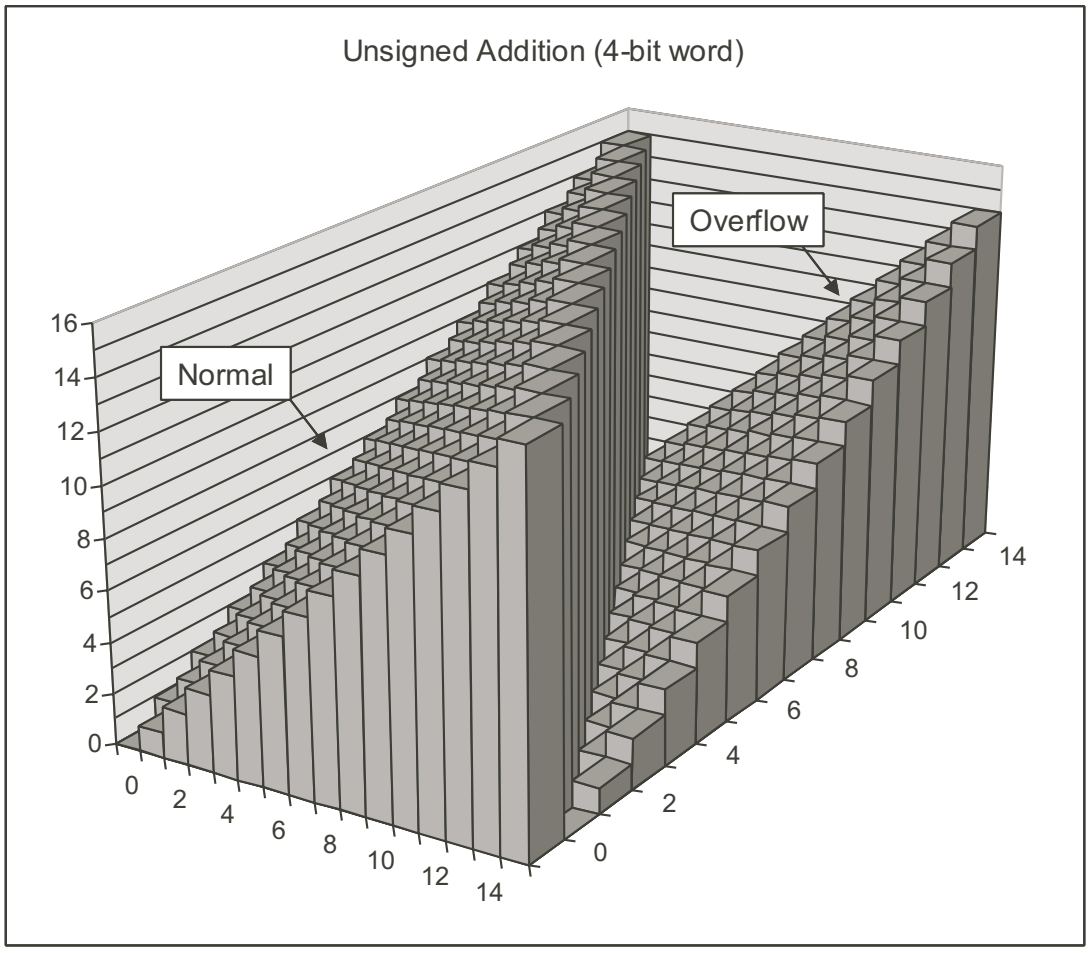


Figure 2.16: **Unsigned Addition.** With a four-bit word size, addition is performed modulo 16.

0, and hence discarding it will not change the numeric value. On the other hand, if $2^w \leq x + y < 2^{w+1}$, the leading bit in the $w + 1$-bit representation of the sum will equal 1, and hence discarding it is equivalent to subtracting 2^w from the sum. These two cases are illustrated in Figure 2.15. This will give us a value in the range $0 \leq x + y - 2^w < 2^{w+1} - 2^w = 2^w$, which is precisely the modulo 2^w sum of x and y. Let us define the operation $+^{\mathrm{u}}_w$ for arguments x and y such that $0 \leq x, y < 2^w$ as:

$$x +^{\mathrm{u}}_w y \quad = \quad \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \end{cases} \tag{2.9}$$

This is precisely the result we get in C when performing addition on two w-bit unsigned values.

An arithmetic operation is said to *overflow* when the full integer result cannot fit within the word size limits of the data type. As Equation 2.9 indicates, overflow occurs when the two operands sum to 2^w or more. Figure 2.16 shows a plot of the unsigned addition function for word size $w = 4$. The sum is computed modulo $2^4 = 16$. When $x + y < 16$, there is no overflow, and $x +^{\mathrm{u}}_4 y$ is simply $x + y$. This is shown as the region forming a sloping plane labeled "Normal." When $x + y \geq 16$, the addition overflows, having

the effect of decrementing the sum by 16. This is shown as the region forming a sloping plane labeled "Overflow."

When executing C programs, overflows are not signalled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute $s \doteq x +^{\mathrm{u}}_{w} y$, and we wish to determine whether s equals $x + y$. We claim that overflow has occurred if and only if $s < x$ (or equivalently $s < y$.) To see this, observe that $x + y \geq x$, and hence if s did not overflow, we will surely have $s \geq x$. On the other hand, if s did overflow, we have $s = x + y - 2^w$. Given that $y < 2^w$, we have $y - 2^w < 0$, and hence $s = x + y - 2^w < x$. In our earlier example, we saw that $9 +^{\mathrm{u}}_{4} 12 = 5$. We can see that overflow occurred, since $5 < 9$.

Modular addition forms a mathematical structure known as an *Abelian group*, named after the Danish mathematician Niels Henrik Abel (1802–1829). That is, it is commutative (that's where the "Abelian" part comes in) and associative. It has an identity element 0, and every element has an additive inverse. Let us consider the set of w-bit unsigned numbers with addition operation $+^{\mathrm{u}}_{w}$. For every value x, there must be some value $-^{\mathrm{u}}_{w} x$ such that $-^{\mathrm{u}}_{w} x +^{\mathrm{u}}_{w} x = 0$. When $x = 0$, the additive inverse is clearly 0. For $x > 0$, consider the value $2^w - x$. Observe that this number is in the range $0 \leq 2^w - x < 2^w$, and $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Hence it is the inverse of x under $+^{\mathrm{u}}_{w}$. These two cases lead to the following equation for $0 \leq x < 2^w$:

$$-^{\mathrm{u}}_{w} x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{2.10}$$

Practice Problem 2.17:

We can represent a bit pattern of length $w = 4$ with a single hex digit. For an unsigned interpretation of these digits use Equation 2.10 fill in the following table giving the values and the bit representations (in hex) of the unsigned additive inverses of the digits shown.

x		$-^{\mathrm{u}}_{4} x$	
Hex	Decimal	Decimal	Hex
0			
3			
8			
A			
F			

2.3.2 Two's Complement Addition

A similar problem arises for two's complement addition. Given integer values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, their sum is in the range $-2^w \leq x + y \leq 2^w - 2$, potentially requiring $w + 1$ bits to represent exactly. As before, we avoid ever-expanding data sizes by truncating the representation to w bits. The result is not as familiar mathematically as modular addition, however.

The w-bit two's complement sum of two numbers has the exact same bit-level representation as the unsigned sum. In fact, most computers use the same machine instruction to perform either unsigned or signed

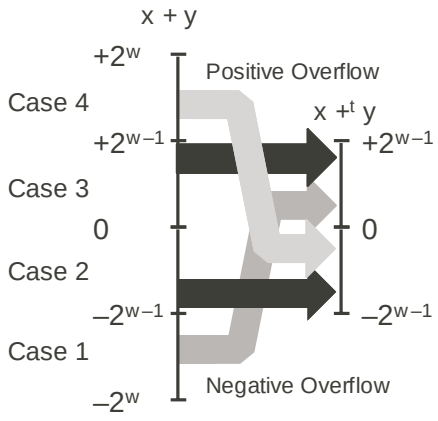


Figure 2.17: **Relation Between Integer and Two's Complement Addition.** When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than $2^{w-1} + 1$, there is a positive overflow.

addition. Thus, we can define two's complement addition for word size w, denoted as $+^{\mathrm{t}}_w$ on operands x and y such that $-2^{w-1} \leq x, y < 2^{w-1}$ as

$$x +^{\mathrm{t}}_w y \doteq U2T_w(T2U_w(x) +^{\mathrm{u}}_w T2U_w(y)) \tag{2.11}$$

By Equation 2.3 we can write $T2U_w(x)$ as $x_{w-1}2^w + x$, and $T2U_w(y)$ as $y_{w-1}2^w + y$. Using the property that $+^{\mathrm{u}}_w$ is simply addition modulo 2^w, along with the properties of modular addition, we then have

$$\begin{aligned} x +^{\mathrm{t}}_w y &= U2T_w(T2U_w(x) +^{\mathrm{u}}_w T2U_w(y)) \\ &= U2T_w[(-x_{w-1}2^w + x + -y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x + y) \bmod 2^w] \end{aligned}$$

The terms $x_{w-1}2^w$ and $y_{w-1}2^w$ drop out since they equal 0 modulo 2^w.

To better understand this quantity, let us define z as the integer sum $z \doteq x + y$, z' as $z' \doteq z \bmod 2^w$, and z'' as $z'' \doteq U2T_w(z')$. The value z'' is equal to $x +^{\mathrm{t}}_w y$. We can divide the analysis into four cases as illustrated in Figure 2.17:

1. $-2^w \leq z < -2^{w-1}$. Then we will have $z' = z + 2^w$. This gives $0 \leq z' < -2^{w-1} + 2^w = 2^{w-1}$. Examining Equation 2.6, we see that z' is in the range such that $z'' = z'$. This case is referred to as *negative overflow*. We have added two negative numbers x and y (that's the only way we can have $z < -2^{w-1}$) and obtained a nonnegative result $z'' = x + y + 2^w$.

2. $-2^{w-1} \leq z < 0$. Then we will again have $z' = z + 2^w$, giving $-2^{w-1} + 2^w = 2^{w-1} \leq z' < 2^w$. Examining Equation 2.6, we see that z' is in such a range that $z'' = z' - 2^w$, and therefore $z'' = z' - 2^w = z + 2^w - 2^w = z$. That is, our two's complement sum z'' equals the integer sum $x + y$.

3. $0 \leq z < 2^{w-1}$. Then we will have $z' = z$, giving $0 \leq z' < 2^{w-1}$, and hence $z'' = z' = z$. Again, the two's complement sum z'' equals the integer sum $x + y$.

x	y	$x+y$	$x +^{\mathrm{t}}_4 y$	Case
-8	-5	-13	3	1
$[1000]$	$[1011]$		$[0011]$	
-8	-8	-16	0	1
$[1000]$	$[1000]$		$[0000]$	
-8	5	-3	-3	2
$[1000]$	$[0101]$		$[1101]$	
2	5	7	7	3
$[0010]$	$[0101]$		$[0111]$	
5	5	10	-6	4
$[0101]$	$[0101]$		$[1010]$	

Figure 2.18: **Two's Complement Addition Examples.** The bit-level representation of the four-bit two's complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

4. $2^{w-1} \leq z < 2^w$. We will again have $z' = z$, giving $2^{w-1} \leq z' < 2^w$. But in this range we have $z'' = z' - 2^w$, giving $z'' = x + y - 2^w$. This case is referred to as *positive overflow*. We have added two positive numbers x and y (that's the only way we can have $z \geq 2^{w-1}$) and obtained a negative result $z'' = x + y - 2^w$.

By the preceding analysis, we have shown that when operation $+^{\mathrm{t}}_w$ is applied to values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, we have

$$x +^{\mathrm{t}}_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y & \text{Positive Overflow} \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} & \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{Negative Overflow} \end{cases} \tag{2.12}$$

As an illustration, Figure 2.18 shows some examples of four-bit two's complement addition. Each example is labeled by the case to which it corresponds in the derivation of Equation 2.12. Note that $2^4 = 16$, and hence negative overflow yields a result 16 more than the integer sum, and positive overflow yields a result 16 less. We include bit-level representations of the operands and the result. Observe that the result can be obtained by performing binary addition of the operands and truncating the result to four bits.

Figure 2.19 illustrates two's complement addition for word size $w = 4$. The operands range between -8 and 7. When $x + y < -8$, two's complement addition has a negative underflow, causing the sum to be incremented by 16. When $-8 \leq x + y < 8$, the addition yields $x + y$. When $x + y \geq 8$, the addition has a positive overflow, causing the sum to be decremented by 16. Each of these three ranges forms a sloping plane in the figure.

Equation 2.12 also lets us identify the cases where overflow has occurred. When both x and y are negative, but $x +^{\mathrm{t}}_w y \geq 0$, we have negative overflow. When both x and y are positive, but $x +^{\mathrm{t}}_w y < 0$, we have positive overflow.

Practice Problem 2.18:

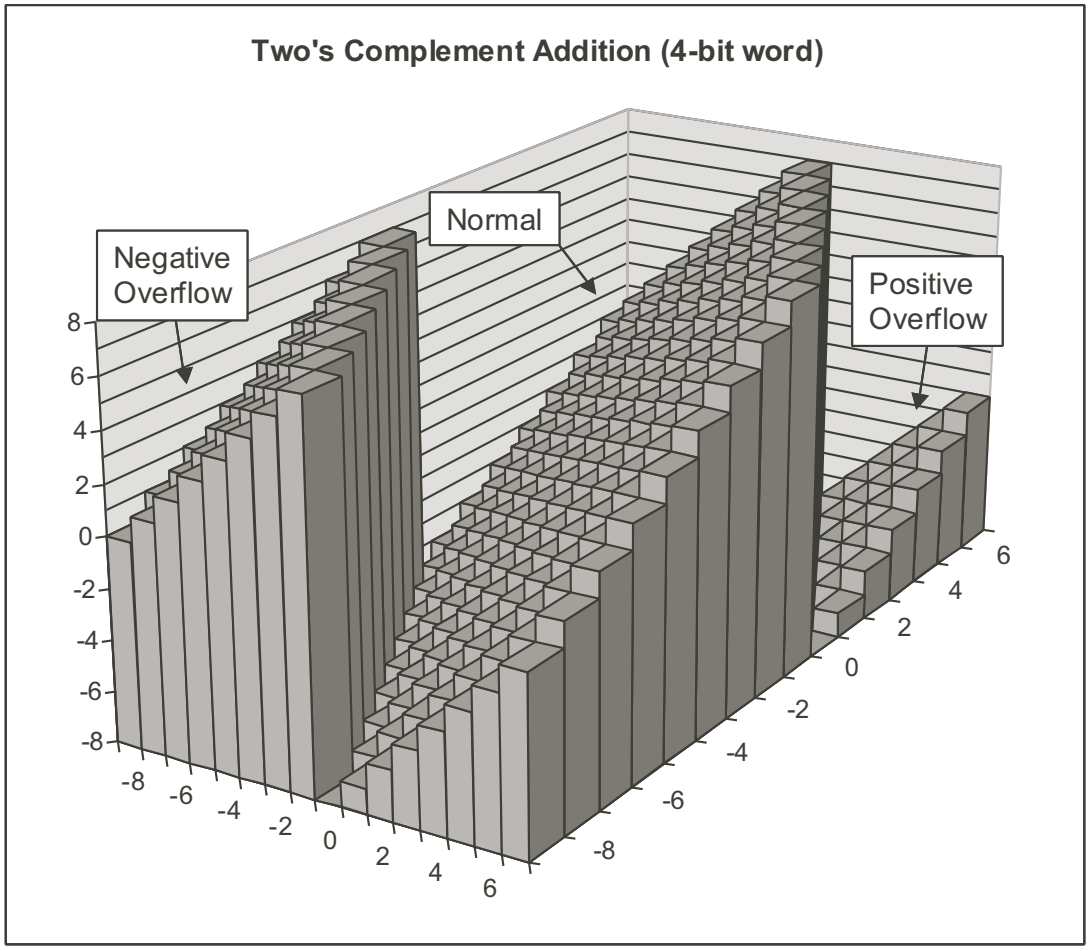


Figure 2.19: **Two's Complement Addition.** With a four-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

Fill in the following table in the style of Figure 2.18. Give the integer values of the 5-bit arguments, the values of both their integer and two's complement sums, the bit-level representation of the two's complement sum, and the case from the derivation of Equation 2.12.

x	y	$x + y$	$x +^t_4 y$	Case
[10000]	[10101]			
[10000]	[10000]			
[11000]	[00111]			
[11110]	[00101]			
[01000]	[01000]			

2.3.3 Two's Complement Negation

We can see that every number x in the range $-2^{w-1} \leq x < 2^{w-1}$ has an additive inverse under $+^t_w$ as follows. First, for $x \neq -2^{w-1}$, we can see that its additive inverse is simply $-x$. That is, we have $-2^{w-1} < -x < 2^{w-1}$ and $-x +^t_w x = -x + x = 0$. For $x = -2^{w-1} = TMin_w$, on the other hand, $-x = 2^{w-1}$ cannot be represented as a w-bit number. We claim that this special value has itself as the additive inverse under $+^t_w$. The value of $-2^{w+1} +^t_w -2^{w+1}$ is given by the third case of Equation 2.12, since $-2^{w-1} + -2^{w-1} = -2^w$. This gives $-2^{w+1} +^t_w -2^{w+1} = -2^w + 2^w = 0$. From this analysis we can define the two's complement negation operation $-^t_w$ for x in the range $-2^{2-1} \leq x < 2^{w-1}$ as:

$$-^t_w x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \tag{2.13}$$

Practice Problem 2.19:

We can represent a bit pattern of length $w = 4$ with a single hex digit. For a two's complement interpretation of these digits, fill in the following table to determine the additive inverses of the digits shown.

x		$-^t_4 x$	
Hex	Decimal	Decimal	Hex
0			
3			
8			
A			
F			

What do you observe about the bit patterns generated by two's complement and unsigned (Problem 2.17) negation.

A well-known technique for performing two's complement negation at the bit level is to complement the bits and then increment the result. In C, this can be written as `~x + 1`. To justify the correctness of this technique, observe that for any single bit x_i, we have $\tilde{}x_i = 1 - x_i$. Let $\vec{x}$ be a bit vector of length w and $x \doteq B2T_w(\vec{x})$ be the two's complement number it represents. By Equation 2.2, the complemented bit vector $\tilde{}\vec{x}$ has numeric value

$$\begin{aligned} B2T_w(\tilde{}\vec{x}) &= -(1 - x_{w-1})2^{w-1} + \sum_{i=0}^{w-2}(1 - x_i)2^i \\ &= \left[-2^{w-1} + \sum_{i=0}^{w-2} 2^i\right] - \left[-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i\right] \\ &= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\vec{x}) \\ &= -1 - x \end{aligned}$$

The key simplification in the above derivation is that $\sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. It follows that by incrementing $\tilde{}\vec{x}$ we obtain $-x$.

To increment a number x represented at the bit-level as $\vec{x} \doteq [x_{w-1}, x_{w-2}, \ldots, x_0]$, define the operation $incr$ as follows. Let k be the position of the rightmost zero, such that $\vec{x}$ is of the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 0, 1, \ldots, 1]$. We then define $incr(\vec{x})$ to be $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0]$. For the special case where the bit-level representation of x is $[1, 1, \ldots, 1]$, define $incr(\vec{x})$ to be $[0, \ldots, 0]$. To show that $incr(\vec{x})$ yields the bit-level representation of $x +^t_w 1$, consider the following cases:

1. When $\vec{x} = [1, 1, \ldots, 1]$, we have $x = -1$. The incremented value $incr(\vec{x}) \doteq [0, \ldots, 0]$ has numeric value 0.

2. When $k = w - 1$, i.e., $\vec{x} = [0, 1, \ldots, 1]$, we have $x = TMax_w$. The incremented value $incr(\vec{x}) = [1, 0, \ldots, 0]$ has numeric value $TMin_w$. From Equation 2.12, we can see that $TMax_w +^t_w 1$ is one of the positive overflow cases, yielding $TMin_w$.

3. When $k < w - 1$, i.e., $x \neq TMax_w$ and $x \neq -1$, we can see that the low-order $k + 1$ bits of $incr(\vec{x})$ has numeric value 2^k, while the low-order $k + 1$ bits of $\vec{x}$ has numeric value $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. The high-order $w - k + 1$ bits have matching numeric values. Thus, $incr(\vec{x})$ has numeric value $x + 1$. In addition, for $x \neq TMax_w$, adding 1 to x will not cause an overflow, and hence $x +^t_w 1$ has numeric value $x + 1$ as well.

As illustrations, Figure 2.20 shows how complementing and incrementing affect the numeric values of several four-bit vectors.

2.3.4 Unsigned Multiplication

Integers x and y in the range $0 \leq x, y \leq 2^w - 1$ can be represented as w-bit unsigned numbers, but their product $x \cdot y$ can range between 0 and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$. This could require as many as $2w$ bits to represent. Instead, unsigned multiplication in C is defined to yield the w-bit value given by the low-order

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
$[0101]$	5	$[1010]$	-6	$[1011]$	-5
$[0111]$	7	$[1000]$	-8	$[1001]$	-7
$[1100]$	-4	$[0011]$	3	$[0100]$	4
$[0000]$	0	$[1111]$	-1	$[0000]$	0
$[1000]$	-8	$[0111]$	7	$[1000]$	-8

Figure 2.20: **Examples of Complementing and Incrementing four-bit numbers.** The effect is to compute the two's value negation.

w bits of the $2w$-bit integer product. By Equation 2.7, this can be seen to be equivalent to computing the product modulo 2^w. Thus, the effect of the w-bit unsigned multiplication operation $*^{\mathrm{u}}_w$ is:

$$x *^{\mathrm{u}}_w y = (x \cdot y) \bmod 2^w \tag{2.14}$$

It is well known that modular arithmetic forms a ring. We can therefore deduce that unsigned arithmetic over w-bit numbers forms a ring $\langle \{0, \ldots, 2^w - 1\}, +^{\mathrm{u}}_w, *^{\mathrm{u}}_w, -^{\mathrm{u}}_w, 0, 1 \rangle$.

2.3.5 Two's Complement Multiplication

Integers x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ can be represented as w-bit two's complement numbers, but their product $x \cdot y$ can range between $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ and $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$. This could require as many as $2w$ bits to represent in two's complement form—most cases would fit into $2w - 1$ bits, but the special case of 2^{2w-2} requires the full $2w$ bits (to include a sign bit of 0). Instead, signed multiplication in C is generally performed by truncating the $2w$-bit product to w bits. By Equation 2.8, the effect of the w-bit two's complement multiplication operation $*^{\mathrm{t}}_w$ is:

$$x *^{\mathrm{t}}_w y = U2T_w((x \cdot y) \bmod 2^w) \tag{2.15}$$

We claim that the bit-level representation of the product operation is identical for both unsigned and two's complement multiplication. That is, given bit vectors $\vec{x}$ and $\vec{y}$ of length w, the bit-level representation of the unsigned product $B2U_w(\vec{x}) *^{\mathrm{u}}_w B2U_w(\vec{y})$ is identical to the bit-level representation of the two's complement product $B2T_w(\vec{x}) *^{\mathrm{t}}_w B2T_w(\vec{x})$. This implies that the machine can use a single type of multiply instruction to multiply both signed and unsigned integers.

To see this, let $x = B2T_w(\vec{x})$ and $y = B2T_w(\vec{y})$ be the two's complement values denoted by these bit patterns, and let $x' = B2U_w(\vec{x})$ and $y' = B2U_w(\vec{y})$ be the unsigned values. From Equation 2.3, we have $x' = x + x_{w-1}2^w$, and $y' = y + y_{w-1}2^w$. Computing the product of these values modulo 2^w gives:

$$\begin{aligned}
(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w && (2.16)\\
&= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w && (2.17)\\
&= (x \cdot y) \bmod 2^w && (2.18)
\end{aligned}$$

Thus, the low-order w bits of $x \cdot y$ and $x' \cdot y'$ are identical.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's Comp.	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's Comp.	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's Comp.	3	[011]	3	[011]	9	[001001]	1	[001]

Figure 2.21: **3-Bit Unsigned and Two's Complement Multiplication Examples.** Although the bit-level representations of the full products may differ, those of the truncated products are identical.

As illustrations, Figure 2.21 shows the results of multiplying different 3-bit numbers. For each pair of bit-level operands, we perform both unsigned and two's complement multiplication. Note that the unsigned, truncated product always equals $x \cdot y \mod 8$, and that the bit-level representations of both truncated products are identical.

Practice Problem 2.20:

Fill in the following table showing the results of multiplying different 3-bit numbers, in the style of Figure 2.21

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
Unsigned	[110]	[010]		
Two's Comp.	[110]	[010]		
Unsigned	[001]	[111]		
Two's Comp.	[001]	[111]		
Unsigned	[111]	[111]		
Two's Comp.	[111]	[111]		

We can see that unsigned arithmetic and two's complement arithmetic over w-bit numbers are isomorphic—the operations $+^{\mathrm{u}}_w$, $-^{\mathrm{u}}_w$, and $*^{\mathrm{u}}_w$ have the exact same effect at the bit level as do $+^{\mathrm{t}}_w$, $-^{\mathrm{t}}_w$, and $*^{\mathrm{t}}_w$. From this we can deduce that two's complement arithmetic forms a ring $\langle \{-2^{w-1}, \ldots, 2^{w-1} - 1\}, +^{\mathrm{t}}_w, *^{\mathrm{t}}_w, -^{\mathrm{t}}_w, 0, 1 \rangle$.

2.3.6 Multiplying by Powers of Two

On most machines, the integer multiply instruction is fairly slow—requiring 12 or more clock cycles—whereas other integer operations such as addition, subtraction, bit-level operations, and shifting require only one clock cycle. As a consequence, one important optimization used by compilers is to attempt to replace multiplications by constant factors with combinations of shift and addition operations.

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, we claim the bit-level representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k 0s have been

added to the right. This property can be derived using Equation 2.1:

$$\begin{aligned} B2U_{w+k}([x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\ &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \cdot 2^k \\ &= x2^k \end{aligned}$$

For $k < w$, we can truncate the shifted bit vector to be of length w, giving $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. By Equation 2.7, this bit-vector has numeric value $x2^k \mod 2^w = x *^{\mathrm{u}}_w 2^k$. Thus, for unsigned variable `x`, the C expression `x << k` is equivalent to `x * pwr2k`, where `pwr2k` equals 2^k. In particular, we can compute `pwr2k` as `1U << k`.

By similar reasoning, we can show that for a two's complement number x having bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and any k in the range $0 \leq k < w$, bit pattern $[x_{w-k-1}, \ldots, x_0, 0, \ldots, 0]$ will be the two's complement representation of $x *^{\mathrm{t}}_w 2^k$. Therefore, for signed variable `x` , the C expression `x << k` is equivalent to `x * pwr2k`, where `pwr2k` equals 2^k.

Note that multiplying by a power of two can cause overflow with either unsigned or two's complement arithmetic. Our result shows that even then we will get the same effect by shifting.

Practice Problem 2.21:

As we will see in Chapter 3, the `leal` instruction on an Intel-compatible processor can perform computations of the form `a<<k + b`, where `k` is either 0, 1, or 2, and `b` is either 0 or some program value. The compiler often uses this instruction to perform multiplications by constant factors. For example, we can compute `3*a` as `a<<1 + a`.

What multiples of `a` can be computed with this instruction?

2.3.7 Dividing by Powers of Two

Integer division on most machines is even slower than integer multiplication—requiring 30 or more clock cycles. Dividing by a power of two can also be performed using shift operations, but we use a right shift rather than a left shift. The two different shifts—logical and arithmetic—serves this purpose for unsigned and two's complement numbers, respectively.

Integer division always rounds toward zero. For $x \geq 0$ and $y > 0$, the result should be $\lfloor x/y \rfloor$, where for any real number a, $\lfloor a \rfloor$ is defined to be the unique integer a' such that $a' \leq a < a'+1$. As examples $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$, and $\lfloor 3 \rfloor = 3$.

Consider the effect of performing a logical right shift on an unsigned number. Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \leq k < w$. Let x' be the unsigned number with $w-k$-bit representation $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and x'' be the unsigned number with k-bit representation $[x_{k-1}, \ldots, x_0]$. We claim that $x' = \lfloor x/2^k \rfloor$. To see this, by Equation 2.1, we have $x = \sum_{i=0}^{w-1} x_i 2^i$, $x' = \sum_{i=k}^{w-k-1} x_i 2^{i-k}$ and $x'' = \sum_{i=0}^{k-1} x_i 2^i$. We can therefore write x as $x = 2^k x' + x''$.

Observe that $0 \leq x'' \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$, and hence $0 \leq x'' < 2^k$, implying that $\lfloor x''/2^k \rfloor = 0$. Therefore $\lfloor x/2^k \rfloor = \lfloor x' + x''/2^k \rfloor = x' + \lfloor x''/2^k \rfloor = x'$.

Observe that performing a logical right shift of bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ by k yields bit vector

$$[0, ..., 0, x_{w-1}, x_{w-2}, \ldots, x_k].$$

This bit vector has numeric value x'. That is, logically right shifting an unsigned number by k is equivalent to dividing it by 2^k. Therefore, for unsigned variable `x`, the C expression `x >> k` is equivalent to `x / pwr2k`, where `pwr2k` equals $2^{\texttt{k}}$.

Now consider the effect of performing an arithmetic right shift on a two's complement number. Let x be the two's complement integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \leq k < w$. Let x' be the two's complement number represented by the $w - k$ bits $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and x'' be the *unsigned* number represented by the low-order k bits $[x_{k-1}, \ldots, x_0]$. By a similar analysis as the unsigned case, we have $x = 2^k x' + x''$, and $0 \leq x'' < 2^k$, giving $x' = \lfloor x/2^k \rfloor$. Furthermore, observe that shifting bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ right *arithmetically* by k yields a bit vector

$$[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_k],$$

which is the sign extension from $w - k$ bits to w bits of $[x_{w-1}, x_{w-2}, \ldots, x_k]$. Thus, this shifted bit vector is the two's complement representation of $\lfloor x/y \rfloor$.

For $x \geq 0$, our analysis shows that this shifted result is the desired value. For $x < 0$ and $y > 0$, however, the result of integer division should be $\lceil x/y \rceil$, where for any real number a, $\lceil a \rceil$ is defined to be the unique integer a' such that $a' - 1 < a \leq a'$. That is, integer division should round negative results upward toward zero. For example the C expression `-5/2` yields `-2`. Thus, right shifting a negative number by k is not equivalent to dividing it by 2^k when rounding occurs. For example, the four-bit representation of -5 is $[1011]$. If we shift it right by one arithmetically we get $[1101]$, which is the two's complement representation of -3.

We can correct for this improper rounding by "biasing" the value before shifting. This technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. Thus, for $x < 0$, if we first add $2^k - 1$ to x before right shifting, we will get a correctly rounded result. This analysis shows that for a two's complement machine using arithmetic right shifts, the C expression `(x<0 ? (x + (1<<k)-1) : x) >> k` is equivalent to `x/pwr2k`, where `pwr2k` equals $2^{\texttt{k}}$. For example, to divide -5 by 2, we first add bias $2 - 1 = 1$ giving bit pattern $[1100]$. Right shifting this by one arithmetically gives bit pattern $[1110]$, which is the two's complement representation of -2.

Practice Problem 2.22:

In the following code, we have omitted the definitions of constants `M` and `N`:

```
#define M      /* Mystery number 1 */
#define N      /* Mystery number 2 */
int arith(int x, int y)
{
  int result = 0;
  result = x*M + y/N; /* M and N are mystery numbers. */
  return result;
}
```

We compiled this code for particular values of `M` and `N`. The compiler optimized the multiplication and division using the methods we have discussed. The following is a translation of the generated machine code back into C:

```
/* Translation of assembly code for arith */
int optarith(int x, int y)
{
  int t = x;
  x <<= 4;
  x -= t;
  if (y < 0) y += 3;
  y >>= 2;  /* Arithmetic shift */
  return x+y;
}
```

What are the values of `M` and `N`?

2.4 Floating Point

Floating-point representation encodes rational numbers of the form $V = x \times 2^y$. It is useful for performing computations involving very large numbers ($|V| \gg 0$), numbers very close to 0 ($|V| \ll 1$), and more generally as an approximation to real arithmetic.

Up until the 1980s, every computer manufacturer devised its own conventions for how floating-point numbers were represented and the details of the operations performed on them. In addition, they often did not worry too much about the accuracy of the operations, viewing speed and ease of implementation as being more critical than numerical precision.

All of this changed around 1985 with the advent of IEEE Standard 754, a carefully crafted standard for representing floating-point numbers and the operations performed on them. This effort started in 1976 under Intel's sponsorship with the design of the 8087, a chip that provided floating-point support for the 8086 processor. They hired William Kahan, a professor at the University of California, Berkeley, as a consultant to help design a floating point standard for its future processors. They allowed Kahan to join forces with a committee generating an industry-wide standard under the auspices of the Institute of Electrical and Electronics Engineers (IEEE). The committee ultimately adopted a standard close to the one Kahan had devised for Intel. Nowadays virtually all computers support what has become known as *IEEE floating point*. This has greatly improved the portability of scientific application programs across different machines.

Aside: The IEEE
The Institute of Electrical and Electronic Engineers (IEEE—pronounced "I-Triple-E") is a professional society that encompasses all of electronic and computer technology. They publish journals, sponsor conferences, and set up committees to define standards on topics ranging from power transmission to software engineering. **End Aside.**

In this section we will see how numbers are represented in the IEEE floating-point format. We will also explore issues of *rounding*, when a number cannot be represented exactly in the format and hence must be

adjusted upward or downward. We will then explore the mathematical properties of addition, multiplication, and relational operators. Many programmers consider floating point to be, at best, uninteresting and at worst, arcane and incomprehensible. We will see that since the IEEE format is based on a small and consistent set of principles, it is really quite elegant and understandable.

2.4.1 Fractional Binary Numbers

A first step in understanding floating-point numbers is to consider binary numbers having fractional values.

Let us first examine the more familiar decimal notation. Decimal notation uses a representation of the form: $d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$, where each decimal digit d_i ranges between 0 and 9. This notation represents a number

$$d \quad = \quad \sum_{i=-n}^{m} 10^i \times d_i$$

The weighting of the digits is defined relative to the decimal point symbol '.': digits to the left are weighted by positive powers of ten, giving integral values, while digits to the right are weighted by negative powers of ten, giving fractional values. For example, 12.34_{10} represents the number $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12\frac{34}{100}$.

By analogy, consider a notation of the form $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$, where each binary digit, or bit, b_i ranges between 0 and 1. This notation represents a number

$$b \quad = \quad \sum_{i=-n}^{m} 2^i \times b_i \tag{2.19}$$

The symbol '.' now becomes a *binary point*, with bits on the left being weighted by positive powers of two, and those on the right being weighted by negative powers of two. For example, 101.11_2 represents the number $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$,

One can readily see from Equation 2.19 that shifting the binary point one position to the left has the effect of dividing the number by two. For example, while 101.11_2 represents the number $5\frac{3}{4}$, 10.111_2 represents the number $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$. Similarly, shifting the binary point one position to the right has the effect of multiplying the number by two. For example, 1011.1_2 represents the number $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Note that numbers of the form $0.11 \cdots 1_2$ represent numbers just below 1. For example, 0.111111_2 represents $\frac{63}{64}$. We will use the shorthand notation $1.0 - \epsilon$ to represent such values.

Assuming we consider only finite-length encodings, decimal notation cannot represent numbers such as $\frac{1}{3}$ and $\frac{5}{7}$ exactly. Similarly, fractional binary notation can only represent numbers that can be written $x \times 2^y$. Other values can only be approximated. For example, although the number $\frac{1}{5}$ can be approximated with increasing accuracy by lengthening the binary representation, we cannot represent it exactly as a fractional binary number:

Representation	Value	Decimal
0.0_2	0	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

Practice Problem 2.23:

Fill in the missing information in the table below

Fractional Value	Binary Rep.	Decimal Rep.
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$		
$\frac{23}{16}$		
	10.1101	
	1.011	
		5.375
		3.0625

Practice Problem 2.24:

The imprecision of floating point arithmetic can have disastrous effects, as shown by the following (true) story. On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. The U. S. General Accounting Office (GAO) conducted a detailed analysis of the failure [49] and determined that the underlying cause was an imprecision in a numeric calculation. In this exercise, you will reproduce part of the GAO's analysis.

The Patriot system contains an internal clock, implemented as a counter that is incremented every 0.1 seconds. To determine the time in seconds, the program would multiply the value of this counter by a 24-bit quantity that was a fractional binary approximation to $\frac{1}{10}$. In particular, the binary representation of $\frac{1}{10}$ is the nonterminating sequence:

$$0.000110011[0011]\cdots_2$$

where the portion in brackets is repeated indefinitely. The computer approximated 0.1 using just the leading bit plus the first 23 bits of this sequence to the right of the binary point. Let us call this number x.

A. What is the binary representation of $x - 0.1$?

B. What is the approximate decimal value of $x - 0.1$?

C. The clock starts at 0 when the system is first powered up and keeps counting up from there. In this case, the system had been running for around 100 hours. What was the difference between the time computed by the software and the actual time?

D. The system predicts where an incoming missile will appear based on its velocity and the time of the last radar detection. Given that a Scud travels at around 2,000 meters per second, how far off was its prediction?

Normally, a slight error in the absolute time reported by a clock reading would not affect a tracking computation. Instead, it should depend on the relative time between two successive readings. The problem was that the Patriot software had been upgraded to use a more accurate function for reading time, but not all of the function calls had been replaced by the new code. As a result, the tracking software used the accurate time for one reading and the inaccurate time for the other [67].

2.4.2 IEEE Floating-Point Representation

Positional notation such as considered in the previous section would be very inefficient for representing very large numbers. For example, the representation of 5×2^{100} would consist of the bit pattern 101 followed by one hundred 0's. Instead, we would like to represent numbers in a form $x \times 2^y$ by giving the values of x and y.

The IEEE floating point standard represents a number in a form $V = (-1)^s \times M \times 2^E$:

- The *sign* s determines whether the number is negative ($s = 1$) or positive ($s = 0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The *significand* M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The *exponent* E weights the value by a (possibly negative) power of two.

The bit representation of a floating-point number is divided into three fields to encode these values:

- The single sign bit `s` directly encodes the sign s.
- The k-bit exponent field `exp` $= e_{k-1} \cdots e_1 e_0$ encodes the exponent E.
- The n-bit fraction field `frac` $= f_{n-1} \cdots f_1 f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

In the single-precision floating-point format (a `float` in C), fields `s`, `exp`, and `frac` are 1, $k = 8$, and $n = 23$ bits each, yielding a 32-bit representation. In the double-precision floating-point format (a `double` in C), fields `s`, `exp`, and `frac` are 1, $k = 11$, and $n = 52$ bits each, yielding a 64-bit representation.

The value encoded by a given bit representation can be divided into three different cases, depending on the value of `exp`.

Normalized Values

This is the most common case. They occur when the bit pattern of `exp` is neither all 0s (numeric value 0) or all 1s (numeric value 255 for single precision, 2047 for double). In this case, the exponent field is interpreted as representing a signed integer in *biased* form. That is, the exponent value is $E = e - Bias$ where e is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$, and $Bias$ is a bias value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double). This yields exponent ranges from -126 to $+127$ for single precision and -1022 to $+1023$ for double precision.

The fraction field `frac` is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the binary point to the left of the most significant bit. The significand is defined to be $M = 1 + f$. This is sometimes called an *implied leading 1* representation, because we can view M to be the number with binary representation $1.f_{n-1}f_{n-2} \cdots f_0$. This representation is a trick for getting an additional bit of precision for free, since we can always adjust the exponent E so that significand M is in the range $1 \leq M < 2$ (assuming there is no overflow). We therefore do not need to explicitly represent the leading bit, since it always equals 1.

Denormalized Values

When the exponent field is all 0s, the represented number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$, and the significand value is $M = f$, that is, the value of the fraction field without an implied leading 1.

> **Aside: Why set the bias this way for denormlized values?**
> Having the exponent value be $1 - Bias$ rather than simply $-Bias$ might seem counterintuitive. We will see shortly that it provides for smooth transition from denormalized to normalized values.**End Aside.**

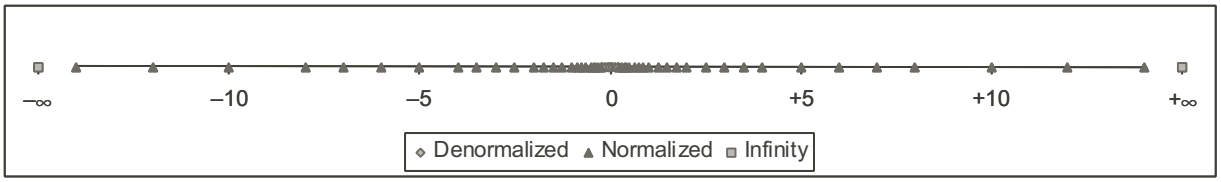
Denormalized numbers serve two purposes. First, they provide a way to represent numeric value 0, since with a normalized number we must always have $M \geq 1$, and hence we cannot represent 0. In fact the floating-point representation of $+0.0$ has a bit pattern of all 0s: the sign bit is 0, the exponent field is all 0s (indicating a denormalized value), and the fraction field is all 0s, giving $M = f = 0$. Curiously, when the sign bit is 1, but the other fields are all 0s, we get the value -0.0. With IEEE floating-point format, the values -0.0 and $+0.0$ are considered different in some ways and the same in others.

A second function of denormalized numbers is to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* in which possible numeric values are spaced evenly near 0.0.

Special Values

A final category of values occurs when the exponent field is all 1s. When the fraction field is all 0s, the resulting values represent infinity, either $+\infty$ when $s = 0$, or $-\infty$ when $s = 1$. Infinity can represent results that *overflow*, as when we multiply two very large numbers, or when we divide by zero. When the fraction field is nonzero, the resulting value is called a "NaN," short for "Not a Number." Such values are returned as the result of an operation where the result cannot be given as a real number or as infinity, as when computing $\sqrt{-1}$ or $\infty - \infty$. They can also be useful in some applications for representing uninitialized data.

A. Complete Range



B. Values between -1.0 and $+1.0$.

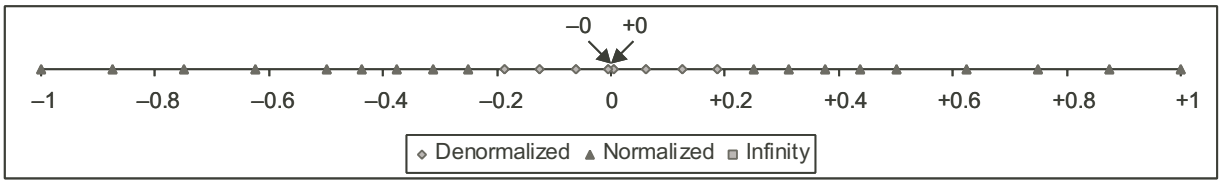


Figure 2.22: **Representable Values for 6-Bit Floating-Point Format.** There are $k = 3$ exponent bits and $n = 2$ significand bits. The bias is 3.

2.4.3 Example Numbers

Figure 2.22 shows the set of values that can be represented in a hypothetical 6-bit format having $k = 3$ exponent bits and $n = 2$ significand bits. The bias is $2^{3-1} - 1 = 3$. Part A of the figure shows all representable values (other than NaN). The two infinities are at the extreme ends. The normalized numbers with maximum magnitude are ± 14. The denormalized numbers are clustered around 0. These can be seen more clearly in part B of the figure, where we show just the numbers between -1.0 and $+1.0$. The two zeros are special cases of denormalized numbers. Observe that the representable numbers are not uniformly distributed—they are denser nearer the origin.

Figure 2.23 shows some examples for a hypothetical eight-bit floating-point format having $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is $2^{4-1} - 1 = 7$. The figure is divided into three regions representing the three classes of numbers. Closest to 0 are the denormalized numbers, starting with 0 itself. Denormalized numbers in this format have $E = 1 - 7 = -6$, giving a weight $2^E = \frac{1}{64}$. The fractions f range over the values $0, \frac{1}{8}, \ldots, \frac{7}{8}$, giving numbers V in the range 0 to $\frac{7}{8 \times 64} = \frac{7}{512}$.

The smallest normalized numbers in this format also have $E = 1 - 7 = -6$, and the fractions also range over the values $0, \frac{1}{8}, \ldots \frac{7}{8}$. However, the significands then range from $1 + 0 = 1$ to $1 + \frac{7}{8} = \frac{15}{8}$, giving numbers V in the range $\frac{8}{512}$ to $\frac{15}{512}$.

Observe the smooth transition between the largest denormalized number $\frac{7}{512}$ and the smallest normalized number $\frac{8}{512}$. This smoothness is due to our definition of E for denormalized values. By making it $1 - Bias$ rather than $-Bias$, we compensate for the fact that the significand of a denormalized number does not have an implied leading 1.

As we increase the exponent, we get successively larger normalized values, passing through 1.0 and then to the largest normalized number. This number has exponent $E = 7$, giving a weight $2^E = 128$. The fraction equals $\frac{7}{8}$ giving a significand $M = \frac{15}{8}$. Thus the numeric value is $V = 240$. Going beyond this overflows to $+\infty$.

Description	Bit Rep.	e	E	f	M	V
Zero	`0 0000 000`	0	-6	0	0	0
Smallest Pos.	`0 0000 001`	0	-6	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$
	`0 0000 010`	0	-6	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$
	`0 0000 011`	0	-6	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$
	...					
	`0 0000 110`	0	-6	$\frac{6}{8}$	$\frac{6}{8}$	$\frac{6}{512}$
Largest Denorm.	`0 0000 111`	0	-6	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$
Smallest Norm.	`0 0001 000`	1	-6	0	$\frac{8}{8}$	$\frac{8}{512}$
	`0 0001 001`	1	-6	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$
	...					
	`0 0110 110`	6	-1	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$
	`0 0110 111`	6	-1	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$
One	`0 0111 000`	7	0	0	$\frac{8}{8}$	1
	`0 0111 001`	7	0	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	`0 0111 010`	7	0	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$
	...					
	`0 1110 110`	14	7	$\frac{6}{8}$	$\frac{14}{8}$	224
Largest Norm.	`0 1110 111`	14	7	$\frac{7}{8}$	$\frac{15}{8}$	240
Infinity	`0 1111 000`	–	–	–	–	$+\infty$

Figure 2.23: **Example Nonnegative Values for eight-bit Floating-Point Format.** There are $k = 4$ exponent bits and $n = 3$ significand bits. The bias is 7.

One interesting property of this representation is that if we interpret the bit representations of the values in Figure 2.23 as unsigned integers, they occur in ascending order, as do the values they represent as floating-point numbers. This is no accident—the IEEE format was designed so that floating-point numbers could be sorted using an integer-sorting routine. A minor difficulty is in dealing with negative numbers, since they have a leading one, and they occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons (see Problem 2.47).

Practice Problem 2.25:

Consider a 5-bit floating-point representation based on the IEEE floating-point format, with one sign bit, two exponent bits ($k = 2$), and two fraction bits ($n = 2$). The exponent bias is $2^{2-1} - 1 = 1$.

The table below enumerates the entire nonnegative range for this 5-bit floating-point representation. Fill in the blank table entries using the following directions:

e: The value represented by considering the exponent field to be an unsigned integer.

E: The value of the exponent after biasing.

f: The value of the fraction.

M: The value of the significand.

V: The numeric value represented.

Express the values of f, M and V as fractions of the form $\frac{x}{4}$. You need not fill in entries marked "—".

Bits	e	E	f	M	V
0 00 00					
0 00 01					
0 00 10					
0 00 11					
0 01 00					
0 01 01					
0 01 10					
0 01 11					
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01					
0 10 10					
0 10 11					
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	NaN
0 11 10	—	—	—	—	NaN
0 11 11	—	—	—	—	NaN

Figure 2.24 shows the representations and numeric values of some important single and double-precision floating-point numbers. As with the eight-bit format shown in Figure 2.23 we can see some general properties for a floating-point representation with a k-bit exponent and an n-bit fraction:

Description	exp	frac	Single Precision		Double Precision	
			Value	Decimal	Value	Decimal
Zero	$00\cdots00$	$0\cdots00$	0	0.0	0	0.0
Smallest denorm.	$00\cdots00$	$0\cdots01$	$2^{-23}\times 2^{-126}$	1.4×10^{-45}	$2^{-52}\times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00\cdots00$	$1\cdots11$	$(1-\epsilon)\times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon)\times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	$00\cdots01$	$0\cdots00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01\cdots11$	$0\cdots00$	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	$11\cdots10$	$1\cdots11$	$(2-\epsilon)\times 2^{127}$	3.4×10^{38}	$(2-\epsilon)\times 2^{1023}$	1.8×10^{308}

Figure 2.24: **Examples of Nonnegative Floating-Point Numbers.**

- The value $+0.0$ always has a bit representation of all 0's.
- The smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all 0s. It has a fraction (and significand) value $M = f = 2^{-n}$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-n-2^{k-1}+2}$.
- The largest denormalized value has a bit representation consisting of an exponent field of all 0s and a fraction field of all 1s. It has a fraction (and significand) value $M = f = 1 - 2^{-n}$ (which we have written $1 - \epsilon$) and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$, which is just slightly smaller than the smallest normalized value.
- The smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all 0s. It has a significand value $M = 1$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-2^{k-1}+2}$.
- The value 1.0 has a bit representation with all but the most significant bit of the exponent field equal to 1 and all other bits equal to 0. Its significand value is $M = 1$ and its exponent value is $E = 0$.
- The largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1. It has a fraction value of $f = 1 - 2^{-n}$, giving a significand $M = 2 - 2^{-n}$ (which we have written $2 - \epsilon$). It has an exponent value $E = 2^{k-1} - 1$, giving a numeric value $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

  Practice Problem 2.26:

  A. For a floating-point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an $n + 1$-bit fraction to be exact).

  B. What is the numeric value of this integer for single-precision format ($k = 8$, $n = 23$)?

2.4.4 Rounding

Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value x, we generally want a systematic method of finding the "closest" matching

Mode	$1.40	$1.60	$1.50	$2.50	$–1.50
Round-to-even	$1	$2	$2	$2	$–2
Round-toward-zero	$1	$1	$1	$2	$–1
Round-down	$1	$1	$1	$2	$–2
Round-up	$2	$2	$2	$3	$–1

Figure 2.25: **Illustration of Rounding Modes for Dollar Rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

value x' that can be represented in the desired floating-point format. This is the task of the *rounding* operation. The key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have $1.50 and want to round it to the nearest dollar, should the result be $1 or $2? An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values x^- and x^+ such that the value x is guaranteed to lie between them: $x^- \leq x \leq x^+$. The IEEE floating-point format defines four different *rounding modes*. The default method finds a closest match, while the other three can be used for computing upper and lower bounds.

Figure 2.25 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match. Thus, it rounds $1.40 to $1 and $1.60 to $2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-even mode adopts the convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both $1.50 and $2.50 to $2.

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications. Round-toward-zero mode rounds positive numbers downward and negative numbers upward, giving a value $\hat{x}$ such that $|\hat{x}| \leq |x|$. Round-down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since four is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern of the form $XX\cdots X.YY\cdots Y100\cdots$, where X and Y denote arbitary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point). We would round 10.00011_2 ($2\frac{3}{32}$) down to 10.00_2 (2), and 10.00110_2 ($2\frac{3}{16}$) up to 10.01_2 ($2\frac{1}{4}$), because these values are not halfway between two possible values. We would round 10.11100_2 ($2\frac{7}{8}$) up to 11.00_2 (3) and 10.10100_2 down to 10.10_2 ($2\frac{1}{2}$), since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

2.4.5 Floating-Point Operations

The IEEE standard specifies a simple rule for determining the result of an arithmetic operation such as addition or multiplication. Viewing floating-point values x and y as real numbers, and some operation $\odot$ defined over real numbers, the computation should yield $Round(x \odot y)$, the result of applying rounding to the exact result of the real operation. In practice, there are clever tricks floating-point unit designers use to avoid performing this exact computation, since the computation need only be sufficiently precise to guarantee a correctly rounded result. When one of the arguments is a special value such as -0, ∞ or NaN, the standard specifies conventions that attempt to be reasonable. For example $1/-0$ is defined to yield $-\infty$, while $1/+0$ is defined to yield $+\infty$.

One strength of the IEEE standard's method of specifying the behavior of floating-point operations is that it is independent of any particular hardware or software realization. Thus, we can examine its abstract mathematical properties without considering how it is actually implemented.

We saw earlier that integer addition, both unsigned and two's complement, forms an Abelian group. Addition over real numbers also forms an Abelian group, but we must consider what effect rounding has on these properties. Let us define $x +^f y$ to be $Round(x + y)$. This operation is defined for all values of x and y, although it may yield infinity even when both x and y are real numbers due to overflow. The operation is commutative, with $x +^f y = y +^f x$ for all values of x and y. On the other hand, the operation is not associative. For example, with single-precision floating point the expression `(3.14+1e10)-1e10` would evaluate to `0.0`—the value 3.14 would be lost due to rounding. On the other hand, the expression `3.14+(1e10-1e10)` would evaluate to `3.14`. As with an Abelian group, most values have inverses under floating-point addition, that is, $x +^f -x = 0$. The exceptions are infinities (since $+\infty - \infty = NaN$), and NaN's, since $NaN +^f x = NaN$ for any x.

The lack of associativity in floating-point addition is the most important group property that is lacking. It has important implications for scientific programmers and compiler writers. For example, suppose a compiler is given the following code fragment:

```
x = a + b + c;
y = b + c + d;
```

The compiler might be tempted to save one floating-point addition by generating the code:

```
t = b + c;
```

```
x = a + t;
y = t + d;
```

However, this computation might yield a different value for `x` than would the original, since it uses a different association of the addition operations. In most applications, the difference would be so small as to be inconsequential. Unfortunately, compilers have no way of knowing what trade-offs the user is willing to make between efficiency and faithfulness to the exact behavior of the original program. As a result, they tend to be very conservative, avoiding any optimizations that could have even the slightest effect on functionality.

On the other hand, floating-point addition satisfies the following monotonicity property: if $a \geq b$ then $x + a \geq x + b$ for any values of a, b, and x other than NaN. This property of real (and integer) addition is not obeyed by unsigned or two's complement addition.

Floating-point multiplication also obeys many of the properties one normally associates with multiplication, namely those of a ring. Let us define $x \ {*^{\mathrm{f}}} \ y$ to be $Round(x \times y)$. This operation is closed under multiplication (although possibly yielding infinity or NaN), it is commutative, and it has 1.0 as a multiplicative identity. On the other hand, it is not associative due to the possibility of overflow or the loss of precision due to rounding. For example, with single-precision floating point, the expression `(1e20*1e20)*1e-20` will evaluate to $+\infty$, while `1e20*(1e20*1e-20)` will evaluate to `1e20`. In addition, floating-point multiplication does not distribute over addition. For example, with single-precision floating point, the expression `1e20*(1e20-1e20)` will evaluate to `0.0`, while `1e20*1e20-1e20*1e20` will evaluate to `NaN`.

On the other hand, floating-point multiplication satisfies the following monotonicity properties for any values of a, b, and c other than NaN:

$$a \geq b \ \text{ and } \ c \geq 0 \quad \Rightarrow \quad a \ {*^{\mathrm{f}}} \ c \geq b \ {*^{\mathrm{f}}} \ c$$
$$a \geq b \ \text{ and } \ c \leq 0 \quad \Rightarrow \quad a \ {*^{\mathrm{f}}} \ c \leq b \ {*^{\mathrm{f}}} \ c$$

In addition, we are also guaranteed that $a \ {*^{\mathrm{f}}} \ a \geq 0$, as long as $a \neq NaN$. As we saw earlier, none of these monotonicity properties hold for unsigned or two's complement multiplication.

This lack of associativity and distributivity is of serious concern to scientific programmers and to compiler writers. Even such a seemingly simple task as writing code to determine whether two lines intersect in three-dimensional space can be a major challenge.

2.4.6 Floating Point in C

C provides two different floating-point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single and double-precision floating point. In addition, the machines use the round-to-even rounding mode. Unfortunately, since the C standard does require the machine use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as -0, $+\infty$, $-\infty$, or NaN. Most systems provide a combination of include ('`.h`') files and procedure libraries to provide access to these features, but the details vary from one system to another. For example, the GNU compiler GCC defines macros `INFINITY` (for $+\infty$) and `NAN` (for NaN) when the following sequence occurs in the program file:

```
#define _GNU_SOURCE 1
```

```
#include <math.h>
```

Practice Problem 2.27:

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0.

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
#endif
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

When casting values between `int`, `float`, and `double` formats, the program changes the numeric values and the bit representations as follows (assuming a 32-bit `int`):

- From `int` to `float`, the number cannot overflow, but it may be rounded.
- From `int` or `float` to `double`, the exact numeric value can be preserved because `double` has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From `double` to `float`, the value can overflow to $+\infty$ or $-\infty$, since the range is smaller. Otherwise it may be rounded since the precision is smaller.
- From `float` or `double` to `int` the value will be truncated toward zero. For example 1.999 will be converted to 1, while -1.999 will be converted to -1. Note that this behavior is very different from rounding. Furthermore, the value may overflow. The C standard does not specify a fixed result for this case, but on most machines the result will either be $TMax_w$ or $TMin_w$, where w is the number of bits in an `int`.

Aside: Ariane 5: the high cost of floating-point overflow

Converting large floating-point numbers to integers is a common source of programming errors. Such an error had particularly disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after lift-off, the rocket veered off its flight path, broke up, and exploded. On board the rocket were communication satellites, valued at $500 million.

A later investigation [46] showed that the computer controlling the inertial navigation system had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that, in an effort to convert a 64-bit floating point number into a 16-bit signed integer, an overflow had been encountered.

The value that overflowed measured the horizontal velocity of the rocket, which could be more than five times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based. **End Aside.**

Practice Problem 2.28:

Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or NaN. For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(float) x`
B. `x == (int)(double) x`
C. `f == (float)(double) f`
D. `d == (float) d`
E. `f == -(-f)`
F. `2/3 == 2/3.0`
G. `(d >= 0.0) || ((d*2) < 0.0)`
H. `(d+f)-d == f`

2.5 Summary

Computers encode information as bits, generally organized as sequences of bytes. Different encodings are used for representing integers, real numbers, and character strings. Different models of computers use different conventions for encoding numbers and for ordering the bytes within multibyte data.

The C language is designed to accomodate a wide range of different implementations in terms of word sizes and numeric encodings. Most current machines have 32-bit word sizes, although high-end machines increasingly have 64-bit words. Most machines use two's complement encoding of integers and IEEE encoding of floating point. Understanding these encodings at the bit level, and the mathematical characteristics of the arithmetic operations is important for writing programs that operate correctly over the full range of numeric values.

The C standard dictates that when casting between signed and unsigned integers, the underlying bit pattern should not change. On a two's complement machine, this behavior is characterized by functions $T2U_w$ and $U2T_w$, for a w-bit value. The implicit casting of C gives results that many programmers do not anticipate, often leading to program bugs.

Due to the finite lengths of the encodings, computer arithmetic has properties quite different from conventional integer and real arithmetic. The finite length can cause numbers to overflow, when they exceed the range of the representation. Floating point values can also underflow, when they are so close to 0.0 that they are changed to zero.

The finite integer arithmetic implemented by C, as well as most other programming languages, has some peculiar properties compared to true integer arithmetic. For example, the expression `x*x` can evaluate to a negative number due to overflow. Nonetheless, both unsigned and two's complement arithmetic satisfies the properties of a ring. This allows compilers to do many optimizations. For example, in replacing the expression `7*x` by `(x<<3)-x`, we make use of the associative, commutative and distributive properties, along with the relationship between shifting and multiplying by powers of two.

We have seen several clever ways to exploit combinations bit-level operations and arithmetic operations. For example, we saw that with two's complement arithmetic, `~x+1` is equivalent to `-x`. As another example, suppose we want a bit pattern of the form $[0, \ldots, 0, 1, \ldots, 1]$, consisting of $w - k$ 0s followed by k 1s. Such bit patterns are useful for masking operations. This pattern can be generated by the C expression `(1<<k)-1`, exploiting the property that the desired bit pattern has numeric value $2^k - 1$. For example, the expression `(1<<8)-1` will generate the bit pattern `0xFF`.

Floating point representations approximate real numbers by encoding numbers of the form $x \times 2^y$. The most common floating point representation was defined by IEEE Standard 754. It provides for several different precisions, with the most common being single (32 bits) and double (64 bits). IEEE floating point also has representations for special values ∞ and not-a-number.

Floating point arithmetic must be used very carefully, since it has only limited range and precision, and since it does not obey common mathematical properties such as associativity.

Bibliographic Notes

Reference books on C [37, 30] discuss properties of the different data types and operations. The C standard does not specify details such as precise word sizes or numeric encodings. Such details are intentionally omitted to make it possible to implement C on a wide range of different machines. Several books have been written giving advice to C programmers [38, 47] that warn about problems with overflow, implicit casting to unsigned, and some of the other pitfalls we have covered in this chapter. These books also provide helpful advice on variable naming, coding styles, and code testing. Books on Java (we recommend the one coauthored by James Gosling, the creator of the language [1]) describe the data formats and arithmetic operations supported by Java.

Most books on logic design [82, 36] have a section on encodings and arithmetic operations. Such books describe different ways of implementing arithmetic circuits. Appendix A of Hennessy and Patterson's computer architecture textbook [31] does a particularly good job of describing different encodings (including IEEE floating point) as well as different implementation techniques.

Overton's book on IEEE floating point [53] provides a detailed description of the format as well as the properties from the perspective of a numerical applications programmer.

Homework Problems

Homework Problem 2.29 [Category 1]:

Compile and run the sample code that uses `show_bytes` (file `show-bytes.c`) on different machines to which you have access. Determine the byte orderings used by these machines.

Homework Problem 2.30 [Category 1]:

Try running the code for `show_bytes` for different sample values.

Homework Problem 2.31 [Category 1]:

Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of

C objects of types `short int`, `long int`, and `double` respectively. Try these out on several machines.

Homework Problem 2.32 [Category 2]:

Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word size.

Homework Problem 2.33 [Category 2]:

Write a C expression that will yield a word consisting of the least significant byte of `x`, and the remaining bytes of `y`. For operands `x` = `0x89ABCDEF` and `y` = `0x76543210`, this would give `0x765432EF`.

Homework Problem 2.34 [Category 2]:

Using only bit-level and logical operations, write C expressions that yield 1 for the described condition and 0 otherwise. Your code should work on a machine with any word size. Assume `x` is an integer.

A. Any bit of `x` equals 1.

B. Any bit of `x` equals 0.

C. Any bit in the least significant byte of `x` equals 1.

D. Any bit in the least significant byte of `x` equals 0.

Homework Problem 2.35 [Category 3]:

Write a procedure `int_shifts_are_arithmetic()` that yields 1 when run a machine that uses arithmetic right shifts for `int`'s and 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines. Write and test a procedure `unsigned_shifts_are_arithmetic()` that determines the form of shifts used for `unsigned int`'s.

Homework Problem 2.36 [Category 2]:

You are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. Here is a first attempt:

```
1 /* The following code does not run properly on some machines */
2 int bad_int_size_is_32()
3 {
4     /* Set most significant bit (msb) of 32-bit machine */
5     int set_msb = 1 << 31;
6     /* Shift past msb of 32-bit word */
7     int beyond_msb = 1 << 32;
8
9     /* set_msb is nonzero when word size >= 32
10       beyond_msb is zero when word size <= 32  */
11    return set_msb && !beyond_msb;
12 }
```

When compiled and run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

A. In what way does our code fail to comply with the C standard?

B. Modify the code to run properly on any machine for which `int`'s are at least 32 bits.

C. Modify the code to run properly on any machine for which `int`'s are at least 16 bits.

Homework Problem 2.37 [Category 1]:

You just started working for a company that is implementing a set of procedures to operate on a data structure where four signed bytes are packed into a 32-bit `unsigned`. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two's complement arithmetic and arithmetic right shifts with the following prototype:

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word.  Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

That is, the function will extract the designated byte and sign extend it to be a 32-bit `int`.

Your predecessor (who was fired for his incompetence) wrote the following code:

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
  return
    (word >> (bytenum << 3)) & 0xFF;
}
```

A. What is wrong with this code?

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

Homework Problem 2.38 [Category 1]:

Fill in the following table showing the effects of complementing and incrementing several 5-bit vectors, in the style of Figure 2.20. Show both the bit vectors and the numeric values.

$\vec{x}$	$\tilde{}\vec{x}$	$incr(\tilde{}\vec{x})$
$[01101]$		
$[01111]$		
$[11000]$		
$[11111]$		
$[10000]$		

Homework Problem 2.39 [Category 2]:

Show that first decrementing and then complementing is equivalent to complementing and then incrementing. That is, for any signed value `x`, the C expressions `-x`, `~x+1`, and `~(x-1)` yield identical results. What mathematical properties of two's complement addition does your derivation rely on?

Homework Problem 2.40 [Category 3]:

Suppose we want to compute the complete $2w$-bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type `unsigned` is w bits. The low-order w bits of the product can be computed with the expression `x*y`, so we only require a procedure with prototype

```
unsigned int unsigned_high_prod(unsigned x, unsigned y);
```

that computes the high-order w bits of $x \cdot y$ for unsigned variables.

We have access to a library function with prototype:

```
int signed_high_prod(int x, int y);
```

that computes the high-order w bits of $x \cdot y$ for the case where x and y are in two's complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.

[Hint:] Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

Homework Problem 2.41 [Category 2]:

Suppose we are given the task of generating code to multiply integer variable `x` by various different constant factors K. To be efficient we want to use only the operations `+`, `-`, and `<<`. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.

A. $K = 5$:

B. $K = 9$:

C. $K = 14$:

D. $K = -56$:

Homework Problem 2.42 [Category 2]:

Write C expressions to generate the following bit patterns, where a^k represents k repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters `j` and `k`, representing the values of j and k, but not a parameter representing w.

A. $1^{w-k}0^k$.

B. $0^{w-k-j}1^k0^j$.

Homework Problem 2.43 [Category 2]:

Suppose we number the bytes in a w-bit word from 0 (least significant) to $w/8-1$ (most significant). Write code for the following C function, that will return an unsigned value in which byte `i` of argument `x` has been replaced by byte `b`.

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Here are some examples showing how the function should work

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

Homework Problem 2.44 [Category 3]:

Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may assume that `int`'s are 32-bits long. The shift amount `k` can range from 0 to 31.

```
unsigned srl(unsigned x, int k)
{
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;

    /* ... */

}

int sra(int x, int k)
{
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;
```

```
    /* ... */
}
```

Homework Problem 2.45 [Category 2]:

Assume we are running code on a 32-bit machine using two's complement arithmetic for signed variables. The variables are declared and initialized as follows:

```
int x = foo();   /* Arbitrary value */
int y = bar();   /* Arbitrary value */

unsigned ux = x;
unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (i.e., evaluates to 1) for all values of `x` and `y`, or (2) give example values of `x` and `y` for which it is false (i.e., evaluates to 0.)

A. `(x >= 0) || ((2*x) < 0)`

B. `(x & 7) != 7 || (x<<30 < 0)`

C. `(x * x) >= 0`

D. `x < 0 || -x <= 0`

E. `x > 0 || -x >= 0`

F. `x*y == ux*uy`

G. `~x*y + uy*ux == -y`

Homework Problem 2.46 [Category 2]:

Consider numbers having a binary representation consisting of an infinite string of the form $0.y\,y\,y\,y\,y\,y\cdots$, where y is a k-bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101\cdots$ ($y = 01$), while the representation of $\frac{1}{5}$ is $0.001100110011\cdots$ ($y = 0011$).

A. Let $Y = B2U_k(y)$, that is, the number having binary representation y. Give a formula in terms of Y and k for the value represented by the infinite string. [Hint: Consider the effect of shifting the binary point k positions to the right.]

B. What is the numeric value of the string for the following values of y?

   (a) 001

(b) 1001

(c) 000111

Homework Problem 2.47 [Category 1]:

Fill in the return value for the following procedure that tests whether its first argument is greater than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is NaN. The two flavors of zero: $+0$ and -0 are considered equal.

```
int float_ge(float x, float y)
{
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return /* ... */ ;
}
```

Homework Problem 2.48 [Category 1]:

Given a floating-point format with a k-bit exponent and an n-bit fraction, give formulas for the exponent E, significand M, the fraction f, and the value V for the following quantities. In addition, describe the bit representation.

A. The number 5.0.

B. The largest odd integer that can be represented exactly.

C. The reciprocal of the smallest positive normalized value.

Homework Problem 2.49 [Category 1]:

Intel-compatible processors also support an "extended precision" floating-point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single *integer* bit, and $n = 63$ fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

Description	Extended Precision	
	Value	Decimal
Smallest denormalized		
Smallest normalized		
Largest normalized		

Homework Problem 2.50 [Category 1]:

Consider a 16-bit floating-point representation based on the IEEE floating-point format, with one sign bit, seven exponent bits ($k = 7$), and eight fraction bits ($n = 8$). The exponent bias is $2^{7-1} - 1 = 63$.

Fill in the table below for the following numbers, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include: 0, $\frac{67}{64}$, and $\frac{1}{256}$.

E: The integer value of the exponent.

V: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $\frac{7}{2}$, we would have $s = 0$, $M = \frac{7}{4}$, and $E = 1$. Our number would therefore have an exponent field of `0x40` (decimal value $63 + 1 = 64$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `40C0`.

You need not fill in entries marked "—".

Description	Hex	M	E	V
-0				—
Smallest value > 1				
256				—-
Largest Denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

Homework Problem 2.51 [Category 1]:

You have been assigned the task of writing a C function to compute a floating-point representation of 2^x. You realize that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the following code to compute the correct result. Assume the function `u2f` returns a floating-point value having an identical bit representation as its unsigned argument.

```
float fpwr2(int x)


    /* Result exponent and significand */
    unsigned exp, sig;
    unsigned u;

    if (x < ______)
        /* Too small.  Return 0.0 */
        exp = _____________;
        sig = _____________;
     else if (x < ______)
        /* Denormalized result */
        exp = _____________;
        sig = _____________;
     else if (x < ______)
        /* Normalized result. */
        exp = _____________;
        sig = _____________;
     else
        /* Too big.  Return +oo */
        exp = _____________;
        sig = _____________;


    /* Pack exp and sig into 32 bits */
    u = exp << 23 | sig;
    /* Return as float */
    return u2f(u);
```

Homework Problem 2.52 [Category 1]:

Around 250 B.C., the Greek mathematician Archimedes proved that $\frac{223}{71} < \pi < \frac{22}{7}$. Had he had access to a computer and the standard library `<math.h>`, he would have been able to determine that the single-precision floating-point approximation of π has the hexadecimal representation `0x40490FDB`. Of course, all of these are just approximations, since π is not rational.

A. What is the fractional binary number denoted by this floating-point value?

B. What is the fractional binary representation of $\frac{22}{7}$? [Hint: See Problem 2.46].

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

Chapter 3

Machine-Level Representation of C Programs

When programming in a high-level language, such as C, we are shielded from the detailed, machine-level implementation of our program. In contrast, when writing programs in assembly code, a programmer must specify exactly how the program manages memory and the low-level instructions the program uses to carry out the computation. Most of the time, it is much more productive and reliable to work at the higher level of abstraction provided by a high-level language. The type checking provided by a compiler helps detect many program errors and makes sure we reference and manipulate data in consistent ways. With modern, optimizing compilers, the generated code is usually at least as efficient as what a skilled, assembly-language programmer would write by hand. Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.

Even though optimizing compilers are available, being able to read and understand assembly code is an important skill for serious programmers. By invoking the compiler with appropriate flags, the compiler will generate a file showing its output in assembly code. Assembly code is very close to the actual machine code that computers execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter 5, programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the run-time behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package, as covered in Chapter 11, it is important to know what type of storage is used to hold the different program variables. This information is visible at the assembly code level. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by optimizing compilers.

In this chapter, we will learn the details of a particular assembly language and see how C programs get compiled into this form of machine code. Reading the assembly code generated by a compiler involves a different set of skills than writing assembly code by hand. We must understand the transformations typical

compilers make in converting the constructs of C into machine code. Relative to the computations expressed in the C code, optimizing compilers can rearrange execution order, eliminate unneeded computations, replace slow operations such as multiplication by shifts and adds, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can often be a challenge—much like putting together a puzzle having a slightly different design than the picture on the box. It is a form of *reverse engineering*—trying to understand the process by which a system was created by studying the system and working backward. In this case, the system is a machine-generated, assembly-language program, rather than something designed by a human. This simplifies the task of reverse engineering, because the generated code follows fairly regular patterns, and we can run experiments, having the compiler generate code for many different programs. In our presentation, we give many examples and provide a number of exercises illustrating different aspects of assembly language and compilers. This is a subject matter where mastering the details is a prerequisite to understanding the deeper and more fundamental concepts. Spending time studying the examples and working through the exercises will be well worthwhile.

We give a brief history of the Intel architecture. Intel processors have grown from rather primitive 16-bit processors in 1978 to the mainstream machines for today's desktop computers. The architecture has grown correspondingly with new features added and the 16-bit architecture transformed to support 32-bit data and addresses. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by GCC and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

Our technical presentation starts a quick tour to show the relation between C, assembly code, and object code. We then proceed to the details of IA32, starting with the representation and manipulation of data and the implementation of control. We see how control constructs in C, such as `if`, `while`, and `switch` statements, are implemented. We then cover the implementation of procedures, including how the run-time stack supports the passing of data and control between procedures, as well as storage for local variables. Next, we consider how data structures such as arrays, structures, and unions are implemented at the machine level. With this background in machine-level programming, we can examine the problems of out of bounds memory references and the vulnerability of systems to buffer overflow attacks. We finish this part of the presentation with some tips on using the GDB debugger for examining the runtime behavior of a machine-level program.

We then move into material that is marked with a "*" and is intended for the truly dedicated machine-language enthusiasts. We give a presentation of IA32 support for floating-point code. This is a particularly arcane feature of IA32, and so we advise that only people determined to work with floating-point code attempt to study this section. We give a brief presentation of GCC's support for embedding assembly code within C programs. In some applications, the programmer must drop down to assembly code to access low-level features of the machine. Embedded assembly is the best way to do this.

3.1 A Historical Perspective

The Intel processor line has a long, evolutionary development. It started with one of the first single-chip, 16-bit microprocessors, where many compromises had to be made due to the limited capabilities of integrated

circuit technology at the time. Since then it has grown to take advantage of technology improvements as well as to satisfy the demands for higher performance and for supporting more advanced operating systems.

The following list shows the successive models of Intel processors, and some of their key features. We use the number of transistors required to implement the processors as an indication of how they have evolved in complexity ('K' denotes 1,000, and 'M' denotes 1,000,000).

8086: (1978, 29 K transistors). One of the first single-chip, 16-bit microprocessors. The 8088, a version of the 8086 with an 8-bit external bus, formed the heart of the original IBM personal computers. IBM contracted with then-tiny Microsoft to develop the MS-DOS operating system. The original models came with 32,768 bytes of memory and two floppy drives (no hard drive). Architecturally, the machines were limited to a 655,360-byte address space—addresses were only 20 bits long (1,048,576 bytes addressable), and the operating system reserved 393,216 bytes for its own use.

80286: (1982, 134 K transistors). Added more (and now obsolete) addressing modes. Formed the basis of the IBM PC-AT personal computer, the original platform for MS Windows.

i386: (1985, 275 K transistors). Expanded the architecture to 32 bits. Added the flat addressing model used by Linux and recent versions of the Windows family of operating system. This was the first machine in the series that could support a Unix operating system.

i486: (1989, 1.9 M transistors). Improved performance and integrated the floating-point unit onto the processor chip but did not change the instruction set.

Pentium: (1993, 3.1 M transistors). Improved performance, but only added minor extensions to the instruction set.

PentiumPro: (1995, 6.5 M transistors). Introduced a radically new processor design, internally known as the *P6* microarchitecture. Added a class of "conditional move" instructions to the instruction set.

Pentium/MMX: (1997, 4.5 M transistors). Added new class of instructions to the Pentium processor for manipulating vectors of integers. Each datum can be 1, 2, or 4-bytes long. Each vector totals 64 bits.

Pentium II: (1997, 7 M transistors). Merged the previously separate PentiumPro and Pentium/MMX lines by implementing the MMX instructions within the P6 microarchitecture.

Pentium III: (1999, 8.2 M transistors). Introduced yet another class of instructions for manipulating vectors of integer or floating-point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits. Later versions of this chip went up to 24 M transistors, due to the incorporation of the level-2 cache on chip.

Pentium 4: (2001, 42 M transistors). Added 8-byte integer and floating-point formats to the vector instructions, along with 144 new instructions for these formats. Intel shifted away from Roman numerals in their numbering convention.

Each successive processor has been designed to be backward compatible—able to run code compiled for any earlier version. As we will see, there are many strange artifacts in the instruction set due to this evolutionary heritage. Intel now calls its instruction set *IA32*, for "Intel Architecture 32-bit." The processor line is also referred to by the colloquial name "x86," reflecting the processor naming conventions up through the i486.

Aside: Why not the i586?
Intel discontinued their numeric naming convention, because they were not able to obtain trademark protection for their CPU numbers. The U. S. Trademark office does not allow numbers to be trademarked. Instead, they coined the name "Pentium" using the the Greek root word *penta* as an indication that this was their fifth generation machine. Since then, they have used variants of this name, even though the PentiumPro is a sixth generation machine (hence the internal name P6), and the Pentium 4 is a seventh generation machine. Each new generation involves a major change in the processor design. **End Aside.**

Over the years, several companies have produced processors that are compatible with Intel processors, capable of running the exact same machine-level programs. Chief among these is AMD. For years, AMD's strategy was to run just behind Intel in technology, producing processors that were less expensive although somewhat lower in performance. More recently, AMD has produced some of the highest performing processors for IA32. They were the first to the break the 1-gigahertz clock speed barrier for a commercially available microprocessor. Although we will talk about Intel processors, our presentation holds just as well for the compatible processors produced by Intel's rivals.

Much of the complexity of IA32 is not of concern to those interested in programs for the Linux operating system as generated by the GCC compiler. The memory model provided in the original 8086 and its extensions in the 80286 are obsolete. Instead, Linux uses what is referred to as *flat* addressing, where the entire memory space is viewed by the programmer as a large array of bytes.

As we can see in the list of developments, a number of formats and instructions have been added to IA32 for manipulating vectors of small integers and floating-point numbers. These features were added to allow improved performance on multimedia applications, such as image processing, audio and video encoding and decoding, and three-dimensional computer graphics. Unfortunately, current versions of GCC will not generate any code that uses these new features. In fact, in its default invocations GCC assumes it is generating code for an i386. The compiler makes no attempt to exploit the many extensions added to what is now considered a very old architecture.

3.2 Program Encodings

Suppose we write a C program as two files `p1.c` and `p2.c`. We would then compile this code using a Unix command line:

```
unix> gcc -O2 -o p p1.c p2.c
```

The command `gcc` indicates the GNU C compiler GCC. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The flag `-O2` instructs the compiler to apply level-two optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code. Level-two optimization is a good compromise between optimized performance and ease of use. All code in this book was compiled with this optimization level.

This command actually invokes a sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros. Second, the *compiler* generates assembly code versions of the two source files having names `p1.s` and `p2.s`. Next, the *assembler* converts the assembly code into binary object code files `p1.o`

and `p2.o`. Finally, the *linker* merges these two object files along with code implementing standard Unix library functions (e.g., `printf`) and generates the final executable file. Linking is described in more detail in Chapter 7.

3.2.1 Machine-Level Code

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes. The assembly code-representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of object code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

The assembly programmer's view of the machine differs significantly from that of a C programmer. Parts of the processor state are visible that are normally hidden from the C programmer:

- The program counter (called `%eip`) indicates the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing 32-bit values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the local variables of a procedure.
- The condition code registers hold status information about the most recently executed arithmetic instruction. These are used to implement conditional changes in the control flow, such as is required to implement `if` or `while` statements.
- The floating-point register file contains eight locations for storing floating-point data.

Whereas C provides a model where objects of different data types can be declared and allocated in memory, assembly code views the memory as simply a large, byte-addressable array. Aggregate data types in C such as arrays and structures are represented in assembly code as contiguous collections of bytes. Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.

The program memory contains the object code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user, (for example, by using the `malloc` library procedure).

The program memory is addressed using *virtual* addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the 32-bit addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

A single machine instruction performs only a very elementary operation. For example, it might add two numbers stored in registers, transfer data between memory and a register, or conditionally branch to a new

instruction address. The compiler must generate sequences of such instructions to implement program constructs such as arithmetic expression evaluation, loops, or procedure calls and returns.

3.2.2 Code Examples

Suppose we write a C code file `code.c` containing the following procedure definition:

```
1 int accum = 0;
2
3 int sum(int x, int y)
4 {
5     int t = x + y;
6     accum += t;
7     return t;
8 }
```

To see the assembly code generated by the C compiler, we can use the "`-S`" option on the command line:

```
unix> gcc -O2 -S code.c
```

This will cause the compiler to generate an assembly file `code.s` and go no further. (Normally it would then invoke the assembler to generate an object code file). The assembly-code file contains various declarations including the set of lines:

```
sum:
  pushl %ebp
  movl %esp,%ebp
  movl 12(%ebp),%eax
  addl 8(%ebp),%eax
  addl %eax,accum
  movl %ebp,%esp
  popl %ebp
  ret
```

Each indented line in the above code corresponds to a single machine instruction. For example, the `pushl` instruction indicates that the contents of register `%ebp` should be pushed onto the program stack. All information about local variable names or data types has been stripped away. We still see a reference to the global variable `accum`, since the compiler has not yet determined where in memory this variable will be stored.

If we use the '`-c`' command line option, GCC will both compile and assemble the code:

```
unix> gcc -O2 -c code.c
```

This will generate an object code file `code.o` that is in binary format and hence cannot be viewed directly. Embedded within the 852 bytes of the file `code.o` is a 19 byte sequence having hexadecimal representation:

```
55 89 e5 8b 45 0c 03 45 08 01 05 00 00 00 00 89 ec 5d c3
```

This is the object code corresponding to the assembly instructions listed above. A key lesson to learn from this is that the program actually executed by the machine is simply a sequence of bytes encoding a series of instructions. The machine has very little information about the source code from which these instructions were generated.

> **Aside: How do I find the byte representation of a program?**
> First we used a disassembler (to be described shortly) to determine that the code for `sum` is 19 bytes long. Then we ran the GNU debugging tool GDB on file `code.o` and gave it the command:
>
> ```
> (gdb)  x/19xb sum
> ```
>
> telling it to examine (abbreviated 'x') 19 hex-formatted (also abbreviated 'x') bytes (abbreviated 'b'). You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 3.12. **End Aside.**

To inspect the contents of object code files, a class of programs known as *disassemblers* can be invaluable. These programs generate a format similar to assembly code from the object code. With Linux systems, the program OBJDUMP (for "object dump") can serve this role given the '`-d`' command line flag:

```
unix> objdump -d code.o
```

The result is (where we have added line numbers on the left and annotations on the right):

```
     Disassembly of function sum in file code.o
 1 00000000 <sum>:
    Offset   Bytes                    Equivalent assembly language
 2    0:   55                        push   %ebp
 3    1:   89 e5                     mov    %esp,%ebp
 4    3:   8b 45 0c                  mov    0xc(%ebp),%eax
 5    6:   03 45 08                  add    0x8(%ebp),%eax
 6    9:   01 05 00 00 00 00         add    %eax,0x0
 7    f:   89 ec                     mov    %ebp,%esp
 8   11:   5d                        pop    %ebp
 9   12:   c3                        ret
10   13:   90                        nop
```

On the left we see the 19 hexadecimal byte values listed in the byte sequence earlier, partitioned into groups of 1 to 5 bytes each. Each of these groups is a single instruction, with the assembly language equivalent shown on the right. Several features are worth noting:

- IA32 instructions can range in length from 1 to 15 bytes. The instruction encoding is designed so that commonly used instructions and ones with fewer operands require a smaller number of bytes than do less common ones or ones with more operands.
- The instruction format is designed in such a way that from a given starting position, there is a unique decoding of the bytes into machine instructions. For example, only the instruction `pushl %ebp` can start with byte value `55`.

- The disassembler determines the assembly code based purely on the byte sequences in the object file. It does not require access to the source or assembly-code versions of the program.
- The disassembler uses a slightly different naming convention for the instructions than does GAS. In our example, it has omitted the suffix '`l`' from many of the instructions.
- Compared to the assembly code in `code.s` we also see an additional `nop` instruction at the end. This instruction will never be executed (it comes after the procedure return instruction), nor would it have any effect if it were (hence the name `nop`, short for "no operation" and commonly spoken as "no op"). The compiler inserted this instruction as a way to pad the space used to store the procedure.

Generating the actual executable code requires running a linker on the set of object code files, one of which must contain a function `main`. Suppose in file `main.c` we had the function:

```
1 int main()
2 {
3     return sum(1, 3);
4 }
```

Then we could generate an executable program `test` as follows:

```
unix> gcc -O2 -o prog code.o main.c
```

The file `prog` has grown to 11,667 bytes, since it contains not just the code for our two procedures but also information used to start and terminate the program as well as to interact with the operating system. We can also disassemble the file `prog`:

```
unix> objdump -d prog
```

The disassembler will extract various code sequences, including the following:

```
    Disassembly of function sum in executable file prog
1 080483b4 <sum>:
2  80483b4:  55                    push   %ebp
3  80483b5:  89 e5                 mov    %esp,%ebp
4  80483b7:  8b 45 0c              mov    0xc(%ebp),%eax
5  80483ba:  03 45 08              add    0x8(%ebp),%eax
6  80483bd:  01 05 64 94 04 08     add    %eax,0x8049464
7  80483c3:  89 ec                 mov    %ebp,%esp
8  80483c5:  5d                    pop    %ebp
9  80483c6:  c3                    ret
10 80483c7:  90                    nop
```

Note that this code is almost identical to that generated by the disassembly of `code.c`. One main difference is that the addresses listed along the left are different—the linker has shifted the location of this code to a different range of addresses. A second difference is that the linker has finally determined the location for storing global variable `accum`. On line 5 of the disassembly for `code.o` the address of `accum` was still listed as `0`. In the disassembly of `prog`, the address has been set to `0x8049444`. This is shown in the assembly code rendition of the instruction. It can also be seen in the last four bytes of the instruction, listed from least-significant to most as `44 94 04 08`.

3.2.3 A Note on Formatting

The assembly code generated by GCC is somewhat difficult to read. It contains some information with which we need not be concerned. On the other hand, it does not provide any description of the program or how it works. For example, suppose file `simple.c` contains the code:

```
1 int simple(int *xp, int y)
2 {
3   int t = *xp + y;
4   *xp = t;
5   return t;
6 }
```

when GCC is run with the '`-S`' flag it generates the following file for `simple.s`.

```
  .file   "simple.c"
  .version        "01.01"
gcc2_compiled.:
.text
  .align 4
.globl simple
  .type    simple,@function
simple:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  movl (%eax),%edx
  addl 12(%ebp),%edx
  movl %edx,(%eax)
  movl %edx,%eax
  movl %ebp,%esp
  popl %ebp
  ret
.Lfe1:
  .size    simple,.Lfe1-simple
  .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

The file contains more information than we really require. All of the lines beginning with '`.`' are directives to guide the assembler and linker. We can generally ignore these. On the other hand, there are no explanatory remarks about what the instructions do or how they relate to the source code.

To provide a clearer presentation of assembly code, we will show it in a form that includes line numbers and explanatory annotations. For our example, an annotated version would appear as follows:

```
1 simple:
2   pushl %ebp           Save frame pointer
3   movl %esp,%ebp       Create new frame pointer
4   movl 8(%ebp),%eax    Get xp
```

C declaration	Intel Data Type	GAS suffix	Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double Word	`l`	4
`unsigned`	Double Word	`l`	4
`long int`	Double Word	`l`	4
`unsigned long`	Double Word	`l`	4
`char *`	Double Word	`l`	4
`float`	Single Precision	`s`	4
`double`	Double Precision	`l`	8
`long double`	Extended Precision	`t`	10/12

Figure 3.1: Sizes of standard data types

```
 5   movl (%eax),%edx     Retrieve *xp
 6   addl 12(%ebp),%edx   Add y to get t
 7   movl %edx,(%eax)     Store t at *xp
 8   movl %edx,%eax       Set t as return value
 9   movl %ebp,%esp       Reset stack pointer
10   popl %ebp            Reset frame pointer
11   ret                  Return
```

We typically show only the lines of code relevant to the point being discussed. Each line is numbered on the left for reference and annotated on the right by a brief description of the effect of the instruction and how it relates to the computations of the original C code. This is a stylized version of the way assembly-language programmers format their code.

3.3 Data Formats

Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term "word" to refer to a 16-bit data type. Based on this, they refer to 32-bit quantities as "double words." They refer to 64-bit quantities as "quad words." Most instructions we will encounter operate on bytes or double words.

Figure 3.1 shows the machine representations used for the primitive data types of C. Note that most of the common data types are stored as double words. This includes both regular and long `int`'s, whether or not they are signed. In addition, all pointers (shown here as `char *`) are stored as 4-byte double words. Bytes are commonly used when manipulating string data. Floating-point numbers come in three different forms: single-precision (4-byte) values, corresponding to C data type `float`; double-precision (8-byte) values, corresponding to C data type `double`; and extended-precision (10-byte) values. GCC uses the data type `long double` to refer to extended-precision floating-point values. It also stores them as 12-byte quantities to improve memory system performance, as will be discussed later. Although the ANSI C standard includes `long double` as a data type, they are implemented for most combinations of compiler and machine using the same 8-byte format as ordinary `double`. The support for extended precision is

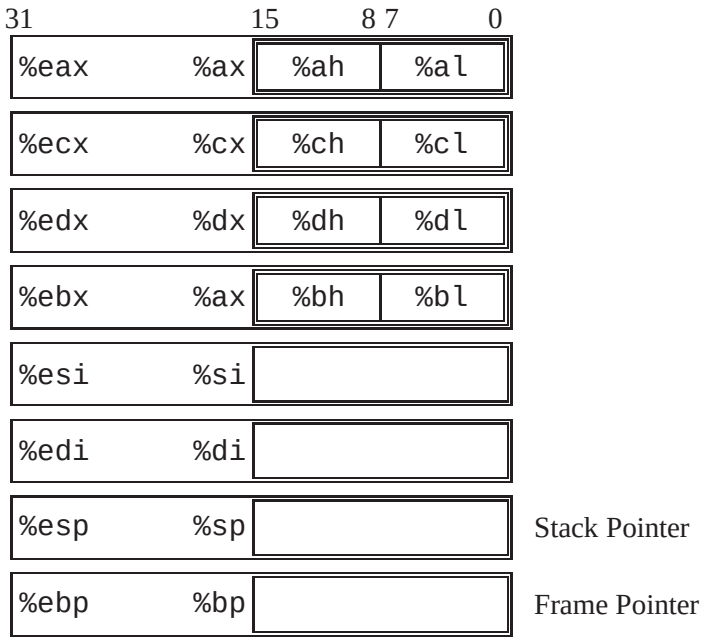


Figure 3.2: **Integer Registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The two low-order bytes of the first four registers can be accessed independently.

unique to the combination of GCC and IA32.

As the table indicates, every operation in GAS has a single-character suffix denoting the size of the operand. For example, the `mov` (move data) instruction has 3 variants: `movb` (move byte), `movw` (move word), and `movl` (move double word). The suffix '`l`' is used for double words, since on many machines 32-bit quantities are referred to as "long words," a holdover from an era when 16-bit word sizes were standard. Note that GAS uses the suffix '`l`' to denote both a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating point involves an entirely different set of instructions and registers.

3.4 Accessing Information

An IA32 central processing unit (CPU) contains a set of eight *registers* storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 3.2 diagrams the eight registers. Their names all begin with `%e`, but otherwise they have peculiar names. With the original 8086, the registers were 16-bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first 6 registers can be considered general-purpose registers with no restrictions placed on their use. We said "for the most part," because some instructions use fixed registers as sources and/or destinations. In addition, within procedures there are different conventions for saving and restoring the first three registers (`%eax`, `%ecx`, and `%edx`), than for the next three (`%ebx`, `%edi`, and `%esi`). This will be discussed in Section 3.7. The final two

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{E}_a$	$Reg[\mathtt{E}_a]$	Register
Memory	Imm	$Mem[Imm]$	Absolute
Memory	$(\mathtt{E}_a)$	$Mem[Reg[\mathtt{E}_a]]$	Indirect
Memory	$Imm(\mathtt{E}_b)$	$Mem[Imm + Reg[\mathtt{E}_b]]$	Base + Displacement
Memory	$(\mathtt{E}_b,\mathtt{E}_i)$	$Mem[Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i]]$	Indexed
Memory	$Imm(\mathtt{E}_b,\mathtt{E}_i)$	$Mem[Imm + Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i]]$	Indexed
Memory	$(,\mathtt{E}_i,s)$	$Mem[Reg[\mathtt{E}_i] \cdot s]$	Scaled Indexed
Memory	$Imm(,\mathtt{E}_i,s)$	$Mem[Imm + Reg[\mathtt{E}_i] \cdot s]$	Scaled Indexed
Memory	$(\mathtt{E}_b,\mathtt{E}_i,s)$	$Mem[Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i] \cdot s]$	Scaled Indexed
Memory	$Imm(\mathtt{E}_b,\mathtt{E}_i,s)$	$Mem[Imm + Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i] \cdot s]$	Scaled Indexed

Figure 3.3: **Operand Forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

registers (`%ebp` and `%esp`) contain pointers to important places in the program stack. They should only be altered according to the set of standard conventions for stack management.

As indicated in Figure 3.2, the low-order two bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining three bytes of the register do not change. Similarly, the low-order 16 bits of each register can be read or written by word operation instructions. This feature stems from IA32's evolutionary heritage as a 16-bit microprocessor.

3.4.1 Operand Specifiers

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. IA32 supports a number of operand forms (Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. With GAS, these are written with a '`$`' followed by an integer using standard C notation, such as, `$-577` or `$0x1F`. Any value that fits in a 32-bit word can be used, although the assembler will use one or two-byte encodings when possible. The second type, *register*, denotes the contents of one of the registers, either one of the eight 32-bit registers (e.g., `%eax`) for a double-word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In our figure, we use the notation $\mathtt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $Reg[\mathtt{E}_a]$, viewing the set of registers as an array Reg indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. As the table shows, there are many different *addressing modes* allowing different forms of memory references. The most general form is shown at the bottom of the table with syntax $Imm(\mathtt{E}_b,\mathtt{E}_i,s)$. Such a reference has four components: an immediate offset Imm, a base

Instruction		Effect	Description
`movl`	S, D	$D \leftarrow S$	Move Double Word
`movw`	S, D	$D \leftarrow S$	Move Word
`movb`	S, D	$D \leftarrow S$	Move Byte
`movsbl`	S, D	$D \leftarrow SignExtend(S)$	Move Sign-Extended Byte
`movzbl`	S, D	$D \leftarrow ZeroExtend(S)$	Move Zero-Extended Byte
`pushl`	S	$Reg[\texttt{\%esp}] \leftarrow Reg[\texttt{\%esp}] - 4$; $Mem[Reg[\texttt{\%esp}]] \leftarrow S$	Push
`popl`	D	$D \leftarrow Mem[Reg[\texttt{\%esp}]]$; $Reg[\texttt{\%esp}] \leftarrow Reg[\texttt{\%esp}] + 4$	Pop

Figure 3.4: **Data Movement Instructions.**

register $\texttt{E}_b$, an index register $\texttt{E}_i$, and a scale factor s, where s must be 1, 2, 4, or 8. The effective address is then computed as $Imm + Reg[\texttt{E}_b] + Reg[\texttt{E}_i] \cdot s$. This general form is often seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted. As we will see, the more complex addressing modes are useful when referencing array and structure elements.

Practice Problem 3.1:

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the values for the indicated operands

Operand	Value
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

3.4.2 Data Movement Instructions

Among the most heavily used instructions are those that perform data movement. The generality of the operand notation allows a simple move instruction to perform what in many machines would require a number of instructions. Figure 3.4 lists the important data movement instructions. The most common is the `movl` instruction for moving double words. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination.

The following are some examples of `movl` instructions showing the five possible combinations of source and destination types. Recall that the source operand comes first and the destination second.

```
1   movl $0x4050,%eax          Immediate--Register
2   movl %ebp,%esp             Register--Register
3   movl (%edi,%ecx),%eax      Memory--Register
4   movl $-17,(%esp)           Immediate--Memory
5   movl %eax,-12(%ebp)        Register--Memory
```

The `movb` instruction is similar, except that it moves just a single byte. When one of the operands is a register, it must be one of the eight single-byte register elements illustrated in Figure 3.2. Similarly, the `movw` instruction moves two bytes. When one of its operands is a register, it must be one of the eight two-byte register elements shown in Figure 3.2.

Both the `movsbl` and the `movzbl` instruction serve to copy a byte and to set the remaining bits in the destination. The `movsbl` instruction takes a single-byte source operand, performs a sign extension to 32 bits (i.e., it sets the high-order 24 bits to the most significant bit of the source byte), and copies this to a double-word destination. Similarly, the `movzbl` instruction takes a single-byte source operand, expands it to 32 bits by adding 24 leading zeros, and copies this to a double-word destination.

Aside: Comparing byte movement instructions.
Observe that the three byte movement instructions `movb`, `movsbl`, and `movzbl` differ from each other in subtle ways. Here is an example:

```
     Assume initially that %dh = 8D, %eax = 98765432
1   movb %dh,%al               %eax = 9876548D
2   movsbl %dh,%eax            %eax = FFFFFF8D
3   movzbl %dh,%eax            %eax = 0000008D
```

In these examples, all set the low-order byte of register `%eax` to the second byte of `%edx`. The `movb` instruction does not change the other three bytes. The `movsbl` instruction sets the other three bytes to either all ones or all zeros depending on the high-order bit of the source byte. The `movzbl` instruction sets the other three bytes to all zeros in any case. **End Aside.**

The final two data movement operations are used to push data onto and pop data from the program stack. As we will see, the stack plays a vital role in the handling of procedure calls. Both the `pushl` and the `popl` instructions take a single operand—the data source for pushing and the data destination for popping. The

code/asm/exchange.c

```
1 int exchange(int *xp, int y)
2 {
3     int x = *xp;
4
5     *xp = y;
6     return x;
7 }
```

code/asm/exchange.c

(a) C code

```
1   movl 8(%ebp),%eax     Get xp
2   movl 12(%ebp),%edx    Get y
3   movl (%eax),%ecx      Get x at *xp
4   movl %edx,(%eax)      Store y at *xp
5   movl %ecx,%eax        Set x as return value
```

(b) Assembly code

Figure 3.5: **C and Assembly Code for Exchange Routine Body.** The stack set-up and completion portions have been omitted.

program stack is stored in some region of memory. The stack grows downward such that the top element of the stack has the lowest address of all stack elements. The stack pointer `%esp` holds the address of this lowest stack element. Pushing a double-word value onto the stack therefore involves first decrementing the stack pointer by 4 and then writing the value at the new top of stack address. Therefore, the instruction `pushl %ebp` has equivalent behavior to the following pair of instructions:

```
subl $4,%esp
movl %ebp,(%esp)
```

except that the `pushl` instruction is encoded in the object code as a single byte, whereas the pair of instruction shown above requires a total of 6 bytes. Popping a double word involves reading from the top of stack location and then incrementing the stack pointer by 4. Therefore the instruction `popl %eax` is equivalent to the following pair of instructions:

```
movl (%esp),%eax
addl $4,%esp
```

3.4.3 Data Movement Example

New to C?
Function `exchange` (Figure 3.5) provides a good illustration of the use of pointers in C. Argument `xp` is a pointer to an integer, while `y` is an integer itself. The statement

```
int x = *xp;
```

indicates that we should read the value stored in the location designated by `xp` and store it as a local variable named `x`. This read operation is known as pointer *dereferencing*. The C operator `*` performs pointer dereferencing.

The statement

```
*xp = y;
```

does the reverse—it writes the value of parameter `y` at the location designated by `xp`. This also a form of pointer dereferencing (and hence the operator `*`), but it indicates a write operation since it is on the left hand side of the assignment statement.

Here is an example of `exchange` in action:

```
int a = 4;
int b = exchange(&a, 3);
printf("a = %d, b = %d\n", a, b);
```

This code will print

```
a = 3, b = 4
```

The C operator `&` (called the "address of" operator) *creates* a pointer, in this case to the location holding local variable `a`. Function `exchange` then overwrote the value stored in `a` with 3 but returned 4 as the function value. Observe how by passing a pointer to `exchange`, it could modify data held at some remote location. **End**

As an example of code that uses data movement instructions, consider the data exchange routine shown in Figure 3.5, both as C code and as assembly code generated by GCC. We omit the portion of the assembly code that allocates space on the run-time stack on procedure entry and deallocates it prior to return. The details of this set-up and completion code will be covered when we discuss procedure linkage. The code we are left with is called the "body."

When the body of the procedure starts execution, procedure parameters `xp` and `y` are stored at offsets 8 and 12 relative to the address in register `%ebp`. Instructions 1 and 2 then move these parameters into registers `%eax` and `%edx`. Instruction 3 dereferences `xp` and stores the value in register `%ecx`, corresponding to program value `x`. Instruction 4 stores `y` at `xp`. Instruction 5 moves `x` to register `%eax`. By convention, any function returning an integer or pointer value does so by placing the result in register `%eax`, and so this instruction implements line 6 of the C code. This example illustrates how the `movl` instruction can be used to read from memory to a register (instructions 1 to 3), to write from a register to memory (instruction 4), and to copy from one register to another (instruction 5).

Two features about this assembly code are worth noting. First, we see that what we call "pointers" in C are simply addresses. Dereferencing a pointer involves putting that pointer in a register, and then using this register in an indirect memory reference. Second, local variables such as `x` are often kept in registers rather than stored in memory locations. Register access is much faster than memory access.

Practice Problem 3.2:

You are given the following information. A function with prototype

```
void decode1(int *xp, int *yp, int *zp);
```

is compiled into assembly code. The body of the code is as follows:

```
1   movl 8(%ebp),%edi
2   movl 12(%ebp),%ebx
3   movl 16(%ebp),%esi
```

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow$ &S	Load Effective Address
`incl`	D	$D \leftarrow D$ + 1	Increment
`decl`	D	$D \leftarrow D$ - 1	Decrement
`negl`	D	$D \leftarrow$ -D	Negate
`notl`	D	$D \leftarrow$ ~D	Complement
`addl`	S, D	$D \leftarrow D$ + S	Add
`subl`	S, D	$D \leftarrow D$ - S	Subtract
`imull`	S, D	$D \leftarrow D$ * S	Multiply
`xorl`	S, D	$D \leftarrow D$ ^ S	Exclusive-Or
`orl`	S, D	$D \leftarrow D$ | S	Or
`andl`	S, D	$D \leftarrow D$ & S	And
`sall`	k, D	$D \leftarrow D$ << k	Left Shift
`shll`	k, D	$D \leftarrow D$ << k	Left Shift (same as `sall`)
`sarl`	k, D	$D \leftarrow D$ >> k	Arithmetic Right Shift
`shrl`	k, D	$D \leftarrow D$ >> k	Logical Right Shift

Figure 3.6: **Integer Arithmetic Operations.** The Load Effective Address `leal` is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. Note the nonintuitive ordering of the operands with GAS.

```
4   movl (%edi),%eax
5   movl (%ebx),%edx
6   movl (%esi),%ecx
7   movl %eax,(%ebx)
8   movl %edx,(%esi)
9   movl %ecx,(%edi)
```

Parameters `xp`, `yp`, and `zp` are stored at memory locations with offsets 8, 12, and 16, respectively, relative to the address in register `%ebp`.

Write C code for `decode1` that will have an effect equivalent to the assembly code above. You can test your answer by compiling your code with the `-S` switch. Your compiler may generate code that differs in the usage of registers or the ordering of memory references, but it should still be functionally equivalent.

3.5 Arithmetic and Logical Operations

Figure 3.6 lists some of the double-word integer operations, divided into four groups. *Binary* operations have two operands, while *unary* operations have one operand. These operands are specified using the same notation as described in Section 3.4. With the exception of `leal`, each of these instructions has a counterpart that operates on words (16 bits) and on bytes. The suffix '`l`' is replaced by '`w`' for word operations and '`b`' for the byte operations. For example, `addl` becomes `addw` or `addb`.

3.5.1 Load Effective Address

The Load Effective Address `leal` instruction is actually a variant of the `movl` instruction. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 3.6 using the C address operator $\&S$. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. The destination operand must be a register.

Practice Problem 3.3:

Suppose register `%eax` holds value x and `%ecx` holds value y. Fill in the table below with formulas indicating the value that will be stored in register `%edx` for each of the following assembly code instructions.

Expression	Result
`leal 6(%eax), %edx`	
`leal (%eax,%ecx), %edx`	
`leal (%eax,%ecx,4), %edx`	
`leal 7(%eax,%eax,8), %edx`	
`leal 0xA(,$ecx,4), %edx`	
`leal 9(%eax,%ecx,2), %edx`	

3.5.2 Unary and Binary Operations

Operations in the second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or a memory location. For example, the instruction `incl (%esp)` causes the element on the top of the stack to be incremented. This syntax is reminiscent of the C increment (`++`) and decrement operators (`--`).

The third group consists of binary operations, where the second operand is used as both a source and a destination. This syntax is reminiscent of the C assignment operators such as `+=`. Observe, however, that the source operand is given first and the destination second. This looks peculiar for noncommutative operations. For example, the instruction `subl %eax,%edx` decrements register `%edx` by the value in `%eax`. The first operand can be either an immediate value, a register, or a memory location. The second can be either a register or a memory location. As with the `movl` instruction, however, the two operands cannot both be memory locations.

Practice Problem 3.4:

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value.

Instruction	Destination	Value
`addl %ecx,(%eax)`		
`subl %edx,4(%eax)`		
`imull $16,(%eax,%edx,4)`		
`incl 8(%eax)`		
`decl %ecx`		
`subl %edx,%eax`		

3.5.3 Shift Operations

The final group consists of shift operations, where the shift amount is given first, and the value to shift is given second. Both arithmetic and logical right shifts are possible. The shift amount is encoded as a single byte, since only shifts amounts between 0 and 31 are allowed. The shift amount is given either as an immediate or in the single-byte register element `%cl`. As Figure 3.6 indicates, there are two names for the left shift instruction: `sall` and `shll`. Both have the same effect, filling from the right with 0s. The right shift instructions differ in that `sarl` performs an arithmetic shift (fill with copies of the sign bit), whereas `shrl` performs a logical shift (fill with 0s).

Practice Problem 3.5:

Suppose we want to generate assembly code for the following C function:

```
int shift_left2_rightn(int x, int n)
{
  x <<= 2;
  x >>= n;
  return x;
}
```

The following is a portion of the assembly code that performs the actual shifts and leaves the final value in register `%eax`. Two key instructions have been omitted. Parameters `x` and `n` are stored at memory locations with offsets 8 and 12, respectively, relative to the address in register `%ebp`.

```
1   movl 12(%ebp),%ecx    Get x
2   movl 8(%ebp),%eax     Get n
3   _______________       x <<= 2
4   _______________       x >>= n
```

Fill in the missing instructions, following the annotations on the right. The right shift should be performed arithmetically.

code/asm/arith.c

```
1 int arith(int x,
2           int y,
3           int z)
4 {
5     int t1 = x+y;
6     int t2 = z*48;
7     int t3 = t1 & 0xFFFF;
8     int t4 = t2 * t3;
9
10    return t4;
11 }
```

code/asm/arith.c

(a) C code

```
1   movl 12(%ebp),%eax          Get y
2   movl 16(%ebp),%edx          Get z
3   addl 8(%ebp),%eax           Compute t1 = x+y
4   leal (%edx,%edx,2),%edx     Compute z*3
5   sall $4,%edx                Compute t2 = z*48
6   andl $65535,%eax            Compute t3 = t1&0xFFFF
7   imull %eax,%edx             Compute t4 = t2*t3
8   movl %edx,%eax              Set t4 as return val
```

(b) Assembly code

Figure 3.7: **C and Assembly Code for Arithmetic Routine Body.** The stack set-up and completion portions have been omitted.

3.5.4 Discussion

With the exception of the right shift operations, none of the instructions distinguish between signed and unsigned operands. Two's complement arithmetic has the same bit-level behavior as unsigned arithmetic for all of the instructions listed.

Figure 3.7 shows an example of a function that performs arithmetic operations and its translation into assembly. As before, we have omitted the stack set-up and completion portions. Function arguments `x`, `y`, and `z` are stored in memory at offsets 8, 12, and 16 relative to the address in register `%ebp`, respectively.

Instruction 3 implements the expression `x+y`, getting one operand `y` from register `%eax` (which was fetched by instruction 1) and the other directly from memory. Instructions 4 and 5 perform the computation `z*48`, first using the `leal` instruction with a scaled-indexed addressing mode operand to compute $(z + 2z) = 3z$, and then shifting this value left 4 bits to compute $2^4 \cdot 3z = 48z$. The C compiler often generates combinations of add and shift instructions to perform multiplications by constant factors, as was discussed in Section 2.3.6 (page 63). Instruction 6 performs the AND operation and instruction 7 performs the final multiplication. Then instruction 8 moves the return value into register `%eax`.

In the assembly code of Figure 3.7, the sequence of values in register `%eax` correspond to program values `y`, `t1`, `t3`, and `t4` (as the return value). In general, compilers generate code that uses individual registers for multiple program values and that move program values among the registers.

Practice Problem 3.6:

In the compilation of the following loop:

```
for (i = 0; i < n; i++)
    v += i;
```

we find the following assembly code line:

Instruction	Effect	Description
`imull` S	$Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \leftarrow S \times Reg[\texttt{\%eax}]$	Signed Full Multiply
`mull` S	$Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \leftarrow S \times Reg[\texttt{\%eax}]$	Unsigned Full Multiply
`cltd`	$Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \leftarrow SignExtend(Reg[\texttt{\%eax}])$	Convert to Quad Word
`idivl` S	$Reg[\texttt{\%edx}] \leftarrow Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \bmod S$; $Reg[\texttt{\%eax}] \leftarrow Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \div S$	Signed Divide
`divl` S	$Reg[\texttt{\%edx}] \leftarrow Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \bmod S$; $Reg[\texttt{\%eax}] \leftarrow Reg[\texttt{\%edx}]{:}Reg[\texttt{\%eax}] \div S$	Unsigned Divide

Figure 3.8: **Special Arithmetic Operations.** These operations provide full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%edx` and `%eax` are viewed as forming a single 64-bit quad word.

```
xorl %edx,%edx
```

Explain why this instruction would be there, even though there are no EXCLUSIVE-OR operators in our C code. What operation in the C program does this instruction implement?

3.5.5 Special Arithmetic Operations

Figure 3.8 describes instructions that support generating the full 64-bit product of two 32-bit numbers, as well as integer division.

The `imull` instruction listed in Figure 3.6 is known as the "two-operand" multiply instruction. It generates a 32-bit product from two 32-bit operands, implementing the operations $*^{u}_{32}$ and $*^{t}_{32}$ described in Sections 2.3.4 and 2.3.5 (pages 61 and 62). Recall that when truncating the product to 32 bits, both unsigned multiply and two's complement multiply have the same bit-level behavior. IA32 also provides two different "one-operand" multiply instructions to compute the full 64-bit product of two 32-bit values—one for unsigned (`mull`), and one for two's complement (`imull`) multiplication. For both of these, one argument must be in register `%eax`, and the other is given as the instruction source operand. The product is then stored in registers `%edx` (high-order 32 bits) and `%eax` (low-order 32 bits). Note that although the name `imull` is used for two distinct multiplication operations, the assembler can tell which one is intended by counting the number of operands.

As an example, suppose we have signed numbers `x` and `y` stored at positions 8 and 12 relative to `%ebp`, and we want to store their full 64-bit product as 8 bytes on top of the stack. The code would proceed as follows:

```
     x at %ebp+8, y at %ebp+12
1    movl 8(%ebp),%eax          Put x in %eax
2    imull 12(%ebp)             Multiply by y
3    pushl %edx                 Push high-order 32 bits
4    pushl %eax                 Push low-order 32 bits
```

Observe that the order in which we push the two registers is correct for a little-endian machine in which the stack grows toward lower addresses, i.e., the low-order bytes of the product will have lower addresses than the high-order bytes.

Our earlier table of arithmetic operations (Figure 3.6) does not list any division or modulus operations. These operations are provided by the single-operand divide instructions similar to the single-operand multiply instructions. The signed division instruction `idivl` takes as dividend the 64-bit quantity in registers `%edx` (high-order 32 bits) and `%eax` (low-order 32 bits). The divisor is given as the instruction operand. The instructions store the quotient in register `%eax` and the remainder in register `%edx`. The `cltd`[1] instruction can be used to form the 64-bit dividend from a 32-bit value stored in register `%eax`. This instruction sign extends `%eax` into `%edx`.

As an example, suppose we have signed numbers `x` and `y` stored in positions 8 and 12 relative to `%ebp`, and we want to store values `x/y` and `x%y` on the stack. The code would proceed as follows:

```
     x at %ebp+8, y at %ebp+12
1    movl 8(%ebp),%eax          Put x in %eax
2    cltd                       Sign extend into %edx
3    idivl 12(%ebp)             Divide by y
4    pushl %eax                 Push x / y
5    pushl %edx                 Push x % y
```

The `divl` instruction performs unsigned division. Typically register `%edx` is set to 0 beforehand.

3.6 Control

Up to this point, we have considered ways to access and operate on data. Another important part of program execution is to control the sequence of operations that are performed. The default for statements in C as well as for assembly code is to have control flow sequentially, with statements or instructions executed in the order they appear in the program. Some constructs in C, such as conditionals, loops, and switches, allow the control to flow in nonsequential order, with the exact sequence depending on the values of program data.

Assembly code provides lower-level mechanisms for implementing nonsequential control flow. The basic operation is to jump to a different part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon these low-level mechanisms to implement the control constructs of C.

In our presentation, we first cover the machine-level mechanisms and then show how the different control constructs of C are implemented with them.

3.6.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. The most useful condition codes are:

CF: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

[1] This instruction is called `cdq` in the Intel documentation, one of the few cases where the GAS name for an instruction bears no relation to the Intel name.

ZF: Zero Flag. The most recent operation yielded zero.

SF: Sign Flag. The most recent operation yielded a negative value.

OF: Overflow Flag. The most recent operation caused a two's complement overflow—either negative or positive.

For example, suppose we used the `addl` instruction to perform the equivalent of the C expression `t=a+b`, where variables `a`, `b`, and `t` are of type `int`. Then the condition codes would be set according to the following C expressions:

CF:	`(unsigned t) < (unsigned a)`	Unsigned overflow
ZF:	`(t == 0)`	Zero
SF:	`(t < 0)`	Negative
OF:	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

The `leal` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 3.6 cause the condition codes to be set. For the logical operations, such as `xorl`, the carry and overflow flags are set to 0. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to 0.

In addition to the operations of Figure 3.6, two operations (having 8, 16, and 32-bit forms) set conditions codes without altering any other registers:

Instruction		Based on	Description
`cmpb`	S_2, S_1	S_1 - S_2	Compare bytes
`testb`	S_2, S_1	S_1 & S_2	Test byte
`cmpw`	S_2, S_1	S_1 - S_2	Compare words
`testw`	S_2, S_1	S_1 & S_2	Test word
`cmpl`	S_2, S_1	S_1 - S_2	Compare double words
`testl`	S_2, S_1	S_1 & S_2	Test double word

The `cmpb`, `cmpw`, and `cmpl` instructions set the condition codes according to the difference of their two operands. With GAS format, the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands.

The `testb`, `testw`, and `testl` instructions set the zero and negative flags based on the AND of their two operands. Typically, the same operand is repeated (e.g., `testl %eax,%eax` to see whether `%eax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

3.6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, the two most common methods of accessing them are to set an integer register or to perform a conditional branch based on some combination of condition codes.

The different `set` instructions described in Figure 3.9 set a single byte to 0 or to 1 depending on some combination of the conditions codes. The destination operand is either one of the eight single-byte register

Instruction	Synonym	Effect	Set Condition
`sete` D	`setz`	$D \leftarrow$ `ZF`	Equal / Zero
`setne` D	`setnz`	$D \leftarrow$ `~ZF`	Not Equal / Not Zero
`sets` D		$D \leftarrow$ `SF`	Negative
`setns` D		$D \leftarrow$ `~SF`	Nonnegative
`setg` D	`setnle`	$D \leftarrow$ `~(SF^OF)&~ZF`	Greater (Signed >)
`setge` D	`setnl`	$D \leftarrow$ `~(SF^OF)`	Greater or Equal (Signed >=)
`setl` D	`setnge`	$D \leftarrow$ `SF^OF`	Less (Signed <)
`setle` D	`setng`	$D \leftarrow$ `(SF^OF)|ZF`	Less or Equal (Signed <=)
`seta` D	`setnbe`	$D \leftarrow$ `~CF&~ZF`	Above (Unsigned >)
`setae` D	`setnb`	$D \leftarrow$ `~CF`	Above or Equal (Unsigned >=)
`setb` D	`setnae`	$D \leftarrow$ `CF`	Below (Unsigned <)
`setbe` D	`setna`	$D \leftarrow$ `CF&~ZF`	Below or Equal (Unsigned <=)

Figure 3.9: **The `set` Instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," i.e., alternate names for the same machine instruction.

elements (Figure 3.2) or a memory location where the single byte is to be stored. To generate a 32-bit result, we must also clear the high-order 24 bits. A typical instruction sequence for a C predicate such as `a<b` is therefore as follows

```
     Note: a is in %edx, b is in %eax
1    cmpl %eax,%edx    Compare a:b
2    setl %al          Set low order byte of %eax to 0 or 1
3    movzbl %al,%eax   Set remaining bytes of %eax to 0
```

using the `movzbl` instruction to clear the high-order three bytes.

For some of the underlying machine instructions, there are multiple possible names, which we list as "synonyms." For example both "`setg`" (for "SET-Greater") and "`setnle`" (for "SET-Not-Less-or-Equal") refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic operations set the condition codes, the descriptions of the different `set` commands apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t=a-b`. For example, consider the `sete`, or "Set when equal" instruction. When $\texttt{a} = \texttt{b}$, we will have $\texttt{t} = 0$, and hence the zero flag indicates equality.

Similarly, consider testing a signed comparison with the `setl`, or "Set when less," instruction. When `a` and `b` are in two's complement form, then for $\texttt{a} < \texttt{b}$ we will have $\texttt{a} - \texttt{b} < 0$ if the true difference were computed. When there is no overflow, this would be indicated by having the sign flag set. When there is positive overflow, because $\texttt{a} - \texttt{b}$ is a large positive number, however, we will have $\texttt{t} < 0$. When there is negative overflow, because $\texttt{a} - \texttt{b}$ is a small negative number, we will have $\texttt{t} > 0$. In either case, the sign flag will indicate the opposite of the sign of the true difference. Hence, the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether $\texttt{a} < \texttt{b}$. The other signed comparison tests are based on

other combinations of SF ^ OF and ZF.

For the testing of unsigned comparisons, the carry flag will be set by the cmpl instruction when the integer difference a – b of the unsigned arguments a and b would be negative, that is, when (unsigned) a < (unsigned) b. Thus, these tests use combinations of the carry and zero flags.

Practice Problem 3.7:

In the following C code, we have replaced some of the comparison operators with "__" and omitted the data types in the casts.

```
1 char ctest(int a, int b, int c)
2 {
3   char t1 =         a __             b;
4   char t2 =         b __ (     )     a;
5   char t3 = (     ) c __ (     )     a;
6   char t4 = (     ) a __ (     )     c;
7   char t5 =         c __             b;
8   char t6 =         a __             0;
9   return t1 + t2 + t3 + t4 + t5 + t6;
10 }
```

For the original C code, GCC generates the following assembly code

```
1   movl 8(%ebp),%ecx          Get a
2   movl 12(%ebp),%esi         Get b
3   cmpl %esi,%ecx             Compare a:b
4   setl %al                   Compute t1
5   cmpl %ecx,%esi             Compare b:a
6   setb -1(%ebp)              Compute t2
7   cmpw %cx,16(%ebp)          Compare c:a
8   setge -2(%ebp)             Compute t3
9   movb %cl,%dl
10  cmpb 16(%ebp),%dl          Compare a:c
11  setne %bl                  Compute t4
12  cmpl %esi,16(%ebp)         Compare c:b
13  setg -3(%ebp)              Compute t5
14  testl %ecx,%ecx            Test a
15  setg %dl                   Compute t4
16  addb -1(%ebp),%al          Add t2 to t1
17  addb -2(%ebp),%al          Add t3 to t1
18  addb %bl,%al               Add t4 to t1
19  addb -3(%ebp),%al          Add t5 to t1
20  addb %dl,%al               Add t6 to t1
21  movsbl %al,%eax            Convert sum from char to int
```

Based on this assembly code, fill in the missing parts (the comparisons and the casts) in the C code.

Instruction		Synonym	Jump Condition	Description
`jmp`	*Label*		1	Direct Jump
`jmp`	**Operand*		1	Indirect Jump
`je`	*Label*	`jz`	`ZF`	Equal / Zero
`jne`	*Label*	`jnz`	`~ZF`	Not Equal / Not Zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (Signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or Equal (Signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (Signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or Equal (Signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (Unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or Equal (Unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (Unsigned <)
`jbe`	*Label*	`jna`	`CF & ~ZF`	Below or Equal (Unsigned <=)

Figure 3.10: **The `jump` Instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

3.6.3 Jump Instructions and their Encodings

Under normal execution, instructions follow each other in the order they are listed. A jump instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated by a *label*. Consider the following assembly code sequence:

```
1   xorl %eax,%eax          Set %eax to 0
2   jmp .L1                 Goto .L1
3   movl (%eax),%edx        Null pointer dereference
4 .L1:
5   popl %edx
```

The instruction `jmp .L1` will cause the program to skip over the `movl` instruction and instead resume execution with the `popl` instruction. In generating the object code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly by giving a label as the jump target, e.g., the label "`.L1`" in the code above. Indirect jumps are written using '`*`' followed by an operand specifier using the same syntax as used for the `movl` instruction. As examples, the instruction

```
jmp *%eax
```

uses the value in register `%eax` as the jump target, while

```
jmp *(%eax)
```

reads the jump target from memory, using the value in `%eax` as the read address.

The other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the condition codes. Note that the names of these instructions and the conditions under which they jump match those of the `set` instructions. As with the `set` instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

Although we will not concern ourselves with the detailed format of object code, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter 7. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are *PC-relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using one, two, or four bytes. A second encoding method is to give an "absolute" address, using four bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

As an example, the following fragment of assembly code was generated by compiling a file `silly.c`. It contains two jumps: the `jle` instruction on line 1 jumps forward to a higher address, while the `jg` instruction on line 8 jumps back to a lower one.

```
1   jle .L4                  If <, goto dest2
2   .p2align 4,,7            Aligns next instruction to multiple of 8
3 .L5:                     dest1:
4   movl %edx,%eax
5   sarl $1,%eax
6   subl %eax,%edx
7   testl %edx,%edx
8   jg .L5                   If >, goto dest1
9 .L4:                     dest2:
10  movl %edx,%eax
```

Note that line 2 is a directive to the assembler that causes the address of the following instruction to begin on a multiple of 16, but leaving a maximum of 7 wasted bytes. This directive is intended to allow the processor to make optimal use of the instruction cache memory.

The disassembled version of the "`.o`" format generated by the assembler is as follows:

```
1    8:   7e 11                   jle    1b <silly+0x1b>    Target = dest2
2    a:   8d b6 00 00 00 00       lea    0x0(%esi),%esi     Added nops
3   10:   89 d0                   mov    %edx,%eax        dest1:
4   12:   c1 f8 01                sar    $0x1,%eax
5   15:   29 c2                   sub    %eax,%edx
6   17:   85 d2                   test   %edx,%edx
7   19:   7f f5                   jg     10 <silly+0x10>    Target = dest1
8   1b:   89 d0                   mov    %edx,%eax        dest2:
```

The "`lea 0x0(%esi),%esi`" instruction in line 2 has no real effect. It serves as a 6-byte `nop` so that the next instruction (line 3) has a starting address that is a multiple of 16.

In the annotations generated by the disassembler on the right, the jump targets are indicated explicitly as `0x1b` for instruction 1 and `0x10` for instruction 7. Looking at the byte encodings of the instructions, however, we see that the target of jump instruction 1 is encoded (in the second byte) as `0x11` (decimal 17). Adding this to `0xa` (decimal 10), the address of the following instruction, we get jump target address `0x1b` (decimal 27), the address of instruction 8.

Similarly, the target of jump instruction 7 is encoded as `0xf5` (decimal -11) using a single-byte, two's complement representation. Adding this to `0x1b` (decimal 27), the address of instruction 8, we get `0x10` (decimal 16), the address of instruction 3.

The following shows the disassembled version of the program after linking:

```
1  80483c8:  7e 11                  jle    80483db <silly+0x1b>
2  80483ca:  8d b6 00 00 00 00       lea    0x0(%esi),%esi
3  80483d0:  89 d0                  mov    %edx,%eax
4  80483d2:  c1 f8 01               sar    $0x1,%eax
5  80483d5:  29 c2                  sub    %eax,%edx
6  80483d7:  85 d2                  test   %edx,%edx
7  80483d9:  7f f5                  jg     80483d0 <silly+0x10>
8  80483db:  89 d0                  mov    %edx,%eax
```

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 1 and 7 remain unchanged. By using a PC-relative encoding of the jump targets, the instructions can be compactly encoded (requiring just two bytes), and the object code can be shifted to different positions in memory without alteration.

Practice Problem 3.8:

In the following excerpts from a disassembled binary, some of the information has been replaced by `X`'s. Determine the following information about these instructions.

A. What is the target of the `jbe` instruction below?

```
8048d1c:  76 da           jbe    XXXXXXX
8048d1e:  eb 24           jmp    8048d44
```

B. What is the address of the `mov` instruction?

```
XXXXXXX:  eb 54           jmp    8048d44
XXXXXXX:  c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

C. In the following, the jump target is encoded in PC-relative form as a 4-byte, two's complement number. The bytes are listed from least significant to most, reflecting the little endian byte ordering of IA32. What is the address of the jump target?

```
8048902:  e9 cb 00 00 00  jmp    XXXXXXX
8048907:  90              nop
```

D. Explain the relation between the annotation on the right and the byte coding on the left. Both lines are part of the encoding of the `jmp` instruction.

```
80483f0:  ff 25 e0 a2 04  jmp    *0x804a2e0
80483f5:  08
```

To implement the control constructs of C, the compiler must use the different types of jump instructions we have just seen. We will go through the most common constructs, starting from simple conditional branches, and then considering loops and switch statements.

3.6.4 Translating Conditional Branches

Conditional statements in `C` are implemented using combinations of conditional and unconditional jumps. For example, Figure 3.11 shows the C code for a function that computes the absolute value of the difference of two numbers (a). GCC generates the assembly code shown as (c). We have created a version in C, called `gotodiff` (b), that more closely follows the control flow of this assembly code. It uses the `goto` statement in C, which is similar to the unconditional jump of assembly code. The statement `goto less` on line 6 causes a jump to the label `less` on line 8, skipping the statement on line 7. Note that using `goto` statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of assembly-code programs. We call such C programs "goto code."

The assembly code implementation first compares the two operands (line 3), setting the condition codes. If the comparison result indicates that `x` is less than `y`, it then jumps to a block of code that computes `x-y` (line 9). Otherwise it continues with the execution of code that computes `y-x` (lines 5 and 6). In both cases the computed result is stored in register `%eax`, and ends up at line 10, at which point it executes the stack completion code (not shown).

The general form of an if-else statement in C is given by the `if-else` statement following template:

```
if (test-expr)
  then-statement
else
  else-statement
```

where *test-expr* is an integer expression that evaluates either to 0 (interpreted as meaning "false") or to a nonzero value (interpreted as meaning "true"). Only one of the two branch statements (*then-statement* or *else-statement*) is executed.

For this general form, the assembly implementation typically follows the form shown below, where we use C syntax to describe the control flow:

```
  t = test-expr;
  if (t)
    goto true;
  else-statement
  goto done;
true:
  then-statement
done:
```

code/asm/abs.c

```
1 int absdiff(int x, int y)
2 {
3     if (x < y)
4         return y - x;
5     else
6         return x - y;
7 }
```

code/asm/abs.c

(a) Original C code.

code/asm/abs.c

```
1 int gotodiff(int x, int y)
2 {
3     int rval;
4
5     if (x < y)
6         goto less;
7     rval = x - y;
8     goto done;
9  less:
10    rval = y - x;
11 done:
12    return rval;
13 }
```

code/asm/abs.c

(b) Equivalent goto version of (a).

```
1   movl 8(%ebp),%edx          Get x
2   movl 12(%ebp),%eax         Get y
3   cmpl %eax,%edx             Compare x:y
4   jl .L3                     If <, goto less:
5   subl %eax,%edx             Compute y-x
6   movl %edx,%eax             Set as return value
7   jmp .L5                    Goto done:
8 .L3:                       less:
9   subl %edx,%eax             Compute x-y as return value
10 .L5:                      done: Begin completion code
```

(c) Generated assembly code.

Figure 3.11: **Compilation of Conditional Statements** C procedure `absdiff` (a) contains an if-else statement. The generated assembly code is shown (c), along with a C procedure `gotodiff` (b) that mimics the control flow of the assembly code. The stack set-up and completion portions of the assembly code have been omitted

That is, the compiler generates separate blocks of code for *then-statement* and *else-statement*. It inserts conditional and unconditional branches to make sure the correct block is executed.

Practice Problem 3.9:

When given the following C code:

code/asm/simple-if.c

```
1 void cond(int a, int *p)
2 {
3   if (p && a > 0)
4     *p += a;
5 }
```

code/asm/simple-if.c

GCC generates the following assembly code.

```
1   movl 8(%ebp),%edx
2   movl 12(%ebp),%eax
3   testl %eax,%eax
4   je .L3
5   testl %edx,%edx
6   jle .L3
7   addl %edx,(%eax)
8 .L3:
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.11(b). You might find it helpful to first annotate the assembly code as we have done in our examples.

B. Explain why the assembly code contains two conditional branches, even though the C code has only one if statement.

3.6.5 Loops

C provides several looping constructs, namely `while`, `for`, and `do-while`. No corresponding instructions exist in assembly. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Interestingly, most compilers generate loop code based on the `do-while` form of a loop, even though this form is relatively uncommon in actual programs. Other loops are transformed into `do-while` form and then compiled into machine code. We will study the translation of loops as a progression, starting with `do-while` and then working toward ones with more complex implementations.

Do-While Loops

The general form of a `do-while` statement is as follows:

```
do
  body-statement
  while (test-expr);
```

The effect of the loop is to repeatedly execute *body-statement*, evaluate *test-expr* and continue the loop if the evaluation result is nonzero. Observe that *body-statement* is executed at least once.

Typically, the implementation of `do-while` has the following general form:

```
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
```

As an example, Figure 3.12 shows an implementation of a routine to compute the nth element in the Fibonacci sequence using a `do-while` loop. This sequence is defined by the recurrence:

$$\begin{array}{lcl} F_1 & = & 1 \\ F_2 & = & 1 \\ F_n & = & F_{n-2} + F_{n-3}, \quad n \geq 3 \end{array}$$

For example, the first ten elements of the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, and 55. To implement this using a `do-while` loop, we have started the sequence with values $F_0 = 0$ and $F_1 = 1$, rather than with F_1 and F_2.

The assembly code implementing the loop is also shown, along with a table showing the correspondence between registers and program values. In this example, *body-statement* consists of lines 8 through 11, assigning values to `t`, `val`, and `nval`, along with the incrementing of `i`. These are implemented by lines 2 through 5 of the assembly code. The expression `i < n` comprises *test-expr*. This is implemented by line 6 and by the test condition of the jump instruction on line 7. Once the loop exits, `val` is copy to register `%eax` as the return value (line 8).

Creating a table of register usage, such as we have shown in Figure 3.12(b) is a very helpful step in analyzing an assembly language program, especially when loops are present.

Practice Problem 3.10:

For the following C code:

```
1 int dw_loop(int x, int y, int n)
2 {
3   do {
4     x += n;
5     y *= n;
6     n--;
```

code/asm/fib.c

```
1 int fib_dw(int n)
2 {
3     int i = 0;
4     int val = 0;
5     int nval = 1;
6
7     do {
8         int t = val + nval;
9         val = nval;
10        nval = t;
11        i++;
12    } while (i < n);
13
14    return val;
15 }
```

code/asm/fib.c

(a) C code.

Register Usage		
Register	Variable	Initially
%ecx	i	0
%esi	n	n
%ebx	val	0
%edx	nval	1
%eax	t	–

```
1 .L6:                          loop:
2   leal (%edx,%ebx),%eax         Compute t = val + nval
3   movl %edx,%ebx                copy nval to val
4   movl %eax,%edx                Copy t to nval
5   incl %ecx                     Increment i
6   cmpl %esi,%ecx                Compare i:n
7   jl .L6                        If less, goto loop
8   movl %ebx,%eax                Set val as return value
```

(b) Corresponding assembly language code.

Figure 3.12: **C and Assembly Code for Do-While Version of Fibonacci Program.** Only the code inside the loop is shown.

```
7   } while ((n > 0) & (y < n)); /* Note use of bitwise '&' */
8   return x;
9 }
```

GCC generates the following assembly code:

```
   Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
1   movl 8(%ebp),%esi
2   movl 12(%ebp),%ebx
3   movl 16(%ebp),%ecx
4   .p2align 4,,7     Inserted to optimize cache performance
5 .L6:
6   imull %ecx,%ebx
7   addl %ecx,%esi
8   decl %ecx
9   testl %ecx,%ecx
10  setg %al
11  cmpl %ecx,%ebx
12  setl %dl
13  andl %edx,%eax
14  testb $1,%al
15  jne .L6
```

A. Make a table of register usage, similar to the one shown in Figure 3.12(b).

B. Identify *test-expr* and *body-statement* in the C code, and the corresponding lines in the assembly code.

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.12(b).

While Loops

The general form of a while statement is as follows:

`while (`*test-expr*`)`
body-statement

It differs from `do-while` in that *test-expr* is evaluated and the loop is potentially terminated before the first execution of *body-statement*. A direct translation into a form using goto's would be:

```
loop:
  t = test-expr;
  if (!t)
    goto done;
  body-statement
  goto loop;
done:
```

This translation requires two control statements within the inner loop—the part of the code that is executed the most. Instead, most C compilers transform the code into a `do-while` loop by using a conditional branch to skip the first execution of the body if needed:

```
  if (!test-expr)
    goto done;
  do
    body-statement
    while (test-expr);
  done:
```

This, in turn, can be transformed into goto code as:

```
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
done:
```

As an example, Figure 3.13 shows an implementation of the Fibonacci sequence function using a `while` loop (a). Observe that this time we have started the recursion with elements F_1 (`val`) and F_2 (`nval`). The adjacent C function `fib_w_goto` (b) shows how this code has been translated into assembly. The assembly code in (c) closely follows the C code shown in `fib_w_goto`. The compiler has performed several interesting optimizations, as can be seen in the goto code (b). First, rather than using variable `i` as a loop variable and comparing it to `n` on each iteration, the compiler has introduced a new loop variable that we call "`nmi`", since relative to the original code, its value equals $n - i$. This allows the compiler to use only three registers for loop variables, compared to four otherwise. Second, it has optimized the initial test condition `(i < n)` into `(val < n)`, since the initial values of both `i` and `val` are 1. By this means, the compiler has totally eliminated variable `i`. Often the compiler can make use of the initial values of the variables to optimize the initial test. This can make deciphering the assembly code tricky. Third, for

code/asm/fib.c

```
1 int fib_w(int n)
2 {
3     int i = 1;
4     int val = 1;
5     int nval = 1;
6
7     while (i < n) {
8         int t = val+nval;
9         val = nval;
10        nval = t;
11        i++;
12    }
13
14    return val;
15 }
```

code/asm/fib.c

(a) C code.

code/asm/fib.c

```
1 int fib_w_goto(int n)
2 {
3     int val = 1;
4     int nval = 1;
5     int nmi, t;
6
7     if (val >= n)
8         goto done;
9     nmi = n-1;
10
11  loop:
12    t = val+nval;
13    val = nval;
14    nval = t;
15    nmi--;
16    if (nmi)
17        goto loop;
18
19  done:
20    return val;
21 }
```

code/asm/fib.c

(b) Equivalent goto version of (a).

Register Usage		
Register	Variable	Initially
%edx	nmi	n-1
%ebx	val	1
%ecx	nval	1

```
1   movl 8(%ebp),%eax          Get n
2   movl $1,%ebx               Set val to 1
3   movl $1,%ecx               Set nval to 1
4   cmpl %eax,%ebx             Compare val:n
5   jge .L9                    If >= goto done:
6   leal -1(%eax),%edx         nmi = n-1
7 .L10:                      loop:
8   leal (%ecx,%ebx),%eax      Compute t = nval+val
9   movl %ecx,%ebx             Set val to nval
10  movl %eax,%ecx             Set nval to t
11  decl %edx                  Decrement nmi
12  jnz .L10                   if != 0, goto loop:
13 .L9:                      done:
```

(c) Corresponding assembly language code.

Figure 3.13: **C and Assembly Code for While Version of Fibonacci.** The compiler has performed a number of optimizations, including replacing the value denoted by variable `i` with one we call `nmi`.

successive executions of the loop we are assured that $i \leq n$, and so the compiler can assume that `nmi` is nonnegative. As a result, it can test the loop condition as `nmi != 0` rather than `nmi >= 0`. This saves one instruction in the assembly code.

Practice Problem 3.11:

For the following C code:

```
1 int loop_while(int a, int b)
2 {
3   int i = 0;
4   int result = a;
5   while (i < 256) {
6     result += a;
7     a -= b;
8     i += b;
9   }
10  return result;
11 }
```

GCC generates the following assembly code:

```
   Initially a and b are at offsets 8 and 12 from %ebp
1   movl 8(%ebp),%eax
2   movl 12(%ebp),%ebx
3   xorl %ecx,%ecx
4   movl %eax,%edx
5   .p2align 4,,7
6 .L5:
7   addl %eax,%edx
8   subl %ebx,%eax
9   addl %ebx,%ecx
10  cmpl $255,%ecx
11  jle .L5
```

A. Make a table of register usage within the loop body, similar to the one shown in Figure 3.13(c).

B. Identify *test-expr* and *body-statement* in the C code, and the corresponding lines in the assembly code. What optimizations has the C compiler performed on the initial test?

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.13(c).

D. Write a goto version (in C) of the function that has similar structure to the assembly code, as was done in Figure 3.13(b).

For Loops

The general form of a `for` loop is as follows:

```
for (init-expr; test-expr; update-expr)
  body-statement
```

The C language standard states that the behavior of such a loop is identical to the following code using a while loop:

```
init-expr;
while (test-expr) {
  body-statement
  update-expr;
}
```

That is, the program first evaluates the initialization expression *init-expr*. It then enters a loop where it first evaluates the test condition *test-expr*, exiting if the test fails, then executes the body of the loop *body-statement*, and finally evaluates the update expression *update-expr*.

The compiled form of this code then is based on the transformation from while to `do-while` described previously, first giving a `do-while` form:

```
init-expr;
if (!test-expr)
  goto done;
do {
  body-statement
  update-expr;
} while (test-expr);
done:
```

This, in turn, can be transformed into goto code as:

```
    init-expr;
    t = test-expr;
    if (!t)
      goto done;
  loop:
    body-statement
    update-expr;
    t = test-expr;
    if (t)
      goto loop;
  done:
```

As an example, the following code shows an implementation of the Fibonacci function using a `for` loop:

code/asm/fib.c

```
1  int fib_f(int n)
2  {
3      int i;
4      int val = 1;
5      int nval = 1;
6
7      for (i = 1; i < n; i++) {
8          int t = val+nval;
9          val = nval;
10         nval = t;
11     }
12
13     return val;
14 }
```

code/asm/fib.c

The transformation of this code into the while loop form gives code identical to that for the function `fib_w` shown in Figure 3.13. In fact, GCC generates identical assembly code for the two functions.

Practice Problem 3.12:

The following assembly code:

```
   Initially x, y, and n are offsets 8, 12, and 16 from %ebp
1    movl 8(%ebp),%ebx
2    movl 16(%ebp),%edx
3    xorl %eax,%eax
4    decl %edx
5    js .L4
6    movl %ebx,%ecx
```

```
 7   imull 12(%ebp),%ecx
 8   .p2align 4,,7    Inserted to optimize cache performance
 9 .L6:
10   addl %ecx,%eax
11   subl %ebx,%edx
12   jns .L6
13 .L4:
```

was generated by compiling C code that had the following overall form

```
1 int loop(int x, int y, int n)
2 {
3   int result = 0;
4   int i;
5   for (i = ____; i ____ ; i = ___ ) {
6     result += _____ ;
7   }
8   return result;
9 }
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%eax`. To solve this problem, you may need to do a little bit of guessing about register usage and then see whether that guess makes sense.

A. Which registers hold program values `result` and `i`?

B. What is the initial value of `i`?

C. What is the test condition on `i`?

D. How does `i` get updated?

E. The C expression describing how to increment `result` in the loop body does not change value from one iteration of the loop to the next. The compiler detected this and moved its computation to before the loop. What is the expression?

F. Fill in all the missing parts of the C code.

3.6.6 Switch Statements

Switch statements provide a multi-way branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, they also allow an efficient implementation using a data structure called a *jump table*. A jump table is an array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i. The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction. The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases. GCC selects the method of translating a switch statement based on the number of cases and the sparsity of the case values. Jump tables are used when there are a number of cases (e.g., four or more) and they span a small range of values.

code/asm/switch.c

```
1 int switch_eg(int x)
2 {
3     int result = x;
4
5     switch (x) {
6
7     case 100:
8         result *= 13;
9         break;
10
11    case 102:
12        result += 10;
13        /* Fall through */
14
15    case 103:
16        result += 11;
17        break;
18
19    case 104:
20    case 106:
21        result *= result;
22        break;
23
24    default:
25        result = 0;
26    }
27
28    return result;
29 }
```

code/asm/switch.c

(a) Switch statement.

code/asm/switch.c

```
1 /* Next line is not legal C */
2 code *jt[7] = {
3     loc_A, loc_def, loc_B, loc_C,
4     loc_D, loc_def, loc_D
5 };
6
7 int switch_eg_impl(int x)
8 {
9     unsigned xi = x - 100;
10    int result = x;
11
12    if (xi > 6)
13        goto loc_def;
14
15    /* Next goto is not legal C */
16    goto jt[xi];
17
18 loc_A:      /* Case 100 */
19    result *= 13;
20    goto done;
21
22 loc_B:      /* Case 102 */
23    result += 10;
24    /* Fall through */
25
26 loc_C:    /* Case 103 */
27    result += 11;
28    goto done;
29
30 loc_D:    /* Cases 104, 106 */
31    result *= result;
32    goto done;
33
34 loc_def:  /* Default case*/
35    result = 0;
36
37 done:
38    return result;
39 }
```

code/asm/switch.c

(b) Translation into extended C.

Figure 3.14: **Switch Statement Example with Translation into Extended C.** The translation shows the structure of jump table `jt` and how it is accessed. Such tables and accesses are not actually allowed in C.

```
     Set up the jump table access
 1   leal -100(%edx),%eax          Compute xi = x-100
 2   cmpl $6,%eax                  Compare xi:6
 3   ja .L9                        if >, goto done
 4   jmp *.L10(,%eax,4)            Goto jt[xi]

     Case 100
 5 .L4:                            loc_A:
 6   leal (%edx,%edx,2),%eax       Compute 3*x
 7   leal (%edx,%eax,4),%edx       Compute x+4*3*x
 8   jmp .L3                       Goto done

     Case 102
 9 .L5:                            loc_B:
10   addl $10,%edx                 result += 10, Fall through

     Case 103
11 .L6:                            loc_C:
12   addl $11,%edx                 result += 11
13   jmp .L3                       Goto done

     Cases 104, 106
14 .L8:                            loc_D:
15   imull %edx,%edx               result *= result
16   jmp .L3                       Goto done

     Default case
17 .L9:                            loc_def:
18   xorl %edx,%edx                result = 0

     Return result
19 .L3:                            done:
20   movl %edx,%eax                Set result as return value
```

Figure 3.15: **Assembly Code for Switch Statement Example in Figure 3.14.**

Figure 3.14(a) shows an example of a C `switch` statement. This example has a number of interesting features, including case labels that do not span a contiguous range (there are no labels for cases 101 and 105), cases with multiple labels (cases 104 and 106), and cases that "fall through" to other cases (case 102), because the code for the case does not end with a `break` statement.

Figure 3.15 shows the assembly code generated when compiling `switch_eg`. The behavior of this code is shown using an extended form of C as the procedure `switch_eg_impl` in Figure 3.14(b). We say "extended" because C does not provide the necessary constructs to support this style of jump table, and hence our code is not legal C. The array `jt` contains 7 entries, each of which is the address of a block of code. We extend C with a data type `code` for this purpose.

Lines 1 to 4 set up the jump table access. To make sure that values of `x` that are either less than 100 or greater than 106 cause the computation specified by the `default` case, the code generates an unsigned value `xi` equal to `x-100`. For values of `x` between 100 and 106, `xi` will have values 0 through 6. All other values will be greater than 6, since negative values of `x-100` will wrap around to be very large unsigned numbers. The code therefore uses the `ja` (unsigned greater) instruction to jump to code for the default case when `xi` is greater than 6. Using `jt` to indicate the jump table, the code then performs a jump to the address at entry `xi` in this table. Note that this form of `goto` is not legal C. Instruction 4 implements the jump to an entry in the jump table. Since it is an indirect jump, the target is read from memory. The effective address of the read is determined by adding the base address specified by label `.L10` to the scaled (by 4 since each jump table entry is 4 bytes) value of variable `xi` (in register `%eax`).

In the assembly code, the jump table is indicated by the following declarations, to which we have added comments:

```
1 .section .rodata
2   .align 4          Align address to multiple of 4
3 .L10:
4   .long .L4         Case 100: loc_A
5   .long .L9         Case 101: loc_def
6   .long .L5         Case 102: loc_B
7   .long .L6         Case 103: loc_C
8   .long .L8         Case 104: loc_D
9   .long .L9         Case 105: loc_def
10  .long .L8         Case 106: loc_D
```

These declarations state that within the segment of the object code file called "`.rodata`" (for "Read-Only Data"), there should be a sequence of seven "long" (4-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly code labels (e.g., `.L4`). Label `.L10` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (instruction 4).

The code blocks starting with labels `loc_A` through `loc_D` and `loc_def` in `switch_eg_impl` (Figure 3.14(b)) implement the five different branches of the switch statement. Observe that the block of code labeled `loc_def` will be executed either when `x` is outside the range 100 to 106 (by the initial range checking) or when it equals either 101 or 105 (based on the jump table). Note how the code for the block labeled `loc_B` falls through to the block labeled `loc_C`.

Practice Problem 3.13:

In the following C function, we have omitted the body of the switch statement. In the C code, the case labels did not span a contiguous range, and some cases had multiple labels.

```
int switch2(int x) {
  int result = 0;
  switch (x) {
    /* Body of switch statement omitted */
  }
  return result;
}
```

In compiling the function, GCC generates the following assembly code for the initial part of the procedure and for the jump table. Variable `x` is initially at offset 8 relative to register `%ebp`.

```
   Setting up jump table access
1    movl 8(%ebp),%eax    Retrieve x
2    addl $2,%eax
3    cmpl $6,%eax
4    ja .L10
5    jmp *.L11(,%eax,4)
```

```
   Jump table for switch2
1  .L11:
2    .long .L4
3    .long .L10
4    .long .L5
5    .long .L6
6    .long .L8
7    .long .L8
8    .long .L9
```

From this determine:

A. What were the values of the case labels in the switch statement body?

B. What cases had multiple labels in the C code?

3.7 Procedures

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of the code to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

3.7.1 Stack Frame Structure

IA32 programs make use of the program stack to support procedure calls. The stack is used to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a *stack frame*. Figure 3.16 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register `%ebp` serving as the *frame pointer*, and register `%esp` serving as the *stack pointer*. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

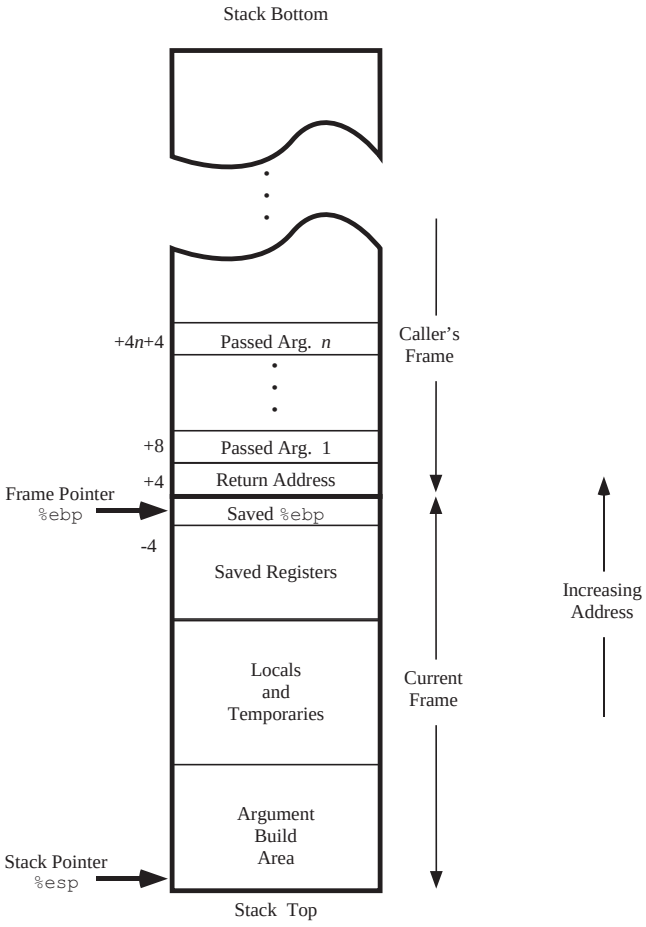


Figure 3.16: **Stack Frame Structure.** The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

Suppose procedure P (the *caller*) calls procedure Q (the *callee*). The arguments to Q are contained within the stack frame for P. In addition, when P calls Q, the *return address* within P where the program should resume execution when it returns from Q is pushed on the stack, forming the end of P's stack frame. The stack frame for Q starts with the saved value of the frame pointer (i.e., %ebp). followed by copies of any other saved register values.

Procedure Q also uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:

- There are not enough registers to hold all of the local data.
- Some of the local variables are arrays or structures and hence must be accessed by array or structure references.
- The address operator '&' is applied to one of the local variables, and hence we must be able to generate an address for it.

Finally, Q will use the stack frame for storing arguments to any procedures it calls.

As described earlier, the stack grows toward lower addresses and the stack pointer %esp points to the top element of the stack. Data can be stored on and retrieved from the stack using the pushl and popl instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated by incrementing the stack pointer.

3.7.2 Transferring Control

The instructions supporting procedure calls and returns are as follows:

Instruction		Description
call	*Label*	Procedure Call
call	**Operand*	Procedure Call
leave		Prepare stack for return
ret		Return from call

The call instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can either be direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by a * followed by an operand specifier having the same syntax as is used for the operands of the movl instruction (Figure 3.3).

The effect of a call instruction is to push a return address on the stack and jump to the start of the called procedure. The return address is the address of the instruction immediately following the call in the program, so that execution will resume at this location when the called procedure returns. The ret instruction pops an address off the stack and jumps to this location. The proper use of this instruction is to have prepared the stack so that the stack pointer points to the place where the preceding call instruction stored its return address. The leave instruction can be used to prepare the stack for returning. It is equivalent to the following code sequence:

```
1   movl %ebp, %esp   Set stack pointer to beginning of frame
2   popl %ebp         Restore saved %ebp and set stack ptr to end of caller's frame
```

Alternatively, this preparation can be performed by an explicit sequence of move and pop operations.

Register %eax is used for returning the value of any function that returns an integer or pointer.

Practice Problem 3.14:

The following code fragment occurs often in the compiled version of library routines:

```
1   call next
2 next:
3   popl %eax
```

A. To what value does register %eax get set?

B. Explain why there is no matching ret instruction to this call.

C. What useful purpose does this code fragment serve?

3.7.3 Register Usage Conventions

The set of program registers acts as a single resource shared by all of the procedures. Although only one procedure can be active at a given time, we must make sure that when one procedure (the *caller*) calls another (the *callee*), the callee does not overwrite some register value that the caller planned to use later. For this reason, IA32 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

By convention, registers %eax, %edx, and %ecx are classified as *caller save* registers. When procedure Q is called by P, it can overwrite these registers without destroying any data required by P. On the other hand, registers %ebx, %esi, and %edi are classified as *callee save* registers. This means that Q must save the values of any of these registers on the stack before overwriting them, and restore them before returning, because P (or some higher level procedure) may need these values for its future computations. In addition, registers %ebp and %esp must be maintained according to the conventions described here.

Aside: Why the names "callee save" and "caller save?"
Consider the following scenario:

```
int P()
{
    int x = f();  /* Some computation */
    Q();
    return x;
}
```

Procedure P wants the value it has computed for x to remain valid across the call to Q. If x is in a *caller save* register, then P (the caller) must save the value before calling P and restore it after Q returns. If x is in a *callee save* register, and Q (the callee) wants to use this register, then Q must save the value before using the register and restore it before returning. In either case, saving involves pushing the register value onto the stack, while restoring involves popping from the stack back to the register. **End Aside.**

As an example, consider the following code:

```
1 int P(int x)
2 {
3     int y = x*x;
4     int z = Q(y);
5
6     return y + z;
7 }
```

Procedure P computes y before calling Q, but it must also ensure that the value of y is available after Q returns. It can do this by one of two means:

- Store the value of y in its own stack frame before calling Q. When Q returns, it can then retrieve the value of y from the stack.
- Store the value of y in a callee save register. If Q, or any procedure called by Q, wants to use this register, it must save the register value in its stack frame and restore the value before it returns. Thus, when Q returns to P, the value of y will be in the callee save register, either because the register was never altered or because it was saved and restored.

Most commonly, GCC uses the latter convention, since it tends to reduce the total number of stack writes and reads.

Practice Problem 3.15:

The following code sequence occurs right near the beginning of the assembly code generated by GCC for a C procedure:

```
1   pushl %edi
2   pushl %esi
3   pushl %ebx
4   movl 24(%ebp),%eax
5   imull 16(%ebp),%eax
6   movl 24(%ebp),%ebx
7   leal 0(,%eax,4),%ecx
8   addl 8(%ebp),%ecx
9   movl %ebx,%edx
```

We see that just three registers (%edi, %esi, and %ebx) are saved on the stack. The program then modifies these and three other registers (%eax, %ecx, and %edx). At the end of the procedure, the values of registers %edi, %esi, and %ebx are restored using popl instructions, while the other three are left in their modified states.

Explain this apparently inconsistency in the saving and restoring of register states.

code/asm/swapadd.c

```
1 int swap_add(int *xp, int *yp)
2 {
3     int x = *xp;
4     int y = *yp;
5
6     *xp = y;
7     *yp = x;
8     return x + y;
9 }
10
11 int caller()
12 {
13     int arg1 = 534;
14     int arg2 = 1057;
15     int sum = swap_add(&arg1, &arg2);
16     int diff = arg1 - arg2;
17
18     return sum * diff;
19 }
```

code/asm/swapadd.c

Figure 3.17: **Example of Procedure Definition and Call.**

3.7.4 Procedure Example

As an example, consider the C procedures defined in Figure 3.17. Figure 3.18 shows the stack frames for the two procedures. Observe that `swap_add` retrieves its arguments from the stack frame for `caller`. These locations are accessed relative to the frame pointer in register `%ebp`. The numbers along the left of the frames indicate the address offsets relative to the frame pointer.

The stack frame for `caller` includes storage for local variables `arg1` and `arg2`, at positions -8 and -4 relative to the frame pointer. These variables must be stored on the stack, since we must generate addresses for them. The following assembly code from the compiled version of `caller` shows how it calls `swap_add`.

```
     Calling code in caller
1    leal -4(%ebp),%eax        Compute &arg2
2    pushl %eax                Push &arg2
3    leal -8(%ebp),%eax        Compute &arg1
4    pushl %eax                Push &arg1
5    call swap_add             Call the swap_add function
```

Observe that this code computes the addresses of local variables `arg2` and `arg1` (using the `leal` instruction) and pushes them on the stack. It then calls `swap_add`.

The compiled code for `swap_add` has three parts: the "setup," where the stack frame is initialized; the "body," where the actual computation of the procedure is performed; and the "finish," where the stack state

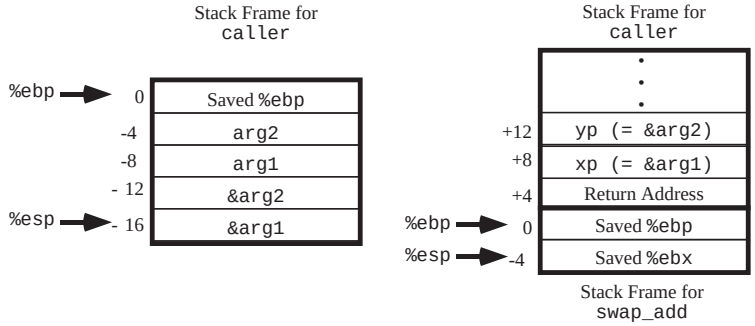


Figure 3.18: **Stack Frames for** `caller` **and** `swap_add`**.** Procedure `swap_add` retrieves its arguments from the stack frame for `caller`.

is restored and the procedure returns.

The following is the setup code for `swap_add`. Recall that the `call` instruction will already push the return address on the stack.

```
     Setup code in swap_add
1 swap_add:
2   pushl %ebp                  Save old %ebp
3   movl %esp,%ebp              Set %ebp as frame pointer
4   pushl %ebx                  Save %ebx
```

Procedure `swap_add` requires register `%ebx` for temporary storage. Since this is a callee save register, it pushes the old value on the stack as part of the stack frame setup.

The following is the body code for `swap_add`:

```
     Body code in swap_add
1   movl 8(%ebp),%edx           Get xp
2   movl 12(%ebp),%ecx          Get yp
3   movl (%edx),%ebx            Get x
4   movl (%ecx),%eax            Get y
5   movl %eax,(%edx)            Store y at *xp
6   movl %ebx,(%ecx)            Store x at *yp
7   addl %ebx,%eax              Set return value = x+y
```

This code retrieves its arguments from the stack frame for `caller`. Since the frame pointer has shifted, the locations of these arguments has shifted from positions -12 and -16 relative to the old value of `%ebp` to positions $+12$ and $+8$ relative to new value of `%ebp`. Observe that the sum of variables `x` and `y` is stored in register `%eax` to be passed as the returned value.

The following is the finishing code for `swap_add`:

```
     Finishing code in swap_add
1   popl %ebx                   Restore %ebx
2   movl %ebp,%esp              Restore %esp
3   popl %ebp                   Restore %ebp
4   ret                         Return to caller
```

This code simply restores the values of the three registers %ebx, %esp, and %ebp, and then executes the ret instruction. Note that instructions F2 and F3 could be replaced by a single leave instruction. Different versions of GCC seem to have different preferences in this regard.

The following code in caller comes immediately after the instruction calling swap_add:

```
1   movl %eax,%edx             Resume here
```

Upon return from swap_add, procedure caller will resume execution with this instruction. Observe that this instruction copies the return value from %eax to a different register.

Practice Problem 3.16:

Given the following C function:

```
1 int proc(void)
2 {
3   int x,y;
4   scanf("%x %x", &y, &x);
5   return x-y;
6 }
```

GCC generates the following assembly code

```
1 proc:
2   pushl %ebp
3   movl %esp,%ebp
4   subl $24,%esp
5   addl $-4,%esp
6   leal -4(%ebp),%eax
7   pushl %eax
8   leal -8(%ebp),%eax
9   pushl %eax
10  pushl $.LC0       Pointer to string "%x %x"
11  call scanf
    Diagram stack frame at this point
12  movl -8(%ebp),%eax
13  movl -4(%ebp),%edx
14  subl %eax,%edx
15  movl %edx,%eax
16  movl %ebp,%esp
17  popl %ebp
18  ret
```

Assume that procedure proc starts executing with the following register values:

Register	Value
%esp	0x800040
%ebp	0x800060

code/asm/fib.c

```
1 int fib_rec(int n)
2 {
3     int prev_val, val;
4
5     if (n <= 2)
6         return 1;
7     prev_val = fib_rec(n-2);
8     val = fib_rec(n-1);
9     return prev_val + val;
10 }
```

code/asm/fib.c

Figure 3.19: **C Code for Recursive Fibonacci Program.**

Suppose `proc` calls `scanf` (line 12), and that `scanf` reads values `0x46` and `0x53` from the standard input. Assume that the string `"%x %x"` is stored at memory location `0x300070`.

A. What value does `%ebp` get set to on line 3?

B. At what addresses are local variables `x` and `y` stored?

C. What is the value of `%esp` at line 11?

D. Draw a diagram of the stack frame for `proc` right after `scanf` returns. Include as much information as you can about the addresses and the contents of the stack frame elements.

E. Indicate the regions of the stack frame that are not used by `proc` (these wasted areas are allocated to improve the cache performance).

3.7.5 Recursive Procedures

The stack and linkage conventions described in the previous section allow procedures to call themselves recursively. Since each call has its own private space on the stack, the local variables of the multiple outstanding calls do not interfere with one another. Furthermore, the stack discipline naturally provides the proper policy for allocating local storage when the procedure is called and deallocating it when it returns.

Figure 3.19 shows the C code for a recursive Fibonacci function. (Note that this code is very inefficient—we intend it to be an illustrative example, not a clever algorithm). The complete assembly code is shown as well in Figure 3.20.

Although there is a lot of code, it is worth studying closely. The set-up code (lines 2 to 6) creates a stack frame containing the old version of `%ebp`, 16 unused bytes,[2] and saved values for the callee save registers `%esi` and `%ebx`, as diagrammed on the left side of Figure 3.21. It then uses register `%ebx` to hold the procedure parameter `n` (line 7). In the event of a terminal condition, the code jumps to line 22, where the return value is set to 1.

[2]It is unclear why the C compiler allocates so much unused storage on the stack for this function.

```
1  fib_rec:
       Setup code
2    pushl %ebp                 Save old %ebp
3    movl %esp,%ebp             Set %ebp as frame pointer
4    subl $16,%esp              Allocate 16 bytes on stack
5    pushl %esi                 Save %esi (offset -20)
6    pushl %ebx                 Save %ebx (offset -24)

       Body code
7    movl 8(%ebp),%ebx          Get n
8    cmpl $2,%ebx               Compare n:2
9    jle .L24                   if <=, goto terminate
10   addl $-12,%esp             Allocate 12 bytes on stack
11   leal -2(%ebx),%eax         Compute n-2
12   pushl %eax                 Push as argument
13   call fib_rec               Call fib_rec(n-2)
14   movl %eax,%esi             Store result in %esi
15   addl $-12,%esp             Allocate 12 bytes to stack
16   leal -1(%ebx),%eax         Compute n-1
17   pushl %eax                 Push as argument
18   call fib_rec               Call fib_rec(n-1)
19   addl %esi,%eax             Compute val+nval
20   jmp .L25                   Go to done

       Terminal condition
21 .L24:                      terminate:
22   movl $1,%eax                 Return value 1

       Finishing code
23 .L25:                      done:
24   leal -24(%ebp),%esp          Set stack to offset -24
25   popl %ebx                    Restore %ebx
26   popl %esi                    Restore %esi
27   movl %ebp,%esp               Restore stack pointer
28   popl %ebp                    Restore %ebp
29   ret                          Return
```

Figure 3.20: **Assembly Code for the Recursive Fibonacci Program in Figure 3.19.**

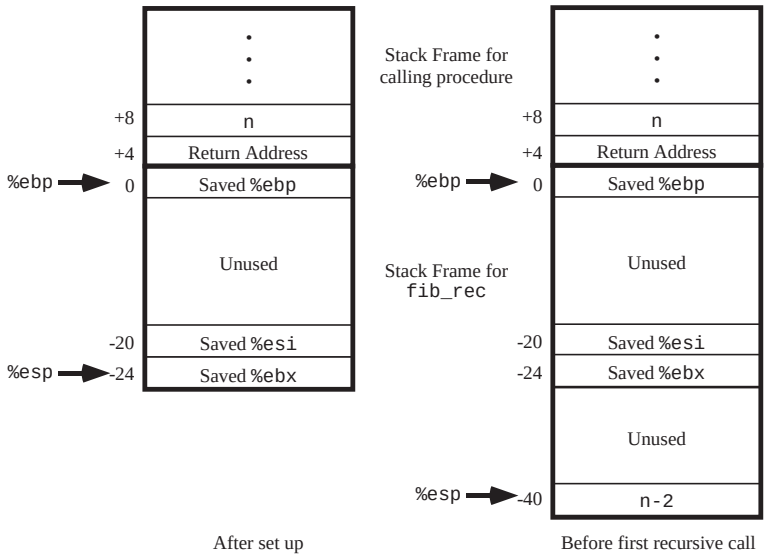


Figure 3.21: **Stack Frame for Recursive Fibonacci Function.** State of frame is shown after initial set up (left), and just before the first recursive call (right).

For the nonterminal condition, instructions 10 to 12 set up the first recursive call. This involves allocating 12 bytes on the stack that are never used, and then pushing the computed value `n-2`. At this point, the stack frame will have the form shown on the right side of Figure 3.21. It then makes the recursive call, which will trigger a number of calls that allocate stack frames, perform operations on local storage, and so on. As each call returns, it deallocates any stack space and restores any modified callee save registers. Thus, when we return to the current call at line 14 we can assume that register `%eax` contains the value returned by the recursive call, and that register `%ebx` contains the value of function parameter `n`. The returned value (local variable `prev_val` in the C code) is stored in register `%esi` (line 14). By using a callee save register, we can be sure that this value will still be available after the second recursive call.

Instructions 15 to 17 set up the second recursive call. Again it allocates 12 bytes that are never used, and pushes the value of `n-1`. Following this call (line 18), the computed result will be in register `%eax`, and we can assume that the result of the previous call is in register `%esi`. These are added to give the return value (instruction 19).

The completion code restores the registers and deallocates the stack frame. It starts (line 24) by setting the stack frame to the location of the saved value of `%ebx`. Observe that by computing this stack position relative to the value of `%ebp`, the computation will be correct regardless of whether or not the terminal condition was reached.

3.8 Array Allocation and Access

Arrays in C are one means of aggregating scalar data into larger data types. C uses a particularly simple implementation of arrays, and hence the translation into machine code is fairly straightforward. One unusual feature of C is that one can generate pointers to elements within arrays and perform arithmetic with these

pointers. These are translated into address computations in assembly code.

Optimizing compilers are particularly good at simplifying the address computations used by array indexing. This can make the correspondence between the C code and its translation into machine code somewhat difficult to decipher.

3.8.1 Basic Principles

For data type T and integer constant N, the declaration

```
T A[N];
```

has two effects. First, it allocates a contiguous region of $L \cdot N$ bytes in memory, where L is the size (in bytes) of data type T. Let us denote the starting location as x_{A}. Second, it introduces an identifier A that can be used as a pointer to the beginning of the array. The value of this pointer will be x_{A}. The array elements can be accessed using an integer index ranging between 0 and $N - 1$. Array element i will be stored at address $x_{\mathsf{A}} + L \cdot i$.

As examples, consider the following declarations:

```
char    A[12];
char   *B[8];
double  C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element Size	Total Size	Start Address	Element i
A	1	12	x_{A}	$x_{\mathsf{A}} + i$
B	4	32	x_{B}	$x_{\mathsf{B}} + 4i$
C	8	48	x_{C}	$x_{\mathsf{C}} + 8i$
D	4	20	x_{D}	$x_{\mathsf{D}} + 4i$

Array A consists of 12 single-byte (`char`) elements. Array C consists of 6 double-precision floating-point values, each requiring 8 bytes. B and D are both arrays of pointers, and hence the array elements are 4 bytes each.

The memory referencing instructions of IA32 are designed to simplify array access. For example, suppose E is an array of `int`'s, and we wish to compute `E[i]` where the address of E is stored in register `%edx` and `i` is stored in register `%ecx`. Then the instruction:

```
movl (%edx,%ecx,4),%eax
```

will perform the address computation $x_{\mathsf{E}} + 4i$, read that memory location, and store the result in register `%eax`. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the primitive data types.

Practice Problem 3.17:

Consider the following declarations:

```
short         S[7];
short        *T[3];
short       **U[6];
long double   V[8];
long double  *W[4];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element Size	Total Size	Start Address	Element i
S			$x_{\mathtt{S}}$	
T			$x_{\mathtt{T}}$	
U			$x_{\mathtt{U}}$	
V			$x_{\mathtt{V}}$	
W			$x_{\mathtt{W}}$	

3.8.2 Pointer Arithmetic

C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer. That is, if `p` is a pointer to data of type T, and the value of `p` is $x_{\mathtt{p}}$, then the expression `p+i` has value $x_{\mathtt{p}} + L \cdot i$ where L is the size of data type T.

The unary operators `&` and `*` allow the generation and dereferencing of pointers. That is, for an expression *Expr* denoting some object, `&`*Expr* is a pointer giving the address of the object. For an expression *Addr-Expr* denoting an address, `*`*Addr-Expr* gives the value at that address. The expressions *Expr* and `*&`*Expr* are therefore equivalent. The array subscripting operation can be applied to both arrays and pointers. The array reference `A[i]` is identical to the expression `*(A+i)`. It computes the address of the ith array element and then accesses this memory location.

Expanding on our earlier example, suppose the starting address of integer array `E` and integer index `i` are stored in registers `%edx` and `%ecx`, respectively. The following are some expressions involving `E`. We also show an assembly code implementation of each expression, with the result being stored in register `%eax`.

Expression	Type	Value	Assembly Code
`E`	`int *`	$x_{\mathtt{E}}$	`movl %edx,%eax`
`E[0]`	`int`	$Mem[x_{\mathtt{E}}]$	`movl (%edx),%eax`
`E[i]`	`int`	$Mem[x_{\mathtt{E}} + 4i]$	`movl (%edx,%ecx,4),%eax`
`&E[2]`	`int *`	$x_{\mathtt{E}} + 8$	`leal 8(%edx),%eax`
`E+i-1`	`int *`	$x_{\mathtt{E}} + 4i - 4$	`leal -4(%edx,%ecx,4),%eax`
`*(&E[i]+i)`	`int`	$Mem[x_{\mathtt{E}} + 4i + 4i]$	`movl (%edx,%ecx,8),%eax`
`&E[i]-E`	`int`	i	`movl %ecx,%eax`

In these examples, the `leal` instruction is used to generate an address, while `movl` is used to reference memory (except in the first case, where it copies an address). The final example shows that one can compute the difference of two pointers within the same data structure, with the result divided by the size of the data type.

Practice Problem 3.18:

Suppose the address of `short` integer array `S` and integer index `i` are stored in registers `%edx` and `%ecx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in register `%eax` if it a pointer and register element `%ax` if it is a `short` integer.

Expression	Type	Value	Assembly Code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

3.8.3 Arrays and Loops

Array references within loops often have very regular patterns that can be exploited by an optimizing compiler. For example, the function `decimal5` shown in Figure 3.22(a) computes the integer represented by an array of 5 decimal digits. In converting this to assembly code, the compiler generates code similar to that shown in Figure 3.22(b) as C function `decimal5_opt`. First, rather than using a loop index `i`, it uses pointer arithmetic to step through successive array elements. It computes the address of the final array element and uses a comparison to this address as the loop test. Finally, it can use a `do-while` loop since there will be at least one loop iteration.

The assembly code shown in Figure 3.22(c) shows a further optimization to avoid the use of an integer multiply instruction. In particular, it uses `leal` (line 5) to compute `5*val` as `val+4*val`. It then uses `leal` with a scaling factor of 2 (line 7) to scale to `10*val`.

Aside: Why avoid integer multiply?
In older models of the IA32 processor, the integer multiply instruction took as many as 30 clock cycles, and so compilers try to avoid it whenever possible. In the most recent models it requires only 3 clock cycles, and therefore these optimizations are not warranted. **End Aside.**

3.8.4 Nested Arrays

The general principles of array allocation and referencing hold even when we create arrays of arrays. For example, the declaration:

```
int A[4][3];
```

is equivalent to the declaration:

```
typedef int row3_t[3];
row3_t A[4];
```

code/asm/decimal5.c

```
1 int decimal5(int *x)
2 {
3     int i;
4     int val = 0;
5
6     for (i = 0; i < 5; i++)
7         val = (10 * val) + x[i];
8
9     return val;
10 }
```

code/asm/decimal5.c

(a) Original C code

code/asm/decimal5.c

```
1 int decimal5_opt(int *x)
2 {
3     int val = 0;
4     int *xend = x + 4;
5
6     do {
7         val = (10 * val) + *x;
8         x++;
9     } while (x <= xend);
10
11    return val;
12 }
```

code/asm/decimal5.c

(b) Equivalent pointer code

```
    Body code
1   movl 8(%ebp),%ecx
2   xorl %eax,%eax
3   leal 16(%ecx),%ebx
4 .L12:
5   leal (%eax,%eax,4),%edx
6   movl (%ecx),%eax
7   leal (%eax,%edx,2),%eax
8   addl $4,%ecx
9   cmpl %ebx,%ecx
10  jbe .L12
```

```
Get base addr of array x
val = 0;
xend = x+4 (16 bytes = 4 double words)
loop:
Compute 5*val
Compute *x
Compute *x + 2*(5*val)
x++
Compare x:xend
if <=, goto loop:
```

(c) Corresponding assembly code.

Figure 3.22: **C and Assembly Code for Array Loop Example.** The compiler generates code similar to the pointer code shown in `decimal5_opt`.

Data type `row3_t` is defined to be an array of three integers. Array `A` contains four such elements, each requiring 12 bytes to store the three integers. The total array size is then $4 \cdot 4 \cdot 3 = 48$ bytes.

Array `A` can also be viewed as a two-dimensional array with four rows and three columns, referenced as `A[0][0]` through `A[3][2]`. The array elements are ordered in memory in "row major" order, meaning all elements of row 0, followed by all elements of row 1, and so on.

Element	Address
`A[0][0]`	$x_{\mathtt{A}}$
`A[0][1]`	$x_{\mathtt{A}} + 4$
`A[0][2]`	$x_{\mathtt{A}} + 8$
`A[1][0]`	$x_{\mathtt{A}} + 12$
`A[1][1]`	$x_{\mathtt{A}} + 16$
`A[1][2]`	$x_{\mathtt{A}} + 20$
`A[2][0]`	$x_{\mathtt{A}} + 24$
`A[2][1]`	$x_{\mathtt{A}} + 28$
`A[2][2]`	$x_{\mathtt{A}} + 32$
`A[3][0]`	$x_{\mathtt{A}} + 36$
`A[3][1]`	$x_{\mathtt{A}} + 40$
`A[3][2]`	$x_{\mathtt{A}} + 44$

This ordering is a consequence of our nested declaration. Viewing `A` as an array of four elements, each of which is an array of three `int`'s, we first have `A[0]` (i.e., row 0), followed by `A[1]`, and so on.

To access elements of multidimensional arrays, the compiler generates code to compute the offset of the desired element and then uses a `movl` instruction using the start of the array as the base address and the (possibly scaled) offset as an index. In general, for an array declared as:

T `D[`R`][`C`];`

array element `D[i][j]` is at memory address $x_{\mathtt{D}} + L(C \cdot i + j)$, where L is the size of data type T in bytes.

As an example, consider the 4×3 integer array `A` defined earlier. Suppose register `%eax` contains $x_{\mathtt{A}}$, that `%edx` holds `i`, and `%ecx` holds `j`. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
     A in %eax, i in %edx, j in %ecx
1    sall $2,%ecx                        j * 4
2    leal (%edx,%edx,2),%edx             i * 3
3    leal (%ecx,%edx,4),%edx             j * 4 + i * 12
4    movl (%eax,%edx),%eax       Read Mem[x_A + 4(3 · i + j)]
```

Practice Problem 3.19:

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
1 int mat1[M][N];
```

```
2 int mat2[N][M];
3
4 int sum_element(int i, int j)
5 {
6   return mat1[i][j] + mat2[j][i];
7 }
```

In compiling this program, GCC generates the following assembly code:

```
1   movl 8(%ebp),%ecx
2   movl 12(%ebp),%eax
3   leal 0(,%eax,4),%ebx
4   leal 0(,%ecx,8),%edx
5   subl %ecx,%edx
6   addl %ebx,%eax
7   sall $2,%eax
8   movl mat2(%eax,%ecx,4),%eax
9   addl mat1(%ebx,%edx,4),%eax
```

Use your reverse engineering skills to determine the values of `M` and `N` based on this assembly code.

3.8.5 Fixed Size Arrays

The C compiler is able to make many optimizations for code operating on multi-dimensional arrays of fixed size. For example, suppose we declare data type `fix_matrix` to be 16×16 arrays of integers as follows:

```
1 #define N 16
2 typedef int fix_matrix[N][N];
```

The code in Figure 3.23(a) computes element i, k of the product of matrices `A` and `B`. The C compiler generates code similar to that shown in Figure 3.23(b). This code contains a number of clever optimizations. It recognizes that the loop will access the elements of array `A` as `A[i][0]`, `A[i][1]`, ..., `A[i][15]` in sequence. These elements occupy adjacent positions in memory starting with the address of array element `A[i][0]`. The program can therefore use a pointer variable `Aptr` to access these successive locations. The loop will access the elements of array `B` as `B[0][k]`, `B[1][k]`, ..., `B[15][k]` in sequence. These elements occupy positions in memory starting with the address of array element `B[0][k]` and spaced 64 bytes apart. The program can therefore use a pointer variable `Bptr` to access these successive locations. In C, this pointer is shown as being incremented by 16, although in fact the actual pointer is incremented by $4 \cdot 16 = 64$. Finally, the code can use a simple counter to keep track of the number of iterations required.

We have shown the C code `fix_prod_ele_opt` to illustrate the optimizations made by the C compiler in generating the assembly. The actual assembly code for the loop is shown below.

```
   Aptr is in %edx, Bptr in %ecx, result in %esi, cnt in %ebx
1 .L23:                          loop:
2   movl (%edx),%eax               Compute t = *Aptr
```

code/asm/array.c

```
1 #define N 16
2 typedef int fix_matrix[N][N];
3
4 /* Compute i,k of fixed matrix product */
5 int fix_prod_ele (fix_matrix A, fix_matrix B,  int i, int k)
6 {
7     int j;
8     int result = 0;
9
10    for (j = 0; j < N; j++)
11        result += A[i][j] * B[j][k];
12
13    return result;
14 }
```

code/asm/array.c

(a) Original C code

code/asm/array.c

```
1 /* Compute i,k of fixed matrix product */
2 int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
3 {
4     int *Aptr = &A[i][0];
5     int *Bptr = &B[0][k];
6     int cnt = N - 1;
7     int result = 0;
8
9     do {
10        result += (*Aptr) * (*Bptr);
11        Aptr += 1;
12        Bptr += N;
13        cnt--;
14    } while (cnt >= 0);
15
16    return result;
17 }
```

code/asm/array.c

(b) Optimized C code.

Figure 3.23: **Original and Optimized Code to Compute Element i, k of Matrix Product for Fixed Length Arrays.** The compiler performs these optimizations automatically.

```
3   imull (%ecx),%eax          Compute v = *Bptr * t
4   addl %eax,%esi             Add v result
5   addl $64,%ecx              Add 64 to Bptr
6   addl $4,%edx               Add 4 to Aptr
7   decl %ebx                  Decrement cnt
8   jns .L23                   if >=, goto loop
```

Note that in the above code, all pointer increments are scaled by a factor of 4 relative to the C code.

Practice Problem 3.20:

The following C code sets the diagonal elements of a fixed-size array to `val`

```
1 /* Set all diagonal elements to val */
2 void fix_set_diag(fix_matrix A, int val)
3 {
4   int i;
5   for (i = 0; i < N; i++)
6     A[i][i] = val;
7 }
```

When compiled GCC generates the following assembly code:

```
1   movl 12(%ebp),%edx
2   movl 8(%ebp),%eax
3   movl $15,%ecx
4   addl $1020,%eax
5   .p2align 4,,7          Added to optimize cache performance
6 .L50:
7   movl %edx,(%eax)
8   addl $-68,%eax
9   decl %ecx
10  jns .L50
```

Create a C code program `fix_set_diag_opt` that uses optimizations similar to those in the assembly code, in the same style as the code in Figure 3.23(b).

3.8.6 Dynamically Allocated Arrays

C only supports multidimensional arrays where the sizes (with the possible exception of the first dimension) are known at compile time. In many applications, we require code that will work for arbitrary size arrays that have been dynamically allocated. For these we must explicitly encode the mapping of multidimensional arrays into one-dimensional ones. We can define a data type `var_matrix` as simply an `int *`:

```
typedef int *var_matrix;
```

To allocate and initialize storage for an $n \times n$ array of integers, we use the Unix library function `calloc`:

```
1 var_matrix new_var_matrix(int n)
2 {
3     return (var_matrix) calloc(sizeof(int), n * n);
4 }
```

The `calloc` function (documented as part of ANSI C [30, 37]) takes two arguments: the size of each array element and the number of array elements required. It attempts to allocate space for the entire array. If successful, it initializes the entire region of memory to 0s and returns a pointer to the first byte. If insufficient space is available, it returns null.

> **New to C?**
> In C, storage on the heap (a pool of memory available for storing data structures) is allocated using the library function `malloc` or its cousin `calloc`. Their effect is similar to that of the `new` operation in C++ and Java. Both C and C++ require the program to explictly free allocated space using the
>
> `free` function. In Java, freeing is performed automatically by the run-time system via a process called *garbage collection*, as will be discussed in Chapter 10. **End**

We can then use the indexing computation of row-major ordering to determine the position of element i, j of the matrix as $i \cdot n + j$:

```
1 int var_ele(var_matrix A, int i, int j, int n)
2 {
3     return A[(i*n) + j];
4 }
```

This referencing translates into the following assembly code:

```
1   movl 8(%ebp),%edx          Get A
2   movl 12(%ebp),%eax         Get i
3   imull 20(%ebp),%eax        Compute n*i
4   addl 16(%ebp),%eax         Compute n*i + j
5   movl (%edx,%eax,4),%eax    Get A[i*n + j]
```

Comparing this code to that used to index into a fixed-size array, we see that the dynamic version is somewhat more complex. It must use a multiply instruction to scale i by n, rather than a series of shifts and adds. In modern processors, this multiplication does not incur a significant performance penalty.

In many cases, the compiler can simplify the indexing computations for variable-sized arrays using the same principles as we saw for fixed-size ones. For example, Figure 3.24(a) shows C code to compute element i, k of the product of two variable-sized matrices `A` and `B`. In Figure 3.24(b) we show an optimized version derived by reverse engineering the assembly code generated by compiling the original version. The compiler is able to eliminate the integer multiplications `i*n` and `j*n` by exploiting the sequential access pattern resulting from the loop structure. In this case, rather than generating a pointer variable `Bptr`, the compiler creates an integer variable we call `nTjPk`, for "`n` Times `j` Plus `k`," since its value equals `n*j+k` relative to the original code. Initially `nTjPk` equals `k`, and it is incremented by `n` on each iteration.

The assembly code for the loop is shown below. The registers values are: `%edx` holds `cnt`, `%ebx` holds `Aptr`, `%ecx` holds `nTjPk`, and `%esi` holds `result`.

code/asm/array.c

```
1 typedef int *var_matrix;
2
3 /* Compute i,k of variable matrix product */
4 int var_prod_ele(var_matrix A, var_matrix B, int i, int k, int n)
5 {
6     int j;
7     int result = 0;
8
9     for (j = 0; j < n; j++)
10        result += A[i*n + j] * B[j*n + k];
11
12    return result;
13 }
```

code/asm/array.c

(a) Original C code

code/asm/array.c

```
1 /* Compute i,k of variable matrix product */
2 int var_prod_ele_opt(var_matrix A, var_matrix B, int i, int k, int n)
3 {
4     int *Aptr = &A[i*n];
5     int nTjPk = n;
6     int cnt = n;
7     int result = 0;
8
9     if (n <= 0)
10        return result;
11
12    do {
13        result += (*Aptr) * B[nTjPk];
14        Aptr += 1;
15        nTjPk += n;
16        cnt--;
17    } while (cnt);
18
19    return result;
20 }
```

code/asm/array.c

(b) Optimized C code

Figure 3.24: **Original and Optimized Code to Compute Element i, k of Matrix Product for Variable Length Arrays.** The compiler performs these optimizations automatically.

```
1 .L37:                                  loop:
2   movl 12(%ebp),%eax                     Get B
3   movl (%ebx),%edi                       Get *Aptr
4   addl $4,%ebx                           Increment Aptr
5   imull (%eax,%ecx,4),%edi               Multiply by B[nTjPk]
6   addl %edi,%esi                         Add to result
7   addl 24(%ebp),%ecx                     Add n to nTjPk
8   decl %edx                              Decrement cnt
9   jnz .L37                               If cnt <> 0, goto loop
```

Observe that in the above code, variables `B` and `n` must be retrieved from memory on each iteration. This is an example of *register spilling*. There are not enough registers to hold all of the needed temporary data, and hence the compiler must keep some local variables in memory. In this case the compiler chose to spill variables `B` and `n` because they are read only—they do not change value within the loop. Spilling is a common problem for IA32, since the processor has so few registers.

3.9 Heterogeneous Data Structures

C provides two mechanisms for creating data types by combining objects of different types. Structures, declared using the keyword `struct`, aggregate multiple objects into a single one. Unions, declared using the keyword `union`, allow an object to be referenced using any of a number of different types.

3.9.1 Structures

The C `struct` declaration creates a data type that groups objects of possibly different types into a single object. The different components of a structure are referenced by names. The implementation of structures is similar to that of arrays in that all of the components of a structure are stored in a contiguous region of memory, and a pointer to a structure is the address of its first byte. The compiler maintains information about each structure type indicating the byte offset of each field. It generates references to structure elements using these offsets as displacements in memory referencing instructions.

New to C?
The `struct` data type constructor is the closest thing C provides to the objects of C++ and Java. It allows the programmer to keep information about some entity in a single data structure, and reference that information with names.

For example, a graphics program might represent a rectangle as a structure:

```
struct rect {
    int llx;    /* X coordinate of lower-left corner */
    int lly;    /* Y coordinate of lower-left corner */
    int color;  /* Coding of color                   */
    int width;  /* Width (in pixels)                 */
    int height; /* Height (in pixels)                */
};
```

We could declare a variable `r` of type `struct rect` and set its field values as follows:

```
    struct rect r;
    r.llx = r.lly = 0;
    r.color = 0xFF00FF;
    r.width = 10;
    r.height = 20;
```

where the expression `r.llx` selects field `llx` of structure `r`.

It is common to pass pointers to structures from one place to another rather than copying them. For example, the following function computes the area of a rectangle, where a pointer to the rectange `struct` is passed to the function:

```
int area(struct rect *rp)
{
    return (*rp).width * (*rp).height;
}
```

The expression `(*rp).width` dereferences the pointer and selects the `width` field of the resulting structure. Parentheses are required, because the compiler would interpret the expression `*rp.width` as `*(rp.width)`, which is not valid. This combination of dereferencing and field selection is so common that C provides an alternative notation using `->`. That is, `rp->width` is equivalent to the expression `(*rp).width`. For example, we could write a function that rotates a rectangle left by 90 degrees as

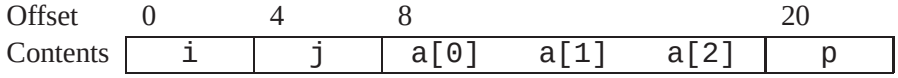
```
void rotate_left(struct rect *rp)
{
    /* Exchange width and height */
    int t      = rp->height;
    rp->height = rp->width;
    rp->width  = t;
}
```

The objects of C++ and Java are more elaborate than structures in C, in that they also associate a set of *methods* with an object that can be invoked to perform computation. In C, we would simply write these as ordinary functions, such as the functions `area` and `rotate_left` shown above. **End**

As an example, consider the following structure declaration:

```
struct rec {
    int i;
    int j;
    int a[3];
    int *p;
};
```

This structure contains four fields: two 4-byte `int`'s, an array consisting of three 4-byte `int`'s, and a 4-byte integer pointer, giving a total of 24 bytes:



Observe that array `a` is embedded within the structure. The numbers along the top of the diagram give the byte offsets of the fields from the beginning of the structure.

To access the fields of a structure, the compiler generates code that adds the appropriate offset to the address of the structure. For example, suppose variable `r` of type `struct rec *` is in register `%edx`. Then the following code copies element `r->i` to element `r->j`:

```
1   movl (%edx),%eax                  Get r->i
2   movl %eax,4(%edx)                 Store in r->j
```

Since the offset of field `i` is 0, the address of this field is simply the value of `r`. To store into field `j`, the code adds offset 4 to the address of `r`.

To generate a pointer to an object within a structure, we can simply add the field's offset to the structure address. For example, we can generate the pointer `&(r->a[1])` by adding offset $8+4\cdot 1 = 12$. For pointer `r` in register `%edx` and integer variable `i` in register `%eax`, we can generate the pointer value `&(r->a[i])` with the single instruction:

```
     r in %eax, i in %edx
1   leal 8(%eax,%edx,4),%ecx    %ecx = &r->a[i]
```

As a final example, the following code implements the statement:

```
r->p = &r->a[r->i + r->j];
```

starting with `r` in register `%edx`:

```
1   movl 4(%edx),%eax                 Get r->j
2   addl (%edx),%eax                  Add r->i
3   leal 8(%edx,%eax,4),%eax          Compute &r->[r->i + r->j]
4   movl %eax,20(%edx)                Store in r->p
```

As these examples show, the selection of the different fields of a structure is handled completely at compile time. The machine code contains no information about the field declarations or the names of the fields.

Practice Problem 3.21:

Consider the following structure declaration.

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

This declaration illustrates that one structure can be embedded within another, just as arrays can be embedded within structures, and arrays can be embedded within arrays.

The following procedure (with some expressions omitted) operates on this structure:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = _________;
    sp->p     = _________;
    sp->next  = _________;
}
```

A. What are the offsets (in bytes) of the following fields:

```
   p:
 s.x:
 s.y:
next:
```

B. How many total bytes does the structure require?

C. The compiler generates the following assembly code for the body of `sp_init`:

```
1   movl 8(%ebp),%eax
2   movl 8(%eax),%edx
3   movl %edx,4(%eax)
4   leal 4(%eax),%edx
5   movl %edx,(%eax)
6   movl %eax,12(%eax)
```

Based on this, fill in the missing expressions in the code for `sp_init`.

3.9.2 Unions

Unions provide a way to circumvent the type system of C, allowing a single object to be referenced according to multiple types. The syntax of a union declaration is identical to that for structures, but its semantics are very different. Rather than having the different fields reference different blocks of memory, they all reference the same block.

Consider the following declarations:

```
struct S3 {
    char c;
    int i[2];
    double v;
};

union U3 {
    char c;
    int i[2];
    double v;
};
```

The offsets of the fields, as well as the total size of data types `S3` and `U3`, are:

Type	c	i	v	Size
S3	0	4	12	20
U3	0	0	0	8

(We will see shortly why `i` has offset 4 in `S3` rather than 1). For pointer `p` of type `union U3 *`, references `p->c`, `p->i[0]`, and `p->v` would all reference the beginning of the data structure. Observe also that the overall size of a union equals the maximum size of any of its fields.

Unions can be useful in several contexts. However, they can also lead to nasty bugs, since they bypass the safety provided by the C type system. One application is when we know in advance that the use of two different fields in a data structure will be mutually exclusive. Then declaring these two fields as part of a union rather than a structure will reduce the total space allocated.

For example, suppose we want to implement a binary tree data structure where each leaf node has a `double` data value, while each internal node has pointers to two children, but no data. If we declare this as:

```
struct NODE {
    struct NODE *left;
    struct NODE *right;
    double data;
};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as:

```
union NODE {
    struct {
        union NODE *left;
        union NODE *right;
    } internal;
    double data;
};
```

then every node will require just 8 bytes. If `n` is a pointer to a node of type `union NODE *`, we would reference the data of a leaf node as `n->data`, and the children of an internal node as `n->internal.left` and `n->internal.right`.

With this encoding, however, there is no way to determine whether a given node is a leaf or an internal node. A common method is to introduce an additional tag field:

```
struct NODE {
    int is_leaf;
    union {
        struct {
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data;
    } info;
};
```

where the field `is_leaf` is 1 for a leaf node and is 0 for an internal node. This structure requires a total of 12 bytes: 4 for `is_leaf`, and either 4 each for `info.internal.left` and `info.internal.right`, or 8 for `info.data`. In this case, the savings gain of using a union is small relative to the awkwardness of the resulting code. For data structures with more fields, the savings can be more compelling.

Unions can also be used to access the bit patterns of different data types. For example, the following code returns the bit representation of a `float` as an `unsigned`:

```
1 unsigned float2bit(float f)
2 {
3     union {
4         float f;
5         unsigned u;
6     } temp;
7     temp.f = f;
8     return temp.u;
9 };
```

In this code we store the argument in the union using one data type, and access it using another. Interestingly, the code generated for this procedure is identical to that for the procedure:

```
1 unsigned copy(unsigned u)
2 {
3     return u;
4 }
```

The body of both procedures is just a single instruction:

```
1   movl 8(%ebp),%eax
```

This demonstrates the lack of type information in assembly code. The argument will be at offset 8 relative to `%ebp` regardless of whether it is a `float` or an `unsigned`. The procedure simply copies its argument as the return value without modifying any bits.

When using unions combining data types of different sizes, byte ordering issues can become important. For example suppose we write a procedure that will create an 8-byte `double` using the bit patterns given by two 4-byte `unsigned`'s:

```
1 double bit2double(unsigned word0, unsigned word1)
2 {
3     union {
4         double d;
5         unsigned u[2];
6     } temp;
7
8     temp.u[0] = word0;
9     temp.u[1] = word1;
10    return temp.d;
11 }
```

On a little-endian machine such as IA32, argument `word0` will become the low-order four bytes of `d`, while `word1` will become the high-order four bytes. On a big-endian machine, the role of the two arguments will be reversed.

Practice Problem 3.22:

Consider the following union declaration.

```
union ele {
    struct {
        int *p;
        int y;
    } e1;
    struct {
        int x;
        union ele *next;
    } e2;
};
```

This declaration illustrates that structures can be embedded within unions.

The following procedure (with some expressions omitted) operates on link list having these unions as list elements:

```
void proc (union ele *up)
{
    up->__________ = *(up->__________) - up->__________;
}
```

A. What would be the offsets (in bytes) of the following fields:

`e1.p`:
`e1.y`:
`e2.x`:
`e2.next`:

B. How many total bytes would the structure require?

C. The compiler generates the following assembly code for the body of `proc`:

```
1   movl 8(%ebp),%eax
2   movl 4(%eax),%edx
3   movl (%edx),%ecx
4   movl %ebp,%esp
5   movl (%eax),%eax
6   movl (%ecx),%ecx
7   subl %eax,%ecx
8   movl %ecx,4(%edx)
```

Based on this, fill in the missing expressions in the code for `proc`. [**Hint:** Some union references can have ambiguous interpretations. These ambiguities get resolved as you see where the references lead. There is only one answer that does not perform any casting and does not violate any type constraints.]

3.10 Alignment

Many computer systems place restrictions on the allowable addresses for the primitive data types, requiring that the address for some type of object must be a multiple of some value k (typically 2, 4, or 8). Such *alignment restrictions* simplify the design of the hardware forming the interface between the processor and the memory system. For example, suppose a processor always fetches 8 bytes from memory with an address that must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address be a multiple of 8, then the value can be read or written with a single memory operation. Otherwise, we may need to perform two memory accesses, since the object might be split across two 8-byte memory blocks.

The IA32 hardware will work correctly regardless of the alignment of data. However, Intel recommends that data be aligned to improve memory system performance. Linux follows an alignment policy where 2-byte data types (e.g., `short`) must have an address that is a multiple of 2, while any larger data types (e.g., `int`, `int *`, `float`, and `double`) must have an address that is a multiple of 4. Note that this requirement means that the least significant bit of the address of an object of type `short` must equal 0. Similarly, any object of type `int`, or any pointer, must be at an address having the low-order two bits equal to 0.

> **Aside: Alignment with Microsoft Windows.**
> Microsoft Windows requires a stronger alignment requirement—any k-byte (primitive) object must have an address that is a multiple of k. In particular, it requires that the address of a `double` be a multiple of 8. This requirement enhances the memory performance at the expense of some wasted space. The design decision made in Linux was probably good for the i386, back when memory was scarce and memory busses were only 4 bytes wide. With modern processors, Microsoft's alignment is a better design decision.
>
> The command line flag `-malign-double` causes GCC on Linux to use 8-byte alignment for data of type `double`. This will lead to improved memory performance, but it can cause incompatibilities when linking with library code that has been compiled assuming a 4-byte alignment. **End Aside.**

Alignment is enforced by making sure that every data type is organized and allocated in such a way that every object within the type satisfies its alignment restrictions. The compiler places directives in the assembly code indicating the desired alignment for global data. For example, the assembly code declaration of the jump table on page 131 contains the following directive on line 2:

```
.align 4
```

This ensures that the data following it (in this case the start of the jump table) will start with an address that is a multiple of 4. Since each table entry is 4 bytes long, the successive elements will obey the 4-byte alignment restriction.

Library routines that allocate memory, such as `malloc`, must be designed so that they return a pointer that satisfies the worst-case alignment restriction for the machine it is running on, typically 4 or 8.

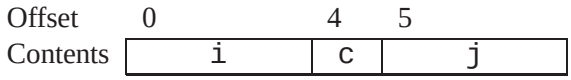
For code involving structures, the compiler may need to insert gaps in the field allocation to ensure that each structure element satisfies its alignment requirement. The structure then has some required alignment for its starting address.

For example, consider the structure declaration:

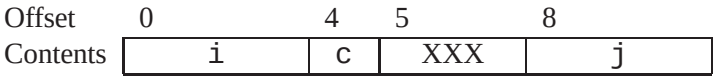
```
struct S1 {
```

```
    int  i;
    char c;
    int  j;
};
```

Suppose the compiler used the minimal 9-byte allocation, diagrammed as follows:



Then it would be impossible to satisfy the 4-byte alignment requirement for both fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap (shown below as "XXX") between fields `c` and `j`:



so that `j` has offset 8, and the overall structure size is 12 bytes. Furthermore, the compiler must ensure that any pointer `p` of type `struct S1 *` satisfies a 4-byte alignment. Using our earlier notation, let pointer `p` have value $x_{\texttt{p}}$. Then $x_{\texttt{p}}$ must be a multiple of 4. This guarantees that both `p->i` (address $x_{\texttt{p}}$) and `p->j` (address $x_{\texttt{p}} + 4$) will satisfy their 4-byte alignment requirements.

In addition, the compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement. For example, consider the following structure declaration:

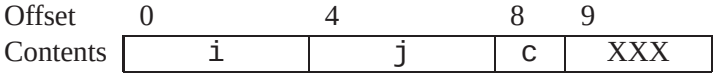
```
struct S2 {
    int  i;
    int  j;
    char c;
};
```

If we pack this structure into 9 bytes, we can still satisfy the alignment requirements for fields `i` and `j` by making sure that the starting address of the structure satisfies a 4-byte alignment requirement. Consider, however, the following declaration:

```
struct S2 d[4];
```

With the 9-byte allocation, it is not possible to satisfy the alignment requirement for each element of `d`, because these elements will have addresses $x_{\texttt{d}}$, $x_{\texttt{d}} + 9$, $x_{\texttt{d}} + 18$, and $x_{\texttt{d}} + 27$.

Instead the compiler will allocate 12 bytes for structure `S1`, with the final 3 bytes being wasted space:



That way the elements of `d` will have addresses $x_{\texttt{d}}$, $x_{\texttt{d}} + 12$, $x_{\texttt{d}} + 24$, and $x_{\texttt{d}} + 36$. As long as $x_{\texttt{d}}$ is a multiple of 4, all of the alignment restrictions will be satisfied.

Practice Problem 3.23:

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement under Linux/IA32.

A. `struct P1 { int i; char c; int j; char d; };`
B. `struct P2 { int i; char c; char d; int j; };`
C. `struct P3 { short w[3]; char c[3] };`
D. `struct P4 { short w[3]; char *c[3] };`
E. `struct P3 { struct P1 a[2]; struct P2 *p };`

3.11 Putting it Together: Understanding Pointers

Pointers are a central feature of the C programming language. They provide a uniform way to provide remote access to data structures. Pointers are a source of confusion for novice programmers, but the underlying concepts are fairly simple. The code in Figure 3.25 lets us illustrate a number of these concepts.

- *Every pointer has a type.* This type indicates what kind of object the pointer points to. In our example code, we see the following pointer types:

Pointer Type	Object Type	Pointers
`int *`	`int`	`xp`, `ip[0]`, `ip[1]`
`union uni *`	`union uni`	`up`

  Note in the above table, that we indicate the type of the pointer itself, as well as the type of the object it points to. In general, if the object has type T, then the pointer has type `*`T. The special `void *` type represents a generic pointer. For example, the `malloc` function returns a generic pointer, which is converted to a typed pointer via a cast (line 21).

- *Every pointer has a value.* This value is an address of some object of the designated type. The special `NULL` (0) value indicates that the pointer does not point anywhere. We will see the values of our pointers shortly.

- *Pointers are created with the* & *operator.* This operator can be applied to any C expression that is categorized as an *lvalue*, meaning an expression that can appear on the left side of an assignment. Examples include variables and the elements of structures, unions, and arrays. In our example code, we see this operator being applied to global variable `g` (line 24), to structure element `s.v` (line 32), to union element `up->v` (line 33), and to local variable `x` (line 42).

- *Pointers are dereferenced with the* `*` *operator.* The result is a value having the type associated with the pointer. We see dereferencing applied to both `ip` and `*ip` (line 29), to `ip[1]` (line 31), and `xp` (line 35). In addition, the expression `up->v` (line 33) both derefences pointer `up` and selects field `v`.

```
1  struct str {  /* Example Structure */
2      int t;
3      char v;
4  };
5
6  union uni {   /* Example Union */
7      int t;
8      char v;
9  } u;
10
11 int g = 15;
12
13 void fun(int* xp)
14 {
15     void (*f)(int*) = fun;   /* f is a function pointer */
16
17     /* Allocate structure on stack */
18     struct str s = {1,'a'}; /* Initialize structure */
19
20     /* Allocate union from heap */
21     union uni *up = (union uni *) malloc(sizeof(union uni));
22
23     /* Locally declared array */
24     int *ip[2] = {xp, &g};
25
26     up->v = s.v+1;
27
28     printf("ip     = %p, *ip    = %p, **ip   = %d\n",
29            ip, *ip, **ip);
30     printf("ip+1   = %p, ip[1]  =  %p, *ip[1] = %d\n",
31            ip+1, ip[1], *ip[1]);
32     printf("&s.v   = %p, s.v    = '%c'\n", &s.v, s.v);
33     printf("&up->v =  %p, up->v  = '%c'\n", &up->v, up->v);
34     printf("f      =  %p\n", f);
35     if (--(*xp) > 0)
36         f(xp);               /* Recursive call of fun */
37 }
38
39 int test()
40 {
41     int x = 2;
42     fun(&x);
43     return x;
44 }
```

Figure 3.25: **Code Illustrating Use of Pointers in C.** In C, pointers can be generated to any data type.

- *Arrays and pointers are closely related.* The name of an array can be referenced (but not updated) as if it were a pointer variable. Array referencing (e.g., `a[3]`) has the exact same effect as pointer arithmetic and dereferencing (e.g., `*(a+3)`). We can see this in line 29, where we print the pointer value of array `ip`, and reference its first (element 0) entry as `*ip`.

- *Pointers can also point to functions.* This provides a powerful capability for storing and passing references to code, which can be invoked in some other part of the program. We see this with variable `f` (line 15), which is declared to be a variable that points to a function taking an `int *` as argument and returning `void`. The assignment makes `f` point to `fun`. When we later apply `f` (line 36), we are making a recursive call.

New to C?
The syntax for declaring function pointers is especially difficult for novice programmers to understand. For a declaration such as

```
void (*f)(int*);
```

it helps to read it starting from the inside (starting with "`f`") and working outward. Thus, we see that `f` is a pointer, as indicated by "`(*f)`." It is a pointer to a function that has a single `int *` as an argument as indicated by "`(*f)(int*)`." Finally, we see that it is a pointer to a function that takes an `int *` as an argument and returns `void`.

The parentheses around `*f` are required, because otherwise the declaration:

```
void *f(int*);
```

would be read as:

```
(void *) f(int*);
```

That is, it would be interpreted as a function prototype, declaring a function `f` that has an `int *` as its argument and returns a `void *`.

Kernighan & Ritchie [37, Sect. 5.12] present a very helpful tutorial on reading C declarations. **End**

Our code contains a number of calls to `printf`, printing some of the pointers (using directive `%p`) and values. When executed, it generates the following output:

```
1  ip     = 0xbfffefa8, *ip    = 0xbfffefe4, **ip   = 2    ip[0] = xp. *xp = x = 2
2  ip+1   = 0xbfffefac, ip[1]  =  0x804965c, *ip[1] = 15   ip[1] = &g. g = 15
3  &s.v   = 0xbfffefb4, s.v    = 'a'                       s in stack frame
4  &up->v =  0x8049760, up->v  = 'b'                       up points to area in heap
5  f      =  0x8048414                                     f points to code for fun
6  ip     = 0xbfffef68, *ip    = 0xbfffefe4, **ip   = 1    ip in new frame, x = 1
7  ip+1   = 0xbfffef6c, ip[1]  =  0x804965c, *ip[1] = 15   ip[1] same as before
8  &s.v   = 0xbfffef74, s.v    = 'a'                       s in new frame
9  &up->v =  0x8049770, up->v  = 'b'                       up points to new area in heap
10 f      =  0x8048414                                     f points to code for fun
```

We see that the function is executed twice—first by the direct call from `test` (line 42), and second by the indirect, recursive call (line 36). We can see that the printed values of the pointers all correspond to addresses. Those starting with `0xbfffef` point to locations on the stack, while the rest are part of the global storage (`0x804965c`), part of the executable code (`0x8048414`), or locations on the heap (`0x8049760` and `0x8049770`).

Array `ip` is instantiated twice—once for each call to `fun`. The second value (`0xbfffef68`) is smaller than the first (`0xbfffefa8`), because the stack grows downward. The contents of the array, however, are the same in both cases. Element 0 (`*ip`) is a pointer to variable `x` in the stack frame for `test`. Element 1 is a pointer to global variable `g`.

We can see that structure `s` is instantiated twice, both times on the stack, while the union pointed to by variable `up` is allocated on the heap.

Finally, variable `f` is a pointer to function `fun`. In the disassembled code, we find the following as the initial code for `fun`:

```
1 08048414 <fun>:
2  8048414:  55                    push   %ebp
3  8048415:  89 e5                 mov    %esp,%ebp
4  8048417:  83 ec 1c              sub    $0x1c,%esp
5  804841a:  57                    push   %edi
```

The value `0x8048414` printed for pointer `f` is exactly the address of the first instruction in the code for `fun`.

New to C?
Other languages, such as Pascal, provide two different ways to pass parameters to procedures—by *value* (identified in Pascal by keyword `var`), where the caller provides the actual parameter value, and by *reference*, where the caller provides a pointer to the value. In C, all parameters are passed by value, but we can simulate the effect of a reference parameter by explicitly generating a pointer to a value and passing this pointer to a procedure. We saw this in function `fun` (Figure 3.25) with the parameter `xp`. With the initial call `fun(&x)` (line 42), the function is given a reference to local variable `x` in `test`. This variable is decremented by each call to `fun` (line 35), causing the recursion to stop after two calls.

C++ reintroduced the concept of a reference parameter, but many feel this was a mistake. **End**

3.12 Life in the Real World: Using the GDB Debugger

The GNU debugger GDB provides a number of useful features to support the run-time evaluation and analysis of machine-level programs. With the examples and exercises in this book, we attempt to infer the behavior of a program by just looking at the code. Using GDB, it becomes possible to study the behavior by watching the program in action, while having considerable control over its execution.

Figure 3.26 shows examples of some GDB commands that help when working with machine-level, IA32 programs. It is very helpful to first run OBJDUMP to get a disassembled version of the program. Our examples were based on running GDB on the file `prog`, described and disassembled on page 96. We would start GDB with the command line:

```
unix> gdb prog
```

Command	Effect
Starting and Stopping	
quit	Exit GDB
run	Run your program (give command line arguments here)
kill	Stop your program
Breakpoints	
break sum	Set breakpoint at entry to function `sum`
*break *0x80483c3*	Set breakpoint at address `0x80483c3`
delete 1	Delete breakpoint 1
delete	Delete all breakpoints
Execution	
stepi	Execute one instruction
stepi 4	Execute four instructions
nexti	Like `stepi`, but proceed through function calls
continue	Resume execution
finish	Run until current function returns
Examining code	
disas	Disassemble current function
disas sum	Disassemble function sum
disas 0x80483b7	Disassemble function around address `0x80483b7`
disas 0x80483b7 0x80483c7	Disassemble code within specified address range
print /x $eip	Print program counter in hex
Examining data	
print $eax	Print contents of `%eax` in decimal
print /x $eax	Print contents of `%eax` in hex
print /t $eax	Print contents of `%eax` in binary
print 0x100	Print decimal representation of `0x100`
print /x 555	Print hex representation of 555
print /x ($ebp+8)	Print contents of `%ebp` plus 8 in hex
*print *(int *) 0xbffff890*	Print integer at address `0xbffff890`
*print *(int *) ($ebp+8)*	Print integer at address `%ebp` + 8
x/2w 0xbffff890	Examine two (4-byte) words starting at address `0xbffff890`
x/20b sum	Examine first 20 bytes of function `sum`
Useful information	
info frame	Information about current stack frame
info registers	Values of all the registers
help	Get information about GDB

Figure 3.26: **Example** GDB **Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

The general scheme is to set breakpoints near points of interest in the program. These can be set to just after the entry of a function, or at a program address. When one of the breakpoints is hit during program execution, the program will halt and return control to the user. From a breakpoint, we can examine different registers and memory locations in various formats. We can also single-step the program, running just a few instructions at a time, or we can proceed to the next breakpoint.

As our examples suggests, GDB has an obscure command syntax, but the online help information (invoked within GDB with the `help` command) overcomes this shortcoming.

3.13 Out-of-Bounds Memory References and Buffer Overflow

We have seen that C does not perform any bounds checking for array references, and that local variables are stored on the stack along with state information such as register values and return pointers. This combination can lead to serious program errors, where the state stored on the stack gets corrupted by a write to an out-of-bounds array element. When the program then tries to reload the register or execute a `ret` instruction with this corrupted state, things can go seriously wrong.

A particularly common source of state corruption is known as *buffer overflow*. Typically some character array is allocated on the stack to hold a string, but the size of the string exceeds the space allocated for the array. This is demonstrated by the following program example.

```
1 /* Implementation of library function gets() */
2 char *gets(char *s)
3 {
4     int c;
5     char *dest = s;
6     while ((c = getchar()) != '\n' && c != EOF)
7         *dest++ = c;
8     *dest++ = '\0'; /* Terminate String */
9     if (c == EOF)
10        return NULL;
11    return s;
12 }
13
14 /* Read input line and write it back */
15 void echo()
16 {
17    char buf[4];  /* Way too small! */
18    gets(buf);
19    puts(buf);
20 }
```

The above code shows an implementation of the library function `gets` to demonstrate a serious problem with this function. It reads a line from the standard input, stopping when either a terminating newline character or some error condition is encountered. It copies this string to the location designated by argument `s`, and terminates the string with a null character. We show the use of `gets` in the function `echo`, which simply reads a line from standard input and echos it back to standard output.

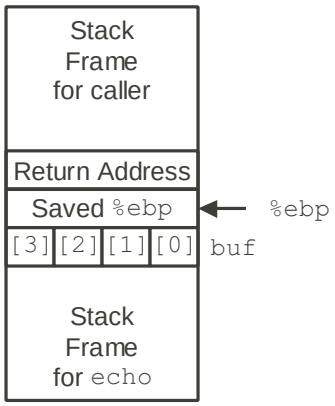


Figure 3.27: **Stack Organization for `echo` Function.** Character array `buf` is just below part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

The problem with `gets` is that it has no way to determine whether sufficient space has been allocated to hold the entire string. In our `echo` example, we have purposely made the buffer very small—just four characters long. Any string longer than three characters will cause an out-of-bounds write.

Examining a portion of the assembly code for `echo` shows how the stack is organized.

```
1 echo:
2   pushl %ebp              Save %ebp on stack
3   movl %esp,%ebp
4   subl $20,%esp           Allocate space on stack
5   pushl %ebx              Save %ebx
6   addl $-12,%esp          Allocate more space on stack
7   leal -4(%ebp),%ebx      Compute buf as %ebp-4
8   pushl %ebx              Push buf on stack
9   call gets               Call gets
```

We can see in this example that the program allocates a total of 32 bytes (lines 4 and 6) for local storage. However, the location of character array `buf` is computed as just four bytes below `%ebp` (line 7). Figure 3.27 shows the resulting stack structure. As can be seen, any write to `buf[4]` through `buf[7]` will cause the saved value of `%ebp` to be corrupted. When the program later attempts to restore this as the frame pointer, all subsequent stack references will be invalid. Any write to `buf[8]` through `buf[11]` will cause the return address to be corrupted. When the `ret` instruction is executed at the end of the function, the program will "return" to the wrong address. As this example illustrates, buffer overflow can cause a program to seriously misbehave.

Our code for `echo` is simple but sloppy. A better version involves using the function `fgets`, which includes as an argument a count on the maximum number bytes to read. Homework problem 3.37 asks you to write an echo function that can handle an input string of arbitrary length. In general, using `gets` or any function that can overflow storage is considered a bad programming practice. The C compiler even produces the following error message when compiling a file containing a call to `gets`: "the `gets` function is dangerous and should not be used."

code/asm/bufovf.c

```
1 /* This is very low quality code.
2    It is intended to illustrate bad programming practices.
3    See Practice Problem 3.24. */
4 char *getline()
5 {
6     char buf[8];
7     char *result;
8     gets(buf);
9     result = malloc(strlen(buf));
10    strcpy(result, buf);
11    return(result);
12 }
```

code/asm/bufovf.c

C Code

```
1 08048524 <getline>:
2  8048524:  55                      push   %ebp
3  8048525:  89 e5                   mov    %esp,%ebp
4  8048527:  83 ec 10                sub    $0x10,%esp
5  804852a:  56                      push   %esi
6  804852b:  53                      push   %ebx
    Diagram stack at this point
7  804852c:  83 c4 f4                add    $0xfffffff4,%esp
8  804852f:  8d 5d f8                lea    0xfffffff8(%ebp),%ebx
9  8048532:  53                      push   %ebx
10 8048533:  e8 74 fe ff ff          call   80483ac <_init+0x50>       gets
    Modify diagram to show values at this point
```

Disassembly up through call to `gets`

Figure 3.28: **C and Disassembled Code for Problem 3.24.**

Practice Problem 3.24:

Figure 3.28 shows a (low quality) implementation of a function that reads a line from standard input, copies the string to newly allocated storage, and returns a pointer to the result.

Consider the following scenario. Procedure `getline` is called with the return address equal to `0x8048643`, register `%ebp` equal to `0xbffffc94`, register `%esi` equal to `0x1`, and register `%ebx` equal to `0x2`. You type in the string "`012345678901`." The program terminates with a segmentation fault. You run GDB and determine that the error occurs during the execution of the `ret` instruction of `getline`.

A. Fill in the diagram below indicating as much as you can about the stack just after executing the instruction at line 6 in the disassembly. Label the quantities stored on the stack (e.g., "`Return Address`") on the right, and their hexadecimal values (if known) within the box. Each box represents four bytes. Indicate the position of `%ebp`.

```
+-------------+
| 08 04 86 43 |  Return Address
+-------------+
|             |
+-------------+
|             |
+-------------+
|             |
+-------------+
|             |
+-------------+
|             |
+-------------+
|             |
+-------------+
|             |
+-------------+
```

B. Modify your diagram to show the effect of the call to `gets` (line 10).

C. To what address does the program attempt to return?

D. What register(s) have corrupted value(s) when `getline` returns?

E. Besides the potential for buffer overflow, what two other things are wrong with the code for `getline`?

A more pernicious use of buffer overflow is to get a program to perform a function that it would otherwise be unwilling to do. This is one of the most common methods to attack the security of a system over a computer network. Typically, the program is fed with a string that contains the byte encoding of some executable code, called the *exploit code*, plus some extra bytes that overwrite the return pointer with a pointer to the code in the buffer. The effect of executing the `ret` instruction is then to jump to the exploit code.

In one form of attack, the exploit code then uses a system call to start up a shell program, providing the attacker with a range of operating system functions. In another form, the exploit code performs some otherwise unauthorized task, repairs the damage to the stack, and then executes `ret` a second time, causing an (apparently) normal return to the caller.

As an example, the famous Internet worm of November, 1988 used four different ways to gain access to many of the computers across the Internet. One was a buffer overflow attack on the finger daemon `fingerd`, which serves requests by the FINGER command. By invoking FINGER with an appropriate string, the worm could make the daemon at a remote site have a buffer overflow and execute code that gave the worm access to the remote system. Once the worm gained access to a system, it would replicate itself and consume virtually all of the machine's computing resources. As a consequence, hundreds of machines were effectively paralyzed until security experts could determine how to eliminate the worm. The author of the worm was caught and prosecuted. He was sentenced to three years probation, 400 hours of community service, and a $10,500 fine. Even to this day, however, people continue to find security leaks in systems that leave them vulnerable to buffer overflow attacks. This highlights the need for careful programming. Any interface to the external environment should be made "bullet proof" so that no behavior by an external agent can cause the system to misbehave.

Aside: Worms and viruses

Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [69], a *worm* is a program that can run by itself and can propagate a fully working version of itself to other machines. A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently. In the popular press, the term "virus" is used to refer to a variety of different strategies for spreading attacking code among systems, and so you will hear people saying "virus" for what more properly should be called a "worm." **End Aside.**

In Problem 3.38, you can gain first-hand experience at mounting a buffer overflow attack. Note that we do not condone using this or any other method to gain unauthorized access to a system. Breaking into computer systems is like breaking into a building—it is a criminal act even when the perpetrator does not have malicious intent. We give this problem for two reasons. First, it requires a deep understanding of machine-language programming, combining such issues as stack organization, byte ordering, and instruction encoding. Second, by demonstrating how buffer overflow attacks work, we hope you will learn the importance of writing code that does not permit such attacks.

Aside: Battling Microsoft via buffer overflow

In July, 1999, Microsoft introduced an instant messaging (IM) system whose clients were compatible with the popular AOL IM servers. This allowed Microsoft IM users to chat with AOL IM users. However, one month later, Microsoft IM users were suddenly and mysteriously unable to chat with AOL users. Microsoft released updated clients that restored service to the AOL IM system, but within days these clients no longer worked either. AOL had, possibly unintentionally, written client code that was vulnerable to a buffer overflow attack. Their server applied such an attack on client code when a user logged in to determine whether the client was running AOL code or someone else's.

The AOL exploit code sampled a small number of locations in the memory image of the client, packed them into a network packet, and sent them back to the server. If the server did not receive such a packet, or if the packet it received did not match the expected "footprint" of the AOL client, then the server assumed the client was not an AOL client and denied it access. So if other IM clients, such as Microsoft's, wanted access to the AOL IM servers, they would not only have to incorporate the buffer overflow bug that existed in AOL's clients, but they would also have to have identical binary code and data in the appropriate memory locations. But as soon as they matched these locations and distributed new versions of their client programs to customers, AOL could simply change its exploit code to sample different locations in the client's memory image. This was clearly a war that the non-AOL clients could never win!

The entire episode had a number of unusuals twists and turns. Information about the client bug and AOL's exploitation of it first came out when someone posing to be an independent consultant by the name of Phil Bucking sent

a description via email to Richard Smith, a noted security expert. Smith did some tracing and determined that the email actually originated from within Microsoft. Later Microsoft admitted that one of its employees had sent the email [48]. On the other side of the controversy, AOL never admitted to the bug nor their exploitation of it, even though conclusive evidence was made public by Geoff Chapell of Australia.

So, who violated which code of conduct in this incident? First, AOL had no obligation to open its IM system to non-AOL clients, so they were justified in blocking Microsoft. On the other hand, using buffer overflows is a tricky business. A small bug would have crashed the client computers, and it made the systems more vulnerable to attacks by external agents (although there is no evidence that this occurred). Microsoft would have done well to publicly announce AOL's intentional use of buffer overflow. However, their Phil Bucking subterfuge was clearly the wrong way to spread this information, from both an ethical and a public relations point of view. **End Aside.**

3.14 *Floating-Point Code

The set of instructions for manipulating floating-point values is one least elegant features of the IA32 architecture. In the original Intel machines, floating point was performed by a separate *coprocessor*, a unit with its own registers and processing capabilities that executes a subset of the instructions. This coprocessor was implemented as a separate chip named the 8087, 80287, and i387, to accompany the processor chips 8086, 80286, and i386, respectively. During these product generations, chip capacity was insufficient to include both the main processor and the floating-point coprocessor on a single chip. In addition, lower-budget machines would omit floating-point hardware and simply perform the floating-point operations (very slowly!) in software. Since the i486, floating point has been included as part of the IA32 CPU chip.

The original 8087 coprocessor was introduced to great acclaim in 1980. It was the first single-chip floating-point unit (FPU), and the first implementation of what is now known as IEEE floating point. Operating as a coprocessor, the FPU would take over the execution of floating-point instructions after they were fetched by the main processor. There was minimal connection between the FPU and the main processor. Communicating data from one processor to the other required the sending processor to write to memory and the receiving one to read it. Artifacts of that design remain in the IA32 floating-point instruction set today. In addition, the compiler technology of 1980 was much less sophisticated than it is today. Many features of IA32 floating point make it a difficult target for optimizing compilers.

3.14.1 Floating-Point Registers

The floating-point unit contains eight floating-point registers, but unlike normal registers, these are treated as a shallow stack. The registers are identified as `%st(0)`, `%st(1)`, and so on, up to `%st(7)`, with `%st(0)` being the top of the stack. When more than eight values are pushed onto the stack, the ones at the bottom simply disappear.

Rather than directly indexing the registers, most of the arithmetic instructions pop their source operands from the stack, compute a result, and then push the result onto the stack. Stack architectures were considered a clever idea in the 1970s, since they provide a simple mechanism for evaluating arithmetic instructions, and they allow a very dense coding of the instructions. With advances in compiler technology and with the memory required to encode instructions no longer considered a critical resource, these properties are no longer important. Compiler writers would be much happier with a larger, conventional set of floating-point registers.

Aside: Other stack-based languages.
Stack-based interpreters are still commonly used as an intermediate representation between a high-level language and its mapping onto an actual machine. Other examples of stack-based evaluators include Java byte code, the intermediate format generated by Java compilers, and the Postscript page formatting language. **End Aside.**

Having the floating-point registers organized as a bounded stack makes it difficult for compilers to use these registers for storing the local variables of a procedure that calls other procedures. For storing local integer variables, we have seen that some of the general purpose registers can be designated as callee saved and hence be used to hold local variables across a procedure call. Such a designation is not possible for an IA32 floating-point register, since its identity changes as values are pushed onto and popped from the stack. For a push operation causes the value in `%st(0)` to now be in `%st(1)`.

On the other hand, it might be tempting to treat the floating-point registers as a true stack, with each procedure call pushing its local values onto it. Unfortunately, this approach would quickly lead to a stack overflow, since there is room for only eight values. Instead, compilers generate code that saves every local floating-point value on the main program stack before calling another procedure and then retrieves them on return. This generates memory traffic that can degrade program performance.

3.14.2 Extended-Precision Arithmetic

A second unusual feature of IA32 floating point is that the floating-point registers are all 80 bits wide. They encode numbers in an *extended-precision* format as described in Problem 2.49. It is similar to an IEEE floating-point format with a 15-bit exponent (i.e., $k = 15$) and a 63-bit fraction (i.e., $n = 63$). All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single or double-precision format as they are stored in memory.

This extension to 80 bits for all register data and then contraction to a smaller format for all memory data has some undesirable consequences for programmers. It means that storing a value in memory and then retrieving it can change its value, due to rounding, underflow, or overflow. This storing and retrieving is not always visible to the C programmer, leading to some very peculiar results.

The following example illustrates this property:

code/asm/fcomp.c

```
1 double recip(int denom)
2 {
3   return 1.0/(double) denom;
4 }
5
6 void do_nothing() {} /* Just like the name says */
7
8 void test1(int denom)
9 {
10  double r1, r2;
11  int t1, t2;
12
```

```
13   r1 = recip(denom);  /* Stored in memory              */
14   r2 = recip(denom);  /* Stored in register            */
15   t1 = r1 == r2;      /* Compares register to memory   */
16   do_nothing();       /* Forces register save to memory */
17   t2 = r1 == r2;      /* Compares memory to memory     */
18   printf("test1 t1: r1 %f %c= r2 %f\n", r1, t1 ? '=' : '!', r2);
19   printf("test1 t2: r1 %f %c= r2 %f\n", r1, t2 ? '=' : '!', r2);
20 }
```

code/asm/fcomp.c

Variables `r1` and `r2` are computed by the same function with the same argument. One would expect them to be identical. Furthermmore, both variables `t1` and `t2` are computing by evaluating the expression `r1 == r2`, and so we would expect them both to equal `1`. There are no apparent hidden side effects—function `recip` does a straightforward reciprocal computation, and, as the name suggests, function `do_nothing` does nothing. When the file is compiled with optimization flag '`-O2`' and run with argument `10`, however, we get the following result:

```
test1 t1: r1 0.100000 != r2 0.100000
test1 t2: r1 0.100000 == r2 0.100000
```

The first test indicates the two reciprocals are different, while the second indicates they are the same! This is certainly not what we would expect, nor what we want. The comments in the code provide a clue for why this outcome occurs. Function `recip` returns its result in a floating-point register. Whenever procedure `test1` calls some function, it must store any value currently in a floating-point register onto the main program stack, converting from extended to double precision in the process. (We will see why this happens shortly). Before making the second call to `recip`, variable `r1` is converted and stored as a double-precision number. After the second call, variable `r2` has the extended-precision value returned by the function. In computing `t1`, the double-precision number `r1` is compared to the extended-precision number `r2`. Since 0.1 cannot be represented exactly in either format, the outcome of the test is false. Before calling function `do_nothing`, `r2` is converted and stored as a double-precision number. In computing `t2`, two double-precision numbers are compared, yielding true.

This example demonstrates a deficiency of GCC on IA32 machines (the same result occurs for both Linux and Microsoft Windows). The value associated with a variable changes due to operations that are not visible to the programmer, such as the saving and restoring of floating-point registers. Our experiments with the Microsoft Visual C++ compiler indicate that it does not have this problem.

There are several ways to overcome this problem, although none are ideal. One is to invoke GCC with the command line flag '`-mno-fp-ret-in-387`' indicating that floating-point values should be returned on the main program stack rather than in a floating-point register. Function `test1` will then show that both comparisons are true. This does not solve the problem—it just moves it to a different source of inconsistency. For example, consider the following variant, where we compute the reciprocal `r2` directly rather than calling `recip`:

code/asm/fcomp.c

```
1  void test2(int denom)
2  {
3    double r1, r2;
4    int t1, t2;
5
6    r1 = recip(denom);  /* Stored in memory              */
7    r2 = 1.0/(double) denom;  /* Stored in register      */
8    t1 = r1 == r2;      /* Compares register to memory   */
9    do_nothing();       /* Forces register save to memory */
10   t2 = r1 == r2;      /* Compares memory to memory     */
11   printf("test2 t1: r1 %f %c= r2 %f\n", r1, t1 ? '=' : '!', r2);
12   printf("test2 t2: r1 %f %c= r2 %f\n", r1, t2 ? '=' : '!', r2);
13 }
```

code/asm/fcomp.c

Once again we get `t1` equal to 0—the double-precision value in memory computed by `recip` is compared to the extended-precision value computed directly.

A second method is to disable compiler optimization. This causes the compiler to store every intermediate result on the main program stack, ensuring that all values are converted to double precision. However, this leads to a significant loss of performance.

> **Aside: Why should we be concerned about these inconsistencies?**
> As we will discuss in Chapter 5, one of the fundamental principles of optimizing compilers is that programs should produce the exact same results whether or not optimization is enabled. Unfortunately GCC does not satisfy this requirement for floating-point code. **End Aside.**

Finally, we can have GCC use extended precision in all of its computations by declaring all of the variables to be `long double` as shown in the following code:

code/asm/fcomp.c

```
1  long double recip_l(int denom)
2  {
3    return 1.0/(long double) denom;
4  }
5
6  void test3(int denom)
7  {
8    long double r1, r2;
9    int t1, t2;
10
11   r1 = recip_l(denom); /* Stored in memory              */
12   r2 = recip_l(denom); /* Stored in register            */
13   t1 = r1 == r2;       /* Compares register to memory   */
14   do_nothing();        /* Forces register save to memory */
15   t2 = r1 == r2;       /* Compares memory to memory     */
16   printf("test3 t1: r1 %f %c= r2 %f\n",
17          (double) r1, t1 ? '=' : '!', (double) r2);
```

Instruction	Effect
`load` *S*	Push value at *S* onto stack
`storep` *D*	Pop top stack element and store at *D*
`neg`	Negate top stack element
`addp`	Pop top two stack elements; Push their sum
`subp`	Pop top two stack elements; Push their difference
`multp`	Pop top two stack elements; Push their product
`divp`	Pop top two stack elements; Push their ratio

Figure 3.29: **Hypothetical Stack Instruction Set.** These instructions are used to illustrate stack-based expression evaluation

```
18   printf("test3 t2: r1 %f %c= r2 %f\n",
19          (double) r1, t2 ? '=' : '!', (double) r2);
20 }
```

code/asm/fcomp.c

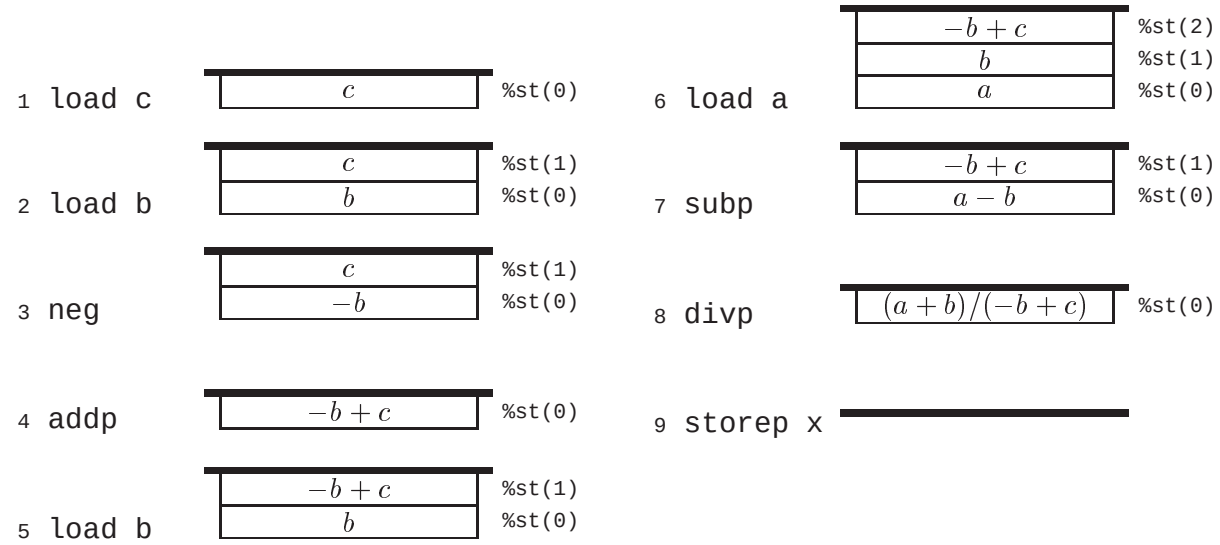
The declaration `long double` is allowed as part of the ANSI C standard, although for most machines and compilers this declaration is equivalent to an ordinary `double`. For GCC on IA32 machines, however, it uses the extended-precision format for memory data as well as for floating point register data. This allows us to take full advantage of the wider range and greater precision provided by the extended-precision format while avoiding the anomalies we have seen in our earlier examples. Unfortunately, this solution comes at a price. GCC uses 12 bytes to store a long double, increasing memory consumption by 50%. (Although 10 bytes would suffice, it rounds this up to 12 to give a better alignment. The same allocation is used on both Linux and Windows machines). Transfering these longer data between registers and memory takes more time, too. Still, this is the best option for programs requiring very consistent numerical results.

3.14.3 Stack Evaluation of Expressions

To understand how IA32 uses its floating-point registers as a stack, let us consider a more abstract version of stack-based evaluation. Assume we have an arithmetic unit that uses a stack to hold intermediate results, having the instruction set illustrated in Figure 3.29. For example, so-called RPN (for Reverse Polish Notation) pocket calculators provide this feature. In addition to the stack, this unit has a memory that can hold values we will refer to by names such as `a`, `b`, and `x`. As Figure 3.29 indicates, we can push memory values onto this stack with the `load` instruction. The `storep` operation pops the top element from the stack and stores the result in memory. A unary operation such as `neg` (negation) uses the top stack element as its argument and overwrites this element with the result. Binary operations such as `addp` and `multp` use the top two elements of the stack as their arguments. They pop both arguments off the stack and then push the result back onto the stack. We use the suffix '`p`' with the store, add, subtract, multiply, and divide instructions to emphasize the fact that these instructions pop their operands.

As an example, suppose we wish to evaluate the expression `x = (a-b)/(-b+c)`. We could translate this expression into the following code. Alongside each line of code, we show the contents of the floating-point

register stack. In keeping with our earlier convention, we show the stack as growing downward, so the "top" of the stack is really at the bottom.



As this example shows, there is a natural recursive procedure for converting an arithmetic expression into stack code. Our expression notation has four types of expressions having the following translation rules:

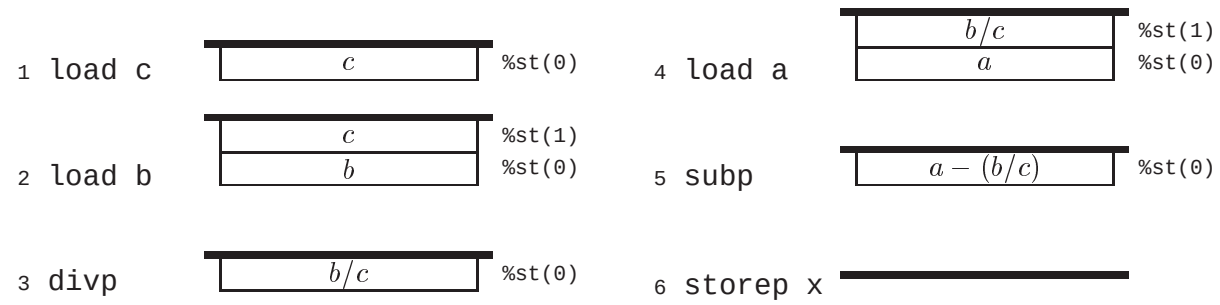
1. A variable reference of the form Var. This is implemented with the instruction `load` Var.
2. A unary operation of the form `-` $Expr$. This is implemented by first generating the code for $Expr$ followed by a `neg` instruction.
3. A binary operation of the form $Expr_1$ `+` $Expr_2$, $Expr_1$ `-` $Expr_2$, $Expr_1$ `*` $Expr_2$, or $Expr_1$ `/` $Expr_2$. This is implemented by generating the code for $Expr_2$, followed by the code for $Expr_1$, followed by an `addp`, `subp`, `multp`, or `divp` instruction.
4. An assignment of the form Var `=` $Expr$. This is implemented by first generating the code for $Expr$, followed by the `storep` Var instruction.

As an example, consider the expression `x = a-b/c`. Since division has precedence over subtraction, this expression can be parenthesized as `x = a-(b/c)`. The recursive procedure would therefore proceed as follows:

1. Generate code for $Expr$ $\doteq$ `a-(b/c)`:
   (a) Generate code for $Expr_2$ $\doteq$ `b/c`:
      i. Generate code for $Expr_2$ $\doteq$ `c` using the instruction `load c`.
      ii. Generate code for $Expr_1$ $\doteq$ `b`, using the instruction `load b`.
      iii. Generate instruction `divp`.
   (b) Generate code for $Expr_1$ $\doteq$ `a`, using the instruction `load a`.
   (c) Generate instruction `subp`.

2. Generate instruction `storep x`.

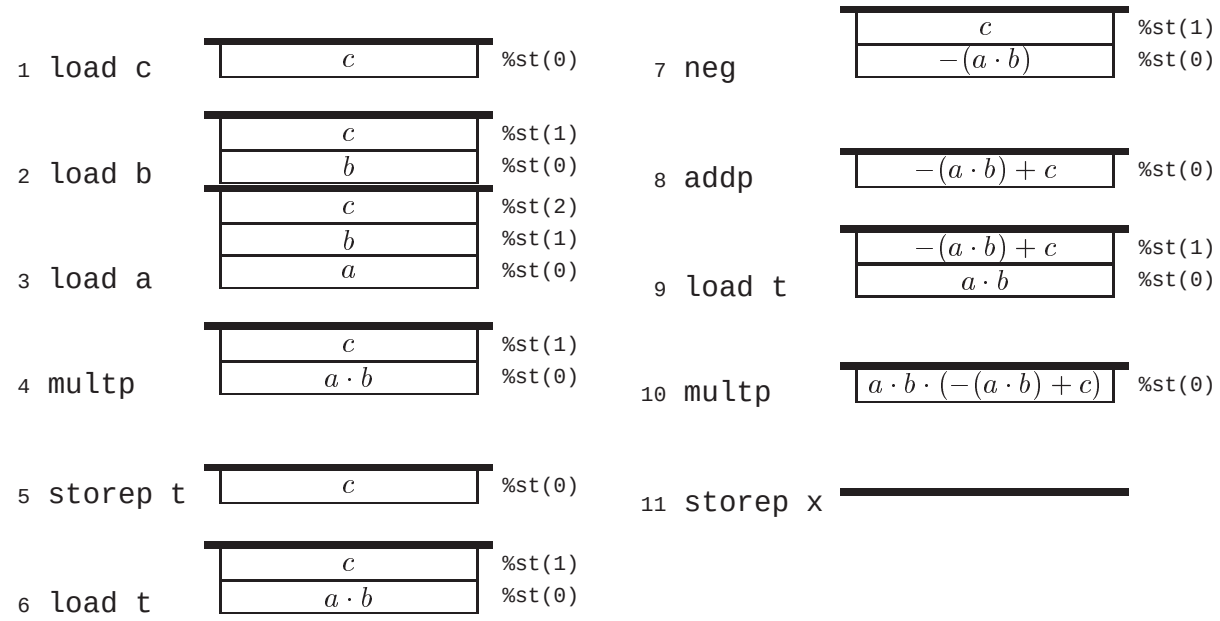
The overall effect is to generate the following stack code:



Practice Problem 3.25:

Generate stack code for the expression `x = a*b/c * -(a+b*c)`. Diagram the contents of the stack for each step of your code. Remember to follow the C rules for precedence and associativity.

Stack evaluation becomes more complex when we wish to use the result of some computation multiple times. For example, consider the expression `x = (a*b)*(-(a*b)+c)`. For efficiency, we would like to compute `a*b` only once, but our stack instructions do not provide a way to keep a value on the stack once it has been used. With the set of instructions listed in Figure 3.29, we would therefore need to store the intermediate result `a+b` in some memory location, say `t`, and retrieve this value for each use. This gives the following code:



This approach has the disadvantage of generating additional memory traffic, even though the register stack has sufficient capacity to hold its intermediate results. The IA32 floating-point unit avoids this inefficiency

Instruction		Source Format	Source Location
`flds`	$Addr$	Single	$Mem_4[Addr]$
`fldl`	$Addr$	double	$Mem_8[Addr]$
`fldt`	$Addr$	extended	$Mem_{10}[Addr]$
`fildl`	$Addr$	integer	$Mem_4[Addr]$
`fld`	`%st(`i`)`	extended	`%st(`i`)`

Figure 3.30: **Floating-Point Load Instructions.** All convert the operand to extended-precision format and push it onto the register stack.

by introducing variants of the arithmetic instructions that leave their second operand on the stack, and that can use an arbitrary stack value as their second operand. In addition, it provides an instruction that can swap the top stack element with any other element. Although these extensions can be used to generate more efficient code, the simple and elegant algorithm for translating arithmetic expressions into stack code is lost.

3.14.4 Floating-Point Data Movement and Conversion Operations

Floating-point registers are referenced with the notation `%st(`i`)`, where i denotes the position relative to the top of the stack. The value i can range between 0 and 7. Register `%st(0)` is the top stack element, `%st(1)` is the second element, and so on. The top stack element can also be referenced as `%st`. When a new value is pushed onto the stack, the value in register `%st(7)` is lost. When the stack is popped, the new value in `%st(7)` is not predictable. Compilers must generate code that works within the limited capacity of the register stack.

Figure 3.30 shows the set of instructions used to push values onto the floating-point register stack. The first group of these read from a memory location, where the argument $Addr$ is a memory address given in one of the memory operand formats listed in Figure 3.3. These instructions differ by the presumed format of the source operand and hence the number of bytes that must be read from memory. We use the notation $Mem_n[Addr]$ to denote accessing of n bytes with starting address $Addr$. All of these instructions convert the operand to extended-precision format before pushing it onto the stack. The final load instruction `fld` is used to duplicate a stack value. That is, it pushes a copy of floating-point register `%st(`i`)` onto the stack. For example, the instruction `fld %st(0)` pushes a copy of the top stack element onto the stack.

Figure 3.31 shows the instructions that store the top stack element either in memory or in another floating-point register. There are both "popping" versions that pop the top element off the stack, similar to the `storep` instruction for our hypothetical stack evaluator, as well as nonpopping versions that leave the source value on the top of the stack. As with the floating-point load instructions, different variants of the instruction generate different formats for the result and therefore store different numbers of bytes. The first group of these store the result in memory. The address is specified using any of the memory operand formats listed in Figure 3.3. The second group copies the top stack element to some other floating-point register.

Practice Problem 3.26:

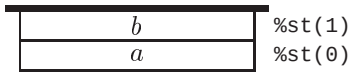
Assume for the following code fragment that register `%eax` contains an integer variable `x` and that the top two stack elements correspond to variables `a` and `b`, respectively. Fill in the boxes to diagram the

Instruction		Pop (Y/N)	Destination Format	Destination Location
`fsts`	$Addr$	N	Single	$Mem_4[Addr]$
`fstps`	$Addr$	Y	Single	$Mem_4[Addr]$
`fstl`	$Addr$	N	Double	$Mem_8[Addr]$
`fstpl`	$Addr$	Y	Double	$Mem_8[Addr]$
`fstt`	$Addr$	N	Extended	$Mem_{10}[Addr]$
`fstpt`	$Addr$	Y	Extended	$Mem_{10}[Addr]$
`fistl`	$Addr$	N	integer	$Mem_4[Addr]$
`fistpl`	$Addr$	Y	integer	$Mem_4[Addr]$
`fst`	`%st(`i`)`	N	Extended	`%st(`i`)`
`fstp`	`%st(`i`)`	Y	Extended	`%st(`i`)`

Figure 3.31: **Floating-Point Store Instructions.** All convert from extended-precision format to the destination format. Instructions with suffix '`p`' pop the top element off the stack.

stack contents after each instruction

```
    testl %eax,%eax
```




```
    jne L11
```


```
    fstp %st(0)

    jmp L9

L11:
```


```
    fstp %st(1)

L9:
```

Write a C expression describing the contents of the top stack element at the end of this code sequence in terms of `x`, `a` and `b`.

A final floating-point data movement operation allows the contents of two floating-point registers to be swapped. The instruction `fxch %st(`i`)` exchanges the contents of floating-point registers `%st(0)` and `%st(`i`)`. The notation `fxch` written with no argument is equivalent to `fxch %st(1)`, that is, swap the top two stack elements.

Instruction	Computation
`fldz`	0
`fld1`	1
`fabs`	$\lvert Op \rvert$
`fchs`	$-Op$
`fcos`	$\cos Op$
`fsin`	$\sin Op$
`fsqrt`	$\sqrt{Op}$
`fadd`	$Op_1 + Op_2$
`fsub`	$Op_1 - Op_2$
`fsubr`	$Op_2 - Op_1$
`fdiv`	Op_1 / Op_2
`fdivr`	Op_2 / Op_1
`fmul`	$Op_1 \cdot Op_2$

Figure 3.32: **Floating-Point Arithmetic Operations.** Each of the binary operations has many variants.

Instruction		Operand 1	Operand 2	(Format)	Destination	Pop `%st(0)` (Y/N)
`fsubs`	$Addr$	`%st(0)`	$Mem_4[Addr]$	Single	`%st(0)`	N
`fsubl`	$Addr$	`%st(0)`	$Mem_8[Addr]$	Double	`%st(0)`	N
`fsubt`	$Addr$	`%st(0)`	$Mem_{10}[Addr]$	Extended	`%st(0)`	N
`fisubl`	$Addr$	`%st(0)`	$Mem_4[Addr]$	integer	`%st(0)`	N
`fsub`	`%st(`i`),%st`	`%st(`i`)`	`%st(0)`	Extended	`%st(0)`	N
`fsub`	`%st,%st(`i`)`	`%st(0)`	`%st(`i`)`	Extended	`%st(`i`)`	N
`fsubp`	`%st,%st(`i`)`	`%st(0)`	`%st(`i`)`	Extended	`%st(`i`)`	Y
`fsubp`		`%st(0)`	`%st(1)`	Extended	`%st(1)`	Y

Figure 3.33: **Floating-Point Subtraction Instructions.** All store their results into a floating-point register in extended-precision format. Instructions with suffix '`p`' pop the top element off the stack.

3.14.5 Floating-Point Arithmetic Instructions

Figure 3.32 documents some of the most common floating-point arithmetic operations. Instructions in the first group have no operands. They push the floating-point representation of some numerical constant onto the stack. There are similar instructions for such constants as π, e, and $\log_2 10$. Instructions in the second group have a single operand. The operand is always the top stack element, similar to the `neg` operation of the hypothetical stack evaluator. They replace this element with the computed result. Instructions in the third group have two operands. For each of these instructions, there are many different variants for how the operands are specified, as will be discussed shortly. For noncommutative operations such as subtraction and division there is both a forward (e.g., `fsub`) and a reverse (e.g., `fsubr`) version, so that the arguments can be used in either order.

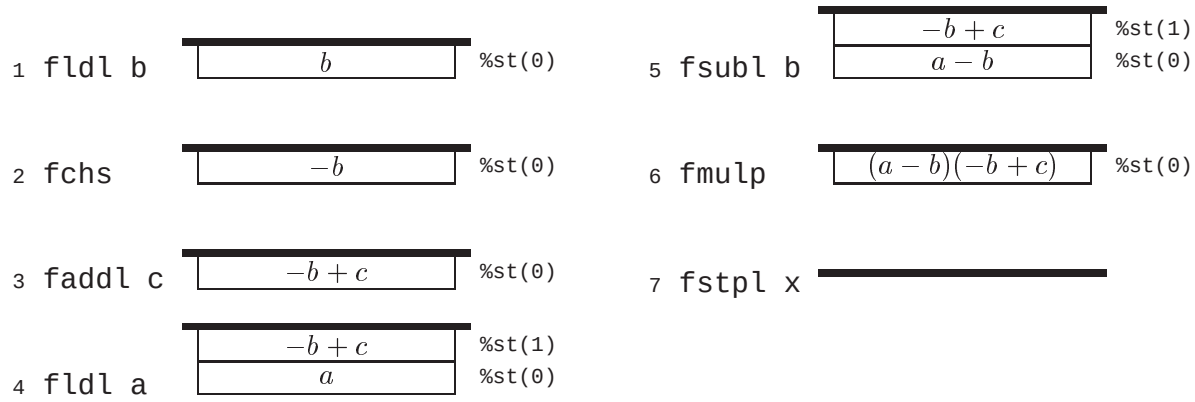
In Figure 3.32 we show just a single form of the subtraction operation `fsub`. In fact, this operation comes in

many different variants, as shown in Figure 3.33. All compute the difference of two operands: $Op_1 - Op_2$ and store the result in some floating-point register. Beyond the simple `subp` instruction we considered for the hypothetical stack evaluator, IA32 has instructions that read their second operand from memory or from some floating-point register other than `%st(1)`. In addition, there are both popping and nonpopping variants. The first group of instructions reads the second operand from memory, either in single-precision, double-precision, or integer format. It then converts this to extended-precision format, subtracts it from the top stack element, and overwrites the top stack element. These can be seen as a combination of a floating-point load following by a stack-based subtraction operation.

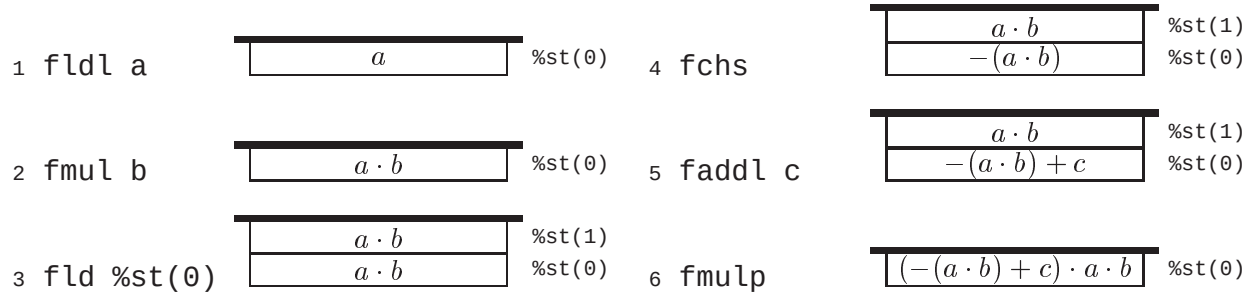
The second group of subtraction instructions use the top stack element as one argument and some other stack element as the other, but they vary in the argument ordering, the result destination, and whether or not they pop the top stack element. Observe that the assembly code line `fsubp` is shorthand for `fsubp %st,%st(1)`. This line corresponds to the `subp` instruction of our hypothetical stack evaluator. That is, it computes the difference between the top two stack elements, storing the result in `%st(1)`, and then popping `%st(0)` so that the computed value ends up on the top of the stack.

All of the binary operations listed in Figure 3.32 come in all of the variants listed for `fsub` in Figure 3.33.

As an example, we can rewrite the code for the expression `x = (a-b)*(-b+c)` using the IA32 instructions. For exposition purposes we will still use symbolic names for memory locations and we assume these are double-precision values.

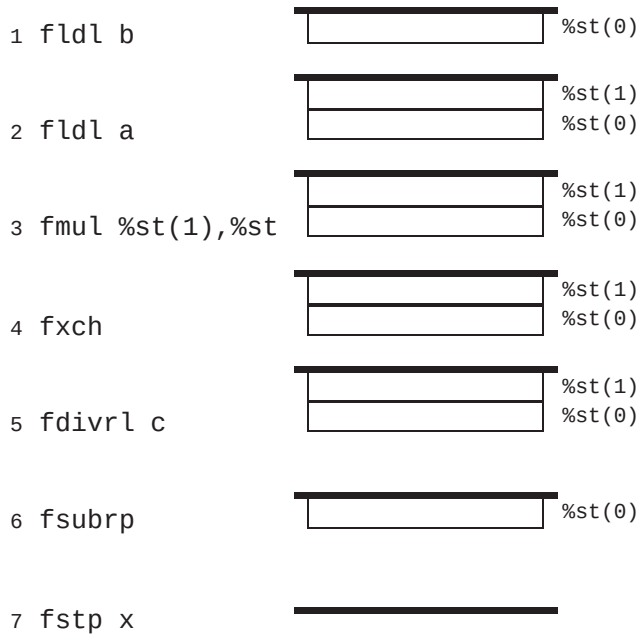


As another example, we can write the code for the expression `x = (a*b)+(-(a*b)+c)` as follows. Observe how the instruction `fld %st(0)` is used to create two copies of `a*b` on the stack, avoiding the need to save the value in a temporary memory location.



Practice Problem 3.27:

Diagram the stack contents after each step of the following code:



Give an expression describing this computation.

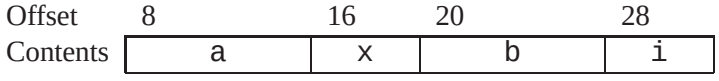
3.14.6 Using Floating Point in Procedures

Floating-point arguments are passed to a calling procedure on the stack, just as are integer arguments. Each parameter of type `float` requires 4 bytes of stack space, while each parameter of type `double` requires 8. For functions whose return values are of type `float` or `double`, the result is returned on the top of the floating-point register stack in extended-precision format.

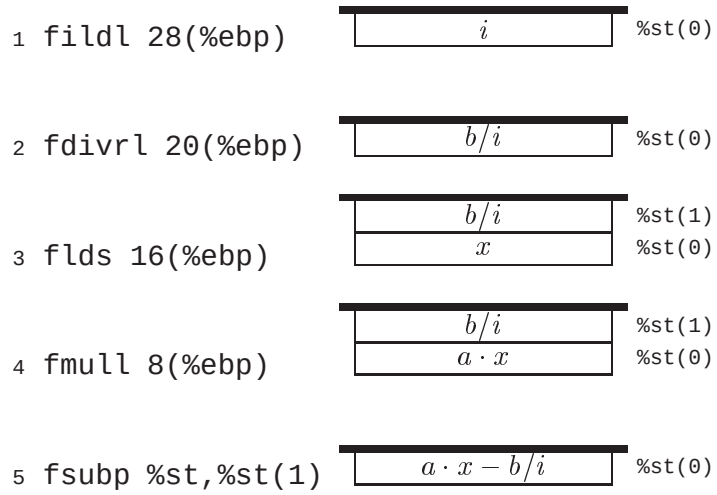
As an example, consider the following function

```
1 double funct(double a, float x, double b, int i)
2 {
3     return a*x - b/i;
4 }
```

Arguments `a`, `x`, `b`, and `i` will be at byte offsets 8, 16, 20, and 28 relative to `%ebp`, respectively, as diagrammed below:



The body of the generated code, and the resulting stack values are as follows:



Practice Problem 3.28:

For a function `funct2` with arguments `a`, `x`, `b`, and `i` (and a different declaration than that of `funct`, the compiler generates the following code for the function body:

```
1   movl 8(%ebp),%eax
2   fldl 12(%ebp)
3   flds 20(%ebp)
4   movl %eax,-4(%ebp)
5   fildl -4(%ebp)
6   fxch %st(2)
7   faddp %st,%st(1)
8   fdivrp %st,%st(1)
9   fld1
10  flds 24(%ebp)
11  faddp %st,%st(1)
```

The returned value is of type `double`. Write C code for `funct2`. Be sure to correctly declare the argument types.

3.14.7 Testing and Comparing Floating-Point Values

Similar to the integer case, determining the relative values of two floating-point numbers involves using a comparison instruction to set condition codes and then testing these condition codes. For floating point, however, the condition codes are part of the *floating-point status word*, a 16-bit register that contains various flags about the floating-point unit. This status word must be transferred to an integer word, and then the particular bits must be tested.

Ordered		Unordered		Op_2	Type	Number of Pops
`fcoms`	*Addr*	`fucoms`	*Addr*	$Mem_4[Addr]$	Single	0
`fcoml`	*Addr*	`fucoml`	*Addr*	$Mem_8[Addr]$	Double	0
`fcom`	`%st(`i`)`	`fucom`	`%st(`i`)`	`%st(`i`)`	Extended	0
`fcom`		`fucom`		`%st(1)`	Extended	0
`fcomps`	*Addr*	`fucomps`	*Addr*	$Mem_4[Addr]$	Single	1
`fcompl`	*Addr*	`fucompl`	*Addr*	$Mem_8[Addr]$	Double	1
`fcomp`	`%st(`i`)`	`fucomp`	`%st(`i`)`	`%st(`i`)`	Extended	1
`fcomp`		`fucomp`		`%st(1)`	Extended	1
`fcompp`		`fucompp`		`%st(1)`	Extended	2

Figure 3.34: **Floating-Point Comparison Instructions.** Ordered vs. unordered comparisons differ in their treatment of *NaN*'s.

$Op_1 : Op_2$	Binary	Decimal
$>$	[00000000]	0
$<$	[00000001]	1
$=$	[00100000]	64
Unordered	[00100101]	69

Figure 3.35: **Encoded Results from Floating-Point Comparison.** The results are encoded in the high-order byte of the floating-point status word after masking out all but bits 0, 2, and 6.

There are a number of different floating-point comparison instructions as documented in Figure 3.34. All of them perform a comparison between operands Op_1 and Op_2, where Op_1 is the top stack element. Each line of the table documents two different comparison types: an *ordered* comparison used for comparisons such as $<$ and $\leq$, and an *unordered* comparison used for equality comparisons. The two comparisons differ only in their treatment of *NaN* values, since there is no relative ordering between *NaN*'s and other values. For example, if variable `x` is a *NaN* and variable `y` is some other value, then both expressions `x < y` and `x >= y` should yield 0.

The various forms of comparison instructions also differ in the location of operand Op_2, analogous to the different forms of floating-point load and floating-point arithmetic instructions. Finally, the various forms differ in the number of elements popped off the stack after the comparison is completed. Instructions in the first group shown in the table do not change the stack at all. Even for the case where one of the arguments is in memory, this value is not on the stack at the end. Operations in the second group pop element Op_1 off the stack. The final operation pops both Op_1 and Op_2 off the stack.

The floating-point status word is transferred to an integer register with the `fnstsw` instruction. The operand for this instruction is one of the 16-bit register identifiers shown in Figure 3.2, for example, `%ax`. The bits in the status word encoding the comparison results are in bit positions 0, 2, and 6 of the high-order byte of the status word. For example, if we use instruction `fnstw %ax` to transfer the status word, then the relevant bits will be in `%ah`. A typical code sequence to select these bits is then:

```
1   fnstsw %ax          Store floating point status word in %ax
```

```
2   andb $69,%ah      Mask all but bits 0, 2, and 6
```

Note that 69_{10} has bit representation $[00100101]$, that is, it has 1s in the three relevant bit positions. Figure 3.35 shows the possible values of byte %ah that would result from this code sequence. Observe that there are only four possible outcomes for comparing operands Op_1 and Op_2: the first is either greater, less, equal, or incomparable to the second, where the latter outcome only occurs when one of the values is a NaN.

As an example, consider the following procedure:

```
1 int less(double x, double y)
2 {
3     return x < y;
4 }
```

The compiled code for the function body is shown below:

```
1   fldl 16(%ebp)     Push y
2   fcompl 8(%ebp)    Compare y:x
3   fnstsw %ax        Store floating point status word in %ax
4   andb $69,%ah      Mask all but bits 0, 2, and 6
5   sete %al          Test for comparison outcome of 0 (>)
6   movzbl %al,%eax   Copy low order byte to result, and set rest to 0
```

Practice Problem 3.29:

Show how by inserting a single line of assembly code into the code sequence shown above you can implement the following function:

```
1 int greater(double x, double y)
2 {
3     return x > y;
4 }
```

This completes our coverage of assembly-level, floating-point programming with IA32. Even experienced programmers find this code arcane and difficult to read. The stack-based operations, the awkwardness of getting status results from the FPU to the main processor, and the many subtleties of floating-point computations combine to make the machine code lengthy and obscure. It is remarkable that the modern processors manufactured by Intel and its competitors can achieve respectable performance on numeric programs given the form in which they are encoded.

3.15 *Embedding Assembly Code in C Programs

In the early days of computing, most programs were written in assembly code. Even large-scale operating systems were written without the help of high-level languages. This becomes unmanageable for programs of significant complexity. Since assembly code does not provide any form of type checking, it is very easy

to make basic mistakes, such as using a pointer as an integer rather than dereferencing the pointer. Even wors, writing in assembly code locks the entire program into a particular class of machine. Rewriting an assembly language program to run on a different machine can be as difficult as writing the entire program from scratch.

> **Aside: Writing large programs in assembly code.**
> Frederick Brooks, Jr., a pioneer in computer systems wrote a fascinating account of the development of OS/360, an early operating system for IBM machines [5] that still provides important object lessons today. He became a devoted believer in high-level languages for systems programming as a result of this effort. Surprisingly, however, there is an active group of programmers who take great pleasure in writing assembly code for IA32. The communicate with one another via the Internet news group `comp.lang.asm.x86`. Most of them write computer games for the DOS operating system. **End Aside.**

Early compilers for higher-level programming languages did not generate very efficient code and did not provide access to the low-level object representations, as is often required by systems programmers. Programs requiring maximum performance or requiring access to object representations were still often written in assembly code. Nowadays, however, optimizing compilers have largely removed performance optimization as a reason for writing in assembly code. Code generated by a high quality compiler is generally as good or even better than what can be achieved manually. The C language has largely eliminated machine access as a reason for writing in assembly code. The ability to access low-level data representations through unions and pointer arithmetic, along with the ability to operate on bit-level data representations, provide sufficient access to the machine for most programmers. For example, almost every part of a modern operating system such as Linux is written in C.

Nonetheless, there are times when writing in assembly code is the only option. This is especially true when implementing an operating system. For example, there are a number of special registers storing process state information that the operating system must access. There are either special instructions or special memory locations for performing input and output operations. Even for application programmers, there are some machine features, such as the values of the condition codes, that cannot be accessed directly in C.

The challenge then is to integrate code consisting mainly of C with a small amount written in assembly language. One method is to write a few key functions in assembly code, using the same conventions for argument passing and register usage as are followed by the C compiler. The assembly functions are kept in a separate file, and the compiled C code is combined with the assembled assembly code by the linker. For example, if file `p1.c` contains C code and file `p2.s` contains assembly code, then the compilation command:

```
unix> gcc -o p p1.c p2.s
```

will cause file `p1.c` to be compiled, file `p2.s` to be assembled, and the resulting object code to be linked to form an executable program `p`.

3.15.1 Basic Inline Assembly

With GCC, it is also possible to mix assembly with C code. Inline assembly allows the user to insert assembly code directly into the code sequence generated by the compiler. Features are provided to specify instruction operands and to indicate to the compiler which registers are being overwritten by the assembly instructions.

The resulting code is, of course, highly machine-dependent, since different types of machines do not have compatible machine instructions. The `asm` directive is also specific to GCC, creating an incompatibility with many other compilers. Nonetheless, this can be a useful way to keep the amount of machine-dependent code to an absolute minimum.

Inline assembly is documented as part of the GCC information archive. Executing the command `info gcc` on any machine with GCC installed will give a hierarchical document reader. Inline assembly is documented by first following the link titled "C Extensions" and then the link titled "Extended Asm." Unfortunately, the documentation is somewhat incomplete and imprecise.

The basic form of inline assembly is to write code that looks like a procedure call:

`asm(` *code-string* `);`

where *code-string* is an assembly code sequence given as a quoted string. The compiler will insert this string verbatim into the assembly code being generated, and hence the compiler-supplied and the user-supplied assembly will be combined. The compiler does not check the string for errors, and so the first indication of a problem might be an error report from the assembler.

We illustrate the use of `asm` by an example where having access to the condition codes can be useful. Consider functions with the following prototypes:

```
int ok_smul(int x, int y, int *dest);

int ok_umul(unsigned x, unsigned y, unsigned *dest);
```

Each is supposed to compute the product of arguments `x` and `y` and store the result in the memory location specified by argument `dest`. As return values, they should return 0 when the multiplication overflows and 1 when it does not. We have separate functions for signed and unsigned multiplication, since they overflow under different circumstances.

Examining the documentation for the IA32 multiply instructions `mul` and `imul`, we see that both set the carry flag `CF` when they overflow. Examining Figure 3.9, we see that the instruction `setae` can be used to set the low-order byte of a register to 0 when this flag is set and to 1 otherwise. Thus, we wish to insert this instruction into the sequence generated by the compiler.

In an attempt to use the least amount of both assembly code and detailed analysis, we attempt to implement `ok_smul` with the following code:

code/asm/okmul.c

```
1 /* First attempt.  Does not work */
2 int ok_smul1(int x, int y, int *dest)
3 {
4     int result = 0;
5
6     *dest = x*y;
7     asm("setae %al");
8     return result;
9 }
```

code/asm/okmul.c

The strategy here is to exploit the fact that register `%eax` is used to store the return value. Assuming the compiler uses this register for variable `result`, the first line will set the register to 0. The inline assembly will insert code that sets the low-order byte of this register appropriately, and the register will be used as the return value.

Unfortunately, GCC has its own ideas of code generation. Instead of setting register `%eax` to 0 at the beginning of the function, the generated code does so at the very end, and so the function always returns 0. The fundamental problem is that the compiler has no way to know what the programmer's intentions are, and how the assembly statement should interact with the rest of the generated code.

By a process of trial and error (we will develop more systematic approaches shortly), we were able to generate working, but less than ideal code as follows:

code/asm/okmul.c

```
1 /* Second attempt.  Works in limited contexts */
2 int dummy = 0;
3
4 int ok_smul2(int x, int y, int *dest)
5 {
6     int result;
7
8     *dest = x*y;
9     result = dummy;
10    asm("setae %al");
11    return result;
12 }
```

code/asm/okmul.c

This code uses the same strategy as before, but it reads a global variable `dummy` to initialize `result` to 0. Compilers are typically more conservative about generating code involving global variables, and therefore less likely to rearrange the ordering of the computations.

The above code depends on quirks of the compiler to get proper behavior. In fact, it only works when compiled with optimization enabled (command line flag `-O`). When compiled without optimization, it stores `result` on the stack and retrieves its value just before returning, overwriting the value set by the `setae` instruction. The compiler has no way of knowing how the inserted assembly language relates to the rest of the code, because we provided the compiler no such information.

3.15.2 Extended Form of `asm`

GCC provides an extended version of the `asm` that allows the programmer to specify which program values are to be used as operands to an assembly code sequence and which registers are overwritten by the assembly code. With this information the compiler can generate code that will correctly set up the required source values, execute the assembly instructions, and make use of the computed results. It will also have information it requires about register usage so that important program values are not overwritten by the assembly code instructions.

The general syntax of an extended assembly sequence is as follows:

`asm(` *code-string* [`:` *output-list* [`:` *input-list* [`:` *overwrite-list*]]] `);`

where the square brackets denote optional arguments. The declaration contains a string describing the assembly code sequence, followed by optional lists of outputs (i.e., results generated by the assembly code), inputs (i.e., source values for the assembly code), and registers that are overwritten by the assembly code. These lists are separated by the colon (‘`:`’) character. As the square brackets show, we only include lists up to the last nonempty list.

The syntax for the code string is reminiscent of that for the format string in a `printf` statement. It consists of a sequence of assembly code instructions separated by the semicolon (‘`;`’) character. Input and output operands are denoted by references `%0`, `%1`, and so on, up to possibly `%9`. Operands are numbered, according to their ordering first in the output list and then in the input list. Register names such as “`%eax`” must be written with an extra ‘`%`’ symbol, e.g., “`%%eax`.”

The following is a better implementation of `ok_smul` using the extended assembly statement to indicate to the compiler that the assembly code generates the value for variable `result`:

code/asm/okmul.c

```
1 /* Uses the extended assembly statement to get reliable code */
2 int ok_smul3(int x, int y, int *dest)
3 {
4     int result;
5
6     *dest = x*y;
7
8     /* Insert the following assembly code:
9        setae %bl            # Set low-order byte
10       movzbl %bl, result   # Zero extend to be result
11    */
12    asm("setae %%bl; movzbl %%bl,%0"
13        : "=r" (result)  /* Output    */
14        :                /* No inputs */
15        : "%ebx"         /* Overwrites */
16        );
17
18    return result;
19 }
```

code/asm/okmul.c

The first assembly instruction stores the test result in the single-byte register `%bl`. The second instruction then zero-extends and copies the value to whatever register the compiler chooses to hold `result`, indicated by operand `%0`. The output list consists of pairs of values separated by spaces. (In this example there is only a single pair). The first element of the pair is a string indicating the operand type, where ‘`r`’ indicates an integer register and ‘`=`’ indicates that the assembly code assigns a value to this operand. The second element of the pair is the operand enclosed in parentheses. It can be any assignable value (known in C as an *lvalue*).

The input list has the same general format, while the overwrite list simply gives the names of the registers (as quoted strings) that are overwritten.

The code shown above works regardless of the compilation flags. As this example illustrates, it may take a little creative thinking to write assembly code that will allow the operands to be described in the required form. For example, there are no direct ways to specify a program value to use as the destination operand for the `setae` instruction, since the operand must be a single byte. Instead, we write a code sequence based on a specific register and then use an extra data movement instruction to copy the resulting value to some part of the program state.

Practice Problem 3.30:

GCC provides a facility for extended-precision arithmetic. This can be used to implement function `ok_smul`, with the advantage that it is portable across machines. A variable declared as type "`long long`" will have twice the size of normal `long` variable. Thus, the statement:

```
long long prod = (long long) x * y;
```

will compute the full 64-bit product of `x` and `y`. Write a version of `ok_smul` that does not use any `asm` statements.

One would expect the same code sequence could be used for `ok_umul`, but GCC uses the `imull` (signed multiply) instruction for both signed and unsigned multiplication. This generates the correct value for either product, but it sets the carry flag according to the rules for signed multiplication. We therefore need to include an assembly-code sequence that explicitly performs unsigned multiplication using the `mull` instruction as documented in Figure 3.8, as follows:

code/asm/okmul.c

```
1  /* Uses the extended assembly statement */
2  int ok_umul(unsigned x, unsigned y, unsigned *dest)
3  {
4      int result;
5
6      /* Insert the following assembly code:
7         movl  x,%eax         # Get x
8         mull  y              # Unsigned multiply by y
9         movl  %eax, *dest    # Store low-order 4 bytes at dest
10        setae %dl            # Set low-order byte
11        movzbl %dl, result   # Zero extend to be result
12     */
13     asm("movl %2,%%eax; mull %3; movl %%eax,%0;
14            setae %%dl; movzbl %%dl,%1"
15         : "=r" (*dest), "=r" (result) /* Outputs    */
16         : "r"  (x),     "r"  (y)      /* Inputs     */
17         : "%eax", "%edx"              /* Overwrites */
18         );
19
20     return result;
21 }
```

code/asm/okmul.c

Recall that the `mull` instruction requires one of its arguments to be in register `%eax` and is given the second argument as an operand. We indicate this in the `asm` statement by using a `movl` to move program value `x` to `%eax` and indicating that program value `y` should be the argument for the `mull` instruction. The instruction then stores the 8-byte product in two registers with `%eax` holding the low-order 4 bytes and `%edx` holding the high-order bytes. We then use register `%edx` to construct the return value. As this example illustrates, comma (‘`,`’) characters are used to separate pairs of operands in the input and output lists, and register names in the overwrite list. Note that we were able to specify `*dest` as an output of the second `movl` instruction, since this is an assignable value. The compiler then generates the correct machine code to store the value in `%eax` at this memory location.

Although the syntax of the `asm` statement is somewhat arcane, and its use makes the code less portable, this statement can be very useful for writing programs that accesses machine-level features using a minimal amount of assembly code. We have found that a certain amount of trial and error is required to get code that works. The best strategy is to compile the code with the `-S` switch and then examine the generated assembly code to see if it will have the desired effect. The code should be tested with different settings of switches such as with and without the `-O` flag.

3.16 Summary

In this chapter, we have peered beneath the layer of abstraction provided by a high-level language to get a view of machine-level programming. By having the compiler generate an assembly-code representation of the machine-level program, we can gain insights into both the compiler and its optimization capabilities, along with the machine, its data types, and its instruction set. As we will see in Chapter 5, knowing the characteristics of a compiler can help when trying to write programs that will have efficient mappings onto the machine. We have also seen examples where the high-level language abstraction hides important details about the operation of a program. For example, we have seen that the behavior of floating-point code can depend on whether values are held in registers or in memory. In Chapter 7, we will see many examples where we need to know whether a program variable is on the runtime stack, in some dynamically-allocated data structure, or in some global storage locations. Understanding how programs map onto machines makes it easier to understand the difference between these kinds of storage.

Assembly language is very different from C code. There is minimal distinction between different data types. The program is expressed as a sequence of instructions, each of which performs a single operation. Parts of the program state, such as registers and the runtime stack, are directly visible to the programmer. Only low-level operations are provided to support data manipulation and program control. The compiler must use multiple instructions to generate and operate on different data structures and to implement control constructs such as conditionals, loops, and procedures. We have covered many different aspects of C and how it gets compiled. We have seen the that the lack of bounds checking in C makes many programs prone to buffer overflows, and that this has made many system vulnerable to attacks.

We have only examined the mapping of C onto IA32, but much of what we have covered is handled in a similar way for other combinations of language and machine. For example, compiling C++ is very similar to compiling C. In fact, early implementations of C++ simply performed a source-to-source conversion from

C++ to C and generated object code by running a C compiler on the result. C++ objects are represented by structures, similar to a C `struct`. Methods are represented by pointers to the code implementing the methods. By contrast, Java is implemented in an entirely different fashion. The object code of Java is a special binary representation known as *Java byte code*. This code can be viewed as a machine-level program for a *virtual machine*. As its name suggests, this machine is not implemented directly in hardware. Instead, software interpreters process the byte code, simulating the behavior of the virtual machine. The advantage of this approach is that the same Java byte code can be executed on many different machines, whereas the machine code we have considered runs only under IA32.

Bibliographic Notes

The best references on IA32 are from Intel. Two useful references are part of their series on software development. The basic architecture manual [17] gives an overview of the architecture from the perspective of an assembly-language programmer, and the instruction set reference manual [18] gives detailed descriptions of the different instructions. These references contain far more information than is required to understand Linux code. In particular, with flat mode addressing, all of the complexities of the segmented addressing scheme can be ignored.

The GAS format used by the Linux assembler is very different from the standard format used in Intel documentation and by other compilers (particularly those produced by Microsoft). One main distinction is that the source and destination operands are given in the opposite order

On a Linux machine, running the command `info as` will display information about the assembler. One of the subsections documents machine-specific information, including a comparison of GAS with the more standard Intel notation. Note that GCC refers to these machines as "`i386`"—it generates code that could even run on a 1985 vintage machine.

Muchnick's book on compiler design [52] is considered the most comprehensive reference on code optimization techniques. It covers many of the techniques we discuss here, such as register usage conventions and the advantages of generating code for loops based on their `do-while` form.

Much has been written about the use of buffer overflow to attack systems over the Internet. Detailed analyses of the 1988 Internet worm have been published by Spafford [69] as well as by members of the team at MIT who helped stop its spread [24]. Since then, a number of papers and projects have generated about both creating and preventing buffer overflow attacks, such as [19].

Homework Problems

Homework Problem 3.31 [Category 1]:

You are given the following information. A function with prototype

```
int decode2(int x, int y, int z);
```

is compiled into assembly code. The body of the code is as follows:

```
1   movl 16(%ebp),%eax
2   movl 12(%ebp),%edx
3   subl %eax,%edx
4   movl %edx,%eax
5   imull 8(%ebp),%edx
6   sall $31,%eax
7   sarl $31,%eax
8   xorl %edx,%eax
```

Parameters `x`, `y`, and `z` are stored at memory locations with offsets 8, 12, and 16 relative to the address in register `%ebp`. The code stores the return value in register `%eax`.

Write C code for `decode2` that will have an effect equivalent to our assembly code. You can test your solution by compiling your code with the `-S` switch. Your compiler may not generate identical code, but it should be functionally equivalent.

Homework Problem 3.32 [Category 2]:

The following C code is almost identical to that in Figure 3.11:

```
1 int absdiff2(int x, int y)
2 {
3     int result;
4
5     if (x < y)
6         result = y-x;
7     else
8         result = x-y;
9     return result;
10 }
```

When compiled, however, it gives a different form of assembly code:

```
1   movl 8(%ebp),%edx
2   movl 12(%ebp),%ecx
3   movl %edx,%eax
4   subl %ecx,%eax
5   cmpl %ecx,%edx
6   jge .L3
7   movl %ecx,%eax
8   subl %edx,%eax
9 .L3:
```

A. What subtractions are performed when $x < y$? When $x \geq y$?

B. In what way does this code deviate from the standard implementation of if-else described previously?

C. Using C syntax (including goto's), show the general form of this translation.

D. What restrictions must be imposed on the use of this translation to guarantee that it has the behavior specified by the C code?

```
    The jump targets
    Arguments p1 and p2 are in registers %ebx and %ecx.
 1 .L15:                MODE_A
 2   movl (%ecx),%edx
 3   movl (%ebx),%eax
 4   movl %eax,(%ecx)
 5   jmp .L14
 6   .p2align 4,,7      Inserted to optimize cache performance
 7 .L16:                MODE_B
 8   movl (%ecx),%eax
 9   addl (%ebx),%eax
10   movl %eax,(%ebx)
11   movl %eax,%edx
12   jmp .L14
13   .p2align 4,,7      Inserted to optimize cache performance
14 .L17:                MODE_C
15   movl $15,(%ebx)
16   movl (%ecx),%edx
17   jmp .L14
18   .p2align 4,,7      Inserted to optimize cache performance
19 .L18:                MODE_D
20   movl (%ecx),%eax
21   movl %eax,(%ebx)
22 .L19:                MODE_E
23   movl $17,%edx
24   jmp .L14
25   .p2align 4,,7      Inserted to optimize cache performance
26 .L20:
27   movl $-1,%edx
28 .L14:                default
29   movl %edx,%eax     Set return value
```

Figure 3.36: **Assembly Code for Problem 3.33.** This code implements the different branches of a `switch` statement.

Homework Problem 3.33 [Category 2]:

The following code shows an example of branching on an enumerated type value in a switch statement. Recall that enumerated types in C are simply a way to introduce a set of names having associated integer values. By default, the values assigned to the names go from 0 upward. In our code, the actions associated with the different case labels have been omitted.

```
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

int switch3(int *p1, int *p2, mode_t action)
{
```

```
  int result = 0;
  switch(action) {
  case MODE_A:

  case MODE_B:

  case MODE_C:

  case MODE_D:

  case MODE_E:

  default:

  }
  return result;
}
```

The part of the generated assembly code implementing the different actions is shown shown in Figure 3.36. The annotations indicate the values stored in the registers and the case labels for the different jump destinations.

A. What register corresponds to program variable `result`?

B. Fill in the missing parts of the C code. Watch out for cases that fall through.

Homework Problem 3.34 [Category 2]:

Switch statements are particularly challenging to reverse engineer from the object code. In the following procedure, the body of the switch statement has been removed.

```
 1 int switch_prob(int x)
 2 {
 3     int result = x;
 4
 5     switch(x) {
 6
 7         /* Fill in code here */
 8     }
 9
10     return result;
11 }
```

Figure 3.37 shows the disassembled object code for the procedure. We are only interested in the part of code shown on lines 4 through 16. We can see on line 4 that parameter `x` (at offset 8 relative to `%ebp`) is loaded into register `%eax`, corresponding to program variable `result`. The "`lea 0x0(%esi),%esi`"

```
 1 080483c0 <switch_prob>:
 2  80483c0:  55                      push   %ebp
 3  80483c1:  89 e5                   mov    %esp,%ebp
 4  80483c3:  8b 45 08                mov    0x8(%ebp),%eax
 5  80483c6:  8d 50 ce                lea    0xffffffce(%eax),%edx
 6  80483c9:  83 fa 05                cmp    $0x5,%edx
 7  80483cc:  77 1d                   ja     80483eb <switch_prob+0x2b>
 8  80483ce:  ff 24 95 68 84 04 08    jmp    *0x8048468(,%edx,4)
 9  80483d5:  c1 e0 02                shl    $0x2,%eax
10  80483d8:  eb 14                   jmp    80483ee <switch_prob+0x2e>
11  80483da:  8d b6 00 00 00 00       lea    0x0(%esi),%esi
12  80483e0:  c1 f8 02                sar    $0x2,%eax
13  80483e3:  eb 09                   jmp    80483ee <switch_prob+0x2e>
14  80483e5:  8d 04 40                lea    (%eax,%eax,2),%eax
15  80483e8:  0f af c0                imul   %eax,%eax
16  80483eb:  83 c0 0a                add    $0xa,%eax
17  80483ee:  89 ec                   mov    %ebp,%esp
18  80483f0:  5d                      pop    %ebp
19  80483f1:  c3                      ret
20  80483f2:  89 f6                   mov    %esi,%esi
```

Figure 3.37: **Disassembled Code for Problem 3.34.**

instruction on line 11 is a nop instruction inserted to make the instruction on line 12 start on an address that is a multiple of 16.

The jump table resides in a different area of memory. Using the debugger GDB we can examine the six 4-byte words of memory starting at address `0x8048468` with the command `x/6w 0x8048468`. GDB prints the following:

```
(gdb) x/6w 0x8048468
0x8048468:  0x080483d5      0x080483eb      0x080483d5      0x080483e0
0x8048478:  0x080483e5      0x080483e8
(gdb)
```

Fill in the body of the switch statement with C code that will have the same behavior as the object code.

Homework Problem 3.35 [Category 2]:

The code generated by the C compiler for `var_prod_ele` (Figure 3.24(b)) is not optimal. Write code for this function based on a hybrid of procedures `fix_prod_ele_opt` (Figure 3.23) and `var_prod_ele_opt` (Figure 3.24) that is correct for all values of `n`, but compiles into code that can keep all of its temporary data in registers.

Recall that the processor only has six registers available to hold temporary data, since registers `%ebp` and `%esp` cannot be used for this purpose. One of these registers must be used to hold the result of the multiply instruction. Hence, you must reduce the number of local variables in the loop from six (`result`, `Aptr`, `B`, `nTjPk`, `n`, and `cnt`) to five.

Homework Problem 3.36 [Category 2]:

You are charged with maintaining a large C program, and you come across the following code:

code/asm/structprob-ans.c

```
1 typedef struct {
2     int left;
3     a_struct a[CNT];
4     int right;
5 } b_struct;
6
7 void test(int i, b_struct *bp)
8 {
9     int n = bp->left + bp->right;
10    a_struct *ap = &bp->a[i];
11    ap->x[ap->idx] = n;
12 }
```

code/asm/structprob-ans.c

Unfortunately, the '`.h`' file defining the compile-time constant `CNT` and the structure `a_struct` are in files for which you do not have access privileges. Fortunately, you have access to a '`.o`' version of code, which you are able to disassemble with the `objdump` program, yielding the disassembly shown in Figure 3.38.

Using your reverse engineering skills, deduce the following:

A. The value of `CNT`.

B. A complete declaration of structure `a_struct`. Assume that the only fields in this structure are `idx` and `x`.

Homework Problem 3.37 [Category 1]:

Write a function `good_echo` that reads a line from standard input and writes it to standard output. Your implementation should work for an input line of arbitrary length. You may use the library function `fgets`, but you must make sure your function works correctly even when the input line requires more space than you have allocated for your buffer. Your code should also check for error conditions and return when one is encounted. You should refer to the definitions of the standard I/O functions for documentation [30, 37].

Homework Problem 3.38 [Category 3]:

In this problem, you will mount a buffer overflow attack on your own program. As stated earlier, we do not condone using this or any other form of attack to gain unauthorized access to a system, but by doing this exercise, you will learn a lot about machine-level programming.

Download the file `bufbomb.c` from the CS:APP website and compile it to create an executable program. In `bufbomb.c`, you will find the following functions:

```
1 int getbuf()
```

```
2 {
3     char buf[12];
4     getxs(buf);
5     return 1;
6 }
7
8 void test()
9 {
10   int val;
11   printf("Type Hex string:");
12   val = getbuf();
13   printf("getbuf returned 0x%x\n", val);
14 }
```

The function `getxs` (also in `bufbomb.c`) is similar to the library `gets`, except that it reads characters encoded as pairs of hex digits. For example, to give it a string "`0123`," the user would type in the string "`30 31 32 33`." The function ignores blank characters. Recall that decimal digit x has ASCII representation `0x3`x.

A typical execution of the program is as follows:

```
unix> ./bufbomb
Type Hex string: 30 31 32 33
getbuf returned 0x1
```

Looking at the code for the `getbuf` function, it seems quite apparent that it will return value 1 whenever it is called. It appears as if the call to `getxs` has no effect. Your task is to make `getbuf` return -559038737 (`0xdeadbeef`) to `test`, simply by typing an appropriate hexadecimal string to the prompt.

Here are some ideas that will help you solve the problem:

- Use OBJDUMP to create a disassembled version of `bufbomb`. Study this closely to determine how the stack frame for `getbuf` is organized and how overflowing the buffer will alter the saved program state.
- Run your program under GDB. Set a breakpoint within `getbuf` and run to this breakpoint. Determine such parameters as the value of `%ebp` and the saved value of any state that will be overwritten when you overflow the buffer.
- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with GCC and disassemble it with OBJDUMP. You should be able to get the exact byte sequence that you will type at the prompt. OBJDUMP will produce some pretty strange looking assembly instructions when it tries to disassemble the data in your file, but the hexadecimal byte sequence should be correct.

Keep in mind that your attack is very machine and compiler specific. You may need to alter your string when running on a different machine or with a different version of GCC.

```
1  00000000 <test>:
2     0:   55                      push   %ebp
3     1:   89 e5                   mov    %esp,%ebp
4     3:   53                      push   %ebx
5     4:   8b 45 08                mov    0x8(%ebp),%eax
6     7:   8b 4d 0c                mov    0xc(%ebp),%ecx
7     a:   8d 04 80                lea    (%eax,%eax,4),%eax
8     d:   8d 44 81 04             lea    0x4(%ecx,%eax,4),%eax
9    11:   8b 10                   mov    (%eax),%edx
10   13:   c1 e2 02                shl    $0x2,%edx
11   16:   8b 99 b8 00 00 00       mov    0xb8(%ecx),%ebx
12   1c:   03 19                   add    (%ecx),%ebx
13   1e:   89 5c 02 04             mov    %ebx,0x4(%edx,%eax,1)
14   22:   5b                      pop    %ebx
15   23:   89 ec                   mov    %ebp,%esp
16   25:   5d                      pop    %ebp
17   26:   c3                      ret
```

Figure 3.38: **Disassembled Code For Problem 3.36.**

Homework Problem 3.39 [Category 2]:

Use the `asm` statement to implement a function with the following prototype:

```
void full_umul(unsigned x, unsigned y, unsigned dest[]);
```

This function should compute the full 64-bit product of its arguments and store the results in the destination array, with `dest[0]` having the low-order 4 bytes and `dest[1]` having the high-order 4 bytes.

Homework Problem 3.40 [Category 2]:

The `fscale` instruction computes the function $x \cdot 2^{RTZ(y)}$ for floating-point values x and y, where RTZ denotes the round-toward-zero function, rounding positive numbers downward and negative numbers upward. The arguments to `fscale` come from the floating-point register stack, with x in `%st(0)` and y in `%st(1)`. It writes the computed value written `%st(0)` without popping the second argument. (The actual implementation of this instruction works by adding $RTZ(y)$ to the exponent of x).

Using an `asm` statement, implement a function with the following prototype

```
double scale(double x, int n, double *dest);
```

that computes $x \cdot 2^n$ using the `fscale` instruction and stores the result at the location designated by pointer `dest`.

Hint: Extended `asm` does not provide very good support for IA32 floating point. In this case, however, you can access the arguments from the program stack.

Chapter 4

Processor Architecture

To appear in the final version of the manuscript.

Chapter 5

Optimizing Program Performance

Writing an efficient program requires two types of activities. First, we must select the best set of algorithms and data structures. Second, we must write source code that the compiler can effectively optimize to turn into efficient executable code. For this second part, it is important to understand the capabilities and limitations of optimizing compilers. Seemingly minor changes in how a program is written can make large differences in how well a compiler can optimize it. Some programming languages are more easily optimized than others. Some features of C, such as the ability to perform pointer arithmetic and casting, make it challenging to optimize. Programmers can often write their programs in ways that make it easier for compilers to generate efficient code.

In approaching the issue of program development and optimization, we must consider how the code will be used and what critical factors affect it. In general, programmers must make a trade-off between how easy a program is to implement and maintain, and how fast it will run. At an algorithmic level, a simple insertion sort can be programmed in a matter of minutes, whereas a highly efficient sort routine may take a day or more to implement and optimize. At the coding level, many low-level optimizations tend to reduce code readability and modularity. This makes the programs more susceptible to bugs and more difficult to modify or extend. For a program that will just be run once to generate a set of data points, it is more important to write it in a way that minimizes programming effort and ensures correctness. For code that will be executed repeatedly in a performance-critical environment, such as in a network router, much more extensive optimization may be appropriate.

In this chapter, we describe a number of techniques for improving code performance. Ideally, a compiler would be able to take whatever code we write and generate the most efficient possible machine-level program having the specified behavior. In reality, compilers can only perform limited transformations of the program, and they can be thwarted by *optimization blockers*—aspects of the program whose behavior depends strongly on the execution environment. Programmers must assist the compiler by writing code that can be optimized readily. In the compiler literature, optimization techniques are classified as either "machine independent," meaning that they should be applied regardless of the characteristics of the computer that will execute the code, or as "machine dependent," meaning they depend on many low-level details of the machine. We organize our presentation along similar lines, starting with program transformations that should be standard practice when writing any program. We then progress to transformations whose efficacy depends on the characteristics of the target machine and compiler. These transformations also tend to reduce

the modularity and readability of the code and hence should be applied when maximum performance is the dominant concern.

To maximize the performance of a program, both the programmer and the compiler need to have a model of the target machine, specifying how instructions are processed and the timing characteristics of the different operations. For example, the compiler must know timing information to be able to decide whether it is should use a multiply instruction or some combinations of shifts and adds. Modern computers use sophisticated techniques to process a machine-level program, executing many instructions in parallel and possibly in a different order than they appear in the program. Programmers must understand how these processors work to be able to tune their programs for maximum speed. We present a high-level model of such a machine based on some recent models of Intel processors. We devise a graphical notation that can be used to visualize the execution of instructions on the processor and to predict program performance.

We conclude by discussing issues related to optimizing large programs. We describe the use of code *profilers*—tools that measure the performance of different parts of a program. This analysis can help find inefficiencies in the code and identify the parts of the program on which we should focus our optimization efforts. Finally, we present an important observation, known as *Amdahl's Law* quantifying the overall effect of optimizing some portion of a system.

In this presentation, we make code optimization look like a simple, linear process of applying a series of transformations to the code in a particular order. In fact, the task is not nearly so straightforward. A fair amount of trial-and-error experimentation is required. This is especially true as we approach the later optimization stages, where seemingly small changes can cause major changes in performance, while some very promising techniques prove ineffective. As we will see in the examples, it can be difficult to explain exactly why a particular code sequence has a particular execution time. Performance can depend on many detailed features of the processor design for which we have relatively little documentation or understanding. This is another reason to try a number of different variations and combinations of techniques.

Studying the assembly code is one of the most effective means of gaining some understanding of the compiler and how the generated code will run. A good strategy is to start by looking carefully at the code for the inner loops. One can identify performance-reducing attributes such as excessive memory references and poor use of registers. Starting with the assembly code, we can even predict what operations will be performed in parallel and how well they will use the processor resources.

5.1 Capabilities and Limitations of Optimizing Compilers

Modern compilers employ sophisticated algorithms to determine what values are computed in a program and how they are used. They can then exploit opportunities to simplify expressions, to use a single computation in several different places, and to reduce the number of times a given computation must be performed. Unfortunately, optimizing compilers have limitations, due to constraints imposed on their behavior, to the limited understanding they have of the program's behavior and how it will be used, and to the requirement that they perform the compilation quickly.

Compiler optimization is supposed to be invisible to the user. When a programmer compiles code with optimization enabled (e.g., using the `-O` command line option), the code should have identical behavior as when compiled otherwise, except that it should run faster. This requirement restricts the ability of the

compiler to perform some types of optimizations.

Consider, for example, the following two procedures:

```
1 void twiddle1(int *xp, int *yp)
2 {
3     *xp += *yp;
4     *xp += *yp;
5 }
6
7 void twiddle2(int *xp, int *yp)
8 {
9     *xp += 2* *yp;
10 }
```

At first glance, both procedures seem to have identical behavior. They both add twice the value stored at the location designated by pointer `yp` to that designated by pointer `xp`. On the other hand, function `twiddle2` is more efficient. It requires only three memory references (read `*xp`, read `*yp`, write `*xp`), whereas `twiddle1` requires six (two reads of `*xp`, two reads of `*yp`, and two writes of `*xp`). Hence, if a compiler is given procedure `twiddle1` to compile, one might think it could generate more efficient code based on the computations performed by `twiddle2`.

Consider however, the case where `xp` and `yp` are equal. Then function `twiddle1` will perform the following computations:

```
3     *xp += *xp;  /* Double value at xp */
4     *xp += *xp;  /* Double value at xp */
```

The result will be that the value at `xp` will be increased by a factor of 4. On the other hand, function `twiddle2` will perform the following computation:

```
9     *xp += 2* *xp;  /* Triple value at xp */
```

The result will be that the value at `xp` will be increased by a factor of 3. The compiler knows nothing about how `twiddle1` will be called, and so it must assume that arguments `xp` and `yp` can be equal. Therefore it cannot generate code in the style of `twiddle2` as an optimized version of `twiddle1`.

This phenomenon is known as *memory aliasing*. The compiler must assume that different pointers may designate a single place in memory. This leads to one of the major *optimization blockers*, aspects of programs that can severely limit the opportunities for a compiler to generate optimized code.

Practice Problem 5.1:

The following problem illustrates the way memory aliasing can cause unexpected program behavior. Consider the following procedure to swap two values:

```
1 /* Swap value x at xp with value y at yp */
```

```
2 void swap(int *xp, int *yp)
3 {
4     *xp = *xp + *yp; /* x+y */
5     *yp = *xp - *yp; /* x+y-y = x */
6     *xp = *xp - *yp; /* x+y-x = y */
7 }
```

If this procedure is called with `xp` equal to `yp`, what effect will it have?

A second optimization blocker is due to function calls. As an example, consider the following two procedures:

```
1 int f(int);
2
3 int func1(x)
4 {
5     return f(x) + f(x) + f(x) + f(x);
6 }
7
8 int func2(x)
9 {
10    return 4*f(x);
11 }
```

It might seem at first that both compute the same result, but with `func2` calling `f` only once, whereas `func1` calls it four times. It is tempting to generate code in the style of `func2` when given `func1` as source.

Consider, however, the following code for `f`

```
1 int counter = 0;
2
3 int f(int x)
4 {
5     return counter++;
6 }
7
```

This function has a *side effect*—it modifies some part of the global program state. Changing the number of times it gets called changes the program behavior. In particular, a call to `func1` would return $0+1+2+3 = 6$, whereas a call to `func2` would return $4 \cdot 0 = 0$, assuming both started with global variable `counter` set to 0.

Most compilers do not try to determine whether a function is free of side effects and hence is a candidate for optimizations such as those attempted in `func2`. Instead, the compiler assumes the worst case and leaves all function calls intact.

Among compilers, the GNU compiler GCC is considered adequate, but not exceptional, in terms of its optimization capabilities. It performs basic optimizations but does not perform the radical transformations on programs that more "aggressive" compilers do. As a consequence, programmers using GCC must put more effort into writing programs in a way that simplifies the compiler's task of generating efficient code.

5.2 Expressing Program Performance

We need a way to express program performance that can guide us in improving the code. A useful measure for many programs is *Cycles Per Element* (CPE). This measure helps us understand the loop performance of an iterative program at a detailed level. Such a measure is appropriate for programs that perform a repetitive computation, such as processing the pixels in an image or computing the elements in a matrix product.

The sequencing of activities by a processor is controlled by a clock providing a regular signal of some frequency, expressed in either *Megahertz* (Mhz), i.e., millions of cycles per second, or *Gigahertz* (GHz), i.e., billions of cycles per second. For example, when product literature characterizes a system as a "1.4 GHz" processor, it means that the processor clock runs at 1,400 Megahertz. The time required for each clock cycle is given by the reciprocal of the clock frequency. These are typically expressed in *nanoseconds*, i.e., billionths of a second. A 2 GHz clock has a 0.5-nanosecond period, while a 500 Mhz clock has a period of 2 nanoseconds. From a programmer's perspective, it is more instructive to express measurements in clock cycles rather than nanoseconds. That way, the measurements are less dependent on the particular model of processor being evaluated, and they help us understand exactly how the program is being executed by the machine.

Many procedures contain a loop that iterates over a set of elements. For example, functions `vsum1` and `vsum2` in Figure 5.1 both compute the sum of two vectors of length n. The first computes one element of the destination vector per iteration. The second uses a technique known as *loop unrolling* to compute two elements per iteration. This version will only work properly for even values of n. Later in this chapter we cover loop unrolling in more detail, including how to make it work for arbitrary values of n.

The time required by such a procedure can be characterized as a constant plus a factor proportional to the number of elements processed. For example, Figure 5.2 shows a plot of the number of clock cycles required by the two functions for a range of values of n. Using a *least squares fit*, we find that the two function run times (in clock cycles) can be approximated by lines with equations $80 + 4.0n$ and $83.5 + 3.5n$, respectively. These equations indicated an overhead of 80 to 84 cycles to initiate the procedure, set up the loop, and complete the procedure, plus a linear factor of 3.5 or 4.0 cycles per element. For large values of n (say greater than 50), the run times will be dominated by the linear factors. We refer to the coefficients in these terms as the effective number of *Cycles per Element*, abbreviated "CPE." Note that we prefer measuring the number of cycles per *element* rather than the number of cycles per *iteration*, because techniques such as loop unrolling allow us to use fewer iterations to complete the computation, but our ultimate concern is how fast the procedure will run for a given vector length. We focus our efforts on minimizing the CPE for our computations. By this measure, `vsum2`, with a CPE of 3.50, is superior to `vsum1`, with a CPE of 4.0.

Aside: What is a least squares fit?
For a set of data points $(x_1, y_1), \ldots (x_n, y_n)$, we often try to draw a line that best approximates the X-Y trend represented by this data. With a least squares fit, we look for a line of the form $y = mx + b$ that minimizes the

code/opt/vsum.c

```
1 void vsum1(int n)
2 {
3     int i;
4
5     for (i = 0; i < n; i++)
6         c[i] = a[i] + b[i];
7 }
8
9 /* Sum vector of n elements (n must be even) */
10 void vsum2(int n)
11 {
12     int i;
13
14     for (i = 0; i < n; i+=2) {
15         /* Compute two elements per iteration */
16         c[i]   = a[i]   + b[i];
17         c[i+1] = a[i+1] + b[i+1];
18     }
19 }
```

code/opt/vsum.c

Figure 5.1: **Vector Sum Functions.** These provide examples for how we express program performance.

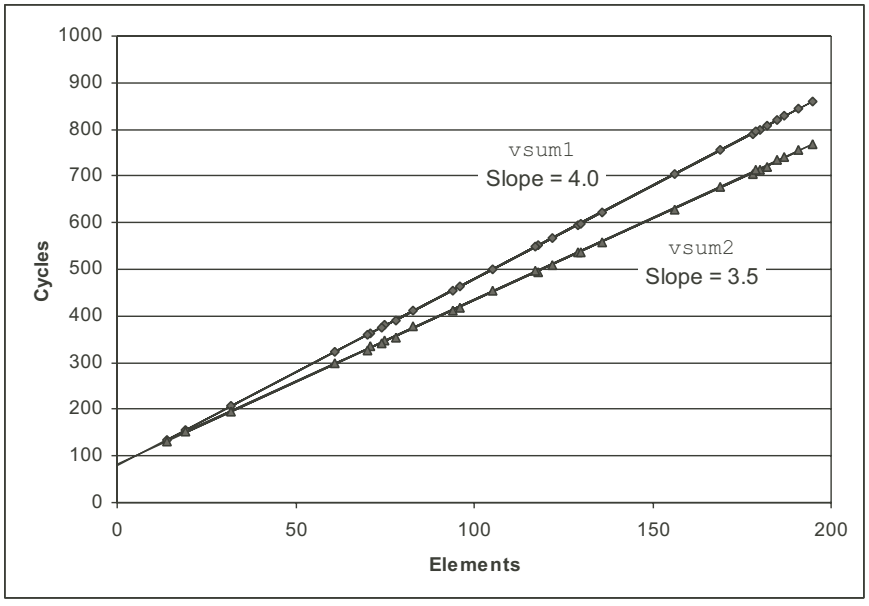


Figure 5.2: **Performance of Vector Sum Functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

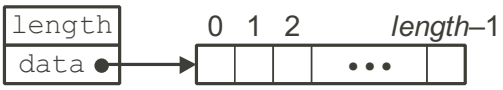


Figure 5.3: **Vector Abstract Data Type.** A vector is represented by header information plus array of designated length.

following error measure:

$$E(m,b) \quad = \quad \sum_{i=1,n} (mx_i + b - y_i)^2.$$

An algorithm for computing m and b can be derived by finding the derivatives of $E(m,b)$ with respect to m and b and setting them to 0. **End Aside.**

5.3 Program Example

To demonstrate how an abstract program can be systematically transformed into more efficient code, consider the simple vector data structure, shown in Figure 5.3. A vector is represented with two blocks of memory. The header is a structure declared as follows:

code/opt/vec.h

```
1 /* Create abstract data type for vector */
2 typedef struct {
3     int len;
4     data_t *data;
5 } vec_rec, *vec_ptr;
```

code/opt/vec.h

The declaration uses data type `data_t` to designate the data type of the underlying elements. In our evaluation, we measure the performance of our code for data types `int`, `float`, and `double`. We do this by compiling and running the program separately for different type declarations, for example:

```
typedef int data_t;
```

In addition to the header, we allocate an array of `len` objects of type `data_t` to hold the actual vector elements.

Figure 5.4 shows some basic procedures for generating vectors, accessing vector elements, and determining the length of a vector. An important feature to note is that `get_vec_element`, the vector access routine, performs bounds checking for every vector reference. This code is similar to the array representations used in many other languages, including Java. Bounds checking reduces the chances of program error, but, as we will see, significantly affects program performance.

As an optimization example, consider the code shown in Figure 5.5, which combines all of the elements in a vector into a single value according to some operation. By using different definitions of compile-time constants `IDENT` and `OPER`, the code can be recompiled to perform different operations on the data.

In particular, using the declarations

code/opt/vec.c

```
1  /* Create vector of specified length */
2  vec_ptr new_vec(int len)
3  {
4      /* allocate header structure */
5      vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
6      if (!result)
7          return NULL;  /* Couldn't allocate storage */
8      result->len = len;
9      /* Allocate array */
10     if (len > 0) {
11         data_t *data = (data_t *)calloc(len, sizeof(data_t));
12         if (!data) {
13             free((void *) result);
14             return NULL; /* Couldn't allocate storage */
15         }
16         result->data = data;
17     }
18     else
19         result->data = NULL;
20     return result;
21 }
22
23 /*
24  * Retrieve vector element and store at dest.
25  * Return 0 (out of bounds) or 1 (successful)
26  */
27 int get_vec_element(vec_ptr v, int index, data_t *dest)
28 {
29     if (index < 0 || index >= v->len)
30         return 0;
31     *dest = v->data[index];
32     return 1;
33 }
34
35 /* Return length of vector */
36 int vec_length(vec_ptr v)
37 {
38     return v->len;
39 }
```

code/opt/vec.c

Figure 5.4: **Implementation of Vector Abstract Data Type.** In the actual program, data type `data_t` is declared to be `int`, `float`, or `double`

.

code/opt/combine.c

```
1 /* Implementation with maximum use of data abstraction */
2 void combine1(vec_ptr v, data_t *dest)
3 {
4     int i;
5
6     *dest = IDENT;
7     for (i = 0; i < vec_length(v); i++) {
8         data_t val;
9         get_vec_element(v, i, &val);
10        *dest = *dest OPER val;
11    }
12 }
```

code/opt/combine.c

Figure 5.5: **Initial Implementation of Combining Operation.** Using different declarations of identity element *IDENT* and combining operation *OPER*, we can measure the routine for different operations.

```
#define IDENT 0
#define OPER  +
```

we sum the elements of the vector. Using the declarations:

```
#define IDENT 1
#define OPER  *
```

we compute the product of the vector elements.

As a starting point, here are the CPE measurements for `combine1` running on an Intel Pentium III, trying all combinations of data type and combining operation. In our measurements, we found that the timings were generally equal for single and double-precision floating point data. We therefore show only the measurements for single precision.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	211	Abstract unoptimized	42.06	41.86	41.44	160.00
`combine1`	211	Abstract `-O2`	31.25	33.25	31.25	143.00

By default, the compiler generates code suitable for stepping with a symbolic debugger. Very little optimization is performed since the intention is to make the object code closely match the computations indicated in the source code. By simply setting the command line switch to '`-O2`' we enable optimizations. As can be seen, this significantly improves the program performance. In general, it is good to get into the habit of enabling this level of optimization, unless the program is being compiled with the intention of debugging it. For the remainder of our measurements we enable this level of compiler optimization.

code/opt/combine.c

```
1 /* Move call to vec_length out of loop */
2 void combine2(vec_ptr v, data_t *dest)
3 {
4     int i;
5     int length = vec_length(v);
6
7     *dest = IDENT;
8     for (i = 0; i < length; i++) {
9         data_t val;
10        get_vec_element(v, i, &val);
11        *dest = *dest OPER val;
12    }
13 }
```

code/opt/combine.c

Figure 5.6: **Improving the Efficiency of the Loop Test.** By moving the call to `vec_length` out of the loop test, we eliminate the need to execute it on every iteration.

Note also that the times are fairly comparable for the different data types and the different operations, with the exception of floating-point multiplication. These very high cycle counts for multiplication are due to an anomaly in our benchmark data. Identifying such anomalies is an important component of performance analysis and optimization. We return to this issue in Section 5.11.1.

We will see that we can improve on this performance considerably.

5.4 Eliminating Loop Inefficiencies

Observe that procedure `combine1`, as shown in Figure 5.5, calls function `vec_length` as the test condition of the `for` loop. Recall from our discussion of loops that the test condition must be evaluated on every iteration of the loop. On the other hand, the length of the vector does not change as the loop proceeds. We could therefore compute the vector length only once and use this value in our test condition.

Figure 5.6 shows a modified version, called `combine2`, that calls `vec_length` at the beginning and assigns the result to a local variable `length`. This local variable is then used in the test condition of the for loop. Surprisingly, this small change has a significant effect on program performance.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	211	Abstract `-O2`	31.25	33.25	31.25	143.00
`combine2`	212	Move `vec_length`	22.61	21.25	21.15	135.00

As the table above shows, we eliminate around 10 clock cycles for each vector element with this simple transformation.

This optimization is an instance of a general class of optimizations known as *code motion*. They involve identifying a computation that is performed multiple times, (e.g., within a loop), but such that the result of the computation will not change. We can therefore move the computation to an earlier section of the code that does not get evaluated as often. In this case, we moved the call to `vec_length` from within the loop to just before the loop.

Optimizing compilers attempt to perform code motion. Unfortunately, as discussed previously, they are typically very cautious about making transformations that change where or how many times a procedure is called. They cannot reliably detect whether or not a function will have side effects, and so they assume that it might. For example, if `vec_length` had some side effect, then `combine1` and `combine2` could have different behaviors. In cases such as these, the programmer must help the compiler by explicitly performing the code motion.

As an extreme example of the loop inefficiency seen in `combine1`, consider the procedure `lower1` shown in Figure 5.7. This procedure is styled after routines submitted by several students as part of a network programming project. Its purpose is to convert all of the upper-case letters in a string to lower case. The procedure steps through the string, converting each upper-case character to lower case.

The library procedure `strlen` is called as part of the loop test of `lower1`. A simple version of `strlen` is also shown in Figure 5.7. Since strings in C are null-terminated character sequences, `strlen` must step through the sequence until it hits a null character. For a string of length n, `strlen` takes time proportional to n. Since `strlen` is called on each of the n iterations of `lower1`, the overall run time of `lower1` is quadratic in the string length.

This analysis is confirmed by actual measurements of the procedure for different length strings, as shown Figure 5.8. The graph of the run time for `lower1` rises steeply as the string length increases. The lower part of the figure shows the run times for eight different lengths (not the same as shown in the graph), each of which is a power of two. Observe that for `lower1` each doubling of the string length causes a quadrupling of the run time. This is a clear indicator of quadratic complexity. For a string of length 262,144, `lower1` requires a full 3.1 minutes of CPU time.

Function `lower2` shown in Figure 5.7 is identical to that of `lower1`, except that we have moved the call to `strlen` out of the loop. The performance improves dramatically. For a string length of 262,144, the function requires just 0.006 seconds—over 30,000 times faster than `lower1`. Each doubling of the string length causes a doubling of the run time—a clear indicator of linear complexity. For longer strings, the run time improvement will be even greater.

In an ideal world, a compiler would recognize that each call to `strlen` in the loop test will return the same result, and hence the call could be moved out of the loop. This would require a very sophisticated analysis, since `strlen` checks the elements of the string and these values are changing as `lower1` proceeds. The compiler would need to detect that even though the characters within the string are changing, none are being set from nonzero to zero, or vice-versa. Such an analysis is well beyond that attempted by even the most aggressive compilers. Programmers must do such transformations themselves.

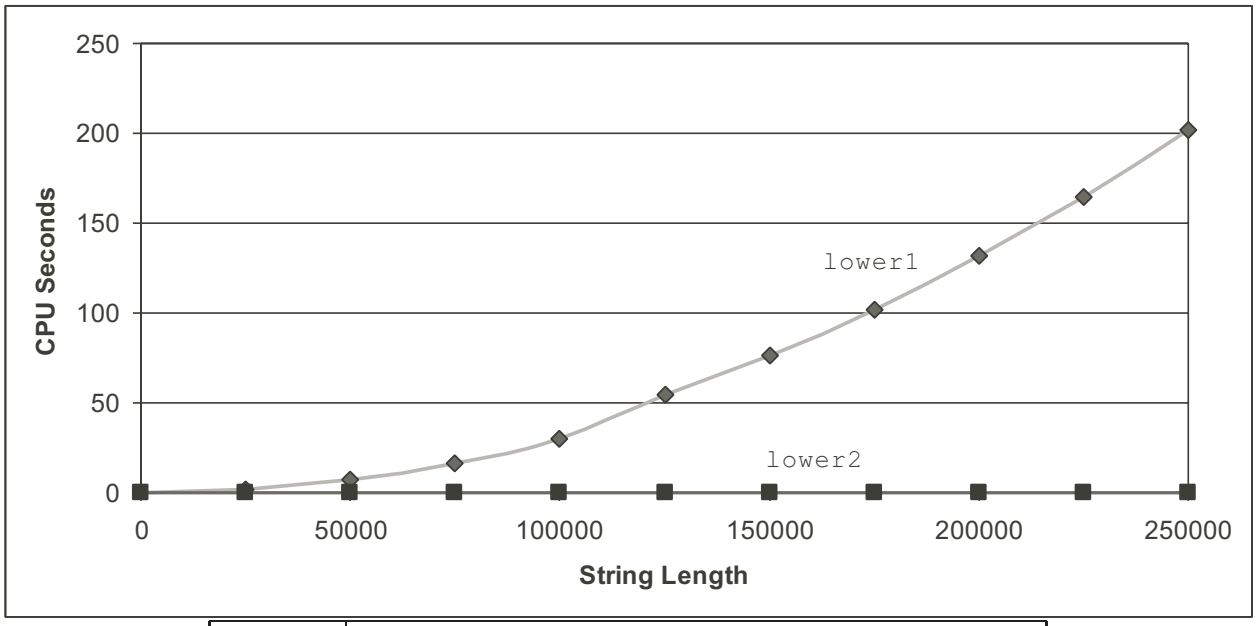
This example illustrates a common problem in writing programs, in which a seemingly trivial piece of code has a hidden asymptotic inefficiency. One would not expect a lower-case conversion routine to be a limiting factor in a program's performance. Typically, programs are tested and analyzed on small data sets, for which the performance of `lower1` is adequate. When the program is ultimately deployed, however, it is

code/opt/lower.c

```
1 /* Convert string to lower case: slow */
2 void lower1(char *s)
3 {
4     int i;
5
6     for (i = 0; i < strlen(s); i++)
7         if (s[i] >= 'A' && s[i] <= 'Z')
8             s[i] -= ('A' - 'a');
9 }
10
11 /* Convert string to lower case: faster */
12 void lower2(char *s)
13 {
14     int i;
15     int len = strlen(s);
16
17     for (i = 0; i < len; i++)
18         if (s[i] >= 'A' && s[i] <= 'Z')
19             s[i] -= ('A' - 'a');
20 }
21
22 /* Implementation of library function strlen */
23 /* Compute length of string */
24 size_t strlen(const char *s)
25 {
26     int length = 0;
27     while (*s != '\0') {
28         s++;
29         length++;
30     }
31     return length;
32 }
```

code/opt/lower.c

Figure 5.7: **Lower-Case Conversion Routines.** The two procedures have radically different performance.



Function	String Length					
	8,192	16,384	32,768	65,536	131,072	262,144
`lower1`	0.15	0.62	3.19	12.75	51.01	186.71
`lower2`	0.0002	0.0004	0.0008	0.0016	0.0031	0.0060

Figure 5.8: **Comparative Performance of Lower-Case Conversion Routines.** The original code `lower1` has quadratic asymptotic complexity due to an inefficient loop structure. The modified code `lower2` has linear complexity.

entirely possible that the procedure could be applied to a string of one million characters, for which `lower1` would over require nearly one hour of CPU time. All of a sudden this benign piece of code has become a major performance bottleneck. By contrast, `lower2` would complete in well under one second. Stories abound of major programming projects in which problems of this sort occur. Part of the job of a competent programmer is to avoid ever introducing such asymptotic inefficiency.

Practice Problem 5.2:

Consider the following functions:

```
int min(int x, int y) { return x < y ? x : y; }
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int square(int x) { return x*x; }
```

Here are three code fragments that call these functions

A.
```
for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);
```

B.
```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

C.
```
int low = min(x, y);
int high = max(x, y);

for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

Assume `x` equals 10 and `y` equals 100. Fill in the table below indicating the number of times each of the four functions is called for each of these code fragments.

Code	`min`	`max`	`incr`	`square`
A.				
B.				
C.				

5.5 Reducing Procedure Calls

As we have seen, procedure calls incur substantial overhead and block most forms of program optimization. We can see in the code for `combine2` (Figure 5.6) that `get_vec_element` is called on every loop iteration to retrieve the next vector element. This procedure is especially costly since it performs bounds checking. Bounds checking might be a useful feature when dealing with arbitrary array accesses, but a simple analysis of the code for `combine2` shows that all references will be valid.

Suppose instead that we add a function `get_vec_start` to our abstract data type. This function returns the starting address of the data array, as shown in Figure 5.9. We could then write the procedure shown as `combine3` in this figure, having no function calls in the inner loop. Rather than making a function call

code/opt/vec.c

```
1 data_t *get_vec_start(vec_ptr v)
2 {
3     return v->data;
4 }
```

code/opt/vec.c

code/opt/combine.c

```
1 /* Direct access to vector data */
2 void combine3(vec_ptr v, data_t *dest)
3 {
4     int i;
5     int length = vec_length(v);
6     data_t *data = get_vec_start(v);
7
8     *dest = IDENT;
9     for (i = 0; i < length; i++) {
10        *dest = *dest OPER data[i];
11    }
12 }
```

code/opt/combine.c

Figure 5.9: **Eliminating Function Calls within the Loop.** The resulting code runs much faster, at some cost in program modularity.

to retrieve each vector element, it accesses the array directly. A purist might say that this transformation seriously impairs the program modularity. In principle, the user of the vector abstract data type should not even need to know that the vector contents are stored as an array, rather than as some other data structure such as a linked list. A more pragmatic programmer would argue the advantage of this transformation based on the following experimental results:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine2`	212	Move `vec_length`	20.66	21.25	21.15	135.00
`combine3`	217	Direct data access	6.00	9.00	8.00	117.00

There is a improvement of up to a factor of 3.5X. For applications where performance is a significant issue, one must often compromise modularity and abstraction for speed. It is wise to include documentation on the transformations applied, and the assumptions that led to them, in case the code needs to be modified later.

Aside: Expressing relative performance.
The best way to express a performance improvement is as a ratio of the form T_{old}/T_{new}, where T_{old} is the time required for the original version and T_{new} is the time required by the modified version. This will be a number greater than 1.0 if any real improvement occurred. We use the suffix 'X' to indicate such a ratio, where the factor "3.5X" is expressed verbally as "3.5 times."

The more traditional way of expressing relative change as a percentage works well when the change is small, but its definition is ambiguous. Should it be $100 \cdot (T_{old} - T_{new})/T_{new}$ or possibly $100 \cdot (T_{old} - T_{new})/T_{old}$, or something else? In addition, it is less instructive for large changes. Saying that "performance improved by 250%" is more difficult to comprehend than simply saying that the performance improved by a factor of 3.5. **End Aside.**

5.6 Eliminating Unneeded Memory References

The code for `combine3` accumulates the value being computed by the combining operation at the location designated by pointer `dest`. This attribute can be seen by examining the assembly code generated for the compiled loop, with integers as the data type and multiplication as the combining operation. In this code, register `%ecx` points to `data`, `%edx` contains the value of `i`, and `%edi` points to `dest`.

```
    combine3: type=INT, OPER = *
    dest in %edi, data in %ecx, i in %edx, length in %esi
1 .L18:                                  loop:
2   movl (%edi),%eax                       Read *dest
3   imull (%ecx,%edx,4),%eax               Multiply by data[i]
4   movl %eax,(%edi)                       Write *dest
5   incl %edx                              i++
6   cmpl %esi,%edx                         Compare i:length
7   jl .L18                                If <, goto loop
```

Instruction 2 reads the value stored at `dest` and instruction 4 writes back to this location. This seems wasteful, since the value read by instruction 1 on the next iteration will normally be the value that has just been written.

code/opt/combine.c

```
1 /* Accumulate result in local variable */
2 void combine4(vec_ptr v, data_t *dest)
3 {
4     int i;
5     int length = vec_length(v);
6     data_t *data = get_vec_start(v);
7     data_t x = IDENT;
8
9     *dest = IDENT;
10    for (i = 0; i < length; i++) {
11        x = x OPER data[i];
12    }
13    *dest = x;
14 }
```

code/opt/combine.c

Figure 5.10: **Accumulating Result in Temporary.** This eliminates the need to read and write intermediate values on every loop iteration.

This leads to the optimization shown as `combine4` in Figure 5.10 where we introduce a temporary variable `x` that is used in the loop to accumulate the computed value. The result is stored at `*dest` only after the loop has been completed. As the following assembly code for the loop shows, the compiler can now use register `%eax` to hold the accumulated value. Comparing to the loop for `combine3`, we have reduced the memory operations per iteration from two reads and one write to just a single read. Registers `%ecx` and `%edx` are used as before, but there is no need to reference `*dest`.

```
    combine4: type=INT, OPER = *
    data in %eax, x in %ecx, i in %edx, length in %esi
1 .L24:                                  loop:
2   imull (%eax,%edx,4),%ecx               Multiply x by data[i]
3   incl %edx                              i++
4   cmpl %esi,%edx                         Compare i:length
5   jl .L24                                If <, goto loop
```

We see a significant improvement in program performance:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine3`	217	Direct data access	6.00	9.00	8.00	117.00
`combine4`	219	Accumulate in temporary	2.00	4.00	3.00	5.00

The most dramatic decline is in the time for floating-point multiplication. Its time becomes comparable to the times for the other combinations of data type and operation. We will examine the cause for this sudden decrease in Section 5.11.1.

Again, one might think that a compiler should be able to automatically transform the `combine3` code shown in Figure 5.9 to accumulate the value in a register, as it does with the code for `combine4` shown in Figure 5.10.

In fact, however, the two functions can have different behavior due to memory aliasing. Consider, for example, the case of integer data with multiplication as the operation and 1 as the identity element. Let `v` be a vector consisting of the three elements $[2, 3, 5]$ and consider the following two function calls:

```
combine3(v, get_vec_start(v) + 2);
combine4(v, get_vec_start(v) + 2);
```

That is, we create an alias between the last element of the vector and the destination for storing the result. The two functions would then execute as follows:

Function	Initial	Before Loop	`i = 0`	`i = 1`	`i = 2`	Final
`combine3`	$[2, 3, 5]$	$[2, 3, 1]$	$[2, 3, 2]$	$[2, 3, 6]$	$[2, 3, 36]$	$[2, 3, 36]$
`combine4`	$[2, 3, 5]$	$[2, 3, 5]$	$[2, 3, 5]$	$[2, 3, 5]$	$[2, 3, 5]$	$[2, 3, 30]$

As shown above, `combine3` accumulates its result at the destination, which in this case is the final vector element. This value is therefore set first to 1, then to $2 \cdot 1 = 2$, and then to $3 \cdot 2 = 6$. On the final iteration this value is then multiplied by itself to yield a final value of 36. For the case of `combine4`, the vector remains unchanged until the end, when the final element is set to the computed result $1 \cdot 2 \cdot 3 \cdot 5 = 30$.

Of course, our example showing the distinction between `combine3` and `combine4` is highly contrived. One could argue that the behavior of `combine4` more closely matches the intention of the function description. Unfortunately, an optimizing compiler cannot make a judgement about the conditions under which a function might be used and what the programmer's intentions might be. Instead, when given `combine3` to compile, it is obligated to preserve its exact functionality, even if this means generating inefficient code.

5.7 Understanding Modern Processors

Up to this point, we have applied optimizations that did not rely on any features of the target machine. They simply reduced the overhead of procedure calls and eliminated some of the critical "optimization blockers" that cause difficulties for optimizing compilers. As we seek to push the performance further, we must begin to consider optimizations that make more use of the means by which processors execute instructions and the capabilities of particular processors. Getting every last bit of performance requires a detailed analysis of the program as well as code generation tuned for the target processor. Nonetheless, we can apply some basic optimizations that will yield an overall performance improvement on a large class of processors. The detailed performance results we report here may not hold for other machines, but the general principles of operation and optimization apply to a wide variety of machines.

To understand ways to improve performance, we require a simple operational model of how modern processors work. Due to the large number of transistors that can be integrated onto a single chip, modern microprocessors employ complex hardware that attempts to maximize program performance. One result is that their actual operation is far different from the view that is perceived by looking at assembly-language programs. At the assembly-code level, it appears as if instructions are executed one at a time, where each

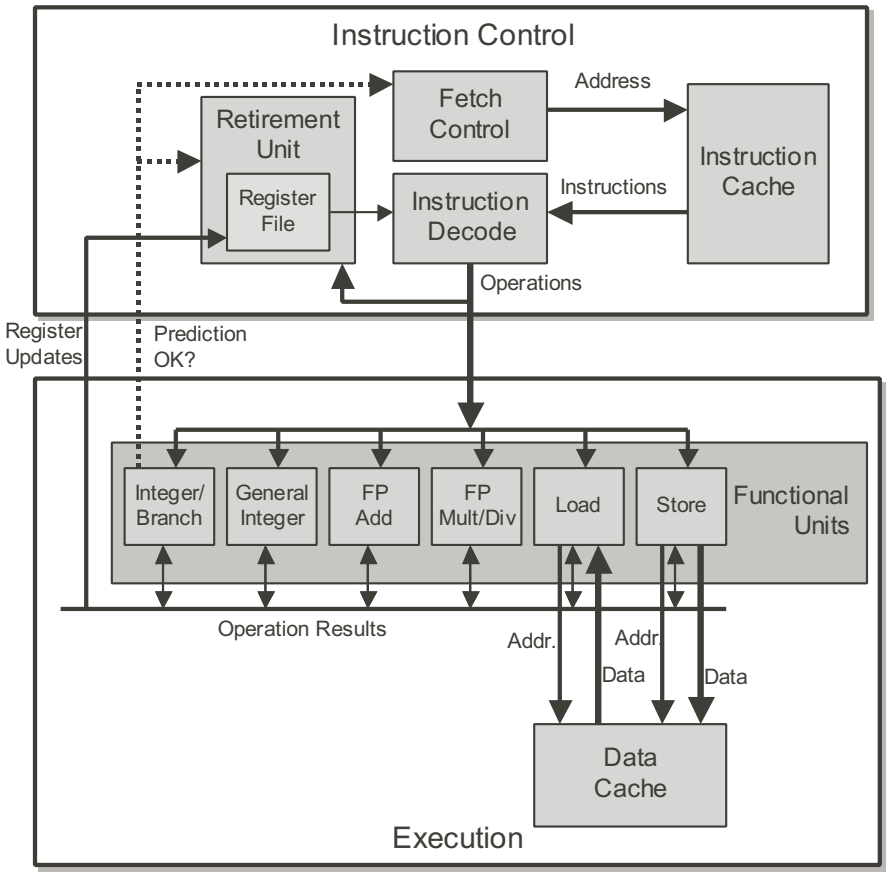


Figure 5.11: **Block Diagram of a Modern Processor.** The Instruction Control Unit is responsible for reading instructions from memory and generating a sequence of primitive operations. The Execution Unit then performs the operations and indicates whether the branches were correctly predicted.

instruction involves fetching values from registers or memory, performing an operation, and storing results back to a register or memory location. In the actual processor, a number of instructions are evaluated simultaneously. In some designs, there can be 80 or more instructions "in flight." Elaborate mechanisms are employed to make sure the behavior of this parallel execution exactly captures the sequential semantic model required by the machine-level program.

5.7.1 Overall Operation

Figure 5.11 shows a very simplified view of a modern microprocessor. Our hypothetical processor design is based loosely on the Intel "P6" microarchitecture [28], the basis for the Intel PentiumPro, Pentium II and Pentium III processors. The newer Pentium 4 has a different microarchitecture, but it has a similar overall structure to the one we present here. The P6 microarchitecture typifies the high-end processors produced by a number of manufacturers since the late 1990s. It is described in the industry as being *superscalar*, which means it can perform multiple operations on every clock cycle, and *out-of-order* meaning that the

order in which instructions execute need not correspond to their ordering in the assembly program. The overall design has two main parts. The *Instruction Control Unit* (ICU) is responsible for reading a sequence of instructions from memory and generating from these a set of primitive operations to perform on program data. The *Execution Unit* (EU) then executes these operations.

The ICU reads the instructions from an *instruction cache*—a special, high-speed memory containing the most recently accessed instructions. In general, the ICU fetches well ahead of the currently executing instructions, so that it has enough time to decode these and send operations down to the EU. One problem, however, is that when a program hits a branch,[1] there are two possible directions the program might go. The branch can be *taken*, with control passing to the branch target. Alternatively, the branch can be *not taken*, with control passing to the next instruction in the instruction sequence. Modern processors employ a technique known as *branch prediction*, where they guess whether or not a branch will be taken, and they also predict the target address for the branch. Using a technique known as *speculative execution*, the processor begins fetching and decoding instructions at where it predicts the branch will go, and even begins executing these operations before it has been determined whether or not the branch prediction was correct. If it later determines that the branch was predicted incorrectly, it resets the state to that at the branch point and begins fetching and executing instructions in the other direction. A more exotic technique would be to begin fetching and executing instructions for both possible directions, later discarding the results for the incorrect direction. To date, this approach has not been considered cost effective. The block labeled *Fetch Control* incorporates branch prediction to perform the task of determining which instructions to fetch.

The *Instruction Decoding* logic takes the actual program instructions and converts them into a set of primitive *operations*. Each of these operations performs some simple computational task such as adding two numbers, reading data from memory, or writing data to memory. For machines with complex instructions, such as an IA32 processor, an instruction can be decoded into a variable number of operations. The details vary from one processor design to another, but we attempt to describe a typical implementation. In this machine, decoding the instruction

```
addl %eax,%edx
```

yields a single addition operation, whereas decoding the instruction

```
addl %eax,4(%edx)
```

yields three operations: one to *load* a value from memory into the processor, one to add the loaded value to the value in register `%eax`, and one to *store* the result back to memory. This decoding splits instructions to allow a division of labor among a set of dedicated hardware units. These units can then execute the different parts of multiple instructions in parallel. For machines with simple instructions, the operations correspond more closely to the original instructions.

The EU receives operations from the instruction fetch unit. Typically, it can receive a number of them on each clock cycle. These operations are dispatched to a set of *functional units* that perform the actual operations. These functional units are specialized to handle specific types of operations. Our figure illustrates a typical set of functional units. It is styled after those found in recent Intel processors. The units in the figure are as follows:

[1] We use the term "branch" specifically to refer to conditional jump instructions. Other instructions that can transfer control to multiple destinations, such as procedure return and indirect jumps, provide similar challenges for the processor.

Integer/Branch: Performs simple integer operations (add, test, compare, logical). Also processes branches, as is discussed below.

General Integer: Can handle all integer operations, including multiplication and division.

Floating-Point Add: Handles simple floating-point operations (addition, format conversion).

Floating-Point Multiplication/Division: Handles floating-point multiplication and division. More complex floating-point instructions, such transcendental functions, are converted into sequences of operations.

Load: Handles operations that read data from the memory into the processor. The functional unit has an adder to perform address computations.

Store: Handles operations that write data from the processor to the memory. The functional unit has an adder to perform address computations.

As shown in the figure, the load and store units access memory via a *data cache*, a high-speed memory containing the most recently accessed data values.

With speculative execution, the operations are evaluated, but the final results are not stored in the program registers or data memory until the processor can be certain that these instructions should actually have been executed. Branch operations are sent to the EU not to determine where the branch should go, but rather to determine whether or not they were predicted correctly. If the prediction was incorrect, the EU will discard the results that have been computed beyond the branch point. It will also signal to the Branch Unit that the prediction was incorrect and indicate the correct branch destination. In this case the Branch Unit begins fetching at the new location. Such a *misprediction* incurs a significant cost in performance. It takes a while before the new instructions can be fetched, decoded, and sent to the execution units. We explore this further in Section 5.12.

Within the ICU, the *Retirement Unit* keeps track of the ongoing processing and makes sure that it obeys the sequential semantics of the machine-level program. Our figure shows a *Register File*, containing the integer and floating-point registers, as part of the Retirement Unit, because this unit controls the updating of these registers. As an instruction is decoded, information about it is placed in a first-in, first-out queue. This information remains in the queue until one of two outcomes occurs. First, once the operations for the instruction have completed and any branch points leading to this instruction are confirmed as having been correctly predicted, the instruction can be *retired*, with any updates to the program registers being made. If some branch point leading to this instruction was mispredicted, on the other hand, the instruction will be *flushed*, discarding any results that may have been computed. By this means, mispredictions will not alter the program state.

As we have described, any updates to the program registers occur only as instructions are being retired, and this takes place only after the processor can be certain that any branches leading to this instruction have been correctly predicted. To expedite the communication of results from one instruction to another, much of this information is exchanged among the execution units, shown in the figure as "Operation Results." As the arrows in the figure show, the execution units can send results directly to each other.

The most common mechanism for controlling the communication of operands among the execution units is called *register renaming*. When an instruction that updates register r is decoded, a *tag* t is generated

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-Point Add	3	1
Floating-Point Multiply	5	2
Floating-Point Divide	38	38
Load (Cache Hit)	3	1
Store (Cache Hit)	3	1

Figure 5.12: **Performance of Pentium III Arithmetic Operations.** Latency represents the total number of cycles for a single operation. Issue time denotes the number of cycles between successive, independent operations. (Obtained from Intel literature).

giving a unique identifier to the result of the operation. An entry (r, t) is added to a table maintaining the association between each program register and the tag for an operation that will update this register. When a subsequent instruction using register r as an operand is decoded, the operation sent to the Execution Unit will contain t as the source for the operand value. When some execution unit completes the first operation, it generates a result (v, t) indicating that the operation with tag t produced value v. Any operation waiting for t as a source will then use v as the source value. By this mechanism, values can be passed directly from one operation to another, rather than being written to and read from the register file. The renaming table only contains entries for registers having pending write operations. When a decoded instruction requires a register r, and there is no tag associated with this register, the operand is retrieved directly from the register file. With register renaming, an entire sequence of operations can be performed speculatively, even though the registers are updated only after the processor is certain of the branch outcomes.

5.7.2 Functional Unit Performance

Figure 5.12 documents the performance of some of basic operations for an Intel Pentium III. These timings are typical for other processors as well. Each operation is characterized by two cycle counts: the *latency*, indicating the total number of cycles the functional unit requires to complete the operation; and the *issue time*, indicating the number of cycles between successive, independent operations. The latencies range from one cycle for basic integer operations; several cycles for loads, stores, integer multiplication, and the more common floating-point operations; and then to many cycles for division and other complex operations.

As the third column in Figure 5.12 shows, several functional units of the processor are *pipelined*, meaning that they can start on a new operation before the previous one is completed. The issue time indicates the number of cycles between successive operations for the unit. In a pipelined unit, the issue time is smaller than the latency. A pipelined function unit is implemented as a series of stages, each of which performs part of the operation. For example, a typical floating-point adder contains three stages: one to process the exponent values, one to add the fractions, and one to round the final result. The operations can proceed through the stages in close succession rather than waiting for one operation to complete before the next begins. This capability can only be exploited if there are successive, logically independent operations to

be performed. As indicated, most of the units can begin a new operation on every clock cycle. The only exceptions are the floating-point multiplier, which requires a minimum of two cycles between successive operations, and the two dividers, which are not pipelined at all.

Circuit designers can create functional units with a range of performance characteristics. Creating a unit with short latency or issue time requires more hardware, especially for more complex functions such as multiplication and floating-point operations. Since there is only a limited amount of space for these units on the microprocessor chip, the CPU designers must carefully balance the number of functional units and their individual performance to achieve optimal overall performance. They evaluate many different benchmark programs and dedicate the most resources to the most critical operations. As Figure 5.12 indicates, integer multiplication and floating-point multiplication and addition were considered important operations in design of the Pentium III, even though a significant amount of hardware is required to achieve the low latencies and high degree of pipelining shown. On the other hand, division is relatively infrequent, and difficult to implement with short latency or issue time, and so these operations are relatively slow.

5.7.3 A Closer Look at Processor Operation

As a tool for analyzing the performance of a machine level program executing on a modern processor, we have developed a more detailed textual notation to describe the operations generated by the instruction decoder, as well as a graphical notation to show the processing of operations by the functional units. Neither of these notations exactly represents the implementation of a specific, real-life processor. They are simply methods to help understand how a processor can take advantage of parallelism and branch prediction in executing a program.

Translating Instructions into Operations

We present our notation by working with `combine4` (Figure 5.10), our fastest code up to this point as an example. We focus just on the computation performed by the loop, since this is the dominating factor in performance for large vectors. We consider the cases of integer data with both multiplication and addition as the combining operations. The compiled code for this loop with multiplication consists of four instructions. In this code, register `%eax` holds the pointer `data`, `%edx` holds `i`, `%ecx` holds `x`, and `%esi` holds `length`.

```
     combine4: type=INT, OPER = *
     data in %eax, x in %ecx, i in %edx, length in %esi
1 .L24:                                  loop:
2   imull (%eax,%edx,4),%ecx               Multiply x by data[i]
3   incl %edx                              i++
4   cmpl %esi,%edx                         Compare i:length
5   jl .L24                                If <, goto loop
```

Every time the processor executes the loop, the instruction decoder translates these four instructions into a sequence of operations for the Execution Unit. On the first iteration, with `i` equal to 0, our hypothetical machine would issue the following sequence of operations:

Assembly Instructions	Execution Unit Operations
`.L24:`	
`imull (%eax,%edx,4),%ecx`	`load (%eax, %edx.0, 4) → t.1`
	`imull t.1, %ecx.0 → %ecx.1`
`incl %edx`	`incl %edx.0 → %edx.1`
`cmpl %esi,%edx`	`cmpl %esi, %edx.1 → cc.1`
`jl .L24`	`jl-taken cc.1`

In our translation, we have converted the memory reference by the multiply instruction into an explicit `load` instruction that reads the data from memory into the processor. We have also assigned *operand labels* to the values that change each iteration. These labels are a stylized version of the tags generated by register renaming. Thus, the value in register `%ecx` is identified by the label `%ecx.0` at the beginning of the loop, and by `%ecx.1` after it has been updated. The register values that do not change from one iteration to the next would be obtained directly from the register file during decoding. We also introduce the label `t.1` to denote the value read by the `load` operation and passed to the `imull` operation, and we explicitly show the destination of the operation. Thus, the pair of operations

```
load (%eax, %edx.0, 4) → t.1
imull t.1, %ecx.0      → %ecx.1
```

indicates that the processor first performs a `load` operation, computing the address using the value of `%eax` (which does not change during the loop), and the value stored in `%edx` at the start of the loop. This will yield a temporary value, which we label `t.1`. The multiply operation then takes this value and the value of `%ecx` at the start of the loop and produces a new value for `%ecx`. As this example illustrates, tags can be associated with intermediate values that are never written to the register file.

The operation

```
incl %edx.0 → %edx.1
```

indicates that the increment operation adds one to the value of `%edx` at the start of the loop to generate a new value for this register.

The operation

```
cmpl %esi, %edx.1 → cc.1
```

indicates that the compare operation (performed by either integer unit) compares the value in `%esi` (which does not change in the loop) with the newly computed value for `%edx`. It then sets the condition codes, identified with the explicit label `cc.1`. As this example illustrates, the processor can use renaming to track changes to the condition code registers.

Finally, the jump instruction was predicted taken. The jump operation

```
jl-taken cc.1
```

Execution Unit Operations		
`load (%eax, %edx.0, 4)`	→	`t.1`
`imull t.1, %ecx.0`	→	`%ecx.1`
`incl %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

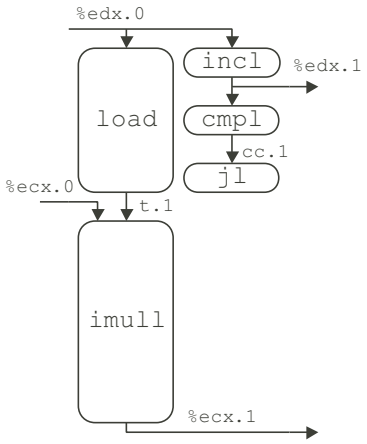


Figure 5.13: **Operations for First Iteration of Inner Loop of `combine4` for integer multiplication.** Memory reads are explicitly converted to loads. Register names are tagged with instance numbers.

checks whether the newly computed values for the condition codes (`cc.1`) indicate this was the correct choice. If not, then it signals the ICU to begin fetching instructions at the instruction following the `jl`. To simplify the notation, we omit any information about the possible jump destinations. In practice, the processor must keep track of the destination for the unpredicted direction, so that it can begin fetching from there in the event the prediction is incorrect.

As this example translation shows, our operations mimic the structure of the assembly-language instructions in many ways, except that they refer to their source and destination operations by labels that identify different instances of the registers. In the actual hardware, register renaming dynamically assigns tags to indicate these different values. Tags are bit patterns rather than symbolic names such as "`%edx.1`," but they serve the same purpose.

Processing of Operations by the Execution Unit

Figure 5.13 shows the operations in two forms: that generated by the instruction decoder and as a *computation graph* where operations are represented by rounded boxes and arrows indicate the passing of data between operations. We only show the arrows for the operands that change from one iteration to the next, since only these values are passed directly between functional units.

The height of each operator box indicates how many cycles the operation requires, that is, the latency of that particular function. In this case, integer multiplication `imull` requires four cycles, load requires three, and the other operations require one. In demonstrating the timing of a loop, we position the blocks vertically to represent the times when operations are performed, with time increasing in the downward direction. We can see that the five operations for the loop form two parallel chains, indicating two series of computations that must be performed in sequence. The chain on the left processes the data, first reading an array element from memory and then multiplying it times the accumulated product. The chain on the right processes the loop index `i`, first incrementing it and then comparing it to `length`. The jump operation checks the result of this comparison to make sure the branch was correctly predicted. Note that there are no outgoing arrows

Execution Unit Operations		
`load (%eax, %edx.0, 4)`	→	`t.1`
`addl t.1, %ecx.0`	→	`%ecx.1`
`incl %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

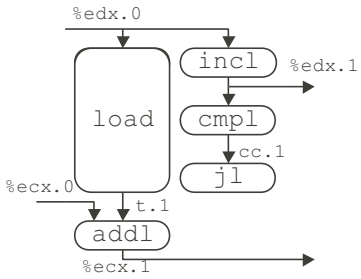


Figure 5.14: **Operations for First Iteration of Inner Loop of `combine4` for Integer Addition.** Compared to multiplication, the only change is that the addition operation requires only one cycle.

from the jump operation box. If the branch was correctly predicted, no other processing is required. If the branch was incorrectly predicted, then the branch function unit will signal the instruction fetch control unit, and this unit will take corrective action. In either case, the other operations do not depend on the outcome of the jump operation.

Figure 5.14 shows the same translation into operations but with integer addition as the combining operation. As the graphical depiction shows, all of the operations, except load, now require just one cycle.

Scheduling of Operations with Unlimited Resources

To see how a processor would execute a series of iterations, imagine first a processor with an unlimited number of functional units and with perfect branch prediction. Each operation could then begin as soon as its data operands were available. The performance of such a processor would be limited only by the latencies and throughputs of the functional units, and the data dependencies in the program. Figure 5.15 shows the computation graph for the first three iterations of the loop in `combine4` with integer multiplication on such a machine. For each iteration, there is a set of five operations with the same configuration as those in Figure 5.13, with appropriate changes to the operand labels. The arrows from the operators of one iteration to those of another show the data dependencies between the different iterations.

Each operator is placed vertically at the highest position possible, subject to the constraint that no arrows can point upward, since this would indicate information flowing backward in time. Thus, the `load` operation of one iteration can begin as soon as the `incl` operation of the previous iteration has generated an updated value of the loop index.

The computation graph shows the parallel execution of operations by the Execution Unit. On each cycle, all of the operations on one horizontal line of the graph execute in parallel. The graph also demonstrates out-of-order, speculative execution. For example, the `incl` operation in one iteration is executed before the `jl` instruction of the previous iteration has even begun. We can also see the effect of pipelining. Each iteration requires at least seven cycles from start to end, but successive iterations are completed every 4 cycles. Thus, the effective processing rate is one iteration every 4 cycles, giving a CPE of 4.0.

The four-cycle latency of integer multiplication constrains the performance of the processor for this program. Each `imull` operation must wait until the previous one has completed, since it needs the result of

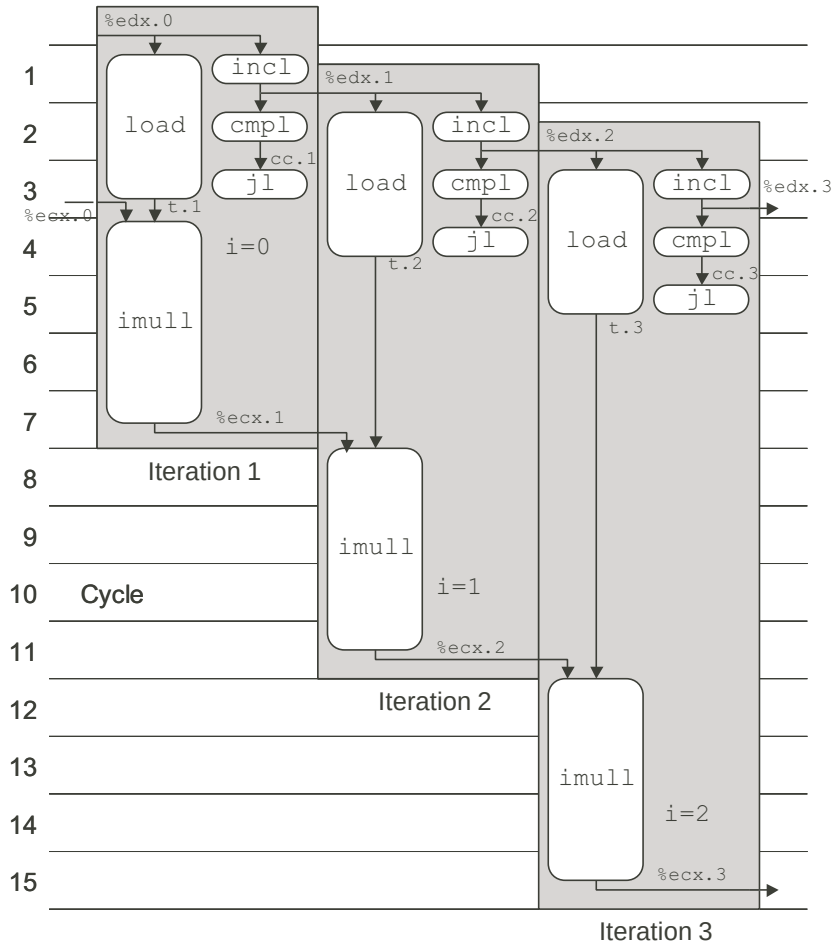


Figure 5.15: **Scheduling of Operations for Integer Multiplication with Unlimited Number of Execution Units.** The 4 cycle latency of the multiplier is the performance-limiting resource.

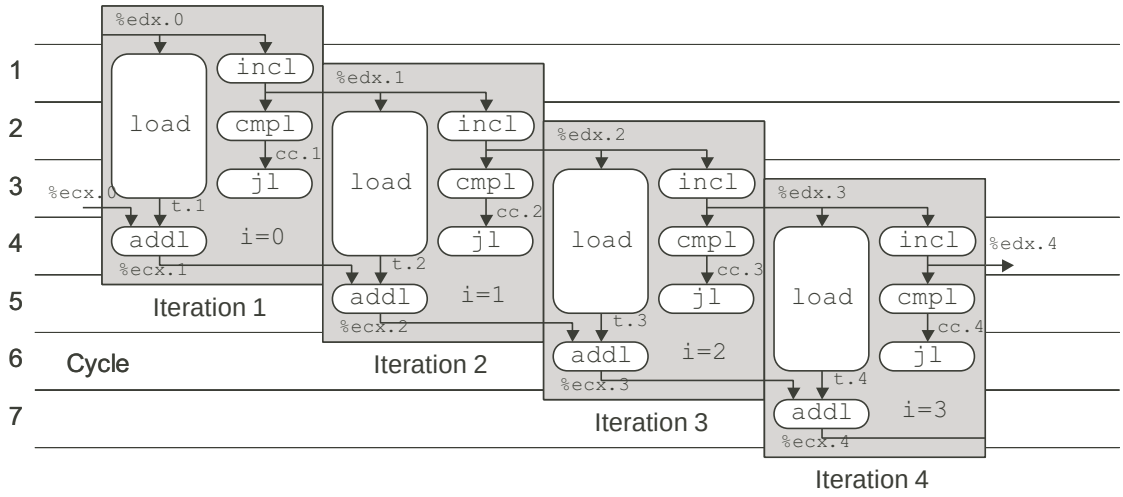


Figure 5.16: **Scheduling of Operations for Integer Addition with Unbounded Resource Constraints.** With unbounded resources the processor could achieve a CPE of 1.0.

this multiplication before it can begin. In our figure, the multiplication operations begin on cycles 4, 8, and 12. With each succeeding iteration, a new multiplication begins every fourth cycle.

Figure 5.16 shows the first four iterations of `combine4` for integer addition on a machine with an unbounded number of functional units. With a single-cycle combining operation, the program could achieve a CPE of 1.0. We see that as the iterations progress, the Execution Unit would perform parts of seven operations on each clock cycle. For example, in cycle 4 we can see that the machine is executing the `addl` for iteration 1; different parts of the `load` operations for iterations 2, 3, and 4; the `jl` for iteration 2; the `cmpl` for iteration 3; and the `incl` for iteration 4.

Scheduling of Operations with Resource Constraints

Of course, a real processor has only a fixed set of functional units. Unlike our earlier examples, where the performance was constrained only by the data dependencies and the latencies of the functional units, performance becomes limited by resource constraints as well. In particular, our processor has only two units capable of performing integer and branch operations. In contrast, the graph of Figure 5.15 has three of these operations in parallel on cycles 3 and four in parallel on cycle 4.

Figure 5.17 shows the scheduling of the operations for `combine4` with integer multiplication on a resource-constrained processor. We assume that the general integer unit and the branch/integer unit can each begin a new operation on every clock cycle. It is possible to have more than two integer or branch operations executing in parallel, as shown in cycle 6, because the `imull` operation is in its third cycle by this point.

With constrained resources, our processor must have some *scheduling policy* that determines which operation to perform when it has more than one choice. For example, in cycle 3 of the graph of Figure 5.15, we show three integer operations being executed: the `jl` of iteration 1, the `cmpl` of iteration 2, and the `incl` of iteration 3. For Figure 5.17, we must delay one of these operations. We do so by keeping track of

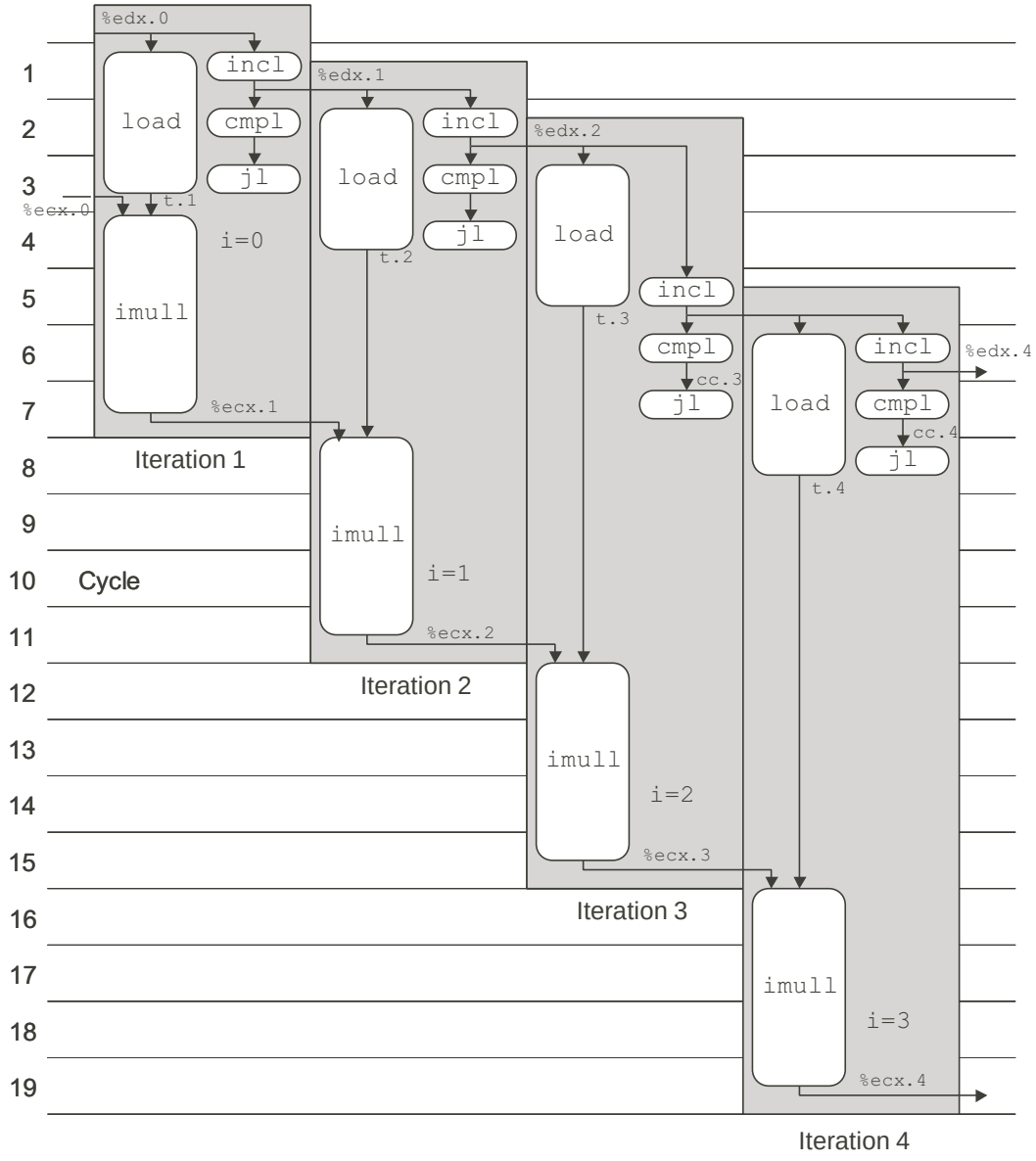


Figure 5.17: **Scheduling of Operations for Integer Multiplication with Actual Resource Constraints.** The multiplier latency remains the performance-limiting factor.

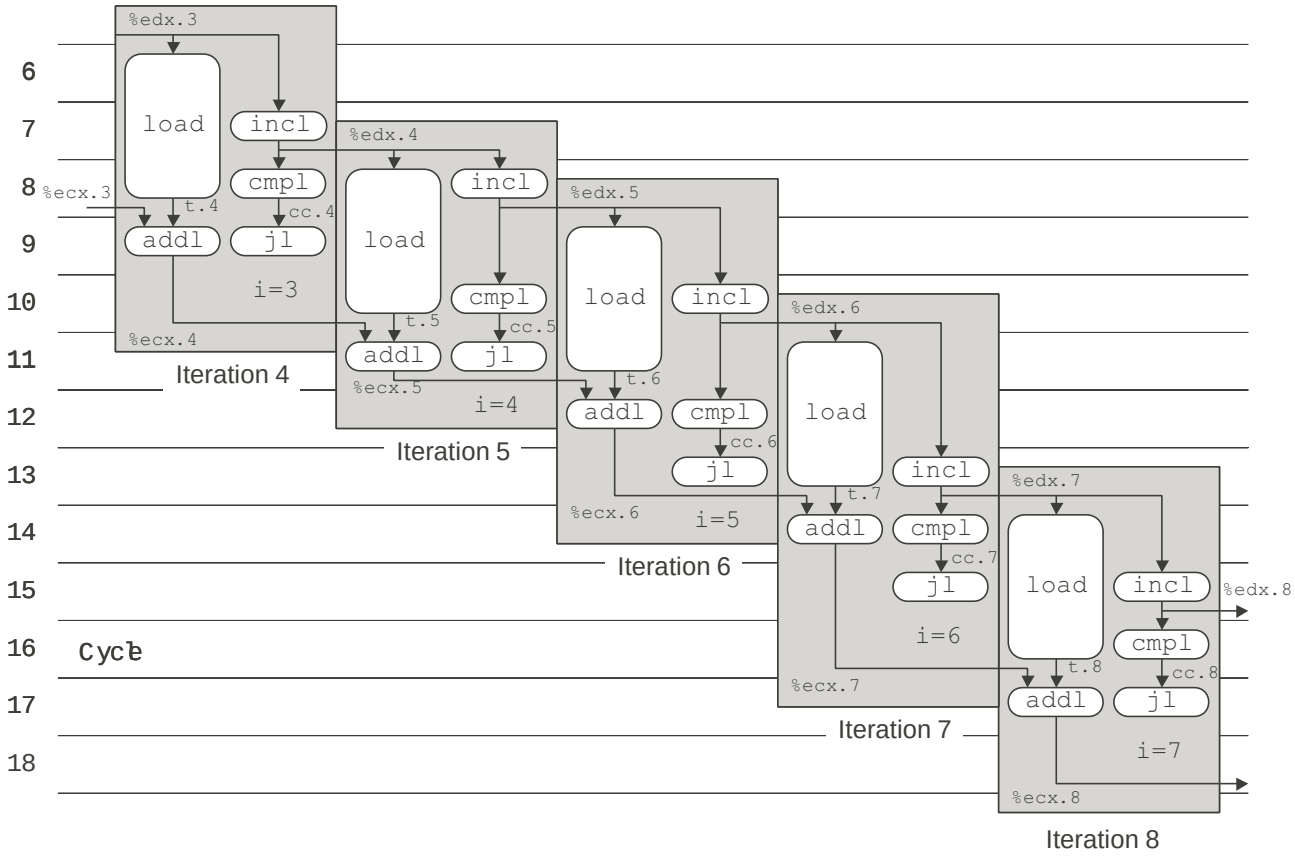


Figure 5.18: **Scheduling of Operations for Integer Addition with Actual Resource Constraints.** The limitation to two integer units constrains performance to a CPE of 2.0.

the *program order* for the operations, that is, the order in which the operations would be performed if we executed the machine-level program in strict sequence. We then give priority to the operations according to their program order. In this example, we would defer the `incl` operation, since any operation of iteration 3 is later in program order than those of iterations 1 and 2. Similarly, in cycle 4, we would give priority to the `imull` operation of iteration 1 and the `jl` of iteration 2 over that of the `incl` operation of iteration 3.

For this example, the limited number of functional units does not slow down our program. Performance is still constrained by the four-cycle latency of integer multiplication.

For the case of integer addition, the resource constraints impose a clear limitation on program performance. Each iteration requires four integer or branch operations, and there are only two functional units for these operations. Thus, we cannot hope to sustain a processing rate any better than two cycles per iteration. In creating the graph for multiple iterations of `combine4` for integer addition, an interesting pattern emerges. Figure 5.18 shows the scheduling of operations for iterations 4 through 8. We chose this range of iterations because it shows a regular pattern of operation timings. Observe how the timing of all operations in iterations 4 and 8 is identical, except that the operations in iteration 8 occur eight cycles later. As the iterations proceed, the patterns shown for iterations 4 to 7 would keep repeating. Thus, we complete four iterations every eight

cycles, achieving the optimum CPE of 2.0.

Summary of `combine4` Performance

We can now consider the measured performance of `combine4` for all four combinations of data type and combining operations:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine4`	219	Accumulate in temporary	2.00	4.00	3.00	5.00

With the exception of integer addition, these cycle times nearly match the latency for the combining operation, as shown in Figure 5.12. Our transformations to this point have reduced the CPE value to the point where the time for the combining operation becomes the limiting factor.

For the case of integer addition, we have seen that the limited number of functional units for branch and integer operations limits the achievable performance. With four such operations per iteration, and just two functional units, we cannot expect the program to go faster than 2 cycles per iteration.

In general, processor performance is limited by three types of constraints. First, the data dependencies in the program force some operations to delay until their operands have been computed. Since the functional units have latencies of one or more cycles, this places a lower bound on the number of cycles in which a given sequence of operations can be performed. Second, the resource constraints limit how many operations can be performed at any given time. We have seen that the limited number of functional units is one such resource constraint. Other constraints include the degree of pipelining by the functional units, as well as limitations of other resources in the ICU and the EU. For example, an Intel Pentium III can only decode three instructions on every clock cycle. Finally, the success of the branch prediction logic constrains the degree to which the processor can work far enough ahead in the instruction stream to keep the execution unit busy. Whenever a misprediction occurs, a significant delay occurs getting the processor restarted at the correct location.

5.8 Reducing Loop Overhead

The performance of `combine4` for integer addition is limited by the fact that each iteration contains four instructions, with only two functional units capable of performing them. Only one of these four instructions operates on the program data. The others are part of the loop overhead of computing the loop index and testing the loop condition.

We can reduce overhead effects by performing more data operations in each iteration, via a technique known as *loop unrolling*. The idea is to access and combine multiple array elements within a single iteration. The resulting program requires fewer iterations, leading to reduced loop overhead.

Figure 5.19 shows a version of our combining code using three-way loop unrolling. The first loop steps through the array three elements at a time. That is, the loop index `i` is incremented by three on each iteration, and the combining operation is applied to array elements i, $i + 1$, and $i + 2$ in a single iteration.

code/opt/combine.c

```
1 /* Unroll loop by 3 */
2 void combine5(vec_ptr v, data_t *dest)
3 {
4     int length = vec_length(v);
5     int limit = length-2;
6     data_t *data = get_vec_start(v);
7     data_t x = IDENT;
8     int i;
9
10    /* Combine 3 elements at a time */
11    for (i = 0; i < limit; i+=3) {
12        x = x OPER data[i] OPER data[i+1] OPER data[i+2];
13    }
14
15    /* Finish any remaining elements */
16    for (; i < length; i++) {
17        x = x OPER data[i];
18    }
19    *dest = x;
20 }
```

code/opt/combine.c

Figure 5.19: **Unrolling Loop by 3.** Loop unrolling can reduce the effect of loop overhead.

Execution Unit Operations		
load (%eax, %edx.0, 4)	→	t.1a
addl t.1a, %ecx.0c	→	%ecx.1a
load 4(%eax, %edx.0, 4)	→	t.1b
addl t.1b, %ecx.1a	→	%ecx.1b
load 8(%eax, %edx.0, 4)	→	t.1c
addl t.1c, %ecx.1b	→	%ecx.1c
addl %edx.0, 3	→	%edx.1
cmpl %esi, %edx.1	→	cc.1
jl-taken cc.1		

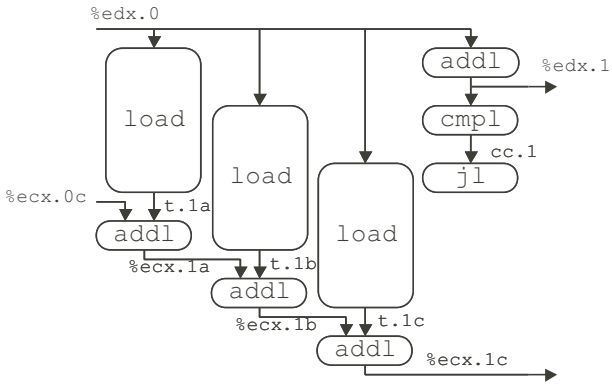


Figure 5.20: **Operations for First Iteration of Inner Loop of Three-Way Unrolled Integer Addition.** With this degree of loop unrolling we can combine three array elements using six integer/branch operations.

In general, the vector length will not be a multiple of 3. We want our code to work correctly for arbitrary vector lengths. We account for this requirement in two ways. First, we make sure the first loop does not overrun the array bounds. For a vector of length n, we set the loop limit to be $n-2$. We are then assured that the loop will only be executed when the loop index i satisfies $i < n-2$, and hence the maximum array index $i+2$ will satisfy $i+2 < (n-2)+2 = n$. In general, if the loop is unrolled by k, we set the upper limit to be $n-k+1$. The maximum loop index $i+k-1$ will then be less than n. In addition to this, we add a second loop to step through the final few elements of the vector one at a time. The body of this loop will be executed between 0 and 2 times.

To better understand the performance of code with loop unrolling, let us look at the assembly code for the inner loop and its translation into operations.

Assembly Instructions	Execution Unit Operations		
.L49:			
addl (%eax,%edx,4),%ecx	load (%eax, %edx.0, 4)	→	t.1a
	addl t.1a, %ecx.0c	→	%ecx.1a
addl 4(%eax,%edx,4),%ecx	load 4(%eax, %edx.0, 4)	→	t.1b
	addl t.1b, %ecx.1a	→	%ecx.1b
addl 8(%eax,%edx,4),%ecx	load 8(%eax, %edx.0, 4)	→	t.1c
	addl t.1c, %ecx.1b	→	%ecx.1c
addl %edx,3	addl %edx.0, 3	→	%edx.1
cmpl %esi,%edx	cmpl %esi, %edx.1	→	cc.1
jl .L49	jl-taken cc.1		

As mentioned earlier, loop unrolling by itself will only help the performance of the code for the case of integer sum, since our other cases are limited by the latency of the functional units. For integer sum, three-way unrolling allows us to combine three elements with six integer/branch operations, as shown in Figure 5.20. With two functional units for these operations, we could potentially achieve a CPE of 1.0. Figure 5.21

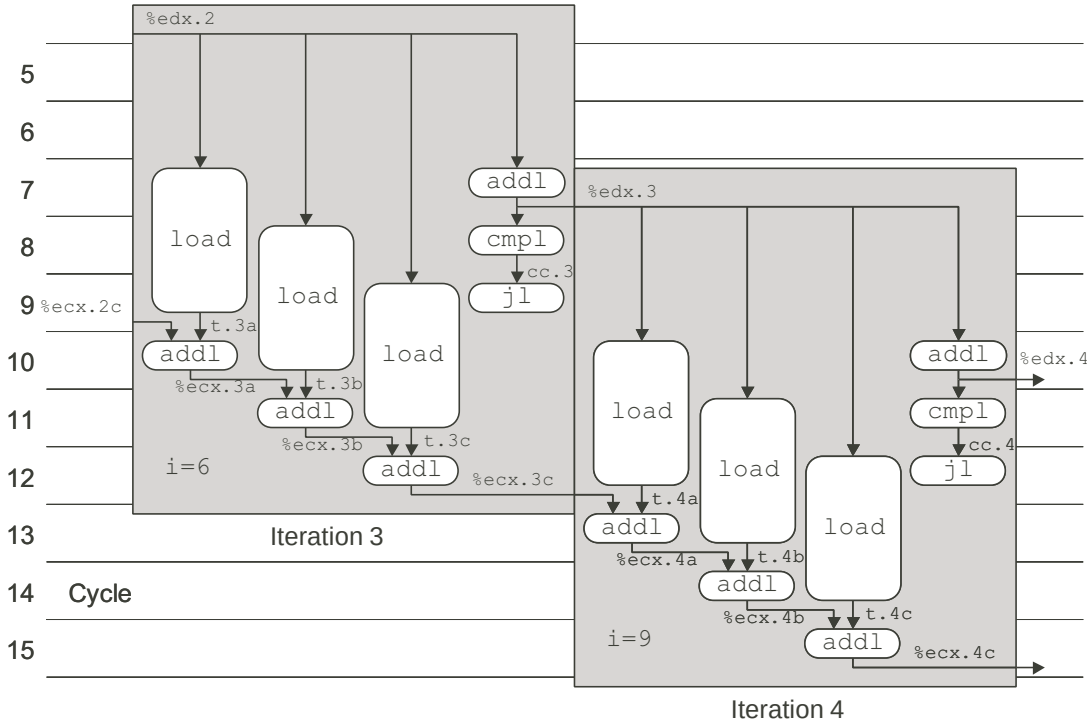


Figure 5.21: **Scheduling of Operations for Three-Way Unrolled Integer Sum with Bounded Resource Constraints.** In principle, the procedure can achieve a CPE of 1.0. The measured CPE, however, is 1.33.

shows that once we reach iteration 3 (`i` $= 6$), the operations would follow a regular pattern. The operations of iteration 4 (`i` $= 9$) have the same timings, but shifted by three cycles. This would indeed yield a CPE of 1.0.

Our measurement for this function shows a CPE of 1.33, that is, we require four cycles per iteration. Evidently some resource constraint we did not account for in our analysis delays the computation by one additional cycle per iteration. Nonetheless, this performance represents an improvement over the code without loop unrolling.

Measuring the performance for different degrees of unrolling yields the following values for the CPE

Vector Length	Degree of Unrolling					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06

As these measurements show, loop unrolling can reduce the CPE. With the loop unrolled by a factor of two, each iteration of the main loop requires three clock cycles, giving a CPE of $3/2 = 1.5$. As we increase the degree of unrolling, we generally get better performance, nearing the theoretical CPE limit of 1.0. It is interesting to note that the improvement is not monotonic—unrolling by three gives better performance than unrolling by four. Evidently the scheduling of operations on the execution units is less efficient for the latter case.

Our CPE measurements do not account for overhead factors such as the cost of the procedure call and of setting up the loop. With loop unrolling, we introduce a new source of overhead—the need to finish any remaining elements when the vector length is not divisible by the degree of unrolling. To investigate the impact of overhead, we measure the *net CPE* for different vector lengths. The net CPE is computed as the total number of cycles required by the procedure divided by the number of elements. For the different degrees of unrolling, and for two different vector lengths we obtain the following:

Vector Length	Degree of Unrolling					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06
31 Net CPE	4.02	3.57	3.39	3.84	3.91	3.66
1024 Net CPE	2.06	1.56	1.40	1.56	1.31	1.12

The distinction between CPE and net CPE is minimal for long vectors, as seen with the measurements for length 1024, but the impact is significant for short vectors, as seen with the measurements for length 31. Our measurements of the net CPE for a vector of length 31 demonstrate one drawback of loop unrolling. Even with no unrolling, the net CPE of 4.02 is considerably higher than the 2.06 measured for long vectors. The overhead of starting and completing the loop becomes far more significant when the loop is executed a smaller number of times. In addition, the benefit of loop unrolling is less significant. Our unrolled code must start and stop two loops, and it must complete the final elements one at a time. The overhead decreases with increased loop unrolling, while the number of operations performed in the final loop increases. With a vector length of 1024, performance generally improves as the degree of unrolling increases. With a vector length of 31, the best performance is achieved by unrolling the loop by only a factor of three.

A second drawback of loop unrolling is that it increases the amount of object code generated. The object code for `combine4` requires 63 bytes, whereas the object code with the loop unrolled by a factor of 16

requires 142 bytes. In this case, that seems like a small price to pay for code that runs nearly twice as fast. In other cases, however, the optimum position in this time-space tradeoff is not so clear.

5.9 Converting to Pointer Code

Before proceeding further, let us attempt one more transformation that can sometimes improve program performance, at the expense of program readability. One of the unique features of C is the ability to create and reference pointers to arbitrary program objects. Pointer arithmetic, in fact, has a close connection to array referencing. The combination of pointer arithmetic and referencing given by the expression `*(a+i)` is exactly equivalent to the array reference `a[i]`. At times, we can improve the performance of a program by using pointers rather than arrays.

Figure 5.22 shows an example of converting the procedures `combine4` and `combine5` to pointer code, giving procedures `combine4p` and `combine5p`, respectively. Instead of keeping pointer `data` fixed at the beginning of the vector, we move it with each iteration. The vector elements are then referenced by a fixed offset (between 0 and 2) of `data`. Most significantly, we can eliminate the iteration variable `i` from the procedure. To detect when the loop should terminate, we compute a pointer `dend` to be an upper bound on pointer `data`.

Comparing the performance of these procedures to their array counterparts yields mixed results:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine4`	219	Accumulate in temporary	2.00	4.00	3.00	5.00
`combine4p`	239	Pointer version	**3.00**	4.00	3.00	5.00
`combine5`	234	Unroll loop $\times 3$	1.33	4.00	3.00	5.00
`combine5p`	239	Pointer version	1.33	4.00	3.00	5.00
`combine5x4`		Unroll loop $\times 4$	1.50	4.00	3.00	5.00
`combine5px4`		Pointer version	**1.25**	4.00	3.00	5.00

For most of the cases, the array and pointer versions have the exact same performance. With pointer code, the CPE for integer sum with no unrolling actually gets worse by one cycle. This result is somewhat surprising, since the inner loops for the pointer and array versions are very similar, as shown in Figure 5.23. It is hard to imagine why the pointer code requires an additional clock cycle per iteration. Just as mysteriously, versions of the procedures with four-way loop unrolling yield a one-cycle-per-iteration improvement with pointer code, giving a CPE of 1.25 (five cycles per iteration) rather then 1.5 (six cycles per iteration).

In our experience, the relative performance of pointer versus array code depends on the machine, the compiler, and even the particular procedure. We have seen compilers that apply very advanced optimizations to array code but only minimal optimizations to pointer code. For the sake of readability, array code is generally preferable.

Practice Problem 5.3:

At times, GCC does its own version of converting array code to pointer code. For example, with integer data and addition as the combining operation, it generates the following code for the inner loop of a variant of `combine5` that uses eight-way loop unrolling:

code/opt/combine.c

```
1 /* Accumulate in local variable, pointer version */
2 void combine4p(vec_ptr v, data_t *dest)
3 {
4     int length = vec_length(v);
5     data_t *data = get_vec_start(v);
6     data_t *dend = data+length;
7     data_t x = IDENT;
8
9     for (; data < dend; data++)
10        x = x OPER *data;
11    *dest = x;
12 }
```

code/opt/combine.c

(a) Pointer version of `combine4`.

code/opt/combine.c

```
1 /* Unroll loop by 3, pointer version */
2 void combine5p(vec_ptr v, data_t *dest)
3 {
4     data_t *data = get_vec_start(v);
5     data_t *dend = data+vec_length(v);
6     data_t *dlimit = dend-2;
7     data_t x = IDENT;
8
9     /* Combine 3 elements at a time */
10    for (; data < dlimit; data += 3) {
11        x = x OPER data[0] OPER data[1] OPER data[2];
12    }
13
14    /* Finish any remaining elements */
15    for (; data < dend; data++) {
16        x = x OPER data[0];
17    }
18    *dest = x;
19 }
```

code/opt/combine.c

(b) Pointer version of `combine5`

Figure 5.22: **Converting Array Code to Pointer Code.** In some cases, this can lead to improved performance.

```
    combine4: type=INT, OPER = '+'
    data in %eax, x in %ecx, i in %edx, length in %esi
1 .L24:                          loop:
2   addl (%eax,%edx,4),%ecx      Add data[i] to x
3   incl %edx                    i++
4   cmpl %esi,%edx               Compare i:length
5   jl .L24                      If <, goto loop
```

(a) Array code

```
    combine4p: type=INT, OPER = '+'
    data in %eax, x in %ecx, dend in %edx
1 .L30:                          loop:
2   addl (%eax),%ecx             Add data[0] to x
3   addl $4,%eax                 data++
4   cmpl %edx,%eax               Compare data:dend
5   jb .L30                      If <, goto loop
```

(b) Pointer code

Figure 5.23: **Pointer Code Performance Anomaly.** Although the two programs are very similar in structure, the array code requires two cycles per iteration, while the pointer code requires three.

```
 1 .L6:
 2   addl (%eax),%edx
 3   addl 4(%eax),%edx
 4   addl 8(%eax),%edx
 5   addl 12(%eax),%edx
 6   addl 16(%eax),%edx
 7   addl 20(%eax),%edx
 8   addl 24(%eax),%edx
 9   addl 28(%eax),%edx
10   addl $32,%eax
11   addl $8,%ecx
12   cmpl %esi,%ecx
13   jl .L6
```

Observe how register `%eax` is being incremented by 32 on each iteration.

Write C code for a procedure `combine5px8` that shows how pointers, loop variables, and termination conditions are being computed by this code. Show the general form with arbitrary data and combining operation in the style of Figure 5.19. Describe how it differs from our handwritten pointer code (Figure 5.22).

code/opt/combine.c

```
1 /* Unroll loop by 2, 2-way parallelism */
2 void combine6(vec_ptr v, data_t *dest)
3 {
4     int length = vec_length(v);
5     int limit = length-1;
6     data_t *data = get_vec_start(v);
7     data_t x0 = IDENT;
8     data_t x1 = IDENT;
9     int i;
10
11    /* Combine 2 elements at a time */
12    for (i = 0; i < limit; i+=2) {
13        x0 = x0 OPER data[i];
14        x1 = x1 OPER data[i+1];
15    }
16
17    /* Finish any remaining elements */
18    for (; i < length; i++) {
19        x0 = x0 OPER data[i];
20    }
21    *dest = x0 OPER x1;
22 }
```

code/opt/combine.c

Figure 5.24: **Unrolling Loop by 2 and Using Two-Way Parallelism.** This approach makes use of the pipelining capability of the functional units.

5.10 Enhancing Parallelism

At this point, our programs are limited by the latency of the functional units. As the third column in Figure 5.12 shows, however, several functional units of the processor are *pipelined*, meaning that they can start on a new operation before the previous one is completed. Our code cannot take advantage of this capability, even with loop unrolling, since we are accumulating the value as a single variable `x`. We cannot compute a new value of `x` until the preceding computation has completed. As a result, the processor will *stall*, waiting to begin a new operation until the current one has completed. This limitation shows clearly in Figures 5.15 and 5.17. Even with unbounded processor resources, the multiplier can only produce a new result every four clock cycles. Similar limitations occur with floating-point addition (three cycles) and multiplication (five cycles).

5.10.1 Loop Splitting

For a combining operation that is associative and commutative, such as integer addition or multiplication, we can improve performance by splitting the set of combining operations into two or more parts and combining

Execution Unit Operations		
`load (%eax, %edx.0, 4)`	→	`t.1a`
`imull t.1a, %ecx.0`	→	`%ecx.1`
`load 4(%eax, %edx.0, 4)`	→	`t.1b`
`imull t.1b, %ebx.0`	→	`%ebx.1`
`addl $2, %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

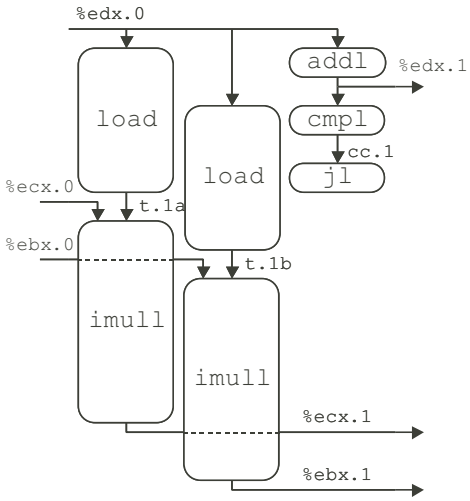


Figure 5.25: **Operations for First Iteration of Inner Loop of Two-Way Unrolled, Two-Way Parallel Integer Multiplication.** The two multiplication operations are logically independent.

the results at the end. For example, let P_n denote the product of elements $a_0, a_1, \ldots, a_{n-1}$:

$$P_n = \prod_{i=0}^{n-1} a_i$$

Assuming n is even, we can also write this as $P_n = PE_n \times PO_n$, where PE_n is the product of the elements with even indices, and PO_n is the product of the elements with odd indices:

$$PE_n = \prod_{i=0}^{n/2-2} a_{2i}$$

$$PO_n = \prod_{i=0}^{n/2-2} a_{2i+1}$$

Figure 5.24 shows code that uses this method. It uses both two-way loop unrolling to combine more elements per iteration, and two-way parallelism, accumulating elements with even index in variable `x0`, and elements with odd index in variable `x1`. As before, we include a second loop to accumulate any remaining array elements for the case where the vector length is not a multiple of 2. We then apply the combining operation to `x0` and `x1` to compute the final result.

To see how this code yields improved performance, let us consider the translation of the loop into operations for the case of integer multiplication:

Assembly Instructions	Execution Unit Operations
`.L151:`	
`imull (%eax,%edx,4),%ecx`	`load (%eax, %edx.0, 4)` → `t.1a`
	`imull t.1a, %ecx.0` → `%ecx.1`
`imull 4(%eax,%edx,4),%ebx`	`load 4(%eax, %edx.0, 4)` → `t.1b`
	`imull t.1b, %ebx.0` → `%ebx.1`
`addl $2,%edx`	`addl $2, %edx.0` → `%edx.1`
`cmpl %esi,%edx`	`cmpl %esi, %edx.1` → `cc.1`
`jl .L151`	`jl-taken cc.1`

Figure 5.25 shows a graphical representation of these operations for the first iteration (`i` $= 0$). As this diagram illustrates, the two multiplications in the loop are independent of each other. One has register `%ecx` as its source and destination (corresponding to program variable `x0`), while the other has register `%ebx` as its source and destination (corresponding to program variable `x1`). The second multiplication can start just one cycle after the first. This makes use of the pipelining capabilities of both the load unit and the integer multiplier.

Figure 5.26 shows a graphical representation of the first three iterations (`i` $= 0, 2,$ and 4) for integer multiplication. For each iteration, the two multiplications must wait until the results from the previous iteration have been computed. Still, the machine can generate two results every four clock cycles, giving a theoretical CPE of 2.0. In this figure we do not take into account the limited set of integer functional units, but this does not prove to be a limitation for this particular procedure.

Comparing loop unrolling alone to loop unrolling with two-way parallelism, we obtain the following performance:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
		Unroll $\times 2$	1.50	4.00	3.00	5.00
`combine6`	241	Unroll $\times 2$, Parallelism $\times 2$	1.50	2.00	2.00	2.50

For integer sum, parallelism does not help, as the latency of integer addition is only one clock cycle. For integer and floating-point product, however, we reduce the CPE by a factor of two. We are essentially doubling the use of the functional units. For floating-point sum, some other resource constraint is limiting our CPE to 2.0, rather than the theoretical value of 1.5.

We have seen earlier that two's complement arithmetic is commutative and associative, even when overflow occurs. Hence for an integer data type, the result computed by `combine6` will be identical to that computed by `combine5` under all possible conditions. Thus, an optimizing compiler could potentially convert the code shown in `combine4` first to a two-way unrolled variant of `combine5` by loop unrolling, and then to that of `combine6` by introducing parallelism. This is referred to as *iteration splitting* in the optimizing compiler literature. Many compilers do loop unrolling automatically, but relatively few do iteration splitting.

On the other hand, we have seen that floating-point multiplication and addition are not associative. Thus, `combine5` and `combine6` could potentially produce different results due to rounding or overflow. Imagine, for example, a case where all the elements with even indices were numbers with very large absolute value, while those with odd indices were very close to 0.0. Then product PE_n might overflow, or PO_n might underflow, even though the final product P_n does not. In most real-life applications, however, such

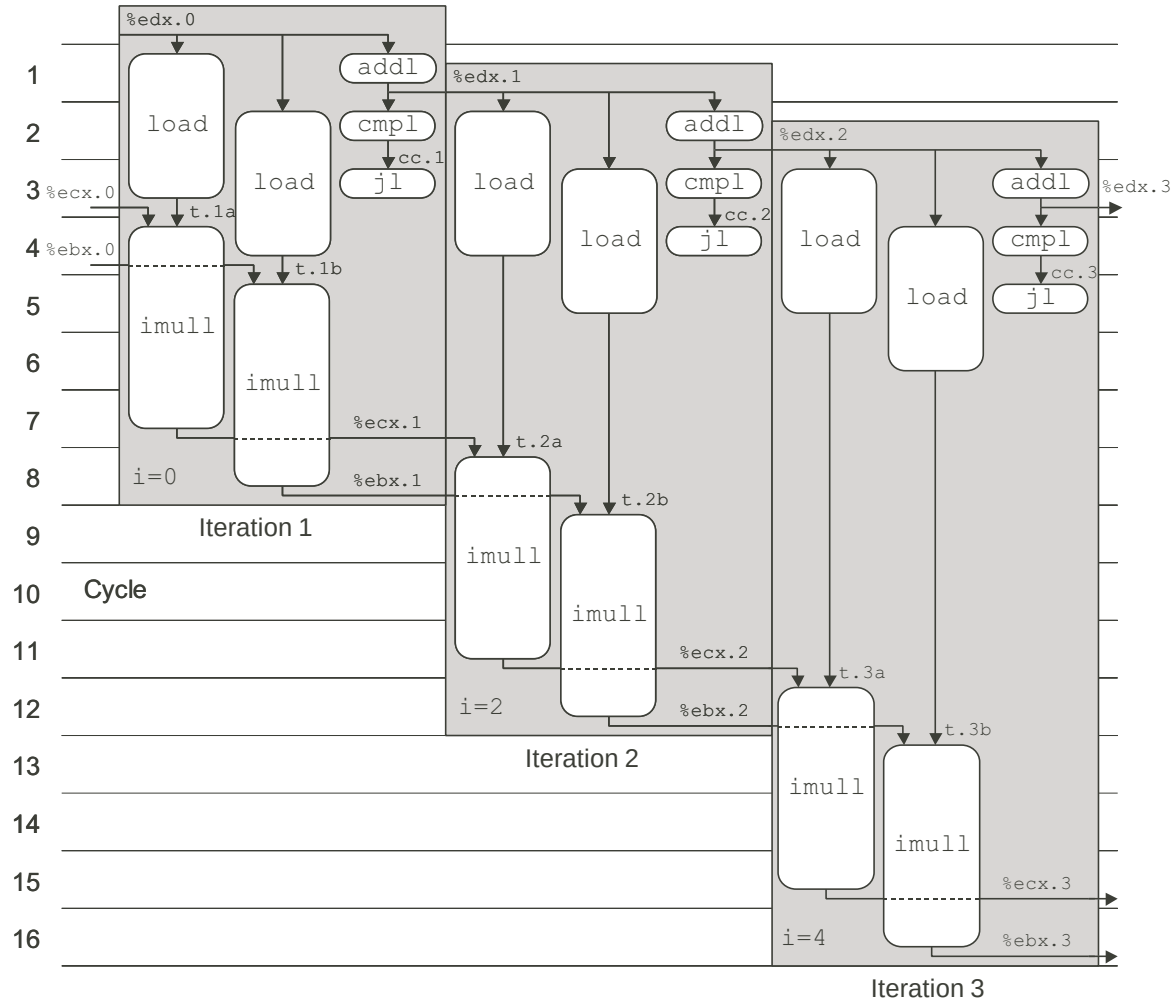


Figure 5.26: **Scheduling of Operations for Two-Way Unrolled, Two-Way Parallel Integer Multiplication with Unlimited Resources.** The multiplier can now generate two values every 4 cycles.

patterns are unlikely. Since most physical phenomena are continous, numerical data tend to be reasonably smooth and well-behaved. Even when there are discontinuities, they do not generally cause periodic patterns that lead to a condition such as that sketched above. It is unlikely that summing the elements in strict order gives fundamentally better accuracy than does summing two groups independently and then adding those sums together. For most applications, achieving a performance gain of 2X outweighs the risk of generating different results for strange data patterns. Nevertheless, a program developer should check with potential users to see if there are particular conditions that may cause the revised algorithm to be unacceptable.

Just as we can unroll loops by an arbitrary factor k, we can also increase the parallelism to any factor p such that k is divisible by p. The following are some results for different degrees of unrolling and parallelism:

Method	Integer		Floating Point	
	+	*	+	*
Unroll $\times 2$	1.50	4.00	3.00	5.00
Unroll $\times 2$, Parallelism $\times 2$	1.50	2.00	2.00	2.50
Unroll $\times 4$	1.50	4.00	3.00	5.00
Unroll $\times 4$, Parallelism $\times 2$	1.50	2.00	1.50	2.50
Unroll $\times 8$	1.25	4.00	3.00	5.00
Unroll $\times 8$, Parallelism $\times 2$	1.25	2.00	1.50	2.50
Unroll $\times 8$, Parallelism $\times 4$	1.25	1.25	1.61	2.00
Unroll $\times 8$, Parallelism $\times 8$	1.75	1.87	1.87	2.07
Unroll $\times 9$, Parallelism $\times 3$	1.22	1.33	1.66	2.00

As this table shows, increasing the degree of loop unrolling and the degree of parallelism helps program performance up to some point, but it yields diminishing improvement or even worse performance when taken to an extreme. In the next section, we will describe two reasons for this phenomenon.

5.10.2 Register Spilling

The benefits of loop parallelism are limited by the ability to express the computation in assembly code. In particular, the IA32 instruction set only has a small number of registers to hold the values being accumulated. If we have a degree of parallelism p that exceeds the number of available registers, then the compiler will resort to *spilling*, storing some of the temporary values on the stack. Once this happens, the performance drops dramatically. This occurs for our benchmarks when we attempt to have $p = 8$. Our measurements show the performance for this case is worse than that for $p = 4$.

For the case of the integer data type, there are only eight total integer registers available. Two of these (`%ebp` and `%esp`) point to regions of the stack. With the pointer version of the code, one of the remaining six holds the pointer `data`, and one holds the stopping position `dend`. This leaves only four integer registers for accumulating values. With the array version of the code, we require three registers to hold the loop index `i`, the stopping index `limit`, and the array address `data`. This leaves only three registers for accumulating values. For the floating-point data type, we need two of eight registers to hold intermediate values, leaving six for accumulating values. Thus, we could have a maximum parallelism of six before register spilling occurs.

This limitation to eight integer and eight floating-point registers is an unfortunate artifact of the IA32 instruc-

tion set. The renaming scheme described previously eliminates the direct correspondence between register names and the actual location of the register data. In a modern processor, register names serve simply to identify the program values being passed between the functional units. IA32 provides only a small number of such identifiers, constraining the amount of parallelism that can be expressed in programs.

The occurrence of spilling can be seen by examining the assembly code. For example, within the first loop for the code with eight-way parallelism we see the following instruction sequence:

```
     type=INT, OPER = '*'
     x6 in -12(%ebp), data+i in %eax
1    movl -12(%ebp),%edi       Get x6 from stack
2    imull 24(%eax),%edi       Multiply by data[i+6]
3    movl %edi,-12(%ebp)       Put x6 back
```

In this code, a stack location is being used to hold `x6`, one of the eight local variables used to accumulate sums. The code loads it into a register, multiplies it by one of the data elements, and stores it back to the same stack location. As a general rule, any time a compiled program shows evidence of register spilling within some heavily used inner loop, it might be preferable to rewrite the code so that fewer temporary values are required. These include explicitly declared local variables as well as intermediate results being saved to avoid recomputation.

Practice Problem 5.4:

The following shows the code generated from a variant of `combine6` that uses eight-way loop unrolling and four-way parallelism.

```
 1 .L152:
 2   addl (%eax),%ecx
 3   addl 4(%eax),%esi
 4   addl 8(%eax),%edi
 5   addl 12(%eax),%ebx
 6   addl 16(%eax),%ecx
 7   addl 20(%eax),%esi
 8   addl 24(%eax),%edi
 9   addl 28(%eax),%ebx
10   addl $32,%eax
11   addl $8,%edx
12   cmpl -8(%ebp),%edx
13   jl .L152
```

A. What program variable has being spilled onto the stack?

B. At what location on the stack?

C. Why is this a good choice of which value to spill?

With floating-point data, we want to keep all of the local variables in the floating-point register stack. We also need to keep the top of stack available for loading data from memory. This limits us to a degree of parallelism less than or equal to 7.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	211	Abstract unoptimized	42.06	41.86	41.44	160.00
`combine1`	211	Abstract `-O2`	31.25	33.25	31.25	143.00
`combine2`	212	Move `vec_length`	20.66	21.25	21.15	135.00
`combine3`	217	Direct data access	6.00	9.00	8.00	117.00
`combine4`	219	Accumulate in temporary	2.00	4.00	3.00	5.00
`combine5`	234	Unroll $\times 4$	1.50	4.00	3.00	5.00
		Unroll $\times 16$	**1.06**	4.00	3.00	5.00
`combine6`	241	Unroll $\times 2$, Parallelism $\times 2$	1.50	2.00	2.00	2.50
		Unroll $\times 4$, Parallelism $\times 2$	1.50	2.00	1.50	2.50
		Unroll $\times 8$, Parallelism $\times 4$	1.25	**1.25**	**1.50**	**2.00**
Worst:Best			39.7	33.5	27.6	80.0

Figure 5.27: **Comparative Result for All Combining Routines.** The best performing version is shown in bold face.

5.10.3 Limits to Parallelism

For our benchmarks, the main performance limitations are due to the capabilities of the functional units. As Figure 5.12 shows, the integer multiplier and the floating-point adder can only initiate a new operation every clock cycle. This, plus a similar limitation on the load unit limits these cases to a CPE of 1.0. The floating-point multiplier can only initiate a new operation every two clock cycles. This limits this case to a CPE of 2.0. Integer sum is limited to a CPE of 1.0, due to the limitations of the load unit. This leads to the following comparison between the achieved performance versus the theoretical limits:

Method	Integer		Floating Point	
	+	*	+	*
Achieved	1.06	1.25	1.50	2.00
Theoretical Limit	1.00	1.00	1.00	2.00

In this table, we have chosen the combination of unrolling and parallelism that achieves the best performance for each case. We have been able to get close to the theoretical limit for integer sum and product and for floating-point product. Some machine-dependent factor limits the achieved CPE for floating-point multiplication to 1.50 rather than the theoretical limit of 1.0.

5.11 Putting it Together: Summary of Results for Optimizing Combining Code

We have now considered six versions of the combining code, some of which had multiple variants. Let us pause to take a look at the overall effect of this effort, and how our code would do on a different machine.

Figure 5.27 shows the measured performance for all of our routines plus several other variants. As can

be seen, we achieve maximum performance for the integer sum by simply unrolling the loop many times, whereas we achieve maximum performance for the other operations by introducing some, but not too much, parallelism. The overall performance gain of 27.6X and better from our original code is quite impressive.

5.11.1 Floating-Point Performance Anomaly

One of the most striking features of Figure 5.27 is the dramatic drop in the cycle time for floating-point multiplication when we go from `combine3`, where the product is accumulated in memory, to `combine4` where the product is accumulated in a floating-point register. By making this small change, the code suddenly runs 23.4 times faster. When an unexpected result such as this one arises, it is important to hypothesize what could cause this behavior and then devise a series of tests to evaluate this hypothesis.

Examining the table, it appears that something strange is happening for the case of floating-point multiplication when we accumulate the results in memory. The performance is far worse than for floating-point addition or integer multiplication, even though the number of cycles for the functional units are comparable. On an IA32 processor, all floating-point operations are performed in extended 80-bit) precision, and the floating-point registers store values in this format. Only when the value in a register is written to memory is it converted to 32-bit (`float`) or 64-bit (`double`) format.

Examining the data used for our measurements, the source of the problem becomes clear. The measurements were performed on a vector of length 1024 having element i equal to $i + 1$. Hence, we are attempting to compute $1024!$, which is approximately 5.4×10^{2639}. Such a large number can be represented in the extended-precision floating-point format (it can represent numbers up to around 10^{4932}), but it far exceeds what can be represented as a single precision (up to around 10^{38}) or double precision (up to around 10^{308}). The single precision case overflows when we reach $i = 34$, while the double precision case overflows when we reach $i = 171$. Once we reach this point, every execution of the statement

```
*dest = *dest OPER val;
```

in the inner loop of `combine3` requires reading the value $+\infty$, from `dest`, multiplying this by `val` to get $+\infty$ and then storing this back at `dest`. Evidently, some part of this computation requires much longer than the normal five clock cycles required by floating-point multiplication. In fact, running measurements on this operation we find it takes between 110 and 120 cycles to multiply a number by infinity. Most likely, the hardware detected this as a special case and issued a *trap* that caused a software routine to perform the actual computation. The CPU designers felt such an occurrence would be sufficiently rare that they did not need to deal with it as part of the hardware design. Similar behavior could happen with underflow.

When we run the benchmarks on data for which every vector element equals 1.0, `combine3` achieves a CPE of 10.00 cycles for both double and single precision. This is much more in line with the times measured for the other data types and operations, and comparable to the time for `combine4`.

This example illustrates one of the challenges of evaluating program performance. Measurements can be strongly affected by characteristics of the data and operating conditions that initially seem insignificant.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	211	Abstract unoptimized	40.14	47.14	52.07	53.71
`combine1`	211	Abstract `-O2`	25.08	36.05	37.37	32.02
`combine2`	212	Move `vec_length`	19.19	32.18	28.73	32.73
`combine3`	217	Direct data access	6.26	12.52	13.26	13.01
`combine4`	219	Accumulate in temporary	1.76	9.01	8.01	8.01
`combine5`	234	Unroll $\times 4$	1.51	9.01	6.32	6.32
		Unroll $\times 16$	1.25	9.01	6.33	6.22
`combine6`	241	Unroll $\times 4$, Parallelism $\times 2$	1.19	4.69	4.44	4.45
		Unroll $\times 8$, Parallelism $\times 4$	1.15	**4.12**	2.34	**2.01**
		Unroll $\times 8$, Parallelism $\times 8$	**1.11**	4.24	**2.36**	2.08
Worst:Best			36.2	11.4	22.3	26.7

Figure 5.28: **Comparative Result for All Combining Routines Running on a Compaq Alpha 21164 Processor.** The same general optimization techniques are useful on this machine as well.

5.11.2 Changing Platforms

Although we presented our optimization strategies in the context of a specific machine and compiler, the general principles also apply to other machine and compiler combinations. Of course, the optimal strategy may be very machine dependent. As an example, Figure 5.28 shows performance results for a Compaq Alpha 21164 processor for conditions comparable to those for a Pentium III shown in Figure 5.27. These measurements were taken for code generated by the Compaq C compiler, which applies more advanced optimizations than GCC. Observe how the cycle times generally decline as we move down the table, just as they did for the other machine. We see that we can effectively exploit a higher (eight-way) degree of parallelism, because the Alpha has 32 integer and 32 floating-point registers. As this example illustrates, the general principles of program optimization apply to a variety of different machines, even if the particular combination of features leading to optimum performance depend on the specific machine.

5.12 Branch Prediction and Misprediction Penalties

As we have mentioned, modern processors work well ahead of the currently executing instructions, reading new instructions from memory, and decoding them to determine what operations to perform on what operands. This *instruction pipelining* works well as long as the instructions follow in a simple sequence. When a branch is encountered, however, the processor must guess which way the branch will go. For the case of a conditional jump, this means predicting whether or not the branch will be taken. For an instruction such as an indirect jump (as we saw in the code to jump to an address specified by a jump table entry) or a procedure return, this means predicting the target address. In this discussion, we focus on conditional branches.

In a processor that employs *speculative execution*, the processor begins executing the instructions at the predicted branch target. It does this in a way that avoids modifying any actual register or memory locations

code/opt/absval.c

```
1 int absval(int val)
2 {
3     return (val<0) ? -val : val;
4 }
```

code/opt/absval.c

(a) C code.

```
1  absval:
2    pushl %ebp
3    movl %esp,%ebp
4    movl 8(%ebp),%eax    Get val
5    testl %eax,%eax      Test it
6    jge .L3              If >0, goto end
7    negl %eax            Else, negate it
8  .L3:                   end:
9    movl %ebp,%esp
10   popl %ebp
11   ret
```

(b) Assembly code.

Figure 5.29: **Absolute Value Code** We use this to measure the cost of branch misprediction.

until the actual outcome has been determined. If the prediction is correct, the processor simply "commits" the results of the speculatively executed instructions by storing them in registers or memory. If the prediction is incorrect, the processor must discard all of the speculatively executed results, and restart the instruction fetch process at the correct location. A significant *branch penalty* is incurred in doing this, because the instruction pipeline must be refilled before useful results are generated.

Once upon a time, the technology required to support speculative execution was considered too costly and exotic for all but the most advanced supercomputers. Since around 1998, integrated circuit technology has made it possible to put so much circuitry on one chip that some can be dedicated to supporting branch prediction and speculative execution. At this point, almost every processor in a desktop or server machine supports speculative execution.

In optimizing our combining procedure, we did not observe any performance limitation imposed by the loop structure. That is, it appeared that the only limiting factor to performance was due to the functional units. For this procedure, the processor was generally able to predict the direction of the branch at the end of the loop. In fact, if it predicted the branch will always be taken, the processor would be correct on all but the final iteration.

Many schemes have been devised for predicting branches, and many studies have been made on their performance. A common heuristic is to predict that any branch to a lower address will be taken, while any branch to a higher address will not be taken. Branches to lower addresses are used to close loops, and since loops are usually executed many times, predicting these branches as being taken is generally a good idea. Forward branches, on the other hand, are used for conditional computation. Experiments have shown that the backward-taken, forward-not-taken heuristic is correct around 65% of the time. Predicting all branches as being taken, on the other other hand, has a success rate of only around 60%. Far more sophisticated strategies have been devised, requiring greater amounts of hardware. For example, the Intel Pentium II and III processors use a branch prediction strategy that is claimed to be correct between 90% and 95% of the time [29].

We can run experiments to test the branch predication capability of a processor and the cost of a misprediction. We use the absolute value routine shown in Figure 5.29 as our test case. This figure also shows the compiled form. For nonnegative arguments, the branch will be taken to skip over the negation instruction.

We time this function computing the absolute value of every element in an array, with the array consisting of various patterns of of $+1$s and -1s. For regular patterns (e.g., all $+1$s, all -1s, or alternating $+1$ and -1s), we find the function requires between 13.01 and 13.41 cycles. We use this as our estimate of the performance with perfect branch condition. On an array set to random patterns of $+1$s and -1s, we find that the function requires 20.32 cycles. One principle of random processes is that no matter what strategy one uses to guess a sequence of values, if the underlying process is truly random, then we will be right only 50% of the time. For example, no matter what strategy one uses to guess the outcome of a coin toss, as long as the coin toss is fair, our probability of success is only 0.5. Thus, we can see that a mispredicted branch with this processor incurs a penalty of around 14 clock cycles, since a misprediction rate of 50% causes the function to run an average of 7 cycles slower. This means that calls to `absval` require between 13 and 27 cycles depending on the success of the branch predictor.

This penalty of 14 cycles is quite large. For example, if our prediction accuracy were only 65%, then the processor would waste, on average, $14 \times 0.35 = 4.9$ cycles for every branch instruction. Even with the 90 to 95% prediction accuracy claimed for the Pentium II and III, around one cycle is wasted for every branch due to mispredictions. Studies of actual programs show that branches constitute around 14 to 16% of all executed instructions in typical "integer" programs (i.e., those that do not process numeric data), and around 3 to 12% of all executed instructions in typical numeric programs[31, Sect. 3.5]. Thus, any wasted time due to inefficient branch handling can have a significant effect on processor performance.

Many data dependent branches are not at all predictable. For example, there is no basis for guessing whether an argument to our absolute value routine will be positive or negative. To improve performance on code involving conditional evaluation, many processor designs have been extended to include *conditional move* instructions. These instructions allow some forms of conditionals to be implemented without any branch instructions.

With the IA32 instruction set, a number of different `cmov` instructions were added starting with the PentiumPro. These are supported by all recent Intel and Intel-compatible processors. These instructions perform an operation similar to the C code:

```
if (COND)
    x = y;
```

where `y` is the source operand and `x` is the destination operand. The condition *COND* determining whether the copy operation takes place is based on some combination of condition code values, similar to the test and conditional jump instructions. As an example, the `cmovll` instruction performs a copy when the condition codes indicate a value less than zero. Note that the first '`l`' of this instruction indicates "less," while the second is the GAS suffix for long word.

The following assembly code shows how to implement absolute value with conditional move.

```
1   movl 8(%ebp),%eax              Get val as result
2   movl %eax,%edx                 Copy to %edx
3   negl %edx                      Negate %edx
4   testl %eax,%eax                Test val
    Conditionally move %edx to %eax
5   cmovll %edx,%eax               If < 0, copy %edx to result
```

As this code shows, the strategy is to set `val` as a return value, compute `-val`, and conditionally move it to register `%eax` to change the return value when `val` is negative. Our measurements of this code shows that it runs for 13.7 cycles regardless of the data patterns. This clearly yields better overall performance than a procedure that requires between 13 and 27 cycles.

Practice Problem 5.5:

A friend of yours has written an optimizing compiler that makes use of conditional move instructions. You try compiling the following C code:

```
1 /* Dereference pointer or return 0 if null */
2 int deref(int *xp)
3 {
4     return xp ? *xp : 0;
5 }
```

The compiler generates the following code for the body of the procedure.

```
1   movl 8(%ebp),%edx          Get xp
2   movl (%edx),%eax           Get *xp as result
3   testl %edx,%edx            Test xp
4   cmovll %edx,%eax           If 0, copy 0 to result
```

Explain why this code does not provide a valid implementation of `deref`

The current version of GCC does not generate any code using conditional moves. Due to a desire to remain compatible with earlier 486 and Pentium processors, the compiler does not take advantage of these new features. In our experiments, we used the handwritten assembly code shown above. A version using GCC's facility to embed assembly code within a C program (Section 3.15) required 17.1 cycles due to poorer quality code generation.

Unfortunately, there is not much a C programmer can do to improve the branch performance of a program, except to recognize that data-dependent branches incur a high cost in terms of performance. Beyond this, the programmer has little control over the detailed branch structure generated by the compiler, and it is hard to make branches more predictable. Ultimately, we must rely on a combination of good code generation by the compiler to minimize the use of conditional branches, and effective branch prediction by the processor to reduce the number of branch mispredictions.

5.13 Understanding Memory Performance

All of the code we have written, and all the tests we have run, require relatively small amounts of memory. For example, the combining routines were measured over vectors of length 1024, requiring no more than 8,096 bytes of data. All modern processors contain one or more *cache* memories to provide fast access to such small amounts of memory. All of the timings in Figure 5.12 assume that the data being read or written

code/opt/list.c

```
1 typedef struct ELE {
2     struct ELE *next;
3     int data;
4 } list_ele, *list_ptr;
5
6 static int list_len(list_ptr ls)
7 {
8     int len = 0;
9
10    for (; ls; ls = ls->next)
11        len++;
12    return len;
13 }
```

code/opt/list.c

Figure 5.30: **Linked List Functions.** These illustrate the latency of the load operation.

is contained in cache. In Chapter 6, we go into much more detail about how caches work and how to write code that makes best use of the cache.

In this section, we will further investigate the performance of load and store operations while maintaining the assumption that the data being read or written are held in cache. As Figure 5.12 shows, both of these units have a latency of 3, and an issue time of 1. All of our programs so far have used only load operations, and they have had the property that the address of one load depended on incrementing some register, rather than as the result of another load. Thus, as shown in Figures 5.15 to 5.18, 5.21 and 5.26, the load operations could take advantage of pipelining to initiate new load operations on every cycle. The relatively long latency of the load operation has not had any adverse affect on program performance.

5.13.1 Load Latency

As an example of code whose performance is constrained by the latency of the load operation, consider the function `list_len`, shown in Figure 5.30. This function computes the length of a linked list. In the loop of this function, each successive value of variable `ls` depends on the value read by the pointer reference `ls->next`. Our measurements show that function `list_len` has a CPE of 3.00, which we claim is a direct reflection of the latency of the load operation. To see this, consider the assembly code for the loop, and the translation of its first iteration into operations:

Assembly Instructions	Execution Unit Operations
`.L27:`	
`incl %eax`	`incl %eax.0` → `%eax.1`
`movl (%edx),%edx`	`load (%edx.0)` → `%edx.1`
`testl %edx,%edx`	`testl %edx.1,%edx.1` → `cc.1`
`jne .L27`	`jne-taken cc.1`

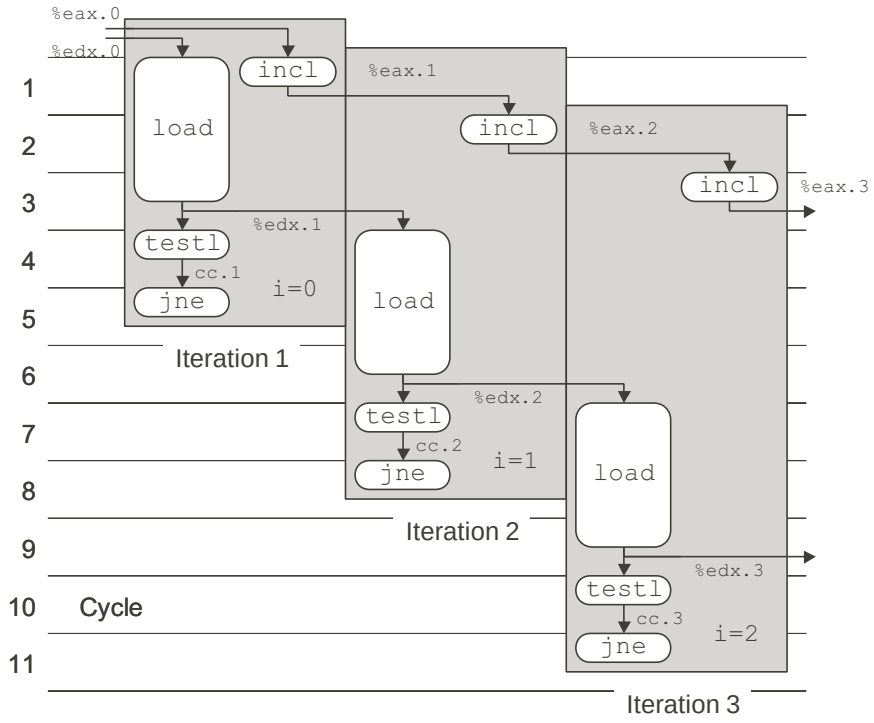


Figure 5.31: **Scheduling of Operations for List Length Function.** The latency of the load operation limits the CPE to a minimum of 3.0.

code/opt/copy.c

```
1 /* Set element of array to 0 */
2 static void array_clear(int *src, int *dest, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         dest[i] = 0;
8 }
9
10 /* Set elements of array to 0, unrolling by 8 */
11 static void array_clear_8(int *src, int *dest, int n)
12 {
13     int i;
14     int len = n - 7;
15
16     for (i = 0; i < len; i+=8) {
17         dest[i] = 0;
18         dest[i+1] = 0;
19         dest[i+2] = 0;
20         dest[i+3] = 0;
21         dest[i+4] = 0;
22         dest[i+5] = 0;
23         dest[i+6] = 0;
24         dest[i+7] = 0;
25     }
26     for (; i < n; i++)
27         dest[i] = 0;
28 }
```

code/opt/copy.c

Figure 5.32: **Functions to Clear Array.** These illustrate the pipelining of the store operation.

Each successive value of register `%edx` depends on the result of a load operation having `%edx` as an operand. Figure 5.31 shows the scheduling of operations for the first three iterations of this function. As can be seen, the latency of the load operation limits the CPE to 3.0.

5.13.2 Store Latency

In all of our examples so far, we have interacted with the memory only by using the load operation to read from a memory location into a register. Its counterpart, the *store* operation, writes a register value to memory. As Figure 5.12 indicates, this operation also has a nominal latency of three cycles, and an issue time of one cycle. However, its behavior, and its interactions with load operations, involve several subtle issues.

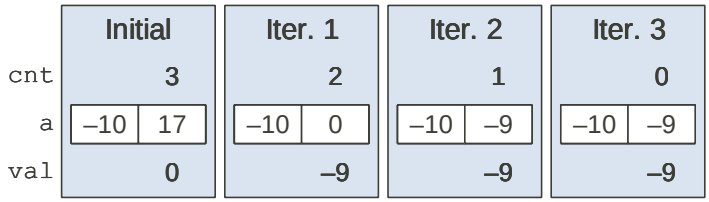
As with the load operation, in most cases the store operation can operate in a fully pipelined mode, beginning

code/opt/copy.c

```
1 /* Write to dest, read from src */
2 static void write_read(int *src, int *dest, int n)
3 {
4     int cnt = n;
5     int val = 0;
6
7     while (cnt--) {
8         *dest = val;
9         val = (*src)+1;
10    }
11 }
```

code/opt/copy.c

Example A: `write_read(&a[0],&a[1],3)`



Example B: `write_read(&a[0],&a[0],3)`

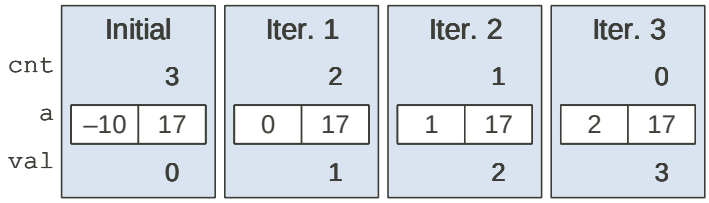


Figure 5.33: **Code to Write and Read Memory Locations, Along with Illustrative Executions.** This function highlights the interactions between stores and loads when arguments `src` and `dest` are equal.

a new store on every cycle. For example, consider the functions shown in Figure 5.32 that set the elements of an array `dest` of length `n` to zero. Our measurements for the first version show a CPE of 2.00. Since each iteration requires a store operation, it is clear that the processor can begin a new store operation at least once every two cycles. To probe further, we try unrolling the loop eight times, as shown in the code for `array_clear_8`. For this one we measure a CPE of 1.25. That is, each iteration requires around ten cycles and issues eight store operations. Thus, we have nearly achieved the optimum limit of one new store operation per cycle.

Unlike the other operations we have considered so far, the store operation does not affect any register values. Thus, by their very nature a series of store operations must be independent from each other. In fact, only a load operation is affected by the result of a store operation, since only a load can read back the memory location that has been written by the store. The function `write_read` shown in Figure 5.33 illustrates the potential interactions between loads and stores. This figure also shows two example executions of this

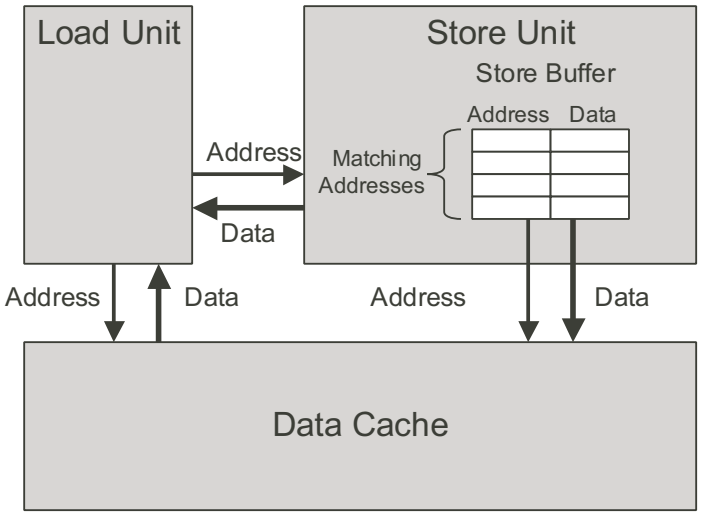


Figure 5.34: **Detail of Load and Store Units.** The store unit maintains a buffer of pending writes. The load unit must check its address with those in the store unit to detect a write/read dependency.

function, when it is called for a two-element array `a`, with initial contents -10 and 17, and with argument `cnt` equal to 3. These executions illustrate some subtleties of the load and store operations.

In example A of Figure 5.33, argument `src` is a pointer to array element `a[0]`, while `dest` is a pointer to array element `a[1]`. In this case, each load by the pointer reference `*src` will yield the value -10. Hence, after two iterations, the array elements will remain fixed at -10 and -9, respectively. The result of the read from `src` is not affected by the write to `dest`. Measuring this example, but over a larger number of iterations, gives a CPE of 2.00.

In example B of Figure 5.33(a), both arguments `src` and `dest` are pointers to array element `a[0]`. In this case, each load by the pointer reference `*src` will yield the value stored by the previous execution of the pointer reference `*dest`. As a consequence, a series of ascending values will be stored in this location. In general, if function `write_read` is called with arguments `src` and `dest` pointing to the same memory location, and with argument `cnt` having some value $n > 0$, the net effect is to set the location to $n - 1$. This example illustrates a phenomenon we will call *write/read dependency*—the outcome of a memory read depends on a very recent memory write. Our performance measurements show that example B has a CPE of 6.00. The write/read dependency causes a slowdown in the processing.

To see how the processor can distinguish between these two cases and why one runs slower than another, we must take a more detailed look at the load and store execution units, as shown in Figure 5.34. The store unit contains a *store buffer* containing the addresses and data of the store operations that have been issued to the store unit, but have not yet been completed, where completion involves updating the data cache. This buffer is provided so that a series of store operations can be executed without having to wait for each one to update the cache. When a load operation occurs, it must check the entries in the store buffer for matching addresses. If it finds a match, it retrieves the corresponding data entry as the result of the load operation.

The assembly code for the inner loop, and its translation into operations during the first iteration, is as follows:

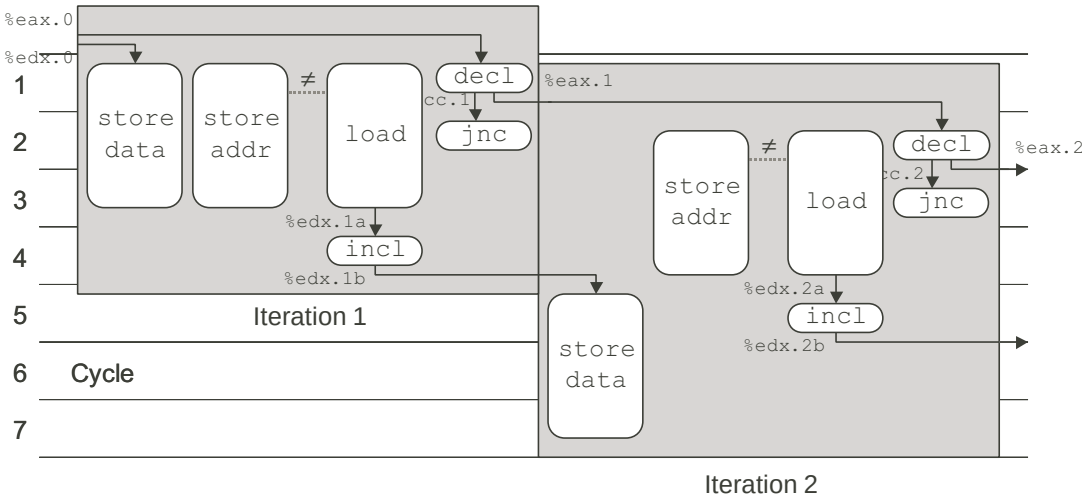


Figure 5.35: **Timing of `write_read` for Example A.** The store and load operations have different addresses, and so the load can proceed without waiting for the store.

Assembly Instructions	Execution Unit Operations
`.L32:`	
`movl %edx,(%ecx)`	`storeaddr (%ecx)`
	`storedata %edx.0`
`movl (%ebx),%edx`	`load (%ebx)` → `%edx.1a`
`incl %edx`	`incl %edx.1a` → `%edx.1b`
`decl %eax`	`decl $eax.0` → `%eax.1`
`jnc .L32`	`jnc-taken cc.1`

Observe that the instruction `movl %edx,(%ecx)` is translated into two operations: the `storeaddr` instruction computes the address for the store operation, creates an entry in the store buffer, and sets the address field for that entry. The `storedata` instruction sets the data field for the entry. Since there is only one store unit, and store operations are processed in program order, there is no ambiguity about how the two operations match up. As we will see, the fact that these two computations are performed independently can be important to program performance.

Figure 5.35 shows the timing of the operations for the first two iterations of `write_read` for the case of example A. As indicated by the dotted line between the `storeaddr` and `load` operations, the `store-addr` operation creates an entry in the store buffer, which is then checked by the `load`. Since these are unequal, the load proceeds to read the data from the cache. Even though the store operation has not been completed, the processor can detect that it will affect a different memory location than the load is trying to read. This process is repeated on the second iteration as well. Here we can see that the `storedata` operation must wait until the result from the previous iteration has been loaded and incremented. Long before this, the `storeaddr` operation and the `load` operations can match up their adddresses, determine they are different, and allow the load to proceed. In our computation graph, we show the load for the second iteration beginning just one cycle after the load from the first. If continued for more iterations, we would find the

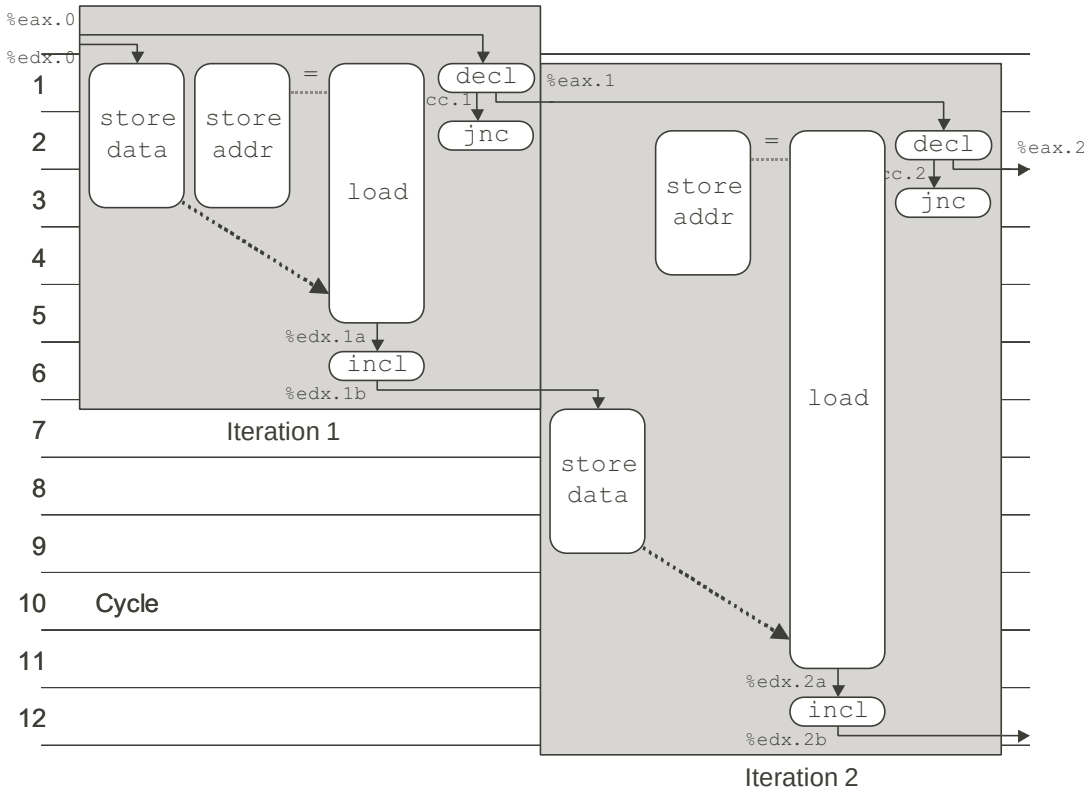


Figure 5.36: **Timing of `write_read` for Example B.** The store and load operations have the same address, and hence the load must wait until it can get the result from the store.

graph indicates a CPE of 1.0. Evidentally, some other resource constraint limits the actual performance to a CPE of 2.0.

Figure 5.36 shows the timing of the operations for the first two iterations of `write_read` for the case of example B. Again, the dotted line between the `storeaddr` and `load` operations indicates that the the `storeaddr` operation creates an entry in the store buffer which is then checked by the `load`. Since these are equal, the load must wait until the `storedata` operation has completed, and then it gets the data from the store buffer. This waiting is indicated in the graph by a much more elongated box for the `load` operation. In addition, we show a dashed arrow from the `storedata` to the `load` operations to indicate that the result of the `storedata` is passed to the `load` as its result. Our timings of these operations are drawn to reflect the measured CPE of 6.0. Exactly how this timing arises is not totally clear, however, and so these figures are intended to be more illustrative than factual. In general, the processor/memory interface is one of the most complex portions of a processor design. Without access to detailed documentation and machine analysis tools, we can only give a hypothetical description of the actual behavior.

As these two examples show, the implementation of memory operations involves many subtleties. With operations on registers, the processor can determine which instructions will affect which others as they are being decoded into operations. With memory operations, on the other hand, the processor cannot predict which will affect which others until the load and store addresses have been computed. Since memory

operations make up a significant fraction of the program, the memory subsystem is optimized to run with greater parallelism for independent memory operations.

Practice Problem 5.6:

As another example of code with potential load-store interactions, consider the following function to copy the contents of one array to another:

```
1 static void copy_array(int *src, int *dest, int n)
2 {
3     int i;
4
5     for (i = 0; i < n; i++)
6         dest[i] = src[i];
7 }
```

Suppose `a` is an array of length 1000 initialized so that each element `a[`i`]` equals i.

A. What would be the effect of the call `copy_array(a+1,a,999)`?

B. What would be the effect of the call `copy_array(a,a+1,999)`?

C. Our performance measurements indicate that the call of part A has a CPE of 3.00, while the call of part B has a CPE of 5.00. To what factor do you attribute this performance difference?

D. What performance would you expect for the call `copy_array(a,a,999)`?

5.14 Life in the Real World: Performance Improvement Techniques

Although we have only considered a limited set of applications, we can draw important lessons on how to write efficient code. We have described a number of basic strategies for optimizing program performance:

1. High-level design. Choose appropriate algorithms and data structures for the problem at hand. Be especially vigilant to avoid algorithms or coding techniques that yield asymptotically poor performance.

2. Basic coding principles. Avoid optimization blockers so that a compiler can generate efficient code.

   (a) Eliminate excessive function calls. Move computations out of loops when possible. Consider selective compromises of program modularity to gain greater efficiency.

   (b) Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.

3. Low-level optimizations.

   (a) Try various forms of pointer versus array code.

   (b) Reduce loop overhead by unrolling loops.

   (c) Find ways to make use of the pipelined functional units by techniques such as iteration splitting.

A final word of advice to the reader is to be careful to avoid expending effort on misleading results. One useful technique is to use checking code to test each version of the code as it is being optimized to make sure no bugs are introduced during this process. Checking code applies a series of tests to the program and makes sure it obtains the desired results. It is very easy to make mistakes when one is introducing new variables, changing loop bounds, and making the code more complex overall. In addition, it is important to notice any unusual or unexpected changes in performance. As we have shown, the selection of the benchmark data can make a big difference in performance comparisons due to performance anomalies, and because we are only executing short instruction sequences.

5.15 Identifying and Eliminating Performance Bottlenecks

Up to this point, we have only considered optimizing small programs, where there is some clear place in the program that requires optimization. When working with large programs, even knowing where to focus our optimizations efforts can be difficult. In this section we describe how to use *code profilers*, analysis tools that collect performance data about a program as it executes. We also present a general principle of system optimization known as *Amdahl's Law*.

5.15.1 Program Profiling

Program *profiling* involves running a version of a program in which instrumentation code has been incorporated to determine how much time the different parts of the program require. It can be very useful for identifying the parts of a program on which we should focus our optimization efforts. One strength of profiling is that it can be performed while running the actual program on realistic benchmark data.

Unix systems provide the profiling program GPROF. This program generates two forms of information. First, it determines how much CPU time was spent for each of the functions in the program. Second, it computes a count of how many times each function gets called, categorized by which function performs the call. Both forms of information can be quite useful. The timings give a sense of the relative importance of the different functions in determining the overall run time. The calling information allows us to understand the dynamic behavior of the program.

Profiling with GPROF requires three steps. We show this for a C program `prog.c`, to be running with command line argument `file.txt`:

1. The program must be compiled and linked for profiling. With GCC (and other C compilers) this involves simply including the run-time flag '`-pg`' on the command line:

   ```
   unix> gcc -O2 -pg prog.c -o prog
   ```

2. The program is then executed as usual:

   ```
   unix> ./prog file.txt
   ```

   It runs slightly (up to a factor of two) slower than normal, but otherwise the only difference is that it generates a file `gmon.out`.

3. GPROF is invoked to analyze the data in `gmon.out`.

```
unix> gprof prog
```

The first part of the profile report lists the times spent executing the different functions, sorted in descending order. As an example, the following shows this part of the report for the first three functions in a program:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 85.62      7.80     7.80        1  7800.00  7800.00  sort_words
  6.59      8.40     0.60   946596     0.00     0.00  find_ele_rec
  4.50      8.81     0.41   946596     0.00     0.00  lower1
```

Each row represents the time spent for all calls to some function. The first column indicates the percentage of the overall time spent on the function. The second shows the cumulative time spent by the functions up to and including the one on this row. The third shows the time spent on this particular function, and the fourth shows how many times it was called (not counting recursive calls). In our example, the function `sort_words` was called only once, but this single call required 7.80 seconds, while the function `lower1` was called 946,596 times, requiring a total of 0.41 seconds.

The second part of the profile report shows the calling history of the function. The following is the history for a recursive function `find_ele_rec`:

```
                                4872758             find_ele_rec [5]
                0.60    0.01  946596/946596         insert_string [4]
[5]      6.7    0.60    0.01  946596+4872758 find_ele_rec [5]
                0.00    0.01   26946/26946          save_string [9]
                0.00    0.00   26946/26946          new_ele [11]
                                4872758             find_ele_rec [5]
```

This history shows both the functions that called `find_ele_rec`, as well as the functions that it called. In the upper part, we find that the function was actually called 5,819,354 times (shown as "`946596+4872758`")—4,872,758 times by itself, and 946,596 times by function `insert_string` (which itself was called 946,596 times). Function `find_ele_rec` in turn called two other functions: `save_string` and `new_ele`, each a total of 26,946 times.

From this calling information, we can often infer useful information about the program behavior. For example, the function `find_ele_rec` is a recursive procedure that scans a linked list looking for a particular string. Given that the ratio of recursive to top-level calls was 5.15, we can infer that it required scanning an average of around 6 elements each time.

Some properties of GPROF are worth noting:

- The timing is not very precise. It is based on a simple *interval counting* scheme, as will be discussed in Chapter 9. In brief, the compiled program maintains a counter for each function recording the time spent executing that function. The operating system causes the program to be interrupted at some regular time interval δ. Typical values of δ range between 1.0 and 10.0 milliseconds. It then determines what function the program was executing when the interrupt occurs and increments the

counter for that function by δ. Of course, it may happen that this function just started executing and will shortly be completed, but it is assigned the full cost of the execution since the previous interrupt. Some other function may run between two interrupts and therefore not be charged any time at all.

Over a long duration, this scheme works reasonably well. Statistically, every function should be charged according to the relative time spent executing it. For programs that run for less than around one second, however, the numbers should be viewed as only rough estimates.

- The calling information is quite reliable. The compiled program maintains a counter for each combination of caller and callee. The appropriate counter is incremented every time a procedure is called.
- By default, the timings for library functions are not shown. Instead, these times are incorporated into the times for the calling functions.

5.15.2 Using a Profiler to Guide Optimization

As an example of using a profiler to guide program optimization, we created an application that involves several different tasks and data structures. This application reads a text file, creates a table of unique words and how many times each word occurs, and then sorts the words in descending order of occurrence. As a benchmark, we ran it on a file consisting of the complete works of William Shakespeare. From this, we determined that Shakespeare wrote a total of 946,596 words, of which 26,946 are unique. The most common word was "the," occurring 29,801 times. The word "love" occurs 2249 times, while "death" occurs 933.

Our program consists of the following parts. We created a series of versions, starting with naive algorithms for the different parts, and then replacing them with more sophisticated ones:

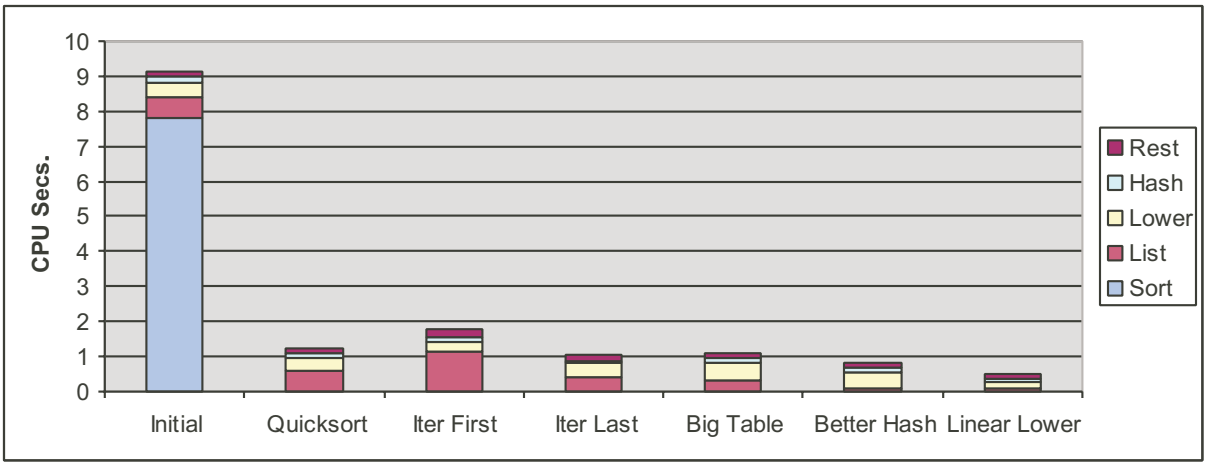
1. Each word is read from the file and converted to lower case. Our initial version used the function `lower1` (Figure 5.7), which we know to have quadratic complexity.
2. A hash function is applied to the string to create a number between 0 and $s - 1$, for a hash table with s buckets. Our initial function simply summed the ASCII codes for the characters modulo s.
3. Each hash bucket is organized as a linked list. The program scans down this list looking for a matching entry. If one is found, the frequency for this word is incremented. Otherwise, a new list element is created. Our initial version performed this operation recursively, inserting new elements at the end of the list.
4. Once the table has been generated, we sort all of the elements according to the frequencies. Our initial version used insertion sort.

Figure 5.37 shows the profile results for different versions of our word-frequency analysis program. For each version, we divide the time into five categories:

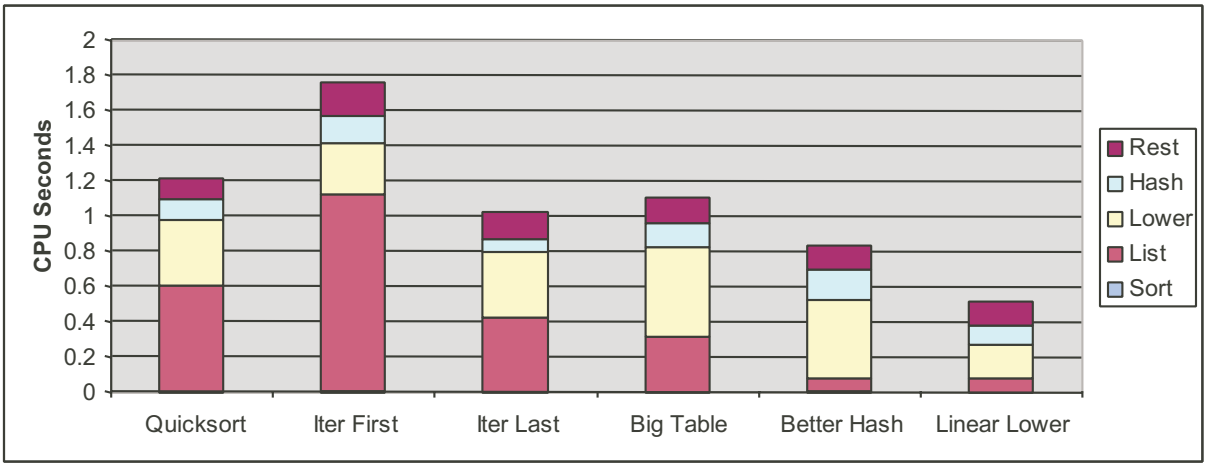
Sort Sorting the words by frequency.

List Scanning the linked list for a matching word, inserting a new element if necessary.

Lower Converting the string to lower case.



(a) All versions.



(b) All but the slowest version.

Figure 5.37: **Profile Results for Different Version of Word Frequency Counting Program.** Time is divided according to the different major operations in the program.

Hash Computing the hash function.

Rest The sum of all other functions.

As part (a) of the figure shows, our initial version requires over 9 seconds, with most of the time spent sorting. This is not surprising, since insertion sort has quadratic complexity, and the program sorted nearly 27,000 values.

In our next version, we performed sorting using the library function `qsort`, which is based on the quicksort algorithm. This version is labeled "Quicksort" in the figure. The more efficient sorting algorithm reduces the time spent sorting to become negligible, and the overall run time to around 1.2 seconds. Part (b) of the figure shows the times for the remaining version on a scale where we can see them better.

With improved sorting, we now find that list scanning becomes the bottleneck. Thinking that the inefficiency is due to the recursive structure of the function, we replaced it by an iterative one, shown as "Iter First." Surprisingly, the run time increases to around 1.8 seconds. On closer study, we find a subtle difference between the two list functions. The recursive version inserted new elements at the end of the list, while the iterative one inserted them at the front. To maximize performance, we want the most frequent words to occur near the beginnings of the lists. That way, the function will quickly locate the common cases. Assuming words are spread uniformly throughout the document, we would expect the first occurrence of a frequent one come before that of a less frequent one. By inserting new words at the end, the first function tended to order words in descending order of frequency, while the second function tended to do just the opposite. We therefore created a third list scanning function that uses iteration but inserts new elements at the end of this list. With this version, shown as "Iter Last," the time dropped to around 1.0 seconds, just slightly better than with the recursive version.

Next, we consider the hash table structure. The initial version had only 1021 buckets (typically, the number of buckets is chosen to be a prime number to enhance the ability of the hash function to distribute keys uniformly among the buckets). For a table with 26,946 entries, this would imply an average *load* of $26946/1007 = 26.4$. That explains why so much of the time is spent performing list operations—the searches involve testing a significant number of candidate words. It also explains why the performance is so sensitive to the list ordering. We then increased the number of buckets to 10,007, reducing the average load to 2.70. Oddly enough, however, our overall run time increased to 1.11 seconds. The profile results indicate that this additional time was mostly spent with the lower-case conversion routine, although this is highly unlikely. Our run times are sufficiently short that we cannot expect very high accuracy with these timings.

We hypothesized that the poor performance with a larger table was due to a poor choice of hash function. Simply summing the character codes does not produce a very wide range of values and does not differentiate according to the ordering of the characters. For example, the words "god" and "dog" would hash to location $147 + 157 + 144 = 448$, since they contain the same characters. The word "foe" would also hash to this location, since $146 + 157 + 145 = 448$. We switched to a hash function that uses shift and EXCLUSIVE-OR operations. With this version, shown as "Better Hash," the time drops to 0.84 seconds. A more systematic approach would be to study the distribution of keys among the buckets more carefully, making sure that it comes close to what one would expect if the hash function had a uniform output distribution.

Finally, we have reduced the run time to the point where one half of the time is spent performing lower-case conversion. We have already seen that function `lower1` has very poor performance, especially for long strings. The words in this document are short enough to avoid the disasterous consequences of quadratic pe-

formance; the longest word ("honorificabilitudinitatibus") is 27 characters long. Still, switching to `lower2`, shown as "Linear Lower" yields a significant performance, with the overall time dropping to 0.52 seconds.

With this exercise, we have shown that code profiling can help drop the time required for a simple application from 9.11 seconds down to 0.52—a factor of 17.5 improvement. The profiler helps us focus our attentionon the most time-consuming parts of the program and also provides useful information about the procedure call structure.

We can see that profiling is a useful tool to have in the toolbox, but it should not be the only one. The timing measurements are imperfect, especially for shorter (under one second) run times. The results apply only to the particular data tested. For example, if we had run the original function on data consisting of a smaller number of longer strings, we would have found that the lower-case conversion routine was the major performance bottleneck. Even worse, if only profiled documents with short words, we might never never detect hidden performance killers such as the quadratic performance of `lower1`. In general, profiling can help us optimize for *typical* cases, assuming we run the program on representative data, but we should also make sure the program will have respectable performance for all possible cases. This is mainly involves avoiding algorithms (such as insertion sort) and bad programming practices (such as `lower1`) that yield poor asymptotic performance.

5.15.3 Amdahl's Law

Gene Amdahl, one of the early pioneers in computing, made a simple, but insightful observation about the effectiveness of improving the performance of one part of a system. This observation is therefore called *Amdahl's Law*. The main idea is that when we speed up one part of a system, the effect on the overall system performance depends on both how significant this part was and how much it sped up. Consider a system where executing some application requires time T_{old}. Suppose, some part of the system requires a fraction α of this time, and that we improve its performance by a factor of k. That is, the component originally required time αT_{old}, and it now requires time $(\alpha T_{old})/k$. The overall execution time will be:

$$\begin{aligned} T_{new} &= (1-\alpha)T_{old} + (\alpha T_{old})/k \\ &= T_{old}[(1-\alpha) + \alpha/k]. \end{aligned}$$

From this, we can compute the speedup $S = T_{old}/T_{new}$ as:

$$S = \frac{1}{(1-\alpha) + \alpha/k} \tag{5.1}$$

As an example, consider the case where a part of the system that initially consumed 60% of the time ($\alpha = 0.6$) is sped up by a factor of 3 ($k = 10$). Then we get a speedup of $1/[0.4 + 0.6/3] = 1.67$. Thus, even though we made a substantial improvement to a major part of the system, our net speedup was significantly less. This is the major insight of Amdahl's Law—to significantly speed up the entire system, we must improve the speed of a very large fraction of the overall system.

Practice Problem 5.7:

The marketing department at your company has promised your customers that the next software release will show a 2X performance improvement. You have been assigned the task of delivering on that

promise. You have determined that only 80% of the system can be improved. How much (i.e., what value of k) would you need to improve this part to meet the overall performance target?

One interesting special case of Amdahl's Law is to consider the case where $k = \infty$. That is, we are able to take some part of the system and speed it up to the point where it takes a negligible amount of time. We then get

$$S_\infty = \frac{1}{(1-\alpha)} \tag{5.2}$$

So, for example, if we can speed up 60% of the system to the point where it requires close to no time, our net speedup will still only be $1/0.4 = 2.5$. We saw this performance with our dictionary program as we replaced insertion sort by quicksort. The initial version spent 7.8 of its 9.1 seconds performing insertion sort, giving $\alpha = .86$. With quicksort, the time spent sorting becomes negligible, giving a predicted speedup of 7.1. In fact the actual speedup was higher: $9.11/1.22 = 7.5$, due to inaccuracies in the profiling measurements for the initial version. We were able to gain a large speedup because sorting constituted a very large fraction of the overall execution time.

Amdahl's Law describes a general principle for improving any process. In addition to applying to speeding up computer systems, it can guide company trying to reduce the cost of manufacturing razor blades, or to a student trying to improve his or her gradepoint average. Perhaps it is most meaningful in the world of computers, where we routinely improve performance by factors of two or more. Such high factors can only be obtained by optimizing a large part of the system.

5.16 Summary

Although most presentations on code optimization describe how compilers can generate efficient code, much can be done by an application programmer to assist the compiler in this task. No compiler can replace an inefficient algorithm or data structure by a good one, and so these aspects of program design should remain a primary concern for programmers. We have also see that optimization blockers, such as memory aliasing and procedure calls, seriously restrict the ability of compilers to perform extensive optimizations. Again, the programmer must take primary responsibility in eliminating these.

Beyond this, we have studied a series of techniques, including loop unrolling, iteration splitting, and pointer arithmetic. As we get deeper into the optimization, it becomes important to study the generated assembly code, and to try to understand how the computation is being performed by the machine. For execution on a modern, out-of-order processor, much can be gained by analyzing how the program would execute on a machine with unlimited processing resources, but where the latencies and the issue times of the functional units match those of the target processor. To refine this analysis, we should also consider such resource constraints as the number and types of functional units.

Programs that involve conditional branches or complex interactions with the memory system are more difficult to analyze and optimize than the simple loop programs we first considered. The basic strategy is to try to make loops more predictable and to try to reduce interactions between store and load operations.

When working with large programs, it becomes important to focus our optimization efforts on the parts that consume the most time. Code profilers and related tools can help us systematically evaluate and improve

program performance. We described GPROF, a standard Unix profiling tool. More sophisticated profilers are available, such as the VTUNE program development system from Intel. These tools can break down the execution time below the procedure level, to measure performance of each *basic block* of the program. A basic block is a sequence of instructions with no conditional operations.

Amdahl's Law provides a simple, but powerful insight into the performance gains obtained by improving just one part of the system. The gain depends both on how much we improve this part and how large a fraction of the overall time this part originally required.

Bibliographic Notes

Many books have been written about compiler optimization techniques. Muchnick's book is considered the most comprehensive [52]. Wadleigh and Crawford's book on software optimization [81] covers some of the material we have, but also describes the process of getting high performance on parallel machines.

Our presentation of the operation of an out-of-order processor is fairly brief and abstract. More complete descriptions of the general principles can be found in advanced computer architecture textbooks, such as the one by Hennessy and Patterson [31, Ch. 4]. Shriver and Smith give a detailed presentation of an AMD processor [65] that bears many similarities to the one we have described.

Amdahl's Law is presented in most books on computer architecture. With its major focus on quantitative system evaluation, Hennessy and Patterson's book [31] provides a particularly good treatment.

Homework Problems

Homework Problem 5.8 [Category 2]:

Suppose that we wish to write a procedure that computes the inner product of two vectors. An abstract version of the function has a CPE of 54 for both integer and floating-point data. By doing the same sort of transformations we did to transform the abstract program `combine1` into the more efficient `combine4`, we get the following code:

```
1  /* Accumulate in temporary */
2  void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3  {
4      int i;
5      int length = vec_length(u);
6      data_t *udata = get_vec_start(u);
7      data_t *vdata = get_vec_start(v);
8      data_t sum = (data_t) 0;
9
10     for (i = 0; i < length; i++) {
11         sum = sum + udata[i] * vdata[i];
12     }
13     *dest = sum;
14 }
```

Our measurements show that this function requires 3.11 cycles per iteration for integer data. The assembly code for the inner loop is:

```
     udata in %esi, vdata in %ebx, i in %edx, sum in %ecx, length in %edi
1 .L24:                                  loop:
2   movl (%esi,%edx,4),%eax                Get udata[i]
3   imull (%ebx,%edx,4),%eax               Multiply by vdata[i]
4   addl %eax,%ecx                         Add to sum
5   incl %edx                              i++
6   cmpl %edi,%edx                         Compare i:length
7   jl .L24                                If <, goto loop
```

Assume that integer multiplication is performed by the general integer functional unit and that this unit is pipelined. This means that one cycle after a multiplication has started, a new integer operation (multiplication or otherwise) can begin. Assume also that the Integer/Branch function unit can perform simple integer operations.

A. Show a translation of these lines of assembly code into a sequence of operations. The `movl` instruction translates into a single `load` operation. Register `%eax` gets updated twice in the loop. Label the different versions `%eax.1a` and `%eax.1b`.

B. Explain how the function can go faster than the number of cycles required for integer multiplication.

C. Explain what factor limits the performance of this code to at best a CPE of 2.5.

D. For floating-point data, we get a CPE of 3.5. Without needing to examine the assembly code, describe a factor that will limit the performance to at best 3 cycles per iteration.

Homework Problem 5.9 [Category 1]:

Write a version of the inner product procedure described in Problem 5.8 that uses four-way loop unrolling.

Our measurements for this procedure give a CPE of 2.20 for integer data and 3.50 for floating point.

A. Explain why any version of any inner product procedure cannot achieve a CPE better than 2.

B. Explain why the performance for floating point did not improve with loop unrolling.

Homework Problem 5.10 [Category 1]:

Write a version of the inner product procedure described in Problem 5.8 that uses four-way loop unrolling and two-way parallelism.

Our measurements for this procedure give a CPE of 2.25 for floating-point data. Describe two factors that limit the performance to a CPE of at best 2.0.

Homework Problem 5.11 [Category 2]:

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iteration:

```
1 int fact(int n)
2 {
3     int i;
4     int result = 1;
5
6     for (i = n; i > 0; i--)
7         result = result * i;
8     return result;
9 }
```

By doing so, they have reduced the number of CPE for the function from 63 to 4, measured on an Intel Pentium III (really!). Still, they would like to do better.

One of the programmers heard about loop unrolling. She generated the following code:

```
1 int fact_u2(int n)
2 {
3     int i;
4     int result = 1;
5     for (i = n; i > 0; i-=2) {
6         result = (result * i) * (i-1);
7     }
8     return result;
9 }
```

Unfortunately, the team discovered that this code returns 0 for some values of argument `n`.

A. For what values of `n` will `fact_u2` and `fact` return different values?

B. Show how to fix `fact_u2`. Note that there is a special trick for this procedure that involves just changing a loop bound.

C. Benchmarking `fact_u2` shows no improvement in performance. How would you explain that?

D. You modify the line inside the loop to read:

```
7         result = result * (i * (i - 1));
```

To everyone's astonishment, the measured performance now has a CPE of 2.5. How do you explain this performance improvement?

Homework Problem 5.12 [Category 1]:

Using the conditional move instruction, write assembly code for the body of the following function:

```
1 /* Return maximum of x and y */
2 int max(int x, int y)
3 {
4     return (x < y) ? y : x;
5 }
```

Homework Problem 5.13 [Category 2]:

Using conditional moves, the general technique for translating a statement of the form:

```
val = cond-expr ? then-expr : else-expr;
```

is to generate code of the form:

```
val = then-expr;
temp = else-expr;
test = cond-expr;
if (test) val = temp;
```

where the last line is implemented with a conditional move instruction. Using the example of Practice Problem 5.5 as a guide, state the general requirements for this translation to be valid.

Homework Problem 5.14 [Category 2]:

The following function computes the sum of the elements in a linked list:

```
1 static int list_sum(list_ptr ls)
2 {
3     int sum = 0;
4
5     for (; ls; ls = ls->next)
6         sum += ls->data;
7     return sum;
8 }
```

The assembly code for the loop, and its translation of the first iteration into operations yields the following:

Assembly Instructions	Execution Unit Operations
.L43:	
addl 4(%edx),%eax	movl 4(%edx.0) → t.1
	addl t.1,%eax.0 → %eax.1
movl (%edx),%edx	load (%edx.0) → %edx.1
testl %edx,%edx	testl %edx.1,%edx.1 → cc.1
jne .L43	jne-taken cc.1

A. Draw a graph showing the scheduling of operations for the first three iterations of the loop, in the style of Figure 5.31. Recall that there is just one load unit.

B. Our measurements for this function give a CPE of 4.00. Is this consistent with the graph you drew in part A?

Homework Problem 5.15 [Category 2]:

The following function is a variant on the list sum function shown in Problem 5.14:

```
 1 static int list_sum2(list_ptr ls)
 2 {
 3     int sum = 0;
 4     list_ptr old;
 5
 6     while (ls) {
 7         old = ls;
 8         ls = ls->next;
 9         sum += old->data;
10     }
11     return sum;
12 }
```

This code is written in such a way that the memory access to fetch the next list element comes before the one to retrieve the data field from the current element.

The assembly code for the loop, and its translation of the first iteration into operations yields the following:

Assembly Instructions	Execution Unit Operations
`.L48:`	
`movl %edx,%ecx`	
`movl (%edx),%edx`	`load (%edx.0)` → `%edx.1`
`addl 4(%ecx),%eax`	`movl 4(%edx.0)` → `t.1`
	`addl t.1,%eax.0` → `%eax.1`
`testl %edx,%edx`	`testl %edx.1,%edx.1` → `cc.1`
`jne .L48`	`jne-taken cc.1`

Note that the register move operation `movl %edx,%ecx` does not require any operations to implement. It is handled by simply associating the tag `edx.0` with register `%ecx`, so that the later instruction `addl 4(%ecx),%eax` is translated to use `edx.0` as its source operand.

A. Draw a graph showing the scheduling of operations for the first three iterations of the loop, in the style of Figure 5.31. Recall that there is just one load unit.

B. Our measurements for this function give a CPE of 3.00. Is this consistent with the graph you drew in part A?

C. How does this function make better use of the load unit than did the function of Problem 5.14?

Chapter 6

The Memory Hierarchy

To this point in our study of systems, we have relied on a simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU. In our simple model, the memory system is a linear array of bytes, and the CPU can access each memory location in a constant amount of time. While this is an effective model as far as it goes, it does not reflect the way that modern systems really work.

In practice, a *memory system* is a hierarchy of storage devices with different capacities, costs, and access times. Registers in the CPU hold the most frequently used data. Small, fast cache memories near the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

In contrast to the uniform access times in our simple system model, memory access times on a real system can vary by factors of ten, or one hundred, or even one million. Unwary programmers who assume a flat, uniform memory risk significant and inexplicable performance slowdowns in their programs. On the other hand, wise programmers who understand the hierarchical nature of memory can use relatively simple techniques to produce efficient programs with fast average memory access times.

In this chapter, we look at the most basic storage technologies of SRAM memory, DRAM memory, and disks. We also introduce a fundamental property of programs known as *locality* and show how locality motivates the organization of memory as a hierarchy of devices. Finally, we focus on the design and performance impact of the cache memories that act as staging areas between the CPU and main memory, and show you how to use your understanding of locality and caching to make your programs run faster.

6.1 Storage Technologies

Much of the success of computer technology stems from the tremendous progress in storage technology. Early computers had a few kilobytes of random-access memory. The earliest IBM PCs didn't even have a hard disk. That changed with the introduction of the IBM PC-XT in 1982, with its 10-megabyte disk. By the year 2000, typical machines had 1000 times as much disk storage and the ratio was increasing by a factor of 10 every two or three years.

6.1.1 Random-Access Memory

Random-access memory (RAM) comes in two varieties—*static* and *dynamic*. *Static RAM* (SRAM) is faster and significantly more expensive than *Dynamic RAM* (DRAM). SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAM stores each bit in a *bistable* memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or *states*. Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable states. Such a memory cell is analogous to the inverted pendulum illustrated in Figure 6.1.

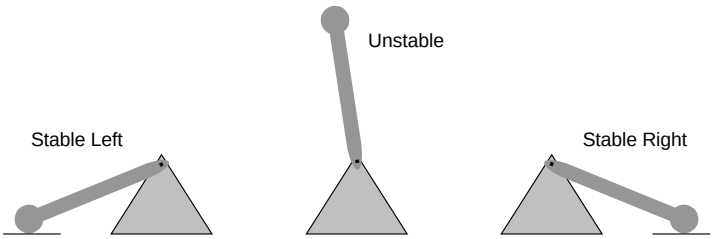


Figure 6.1: **Inverted pendulum.** Like an SRAM cell, the pendulum has only two stable configurations, or *states*.

The pendulum is stable when it is tilted either all the way to the left, or all the way to the right. From any other position, the pendulum will fall to one side or the other. In principle, the pendulum could also remain balanced in a vertical position indefinitely, but this state is *metastable*—the smallest disturbance would make it start to fall, and once it fell it would never return to the vertical position.

Due to its bistable nature, an SRAM memory cell will retain its value indefinitely, as long as it is kept powered. Even when a disturbance, such as electrical noise, perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.

Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads, that is, 30×10^{-15} farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense—each cell consists of a capacitor and a single-access transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

Various sources of leakage current cause a DRAM cell to lose its charge within a time period of around 10 to 100 milliseconds. Fortunately, for computers operating with clock cycles times measured in nanoseconds, this retention time is quite long. The memory system must periodically refresh every bit of memory by reading it out and then rewriting it. Some systems also use error-correcting codes, where the computer words are encoded a few more bits (e.g., a 32-bit word might be encoded using 38 bits), such that circuitry can detect and correct any single erroneous bit within a word.

Figure 6.2 summarizes the characteristics of SRAM and DRAM memory. SRAM is persistent as long as power is applied to them. Unlike DRAM, no refresh is necessary. SRAM can be accessed faster than DRAM. SRAM is not sensitive to disturbances such as light and electrical noise. The tradeoff is that SRAM cells use more transistors than DRAM cells, and thus have lower densities, are more expensive, and consume more power.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative Cost	Applications
SRAM	6	1X	Yes	No	100X	Cache memory
DRAM	1	10X	No	Yes	1X	Main mem, frame buffers

Figure 6.2: **Characteristics of DRAM and SRAM memory.**

Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into d *supercells*, each consisting of w DRAM cells. A $d \times w$ DRAM stores a total of dw bits of information. The supercells are organized as a rectangular array with r rows and c columns, where $rc = d$. Each supercell has an address of the form (i, j), where i denotes the row, and j denotes the column.

For example, Figure 6.3 shows the organization of a 16×8 DRAM chip with $d = 16$ supercells, $w = 8$ bits per supercell, $r = 4$ rows, and $c = 4$ columns. The shaded box denotes the supercell at address $(2, 1)$. Information flows in and out of the chip via external connectors called *pins*. Each pin carries a 1-bit signal. Figure 6.3 shows two of these sets of pins: 8 `data` pins that can transfer one byte in or out of the chip, and 2 `addr` pins that carry 2-bit row and column supercell addresses. Other pins that carry control information are not shown.

Aside: A note on terminology.
The storage community has never settled on a standard name for a DRAM array element. Computer architects tend to refer to it as a "cell", overloading the term with the DRAM storage cell. Circuit designers tend to refer to it as a

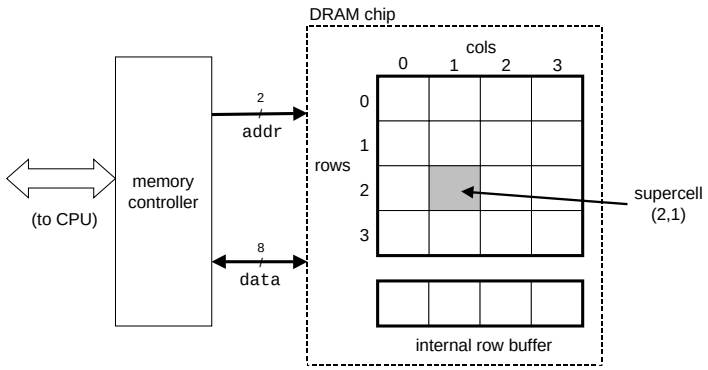
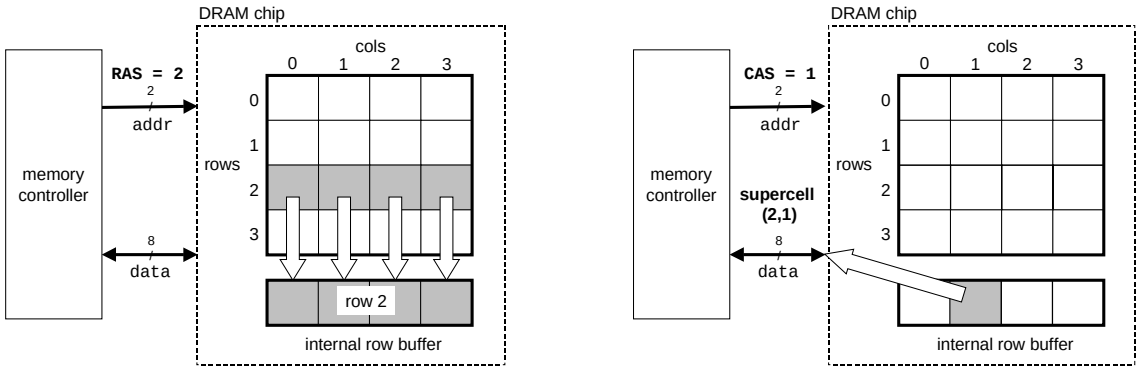


Figure 6.3: **High level view of a 128-bit 16×8 DRAM chip.**

"word", overloading the term with a word of main memory. To avoid confusion, we have adopted the unambiguous term "supercell". **End Aside.**

Each DRAM chip is connected to some circuitry, known as the *memory controller*, that can transfer w bits at a time to and from each DRAM chip. To read the contents of supercell (i, j), the memory controller sends the row address i to the DRAM, followed by the column address j. The DRAM responds by sending the contents of supercell (i, j) back to the controller. The row address i is called a *RAS (Row Access Strobe) request*. The column address j is called a *CAS (Column Access Strobe) request*. Notice that the RAS and CAS requests share the same DRAM address pins.

For example, to read supercell $(2, 1)$ from the 16×8 DRAM in Figure 6.3, the memory controller sends row address 2, as shown in Figure 6.4(a). The DRAM responds by copying the entire contents of row 2 into an internal row buffer. Next, the memory controller sends column address 1, as shown in Figure 6.4(b). The DRAM responds by copying the 8 bits in supercell $(2, 1)$ from the row buffer and sending them to the memory controller.



(a) Select row 2 (RAS request). (b) Select column 1 (CAS request).

Figure 6.4: **Reading the contents of a DRAM supercell.**

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to reduce

the number of address pins on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address pins instead of two. The disadvantage of the two-dimensional array organization is that addresses must be sent in two distinct steps, which increases the access time.

Memory Modules

DRAM chips are packaged in *memory modules* that plug into expansion slots on the main system board (motherboard). Common packages include the 168-pin *Dual Inline Memory Module (DIMM)*, which transfers data to and from the memory controller in 64-bit chunks, and the 72-pin *Single Inline Memory Module (SIMM)*, which transfers data in 32-bit chunks.

Figure 6.5 shows the basic idea of a memory module. The example module stores a total of 64 MB (megabytes) using eight 64-Mbit $8M \times 8$ DRAM chips, numbered 0 to 7. Each supercell stores one byte of *main memory*, and each 64-bit doubleword[1] at byte address A in main memory is represented by the eight supercells whose corresponding supercell address is (i, j). In our example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

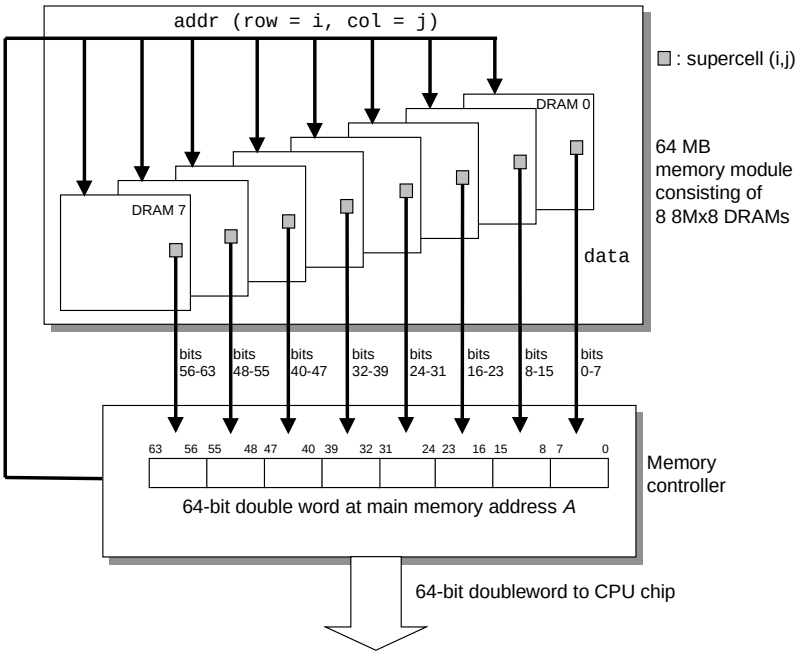


Figure 6.5: **Reading the contents of a memory module.**

To retrieve a 64-bit doubleword at memory address A, the memory controller converts A to a supercell address (i, j) and sends it to the memory module, which then broadcasts i and j to each DRAM. In response, each DRAM outputs the 8-bit contents of its (i, j) supercell. Circuitry in the module collects these outputs and forms them into a 64-bit doubleword, which it returns to the memory controller.

[1] IA32 would call this 64-bit quantity a "quadword."

Main memory can be aggregated by connecting multiple memory modules to the memory controller. In this case, when the controller receives an address A, the controller selects the module k that contains A, converts A to its (i, j) form, and sends (i, j) to module k.

Practice Problem 6.1:

In the following, let r be the number of rows in a DRAM array, c the number of columns, b_r the number of bits needed to address the rows, and b_c the number of bits needed to address the columns. For each of the following DRAMs, determine the power-of-two array dimensions that minimize $\max(b_r, b_c)$, the maximum number of bits needed to address the rows or columns of the array.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1					
16×4					
128×8					
512×4					
1024×4					

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

- *Fast page mode DRAM (FPM DRAM)*. A conventional DRAM copies an entire row of supercells into its internal row buffer, uses one, and then discards the rest. FPM DRAM improves on this by allowing consecutive accesses to the same row to be served directly from the row buffer. For example, to read four supercells from row i of a conventional DRAM, the memory controller must send four RAS/CAS requests, even though the row address i is identical in each case. To read supercells from the same row of an FPM DRAM, the memory controller sends an initial RAS/CAS request, followed by three CAS requests. The initial RAS/CAS request copies row i into the row buffer and returns the first supercell. The next three supercells are served directly from the row buffer, and thus more quickly than the initial supercell.

- *Extended data out DRAM (EDO DRAM)*. An enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time.

- *Synchronous DRAM (SDRAM)*. Conventional, FPM, and EDO DRAMs are asynchronous in the sense that they communicate with the memory controller using a set of explicit control signals. SDRAM replaces many of these control signals with the rising edges of the same external clock signal that drives the memory controller. Without going into detail, the net effect is that an SDRAM can output the contents of its supercells at a faster rate than its asynchronous counterparts.

- *Double Data-Rate Synchronous DRAM (DDR SDRAM)*. DDR SDRAM is an enhancement of SDRAM that doubles the speed of the DRAM by using both clock edges as control signals.

- *Video RAM (VRAM)*. Used in the frame buffers of graphics systems. VRAM is similar in spirit to FPM DRAM. Two major differences are that (1) VRAM output is produced by shifting the entire contents of the internal buffer in sequence, and (2) VRAM allows concurrent reads and writes to the memory. Thus the system can be painting the screen with the pixels in the frame buffer (reads) while concurrently writing new values for the next update (writes).

Aside: Historical popularity of DRAM technologies.
Until 1995, most PC's were built with FPM DRAMs. From 1996-1999, EDO DRAMs dominated the market while FPM DRAMs all but disappeared. SDRAMs first appeared in 1995 in high-end systems, and by 2001 most PC's were built with SDRAMs. **End Aside.**

Nonvolatile Memory

DRAMs and SRAMs are *volatile* in the sense that they lose their information if the supply voltage is turned off. *Nonvolatile memories*, on the other hand, retain their information even when they are powered off. There are a variety of nonvolatile memories. For historical reasons, they are referred to collectively as *read-only memories (ROMs)*, even though some types of ROMs can be written to as well as read. ROMs are distinguished by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them.

A *programmable ROM (PROM)* can be programmed exactly once. PROMs include a sort of fuse with each memory cell that can be blown once by zapping it with a high current. An *erasable programmable ROM (EPROM)* has a small transparent window on the outside of the chip that exposes the memory cells to outside light. The EPROM is reprogrammed by placing it in a special device that shines ultraviolet light onto the storage cells. An EPROM can be reprogrammed on the order of 1,000 times. An *electrically-erasable PROM (EEPROM)* is akin to an EPROM, but it has an internal structure that allows it to be reprogrammed electrically. Unlike EPROMs, EEPROMs do not require a physically separate programming device, and thus can be reprogrammed in-place on printed circuit cards. An EEPROM can be reprogrammed on the order of 10^5 times. *Flash memory* is a family of small nonvolatile memory cards, based on EEPROMs, that can be plugged in and out of a desktop machine, handheld device, or video game console.

Programs stored in ROM devices are often referred to as *firmware*. When a computer system is powered up, it runs firmware stored in a ROM. Some systems provide a small set of primitive input and output functions in firmware, for example, a PC's BIOS (basic input/output system) routines. Complicated devices such as graphics cards and disk drives also rely on firmware to translate I/O (input/output) requests from the CPU.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different

sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of a typical desktop system. The main components are the CPU chip, a chipset that we will call an *I/O bridge* (which includes the memory controller), and the DRAM memory modules that comprise main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O bridge to the main memory.

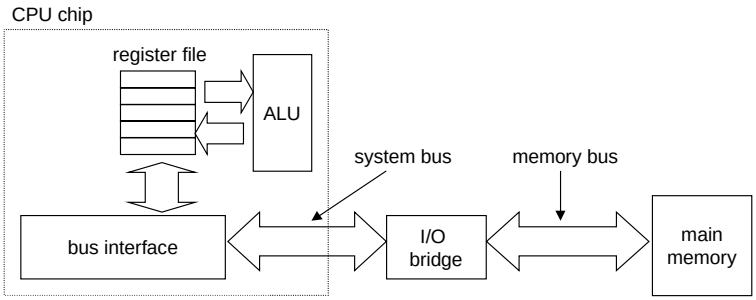


Figure 6.6: **Typical bus structure that connects the CPU and main memory.**

The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

Consider what happens when the CPU performs a load operation such as

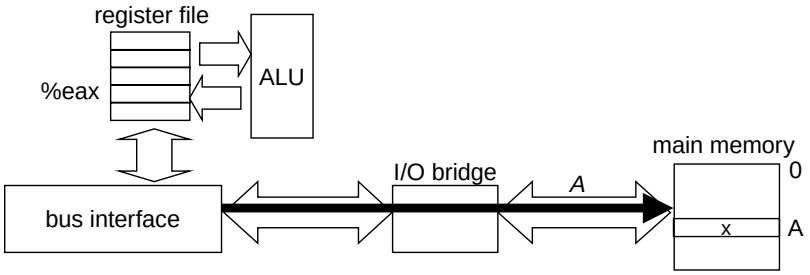
```
movl A,%eax
```

where the contents of address A are loaded into register `%eax`. Circuitry on the CPU chip called the *bus interface* initiates a read transaction on the bus. The read transaction consists of three steps. First, the CPU places the address A on the system bus. The I/O bridge passes the signal along to the memory bus (Figure 6.7(a)). Next, the main memory senses the address signal on the memory bus, reads the address from the memory bus, fetches the data word from the DRAM, and writes the data to the memory bus. The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads it from the bus, and copies it to register `%eax` (Figure 6.7(c)).

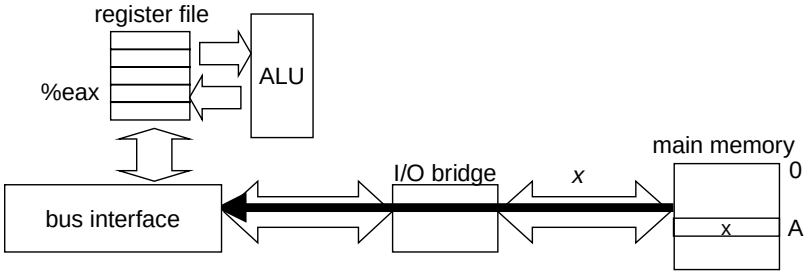
Conversely, when the CPU performs a store instruction such as

```
movl %eax,A
```

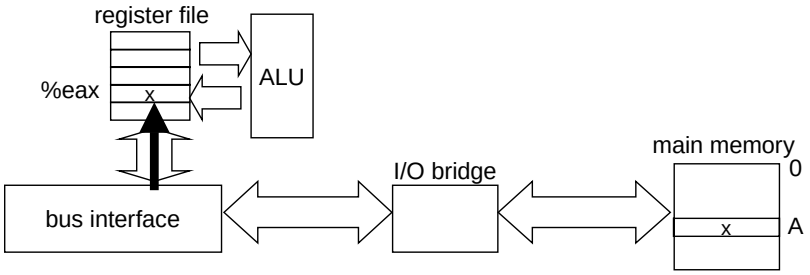
where the contents of register `%eax` are written to address A, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data word in `%eax` to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).



(a) CPU places address A on the memory bus.

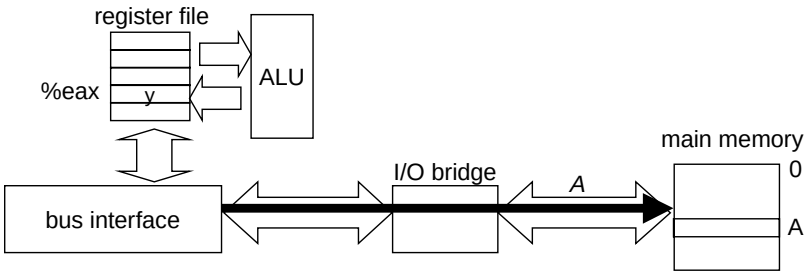


(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.

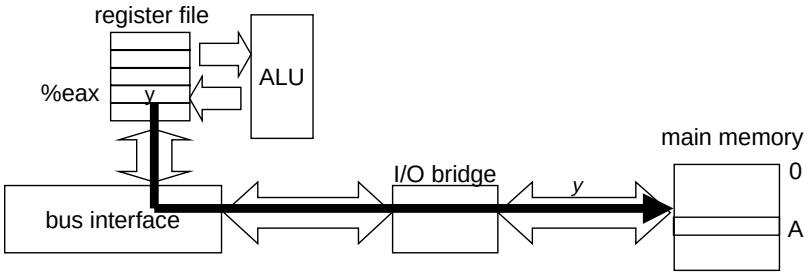


(c) CPU reads word x from the bus, and copies it into register `%eax`.

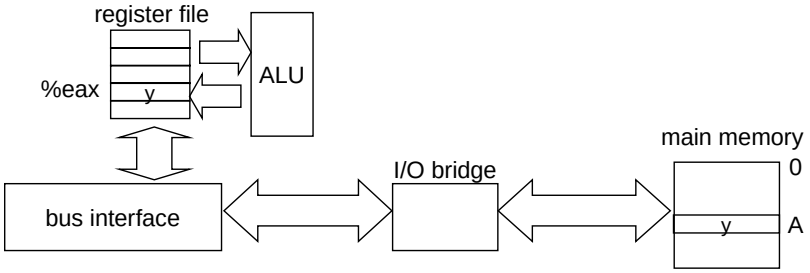
Figure 6.7: **Memory read transaction for a load operation:** `movl A,%eax`**.**



(a) CPU places address A on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word y on the bus.



(c) Main memory reads data word y from the bus and stores it at address A.

Figure 6.8: **Memory write transaction for a store operation:** `movl %eax,A`**.**

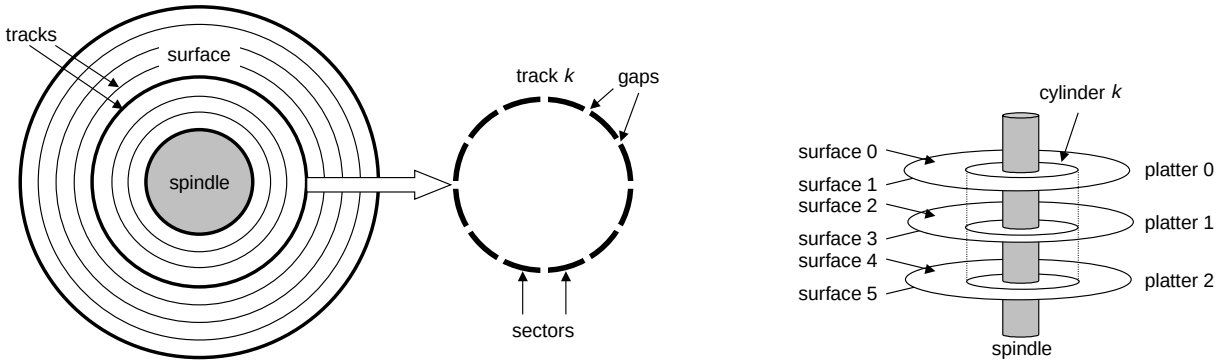
6.1.2 Disk Storage

Disks are workhorse storage devices that hold enormous amounts of data, on the order of tens to hundreds of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.

Disk Geometry

Disks are constructed from *platters*. Each platter consists of two sides, or *surfaces*, that are coated with magnetic recording material. A rotating *spindle* in the center of the platter spins the platter at a fixed *rotational rate*, typically between 5400 and 15,000 *revolutions per minute (RPM)*. A disk will typically contain one or more of these platters encased in a sealed container.

Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called *tracks*. Each track is partitioned into a collection of *sectors*. Each sector contains an equal number of data bits (typically 512 bytes) encoded in the magnetic material on the sector. Sectors are separated by *gaps* where no data bits are stored. Gaps store formatting bits that identify sectors.



(a) Single-platter view. (b) Multiple-platter view.

Figure 6.9: **Disk geometry.**

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a *disk drive*, although we will usually refer to it as simply a *disk*.

Disk manufacturers often describe the geometry of multiple-platter drives in terms of *cylinders*, where a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. For example, if a drive has three platters and six surfaces, and the tracks on each surface are numbered consistently, then cylinder k is the collection of the six instances of track k.

Disk Capacity

The maximum number of bits that can be recorded by a disk is known as its *maximum capacity*, or simply *capacity*. Disk capacity is determined by the following technology factors:

- *Recording density* (*bits*/*in*): The number of bits that can be squeezed into a one-inch segment of a track.
- *Track density* (*tracks*/*in*): The number of tracks that can be squeezed into a one-inch segment of the radius extending from the center of the platter.
- *Areal density* (*bits*/in^2): The product of the recording density and the track density.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every few years. The original disks, designed in an age of low areal density, partitioned every track into the same number of sectors, which was determined by the number of sectors that could be recorded on the innermost track. To maintain a fixed number of sectors per track, the sectors were spaced further apart on the outer tracks. This was a reasonable approach when areal densities were relatively low. However, as areal densities increased, the gaps between sectors (where no data bits were stored) became unacceptably large. Thus, modern high-capacity disks use a technique known as *multiple zone recording*, where the set of tracks is partitioned into disjoint subsets known as *recording zones*. Each zone contains a contiguous collection of tracks. Each track in a zone has the same number of sectors, which is determined by the number of sectors that can be packed into the innermost track of the zone. Note that diskettes (floppy disks) still use the old-fashioned approach, with a constant number of sectors per track.

The capacity of a disk is given by the following:

$$\textit{Disk capacity} = \frac{\textit{\# bytes}}{\textit{sector}} \times \frac{\textit{average \# sectors}}{\textit{track}} \times \frac{\textit{\# tracks}}{\textit{surface}} \times \frac{\textit{\# surfaces}}{\textit{platter}} \times \frac{\textit{\# platters}}{\textit{disk}}$$

For example, suppose we have a disk with 5 platters, 512 bytes per sector, 20,000 tracks per surface, and an average of 300 sectors per track. Then the capacity of the disk is:

$$\begin{aligned} \textit{Disk capacity} &= \frac{\textit{512 bytes}}{\textit{sector}} \times \frac{\textit{300 sectors}}{\textit{track}} \times \frac{\textit{20,000 tracks}}{\textit{surface}} \times \frac{\textit{2 surfaces}}{\textit{platter}} \times \frac{\textit{5 platters}}{\textit{disk}} \\ &= \textit{30,720,000,000 bytes} \\ &= \textit{30.72 GB.} \end{aligned}$$

Notice that manufacturers express disk capacity in units of gigabytes (GB), where $1\ GB = 10^9$ bytes.

Aside: How much is a gigabyte?
Unfortunately, the meanings of prefixes such as kilo (K), mega (M) and giga (G) depend on the context. For measures that relate to the capacity of DRAMs and SRAMs, typically $K = 2^{10}$, $M = 2^{20}$ and $G = 2^{30}$. For measures related to the capacity of I/O devices such as disks and networks, typically $K = 10^3$, $M = 10^6$ and $G = 10^9$. Rates and throughputs usually use these prefix values as well.

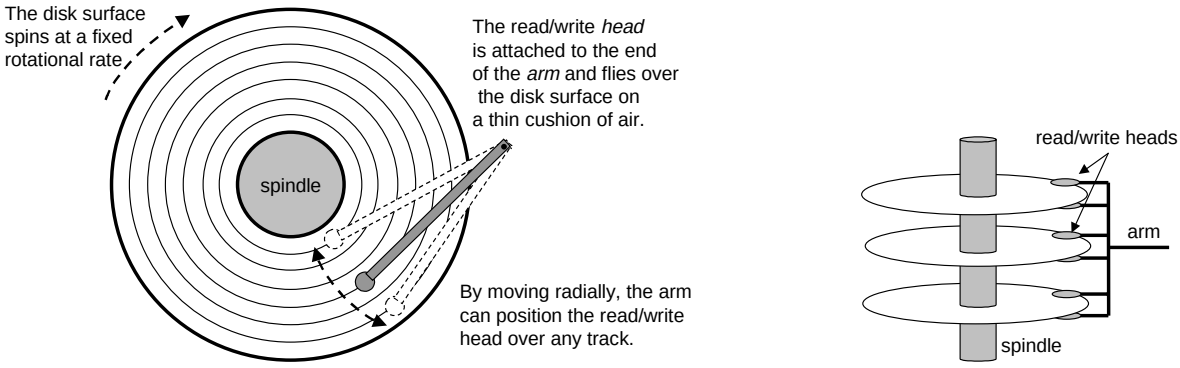
Fortunately, for the back-of-the-envelope estimates that we typically rely on, either assumption works fine in practice. For example, the relative difference between $2^{20} = 1,048,576$ and $10^6 = 1,000,000$ is small: $(2^{20} - 10^6)/10^6 \approx 5\%$. Similarly for $2^{30} = 1,073,741,824$ and $10^9 = 1,000,000,000$: $(2^{30} - 10^9)/10^9 \approx 7\%$. **End Aside.**

Practice Problem 6.2:

What is the capacity of a disk with 2 platters, 10,000 cylinders, an average of 400 sectors per track, and 512 bytes per sector?

Disk Operation

Disks read and write bits stored on the magnetic surface using a *read/write head* connected to the end of an *actuator arm*, as shown in Figure 6.10(a). By moving the arm back and forth along its radial axis the drive can position the head over any track on the surface. This mechanical motion is known as a *seek*. Once the head is positioned over the desired track, then as each bit on the track passes underneath, the head can either sense the value of the bit (read the bit) or alter the value of the bit (write the bit). Disks with multiple platters have a separate read/write head for each surface, as shown in Figure 6.10(b). The heads are lined up vertically and move in unison. At any point in time, all heads are positioned on the same cylinder.



(a) Single-platter view (b) Multiple-platter view

Figure 6.10: **Disk dynamics.**

The read/write head at the end of the arm flies (literally) on a thin cushion of air over the disk surface at a height of about 0.1 microns and a speed of about 80 km/h. This is analogous to placing the Sears Tower on its side and flying it around the world at a height of 2.5 cm (1 inch) above the ground, with each orbit of the earth taking only 8 seconds! At these tolerances, a tiny piece of dust on the surface is a huge boulder. If the head were to strike one of these boulders, the head would cease flying and crash into the surface (a so-called *head crash*). For this reason, disks are always sealed in airtight packages.

Disks read and write data in sector-sized blocks. The *access time* for a sector has three main components: *seek time*, *rotational latency*, and *transfer time*:

- **Seek time:** To read the contents of some target sector, the arm first positions the head over the track that contains the target sector. The time required to move the arm is called the *seek time*. The seek time, T_{seek}, depends on the previous position of the head and the speed that the arm moves across the surface. The average seek time in modern drives, $T_{avg\,seek}$, measured by taking the mean of several

thousand seeks to random sectors, is typically on the order of 6 to 9 ms. The maximum time for a single seek, $T_{max\,seek}$, can be as high as 20 ms.

- **Rotational latency:** Once the head is in position over the track, the drive waits for the first bit of the target sector to pass under the head. The performance of this step depends on the position of the surface when the head arrives at the target sector, and the rotational speed of the disk. In the worst case, the head just misses the target sector, and waits for the disk to make a full rotation. So the maximum rotational latency in seconds is:

$$T_{max\,rotation} = \frac{1}{\textit{RPM}} \times \frac{\textit{60 secs}}{\textit{1 min}}$$

  The average rotational latency, $T_{avg\,rotation}$, is simply half of $T_{max\,rotation}$.

- **Transfer time:** When the first bit of the target sector is under the head, the drive can begin to read or write the contents of the sector. The transfer time for one sector depends on the rotational speed and the number of sectors per track. Thus, we can roughly estimate the average transfer time for one sector in seconds as:

$$T_{avg\,transfer} = \frac{1}{\textit{RPM}} \times \frac{1}{(\textit{average \# sectors/track})} \times \frac{\textit{60 secs}}{\textit{1 min}}$$

We can estimate the average time to access a the contents of a disk sector as the sum of the average seek time, the average rotational latency, and the average transfer time. For example, consider a disk with the following parameters:

Parameter	Value
Rotational rate	7,200 RPM
$T_{avg\,seek}$	9 ms
Average # sectors/track	400

For this disk, the average rotational latency (in ms) is

$$\begin{aligned} T_{avg\,rotation} &= \textit{1/2} \times T_{max\,rotation} \\ &= \textit{1/2} \times (\textit{60 secs / 7,200 RPM}) \times \textit{1000 ms/sec} \\ &\approx \textit{4 ms}. \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\,transfer} &= \textit{60 / 7,200 RPM} \times \textit{1 / 400 sectors/track} \times \textit{1000 ms/sec} \\ &\approx \textit{0.02 ms}. \end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned} T_{access} &= T_{avg\,seek} + T_{avg\,rotation} + T_{avg\,transfer} \\ &= \textit{9 ms} + \textit{4 ms} + \textit{0.02 ms} \\ &= \textit{13.02 ms}. \end{aligned}$$

This example illustrates some important points:

- The time to access the 512 bytes in a disk sector is dominated by the seek time and the rotational latency. Accessing the first byte in the sector takes a long time, but the remaining bytes are essentially free.
- Since the seek time and rotational latency are roughly the same, twice the seek time is a simple and reasonable rule for estimating disk access time.
- The access time for a doubleword stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-sized block from memory is roughly 256 ns for SRAM and 4000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2,500 times greater than DRAM. The difference in access times is even more dramatic if we compare the times to access a single word.

Practice Problem 6.3:

Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational rate	15,000 RPM
$T_{avg\ seek}$	8 ms
Average # sectors/track	500

Logical Disk Blocks

As we have seen, modern disks have complex geometries, with multiple surfaces and different recording zones on those surfaces. To hide this complexity from the operating system, modern disks present a simpler view of their geometry as a sequence of b sector-sized *logical blocks*, numbered $0, 1, \ldots, b-1$. A small hardware/firmware device in the disk, called the *disk controller*, maintains the mapping between logical block numbers and actual (physical) disk sectors.

When the operating system wants to perform an I/O operation such as reading a disk sector into main memory, it sends a command to the disk controller asking it to read a particular logical block number. Firmware on the controller performs a fast table lookup that translates the logical block number into a *(surface, track, sector)* triple that uniquely identifies the corresponding physical sector. Hardware on the controller interprets this triple to move the heads to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed by the head into a small buffer on the controller, and copies them into main memory.

Aside: Formatted disk capacity.
Before a disk can be used to store data, it must be *formatted* by the disk controller. This involves filling in the gaps between sectors with information that identifies the sectors, identifying any cylinders with surface defects and taking them out of action, and setting aside a set of cylinders in each zone as spares that can be called into action if one of more cylinders in the zone goes bad during the lifetime of the disk. The *formatted capacity* quoted by disk manufacturers is less than the maximum capacity because of the existence of these spare cylinders. **End Aside.**

Accessing Disks

Devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an *I/O bus* such as Intel's *Peripheral Component Interconnect* (PCI) bus. Unlike the system bus and memory buses, which are CPU-specific, I/O buses such as PCI are designed to be independent of the underlying CPU. For example, PCs and Macintosh's both incorporate the PCI bus. Figure 6.11 shows a typical I/O bus structure (modeled on PCI) that connects the CPU, main memory, and I/O devices.

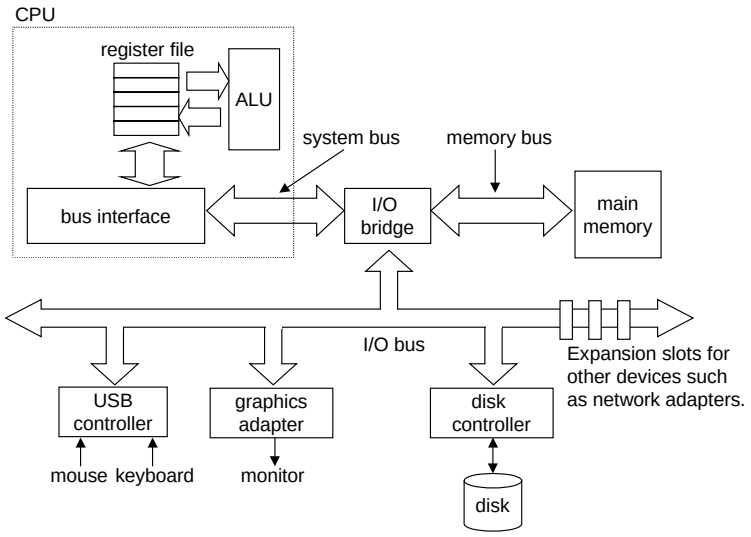


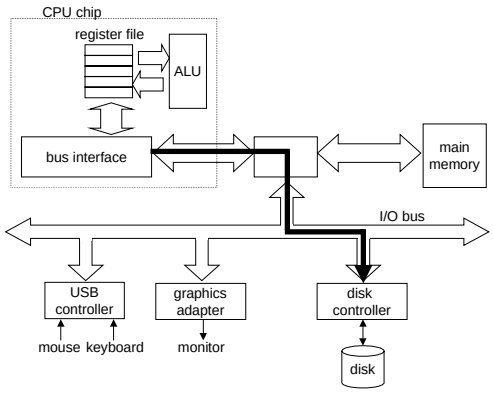
Figure 6.11: **Typical bus structure that connects the CPU, main memory, and I/O devices.**

Although the I/O bus is slower than the system and memory buses, it can accommodate a wide variety of third-party I/O devices. For example, the bus in Figure 6.11 has three different types of devices attached to it.

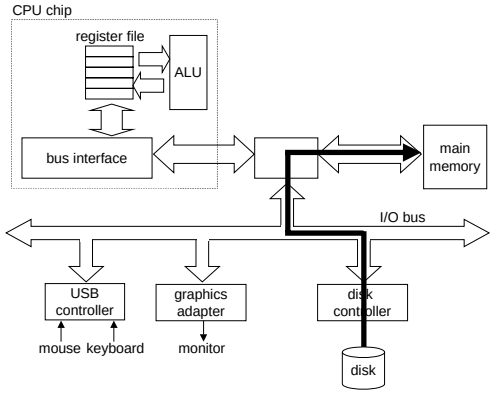
- A *Universal Serial Bus* (USB) controller is a conduit for devices attached to the USB. A USB has a throughput of 12 Mbits/s and is designed for slow to moderate speed serial devices such as keyboards, mice, modems, digital cameras, joysticks, CD-ROM drives, and printers.
- A *graphics card* (or *adapter*) contains hardware and software logic that is responsible for painting the pixels on the display monitor on behalf of the CPU.
- A disk controller contains the hardware and software logic for reading and writing disk data on behalf of the CPU.

Additional devices such as *network adapters* can be attached to the I/O bus by plugging the adapter into empty *expansion slots* on the motherboard that provide a direct electrical connection to the bus.

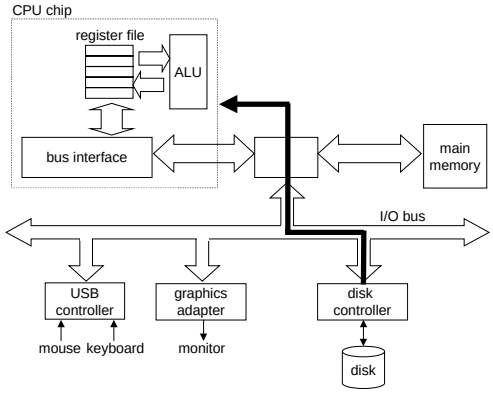
While a detailed description of how I/O devices work and how they are programmed is outside our scope, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



(b) The disk controller reads the sector and performs a DMA transfer into main memory.



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

Figure 6.12: **Reading a disk sector.**

The CPU issues commands to I/O devices using a technique called *memory-mapped I/O* (Figure 6.12(a)). In a system with memory-mapped I/O, a block of addresses in the address space is reserved for communicating with I/O devices. Each of these addresses is known as an *I/O port*. Each device is associated with (or mapped to) one or more ports when it is attached to the bus.

As a simple example, suppose that the disk controller is mapped to port `0xa0`. Then the CPU might initiate a disk read by executing three store instructions to address `0xa`: The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished. (We will discuss interrupts in Section 8.1). The second instruction indicates the number of the logical block that should be read. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is taking place would be enormously wasteful.

After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU (Figure 6.12(b)). This process where a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as *direct memory access (DMA)*. The transfer of data is known as a *DMA transfer*.

After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is currently working on and to jump to an operating system routine. The routine records the fact that the I/O has finished and then returns control to the point where the CPU was interrupted.

Aside: Anatomy of a commercial disk.
Disk manufacturers publish a lot of high-level technical information on their Web pages. For example, if we visit the Web page for the IBM Ultrastar 36LZX disk, we can glean the geometry and performance information shown in Figure 6.13.

Geometry attribute	Value
Platters	6
Surfaces (heads)	12
Sector size	512 bytes
Zones	11
Cylinders	15,110
Recording density (max)	352,000 bits/in.
Track density	20,000 tracks/in.
Areal density (max)	7040 Mbits/sq. in.
Formatted capacity	36 GBbytes

Performance attribute	Value
Rotational rate	10,000 RPM
Avg. rotational latency	2.99 ms
Avg. seek time	4.9 ms
Sustained transfer rate	21–36 MBytes/s

Figure 6.13: **IBM Ultrastar 36LZX geometry and performance.** Source: `www.storage.ibm.com`

Disk manufacturers often neglect to publish detailed technical information about the geometry of the individual recording zones. However, storage researchers have developed a useful tool, called DIXtrac, that automatically discovers a wealth of low-level information about the geometry and performance of SCSI disks [64]. For example,

DIXtrac is able to discover the detailed zone geometry of our example IBM disk, which we've shown in Figure 6.14. Each row in the table characterizes one of the 11 zones on the disk surface, in terms of the number of sectors in the zone, the range of logical blocks mapped to the sectors in the zone, and the range and number of cylinders in the zone.

Zone number	Sectors per track	Starting logical block	Ending logical block	Starting cylinder	Ending cylinder	Cylinders per zone
(outer) 0	504	0	2,292,096	1	380	380
1	476	2,292,097	11,949,751	381	2,078	1,698
2	462	11,949,752	19,416,566	2,079	3,430	1,352
3	420	19,416,567	36,409,689	3,431	6,815	3,385
4	406	36,409,690	39,844,151	6,816	7,523	708
5	392	39,844,152	46,287,903	7,524	8,898	1,375
6	378	46,287,904	52,201,829	8,899	10,207	1,309
7	364	52,201,830	56,691,915	10,208	11,239	1,032
8	352	56,691,916	60,087,818	11,240	12,046	807
9	336	60,087,819	67,001,919	12,047	13,768	1,722
(inner) 10	308	67,001,920	71,687,339	13,769	15,042	1,274

Figure 6.14: **IBM Ultrastar 36LZX zone map.** Source: DIXtrac automatic disk drive characterization tool [64].

The zone map confirms some interesting facts about the IBM disk. First, more tracks are packed into the outer zones (which have a larger circumference) than the inner zones. Second, each zone has more sectors than logical blocks (check this yourself). The unused sectors form a pool of spare cylinders. If the recording material on a sector goes bad, the disk controller will automatically and transparently remap the logical blocks on that cylinder to an available spare. So we see that the notion of a logical block not only provides a simpler interface to the operating system, it also provides a level of indirection that enables the disk to be more robust. This general idea of indirection is very powerful, as we will see when we study virtual memory in Chapter 10. **End Aside.**

6.1.3 Storage Technology Trends

There are several important concepts to take away from our discussion of storage technologies.

- *Different storage technologies have different price and performance tradeoffs.* SRAM is somewhat faster than DRAM, and DRAM is much faster than disk. On the other hand, fast storage is always more expensive than slower storage. SRAM costs more per byte than DRAM. DRAM costs much more than disk.

- *The price and performance properties of different storage technologies are changing at dramatically different rates.* Figure 6.15 summarizes the price and performance properties of storage technologies since 1980, when the first PCs were introduced. The numbers were culled from back issues of trade magazines. Although they were collected in an informal survey, the numbers reveal some interesting trends.

  Since 1980, both the cost and performance of SRAM technology have improved at roughly the same rate. Access times have decreased by a factor of about 100 and cost per megabyte by a factor of 200 (Figure 6.15(a)). However, the trends for DRAM and disk are much more dramatic and divergent.

While the cost per megabyte of DRAM has decreased by a factor of 8000 (almost four orders of magnitude!), DRAM access times have decreased by only a factor of 6 or so (Figure 6.15(b)). Disk technology has followed the same trend as DRAM and in even more dramatic fashion. While the cost of a megabyte of disk storage has plummeted by a factor of 50,000 since 1980, access times have improved much more slowly, by only a factor of 10 or so (Figure 6.15(c)). These startling long-term trends highlight a basic truth of memory and disk technology: it is easier to increase density (and thereby reduce cost) than to decrease access time.

Metric	1980	1985	1990	1995	2000	2000:1980
$/MB	19,200	2,900	320	256	100	190
Access (ns)	300	150	35	15	3	100

(a) SRAM trends

Metric	1980	1985	1990	1995	2000	2000:1980
$/MB	8,000	880	100	30	1	8,000
Access (ns)	375	200	100	70	60	6
Typical size (MB)	0.064	0.256	4	16	64	1,000

(b) DRAM trends

Metric	1980	1985	1990	1995	2000	2000:1980
$/MB	500	100	8	0.30	0.01	50,000
seek time (ms)	87	75	28	10	8	11
typical size (MB)	1	10	160	1,000	20,000	20,000

(c) Disk trends

Metric	1980	1985	1990	1995	2000	2000:1980
Intel CPU	8080	80286	80386	Pentium	P-III	—
CPU clock rate (MHz)	1	6	20	150	600	600
CPU cycle time (ns)	1,000	166	50	6	1.6	600

(d) CPU trends

Figure 6.15: **Storage and processing technology trends.**

- *DRAM and disk access times are lagging behind CPU cycle times.* As we see in Figure 6.15(d), CPU cycle times improved by a factor of 600 between 1980 and 2000. While SRAM performance lags, it is roughly keeping up. However, the gap between DRAM and disk performance and CPU performance is actually widening. The various trends are shown quite clearly in Figure 6.16, which plots the access and cycle times from Figure 6.15 on a semi-log scale.

As we will see in Section 6.4, modern computers make heavy use of SRAM-based caches to try to bridge the processor-memory gap. This approach works because of a fundamental property of application programs known as *locality*, which we discuss next.

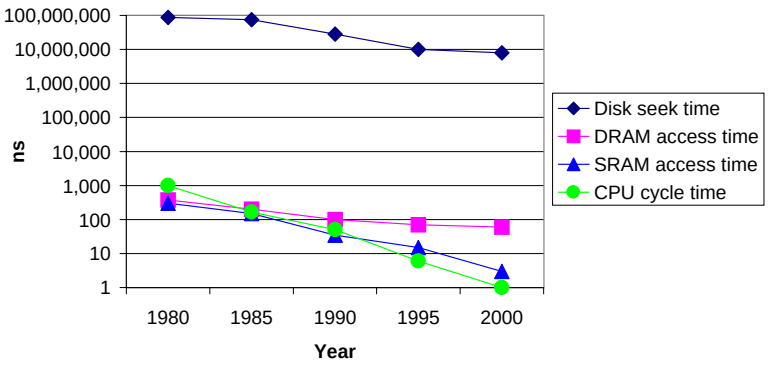


Figure 6.16: **The increasing gap between DRAM, disk, and CPU speeds.**

6.2 Locality

Well-written computer programs tend to exhibit good *locality*. That is, they tend to reference data items that are near other recently referenced data items, or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

Locality is typically described as having two distinct forms: *temporal locality* and *spatial locality*. In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Programmers should understand the principle of locality because, in general, *programs with good locality run faster than programs with poor locality*. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality. At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as *cache memories* that hold blocks of the most recently referenced instructions and data items. At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system. The principle of locality also plays a crucial role in the design of application programs. For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk. High volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

6.2.1 Locality of References to Program Data

Consider the simple function in Figure 6.17(a) that sums the elements of a vector. Does this function have good locality? To answer this question, we look at the reference pattern for each variable. In this example, the `sum` variable is referenced once in each loop iteration, and thus there is good temporal locality with respect to `sum`. On the other hand, since `sum` is a scalar, there is no spatial locality with respect to `sum`.

```
1 int sumvec(int v[N])
2 {
3     int i, sum = 0;
4
5     for (i = 0; i < N; i++)
6         sum += v[i];
7     return sum;
8 }
```

(a)

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

(b)

Figure 6.17: **(a) A function with good locality. (b) Reference pattern for vector `v` ($N = 8$).** Notice how the vector elements are accessed in the same order that they are stored in memory.

As we see in Figure 6.17(b), the elements of vector `v` are read sequentially, one after the other, in the order they are stored in memory (we assume for convenience that the array starts at address 0). Thus, with respect to variable `v`, the function has good spatial locality, but poor temporal locality since each vector element is accessed exactly once. Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the `sumvec` function enjoys good locality.

A function such as `sumvec` that visits each element of a vector sequentially is said to have a *stride-1 reference pattern* (with respect to the element size). Visiting every kth element of a contiguous vector is called a *stride-k reference pattern*. Stride-1 reference patterns are a common and important source of spatial locality in programs. In general, as the stride increases, the spatial locality decreases.

Stride is also an important issue for programs that reference multidimensional arrays. Consider the `sumarrayrows` function in Figure 6.18(a) that sums the elements of a two-dimensional array. The doubly nested loop reads the elements of the array in row-major order. That is, the inner loop reads the elements of the first row, then the second row, and so on. The `sumarrayrows` function enjoys good spatial locality because

```
1 int sumarrayrows(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (i = 0; i < M; i++)
6         for (j = 0; j < N; j++)
7             sum += a[i][j];
8     return sum;
9 }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

(b)

Figure 6.18: **(a) Another function with good locality. (b) Reference pattern for array `a` ($M = 2$, $N = 3$).** There is good spatial locality because the array is accessed in the same row-major order that it is stored in memory.

it references the array in the same row-major order that the array is stored (Figure 6.18(b)). The result is a nice stride-1 reference pattern with excellent spatial locality.

Seemingly trivial changes to a program can have a big impact on its locality. For example, the `sumarraycols` function in Figure 6.19(a) computes the same result as the `sumarrayrows` function in Figure 6.18(a). The only difference is that we have interchanged the i and j loops. What impact does interchanging the loops have on its locality? The `sumarraycols` function suffers from poor spatial locality

```
1 int sumarraycols(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

(b)

Figure 6.19: **(a) A function with poor spatial locality. (b) Reference pattern for array** `a` **(**$M = 2$**,** $N = 3$**).** The function has poor spatial locality because it scans memory with a stride-$(N \times sizeof(int))$ reference pattern.

because it scans the array column-wise instead of row-wise. Since C arrays are laid out in memory row-wise, the result is a stride-$(N \times sizeof(int))$ reference pattern, as shown in Figure 6.19(b).

6.2.2 Locality of Instruction Fetches

Since program instructions are stored in memory and must be fetched (read) by the CPU, we can also evaluate the locality of a program with respect to its instruction fetches. For example, in Figure 6.17 the instructions in the body of the `for` loop are executed in sequential memory order, and thus the loop enjoys good spatial locality. Since the loop body is executed multiple times, it also enjoys good temporal locality.

An important property of code that distinguishes it from program data is that it can not be modified at runtime. While a program is executing, the CPU only reads its instructions from memory. The CPU never overwrites or modifies these instructions.

6.2.3 Summary of Locality

In this section we have introduced the fundamental idea of locality and we have identified some simple rules for qualitatively evaluating the locality in a program:

- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-k reference patterns, the smaller the stride the better the spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.

- Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.

Later in this chapter, after we have learned about cache memories and how they work, we will show you how to quantify the idea of locality in terms of cache hits and misses. It will also become clear to you why programs with good locality typically run faster than programs with poor locality. Nonetheless, knowing how to glance at a source code and getting a high-level feel for the locality in the program is a useful and important skill for a programmer to master.

Practice Problem 6.4:

Permute the loops in the following function so that it scans the three-dimensional array a with a stride-1 reference pattern.

```
1 int sumarray3d(int a[N][N][N])
2 {
3     int i, j, k, sum = 0;
4
5     for (i = 0; i < N; i++) {
6         for (j = 0; j < N; j++) {
7             for (k = 0; k < N; k++) {
8                 sum += a[k][i][j];
9             }
10        }
11    }
12    return sum;
13 }
```

Practice Problem 6.5:

The three functions in Figure 6.20 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

6.3 The Memory Hierarchy

Sections 6.1 and 6.2 described some fundamental and enduring properties of storage technology and computer software:

- Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.

- Well-written programs tend to exhibit good locality.

```
1 #define N 1000
2
3 typedef struct {
4     int vel[3];
5     int acc[3];
6 } point;
7
8 point p[N];
```

(a) An array of `structs`.

```
1 void clear1(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++)
7             p[i].vel[j] = 0;
8         for (j = 0; j < 3; j++)
9             p[i].acc[j] = 0;
10    }
11 }
```

(b) The `clear1` function.

```
1 void clear2(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++) {
7             p[i].vel[j] = 0;
8             p[i].acc[j] = 0;
9         }
10    }
11 }
```

(a) The `clear2` function.

```
1 void clear3(point *p, int n)
2 {
3     int i, j;
4
5     for (j = 0; j < 3; j++) {
6         for (i = 0; i < n; i++)
7             p[i].vel[j] = 0;
8         for (i = 0; i < n; i++)
9             p[i].acc[j] = 0;
10    }
11 }
```

(b) The `clear3` function.

Figure 6.20: **Code examples for Practice Problem 6.5.**

In one of the happier coincidences of computing, these fundamental properties of hardware and software complement each other beautifully. Their complementary nature suggests an approach for organizing memory systems, known as the *memory hierarchy*, that is used in all modern computer systems. Figure 6.21 shows a typical memory hierarchy. In general, the storage devices get faster, cheaper, and larger as we move

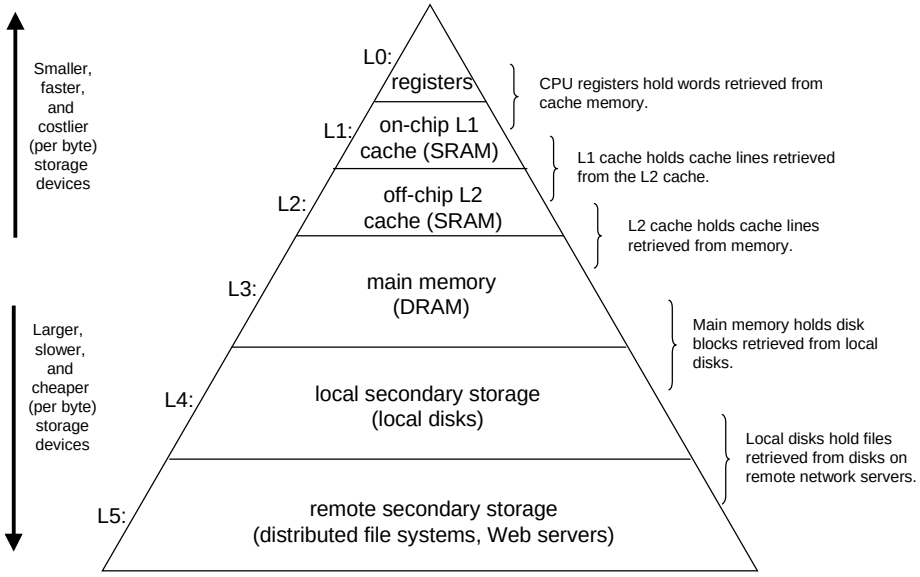


Figure 6.21: **The memory hierarchy.**

from higher to lower *levels*. At the highest level (L0) are a small number of fast CPU registers that the CPU can access in a single clock cycle. Next are one or more small to moderate-sized SRAM-based cache memories that can be accessed in a few CPU clock cycles. These are followed by a large DRAM-based main memory that can be accessed in tens to hundreds of clock cycles. Next are slow but enormous local disks. Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network. For example, distributed file systems such as the Andrew File System (AFS) or the Network File System (NFS) allow a program to access files that are stored on remote network-connected servers. Similarly, the World Wide Web allows programs to access remote files stored on Web servers anywhere in the world.

Aside: Other memory hierarchies.
We have shown you one example of a memory hierarchy, but other combinations are possible, and indeed common. For example, many sites back up local disks onto archival magnetic tapes. At some of these sites, human operators manually mount the tapes onto tape drives as needed. At other sites, tape robots handle this task automatically. In either case, the collection of tapes represents a level in the memory hierarchy, below the local disk level, and the same general principles apply. Tapes are cheaper per byte than disks, which allows sites to archive multiple snapshots of their local disks. The tradeoff is that tapes take longer to access than disks. **End Aside.**

6.3.1 Caching in the Memory Hierarchy

In general, a *cache* (pronounced "cash") is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device. The process of using a cache is known as *caching* (pronounced "cashing").

The central idea of a memory hierarchy is that for each k, the faster and smaller storage device at level k serves as a cache for the larger and slower storage device at level $k+1$. In other words, each level in the hierarchy caches data objects from the next lower level. For example, the local disk serves as a cache for files (such as Web pages) retrieved from remote disks over the network, the main memory serves as a cache for data on the local disks, and so on, until we get to the smallest cache of all, the set of CPU registers.

Figure 6.22 shows the general concept of caching in a memory hierarchy. The storage at level $k+1$ is partitioned into contiguous chunks of data objects called *blocks*. Each block has a unique address or name that distinguishes it from other blocks. Blocks can be either fixed-size (the usual case) or variable-sized (e.g., the remote HTML files stored on Web servers). For example, the level-$k+1$ storage in Figure 6.22 is partitioned into 16 fixed-sized blocks, numbered 0 to 15.

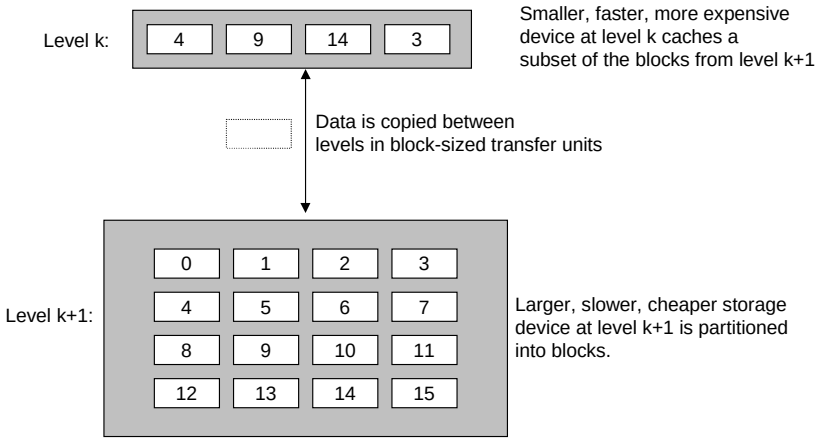


Figure 6.22: **The basic principle of caching in a memory hierarchy.**

Similarly, the storage at level k is partitioned into a smaller set of blocks that are the same size as the blocks at level $k+1$. At any point in time, the cache at level k contains copies of a subset of the blocks from level $k+1$. For example, in Figure 6.22, the cache at level k has room for four blocks and currently contains copies of blocks 4, 9, 14, and 3.

Data is always copied back and forth between level k and level $k+1$ in block-sized *transfer units*. It is important to realize that while the block size is fixed between any particular pair of adjacent levels in the hierarchy, other pairs of levels can have different block sizes. For example, in Figure 6.21, transfers between L1 and L0 typically use 1-word blocks. Transfers between L2 and L1 (and L3 and L2) typically use blocks of 4 to 8 words. And transfers between L4 and L3 use blocks with hundreds or thousands of bytes. In general, devices lower in the hierarchy (further from the CPU) have longer access times, and thus tend to use larger block sizes in order to amortize these longer access times.

Cache Hits

When a program needs a particular data object d from level $k+1$, it first looks for d in one of the blocks currently stored at level k. If d happens to be cached at level k, then we have what is called a *cache hit*. The program reads d directly from level k, which by the nature of the memory hierarchy is faster than reading d from level $k+1$. For example, a program with good temporal locality might read a data object from block 14, resulting in a cache hit from level k.

Cache Misses

If, on the other hand, the data object d is not cached at level k, then we have what is called a *cache miss*. When there is a miss, the cache at level k fetches the block containing d from the cache at level $k+1$, possibly overwriting an existing block if the level k cache is already full.

This process of overwriting an existing block is known as *replacing* or *evicting* the block. The block that is evicted is sometimes referred to as a *victim block*. The decision about which block to replace is governed by the cache's *replacement policy*. For example, a cache with a *random replacement policy* would choose a random victim block. A cache with a least-recently used (LRU) replacement policy would choose the block that was last accessed the furthest in the past.

After the cache at level k has fetched the block from level $k+1$, the program can read d from level k as before. For example, in Figure 6.22, reading a data object from block 12 in the level k cache would result in a cache miss because block 12 is not currently stored in the level k cache. Once it has been copied from level $k+1$ to level k, block 12 will remain there in expectation of later accesses.

Kinds of Cache Misses

It is sometimes helpful to distinguish between different kinds of cache misses. If the cache at level k is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a *cold cache*, and misses of this kind are called *compulsory misses* or *cold misses*. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been *warmed up* by repeated memory accesses.

Whenever there is a miss, the cache at level k must implement some *placement policy* that determines where to place the block it has retrieved from level $k+1$. The most flexible placement policy is to allow any block from level $k+1$ to be stored in any block at level k. For caches high in the memory hierarchy (close to the CPU) that are implemented in hardware and where speed is at a premium, this policy is usually too expensive to implement because randomly placed blocks are expensive to locate.

Thus, hardware caches typically implement a more restricted placement policy that restricts a particular block at level $k+1$ to a small subset (sometimes a singleton) of the blocks at level k. For example, in Figure 6.22, we might decide that a block i at level $k+1$ must be placed in block (i mod 4) at level k. For example, blocks 0, 4, 8, and 12 at level $k+1$ would map to block 0 at level k, blocks 1, 5, 9, and 13 would map to block 1, and so on. Notice that our example cache in Figure 6.22 uses this policy.

Restrictive placement policies of this kind lead to a type of miss known as a *conflict miss*, where the cache

is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing. For example, in Figure 6.22, if the program requests block 0, then block 8, then block 0, then block 8, and so on, each of the references to these two blocks would miss in the cache at level k, even though this cache can hold a total of 4 blocks.

Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the *working set* of the phase. When the size of the working set exceeds the size of the cache, the cache will experience what are known as *capacity misses*. In other words, the cache is just too small to handle this particular working set.

Cache Management

As we have noted, the essence of the memory hierarchy is that the storage device at each level is a cache for the next lower level. At each level, some form of logic must *manage* the cache. By this we mean that something has to partition the cache storage into blocks, transfer blocks between different levels, decide when there are hits and misses, and then deal with them. The logic that manages the cache can be hardware, software, or a combination of the two.

For example, the compiler manages the register file, the highest level of the cache hierarchy. It decides when to issue loads when there are misses, and determines which register to store the data in. The caches at levels L1 and L2 are managed entirely by hardware logic built into the caches. In a system with virtual memory, the DRAM main memory serves as a cache for data blocks stored on disk, and is managed by a combination of operating system software and address translation hardware on the CPU. For a machine with a distributed file system such as AFS, the local disk serves as a cache that is managed by the AFS client process running on the local machine. In most cases, caches operate automatically and do not require any specific or explicit actions from the program.

6.3.2 Summary of Memory Hierarchy Concepts

To summarize, memory hierarchies based on caching work because slower storage is cheaper than faster storage and because programs tend to exhibit locality.

- *Exploiting temporal locality.* Because of temporal locality, the same data objects are likely to be reused multiple times. Once a data object has been copied into the cache on the first miss, we can expect a number of subsequent hits on that object. Since the cache is faster than the storage at the next lower level, these subsequent hits can be served much faster than the original miss.
- *Exploiting spatial locality.* Blocks usually contain multiple data objects. Because of spatial locality, we can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

Caches are used everywhere in modern systems. As you can see from Figure 6.23, caches are used in CPU chips, operating systems, distributed file systems, and on the World-Wide Web. They are built from and managed by various combinations of hardware and software. Note that there are a number of terms and

acronyms in Figure 6.23 that we haven't covered yet. We include them here to demonstrate how common caches are.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	32-byte block	On-chip L1 cache	1	Hardware
L2 cache	32-byte block	Off-chip L2 cache	10	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.23: **The ubiquity of caching in modern computer systems.** Acronyms: TLB: Translation Lookaside Buffer, MMU: Memory Management Unit, OS: Operating System, AFS: Andrew File System, NFS: Network File System.

6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main DRAM memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert a small SRAM memory, called an *L1 cache* (Level 1 cache), between the CPU register file and main memory. In modern systems, the L1 cache is located on the CPU chip (i.e., it is an *on-chip cache*), as shown in Figure 6.24. The L1 cache can be accessed nearly as fast as the registers, typically in one or two clock cycles.

As the performance gap between the CPU and main memory continued to increase, system designers responded by inserting an additional cache, called an *L2 cache*, between the L1 cache and the main memory, that can be accessed in a few clock cycles. The L2 cache can be attached to the memory bus, or it can be attached to its own *cache bus*, as shown in Figure 6.24. Some high-performance systems, such as those based on the Alpha 21164, will even include an additional level of cache on the memory bus, called an *L3 cache*, which sits between the L2 cache and main memory in the hierarchy. While there is considerable variety in the arrangements, the general principles are the same.

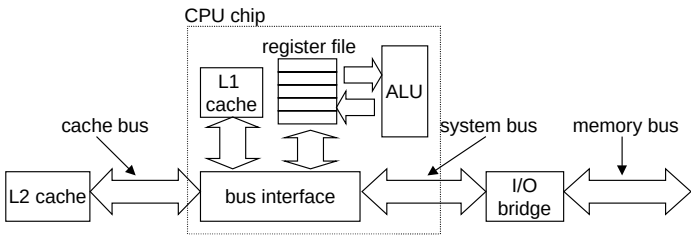
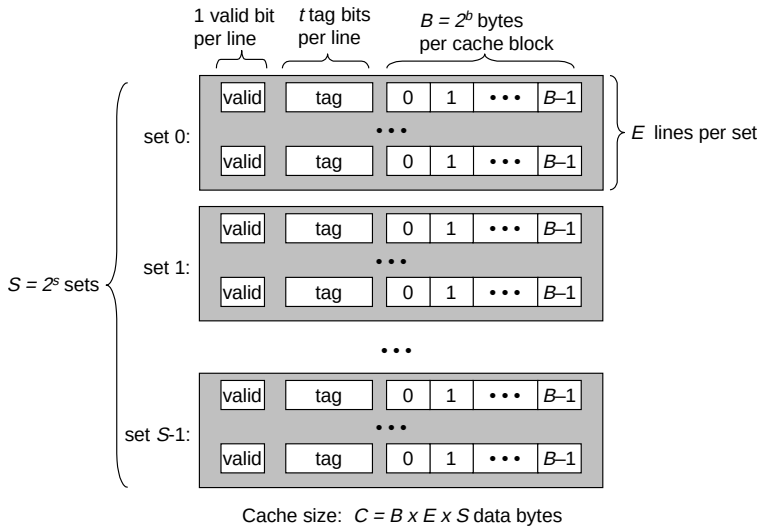


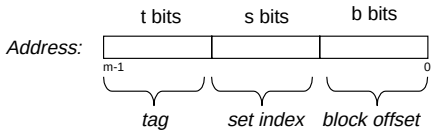
Figure 6.24: **Typical bus structure for L1 and L2 caches.**

6.4.1 Generic Cache Memory Organization

Consider a computer system where each memory address has m bits that form $M = 2^m$ unique addresses. As illustrated in Figure 6.25(a), a cache for such a machine is organized as an array of $S = 2^s$ *cache sets*. Each set consists of E *cache lines*. Each line consists of a data *block* of $B = 2^b$ bytes, a *valid bit* that indicates whether or not the line contains meaningful information, and $t = m - (b + s)$ *tag bits* (a subset of the bits from the current block's memory address) that uniquely identify the block stored in the cache line.



(a)



(b)

Figure 6.25: **General organization of cache** (S, E, B, m)**.** (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.

In general, a cache's organization can be characterized by the tuple (S, E, B, m). The size (or capacity) of a cache, C, is stated in terms of the aggregate size of all the blocks. The tag bits and valid bit are not included. Thus, $C = S \times E \times B$.

When the CPU is instructed by a load instruction to read a word from address A of main memory, it sends the address A to the cache. If the cache is holding a copy of the word at address A, it sends the word immediately back to the CPU. So how does the cache know whether it contains a copy of the word at address A? The cache is organized so that it can find the requested word by simply inspecting the bits of the address, similar to a hash table with an extremely simple hash function. Here is how it works.

The parameters S and B induce a partitioning of the m address bits into the three fields shown in Figure 6.25(b). The s *set index bits* in A form an index into the array of S sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the set index bits tell us which set the word must be stored in. Once we know which set the word must be contained in, the t tag bits in A tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A. Once we have located the line identified by the tag in the set identified by the set index, then the b *block offset bits* give us the offset of the word in the B-byte data block.

As you may have noticed, descriptions of caches use a lot of symbols. Figure 6.26 summarizes these symbols for your reference.

Fundamental parameters	
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

Derived quantities	
Parameter	Description
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

Figure 6.26: **Summary of cache parameters.**

Practice Problem 6.6:

The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1				
2.	32	1024	8	4				
3.	32	1024	32	32				

6.4.2 Direct-Mapped Caches

Caches are grouped into different classes based on E, the number of cache lines per set. A cache with exactly one line per set ($E = 1$) is known as a *direct-mapped* cache (see Figure 6.27). Direct-mapped

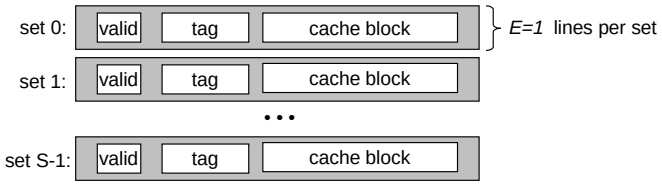


Figure 6.27: **Direct-mapped cache ($E = 1$).** There is exactly one line per set.

caches are the simplest both to implement and to understand, so we will use them to illustrate some general concepts about how caches work.

Suppose we have a system with a CPU, a register file, an L1 cache, and a main memory. When the CPU executes an instruction that reads a memory word w, it requests the word from the L1 cache. If the L1 cache has a cached copy of w, then we have an L1 cache hit, and the cache quickly extracts w and returns it to the CPU. Otherwise, we have a cache miss and the CPU must wait while the L1 cache requests a copy of the block containg w from the main memory. When the requested block finally arrives from memory, the L1 cache stores the block in one of its cache lines, extracts word w from the stored block, and returns it to the CPU. The process that a cache goes through of determining whether a request is a hit or a miss, and then extracting the requested word consists of three steps: (1) *set selection*, (2) *line matching*, and (3) *word extraction*.

Set Selection in Direct-Mapped Caches

In this step, the cache extracts the s set index bits from the middle of the address for w. These bits are interpreted as an unsigned integer that corresponds to a set number. In other words, if we think of the cache as a one-dimensional array of sets, then the set index bits form an index into this array. Figure 6.28 shows how set selection works for a direct-mapped cache. In this example, the set index bits 00001_2 are interpreted as an integer index that selects set 1.

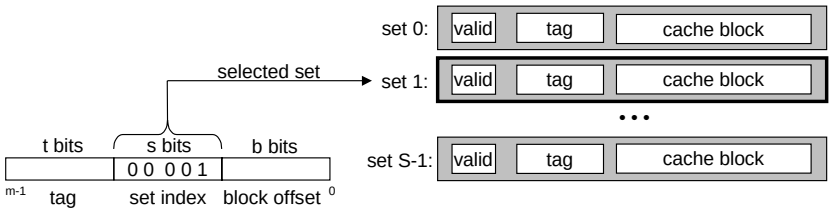


Figure 6.28: **Set selection in a direct-mapped cache.**

Line Matching in Direct-Mapped Caches

Now that we have selected some set i in the previous step, the next step is to determine if a copy of the word w is stored in one of the cache lines contained in set i. In a direct-mapped cache, this is easy and fast because there is exactly one line per set. A copy of w is contained in the line if and only if the valid bit is

set and the tag in the cache line matches the tag in the address of w.

Figure 6.29 shows how line matching works in a direct-mapped cache. In this example, there is exactly one cache line in the selected set. The valid bit for this line is set, so we know that the bits in the tag and block are meaningful. Since the tag bits in the cache line match the tag bits in the address, we know that a copy of the word we want is indeed stored in the line. In other words, we have a cache hit. On the other hand, if either the valid bit were not set or the tags did not match, then we would have had a cache miss.

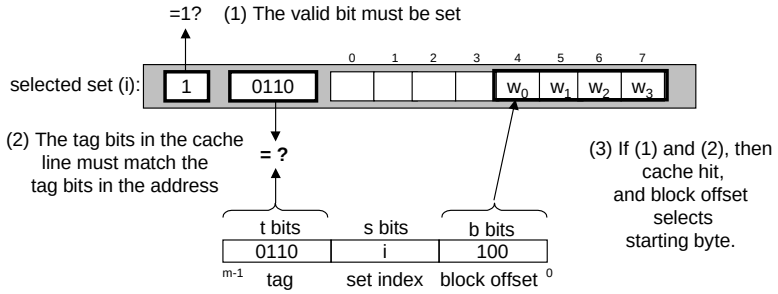


Figure 6.29: **Line matching and word selection in a direct-mapped cache.** Within the cache block, w_0 denotes the low-order byte of the word w, w_1 the next byte, and so on.

Word Selection in Direct-Mapped Caches

Once we have a hit, we know that w is somewhere in the block. This last step determines where the desired word starts in the block. As shown in Figure 6.29, the block offset bits provide us with the offset of the first byte in the desired word. Similar to our view of a cache as an array of lines, we can think of a block as an array of bytes, and the byte offset as an index into that array. In the example, the block offset bits of 100_2 indicate that the copy of w starts at byte 4 in the block. (We are assuming that words are 4 bytes long.)

Line Replacement on Misses in Direct-Mapped Caches

If the cache misses, then it needs to retrieve the requested block from the next level in the memory hierarchy and store the new block in one of the cache lines of the set indicated by the set index bits. In general, if the set is full of valid cache lines, then one of the existing lines must be evicted. For a direct-mapped cache, where each set contains exactly one line, the replacement policy is trivial: the current line is replaced by the newly fetched line.

Putting it Together: A Direct-Mapped Cache in Action

The mechanisms that a cache uses to select sets and identify lines are extremely simple. They have to be, because the hardware must perform them in only a few nanoseconds. However, manipulating bits in this way can be confusing to us humans. A concrete example will help clarify the process. Suppose we have a direct-mapped cache where

$$(S, E, B, m) = (4, 1, 2, 4)$$

In other words, the cache has four sets, one line per set, 2 bytes per block, and 4-bit addresses. We will also assume that each word is a single byte. Of course, these assumptions are totally unrealistic, but they will help us keep the example simple.

When you are first learning about caches, it can be very instructive to enumerate the entire address space and partition the bits, as we've done in Figure 6.30 for our 4-bit example. There are some interesting things

Address	Address bits			
(decimal equivalent)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30: **4-bit address for example direct-mapped cache**

to notice about this enumerated space.

- The concatenation of the tag and index bits uniquely identifies each block in memory. For example, block 0 consists of addresses 0 and 1, block 1 consists of addresses 2 and 3, block 2 consists of addresses 4 and 5, and so on.
- Since there are eight memory blocks but only four cache sets, multiple blocks map to the same cache set (i.e., they have the same set index). For example, blocks 0 and 4 both map to set 0, blocks 1 and 5 both map to set 1, and so on.
- Blocks that map to the same cache set are uniquely identified by the tag. For example, block 0 has a tag bit of 0 while block 4 has a tag bit of 1, block 1 has a tag bit of 0 while block 5 has a tag bit of 1.

Let's simulate the cache in action as the CPU performs a sequence of reads. Remember that for this example, we are assuming that the CPU reads 1-byte words. While this kind of manual simulation is tedious and you may be tempted to skip it, in our experience, students do not really understand how caches work until they work their way through a few of them.

Initially, the cache is empty (i.e., each valid bit is 0).

set	valid	tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

Each row in the table represents a cache line. The first column indicates the set that the line belongs to, but keep in mind that this is provided for convenience and is not really part of the cache. The next three columns represent the actual bits in each cache line. Now let's see what happens when the CPU performs a sequence of reads:

1. **Read word at address 0.** Since the valid bit for set 0 is zero, this is a cache miss. The cache fetches block 0 from memory (or a lower-level cache) and stores the block in set 0. Then the cache returns m[0] (the contents of memory location 0) from block[0] of the newly fetched cache line.

set	valid	tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

2. **Read word at address 1.** This is a cache hit. The cache immediately returns m[1] from block[1] of the cache line. The state of the cache does not change.

3. **Read word at address 13.** Since the cache line in set 2 is not valid, this is a cache miss. The cache loads block 6 into set 2 and returns m[13] from block[1] of the new cache line.

set	valid	tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

4. **Read word at address 8.** This is a miss. The cache line in set 0 is indeed valid, but the tags do not match. The cache loads block 4 into set 0 (replacing the line that was there from the read of address 0) and returns m[8] from block[0] of the new cache line.

set	valid	tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. **Read word at address 0.** This is another miss, due to the unfortunate fact that we just replaced block 0 during the previous reference to address 8. This kind of miss, where we have plenty of room in the cache but keep alternating references to blocks that map to the same set, is an example of a conflict miss.

set	valid	tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Conflict Misses in Direct-Mapped Caches

Conflict misses are common in real programs and can cause baffling performance problems. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of two. For example, consider a function that computes the dot product of two vectors:

```
1 float dotprod(float x[8], float y[8])
2 {
3     float sum = 0.0;
4     int i;
5
6     for (i = 0; i < 8; i++)
7         sum += x[i] * y[i];
8     return sum;
9 }
```

This function has good spatial locality with respect to `x` and `y`, and so we might expect it to enjoy a good number of cache hits. Unfortunately, this is not always true.

Suppose that floats are 4 bytes, that `x` is loaded into the 32 bytes of contiguous memory starting at address 0, and that `y` starts immediately after `x` at address 32. For simplicity, suppose that a block is 16 bytes (big enough to hold four floats) and that the cache consists of two sets, for a total cache size of 32 bytes. We will assume that the variable `sum` is actually stored in a CPU register and thus doesn't require a memory reference. Given these assumptions, each `x[i]` and `y[i]` will map to the identical cache set:

Element	Address	Set index	Element	Address	Set index
x[0]	0	0	y[0]	32	0
x[1]	4	0	y[1]	36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

At runtime, the first iteration of the loop references `x[0]`, a miss that causes the block containing `x[0]`–`x[3]` to be loaded into set 0. The next reference is to `y[0]`, another miss that causes the block containing `y[0]`–`y[3]` to be copied into set 0, overwriting the values of `x` that were copied in by the previous reference. During the next iteration, the reference to `x[1]` misses, which causes the `x[0]`–`x[3]` block to be

loaded back into set 0, overwriting the `y[0]`–`y[3]` block. So now we have a conflict miss, and in fact each subsequent reference to `x` and `y` will result in a conflict miss as we *thrash* back and forth between blocks of `x` and `y`. The term *thrashing* describes any situation where a cache is repeatedly loading and evicting the same sets of cache blocks.

The bottom line is that even though the program has good spatial locality and we have room in the cache to hold the blocks for both `x[i]` and `y[i]`, each reference results in a conflict miss because the blocks map to the same cache set. It is not unusual for this kind of thrashing to result in a slowdown by a factor of 2 or 3. And be aware that even though our example is extremely simple, the problem is real for larger and more realistic direct-mapped caches.

Luckily, thrashing is easy for programmers to fix once they recognize what is going on. One easy solution is to put B bytes of padding at the end of each array. For example, instead of defining `x` to be `float x[8]`, we define it to be `float x[12]`. Assuming `y` starts immediately after `x` in memory, we have the following mapping of array elements to sets:

Element	Address	Set index	Element	Address	Set index
x[0]	0	0	y[0]	48	1
x[1]	4	0	y[1]	52	1
x[2]	8	0	y[2]	56	1
x[3]	12	0	y[3]	60	1
x[4]	16	1	y[4]	64	0
x[5]	20	1	y[5]	68	0
x[6]	24	1	y[6]	72	0
x[7]	28	1	y[7]	76	0

With the padding at the end of `x`, `x[i]` and `y[i]` now map to different sets, which eliminates the thrashing conflict misses.

Practice Problem 6.7:

In the previous `dotprod` example, what fraction of the total references to `x` and `y` will be hits once we have padded array `x`?

Why Index With the Middle Bits?

You may be wondering why caches use the middle bits for the set index instead of the high order bits. There is a good reason why the middle bits are better. Figure 6.31 shows why.

If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set. For example, in the figure, the first four blocks map to the first cache set, the second four blocks map to the second set, and so on. If a program has good spatial locality and scans the elements of an array sequentially, then the cache can only hold a block-sized chunk of the array at any point in time. This is an inefficient use of the cache.

Contrast this with middle-bit indexing, where adjacent blocks always map to different cache lines. In this case, the cache can hold an entire C-sized chunk of the array, where C is the cache size.

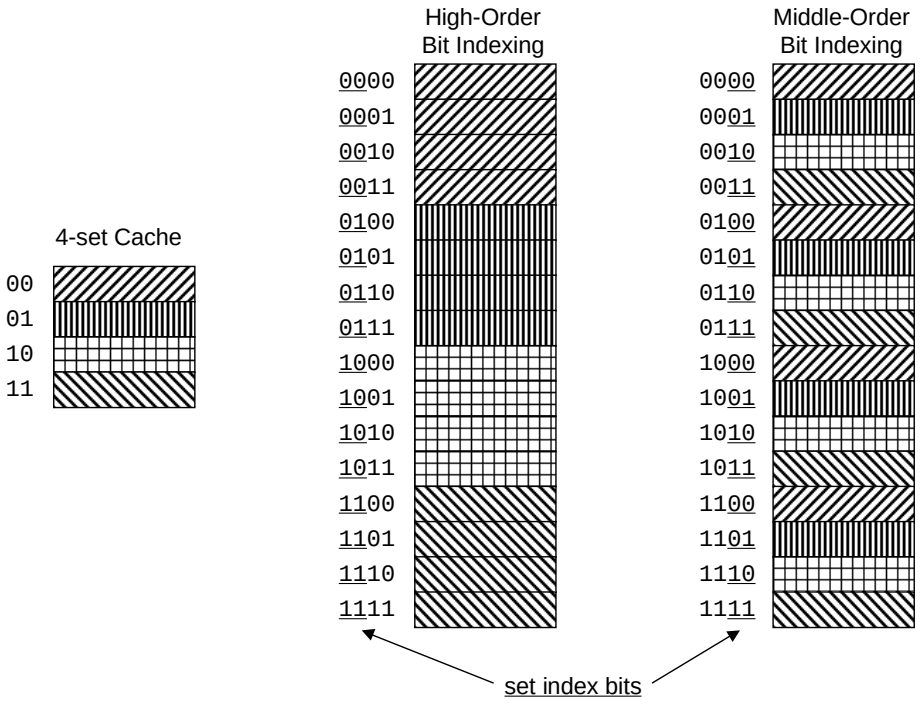


Figure 6.31: **Why caches index with the middle bits.**

Practice Problem 6.8:

In general, if the high-order s bits of an address are used as the set index, contiguous chunks of memory blocks are mapped to the same cache set.

A. How many blocks are in each of these contiguous array chunks?

B. Consider the following code that runs on a system with a cache of the form $(S, E, B, m) = (512, 1, 32, 32)$:

```
int array[4096];

for (i = 0; i < 4096; i++)
    sum += array[i];
```

What is the maximum number of array blocks that are stored in the cache at any point in time?

6.4.3 Set Associative Caches

The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$). A *set associative cache* relaxes this constraint so each set holds more than one cache line. A cache with $1 < E < C/B$ is often called an E-way set associative cache. We will discuss the special case, where $E = C/B$, in the next section. Figure 6.32 shows the organization of a two-way set associative cache.

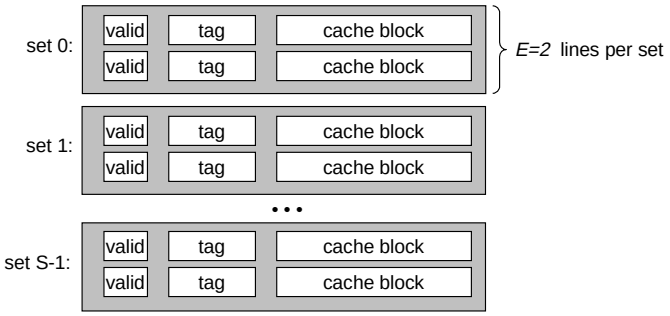


Figure 6.32: **Set associative cache (**$1 < E < C/B$**).** In a set associative cache, each set contains more than one line. This particular example shows a 2-way set associative cache.

Set Selection in Set Associative Caches

Set selection is identical to a direct-mapped cache, with the set index bits identifying the set. Figure 6.33 summarizes this.

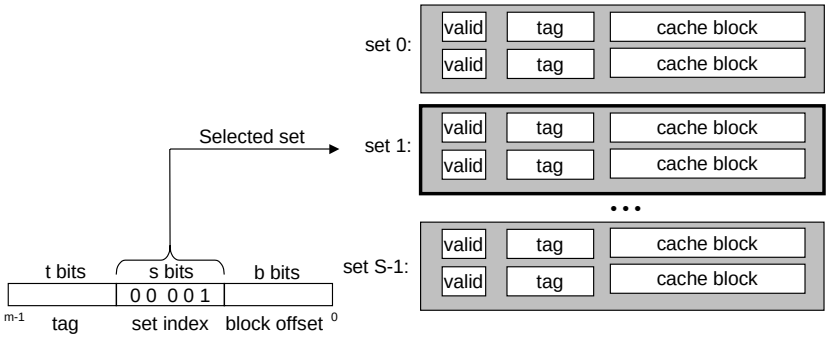


Figure 6.33: **Set selection in a set associative cache.**

Line Matching and Word Selection in Set Associative Caches

Line matching is more involved in a set associative cache than in a direct-mapped cache because it must check the tags and valid bits of multiple lines in order to determine if the requested word is in the set. A conventional memory is an array of values that takes an address as input and returns the value stored at that address. An *associative memory*, on the other hand, is an array of (key,value) pairs that takes as input the key and returns a value from one of the (key,value) pairs that matches the input key. Thus, we can think of each set in a set associative cache as a small associative memory where the keys are the concatenation of the tag and valid bits, and the values are the contents of a block.

Figure 6.34 shows the basic idea of line matching in an associative cache. An important idea here is that any line in the set can contain any of the memory blocks that map to that set. So the cache must search each line in the set, searching for a valid line whose tag matches the tag in the address. If the cache finds such a line, then we have a hit and the block offset selects a word from the block, as before.

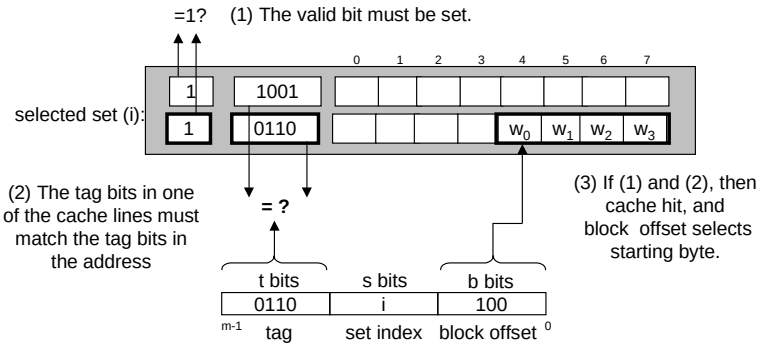


Figure 6.34: **Line matching and word selection in a set associative cache.**

Line Replacement on Misses in Set Associative Caches

If the word requested by the CPU is not stored in any of the lines in the set, then we have a cache miss, and the cache must fetch the block that contains the word from memory. However, once the cache as retrieved the block, which line should it replace? Of course, if there is an empty line, then it would be a good candidate. But if there are no empty lines in the set, then we must choose one of them and hope that the CPU doesn't reference the replaced line anytime soon.

It is very difficult for programmers to exploit knowledge of the cache replacement policy in their codes, so we will not go into much detail. The simplest replacement policy is to choose the line to replace at random. Other more sophisticated policies draw on the principle of locality to try to minimize the probability that the replaced line will be referenced in the near future. For example, a *least-frequently-used (LFU)* policy will replace the line that has been referenced the fewest times over some past time window. A *least-recently-used (LRU)* policy will replace the line that was last accessed the furthest in the past. All of these policies require additional time and hardware. But as we move further down the memory hierarchy, away from the CPU, the cost of a miss becomes more expensive and it becomes more worthwhile to minimize misses with good replacement policies.

6.4.4 Fully Associative Caches

A *fully associative cache* consists of a single set (i.e., $E = C/B$) that contains all of the cache lines. Figure 6.35 shows the basic organization.

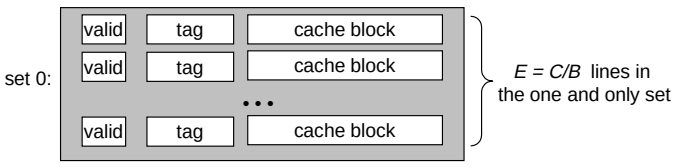


Figure 6.35: **Fully set associative cache ($E = C/B$).** In a fully associative cache, a single set contains all of the lines.

Set Selection in Fully Associative Caches

Set selection in a fully associative cache is trivial because there is only one set. Figure 6.36 summarizes. Notice that there are no set index bits in the address, which is partitioned into only a tag and a block offset.

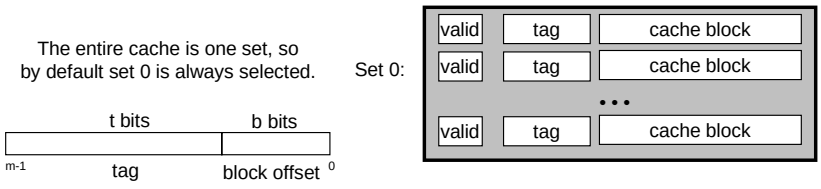


Figure 6.36: **Set selection in a fully associative cache.** Notice that there are no set index bits

Line Matching and Word Selection in Fully Associative Caches

Line matching and word selection in a fully associative cache work the same as with an associated cache, as we show in Figure 6.37. The difference is mainly a question of scale. Because the cache circuitry

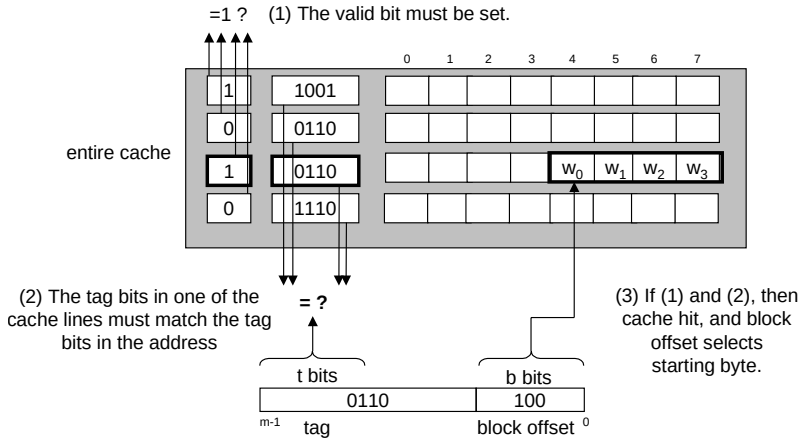


Figure 6.37: **Line matching and word selection in a fully associative cache.**

must search for many matching tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast. As a result, fully associative caches are only appropriate for small caches, such as the translation lookaside buffers (TLBs) in virtual memory systems that cache page table entries (Section 10.6.2).

Practice Problem 6.9:

The following problems will help reinforce your understanding of how caches work. Assume the following:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).

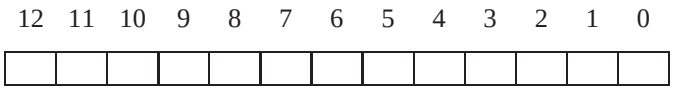
- Addresses are 13 bits wide.
- The cache is 2-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and 8 sets ($S = 8$).

The contents of the cache are as follows. All numbers are given in hexadecimal notation.

2-way Set Associative Cache												
	Line 0						Line 1					
Set Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	–	–	–	–
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	–	–	–	–	0B	0	–	–	–	–
3	06	0	–	–	–	–	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	–	–	–	–
6	91	1	A0	B7	26	2D	F0	0	–	–	–	–
7	46	0	–	–	–	–	DE	1	12	C0	88	37

The box below shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

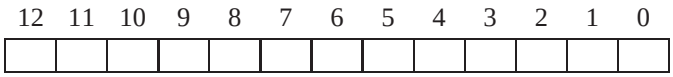
CO The cache block offset
CI The cache set index
CT The cache tag



Practice Problem 6.10:

Suppose a program running on the machine in Problem 6.9 references the 1-byte word at address `0x0E34`. Indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned".

A. Address format (one bit per box):



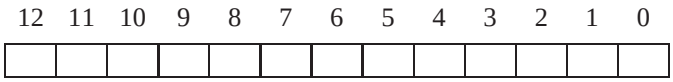
B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x`
Cache set index (CI)	`0x`
Cache tag (CT)	`0x`
Cache hit? (Y/N)	
Cache byte returned	`0x`

Practice Problem 6.11:

Repeat Problem 6.10 for memory address `0x0DD5`.

A. Address format (one bit per box):



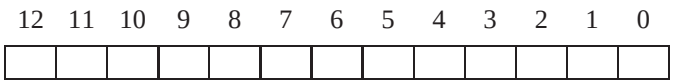
B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x`
Cache set index (CI)	`0x`
Cache tag (CT)	`0x`
Cache hit? (Y/N)	
Cache byte returned	`0x`

Practice Problem 6.12:

Repeat Problem 6.10 for memory address `0x1FE4`.

A. Address format (one bit per box):



B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x`
Cache set index (CI)	`0x`
Cache tag (CT)	`0x`
Cache hit? (Y/N)	
Cache byte returned	`0x`

Practice Problem 6.13:

For the cache in Problem 6.9, list all of the hex memory addresses that will hit in Set 3.

6.4.5 Issues with Writes

As we have seen, the operation of a cache with respect to reads is straightforward. First, look for a copy of the desired word w in the cache. If there is a hit, return word w to the CPU immediately. If there is a miss, fetch the block that contains word w from memory, store the block in some cache line (possibly evicting a valid line), and then return word w to the CPU.

The situation for writes is a little more complicated. Suppose the CPU writes a word w that is already cached (a *write hit*). After the cache updates its copy of w, what does it do about updating the copy of w in memory? The simplest approach, known as *write-through*, is to immediately write w's cache block to memory. While simple, write-through has the disadvantage of causing a write transaction on the bus with every store instruction. Another approach, known as *write-back*, defers the memory update as long as possible by writing the updated block to memory only when it is evicted from the cache by the replacement algorithm. Because of locality, write-back can significantly reduce the number of bus transactions, but it has the disadvantage of additional complexity. The cache must maintain an additional *dirty bit* for each cache line that indicates whether or not the cache block has been modified.

Another issue is how to deal with write misses. One approach, known as *write-allocate*, loads the corresponding memory block into the cache and then updates the cache block. Write-allocate tries to exploit spatial locality of writes, but has the disadvantage that every miss results in a block transfer from memory to cache. The alternative, known as *no-write-allocate*, bypasses the cache and writes the word directly to memory. Write-through caches are typically no-write-allocate. Write-back caches are typically write-allocate.

Optimizing caches for writes is a subtle and difficult issue, and we are only touching the surface here. The details vary from system to system and are often proprietary and poorly documented. To the programmer trying to write reasonably cache-friendly programs, we suggest adopting a mental model that assumes write-back write-allocate caches. There are several reasons for this suggestion.

As a rule, caches at lower levels of the memory hierarchy are more likely to use write-back instead of write-through because of the larger transfer times. For example, virtual memory systems (which use main memory as a cache for the blocks stored on disk) use write-back exclusively. But as logic densities increase, the increased complexity of write-back is becoming less of an impediment and we are seeing write-back caches at all levels of modern systems. So this assumption matches current trends. Another reason for assuming a write-back write-allocate approach is that it is symmetric to the way reads are handled, in that write-back write-allocate tries to exploit locality. Thus, we can develop our programs at a high level to exhibit good spatial and temporal locality rather than trying to optimize for a particular memory system.

6.4.6 Instruction Caches and Unified Caches

So far, we have assumed that caches hold only program data. But in fact, caches can hold instructions as well as data. A cache that holds instructions only is known as an *i-cache*. A cache that holds program data only is known as a *d-cache*. A cache that holds both instructions and data is known as a *unified cache*. A typical desktop systems includes an L1 i-cache and an L1 d-cache on the CPU chip itself, and a separate off-chip L2 unified cache. Figure 6.38 summarizes the basic setup.

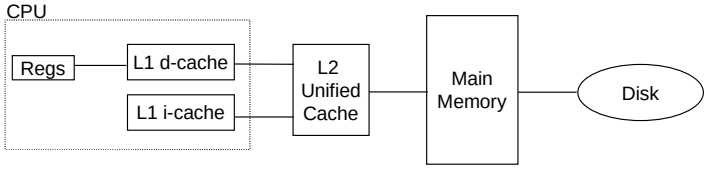


Figure 6.38: **A typical multi-level cache organization.**

Some higher-end systems, such as those based on the Alpha 21164, put the L1 and L2 caches on the CPU chip and have an additional off-chip L3 cache. Modern processors include separate on-chip i-caches and d-caches in order to improve performance. With two separate caches, the processor can read an instruction word and a data word during the same cycle. To our knowledge, no system incorporates an L4 cache, although as processor and memory speeds continue to diverge, it is likely to happen.

Aside: What kind of cache organization does a real system have?
Intel Pentium systems use the cache organization shown in Figure 6.38, with an on-chip L1 i-cache, an on-chip L1 d-cache, and an off-chip unified L2 cache. Figure 6.39 summarizes the basic parameters of these caches. **End Aside.**

Cache type	Associativity (E)	Block size (B)	Sets (S)	Cache size (C)
on-chip L1 i-cache	4	32 B	128	16 KB
on-chip L1 d-cache	4	32 B	128	16 KB
off-chip L2 unified cache	4	32 B	1024–16384	128 KB–2 MB

Figure 6.39: **Intel Pentium cache organization.**

6.4.7 Performance Impact of Cache Parameters

Cache performance is evaluated with a number of metrics:

- *Miss rate.* The fraction of memory references during the execution of a program, or a part of a program, that miss. It is computed as *# misses*/*# references*.
- *Hit rate.* The fraction of memory references that hit. It is computed as $1 - $ *miss rate*.
- *Hit time.* The time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is typically 1 to 2 clock cycle for L1 caches.
- *Miss penalty.* Any additional time required because of a miss. The penalty for L1 misses served from L2 is typically 5 to 10 cycles. The penalty for L1 misses served from main memory is typically 25 to 100 cycles.

Optimizing the cost and performance trade-offs of cache memories is a subtle exercise that requires extensive simulation on realistic benchmark codes and is beyond our scope. However, it is possible to identify some of the qualitative tradeoffs.

Impact of Cache Size

On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make big memories run faster. So larger caches tend to decrease the hit time. This is especially important for on-chip L1 caches that must have a hit time of one clock cycle.

Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems usually compromise with cache blocks that contain 4 to 8 words.

Impact of Associativity

The issue here is the impact of the choice of the parameter E, the number of cache lines per set. The advantage of higher associativity (i.e., larger values of E) is that it decreases the vulnerability of the cache to thrashing due to conflict misses. However, higher associativity comes at a significant cost. Higher associativity is expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. Higher associativity can increase hit time, because of the increased complexity, and can also increase the miss penalty because of the increased complexity of choosing a victim line.

The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for direct-mapped L1 caches (where the miss penalty is only a few cycles) and a small degree of associativity (say 2 to 4) for the lower levels. But there are no hard and fast rules. In Intel Pentium systems, the L1 and L2 caches are all four-way set associative. In Alpha 21164 systems, the L1 instruction and data caches are direct-mapped, the L2 cache is three-way set associative, and the L3 cache is direct-mapped.

Impact of Write Strategy

Write-through caches are simpler to implement and can use a *write buffer* that works independently of the cache to update memory. Furthermore, read misses are less expensive because they do not trigger a memory write. On the other hand, write-back caches result in fewer transfers, which allows more bandwidth to memory for I/O devices that perform DMA. Further, reducing the number of transfers becomes increasingly important as we move down the hierarchy and the transfer times increase. In general, caches further down the hierarchy are more likely to use write-back than write-through.

Aside: Cache lines, sets, and blocks: What's the difference?
It is easy to confuse the distinction between cache lines, sets, and blocks. Let's review these ideas and make sure they are clear:

- A *block* is a fixed sized packet of information that moves back and forth between a cache and main memory (or a lower level cache).
- A *line* is a container in a cache that stores a block, as well as other information such as the valid bit and the tag bits.
- A *set* is a collection of one or more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.

In direct-mapped caches, sets and lines are indeed equivalent. However, in associative caches, sets and lines are very different things and the terms cannot be used interchangeably.

Since a line always stores a single block, the terms "line" and "block" are often used interchangeably. For example, systems professionals usually refer to the "line size" of a cache, when what they really mean is the block size. This usage is very common, and shouldn't cause any confusion, so long as you understand the distinction between blocks and lines. **End Aside.**

6.5 Writing Cache-friendly Code

In Section 6.2 we introduced the idea of locality and talked in general terms about what constitutes good locality. But now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to write code that is *cache-friendly*, in the sense that it has good locality. Here is the basic approach we use to try to ensure that our code is cache-friendly.

1. *Make the common case go fast.* Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core functions and ignore the rest.

2. *Minimize the number of cache misses in each inner loop.* All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

To see how this works in practice, consider the `sumvec` function from Section 6.2.

```
1 int sumvec(int v[N])
2 {
3     int i, sum = 0;
4
5     for (i = 0; i < N; i++)
6         sum += v[i];
7     return sum;
8 }
```

Is this function cache-friendly? First, notice that there is good temporal locality in the loop body with respect to the local variables `i` and `sum`. In fact, because these are local variables, any reasonable optimizing compiler will cache them in the register file, the highest level of the memory hierarchy. Now consider the stride-1 references to vector `v`. In general, if a cache has a block size of B bytes, then a stride-k reference pattern (where k is in expressed in words) results in an average of $\min(1, (wordsize \times k)/B)$ misses per loop iteration. This is minimized for $k = 1$, so the stride-1 references to `v` are indeed cache-friendly. For example, suppose that `v` is block-aligned, words are 4-bytes, cache blocks are 4 words, and the cache is initially empty (a cold cache). Then regardless of the cache organization, the references to `v` will result in the following pattern of hits and misses:

v[i]	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
Access order, [h]it or [m]iss	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

In this example, the reference to `v[0]` misses and the corresponding block, which contains `v[0]`–`v[3]`, is loaded into the cache from memory. Thus, the next three references are all hits. The reference to `v[4]` causes another miss as a new block is loaded into the cache, the next three references are hits, and so on. In general, three out of four references will hit, which is the best we can do in this case with a cold cache.

To summarize, our simple `sumvec` example illustrates two important points about writing cache-friendly code:

- Repeated references to local variables are good because the compiler can cache them in the register file (temporal locality).
- Stride-1 reference patterns are good because caches at all levels of the memory hierarchy store data as contiguous blocks (spatial locality).

Spatial locality is especially important in programs that operate on multidimensional arrays. For example, consider the `sumarrayrows` function from Section 6.2 that sums the elements of a two-dimensional array in row-major order.

```
1 int sumarrayrows(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (i = 0; i < M; i++)
6         for (j = 0; j < N; j++)
7             sum += a[i][j];
8     return sum;
9 }
```

Since C stores arrays in row-major order, the inner loop of this function has the same desirable stride-1 access pattern as `sumvec`. For example, suppose we make the same assumptions about the cache as for `sumvec`. Then the references to the array `a` will result in the following pattern of hits and misses:

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
i = 1	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
i = 2	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
i = 3	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

But consider what happens if we make the seemingly innocuous change of permuting the loops:

```
1 int sumarraycols(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

In this case we are scanning the array column by column instead of row by row. If we are lucky and the entire array fits in the cache, then we will enjoy the same miss rate of 1/4. However, if the array is larger than the cache (the more likely case), then each and every access of `a[i][j]` will miss!

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
i = 1	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
i = 2	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
i = 3	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

Higher miss rates can have a significant impact on running time. For example, on our desktop machine, `sumarraycols` runs in about 20 clock cycles per iteration, while `sumarrayrows` runs in about 10 cycles per iteration. To summarize, programmers should be aware of locality in their programs and try to write programs that exploit it.

Practice Problem 6.14:

Transposing the rows and columns of a matrix is an important problem in signal processing and scientific computing applications. It is also interesting from a locality point of view because its reference pattern is both row-wise and column-wise. For example, consider the following transpose routine:

```
1 typedef int array[2][2];
2
3 void transpose1(array dst, array src)
4 {
5     int i, j;
6
7     for (i = 0; i < 2; i++) {
8         for (j = 0; j < 2; j++) {
9             dst[j][i] = src[i][j];
10        }
11    }
12 }
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4`.

- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with a block size of 8 bytes.
- The cache has a total size of 16 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`dst` array		
	col 0	col 1
row 0	m	
row 1		

`src` array		
	col 0	col 1
row 0	m	
row 1		

B. Repeat the problem for a cache with 32 data bytes.

Practice Problem 6.15:

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1024-byte direct-mapped data cache with 16-byte blocks ($B = 16$). You are given the following definitions:

```
1 struct algae_position {
2     int x;
3     int y;
4 };
5
6 struct algae_position grid[16][16];
7 int total_x = 0, total_y = 0;
8 int i, j;
```

You should also assume:

- `sizeof(int) == 4`.
- `grid` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `grid`. Variables `i`, `j`, `total_x`, and `total_y` are stored in registers.

Determine the cache performance for the following code:

```
1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             total_x += grid[i][j].x;
4         }
5     }
```

```
6
7       for (i = 0; i < 16; i++) {
8           for (j = 0; j < 16; j++) {
9               total_y += grid[i][j].y;
10          }
11      }
```

A. What is the total number of reads? _____.

B. What is the total number of reads that miss in the cache? _____ .

C. What is the miss rate? _____.

Practice Problem 6.16:

Given the assumptions of Problem 6.15, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[j][i].x;
4               total_y += grid[j][i].y;
5           }
6       }
```

A. What is the total number of reads? _____.

B. What is the total number of reads that miss in the cache? _____ .

C. What is the miss rate? _____.

D. What would the miss rate be if the cache were twice as big?

Practice Problem 6.17:

Given the assumptions of Problem 6.15, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[i][j].x;
4               total_y += grid[i][j].y;
5           }
6       }
```

A. What is the total number of reads? _____.

B. What is the total number of reads that miss in the cache? _____ .

C. What is the miss rate? _____.

D. What would the miss rate be if the cache were twice as big?

6.6 Putting it Together: The Impact of Caches on Program Performance

This section wraps up our discussion of the memory hierarchy by studying the impact that caches have on the performance of programs running on real machines.

6.6.1 The Memory Mountain

The rate that a program reads data from the memory system is called the *read throughput*, or sometimes the *read bandwidth*. If a program reads n bytes over a period of s seconds, then the read throughput over that period is n/s, typically expressed in units of MBytes per second (MB/s).

If we were to write a program that issued a sequence of read requests from a tight program loop, then the measured read throughput would give us some insight into the performance of the memory system for that particular sequence of reads. Figure 6.40 shows a pair of functions that measure the read throughput for a particular read sequence.

code/mem/mountain/mountain.c

```
1  void test(int elems, int stride) /* The test function */
2  {
3      int i, result = 0;
4      volatile int sink;
5
6      for (i = 0; i < elems; i += stride)
7          result += data[i];
8      sink = result; /* So compiler doesn't optimize away the loop */
9  }
10
11 /* Run test(elems, stride) and return read throughput (MB/s) */
12 double run(int size, int stride, double Mhz)
13 {
14     double cycles;
15     int elems = size / sizeof(int);
16
17     test(elems, stride);                    /* warm up the cache */
18     cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
19     return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
20 }
```

code/mem/mountain/mountain.c

Figure 6.40: **Functions that measure and compute read throughput.**

The `test` function generates the read sequence by scanning the first `elems` elements of an integer array with a stride of `stride`. The `run` function is a wrapper that calls the `test` function and returns the measured read throughput. The `fcyc2` function in line 18 (not shown) estimates the running time of the `test` function, in CPU cycles, using the K-best measurement scheme described in Chapter 9. Notice that the `size` argument to the `run` function is in units of bytes, while the corresponding `elems` argument to

the `test` function is in units of words. Also, notice that line 19 computes MB/s as 10^6 bytes/s, as opposed to 2^{20} bytes/s.

The `size` and `stride` arguments to the `run` function allow us to control the degree of locality in the resulting read sequence. Smaller values of `size` result in a smaller working set size, and thus more temporal locality. Smaller values of `stride` result in more spatial locality. If we call the `run` function repeatedly with different values of `size` and `stride`, then we can recover a two-dimensional function of read bandwidth versus temporal and spatial locality called the *memory mountain*. Figure 6.41 shows a program, called `mountain`, that generates the memory mountain.

code/mem/mountain/mountain.c

```
1  #include <stdio.h>
2  #include "fcyc2.h" /* K-best measurement timing routines */
3  #include "clock.h" /* routines to access the cycle counter */
4
5  #define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
6  #define MAXBYTES (1 << 23)  /* ... up to 8 MB */
7  #define MAXSTRIDE 16        /* Strides range from 1 to 16 */
8  #define MAXELEMS MAXBYTES/sizeof(int)
9
10 int data[MAXELEMS];         /* The array we'll be traversing */
11
12 int main()
13 {
14     int size;        /* Working set size (in bytes) */
15     int stride;      /* Stride (in array elements) */
16     double Mhz;      /* Clock frequency */
17
18     init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
19     Mhz = mhz(0);              /* Estimate the clock frequency */
20     for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
21         for (stride = 1; stride <= MAXSTRIDE; stride++) {
22             printf("%.1f\t", run(size, stride, Mhz));
23         }
24         printf("\n");
25     }
26     exit(0);
27 }
```

code/mem/mountain/mountain.c

Figure 6.41: `mountain`**: A program that generates the memory mountain.**

The `mountain` program calls the `run` function with different working set sizes and strides. Working set sizes start at 1 KB, increasing by a factor of two, to a maximum of 8 MB. Strides range from 1 to 16. For each combination of working set size and stride, `mountain` prints the read throughout, in units of MB/s. The `mhz` function in line 19 (not shown) is a system-dependent routine that estimates the CPU clock frequency, using techniques described in Chapter 9.

Every computer has a unique memory mountain that characterizes the capabilities of its memory system. For example, Figure 6.42 shows the memory mountain for an Intel Pentium III Xeon system.

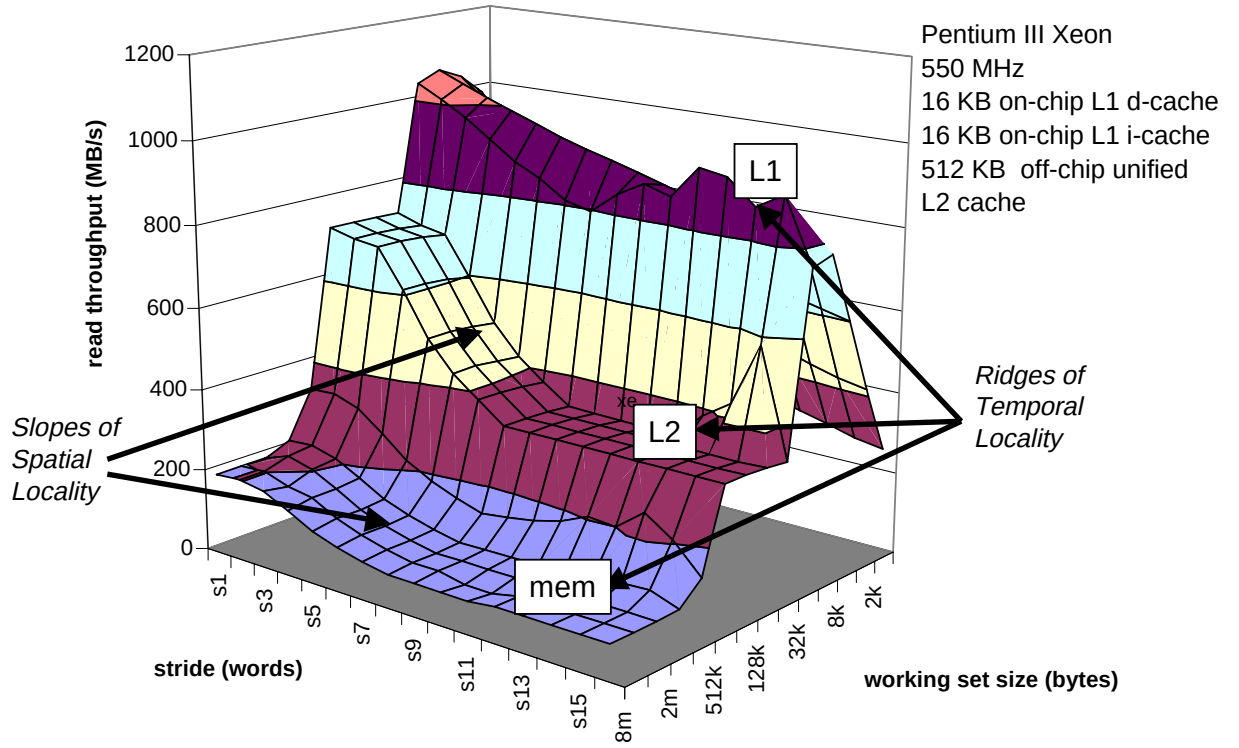


Figure 6.42: **The memory mountain.**

The geography of the Xeon mountain reveals a rich structure. Perpendicular to the `size` axis are three ridges that correspond to the regions of temporal locality where the working set fits entirely in the L1 cache, the L2 cache, and main memory respectively. Notice that there is an order of magnitude difference between the highest peak of the L1 ridge, where the CPU reads at a rate of 1 GB/s, and the lowest point of the main memory ridge, where the CPU reads at a rate of 80 MB/s.

There are two features of the L1 ridge that should be pointed out. First, for a constant stride, notice how the read throughput plummets as the working set size decreases from 16 KB to 1 KB (falling off the back side of the ridge). Second, for a working set size of 16 KB, the peak of the L1 ridge line decreases with increasing stride. Since the L1 cache holds the entire working set, these features do not reflect the true L1 cache performance. They are artifacts of overheads of calling the `test` function and setting up to execute the loop. For the small working set sizes along the L1 ridge, these overheads are not amortized, as they are with the larger working set sizes.

On the L2 and main memory ridges, there is a slope of spatial locality that falls downhill as the stride increases. This slope is steepest on the L2 ridge because of the large absolute miss penalty that the L2 cache suffers when it has to transfer blocks from main memory. Notice that even when the working set is too large to fit in either of the L1 or L2 caches, the highest point on the main memory ridge is a factor of two higher than its lowest point. So even when a program has poor temporal locality, spatial locality can still come to

the rescue and make a significant difference.

If we take a slice through the mountain, holding the stride constant as in Figure 6.43, we can see quite clearly the impact of cache size and temporal locality on performance. For sizes up to and including 16 KB, the working set fits entirely in the L1 d-cache, and thus reads are served from L1 at the peak throughput of about 1 GB/s. For sizes up to and including 256 KB, the working set fits entirely in the unified L2 cache.

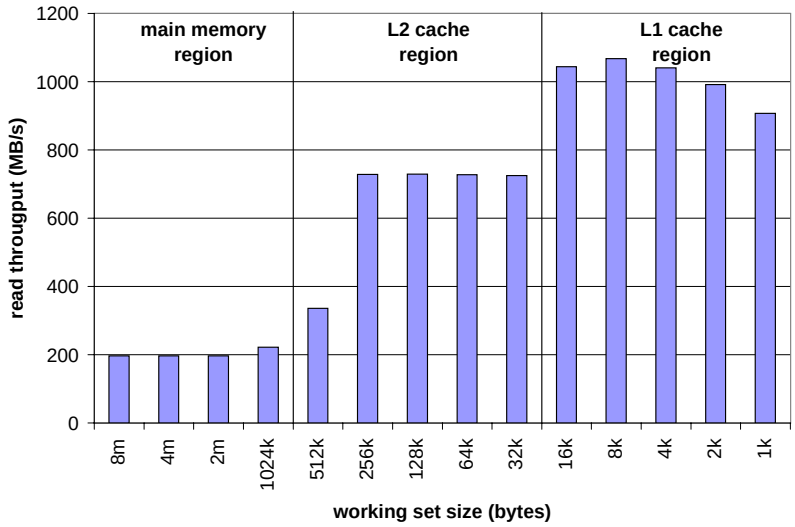


Figure 6.43: **Ridges of temporal locality in the memory mountain.** The graph shows a slice through Figure 6.42 with `stride=1`.

Larger working set sizes are served primarily from main memory. The drop in read throughput between 256 KB and 512 KB is interesting. Since the L2 cache is 512 KB, we might expect the drop to occur at 512 KB instead of 256 KB. The only way to be sure is to perform a detailed cache simulation, but we suspect the reason lies in the fact that the Pentium III L2 cache is a unified cache that holds both instructions and data. What we might be seeing is the effect of conflict misses between instructions and data in L2 that make it impossible for the entire array to fit in the L2 cache.

Slicing through the mountain in the opposite direction, holding the working set size constant, gives us some insight into the impact of spatial locality on the read throughput. For example, Figure 6.44 shows the slice for a fixed working set size of 256 KB. This slice cuts along the L2 ridge in Figure 6.42, where the working set fits entirely in the L2 cache, but is too large for the L1 cache. Notice how the read throughput decreases steadily as the stride increases from 1 to 8 words. In this region of the mountain, a read miss in L1 causes a block to be transferred from L2 to L1. This is followed by some number of hits on the block in L1, depending on the stride. As the stride increases, the ratio of L1 misses to L1 hits increases. Since misses are served slower than hits, the read throughput decreases. Once the stride reaches 8 words, which on this system equals the block size, every read request misses in L1 and must be served from L2. Thus the read throughput for strides of at least 8 words is a constant rate determined by the rate that cache blocks can be transferred from L2 into L1.

To summarize our discussion of the memory mountain: The performance of the memory system is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations

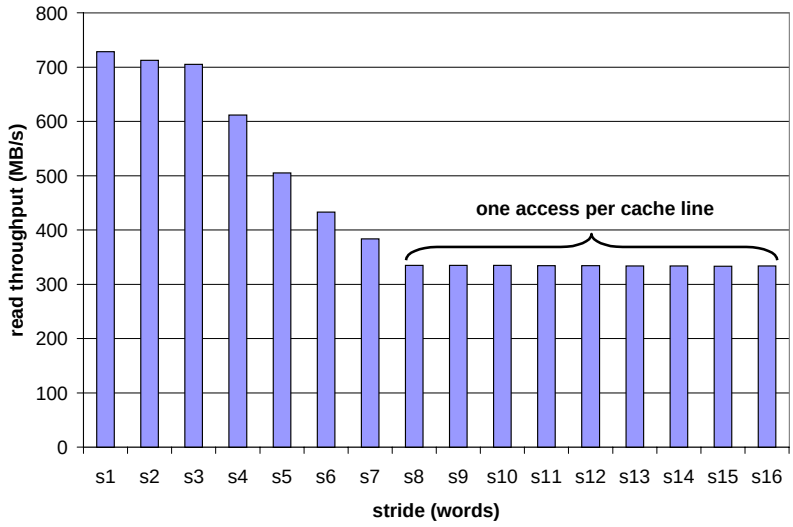


Figure 6.44: **A slope of spatial locality.** The graph shows a slice through Figure 6.42 with `size=256 KB`.

can vary by over an order of magnitude. Wise programmers try to structure their programs so that they run in the peaks instead of the valleys. The aim is to exploit temporal locality so that heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.

Practice Problem 6.18:

The memory mountain in Figure 6.42 has two axes: stride and working set size. Which axis corresponds to spatial locality? Which axis corresponds to temporal locality?

Practice Problem 6.19:

As programmers who care about performance, it is important for us to know rough estimates of the access times to different parts of the memory hierarchy. Using the memory mountain in Figure 6.42, estimate the time, in CPU cycles, to read a 4-byte word from:

A. The on-chip L1 d-cache.

B. The off-chip L2 cache.

C. Main memory.

Assume that the read throughput at (size=16M, stride=16) is 80 MB/s.

6.6.2 Rearranging Loops to Increase Spatial Locality

Consider the problem of multiplying a pair of $n \times n$ matrices: $C = AB$. For example, if $n = 2$, then

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

where

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

Matrix multiply is usually implemented using three nested loops, which are identified by their indexes i, j, and k. If we permute the loops and make some other minor code changes, we can create the six functionally equivalent versions of matrix multiply shown in Figure 6.45. Each version is uniquely identified by the ordering of its loops.

At a high level, the six versions are quite similar. If addition is associative, then each version computes an identical result.[2] Each version performs $O(n^3)$ total operations and an identical number of adds and multiplies. Each of the n^2 elements of A and B is read n times. Each of the n^2 elements of C is computed by summing n values. However, if we analyze the behavior of the innermost loop iterations, we find that there are differences in the number of accesses and the locality. For the purposes of our analysis, let's make the following assumptions:

- Each array is an $n \times n$ array of `double`, with `sizeof(double) == 8`.
- There is a single cache with a 32-byte block size ($B = 32$).
- The array size n is so large that a single matrix row does not fit in the L1 cache.
- The compiler stores local variables in registers, and thus references to local variables do not require any load or store instructions.

Figure 6.46 summarizes the results of our inner loop analysis. Notice that the six versions pair up into three equivalence classes, which we denote by the pair of matrices that are accessed in the inner loop. For example, versions ijk and jik are members of Class AB because they reference arrays A and B (but not C) in their innermost loop. For each class, we have counted the number of loads (reads) and stores (writes) in each inner loop iteration, the number of references to A, B, and C that will miss in the cache in each loop iteration, and the total number of cache misses per iteration.

The inner loops of the Class AB routines (Figure 6.45(a) and (b)) scan a row of array A with a stride of 1. Since each cache block holds four doublewords, the miss rate for A is 0.25 misses per iteration. On the other hand, the inner loop scans a column of B with a stride of n. Since n is large, each access of array B results in a miss, for a total of 1.25 misses per iteration.

The inner loops in the Class AC routines (Figure 6.45(c) and (d)) have some problems. Each iteration performs two loads and a store (as opposed to the Class AB routines, which perform 2 loads and no stores). Second, the inner loop scans the columns of A and C with a stride of n. The result is a miss on each load, for

[2] As we learned in Chapter 2, floating-point addition is commutative, but in general not associative. In practice, if the matrices do not mix extremely large values with extremely small ones, as if often true when the matrices store physical properties, then the assumption of associativity is reasonable.

code/mem/matmult/mm.c

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++) {
3         sum = 0.0;
4         for (k = 0; k < n; k++)
5             sum += A[i][k]*B[k][j];
6         C[i][j] += sum;
7     }
```

code/mem/matmult/mm.c

(a) Version ijk.

code/mem/matmult/mm.c

```
1 for (j = 0; j < n; j++)
2     for (i = 0; i < n; i++) {
3         sum = 0.0;
4         for (k = 0; k < n; k++)
5             sum += A[i][k]*B[k][j];
6         C[i][j] += sum;
7     }
```

code/mem/matmult/mm.c

(b) Version jik.

code/mem/matmult/mm.c

```
1 for (j = 0; j < n; j++)
2     for (k = 0; k < n; k++) {
3         r = B[k][j];
4         for (i = 0; i < n; i++)
5             C[i][j] += A[i][k]*r;
6     }
```

code/mem/matmult/mm.c

(c) Version jki.

code/mem/matmult/mm.c

```
1 for (k = 0; k < n; k++)
2     for (j = 0; j < n; j++) {
3         r = B[k][j];
4         for (i = 0; i < n; i++)
5             C[i][j] += A[i][k]*r;
6     }
```

code/mem/matmult/mm.c

(d) Version kji.

code/mem/matmult/mm.c

```
1 for (k = 0; k < n; k++)
2     for (i = 0; i < n; i++) {
3         r = A[i][k];
4         for (j = 0; j < n; j++)
5             C[i][j] += r*B[k][j];
6     }
```

code/mem/matmult/mm.c

(e) Version kij.

code/mem/matmult/mm.c

```
1 for (i = 0; i < n; i++)
2     for (k = 0; k < n; k++) {
3         r = A[i][k];
4         for (j = 0; j < n; j++)
5             C[i][j] += r*B[k][j];
6     }
```

code/mem/matmult/mm.c

(f) Version ikj.

Figure 6.45: **Six versions of matrix multiply.**

Matrix multiply version (class)	Loads per iter	Stores per iter	A misses per iter	B misses per iter	C misses per iter	Total misses per iter
ijk & jik (AB)	2	0	0.25	1.00	0.00	1.25
jki & kji (AC)	2	1	1.00	0.00	1.00	2.00
kij & ikj (BC)	2	1	0.00	0.25	0.25	0.50

Figure 6.46: **Analysis of matrix multiply inner loops.** The six versions partition into three equivalence classes, denoted by the pair of arrays that are accessed in the inner loop.

a total of two misses per iteration. Notice that interchanging the loops has decreased the amount of spatial locality compared to the Class AB routines.

The BC routines (Figure 6.45(e) and (f)) present an interesting tradeoff. With two loads and a store, they require one more memory operation than the AB routines. On the other hand, since the inner loop scans both B and C row-wise with a stride-1 access pattern, the miss rate on each array is only 0.25 misses per iteration, for a total of 0.50 misses per iteration.

Figure 6.47 summarizes the performance of different versions of matrix multiply on a Pentium III Xeon system. The graph plots the measured number of CPU cycles per inner loop iteration as a function of array size (n).

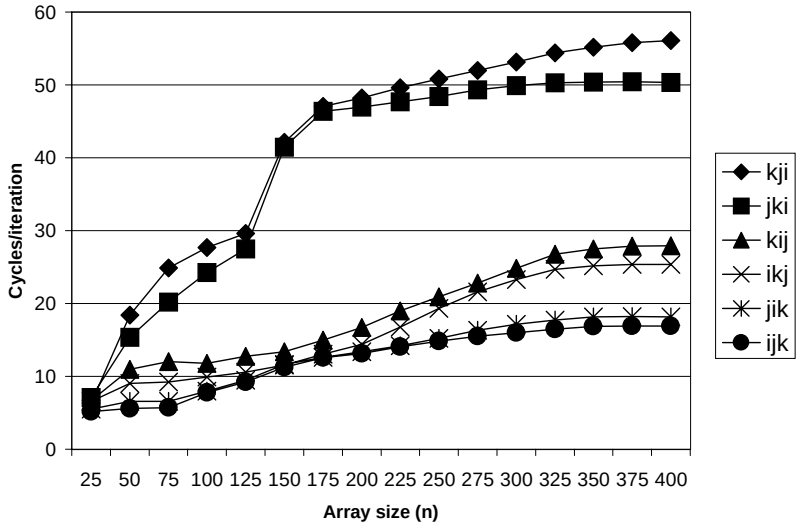


Figure 6.47: **Pentium III Xeon matrix multiply performance.** Legend: kji and jki: Class AC; kij and ikj: Class BC; ijk and jik: Class AB

There are a number of interesting points to notice about this graph:

- For large n, the fastest version runs three times faster than the slowest version, even though each performs the same number of floating-point arithmetic operations.
- Versions with the same number and locality of memory accesses have roughly the same measured performance.
- The two versions with the worst memory behavior, in terms of the number of accesses and misses per iteration, run significantly slower than the other four versions, which have fewer misses or fewer accesses, or both.
- The Class AB routines — 2 memory accesses and 1.25 misses per iteration — perform somewhat better on this particular machine than the Class BC routines — 3 memory accesses and 0.5 misses per iteration — which trade off an additional memory reference for a lower miss rate. The point is that cache misses are not the whole story when it comes to performance. The number of memory accesses

is also important, and in many cases, finding the best performance involves a tradeoff between the two. Problems 6.32 and 6.33 delve into this issue more deeply.

6.6.3 Using Blocking to Increase Temporal Locality

In the last section we saw how some simple rearrangements of the loops could increase spatial locality. But observe that even with good loop nestings, the time per loop iteration increases with increasing array size. What is happening is that as the array size increases, the temporal locality decreases, and the cache experiences an increasing number of capacity misses. To fix this, we can use a general technique called *blocking*. However, we must point out that, unlike the simple loop transformations for improving spatial locality, blocking makes the code harder to read and understand. For this reason it is best suited for optimizing compilers or frequently executed library routines. Still, the technique is interesting to study and understand because it is a general concept that can produce big performance gains.

The general idea of blocking is to organize the data structures in a program into large chunks called blocks. (In this context, the term "block" refers to an application-level chunk of data, *not* a cache block.) The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on.

Blocking a matrix multiply routine works by partitioning the matrices into submatrices and then exploiting the mathematical fact that these submatrices can be manipulated just like scalars. For example, if $n = 8$, then we could partition each matrix into four 4×4 submatrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Figure 6.48 shows one version of blocked matrix multiplication, which we call the $bijk$ version. The basic idea behind this code is to partition A and C into $1 \times$ *bsize row slivers* and to partition B into *bsize* $\times$ *bsize* blocks. The innermost (j, k) loop pair multiplies a sliver of A by a block of B and accumulates the result into a sliver of C. The i loop iterates through n row slivers of A and C, using the same block in B.

Figure 6.49 gives a graphical interpretation of the blocked code from Figure 6.48. The key idea is that it loads a block of B into the cache, uses it up, and then discards it. References to A enjoy good spatial locality because each sliver is accessed with a stride of 1. There is also good temporal locality because the entire sliver is referenced $bsize$ times in succession. References to B enjoy good temporal locality because the entire $bsize \times bsize$ block is accessed n times in succession. Finally, the references to C have good spatial locality because each element of the sliver is written in succession. Notice that references to C do not have good temporal locality because each sliver is only accessed one time.

code/mem/matmult/bmm.c

```
1 void bijk(array A, array B, array C, int n, int bsize)
2 {
3     int i, j, k, kk, jj;
4     double sum;
5     int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */
6
7     for (i = 0; i < n; i++)
8         for (j = 0; j < n; j++)
9             C[i][j] = 0.0;
10
11    for (kk = 0; kk < en; kk += bsize) {
12        for (jj = 0; jj < en; jj += bsize) {
13            for (i = 0; i < n; i++) {
14                for (j = jj; j < jj + bsize; j++) {
15                    sum = C[i][j];
16                    for (k = kk; k < kk + bsize; k++) {
17                        sum += A[i][k]*B[k][j];
18                    }
19                    C[i][j] = sum;
20                }
21            }
22        }
23    }
24 }
```

code/mem/matmult/bmm.c

Figure 6.48: **Blocked matrix multiply.** A simple version that assumes that the array size (`n`) is an integral multiple of the block size (`bsize`).

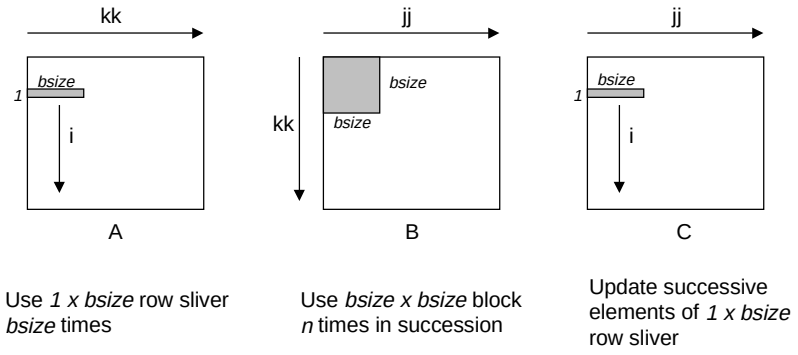


Figure 6.49: **Graphical interpretation of blocked matrix multiply** The innermost (j, k) loop pair multiplies a $1 \times$ *bsize* sliver of `A` by a *bsize* $\times$ *bsize* block of `B` and accumulates into a $1 \times$ *bsize* sliver of `C`.

Blocking can make code harder to read, but it can also pay big performance dividends. Figure 6.50 shows the performance of two versions of blocked matrix multiply on a Pentium III Xeon system ($bsize = 25$). Notice that blocking improves the running time by a factor of two over the best non-blocked version, from about 20 cycles per iteration down to about 10 cycles per iteration. The other interesting impact of blocking is that the time per iteration remains nearly constant with increasing array size. For small array sizes, the additional overhead in the blocked version causes it to run slower than the non-blocked versions. There is a crossover point, at about $n = 100$, after which the blocked version runs faster.

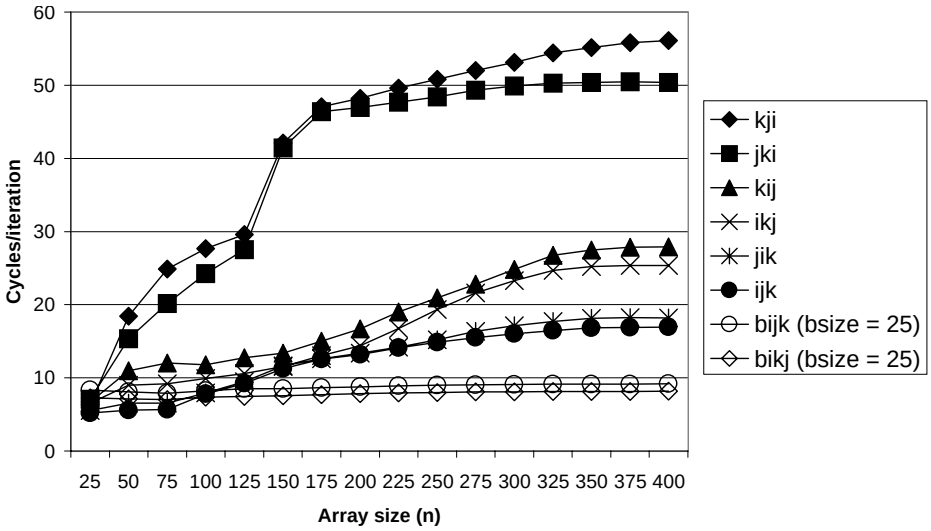


Figure 6.50: **Pentium III Xeon blocked matrix multiply performance.** Legend: $bijk$ and $bikj$: two different versions of blocked matrix multiply. Performance of the unblocked versions from Figure 6.47 is shown for reference.

Aside: Caches and streaming media workloads

Applications that process network video and audio data in real time are becoming increasingly important. In these applications, the data arrive at the machine in a steady stream from some input device such as a microphone, a camera, or a network connection (see Chapter 12). As the data arrive, they are processed, sent to an output device, and eventually discarded to make room for newly arriving data.

How well suited is the memory hierarchy for these *streaming media* workloads? Since the data are processed sequentially as they arrive, we able to derive some benefit from spatial locality, as with our matrix multiply example from Section 6.6. However, since the data are processed once and then discarded, the amount of temporal locality is limited.

To address this problem, system designers and compiler writers have pursued a strategy known as *prefetching*. The idea is to hide the latency of cache misses by anticipating which blocks will be accessed in the near future, and then fetching these blocks into the cache beforehand using special machine instructions. If the prefetching is done perfectly, then each block is copied into the cache just before the program references it, and thus every load instruction results in a cache hit. Prefetching entails risks, though. Since prefetching traffic shares the bus with the DMA traffic that is streaming from an I/O device to main memory, too much prefetching might interfere with the DMA traffic and slow down overall system performance. Another potential problem is that every prefetched cache block must evict an existing block. If we do too much prefetching, we run the risk of *polluting the cache* by evicting a previously prefetched block that the program has not referenced yet, but will in the near future. **End Aside.**

6.7 Summary

The memory system is organized as a hierarchy of storage devices, with smaller, faster devices towards the top and larger, slower devices towards the bottom. Because of this hierarchy, the effective rate that a program can access memory locations is not characterized by a single number. Rather, it is a wildly varying function of program locality (what we have dubbed the memory mountain) that can vary by orders of magnitude. Programs with good locality access most of their data from fast L1 and L2 cache memories. Programs with poor locality access most of their data from the relatively slow DRAM main memory.

Programmers who understand the nature of the memory hierarchy can exploit this understanding to write more efficient programs, regardless of the specific memory system organization. In particular, we recommend the following techniques:

- Focus your attention on the inner loops where the bulk of the computations and memory accesses occur.
- Try to maximize the spatial locality in your programs by reading data objects sequentially, in the order they are stored in memory.
- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.
- Remember that miss rates are only one (albeit important) factor that determines the performance of your code. The number of memory accesses also plays an important role, and sometimes it is necessary to trade off between the two.

Bibliographic Notes

Memory and disk technologies change rapidly. In our experience, the best sources of technical information are the Web pages maintained by the manufacturers. Companies such as Micron, Toshiba, Hyundai, Samsung, Hitachi, and Kingston Technology provide a wealth of current technical information on memory devices. The pages for IBM, Maxtor, and Seagate provide similarly useful information about disks.

Textbooks on circuit and logic design provide detailed information about memory technology [36, 58]. IEEE Spectrum published a series of survey articles on DRAM [33]. The International Symposium on Computer Architecture (ISCA) is a common forum for characterizations of DRAM memory performance [20, 21].

Wilkes wrote the first paper on cache memories [83]. Smith wrote a classic survey [68]. Przybylski wrote an authoritative book on cache design [56]. Hennessy and Patterson provide a comprehensive discussion of cache design issues [31].

Stricker introduced the idea of the memory mountain as a comprehensive characterization of the memory system in [78], and suggested the term “memory mountain” in later presentations of the work. Compiler researchers work to increase locality by automatically performing the kinds manual code transformations we discussed in Section 6.6 [13, 23, 42, 45, 51, 57, 85]. Carter and colleagues have proposed a cache-aware memory controller [10]. Seward developed an open-source cache profiler, called `cacheprof`, that characterizes the miss behavior of C programs on an arbitrary simulated cache (`www.cacheprof.org`).

There is a large body of literature on building and using disk storage. Many storage researchers look for ways to aggregate individual disks into larger, more robust, and more secure storage pools [11, 26, 27, 54, 86]. Others look for ways to use caches and locality to improve the performance of disk accesses [6, 12]. Systems such as Exokernel provide increased user-level control of disk and memory resources [35]. Systems such as the Andrew File System [50] and Coda [63] extend the memory hierarchy across computer networks and mobile notebook computers. Schindler and Ganger have developed an interesting tool that automatically characterizes the geometry and performance of SCSI disk drives [64].

Homework Problems

Homework Problem 6.20 [Category 2]:

Suppose you are asked to design a diskette where the number of bits per track is constant. You know that the number of bits per track is determined by the circumference of the innermost track, which you can assume is also the circumference of the hole. Thus, if you make the hole in the center of the diskette larger, the number of bits per track increases, but the total number of tracks decreases. If you let r denote the radius of the platter, and $x \cdot r$ the radius of the hole, what value of x maximizes the capacity of the diskette?

Homework Problem 6.21 [Category 1]:

The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

Homework Problem 6.22 [Category 1]:

This problem concerns the cache in Problem 6.9.

A. List all of the hex memory addresses that will hit in Set 1.

B. List all of the hex memory addresses that will hit in Set 6.

Homework Problem 6.23 [Category 2]:

Consider the following matrix transpose routine:

```
1 typedef int array[4][4];
2
```

```
 3 void transpose2(array dst, array src)
 4 {
 5     int i, j;
 6
 7     for (i = 0; i < 4; i++) {
 8         for (j = 0; j < 4; j++) {
 9             dst[j][i] = src[i][j];
10         }
11     }
12 }
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4`.
- The *src* array starts at address 0 and the *dst* array starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate,with a block size of 16 bytes.
- The cache has a total size of 32 data bytes and the cache is initially empty.
- Accesses to the *src* and *dst* arrays are the only sources of read and write misses, respectively.

A. For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

dst array				
	col 0	col 1	col 2	col 3
row 0	m			
row 1				
row 2				
row 3				

src array				
	col 0	col 1	col 2	col 3
row 0	m			
row 1				
row 2				
row 3				

Homework Problem 6.24 [Category 2]:

Repeat Problem 6.23 for a cache with a total size of 128 data bytes.

dst array				
	col 0	col 1	col 2	col 3
row 0				
row 1				
row 2				
row 3				

src array				
	col 0	col 1	col 2	col 3
row 0				
row 1				
row 2				
row 3				

Homework Problem 6.25 [Category 1]:

3M decides to make Post-Its by printing yellow squares on white pieces of paper. As part of the printing process, they need to set the CMYK (cyan, magenta, yellow, black) value for every point in the square. 3M hires you to determine the efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks. You are given the following definitions:

```
1 struct point_color {
2     int c;
3     int m;
4     int y;
5     int k;
6 };
7
8 struct point_color square[16][16];
9 int i, j;
```

Assume:

- `sizeof(int) == 4`.
- `square` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `square`. Variables `i` and `j` are stored in registers.

Determine the cache performance of the following code:

```
1     for (i = 0; i < 16; i++){
2         for (j = 0; j < 16; j++) {
3             square[i][j].c = 0;
4             square[i][j].m = 0;
5             square[i][j].y = 1;
6             square[i][j].k = 0;
7         }
8     }
```

A. What is the total number of writes? _____.

B. What is the total number of writes that miss in the cache? _____ .

C. What is the miss rate? _____.

Homework Problem 6.26 [Category 1]:

Given the assumptions in Problem 6.25, determine the cache performance of the following code:

```
1     for (i = 0; i < 16; i++){
2         for (j = 0; j < 16; j++) {
3             square[j][i].c = 0;
4             square[j][i].m = 0;
5             square[j][i].y = 1;
6             square[j][i].k = 0;
7         }
8     }
```

A. What is the total number of writes? _____.

B. What is the total number of writes that miss in the cache? _____ .

C. What is the miss rate? _____.

Homework Problem 6.27 [Category 1]:

Given the assumptions in Problem 6.25, determine the cache performance of the following code:

```
1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             square[i][j].y = 1;
4         }
5     }
6     for (i = 0; i < 16; i++) {
7         for (j = 0; j < 16; j++) {
8             square[i][j].c = 0;
9             square[i][j].m = 0;
10            square[i][j].k = 0;
11        }
12    }
```

A. What is the total number of writes? _____.

B. What is the total number of writes that miss in the cache? _____ .

C. What is the miss rate? _____.

Homework Problem 6.28 [Category 2]:

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640×480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines. The C structures you are using are:

```
1 struct pixel {
2     char r;
3     char g;
4     char b;
5     char a;
6 };
7
8 struct pixel buffer[480][640];
9 int i, j;
10 char *cptr;
11 int *iptr;
```

Assume:

- `sizeof(char) == 1` and `sizeof(int) == 4`
- `buffer` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

What percentage of writes in the following code will miss in the cache?

```
1     for (j = 0; j < 640; j++) {
2         for (i = 0; i < 480; i++){
3             buffer[i][j].r = 0;
4             buffer[i][j].g = 0;
5             buffer[i][j].b = 0;
6             buffer[i][j].a = 0;
7         }
8     }
```

Homework Problem 6.29 [Category 2]:

Given the assumptions in Problem 6.28, what percentage of writes in the following code will miss in the cache?

```
1     char *cptr = (char *) buffer;
2     for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
3         *cptr = 0;
```

Homework Problem 6.30 [Category 2]:

Given the assumptions in Problem 6.28, what percentage of writes in the following code will miss in the cache?

```
    int *iptr = (int *)buffer;
    for (; iptr < ((int *)buffer + 640*480); iptr++)
        *iptr = 0;
```

Homework Problem 6.31 [Category 3]:

Download the `mountain` program from the CS:APP Web site and run it on your favorite PC/Linux system. Use the results to estimate the sizes of the L1 and L2 caches on your system.

Homework Problem 6.32 [Category 4]:

In this assignment you will apply the concepts you learned in Chapters 5 and 6 to the problem of optimizing code for a memory intensive application. Consider a procedure to copy and transpose the elements of an $N \times N$ matrix of type `int`. That is, for source matrix S and destination matrix D, we want to copy each element $s_{i,j}$ to $d_{j,i}$. This code can be written with a simple loop:

```
void transpose(int *dst, int *src, int dim)
{
    int i, j;

    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[j*dim + i] = src[i*dim + j];
}
```

where the arguments to the procedure are pointers to the destination (`dst`) and source (`src`) matrices, as well as the matrix size N (`dim`). Making this code run fast requires two types of optimizations. First, although the routine does a good job exploiting the spatial locality of the source matrix, it does a poor job for large values of N with the destination matrix. Second, the code generated by GCC is not very efficient. Looking at the assembly code, one sees that the inner loop requires 10 instructions, 5 of which reference memory—one for the source, one for the destination, and three to read local variables from the stack. Your job is to address these problems and devise a transpose routine that runs as fast as possible.

Homework Problem 6.33 [Category 4]:

This assignment is an intriguing variation of Problem 6.32. Consider the problem of converting a directed graph g into its undirected counterpart g'. The graph g' has an edge from vertex u to vertex v iff there is an edge from u to v or from v to u in the original graph g. The graph g is represented by its *adjacency matrix* G as follows. If N is the number of vertices in g then G is an $N \times N$ matrix and its entries are all either 0 or 1. Suppose the vertices of g are named $v_0, v_1, v_2, ..., v_{N-1}$. Then $G[i][j]$ is 1 if there is an edge from v_i to v_j and 0 otherwise. Observe, that the elements on the diagonal of an adjacency matrix are always 1 and that the adjacency matrix of an undirected graph is symmetric. This code can be written with a simple loop:

```
void col_convert(int *G, int dim) {
     int i, j;

     for (i = 0; i < dim; i++)
         for (j = 0; j < dim; j++)
             G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
}
```

Your job is to devise a conversion routine that runs as fast as possible. As before, you will need to apply concepts you learned in Chapters 5 and 6 to come up with a good solution.

Part II

Running Programs on a System

Chapter 7

Linking

Linking is the process of collecting and combining the various pieces of code and data that a program needs in order to be *loaded* (copied) into memory and executed. Linking can be performed at *compile time*, when the source code is translated into machine code, at *load time*, when the program is loaded into memory and executed by the *loader*, and even at *run time*, by application programs. On early computer systems, linking was performed manually. On modern systems, linking is performed automatically by programs called *linkers*.

Linkers play a crucial role in software development because they enable *separate compilation*. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files.

Linking is usually handled quietly by the linker, and is not an important issue for students who are building small programs in introductory programming classes. So why bother learning about linking?

- *Understanding linkers will help you build large programs.* Programmers who build large programs often encounter linker errors caused by missing modules, missing libraries, or incompatible library versions. Unless you understand how a linker resolves references, what a library is, and how a linker uses a library to resolve references, these kinds of errors will be baffling and frustrating.
- *Understanding linkers will help you avoid dangerous programming errors.* The decisions that Unix linkers make when they resolve symbol references can silently affect the correctness of your programs. Programs that incorrectly define multiple global variables pass through the linker without any warnings in the default case. The resulting programs can exhibit baffling run-time behavior and are extremely difficult to debug. We will show you how this happens and how to avoid it.
- *Understanding linking will help you understand how language scoping rules are implemented.* For example, what is the difference between global and local variables? What does it really mean when you define a variable or function with the `static` attribute?
- *Understanding linking will help you understand other important systems concepts.* The executable object files produced by linkers play key roles in important systems functions such as loading and running programs, virtual memory, paging, and memory mapping.

- *Understanding linking will enable you to exploit shared libraries.* For many years, linking was considered to be fairly straightforward and uninteresting. However, with the increased importance of shared libraries and dynamic linking in modern operating systems, linking is a sophisticated process that provides the knowledgeable programmer with significant power. For example, many software products use shared libraries to upgrade shrink-wrapped binaries at run time. Also, most Web servers rely on dynamic linking of shared libraries to serve dynamic content.

This chapter is a thorough discussion of all aspects of linking, from traditional static linking, to dynamic linking of shared libraries at load time, to dynamic linking of shared libraries at run time. We will describe the basic mechanisms using real examples, and we will identify situations where linking issues can affect the performance and correctness of your programs. To keep things concrete and understandable, we will couch our discussion in the context of an IA32 machine running a version of Unix, such as

Linux or Solaris, that uses the standard ELF object file format. However, it is important to realize that the basic concepts of linking are universal, regardless of the operating system, the ISA, or the object file format. Details may vary, but the concepts are the same.

7.1 Compiler Drivers

Consider the C program in Figure 7.1. It consists of two source files, `main.c` and `swap.c`. Function `main()` calls `swap`, which swaps the two elements in the external global array `buf`. Granted, this is a strange way to swap two numbers, but it will serve as a small running example throughout this chapter that will allow us to make some important points about how linking works.

Most compilation systems provide a *compiler driver* that invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user. For example, to build the example program using the GNU compilation system, we might invoke the GCC driver by typing the following command to the shell:

```
unix> gcc -O2 -g -o p main.c swap.c
```

Figure 7.2 summarizes the activities of the driver as it translates the example program from an ASCII source file into an executable object file. (If you want to see these steps for yourself, run GCC with the `-v` option.) The driver first runs the C preprocessor (`cpp`), which translates the C source file `main.c` into an ASCII intermediate file `main.i`:

```
cpp [other arguments] main.c /tmp/main.i
```

Next, the driver runs the C compiler (`cc1`), which translates `main.i` into an ASCII assembly language file `main.s`.

```
cc1 /tmp/main.i main.c -O2 [other arguments] -o /tmp/main.s
```

Then, the driver runs the assembler (`as`), which translates `main.s` into a *relocatable object file* `main.o`:

```
as [other arguments] -o /tmp/main.o /tmp/main.s
```

code/link/main.c

```
1 /* main.c */
2 void swap();
3
4 int buf[2] = {1, 2};
5
6 int main()
7 {
8     swap();
9     return 0;
10 }
```

code/link/main.c

(a) `main.c`

code/link/swap.c

```
1 /* swap.c */
2 extern int buf[];
3
4 int *bufp0 = &buf[0];
5 int *bufp1;
6
7 void swap()
8 {
9     int temp;
10
11     bufp1 = &buf[1];
12     temp = *bufp0;
13     *bufp0 = *bufp1;
14     *bufp1 = temp;
15 }
```

code/link/swap.c

(b) `swap.c`

Figure 7.1: **Example program 1:** The example program consists of two source files, `main.c` and `swap.c`. The `main` function initializes a two-element array of ints, and then calls the `swap` function to swap the pair.

The driver goes through the same process to generate `swap.o`. Finally it runs the linker program `ld`, which combines `main.o` and `swap.o`, along with the necessary system object files, to create the *executable object file* `p`:

```
ld -o p [system object files and args] /tmp/main.o /tmp/swap.o
```

To run the executable `p`, we type its name on the Unix shell's command line:

```
unix> ./p
```

The shell invokes a function in the operating system called the *loader*, which copies the code and data in the executable file `p` into memory, and then transfers control to the beginning of the program.

7.2 Static Linking

Static linkers such as the Unix `ld` program take as input a collection of relocatable object files and command line arguments and generate as output a fully linked executable object file that can be loaded and run. The input relocatable object files consist of various code and data sections. Instructions are in one section, initialized global variables are in another section, and uninitialized variables are in yet another section.

To build the executable, the linker must perform two main tasks:

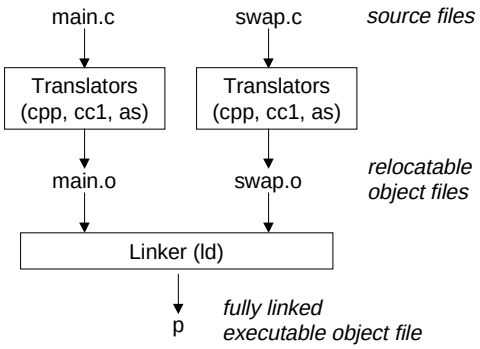


Figure 7.2: **Static linking.** The linker combines relocatable object files to form an executable object file p.

- *Symbol resolution.* Object files define and reference *symbols*. The purpose of symbol resolution is to associate each symbol reference with exactly one symbol definition.
- *Relocation.* Compilers and assemblers generate code and data sections that start at address zero. The linker *relocates* these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location.

The following sections describe these tasks in more detail. As you read, keep in mind the basic facts of linkers: Object files are merely collections of blocks of bytes. Some of these blocks contain program code, others contain program data, and others contain data structures that guide the linker and loader. A linker concatenates blocks together, decides on run-time locations for the concatenated blocks, and modifies various locations within the code and data blocks. Linkers have minimal understanding of the target machine. The compilers and assemblers that generate the object files have already done most of the work.

7.3 Object Files

Object files come in three forms:

- *Relocatable object file.* Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.
- *Executable object file.* Contains binary code and data in a form that can be copied directly into memory and executed.
- *Shared object file.* A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time.

Compilers and assemblers generate relocatable object files (including shared object files). Linkers generate executable object files. Technically, an *object module* is a sequence of bytes, and an *object file* is an object module stored on disk in a file. However, we will use these terms interchangeably.

Object file formats vary from system to system. The first Unix systems from Bell Labs used the `a.out` format. (To this day, executables are still referred to as `a.out` files.) Early versions of System V Unix used the Common Object File format (COFF). Windows NT uses a variant of COFF called the Portable Executable (PE) format. Modern Unix systems — such as Linux, later versions of System V Unix, BSD Unix variants, and Sun Solaris — use the Unix *Executable and Linkable Format (ELF)*. Although our discussion will focus on ELF, the basic concepts are similar, regardless of the particular format.

7.4 Relocatable Object Files

Figure 7.3 shows the format of a typical ELF relocatable object file. The *ELF header* begins with a 16-byte sequence that describes the word size and byte ordering of the system that generated the file. The rest of the ELF header contains information that allows a linker to parse and interpret the object file. This includes the size of the ELF header, the object file type (e.g., relocatable, executable, or shared), the machine type (e.g., IA32) the file offset of the *section header table*, and the size and number of entries in the section header table. The locations and sizes of the various sections are described by the *section header table*, which contains a fixed sized entry for each section in the object file.

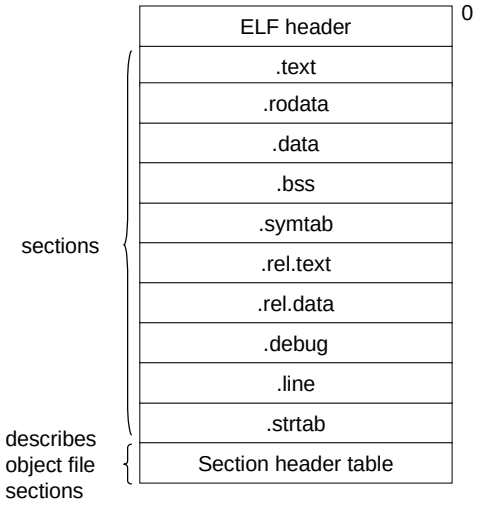


Figure 7.3: **Typical ELF relocatable object file.**

Sandwiched between the ELF header and the section header table are the sections themselves. A typical ELF relocatable object file contains the following sections:

`.text:` The machine code of the compiled program.

`.rodata:` Read-only data such as the format strings in `printf` statements, and jump tables for switch statements (see Problem 7.14).

`.data:` *Initialized* global C variables. Local C variables are maintained at run time on the stack, and do `not` appear in either the `.data` or `.bss` sections.

`.bss:` *Uninitialized* global C variables. This section occupies no actual space in the object file; it is merely a place holder. Object file formats distinguish between initialized and uninitialized variables for space efficiency: uninitialized variables do not have to occupy any actual disk space in the object file.

`.symtab:` A *symbol table* with information about functions and global variables that are defined and referenced in the program. Some programmers mistakenly believe that a program must be compiled with the `-g` option to get symbol table information. In fact, every relocatable object file has a symbol table in `.symtab`. However, unlike the symbol table inside a compiler, the `.symtab` symbol table does not contain entries for local variables.

`.rel.text:` A list of locations in the `.text` section that will need to be modified when the linker combines this object file with others. In general, any instruction that calls an external function or references a global variable will need to be modified. On the other hand, instructions that call local functions do not need to be modified. Note that relocation information is not needed in executable object files, and is usually omitted unless the user explicitly instructs the linker to include it.

`.rel.data:` Relocation information for any global variables that are referenced or defined by the module. In general, any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.

`.debug:` A debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the `-g` option.

`.line:` A mapping between line numbers in the original C source program and machine code instructions in the .text section. It is only present if the compiler driver is invoked with the `-g` option.

`.strtab:` A string table for the symbol tables in the `.symtab` and `.debug` sections, and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

Aside: Why is uninitialized data called `.bss`?
The use of the term `.bss` to denote uninitialized data is universal. It was originally an acronym for the "Block Storage Start" instruction from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the `.data` and `.bss` sections is to think of "bss" as an abbreviation for "Better Save Space!". **End Aside.**

7.5 Symbols and Symbol Tables

Each relocatable object module, m, has a symbol table that contains information about the symbols that are defined and referenced by m. In the context of a linker, there are three different kinds of symbols:

- *Global symbols* that are defined by module m and that can be referenced by other modules. Global linker symbols correspond to *nonstatic* C functions and global variables that are defined *without* the C `static` attribute.
- Global symbols that are referenced by module m but defined by some other module. Such symbols are called *externals* and correspond to C functions and variables that are defined in other modules.

- *Local symbols* that are defined and referenced exclusively by module m. Some local linker symbols correspond to C functions and global variables that are defined with the `static` attribute. These symbols are visible anywhere within module m, but cannot be referenced by other modules. The sections in an object file and the name of the source file that corresponds module m also get local symbols.

It is important to realize that local linker symbols are not the same as local program variables. The symbol table in `.symtab` does not contain any symbols that correspond to local nonstatic program variables. These are managed at run time on the stack and are not of interest to the linker.

Interestingly, local procedure variables that are defined with the C `static` attribute are not managed on the stack. Instead, the compiler allocates space in `.data` or `.bss` for each definition and creates a local linker symbol in the symbol table with a unique name. For example, suppose a pair of functions in the same module define a static local variable `x`:

```
1 int f()
2 {
3     static int x = 0;
4     return x;
5 }
6
7 int g()
8 {
9     static int x = 1;
10    return x;
11 }
```

In this case, the compiler allocates space for two integers in `.bss` and exports a pair of unique local linker symbols to the assembler. For example, it might use `x.1` for the definition in function `f` and `x.2` for the definition in function `g`.

> **New to C?**
> C programmers use the `static` attribute to hide variable and function declarations inside modules, much as you would use *public* and *private* declarations in Java and C++. C source files play the role of modules. Any global variable or function declared with the `static` attribute is private to that module. Similarly, any global variable or function declared without the `static` attribute is public, and can be accessed by any other module. It is good programming practice to protect your variables and functions with the `static` attribute wherever possible. **End**

Symbol tables are built by assemblers, using symbols exported by the compiler into the assembly language `.s` file. An ELF symbol table is contained in the `.symtab` section. It contains an array of entries. Figure 7.4 shows the format of each entry.

The `name` is a byte offset into the string table that points to the null-terminated string name of the symbol. The `value` is the symbol's address. For relocatable modules, the `value` is an offset from the beginning of the section where the object is defined. For executable object files, the value is an absolute run-time address. The `size` is the size (in bytes) of the object. The `type` is usually either data or function. The symbol table can also contain entries for the individual sections and for the path name of the original source file. So there

code/link/elfstructs.c

```
1 typedef struct {
2     int name;        /* string table offset */
3     int value;       /* section offset, or VM address */
4     int size;        /* object size in bytes */
5     char type:4,     /* data, func, section, or src file name (4 bits) */
6          binding:4;  /* local or global (4 bits) */
7     char reserved;   /* unused */
8     char section;    /* section header index, ABS, UNDEF, */
9                      /* or COMMON  */
10 } Elf_Symbol;
```

code/link/elfstructs.c

Figure 7.4: **ELF symbol table entry.** `type` and `binding` are four bits each.

are distinct types for these objects as well. The `binding` field indicates whether the symbol is local or global.

Each symbol is associated with some section of the object file, denoted by the `section` field, which is an index into the section header table. There are three special pseudo-sections that don't have entries in the section header table: ABS is for symbols that should not be relocated. UNDEF is for undefined symbols, that is, symbols that are referenced in this object module but defined elsewhere. COMMON is for uninitialized data objects that are not yet allocated. For COMMON symbols, the `value` field gives the alignment requirement, and `size` gives the minimum size.

For example, here are the last three entries in the symbol table for `main.o`, as displayed by the GNU READELF tool. The first eight entries, which are not shown, are local symbols that the linker uses internally.

```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     8 OBJECT  GLOBAL  0    3 buf
  9:        0    17 FUNC    GLOBAL  0    1 main
 10:        0     0 NOTYPE  GLOBAL  0  UND swap
```

In this example, we see an entry for the definition of global symbol `buf`, an 8-byte object located at an offset (i.e., `value`) of zero in the `.data` section. This is followed by the definition of the global symbol `main`, a 17-byte function located at an offset of zero in the `.text` section. The last entry comes from the reference for the external symbol `swap`. READELF identifies each section by an integer index. `Ndx=1` denotes the `.text` section, and `Ndx=3` denotes the `.data` section.

Similarly, here are the symbol table entries for `swap.o`:

```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     4 OBJECT  GLOBAL  0    3 bufp0
  9:        0     0 NOTYPE  GLOBAL  0  UND buf
 10:        0    39 FUNC    GLOBAL  0    1 swap
 11:        4     4 OBJECT  GLOBAL  0  COM bufp1
```

First, we see an entry for the definition of the global symbol `bufp0`, which is a 4-byte initialized object starting at offset 0 in `.data`. The next symbol comes from the reference to the external `buf` symbol in the initialization code for `bufp0`. This is followed by the global symbol `swap`, a 39-byte function at an offset of 0 in `.text`. The last entry is the global symbol `bufp1`, a 4-byte uninitialized data object (with a 4-byte alignment requirement) that will eventually be allocated as a `.bss` object when this module is linked.

Practice Problem 7.1:

This problem concerns the `swap.o` module from Figure 7.1(b). For each symbol that is defined or referenced in `swap.o`, indicate whether or not it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `main.o`), the symbol type (local, global, or extern) and the section (`.text`, `.data`, or `.bss`) it occupies in that module.

Symbol	`swap.o .symtab` entry?	Symbol type	Module where defined	Section
`buf`				
`bufp0`				
`bufp1`				
`swap`				
`temp`				

7.6 Symbol Resolution

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Symbol resolution is straightforward for references to local symbols that are defined in the same module as the reference. The compiler allows only one definition of each local symbol per module. The compiler also ensures that static local variables, which get local linker symbols, have unique names.

However, resolving references to global symbols is trickier. When the compiler encounters a symbol (either a variable or function name) that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle. If the linker is unable to find a definition for the referenced symbol in any of its input modules, it prints an (often cryptic) error message and terminates. For example, if we try to compile and link the following source file on a Linux machine,

```
1 void foo(void);
2
3 int main() {
4     foo();
5     return 0;
6 }
```

then the compiler runs without a hitch, but the linker terminates when it cannot resolve the reference to `foo`:

```
unix> gcc -Wall -O2 -o linkerror linkerror.c
```

```
/tmp/ccSz5uti.o: In function ‘main’:
/tmp/ccSz5uti.o(.text+0x7): undefined reference to ‘foo’
collect2: ld returned 1 exit status
```

Symbol resolution for global symbols is also tricky because the same symbol might be defined by multiple object files. In this case, the linker must either flag an error, or somehow chose one of the definitions and discard the rest. The approach adopted by Unix systems involves cooperation between the compiler, assembler, and linker, and can introduce some baffling bugs to the unwary programmer.

7.6.1 How Linkers Resolve Multiply-Defined Global Symbols

At compile time, the compiler exports each global symbol to the assembler as either *strong* or *weak*, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols. For the example program in Figure 7.1, `buf`, `bufp0`, `main`, and `swap` are strong symbols; `bufp1` is a weak symbol.

Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiply-defined symbols:

- Rule 1: Multiple strong symbols are not allowed.
- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.
- Rule 3: Given multiple weak symbols, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

```
1 /* foo1.c */
2 int main()
3 {
4     return 0;
5 }
```

```
1 /* bar1.c */
2 int main()
3 {
4     return 0;
5 }
```

In this case the linker will generate an error message because the strong symbol `main` is defined multiple times (Rule 1):

```
unix> gcc foo1.c bar1.c
/tmp/cca015022.o: In function ‘main’:
/tmp/cca015022.o(.text+0x0): multiple definition of ‘main’
/tmp/cca015021.o(.text+0x0): first defined here
```

Similarly, the linker will generate an error message for the following modules because the strong symbol `x` is defined twice (Rule 1):

```
1 /* foo2.c */
2 int x = 15213;
3
4 int main()
5 {
6     return 0;
7 }
```

```
1 /* bar2.c */
2 int x = 15213;
3
4 void f()
5 {
6 }
```

However, if x is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (Rule 2):

```
1 /* foo3.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x = 15213;
6
7 int main()
8 {
9     f();
10    printf("x = %d\n", x);
11    return 0;
12 }
```

```
1 /* bar3.c */
2 int x;
3
4 void f()
5 {
6     x = 15212;
7 }
```

At run time, function f changes the value of x from 15213 to 15212, which might come as a unwelcome surprise to the author of function main! Notice that the linker normally gives no indication that it has detected multiple definitions of x:

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./foobar3
x = 15212
```

The same thing can happen if there are two weak definitions of x (Rule 3):

```
1 /* foo4.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x;
6
7 int main()
8 {
9     x = 15213;
10    f();
11    printf("x = %d\n", x);
12    return 0;
13 }
```

```
1 /* bar4.c */
2 int x;
3
4 void f()
5 {
6     x = 15212;
7 }
```

The application of Rules 2 and 3 can introduce some insidious run-time bugs that are incomprehensible to the unwary programmer, especially if the duplicate symbol definitions have different types. Consider the following example, where `x` is defined as an `int` in one module and a `double` in another:

```
1 /* foo5.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x = 15213;
6 int y = 15212;
7
8 int main()
9 {
10     f();
11     printf("x = 0x%x y = 0x%x \n",
12            x, y);
13     return 0;
14 }
```

```
1 /* bar5.c */
2 double x;
3
4 void f()
5 {
6     x = -0.0;
7 }
```

On an IA32/Linux machine, `doubles` are 8 bytes and `ints` are 4 bytes. Thus, the assignment `x = -0.0` in line 5 of `bar5.c` will overwrite the memory locations for `x` and `y` (lines 5 and 6 in `foo5.c`) with the double-precision floating-point representation of negative one!

```
linux> gcc -o foobar5 foo5.c bar5.c
linux> ./foobar5
x = 0x0 y = 0x80000000
```

This is a subtle and nasty bug, especially because it occurs silently, with no warning from the compilation system, and because it typically manifests itself much later in the execution of the program, far away from where the error occurred. In a large system with hundreds of modules, a bug of this kind is extremely hard to fix, especially because many programmers are not aware of how linkers work. When in doubt, invoke the linker with a flag such as the GCC `-warn-common` flag, which instructs it to print a warning message when it resolves multiply-defined global symbol definitions.

Practice Problem 7.2:

In this problem, let `REF(x.i) --> DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example below, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (Rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (Rule 3), write "UNKNOWN".

A.
```
/* Module 1 */          /* Module 2 */
int main()              int main;
{                       int p2()
}                       {
                        }
```

(a) `REF(main.1) --> DEF(_____.___)`

(b) REF(main.2) --> DEF(_____.___)

B.
```
/* Module 1 */          /* Module 2 */
void main()             int main=1;
{                       int p2()
}                       {
                        }
```
(a) REF(main.1) --> DEF(_____.___)
(b) REF(main.2) --> DEF(_____.___)

C.
```
/* Module 1 */          /* Module 2 */
int x;                  double x=1.0;
void main()             int p2()
{                       {
}                       }
```
(a) REF(x.1) --> DEF(_____.___)
(b) REF(x.2) --> DEF(_____.___)

7.6.2 Linking with Static Libraries

So far we have assumed that the linker reads a collection of relocatable object files and links them together into an output executable file. In practice, all compilation systems provide a mechanism for packaging related object modules into a single file called a *static library*, which can then be supplied as input to the linker. When it builds the output executable, the linker copies only the object modules in the library that are referenced by the application program.

Why do systems support the notion of libraries? Consider ANSI C, which defines an extensive collection of standard I/O, string manipulation, and integer math functions such as `atoi`, `printf`, `scanf`, `strcpy`, and `random`. They are available to every C program in the `libc.a` library. ANSI C also defines an extensive collection of floating point math functions such as `sin`, `cos`, and `sqrt` in the `libm.a` library.

Consider the different approaches that compiler developers might use to provide these functions to users without the benefit of static libraries. One approach would be to have the compiler recognize calls to the standard functions and to generate the appropriate code directly. Pascal, which provides a small set of standard functions, takes this approach, but it is not feasible for C because of the large number of standard functions defined by the C standard. It would add significant complexity to the compiler and would require a new compiler version each time a function was added, deleted, or modified. To application programmers, however, this approach would be quite convenient because the standard functions would always be available.

Another approach would be to put all of the standard C functions in a single relocatable object module, say `libc.o`, that application programmers could link into their executables:

```
unix> gcc main.c /usr/lib/libc.o
```

This approach has the advantage that it would decouple the implementation of the standard functions from the implementation of the compiler, and would still be reasonably convenient for programmers. However, a

big disadvantage is that every executable file in a system would now contain a complete copy of the collection of standard functions, which would be extremely wasteful of disk space. (On a typical system, `libc.a` is about 8 MB and `libm.a` is about 1 MB.) Worse, each running program would now contain its own copy of these functions in memory, which would be extremely wasteful of memory. Another big disadvantage is that any change to any standard function, no matter how small, would require the library developer to recompile the entire source file, a time-consuming operation that would complicate the development and maintenance of the standard functions.

We could address some of these problems by creating a separate relocatable file for each standard function and storing them in a well-known directory. However this approach would require application programmers to explicitly link the appropriate object modules into their executables, a process that would be error prone and time-consuming:

```
unix> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

The notion of a static library was developed to resolve the disadvantages of these various approaches. Related functions can be compiled into separate object modules and then packaged in a single static library file. Application programs can then use any of the functions defined in the library by specifying a single file name on the command line. For example, a program that uses functions from the standard C library and the math library could be compiled and linked with a command of the form:

```
unix> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

At link time, the linker will only copy the object modules that are referenced by the program, which reduces the size of the executable on disk and in memory. On the other hand, the application programmer only needs to include the names of a few library files. (In fact, C compiler drivers always pass `libc.a` to the linker, so the reference to `libc.a` above is unnecessary.)

On Unix systems, static libraries are stored on disk in a particular file format known as an *archive*. An archive is a collection of concatenated relocatable object files, with a header that describes the size and location of each member object file. Archive filenames are denoted with the `.a` suffix. To make our discussion of libraries concrete, suppose that we want to provide the vector routines in Figure 7.5 in a static library called `libvector.a`.

To create the library, we would use the AR tool:

```
unix> gcc -c addvec.c multvec.c
unix> ar rcs libvector.a addvec.o multvec.o
```

To use the library, we might write an application such as `main2.c` in Figure 7.6, which invokes the `addvec` library routine. (The `include` file `vector.h` defines the function prototypes for the routines in `libvector.a`.)

To build the executable, we would compile and link the input files `main.o` and `libvector.a`:

```
unix> gcc -O2 -c main2.c
unix> gcc -static -o p2 main2.o ./libvector.a
```

Figure 7.7 summarizes the activity of the linker. The `-static` argument tells the compiler driver that the linker should build a fully-linked executable object file that can be loaded into memory and run without

code/link/addvec.c

```
1 void addvec(int *x, int *y,
2             int *z, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         z[i] = x[i] + y[i];
8 }
```

code/link/addvec.c

(a) `addvec.o`

code/link/multvec.c

```
1 void multvec(int *x, int *y,
2              int *z, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         z[i] = x[i] * y[i];
8 }
```

code/link/multvec.c

(a) `multvec.o`

Figure 7.5: **Member object files in** `libvector.a`**.**

code/link/main2.c

```
1 /* main2.c */
2 #include <stdio.h>
3 #include "vector.h"
4
5 int x[2] = {1, 2};
6 int y[2] = {3, 4};
7 int z[2];
8
9 int main()
10 {
11     addvec(x, y, z, 2);
12     printf("z = [%d %d]\n", z[0], z[1]);
13     return 0;
14 }
```

code/link/main2.c

Figure 7.6: **Example program 2:** This program calls member functions in the static `libvector.a` library.

any further linking at load time. When the linker runs, it determines that the `addvec` symbol defined by `addvec.o` is referenced by `main.o`, so it copies `addvec.o` into the executable. Since the program doesn't reference any symbols defined by `multvec.o`, the linker does *not* copy this module into the executable. The linker also copies the `printf.o` module from `libc.a`, along with a number of other modules from the C run-time system.

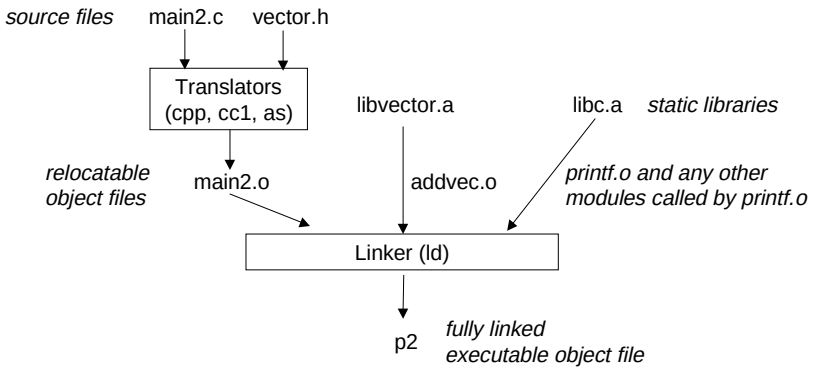


Figure 7.7: **Linking with static libraries.**

7.6.3 How Linkers Use Static Libraries to Resolve References

While static libraries are useful and essential tools, they are also a source of confusion to programmers because of the way the Unix linker uses them to resolve external references. During the symbol resolution phase, the linker scans the relocatable object files and archives left to right in the same sequential order that they appear on the compiler driver's command line. (The driver automatically translates any `.c` files on the command line into `.o` files.) During this scan, the linker maintains a set E of relocatable object files that will be merged to form the executable, a set U of unresolved symbols (i.e., symbols referred to but not yet defined), and a set D of symbols that have been defined in previous input files. Initially, E, U, and D are empty.

- For each input file f on the command line, the linker determines if f is an object file or an archive. If f is an object file, the linker adds f to E, updates U and D to reflect the symbol definitions and references in f, and proceeds to the next input file.
- If f is an archive, the linker attempts to match the unresolved symbols in U against the symbols defined by the members of the archive. If some archive member, m, defines a symbol that resolves a reference in U, then m is added to E, and the linker updates U and D to reflect the symbol definitions and references in m. This process iterates over the member object files in the archive until a fixed point is reached where U and D no longer change. At this point, any member object files not contained in E are simply discarded and the linker proceeds to the next input file.
- If U is nonempty when the linker finishes scanning the input files on the command line, it prints an error and terminates. Otherwise it merges and relocates the object files in E to build the output executable file.

Unfortunately, this algorithm can result in some baffling link-time errors because the ordering of libraries and object files on the command line is significant. If the library that defines a symbol appears on the command line before the object file that references that symbol, then the reference will not be resolved and linking will fail. For example:

```
unix> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function ‘main’:
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to ‘addvec’
```

Here is what happened: When `libvector.a` is processed, U is empty, so no member object files from `libvector.a` are added to E. Thus the reference to `addvec` is never resolved, and the linker emits an error message and terminates..

The general rule for libraries is to place them at the end of the command line. If the members of the different libraries are independent, in that no member references a symbol defined by another member, then the libraries can be placed at the end of the command line in any order.

On the other hand, if the libraries are not independent, then they must be ordered so that for each symbol s that is referenced externally by a member of an archive, at least one definition of s follows a reference to s on the command line. For example, suppose `foo.c` calls functions in `libx.a` and `libz.a` that call functions in `liby.a`. Then `libx.a` and `libz.a` must precede `liby.a` on the command line:

```
unix> gcc foo.c libx.a libz.a liby.a
```

Libraries can be repeated on the command line if necessary to satisfy the dependence requirements. For example, suppose `foo.c` calls a function in `libx.a` that calls a function in `liby.a` that calls a function in `libx.a`. Then `libx.a` must be repeated on the command line:

```
unix> gcc foo.c libx.a liby.a libx.a
```

Alternatively, we could combine `libx.a` and `liby.a` into a single archive.

Practice Problem 7.3:

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of file object file and library arguments) that will allow the static linker to resolve all symbol references.

A. `p.o` → `libx.a`.

B. `p.o` → `libx.a` → `liby.a`.

C. `p.o` → `libx.a` → `liby.a` **and** `liby.a` → `libx.a` →`p.o`.

7.7 Relocation

Once the linker has completed the symbol resolution step, it has associated each symbol reference in the code with exactly one symbol definition (i.e., a symbol table entry in one of its input object modules). At

this point, the linker knows the exact sizes of the code and data sections in its input object modules. It is now ready to begin the relocation step, where it merges the input modules and assigns run-time addresses to each symbol. Relocation consists of two steps:

- *Relocating sections and symbol definitions.* In this step, the linker merges all sections of the same type into a new aggregate section of the same type. For example, the `.data` sections from the input modules are all merged into one section that will become the `.data` section for the output executable object file. The linker then assigns run-time memory addresses to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules. When this step is complete, every instruction and global variable in the program has a unique run-time memory address.
- *Relocating symbol references within sections.* In this step, the linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses. To perform this step, the linker relies on data structures in the relocatable object modules known as relocation entries, which we describe next.

7.7.1 Relocation Entries

When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate location is unknown, it generates a *relocation entry* that tells the linker how to modify the reference when it merges the object file into an executable. Relocation entries for code are placed in `.relo.text`. Relocation entries for initialized data are placed in `.relo.data`.

Figure 7.8 shows the format of an ELF relocation entry. The `offset` is the section offset of the reference that will need to be modified. The `symbol` identifies the symbol that the modified reference should point to. The `type` tells the linker how to the modify the new reference.

code/link/elfstructs.c

```
1 typedef struct {
2     int offset;     /* offset of the reference to relocate */
3     int symbol:24,  /* symbol the reference should point to */
4         type:8;     /* relocation type */
5 } Elf32_Rel;
```

code/link/elfstructs.c

Figure 7.8: **ELF relocation entry.** Each entry identifies a reference that must be relocated.

ELF defines 11 different relocation types, some quite arcane. We are concerned with only the two most basic relocation types:

- `R_386_PC32`: Relocate a reference that uses a 32-bit PC-relative address. Recall from Section 3.6.3 that a PC-relative address is an offset from the current run-time value of the program counter (PC).

When the CPU executes an instruction using PC-relative addressing, it forms the *effective address* (e.g., the target of the `call` instruction) by adding the 32-bit value encoded in the instruction to the current run-time value of the PC, which is always the address of the next instruction in memory.

- `R_386_32`: Relocate a reference that uses a 32-bit absolute address. With absolute addressing, the CPU directly uses the 32-bit value encoded in the instruction as the effective address, without further modifications.

7.7.2 Relocating Symbol References

Figure 7.9 shows the pseudo-code for the linker's relocation algorithm.

```
1 foreach section s {
2     foreach relocation entry r {
3         refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5         /* relocate a PC-relative reference */
6         if (r.type == R_386_PC32) {
7             refaddr = ADDR(s) + r.offset; /* ref's runtime address */
8             *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9         }
10
11        /* relocate an absolute reference */
12        if (r.type == R_386_32)
13            *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14    }
15 }
```

Figure 7.9: **Relocation algorithm.**

Lines 1 and 2 iterate over each section `s` and each relocation entry `r` associated with each section. For concreteness, assume that each section `s` is an array of bytes and that each relocation entry `r` is a `struct` of type `Elf32_Rel`, as defined in Figure 7.8. Also, assume that when the algorithm runs, the linker has already chosen run-time addresses for each section (denoted `ADDR(s)`), and each symbol (denoted `ADDR(r.symbol)`). Line 3 computes the address in the `s` array of the 4-byte reference that needs to be relocated. If this reference uses PC-relative addressing, then it is relocated by lines 5–9. If the reference uses absolute addressing, then it is relocated by lines 11–13.

Relocating PC-Relative References

Recall from our running example in Figure 7.1(a) that the `main` routine in the `.text` section of `main.o` calls the `swap` routine, which is defined in `swap.o`. Here is the disassembled listing for the `call` instruction, as generated by the GNU OBJDUMP tool:

```
6:   e8 fc ff ff ff          call   7 <main+0x7>    swap();
                             7: R_386_PC32 swap     relocation entry
```

From this listing we see that the `call` instruction begins at section offset `0x6` and consists of the 1-byte opcode `0xe8`, followed by the 32-bit reference `0xfffffffc` (-4 decimal), which is stored in little-endian byte order. We also see a relocation entry for this reference displayed on the following line. (Recall that relocation entries and instructions are actually stored in different sections of the object file. The OBJDUMP tool displays them together for convenience.) The relocation entry `r` consists of three fields:

```
r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

that tell the linker to modify the 32-bit PC-relative reference starting at offset `0x7` so that it will point to the `swap` routine at run time. Now suppose that the linker has determined that

```
ADDR(s) = ADDR(.text) = 0x80483b4
```

and

```
ADDR(r.symbol) = ADDR(swap) = 0x80483c8.
```

Using the algorithm in Figure 7.9, the linker first computes the run-time address of the reference (line 7):

```
refaddr = ADDR(s)   + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```

and then updates the reference from its current value (-4) to `0x9` so that it will point to the `swap` routine at run time (line 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
        = (unsigned) (0x80483c8      + (-4)    - 0x80483bb)
        = (unsigned) (0x9)
```

In the resulting executable object file, the `call` instruction has the following relocated form:

```
 80483ba:  e8 09 00 00 00          call   80483c8 <swap>          swap();
```

At run time, the `call` instruction will be stored at address `0x80483ba`. When the CPU executes the `call` instruction, the PC has a value of `0x80483bf`, which is the address of the instruction immediately following the `call` instruction. To execute the instruction, the CPU performs the following steps:

```
1. push PC onto stack
2. PC <- PC + 0x9 = 0x80483bf + 0x9 = 0x80483c8
```

Thus, the next instruction to execute is the first instruction of the `swap` routine, which of course is what we want!

You may wonder why the assembler created the reference in the `call` instruction with an initial value of -4. The assembler uses this value as a bias to account for the fact that the PC always points to the instruction following the current instruction. On a different machine with different instruction sizes and encodings, the assembler for that machine would use a different bias. This is powerful trick that allows the linker to blindly relocate references, blissfully unaware of the instruction encodings for a particular machine.

Relocating Absolute References

Recall that in our example program in Figure 7.1, the `swap.o` module initializes the global pointer `bufp0` to the address of the first element of the global `buf` array:

```
int *bufp0 = &buf[0];
```

Since `bufp0` is an initialized data object, it will be stored in the `.data` section of the `swap.o` relocatable object module. Since it is initialized to the address of a global array, it will need to be relocated. Here is the disassembled listing of the `.data` section from `swap.o`:

```
00000000 <bufp0>:
   0:   00 00 00 00                                    int *bufp0 = &buf[0];
                        0: R_386_32 buf                relocation entry
```

We see that the `.data` section contains a single 32-bit reference, the `bufp0` pointer, which has a value of `0x0`. The relocation entry tells the linker that this is a 32-bit absolute reference, beginning at offset 0, which must relocated so that it points to the symbol `buf`. Now suppose that the linker has determined that

```
ADDR(r.symbol) = ADDR(buf) = 0x8049454
```

The linker updates the reference using line 13 of the algorithm in Figure 7.9:

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr)
        = (unsigned) (0x8049454      + 0)
        = (unsigned) (0x8049454)
```

In the resulting executable object file, the reference has the following relocated form:

```
0804945c <bufp0>:
 804945c:  54 94 04 08                          Relocated!
```

In words, the linker has decided that at run time, the variable `bufp0` will be located at memory address `0x804945c` and will be initialized to `0x8049454`, which is the run-time address of the `buf` array.

The `.text` section in the `swap.o` module contains five absolute references that are relocated in a similar way (See Problem 7.12). Figure 7.10 shows the relocated `.text` and `.data` sections in the final executable object file.

Practice Problem 7.4:

This problem concerns the relocated program in Figure 7.10.

A. What is the hex address of the relocated reference to `swap` in line 5?

B. What is the hex value of the relocated reference to `swap` in line 5?

C. Suppose the linker had decided for some reason to locate the `.text` section at `0x80483b8` instead of `0x80483b4`. What would the hex value of the relocated reference in line 5 be in this case?

code/link/p-exe.d

```
080483b4 <main>:
 80483b4:  55                      push   %ebp
 80483b5:  89 e5                   mov    %esp,%ebp
 80483b7:  83 ec 08                sub    $0x8,%esp
 80483ba:  e8 09 00 00 00          call   80483c8 <swap>          swap();
 80483bf:  31 c0                   xor    %eax,%eax
 80483c1:  89 ec                   mov    %ebp,%esp
 80483c3:  5d                      pop    %ebp
 80483c4:  c3                      ret
 80483c5:  90                      nop
 80483c6:  90                      nop
 80483c7:  90                      nop

080483c8 <swap>:
 80483c8:  55                      push   %ebp
 80483c9:  8b 15 5c 94 04 08       mov    0x804945c,%edx          Get *bufp0
 80483cf:  a1 58 94 04 08          mov    0x8049458,%eax          Get buf[1]
 80483d4:  89 e5                   mov    %esp,%ebp
 80483d6:  c7 05 48 95 04 08 58    movl   $0x8049458,0x8049548    bufp1 = &buf[1]
 80483dd:  94 04 08
 80483e0:  89 ec                   mov    %ebp,%esp
 80483e2:  8b 0a                   mov    (%edx),%ecx
 80483e4:  89 02                   mov    %eax,(%edx)
 80483e6:  a1 48 95 04 08          mov    0x8049548,%eax          Get *bufp1
 80483eb:  89 08                   mov    %ecx,(%eax)
 80483ed:  5d                      pop    %ebp
 80483ee:  c3                      ret
```

code/link/p-exe.d

(a) Relocated `.text` section.

code/link/pdata-exe.d

```
08049454 <buf>:
 8049454:  01 00 00 00 02 00 00 00

0804945c <bufp0>:
 804945c:  54 94 04 08                    Relocated!
```

code/link/pdata-exe.d

(b) Relocated `.data` section.

Figure 7.10: **Relocated `.text` and `data` sections for executable file `p`** The original C code is in Figure 7.1.

7.8 Executable Object Files

We have seen how the linker merges multiple object modules into a single executable object file. Our C program, which began life as a collection of ASCII text files, has been transformed into a single binary file that contains all of the information needed to load the program into memory and run it. Figure 7.11 summarizes the kinds of information in a typical ELF executable file.

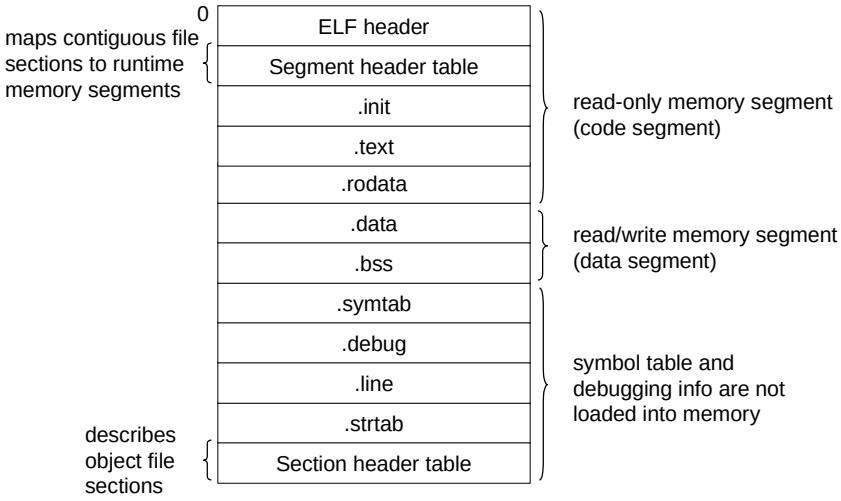


Figure 7.11: **Typical ELF executable object file**

The format of an executable object file is similar to that of a relocatable object file. The ELF header describes the overall format of the file. It also includes the program's *entry point*, which is the address of the first instruction to execute when the program runs. The `.text`, `.rodata`, and `.data` sections are similar to those in a relocatable object file, except that these sections have been relocated to their eventual run-time memory addresses. The `.init` section defines a small function, called `_init`, that will be called by the program's initialization code. Since the executable is *fully linked* (relocated), it needs no `.relo` sections.

ELF executables are designed to be easy to load into memory, with contiguous chunks of the executable file mapped to contiguous memory segments. This mapping is described by the *segment header table*. Figure 7.12 shows the segment header table for our example executable `p`, as displayed by OBJDUMP.

From the segment header table, we see that two memory segments will be initialized with the contents of the executable object file. Lines 1 and 2 tell us that the first segment (the *code segment*) is aligned to a 4 KB (2^{12}) boundary, has read/execute permissions, starts at memory address `0x08048000`, has a total memory size of `0x448` bytes, and is initialized with the first `0x448` bytes of the executable object file, which includes the ELF header, the segment header table, and the `.init`, `.text`, and `.rodata` sections.

Lines 3 and 4 tell us that the second segment (the *data segment*) is aligned to a 4 KB boundary, has read/write permissions, starts at memory address `0x08049448`, has a total memory size of `0x104` bytes, and is initialized with the `0xe8` bytes starting at file offset `0x448`, which in this case is the beginning of the `.data` section. The remaining bytes in the segment correspond to `.bss` data that will initialized to zero at run time.

code/link/p-exe.d

```
   Read-only code segment
1  LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
2       filesz 0x00000448 memsz 0x00000448 flags r-x

   Read/write data segment
3  LOAD off    0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
4       filesz 0x000000e8 memsz 0x00000104 flags rw-
```

code/link/p-exe.d

Figure 7.12: **Segment header table for the example executable** `p`. Legend: `off:` file offset, `vaddr/paddr:` virtual/physical address, `align:`, segment alignment, `filesz:` segment size in the object file, `memsz:` segment size in memory, `flags:` run-time permissions.

7.9 Loading Executable Object Files

To run an executable object file `p`, we can type its name to the Unix shell's command line:

```
unix> ./p
```

Since `p` does not correspond to a built-in shell command, the shell assumes that `p` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the `loader`. Any Unix program can invoke the loader by calling the `execve` function, which we will describe in detail in Section 8.4.6. The loader copies the code and data in the executable object file from disk into memory, and then runs the program by jumping to its first instruction, or *entry point*. This process of copying the program into memory and then running it is known as *loading*.

Every Unix program has a run-time memory image similar to the one in Figure 7.13. On Linux systems, the code segment always starts at address `0x08048000`. The data segment follows at the next 4-KB aligned address. The run-time *heap* follows on the first 4-KB aligned address past the read/write segment and grows up via calls to the `malloc` library. (We will describe `malloc` and the heap in detail in Section 10.9). The segment starting at address `0x40000000` is reserved for shared libraries. The user stack always starts at address `0xbfffffff` and grows down (towards lower memory addresses). The segment starting above the stack at address `0xc0000000` is reserved for the code and data in the memory-resident part of the operating system known as the *kernel*.

When the loader runs, it creates the memory image shown in Figure 7.13. Guided by the segment header table in the executable, it copies chunks of the executable into the code and data segments. Next, the loader jumps to the program's entry point, which is always the address of the `_start` symbol. The *startup code* at the `_start` address is defined in the object file `crt1.o` and is the same for all C programs. Figure 7.14 shows the specific sequence of calls in the startup code. After calling initialization routines in from the `.text` and `.init` sections, the startup code calls the `atexit` routine, which appends a list of routines that should be called when the application calls the `exit` function. The `exit` function runs the functions registered by `atexit`, and then returns control to the operating system by calling `_exit`). Next, the startup code calls the application's `main` routine, which begins executing our C code. After the application returns, the startup code calls the `_exit` routine, which returns control to the operating system.

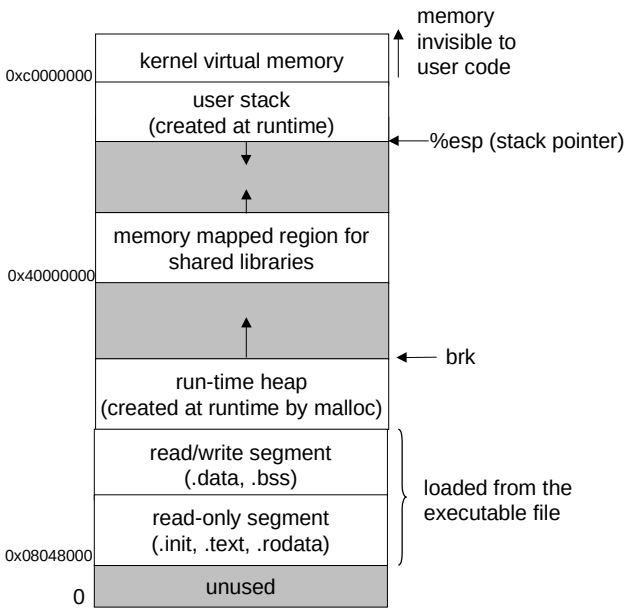


Figure 7.13: **Linux run-time memory image**

```
1 0x080480c0 <_start>:         /* entry point in .text */
2     call __libc_init_first   /* startup code in .text */
3     call _init               /* startup code in .init */
4     call atexit              /* startup code in .text */
5     call main                /* application main routine */
6     call _exit               /* returns control to OS */
7 /* control never reaches here */
```

Figure 7.14: **Pseudo-code for the `crt1.o` startup routine in every C program.** Note: The code that pushes the arguments for each function is not shown.

Aside: How do loaders really work?
Our description of loading is conceptually correct, but intentionally not entirely accurate. To understand how loading really works, you must understand concepts of *processes*, *virtual memory*, and *memory mapping* that we haven't discussed yet. As we encounter these concepts later in Chapters 8 and 10, we will revisit loading and gradually reveal the mystery to you.

For the impatient reader, here is a preview of how loading really works: Each program in a Unix system runs in the context of a process with its own virtual address space. When the shell runs a program, the parent shell process forks a child process that is a duplicate of the parent. The child process invokes the loader via the `execve` system call. The loader deletes the child's existing virtual memory segments, and creates a new set of code, data, heap, and stack segments. The new stack and heap segments are initialized to zero. The new code and data segments are initialized to the contents of the executable file by mapping pages in the virtual address space to page-sized chunks of the executable file. Finally, the loader jumps to the `_start` address, which eventually calls the application's `main` routine. Aside from some header information, there is no copying of data from disk to memory during loading. The copying is deferred until the CPU references a mapped virtual page, at which point the operating system automatically transfers the page from disk to memory using its paging mechanism. **End Aside.**

Practice Problem 7.5:

A. Why does every C program need a routine called `main`?

B. Have you ever wondered why a C `main` routine can end with a call to `exit`, a `return` statement, or neither, and yet the program still terminates properly? Explain.

7.10 Dynamic Linking with Shared Libraries

The static libraries that we studied in Section 7.6.2 address many of the issues associated with making large collections of related functions available to application programs. However, static libraries still have some significant disadvantages. Static libraries, like all software, need to be maintained and updated periodically. If application programmers want to use the most recent version of a library, they must somehow become aware that the library has changed, and then explicitly relink their programs against the updated library.

Another issue is that almost every C program uses standard I/O functions such as `printf` and `scanf`. At run time, the code for these functions is duplicated in the text segment of each running process. On a typical system that is running 50–100 processes, this can be a significant waste of scarce memory system resources. (An interesting property of memory is that it is *always* a scarce resource, regardless of how much there is in a system. Disk space and kitchen trash cans share this same property.)

Shared libraries are a modern innovation that address the disadvantages of static libraries. A shared library is an object module that, *at run time*, can be loaded at an arbitrary memory address and linked with a program in memory. This process is known as *dynamic linking*, and is performed by a program called a *dynamic linker*.

Shared libraries are also referred to as *shared objects* and on Unix systems are typically denoted by the `.so` suffix. Microsoft operating systems refer to shared libraries as DLLs (dynamic link libraries).

Shared libraries are "shared" in two different ways. First, in any given file system, there is exactly one `.so` file for a particular library. The code and data in this `.so` file are shared by all of the executable object files that reference the library, as opposed to the contents of static libraries, which are copied and embedded

in the executables that reference them. Second, a single copy of the `.text` section of a shared library in memory can be shared by different running processes. We will explore this in more detail when we study virtual memory in Chapter 10.

Figure 7.15 summarizes the dynamic linking process for the example program in Figure 7.6. To build a shared library `libvector.so` of our example vector arithmetic routines in Figure 7.5, we would invoke the compiler driver with a special directive to the linker:

```
unix> gcc -shared -fPIC -o libvector.so addvec.c multvec.c
```

The `-fPIC` flag directs the compiler to generate position independent code (more on this in the next section). The `-shared` flag directs the linker to create a shared object file.

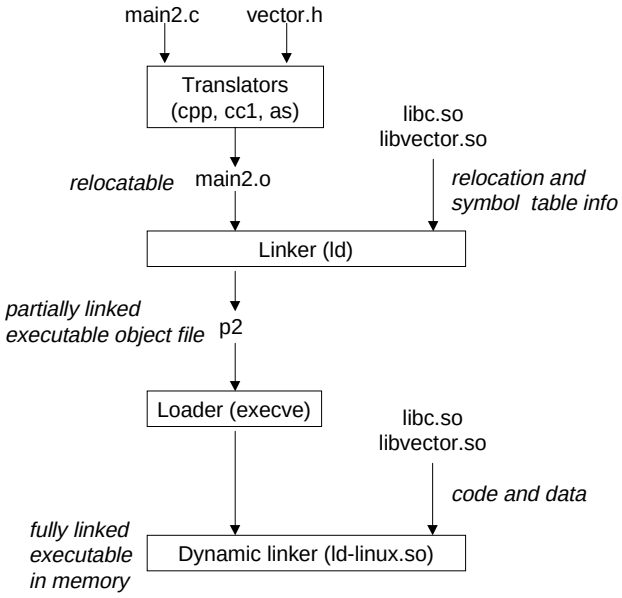


Figure 7.15: **Dynamic linking with shared libraries.**

Once we have created the library, we would then link it into our example program in Figure 7.6.

```
unix> gcc -o p2 main2.c ./libvector.so
```

This creates an executable object file `p2` in a form that can be linked with `libvector.so` at run time. The basic idea is to do some of the linking statically when the executable file is created, and then complete the linking process dynamically when the program is loaded.

It is important to realize that none of the code or data sections from `libvector.so` are actually copied into the executable `p2` at this point. Instead, the linker copies some relocation and symbol table information that will allow references to code and data in `libvector.so` to be resolved at run time.

When the loader loads and runs the executable `p2`, it loads the partially linked executable `p2`, using the techniques discussed in Section 7.9. Next, it notices that `p2` contains a `.interp` section, which contains the path name of the dynamic linker, which is itself a shared object (e.g., LD-LINUX.SO on Linux systems).

Instead of passing control to the application, as it would normally do, the loader loads and runs the dynamic linker.

The dynamic linker then finishes the linking task by:

- Relocating the text and data of `libc.so` into some memory segment. On IA32/Linux systems, shared libraries are loaded in the area starting at address `0x40000000` (See Figure 7.13).
- Relocating the text and data of `libvector.so` into another memory segment.
- Relocating any references in `p2` to symbols defined by `libc.so` and `libvector.so`.

Finally, the dynamic linker passes control to the application. From this point on, the locations of the shared libraries are fixed and do not change during execution of the program.

7.11 Loading and Linking Shared Libraries from Applications

To this point we have discussed the scenario where the dynamic linker loads and links shared libraries when an application is loaded, just before it executes. However, it is also possible for an application to request the dynamic linker to load and link arbitrary shared libraries while the application is running, without having to linked the applications against those libraries at compile time.

Dynamic linking is a powerful and useful technique. For example, developers of Microsoft Windows applications frequently use shared libraries to distribute software updates. They generate a new copy of a shared library, which users can then download and use a replacement for the current version. The next time they run their application, it will automatically link and load the new shared library.

Another example: the servers at many Web sites generate a great deal of *dynamic content* such as personalized Web pages, account balances, and banner ads. The earliest Web servers generated dynamic content by using `fork` and `execve` to create a child process and run a "CGI program" in the context of the child.

However, modern Web servers generate dynamic content using a more efficient and sophisticated approach based on dynamic linking. The idea is to package each function that generates dynamic content in a shared library. When a request arrives from a Web browser, the server dynamically loads and links the appropriate function and then calls it directly, as opposed to using `fork` and `execve` to run the function in the context of a child process. The function remains in the server's address space, so subsequent requests can be handled at the cost of a simple function call. This can have a significant impact on the throughput of a busy site. Further, existing functions can be updated and new functions can be added at run-time, without stopping the server.

Linux and Solaris systems provide a simple interface to the dynamic linker that allows application programs to load and link shared libraries at run time.

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
```

returns: ptr to handle if OK, NULL on error

The `dlopen` function loads and links the shared library `filename`. The external symbols in `filename` are resolved using libraries previously opened with the `RTLD_GLOBAL` flag. If the current executable was compiled with the `-rdynamic` flag, then its global symbols are also available for symbol resolution. The `flag` argument must include either `RTLD_NOW`, which tells the linker to resolve references to external symbols immediately, or the `RTLD_LAZY` flag, which instructs the linker to defer symbol resolution until code from the library is executed. Either of these values can be or'd with the `RTLD_GLOBAL` flag.

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
                                    returns: ptr to symbol if OK, NULL on error
```

The `dlsym` function takes a `handle` to a previously opened shared library and a `symbol` name, and returns the address of the symbol, if it exists, or NULL otherwise.

```
#include <dlfcn.h>

int dlclose (void *handle);                    returns: 0 if OK, -1 on error
```

The `dlclose` function unloads the shared library if no other shared libraries are still using it.

```
#include <dlfcn.h>

const char *dlerror(void);
        returns: error msg if previous call to dlopen, dlsym, or dlclose failed, NULL if previous call was OK
```

The `dlerror` function returns a string describing the most recent error that occurred as a result of calling `dlopen`, `dlsym`, or `dlclose`, or NULL if no error occurred.

Figure 7.16 shows how we would use this interface to dynamically link our `libvector.so` shared library (Figure 7.5), and then invoke its `addvec` routine. To compile the program, we would invoke GCC in the following way:

```
unix> gcc -rdynamic -O2 -o p3 main3.c -ldl
```

7.12 *Position-Independent Code (PIC)

A key motivation for shared libraries is to allow multiple running processes to share the same library code in memory, and thus save precious memory resources. So how might multiple processes share a single copy of a program? One approach would be to assign *a priori* a dedicated chunk of the address space to each shared library, and then require the loader to always load the shared library at that address. While

code/link/dll.c

```
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    /* get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() it just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

code/link/dll.c

Figure 7.16: **An application program that dynamically loads and links the shared library** `libvector.so`**.**

straightforward, this approach creates some serious problems. It would be an inefficient use of the address space since portions of the space would be allocated even if a process didn't use the library. Second, it would difficult to manage. We would have to ensure that none of the chunks overlapped. Every time a library was modified we would have to make sure that it still fit in its assigned chunk. If not, then we would have to find a new chunk. And if we created a new library, we would have to find room for it. Over time, given the hundreds of libraries and versions of libraries in a system, it would be difficult to keep the address space from fragmenting into lots of small unused but unusable holes. Even worse, the assignment of libraries to memory would be different for each system, thus creating even more management headaches.

A better approach is to compile library code so that it can be loaded and executed at any address without being modified by the linker. Such code is known as *position-independent code* (PIC). Users direct GNU compilation systems to generate PIC code with the `-fPIC` option to GCC.

On IA32 systems, calls to procedures in the same object module require no special treatment, since the references are PC-relative, with known offsets, and hence are already PIC (see Problem 7.4). However, calls to externally-defined procedures and references to global variables are not normally PIC, since they require relocation at link time.

PIC Data References

Compilers generate PIC references to global variables by exploiting the following interesting fact: No matter where we load an object module (including shared object modules) in memory, the data segment is always allocated immediately after the code segment. Thus, the *distance* between any instruction in the code segment and any variable in the data segment is a run-time constant, independent of the absolute memory locations of the code and data segments.

To exploit this fact, the compiler creates a table called the *global offset table* (GOT) at the beginning of the data segment. The GOT contains an entry for each global data object that is referenced by the object module. The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each entry in the GOT so that it contains the appropriate absolute address. Each object module that references global data has its own GOT.

At run time, each global variable is referenced indirectly through the GOT using code of the form:

```
    call L1
L1: popl %ebx;           # ebx contains the current PC
    addl $VAROFF, %ebx   # ebx points to the GOT entry for var
    movl (%ebx), %eax    # reference indirect through the GOT
    movl (%eax), %eax
```

In this fascinating piece of code, the call to `L1` pushes the return address (which happens to be the address of the `popl` instruction) on the stack. The `popl` instruction then pops this address into `%ebx`. The net effect of these two instructions is to move the value of the PC into register `%ebx`.

The `addl` instruction adds a constant offset to `%ebx` so that it points to the appropriate entry in the GOT, which contains the absolute address of the data item. At this point, the global variable can be referenced indirectly through the GOT entry contained in `%ebx`. In the example above, the two `movl` instructions load the contents of the global variable (indirectly through the GOT) into register `%eax`.

PIC code has performance disadvantages. Each global variable reference now requires five instructions instead of one, with an additional memory reference to the GOT. Also, PIC code uses an additional register to hold the address of the GOT entry. On machines with large register files, this is not a major issue. But on register-starved IA32 systems, losing even one register can trigger spilling of the registers onto the stack.

PIC Function Calls

It would certainly be possible for PIC code to use the same approach for resolving external procedure calls:

```
    call L1
L1: popl %ebx;           # ebx contains the current PC
    addl $PROCOFF, %ebx  # ebx points to GOT entry for proc
    call *(%ebx)         # call indirect through the GOT
```

However, this approach would require three additional instructions for each run-time procedure call. Instead, ELF compilation systems use an interesting technique, called *lazy binding*, that defers the binding of procedure addresses until the first time the procedure is called. There is a nontrivial run-time overhead the first time the procedure is called, but each call thereafter only costs a single instruction and a memory reference for the indirection.

Lazy binding is implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the *procedure linkage table (PLT)*. If an object module calls any functions that are defined in shared libraries, then it has its own GOT and PLT. The GOT is part of the `.data` section. The PLT is part of the `.text` section.

Figure 7.17 shows the format of the GOT for the example program `main2.o` from Figure 7.6. The first three GOT entries are special: GOT[0] contains the address of the `.dynamic` segment, which contains information that the dynamic linker uses to bind procedure addresses, such as the location of the symbol table and relocation information. GOT[1] contains some information that defines this module. GOT[2] contains an entry point into the lazy binding code of the dynamic linker.

Address	Entry	Contents	Description
08049674	GOT[0]	0804969c	address of .dynamic section
08049678	GOT[1]	4000a9f8	identifying info for the linker
0804967c	GOT[2]	4000596f	entry point in dynamic linker
08049680	GOT[3]	0804845a	address of pushl in PLT[1] (printf)
08049684	GOT[4]	0804846a	address of pushl in PLT[2] (addvec)

Figure 7.17: **The global offset table (GOT) for executable `p2`.** The original code is in Figures 7.5 and 7.6.

Each procedure that is defined in a shared object and called by `main2.o` gets an entry in the GOT, starting with entry GOT[3]. For the example program, we have shown the GOT entries for `printf`, which is defined in `libc.so` and `addvec`, which is defined in `libvector.so`.

Figure 7.18 shows the PLT for our example program `p2`. The PLT is an array of 16-byte entries. The first entry, PLT[0], is a special entry that jumps into the dynamic linker. Each called procedure has an entry in the

PLT, starting at PLT[1]. In the figure, PLT[1] corresponds to `printf` and PLT[2] corresponds to `addvec`.

```
PLT[0]
08048444:  ff 35 78 96 04 08  pushl  0x8049678   # push &GOT[1]
 804844a:  ff 25 7c 96 04 08  jmp  *0x804967c    # jmp to *GOT[2](linker)
 8048450:  00 00                                 # padding
 8048452:  00 00                                 # padding

PLT[1] <printf>
 8048454:  ff 25 80 96 04 08  jmp    *0x8049680  # jmp to *GOT[3]
 804845a:  68 00 00 00 00     pushl  $0x0        # ID for printf
 804845f:  e9 e0 ff ff ff     jmp    8048444     # jmp to PLT[0]

PLT[2] <addvec>
 8048464:  ff 25 84 96 04 08  jmp    *0x8049684  # jump to *GOT[4]
 804846a:  68 08 00 00 00     pushl  $0x8        # ID for addvec
 804846f:  e9 d0 ff ff ff     jmp    8048444     # jmp to PLT[0]

<other PLT entries>
```

Figure 7.18: **The procedure linkage table (PLT) for executable `p2`.** The original code is in Figures 7.5 and 7.6.

Initially, after the program has been dynamically linked and begins executing, procedures `printf` and `addvec` are bound to the first instruction in their respective PLT entries. For example, the call to addvec has the form:

```
80485bb:   e8 a4 fe ff ff   call 8048464 <addvec>
```

When `addvec` is called the first time, control passes to the first instruction in PLT[2], which does an indirect jump through GOT[4]. Initially, each GOT entry contains the address of the `pushl` entry in the corresponding PLT entry. So the indirect jump in the PLT simply transfers control back to the next instruction in PLT[2]. This instruction pushes an ID for the `addvec` symbol onto the stack. The last instruction jumps to PLT[0], which pushes another word of identifying information on the stack from GOT[1], and then jumps into the dynamic linker indirectly through GOT[2]. The dynamic linker uses the two stack entries to determine the location of `addvec`, overwrites GOT[4] with this address, and passes control to `addvec`.

The next time `addvec` is called in the program, control passes to PLT[2] as before. However, this time the indirect jump through GOT[4] transfers control to `addvec`. The only additional overhead from this point on is the memory reference for the indirect jump.

7.13 Tools for Manipulating Object Files

There are a number of tools available on Unix systems to help you understand and manipulate object files. In particular, the GNU *binutils* package is especially helpful and runs on every Unix platform.

AR: Creates static libraries, and inserts, deletes, lists, and extracts members.

STRINGS: Lists all of the printable strings contained in an object file.

STRIP: Deletes symbol table information from an object file.

NM: Lists the symbols defined in the symbol table of an object file.

SIZE: Lists the names and sizes of the sections in an object file.

READELF: Displays the complete structure of an object file, including all of the information encoded in the ELF header. Subsumes the functionality of SIZE and NM.

OBJDUMP: The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the `.text` section.

Unix systems also provide the `ldd` program for manipulating shared libraries:

LDD: Lists the shared libraries that an executable needs at run time.

7.14 Summary

We have learned that linking can be performed at compile time by static linkers, and at load time and run time by dynamic linkers. The main tasks of linkers are symbol resolution, where each global symbol is bound to a unique definition, and relocation, where the ultimate memory address for each symbol is determined and where references to those objects are modified.

Static linkers combine multiple relocatable object files into a single executable object file. Multiple object files can define the same symbol, and the rules that linkers use for silently resolving these multiple definitions can introduce subtle bugs in user programs. Multiple object files can be concatenated in a single static library. Linkers use libraries to resolve symbol references in other object modules. The left-to-right sequential scan that many linkers use to resolve symbol references is another source of confusing link-time errors.

Loaders map the contents of executable files into memory and run the program. Linkers can also produce partially linked executable object files with unresolved references to the routines and data defined in shared library. At load time, the loader maps the partially linked executable into memory and then calls a dynamic linker, which completes the linking task by loading the shared library and relocating the references in the program. Shared libraries that are compiled as position-independent code can be loaded anywhere and shared at run time by multiple processes. Applications can also use the dynamic linker at run time in order to load, link, and access the functions and data in shared libraries.

Bibliographic Notes

Linking is not well documented in the computer science literature. We think there are several reasons for this. First, linking lies at the intersection of compilers, computer architecture, and operating systems,

requiring understanding of code generation, machine language programming, program instantiation, and virtual memory. It does not fit neatly into any of the usual computer science specialties, and thus is not well covered well by the classic texts in these areas. However, Levine's monograph is a good general reference on the subject [44]. The original specifications for ELF and DWARF (a specification for the contents of the `.debug` and `.line` sections) are described in [32].

Some interesting research and commercial activity centers around the notion of *binary translation*, where the contents of an object file are parsed, analyzed, and modified. Binary translation is typically for three purposes [43]: to emulate one system on another system, to observe program behavior, or to perform system-dependent optimizations that are not possible at compile time. Commercial products such as VTune, Purify, and BoundsChecker use binary translation to provide programmers with detailed observations of their programs.

The Atom system provides a flexible mechanism for instrumenting Alpha executable object files and shared libraries with arbitrary C functions [70]. Atom has been used to build a myriad of analysis tools that trace procedure calls, profile instruction counts and memory referencing patterns, simulate memory system behavior, and isolate memory referencing errors. Etch [62] and EEL [43] provide roughly similar capabilities on different platforms. The Shade system uses binary translation for instruction profiling. [14]. Dynamo [2] and Dyninst [7] provide mechanisms for instrumenting and optimizing executables in memory, at run time. Smith and his colleagues have investigated binary translation for program profiling and optimization. [87].

Homework Problems

Homework Problem 7.6 [Category 1]:

Consider the following version of the `swap.c` function that counts the number of times it has been called.

```
 1 extern int buf[];
 2
 3 int *bufp0 = &buf[0];
 4 static int *bufp1;
 5
 6 static void incr()
 7 {
 8     static int count=0;
 9
10     count++;
11 }
12
13 void swap()
14 {
15     int temp;
16
17     incr();
18     bufp1 = &buf[1];
19     temp = *bufp0;
```

```
20      *bufp0 = *bufp1;
21      *bufp1 = temp;
22  }
```

For each symbol that is defined and referenced in `swap.o`, indicate if it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `main.o`), the symbol type (local, global, or extern) and the section (`.text`, `.data`, or `.bss`) it occupies in that module.

Symbol	`swap.o .symtab` entry?	Symbol type	Module where defined	Section
buf				
bufp0				
bufp1				
swap				
temp				
incr				
count				

Homework Problem 7.7 [Category 1]:

Without changing any variable names, modify `bar5.c` on Page 360 so that `foo5.c` prints the correct values of `x` and `y` (i.e., the hex representations of integers 15213 and 15212).

Homework Problem 7.8 [Category 1]:

In this problem, let `REF(x.i) --> DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example below, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (Rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (Rule 3), write "UNKNOWN".

A.
```
/* Module 1 */          /* Module 2 */
int main()              static int main=1;
{                       int p2()
}                       {
                        }
```

(a) `REF(main.1) --> DEF(_____.___)`

(b) `REF(main.2) --> DEF(_____.___)`

B.
```
/* Module 1 */          /* Module 2 */
int x;                  double x;
void main()             int p2()
{                       {
}                       }
```

(a) `REF(x.1) --> DEF(_____.___)`

(b) `REF(x.2) --> DEF(_____.___)`

C.
```
/* Module 1 */          /* Module 2 */
int x=1;                double x=1.0;
void main()             int p2()
{                       {
}                       }
```

(a) `REF(x.1) --> DEF(_____.___)`

(b) `REF(x.2) --> DEF(_____.___)`

Homework Problem 7.9 [Category 1]:

Consider the following program, which consists of two object modules:

```
1 /* foo6.c */
2 void p2(void);
3
4 int main()
5 {
6     p2();
7     return 0;
8 }
```

```
1 /* bar6.c */
2 #include <stdio.h>
3
4 char main;
5
6 void p2()
7 {
8     printf("0x%x\n", main);
9 }
```

When this program is compiled and executed on a Linux system, it prints the string "`0x55\n`" and terminates normally, even though `p2` never initializes variable `main`. Can you explain this?

Homework Problem 7.10 [Category 1]:

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of file object file and library arguments) that will allow the static linker to resolve all symbol references.

A. `p.o` → `libx.a` → `p.o`.

B. `p.o` → `libx.a` → `liby.a` **and** `liby.a` → `libx.a`.

C. `p.o` → `libx.a` → `liby.a` → `libz.a` **and** `liby.a` → `libx.a` → `libz.a`.

Homework Problem 7.11 [Category 1]:

The segment header in Figure 7.12 indicates that the data segment occupies `0x104` bytes in memory. However, only the first `0xe8` bytes of these come from the sections of the executable file. Why the discrepancy?

Homework Problem 7.12 [Category 2]:

The `swap` routine in Figure 7.10 contains five relocated references. For each relocated reference, give its line number in Figure 7.10, its run-time memory address, and its value. The original code and relocation entries in the `swap.o` module are shown in Figure 7.19.

Line # in Fig.7.10	Address	Value

```
1  00000000 <swap>:
2     0:   55                          push   %ebp
3     1:   8b 15 00 00 00 00           mov    0x0,%edx          get *bufp0=&buf[0]
4                                      3: R_386_32     bufp0    relocation entry
5     7:   a1 04 00 00 00              mov    0x4,%eax          get buf[1]
6                                      8: R_386_32     buf      relocation entry
7     c:   89 e5                       mov    %esp,%ebp
8     e:   c7 05 00 00 00 00 04        movl   $0x4,0x0          bufp1 = &buf[1];
9    15:   00 00 00
10                                     10: R_386_32    bufp1    relocation entry
11                                     14: R_386_32    buf      relocation entry
12    18:   89 ec                      mov    %ebp,%esp
13    1a:   8b 0a                      mov    (%edx),%ecx       temp = buf[0];
14    1c:   89 02                      mov    %eax,(%edx)       buf[0]=buf[1];
15    1e:   a1 00 00 00 00             mov    0x0,%eax          get *bufp1=&buf[1]
16                                     1f: R_386_32    bufp1    relocation entry
17    23:   89 08                      mov    %ecx,(%eax)       buf[1]=temp;
18    25:   5d                         pop    %ebp
19    26:   c3                         ret
```

Figure 7.19: **Code and relocation entries for Problem 7.13**

Homework Problem 7.13 [Category 3]:

Consider the C code and corresponding relocatable object module in Figure 7.20.

A. Determine which instructions in `.text` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

B. Determine which data objects in `.data` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

Feel free to use tools such as OBJDUMP to help you solve this problem.

Homework Problem 7.14 [Category 3]:

Consider the C code and corresponding relocatable object module in Figure 7.21.

A. Determine which instructions in `.text` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

B. Determine which data objects in `.rodata` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

Feel free to use tools such as OBJDUMP to help you solve this problem.

Homework Problem 7.15 [Category 3]:

Performing the following tasks will help you become more familiar with the various tools for manipulating object files.

A. How many object files are contained in the versions of `libc.a` and `libm.a` on your system?

B. Does `gcc -O2` produce different executable code than `gcc -O2 -g`?

C. What shared libraries does the GCC driver on your system use?

```
1 extern int p3(void);
2 int x = 1;
3 int *xp = &x;
4
5 void p2(int y) {
6 }
7
8 void p1() {
9     p2(*xp + p3());
10 }
```

(a) C code.

```
1 00000000 <p2>:
2    0:   55                      push   %ebp
3    1:   89 e5                   mov    %esp,%ebp
4    3:   89 ec                   mov    %ebp,%esp
5    5:   5d                      pop    %ebp
6    6:   c3                      ret

7 00000008 <p1>:
8    8:   55                      push   %ebp
9    9:   89 e5                   mov    %esp,%ebp
10   b:   83 ec 08                sub    $0x8,%esp
11   e:   83 c4 f4                add    $0xfffffff4,%esp
12  11:   e8 fc ff ff ff          call   12 <p1+0xa>
13  16:   89 c2                   mov    %eax,%edx
14  18:   a1 00 00 00 00          mov    0x0,%eax
15  1d:   03 10                   add    (%eax),%edx
16  1f:   52                      push   %edx
17  20:   e8 fc ff ff ff          call   21 <p1+0x19>
18  25:   89 ec                   mov    %ebp,%esp
19  27:   5d                      pop    %ebp
20  28:   c3                      ret
```

(b) `.text` section of relocatable object file.

```
1 00000000 <x>:
2    0:   01 00 00 00
3 00000004 <xp>:
4    4:   00 00 00 00
```

(c) `.data` section of relocatable object file.

Figure 7.20: **Example code for Problem 7.13.**

```
1 int relo3(int val)  {
2     switch (val) {
3     case 100:
4         return(val);
5     case 101:
6         return(val+1);
7     case 103: case 104:
8         return(val+3);
9     case 105:
10        return(val+5);
11    default:
12        return(val+6);
13    }
14 }
```

(a) C code.

```
1 00000000 <relo3>:
2    0:   55                      push   %ebp
3    1:   89 e5                   mov    %esp,%ebp
4    3:   8b 45 08                mov    0x8(%ebp),%eax
5    6:   8d 50 9c                lea    0xffffff9c(%eax),%edx
6    9:   83 fa 05                cmp    $0x5,%edx
7    c:   77 17                   ja     25 <relo3+0x25>
8    e:   ff 24 95 00 00 00 00    jmp    *0x0(,%edx,4)
9   15:   40                      inc    %eax
10  16:   eb 10                   jmp    28 <relo3+0x28>
11  18:   83 c0 03                add    $0x3,%eax
12  1b:   eb 0b                   jmp    28 <relo3+0x28>
13  1d:   8d 76 00                lea    0x0(%esi),%esi
14  20:   83 c0 05                add    $0x5,%eax
15  23:   eb 03                   jmp    28 <relo3+0x28>
16  25:   83 c0 06                add    $0x6,%eax
17  28:   89 ec                   mov    %ebp,%esp
18  2a:   5d                      pop    %ebp
19  2b:   c3                      ret
```

(b) `.text` section of relocatable object file.

```
    This is the jump table for the switch statement
1 0000 28000000 15000000 25000000 18000000  4 words at offsets 0x0,0x4,0x8, and 0xc
2 0010 18000000 20000000                    2 words at offsets 0x10 and 0x14
```

(c) `.rodata` section of relocatable object file.

Figure 7.21: **Example code for Problem 7.14.**

Chapter 8

Exceptional Control Flow

From the time you first apply power to a processor until the time you shut it off, the program counter assumes a sequence of values

$$a_0, a_1, \ldots, a_{n-1}.$$

where each a_k is the address of some corresponding instruction I_k. Each transition from a_k to a_{k+1} is called a *control transfer*. A sequence of such control transfers is called the *flow of control*, or *control flow* of the processor.

The simplest kind of control flow is a smooth sequence where each I_k and I_{k+1} are adjacent in memory. Typically, abrupt changes to this smooth flow, where I_{k+1} is not adjacent to I_k, are caused by familiar program instructions such as jumps, calls, and returns. Such instructions are necessary mechanisms that allow programs to react to changes in internal program state represented by program variables.

But systems must also be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program. For example, a hardware timer goes off at regular intervals and must be dealt with. Packets arrive at the network adapter and must be stored in memory. Programs request data from a disk and then sleep until they are notified that the data are ready. Parent processes that create child processes must be notified when their children terminate.

Modern systems react to these situations by making abrupt changes in the control flow. We refer to these abrupt changes in general as *exceptional control flow*. Exceptional control flow occurs at all levels of a computer system. For example, at the hardware level, events detected by the hardware trigger abrupt control transfers to exception handlers. At the operating systems level, the kernel transfers control from one user process to another via context switches. At the application level, a process can send a *Unix signal* to another process that abruptly transfers control to a signal handler in the recipient. An individual program can react to errors by sidestepping the usual stack discipline and making nonlocal jumps to arbitrary locations in other functions (similar to the *exceptions* supported by C++ and Java).

This chapter describes these various forms of exceptional control, and shows you how to use them in your C programs. The techniques you will learn about — creating processes, reaping terminated processes, sending and receiving signals, making non-local jumps — are the foundation of important programs such as Unix shells (Problem 8.20) and Web servers (Chapter 12).

8.1 Exceptions

Exceptions are a form of exceptional control flow that are implemented partly by the hardware and partly by the operating system. Because they are partly implemented in hardware, the details vary from system to system. However, the basic ideas are the same for every system. Our aim in this section is to give you a general understanding of exceptions and exception handling, and to help demystify what is often a confusing aspect of modern computer systems.

An *exception* is an abrupt change in the control flow in response to some change in the processor's state. Figure 8.1 shows the basic idea.

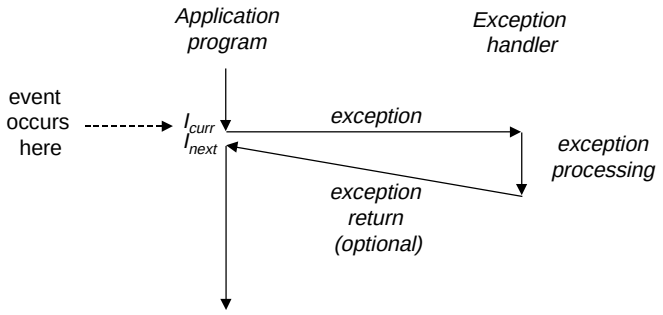


Figure 8.1: **Anatomy of an exception.** A change in the processor's state (event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

In the figure, the processor is executing some current instruction I_{curr} when a significant change in the processor's *state* occurs. The state is encoded in various bits and signals inside the processor. The change in state is known as an *event*. The event might be directly related to the execution of the current instruction. For example, a virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. On the other hand, the event might be unrelated to the execution of the current instruction. For example, a system timer goes off or an I/O request completes.

In any case, when the processor detects that the event has occurred, it makes an indirect procedure call (the exception), through a jump table called an *exception table*, to an operating system subroutine (the *exception handler*) that is specifically designed to process this particular kind of event.

When the exception handler finishes processing, one of three things happens, depending on the type of event that caused the exception:

1. The handler returns control to the current instruction I_{curr}, the instruction that was executing when the event occurred.
2. The handler returns control to I_{next}, the instruction that would have executed next had the exception not occurred.
3. The handler aborts the interrupted program.

Section 8.1.2 says more about these possibilities.

8.1.1 Exception Handling

Exceptions can be difficult to understand because handling them involves close cooperation between hardware and software. It is easy to get confused about which component performs which task. Let's look at the division of labor between hardware and software in more detail.

Each type of possible exception in a system is assigned a unique non-negative integer *exception number*. Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system *kernel* (the memory-resident part of the operating system). Examples of the former include divide by zero, page faults, memory access violations, breakpoints, and arithmetic overflows. Examples of the latter include system calls and signals from external I/O devices.

At system boot time (when the computer is reset or powered on) the operating system allocates and initializes a jump table called an *exception table*, so that entry k contains the address of the handler for exception k. Figure 8.2 shows the format of an exception table.

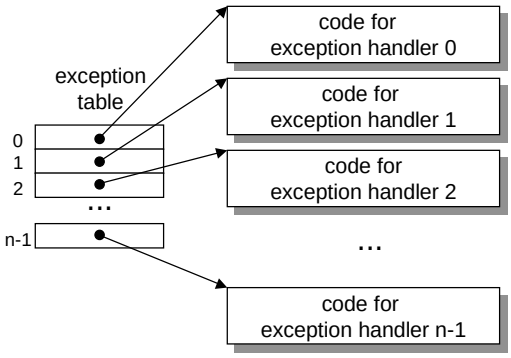


Figure 8.2: **Exception table.** The exception table is a jump table where entry k contains the address of the handler code for exception k.

At runtime (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an indirect procedure call, through entry k of the exception table, to the corresponding handler. Figure 8.3 shows how the processor uses the exception table to form the address of the appropriate exception handler. The exception number is an index into the exception table, whose starting address is contained in a special CPU register called the *exception table base register*.

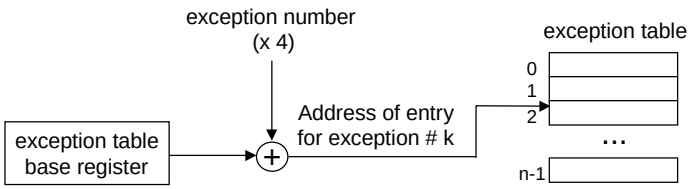


Figure 8.3: **Generating the address of an exception handler.** The exception number is an index into the exception table.

An exception is akin to a procedure call, but with some important differences.

- As with a procedure call, the processor pushes a return address on the stack before branching to the handler. However, depending on the class of exception, the return address is either the current instruction (the instruction that was executing when the event occurred) or the next instruction (the instruction that would have executed after the current instruction had the event not occurred).
- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns. For example, an IA32 system pushes the EFLAGS register containing, among other things, the current condition codes, onto the stack.
- If control is being transferred from a user program to the kernel, all of the above items are pushed on the kernel's stack rather than the user's stack.
- Exception handlers run in *kernel mode* (Section 8.2.3, which means they have complete access to all system resources.

Once the hardware triggers the exception, the rest of the work is done in software by the exception handler. After the handler has processed the event, it optionally returns to the interrupted program by executing a special "return from interrupt" instruction, which pops the appropriate state back into the processor's control and data registers, restores the state to *user mode* (Section 8.2.3) if the exeption interrupted a user program, and then returns control to the interrupted program.

8.1.2 Classes of Exceptions

Exceptions can be divided into four classes: *interrupts*, *traps*, *faults*, and *aborts*. Figure 8.4 summarizes the attributes of these classes.

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4: **Classes of exceptions.** Asynchronous exceptions occur as a result of events external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

Interrupts

Interrupts occur *asynchronously* as a result of signals from I/O devices that are external to the processor. Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction. Exception handlers for hardware interrupts are often called *interrupt handlers*.

Figure 8.5 summarizes the processing for an interrupt. I/O devices such as network adapters, disk controllers, and timer chips trigger interrupts by signalling a pin on the processor chip and placing the exception number on the system bus that identifies the device that caused the interrupt.

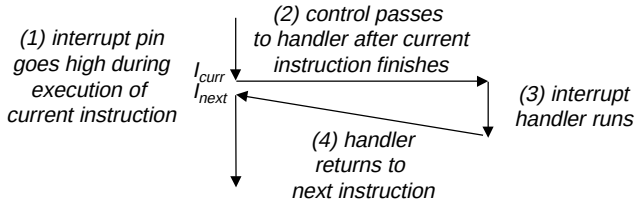


Figure 8.5: **Interrupt handling.** The interrupt handler returns control to the next instruction in the application program's control flow.

After the current instruction finishes executing, the processor notices that the interrupt pin has gone high, reads the exception number from the system bus, and then calls the appropriate interrupt handler. When the handler returns, it returns control to the next instruction (i.e., the instruction that would have followed the current instruction in the control flow had the interrupt not occurred). The effect is that the program continues executing as though the interrupt had never happened.

The remaining classes of exceptions (traps, faults, and aborts) occur *synchronously* as a result of executing the current instruction. We refer to this instruction as the *faulting instruction*.

Traps

Traps are *intentional* exceptions that occur as a result of executing an instruction. Like interrupt handlers, trap handlers return control to the next instruction. The most important use of traps is to provide a procedure-like interface between user programs and the kernel known as a *system call*.

User programs often need to request services from the kernel such as reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), or terminating the current process (`exit`). To allow controlled access to such kernel services, processors provide a special "`syscall` n" instruction that user programs can execute when they want to request service n. Executing the `syscall` instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine. Figure 8.6 summarizes the processing for a system call. From a programmer's perspective, a system call is identical

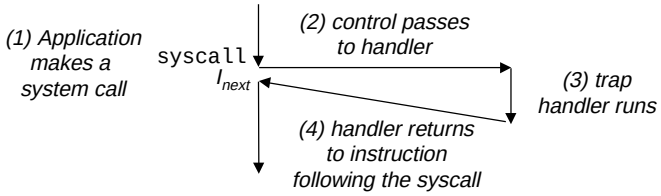


Figure 8.6: **Trap handling.** The trap handler returns control to the next instruction in the application program's control flow.

to a regular function call. However, their implementations are quite different. Regular functions run in

user mode, which restricts the types of instructions they can execute, and they access the same stack as the calling function. A system call runs in *kernel mode*, which allows it to execute instructions, and accesses a stack defined in the kernel. Section 8.2.3 discusses user and kernel modes in more detail.

Faults

Faults result from error conditions that a handler might be able to correct. When a fault occurs, the processor transfers control to the fault handler. If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby reexecuting it. Otherwise, the handler returns to an `abort` routine in the kernel that terminates the application program that caused the fault. Figure 8.7 summarizes the processing for a fault.

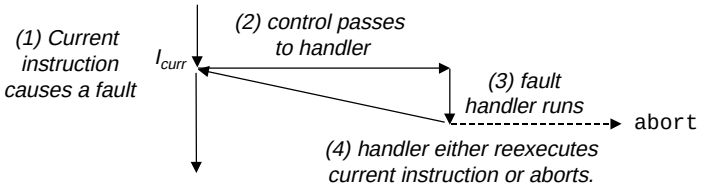


Figure 8.7: **Fault handling.** Depending on the whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

A classic example of a fault is the page fault exception, which occurs when an instruction references a virtual address whose corresponding physical page is not resident in memory and must be retrieved from disk. As we will see in Chapter 10, a page is contiguous block (typically 4 KB) of virtual memory. The page fault handler loads the appropriate page from disk and then returns control to the instruction that caused the fault. When the instruction executes again, the appropriate physical page is resident in memory and the instruction is able to run to completion without faulting.

Aborts

Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program. As shown in Figure 8.8, the handler returns control to an `abort` routine that terminates the application program.

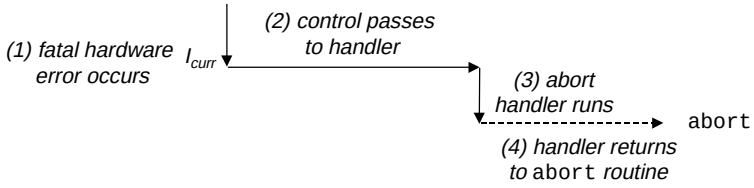


Figure 8.8: **Abort handling.** The abort handler passes control to a kernel `abort` routine that terminates the application program.

8.1.3 Exceptions in Intel Processors

To help make things more concrete, let's look at some of the exceptions defined for Intel systems. A Pentium system can have up to 256 different exception types. Numbers in the range 0 to 31 correspond to exceptions that are defined by the Pentium architecture, and thus are identical for any Pentium-class system. Numbers in the range 32 to 255 correspond to interrupts and traps that are defined by the operating system. Figure 8.9 shows a few examples.

Exception Number	Description	Exception Class
0	divide error	fault
13	general protection fault	fault
14	page fault	fault
18	machine check	abort
32–127	OS-defined exceptions	interrupt or trap
128 (`0x80`)	system call	trap
129–255	OS-defined exceptions	interrupt or trap

Figure 8.9: **Examples of exceptions in Pentium systems.**

A divide error (exception 0) occurs when an application attempts to divide by zero, or when the result of a divide instruction is too big for the destination operand. Unix does not attempt to recover from divide errors, opting instead to abort the program. Unix shells typically report divide errors as "Floating exceptions".

The infamous general protection fault (exception 13) occurs for many reasons, usually because a program references an undefined area of virtual memory, or because the program attempts to write to a read-only text segment. Unix does not attempt to recover from this fault. Unix shells typically report general protection faults as "Segmentation faults".

A page fault (exception 14) is an example of an exception where the faulting instruction is restarted. The handler maps the appropriate page of physical memory on disk into a page of virtual memory, and then restarts the faulting instruction. We will see how this works in detail in Chapter 10.

A machine check (exception 18) occurs as a result of a fatal hardware error that is detected during the execution of the faulting instruction. Machine check handlers never return control to the application program.

System calls are provided on IA32 systems via a trapping instruction called `INT` n, where n can be the index of any of the 256 entries in the exception table. Historically, systems calls are provided through exception 128 (`0x80`).

Aside: A note on terminology.
The terminology for the various classes of exceptions varies from system to system. Processor macro-architecture specifications often distinguish between asynchronous "interrupts" and synchronous "exceptions", yet provide no umbrella term to refer to these very similar concepts. To avoid having to constantly refer to "exceptions and interrupts" and "exceptions or interrupts", we use the word "exception" as the general term and distinguish between asynchronous exceptions (interrupts) and synchronous exceptions (traps, faults, and aborts) only when it is appropriate. As we have noted, the basic ideas are the same for every system, but you should be aware that some manufacturers' manuals use the word "exception" to refer only to those changes in control flow caused by synchronous events. **End Aside.**

8.2 Processes

Exceptions provide the basic building blocks that allow the operating system to provide the notion of a *process*, one of the most profound and successful ideas in computer science.

When we run a program on a modern system, we are presented with the illusion that our program is the only one currently running in the system. Our program appears to have exclusive use of both the processor and the memory. The processor appears to execute the instructions in our program, one after the other, without interruption. And the code and data of our program appear to be the only objects in the system's memory. These illusions are provided to us by the notion of a process.

The classic definition of a process is *an instance of a program in execution*. Each program in the system runs in the *context* of some process. The context consists of the state that the program needs to run correctly. This state includes the program's code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors.

Each time a user runs a program by typing the name of an executable object file to the shell, the shell creates a new process and then runs the executable object file in the context of this new process. Application programs can also create new processes and run either their own code or other applications in the context of the new process.

A detailed discussion of how operating systems implement processes is beyond our scope. Instead we will focus on the key abstractions that a process provides to the application:

- An independent *logical control flow* that provides the illusion that our program has exclusive use of the processor.
- A private address space that provides the illusion that our program has exclusive use of the memory system.

Let's look more closely at these abstractions.

8.2.1 Logical Control Flow

A process provides each program with the illusion that it has exclusive use of the processor, even though many other programs are typically running on the system. If we were to use a debugger to single step the execution of our program, we would observe a series of program counter (PC) values that corresponded exclusively to instructions contained in our program's executable object file or in shared objects linked into our program dynamically at run time. This sequence of PC values is known as a *logical control flow*.

Consider a system that runs three processes, as shown in Figure 8.10. The single physical control flow of the processor is partitioned into three *logical flows*, one for each process. Each vertical line represents a portion of the logical flow for a process. In the example, process A runs for a while, followed by B, which runs to completion. Then C runs for awhile, followed by A, which runs to completion. Finally, C is able to run to completion.

The key point in Figure 8.10 is that processes take turns using the processor. Each process executes a portion of its flow and then is *preempted* (temporarily suspended) while other processes take their turns. To

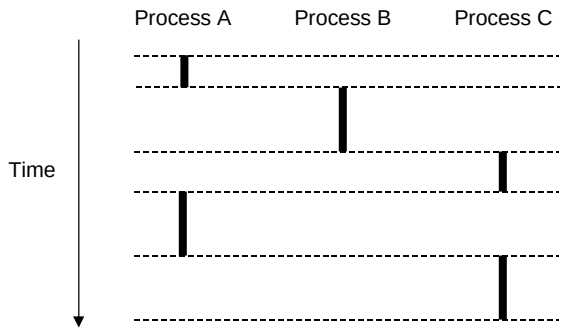


Figure 8.10: **Logical control flows.** Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.

a program running in the context of one of these processes, it appears to have exclusive use of the processor. The only evidence to the contrary is that if we were to precisely measure the elapsed time of each instruction (see Chapter 9), we would notice that the CPU appears to periodically stall between the execution of some of the instructions in our program. However, each time the processor stalls, it subsequently resumes execution of our program without any change to the contents of the program's memory locations or registers.

In general, each logical flow is independent of any other flow in the sense that the logical flows associated with different processes do not affect the states of any other processes. The only exception to this rule occurs when processes use interprocess communication (IPC) mechanisms such as pipes, sockets, shared memory, and semaphores to explicitly interact with each other.

Any process whose logical flow overlaps in time with another flow is called a *concurrent process*, and the two processes are said to run *concurrently*. For example, in Figure 8.10, processes A and B run concurrently, as do A and C. On the other hand, B and C do not run concurrently because the last instruction of B executes before the first instruction of C.

The notion of processes taking turns with other processes is known as *multitasking*. Each time period that a process executes a portion of its flow is called a *time slice*. Thus, multitasking is also referred to as *time slicing*.

8.2.2 Private Address Space

A process also provides each program with the illusion that it has exclusive use of the system's address space. On a machine with n-bit addresses, the *address space* is the set of 2^n possible addresses, $0, 1, \ldots, 2^n - 1$. A process provides each program with its own *private address space*. This space is private in the sense that a byte of memory associated with a particular address in the space cannot in general be read or written by any other process.

Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.11 shows the organization of the address space for a Linux process. The bottom three-fourths of the address space is reserved for the user program, with the usual text, data, heap, and stack segments. The top quarter of the address space is reserved for the kernel. This portion of the address space contains the code, data, and stack that the kernel

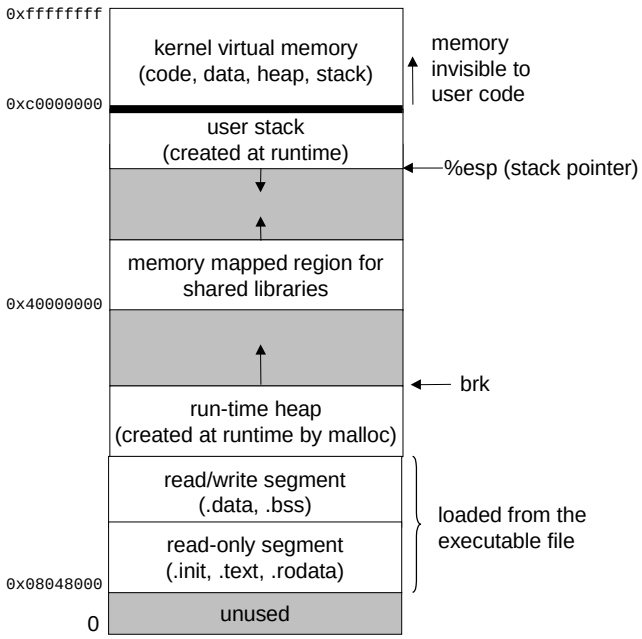


Figure 8.11: **Process address space.**

uses when it executes instructions on behalf of the process (e.g., when the application program executes a system call).

8.2.3 User and Kernel Modes

In order for the operating system kernel to provide an airtight process abstraction, the processor must provide a mechanism that restricts the instructions that an application can execute, as well as the portions of the address space that it can access.

Processors typically provide this capability with a *mode bit* in some control register that characterizes the privileges that the process currently enjoys. When the mode bit is set, the process is running in *kernel mode* (sometimes called *supervisor mode*). A process running in kernel mode can execute any instruction in the instruction set and access any memory location in the system.

When the mode bit is not set, the process is running in *user mode*. A process in user mode is not allowed to execute *privileged instructions* that do things such as halt the processor, change the mode bit, or initiate an I/O operation. Nor is it allowed to directly reference code or data in the kernel area of the address space. Any such attempt results in a fatal protection fault. Instead, user programs must access kernel code and data indirectly via the system call interface.

A process running application code is initially in user mode. The only way for the process to change from user mode to kernel mode is via an exception such as an interrupt, a fault, or a trapping system call. When the exception occurs, and control passes to the exception handler, the processor changes the mode from user mode to kernel mode. The handler runs in kernel mode. When it returns to the application code, the processor changes the mode from kernel mode back to user mode.

Linux and Solaris provides a clever mechanism, called the `/proc` filesystem, that allows user mode processes to access the contents of kernel data structures. The `/proc` filesystem exports the contents of many kernel data structures as a hierarchy of ASCII files that can read by user programs. For example, you can use the Linux `proc` filesystem to find out general system attributes such as CPU type (`/proc/cpuinfo`), or the memory segments used by a particular process (`/proc/<process id>/maps`).

8.2.4 Context Switches

The operating system kernel implements multitasking using a higher-level form of exceptional control flow known as a *context switch*. The context switch mechanism is built on top of the lower-level exception mechanism that we discussed in Section 8.1.

The kernel maintains a *context* for each process. The context is the state that the kernel needs to restart a preempted process. It consists of the values of objects such as the general-purpose registers, the floating-point registers, the program counter, user's stack, status registers, kernel's stack, and various kernel data structures such as a *page table* that characterizes the address space, a *process table* that contains information about the current process, and a *file table* that contains information about the files that the process has opened.

At certain points during the execution of a process, the kernel can decide to preempt the current process and restart a previously preempted process. This decision is known as *scheduling*, and is handled by a part of the kernel called the *scheduler*. When the kernel selects a new process to run, we say that the kernel has *scheduled* that process. After the kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a mechanism called a *context switch* that (1) saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.

A context switch can occur while the kernel is executing a system call on behalf of the user. If the system call blocks because it is waiting for some event to occur, then the kernel can put the current process to sleep and switch to another process. For example, if a `read` system call requires a disk access, the kernel can opt to perform a context switch and run another process instead of waiting for the data to arrive from the disk. Another example is the `sleep` system call, which is an explicit request to put the calling process to sleep. In general, even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process.

A context switch can also occur as a result of an interrupt. For example, all systems have some mechanism for generating periodic timer interrupts, typically every 1 ms or 10 ms. Each time a timer interrupt occurs, the kernel can decide that the current process has run long enough and switch to a new process.

Figure 8.12 shows an example of context switching between a pair of processes A and B. In this example, initially process A is running in user mode until it traps to the kernel by executing a `read` system call. The trap handler in the kernel requests a DMA transfer from the disk controller and arranges for the disk to interrupt the processor after the disk controller has finished transferring the data from disk to memory.

The disk will take a relatively long time to fetch the data (on the order of tens of milliseconds), so instead of waiting and doing nothing in the interim, the kernel performs a context switch from process A to B. Note that before the switch, the kernel is executing instructions in user mode on behalf of process A. During the

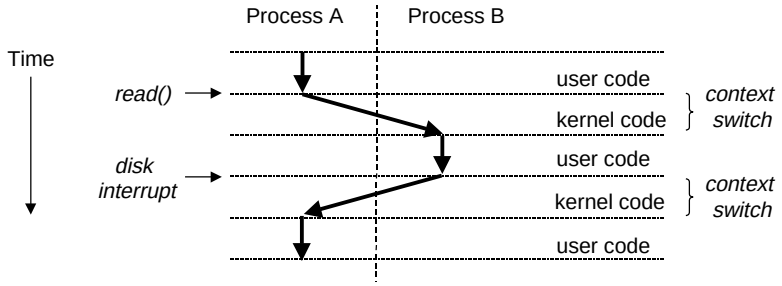


Figure 8.12: **Anatomy of a context switch.**

first part of the switch, the kernel is executing instructions in kernel mode on behalf of process A. Then at some point it begins executing instructions (still in kernel mode) on behalf of process B. And after the switch, the kernel is executing instructions in user mode on behalf of process B.

Process B then runs for a while in user mode until the disk sends an interrupt to signal that data has been transferred from disk to memory. The kernel decides that process B has run long enough and performs a context switch from process B to A, returning control in process A to the instruction immediately following the `read` system call. Process A continues to run until the next exception occurs, and so on.

8.3 System Calls and Error Handling

Unix systems provide a variety of systems calls that application programs use when they want to request services from the kernel such as reading a file or creating a new process. For example, Linux provides about 160 system calls. Typing "`man syscalls`" will give you the complete list.

C programs can invoke any system call directly by using the `_syscall` macro described in "`man 2 intro`". However, it is usually neither necessary nor desirable to invoke system calls directly. The standard C library provides a set of convenient wrapper functions for the most frequently used system calls. The wrapper functions package up the arguments, trap to the kernel with the appropriate system call, and then pass the return status of the system call back to the calling program. In our discussion in the following sections, we will refer to system calls and their associated wrapper functions interchangeably as *system-level functions*.

When Unix system-level functions encounter an error, they typically return -1 and set the global integer variable `errno` to indicate what went wrong. Programmers should *always* check for errors, but unfortunately, many skip error checking because it bloats the code and makes it harder to read. For example, here is how we might check for errors when we call the Unix `fork` function:

```
1     if ((pid = fork()) < 0) {
2         fprintf(stderr, "fork error: %s\n", strerror(errno));
3         exit(0);
4     }
```

The `strerror` function returns a text string that describes the error associated with a particular value of `errno`. We can simplify this code somewhat by defining the following *error-reporting function*:

```
1 void unix_error(char *msg) /* unix-style error */
2 {
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4     exit(0);
5 }
```

Given this function, our call to `fork` reduces from four lines to two lines:

```
1     if ((pid = fork()) < 0)
2         unix_error("fork error");
```

We can simplify our code even further by using a *error-handling wrappers*. For a given base function `foo`, we define a wrapper function `Foo` with identical arguments, but with the first letter of the name capitalized. The wrapper calls the base function, checks for errors and terminates if there are any problems. For example, here is the error-handling wrapper for the `fork` function:

```
1 pid_t Fork(void)
2 {
3     pid_t pid;
4
5     if ((pid = fork()) < 0)
6         unix_error("Fork error");
7     return pid;
8 }
```

Given this wrapper, our call to `fork` shrinks to a single compact line:

```
1     pid = Fork();
```

We will use error-handling wrappers throughout the remainder of this book. They allow us to keep our code examples concise, without giving you the mistaken impression that it is permissible to ignore error-checking. Note that when we discuss system-level functions in the text, we will always refer to them by their lower-case base names, rather than by their upper-case wrapper names.

See Appendix A for a discussion of Unix error-handling and the error-handling wrappers used throughout the book. The wrappers are defined in a file called `csapp.c` and their prototypes are defined in a header file called `csapp.h`. For your reference, Appendix A provides the sources for these files.

8.4 Process Control

Unix provides a number of system calls for manipulating processes from C programs. This section describes the important functions and gives examples of how they are used.

8.4.1 Obtaining Process ID's

Each process has a unique positive (non-zero) *process ID (PID)*. The `getpid` function returns the PID of the calling process. The `getppid` function returns the PID of its *parent* (i.e., the process that created the calling process).

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```

returns: PID of either the caller or the parent

The `getpid` and `getppid` routines return an integer value of type `pid_t`, which on Linux systems is defined in `types.h` as an `int`.

8.4.2 Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states:

- *Running.* The process is either executing on the CPU, or is waiting to be executed and will eventually be scheduled.
- *Stopped.* The execution of the process is *suspended* and will not be scheduled. A process stops as a result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, and it remains stopped until it receives a SIGCONT signal, at which point is becomes running again. (A *signal* is a form of software interrupt that is described in detail in Section 8.5.)
- *Terminated.* The process is stopped permanently. A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process; (2) returning from the main routine; or (3) calling the `exit` function:

```
#include <stdlib.h>

void exit(int status);
```

this function does not return

The `exit` function terminates the process with an *exit status* of `status`. (The other way to set the exit status is to return an integer value from the main routine.)

A *parent process* creates a new running *child process* by calling the `fork` function.

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

returns: 0 to child, PID of child to parent, -1 on error

The newly created child process is almost, but not quite, identical to the parent. The child gets an identical (but separate) copy of the parent's user-level virtual address space, including the text, data, and bss segments, heap, and user stack. The child also gets identical copies of any of the parent's open file descriptors, which means the child can read and write any files that were open in the parent when it called `fork`. The most significant difference between the parent and the newly created child is that they have different PIDs.

The `fork` function is interesting (and often confusing) because it is called *once* but it returns *twice*: once in the calling process (the parent), and once in the newly created child process. In the parent, `fork` returns the PID of the child. In the child, `fork` returns a value of 0. Since the PID of the child is always nonzero, the return value provides an unambiguous way to tell whether the program is executing in the parent or the child.

Figure 8.13 shows a simple example of a parent process that uses `fork` to create a child process. When the `fork` call returns in line 8, `x` has a value of 1 in both the parent and child. The child increments and prints its copy of `x` in line 10. Similarly, the parent decrements and prints its copy of `x` in line 15.

code/ecf/fork.c

```
 1 #include "csapp.h"
 2
 3 int main()
 4 {
 5     pid_t pid;
 6     int x = 1;
 7
 8     pid = Fork();
 9     if (pid == 0) {  /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

code/ecf/fork.c

Figure 8.13: **Using `fork` to create a new process.**

When we run the program on our Unix system, we get the following result:

```
unix> ./fork
parent: x=0
child : x=2
```

There are some subtle aspects to this simple example.

- *Call once, return twice.* The `fork` function is called once by the parent, but it returns twice: once to the parent and once to the newly created child. This is fairly straightforward for programs that create a single child. But programs with multiple instances of `fork` can be confusing and need to be reasoned about carefully.

- *Concurrent execution.* The parent and the child are separate processes that run concurrently. The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way. When we run the program on our system, the parent process completes its `printf` statement first, followed by the child. However, on another system the reverse might be true. In general, as programmers we can never make assumptions about the interleaving of the instructions in different processes.

- *Duplicate but separate address spaces.* If we could halt both the parent and the child immediately after the `fork` function returned in each process, we would see that the address space of each process is identical. Each process has the same user stack, the same local variable values, the same heap, the same global variable values, and the same code. Thus, in our example program, local variable `x` has a value of 1 in both the parent and the child when the `fork` function returns in line 8. However, since the parent and the child are separate processes, they each have their own private address spaces. Any subsequent changes that a parent or child makes to `x` are private and are not reflected in the memory of the other process. This is why the variable `x` has different values in the parent and child when they call their respective `printf` statements.

- *Shared files.* When we run the example program, we notice that both parent and child print their output on the screen. The reason is that the child inherits all of the parent's open files. When the parent calls `fork`, the `stdout` file is open and directed to the screen. The child inherits this file and thus its output is also directed to the screen.

When you are first learning about the `fork` function, it is often helpful to draw a picture of the *process hierarchy*. The process hierarchy is a labeled directed graph, where each node is a process and each directed arc $a \xrightarrow{k} b$ denotes that a is the parent of b and that a created b by executing the kth lexical instance of the `fork` function in the source code.

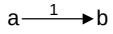
For example, how many lines of output would the program in Figure 8.14(a) generate? Figure 8.14(b) shows the corresponding process hierarchy. The parent a creates the child b when it executes the first (and only) `fork` in the program. Both a and b call `printf` once, so the program prints two output lines.

Now what if we were to call `fork` twice, as shown in Figure 8.14(c)? As we see from the process hierarchy in Figure 8.14(d), the parent a creates child b when it calls the first `fork` function. Then both a and b execute the second `fork` function, which results in the creations of c and d, for a total of four processes. Each process calls `printf`, so the program generates four output lines.

Continuing this line of thought, what would happen if we were to call `fork` three times, as in Figure 8.14(e)? As we see from the process hierarchy in Figure 8.14(f), the first `fork` creates one process, the second `fork`

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     printf("hello!\n");
7     exit(0);
8 }
```

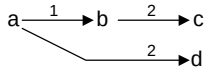
(a) Calls `fork` once.



(b) Prints two output lines.

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     printf("hello!\n");
8     exit(0);
9 }
```

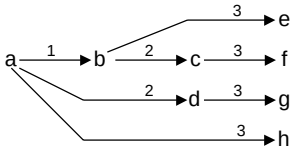
(c) Calls `fork` twice.



(d) Prints four output lines.

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     Fork();
8     printf("hello!\n");
9     exit(0);
10 }
```

(e) Calls `fork` three times.



(f) Prints eight output lines.

Figure 8.14: **Examples of programs and their process hierarchies.**

creates two processes, and the third `fork` creates four processes, for a total of eight processes. Each process calls `printf`, so the program produces eight output lines.

Practice Problem 8.1:

Consider the following program:

code/ecf/forkprob0.c

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int x = 1;
6
7     if (Fork() == 0)
8         printf("printf1: x=%d\n", ++x);
9     printf("printf2: x=%d\n", --x);
10    exit(0);
11 }
```

code/ecf/forkprob0.c

A. What is the output of the child process?

B. What is the output of the parent process?

Practice Problem 8.2:

How many “hello” output lines does this program print?

code/ecf/forkprob1.c

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int i;
6
7     for (i = 0; i < 2; i++)
8         Fork();
9     printf("hello!\n");
10    exit(0);
11 }
```

code/ecf/forkprob1.c

Practice Problem 8.3:

How many “hello” output lines does this program print?

code/ecf/forkprob4.c

```
 1 #include "csapp.h"
 2
 3 void doit()
 4 {
 5     Fork();
 6     Fork();
 7     printf("hello\n");
 8     return;
 9 }
10
11 int main()
12 {
13     doit();
14     printf("hello\n");
15     exit(0);
16 }
```

code/ecf/forkprob4.c

8.4.3 Reaping Child Processes

When a process terminates for any reason, the kernel does not remove it from the system immediately. Instead, the process is kept around in a terminated state until it is *reaped* by its parent. When the parent reaps the terminated child, the kernel passes the child's exit status to the parent, and then discards the terminated process, at which point it ceases to exist. A terminated process that has not yet been reaped is called a *zombie*.

> **Aside: Why are terminated children called zombies?**
> In folklore, a zombie is a living corpse, an entity that is half-alive and half-dead. A zombie process is similar in the sense that while it has already terminated, the kernel maintains some of its state until it can be reaped by the parent.
> **End Aside.**

If the parent process terminates without reaping its zombie children, the kernel arranges for the `init` process to reap them. The `init` process has a PID of 1 and is created by the kernel during system initialization. Long-running programs such as shells or servers should always reap their zombie children. Even though zombies are not running, they still consume system memory resources.

A process waits for its children to terminate or stop by calling the `waitpid` function.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

returns: PID of child if OK, 0 (if WNOHANG) or -1 on error

The `waitpid` function is complicated. By default (when `options = 0`), `waitpid` suspends execution of the calling process until a child process in its *wait set* terminates. If a process in the wait set has already

terminated at the time of the call, then `waitpid` returns immediately. In either case, `waitpid` returns the PID of the terminated child that caused `waitpid` to return, and the terminated child is removed from the system.

Determining the Members of the Wait Set

The members of the wait set are determined by the `pid` argument:

- If `pid > 0`, then the wait set is the singleton child process whose process ID is equal to `pid`.
- If `pid = -1`, then the wait set consists of all of the parent's child processes.

Aside: Waiting on sets of processes.
The `waitpid` function also supports other kinds of wait sets, involving Unix process groups, that we will not discuss. **End Aside.**

Modifying the Default Behavior

The default behavior can be modified by setting `options` to various combinations of the WNOHANG and WUNTRACED constants:

- WNOHANG: Return immediately (with a return value of 0) if the none of the child processes in the wait set has terminated yet.
- WUNTRACED: Suspend execution of the calling process until a process in the wait set becomes terminated or stopped. Return the PID of the terminated or stopped child that caused the return.
- WNOHANG | WUNTRACED : Suspend execution of the calling process until a child in the wait set terminates or stops, and then return the PID of the stopped or terminated child that caused the return. Also, return immediately (with a return value of 0) if none of the processes in the wait set is terminated or stopped.

Checking the Exit Status of a Reaped Child

If the `status` argument is non-NULL, then `waitpid` encodes status information about the child that caused the return in the `status` argument. The `wait.h` include file defines several macros for interpreting the `status` argument:

- WIFEXITED(status): Returns true if the child terminated normally, via a call to `exit` or a return.
- WEXITSTATUS(status): Returns the exit status of a normally terminated child. This status is only defined if WIFEXITED returned true.
- WIFSIGNALED(status): Returns true if the child process terminated because of a signal that was not caught. (Signals are explained in Section 8.5.)

- WTERMSIG(status): Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED(status) returned true.
- WIFSTOPPED(status): Returns true if the child that caused the return is currently stopped.
- WSTOPSIG(status): Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED(status) returned true.

Error Conditions

If the calling process has no children, then `waitpid` returns -1 and sets `errno` to ECHILD. If the `waitpid` function was interrupted by a signal, then it returns -1 and sets `errno` to EINTR.

Aside: Constants associated with Unix functions.
Constants such as WNOHANG and WUNTRACED are defined by system header files. For example, WNOHANG and WUNTRACED are defined (indirectly) by the `wait.h` header file:

```
/* Bits in the third argument to 'waitpid'. */
#define WNOHANG    1   /* Don't block waiting. */
#define WUNTRACED  2   /* Report status of stopped children. */
```

In order to use these constants, you must include the `wait.h` header file in your code:

```
#include <sys/wait.h>
```

The `man` page for each Unix function lists the header files to include whenever you use that function in your code. Also, in order to check return codes such as ECHILD and EINTR, you must include `errno.h`. To simplify our code examples, we include a single header file called `csapp.h` that includes the header files for all of the functions used in the book. The `csapp.h` header file is listed in Appendix A. **End Aside.**

Examples

Figure 8.15 shows a program that creates N children, uses `waitpid` to wait for them to terminate, and then checks the exit status of each terminated child. When we run the program on our Unix system, it produces the following output:

```
unix> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

Notice that the program reaps the children in no particular order. Figure 8.16 shows how we might use `waitpid` to reap the children from Figure 8.15 in the same order that they were created by the parent.

Practice Problem 8.4:

Consider the following program:

code/ecf/waitprob1.c

code/ecf/waitpid1.c

```
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid;

    for (i = 0; i < N; i++)
        if ((pid = Fork()) == 0)  /* child */
            exit(100+i);

    /* parent waits for all of its children to terminate */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                   pid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", pid);
    }
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

code/ecf/waitpid1.c

Figure 8.15: **Using the `waitpid` function to reap zombie children.**

code/ecf/waitpid2.c

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
7     pid_t pid[N+1], retpid;
8
9     for (i = 0; i < N; i++)
10        if ((pid[i] = Fork()) == 0)  /* child */
11            exit(100+i);
12
13    /* parent reaps N children in order */
14    i = 0;
15    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
16        if (WIFEXITED(status))
17            printf("child %d terminated normally with exit status=%d\n",
18                   retpid, WEXITSTATUS(status));
19        else
20            printf("child %d terminated abnormally\n", retpid);
21    }
22
23    /* The only normal termination is if there are no more children */
24    if (errno != ECHILD)
25        unix_error("waitpid error");
26
27    exit(0);
28 }
```

code/ecf/waitpid2.c

Figure 8.16: **Using `waitpid` to reap zombie children in the order they were created.**

```
 1 #include "csapp.h"
 2
 3 int main()
 4 {
 5     int status;
 6     pid_t pid;
 7
 8     printf("Hello\n");
 9     pid = Fork();
10     printf("%d\n", !pid);
11     if (pid != 0) {
12         if (waitpid(-1, &status, 0) > 0) {
13             if (WIFEXITED(status) != 0)
14                 printf("%d\n", WEXITSTATUS(status));
15         }
16     }
17     printf("Bye\n");
18     exit(2);
19 }
```

code/ecf/waitprob1.c

A. How many output lines does this program generate?

B. What is one possible ordering of these output lines?

8.4.4 Putting Processes to Sleep

The `sleep` function suspends a process for some period of time.

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

returns: seconds left to sleep

`Sleep` returns zero if the requested amount of time has elapsed, and the number of seconds still left to sleep otherwise. The latter case is possible if the `sleep` function returns prematurely because it was interrupted by a *signal*. We will discuss signals in detail in Section 8.5.

Another function that we will find useful is the `pause` function, which puts the calling function to sleep until a signal is received by the process.

```
#include <unistd.h>

int pause(void);
```

always returns -1

Practice Problem 8.5:

Write a wrapper function for `sleep`, called `snooze`, with the following interface:

```
unsigned int snooze(unsigned int secs);
```

The `snooze` function behaves exactly as the `sleep` function, except that it prints a message describing how long the process actually slept. For example,

```
Slept for 4 of 5 secs.
```

8.4.5 Loading and Running Programs

The `execve` function loads and runs a new program in the context of the current process.

```
#include <unistd.h>

int execve(char *filename, char *argv[], char *envp);
```

does not return if OK, returns -1 on error

The `execve` function loads and runs the executable object file `filename` with the argument list `argv` and the environment variable list `envp`. `Execve` returns to the calling program only if there is an error such as not being able to find `filename`. So unlike `fork`, which is called once but returns twice, `execve` is called once and never returns.

The argument list is represented by the data structure shown in Figure 8.17. The `argv` variable points to

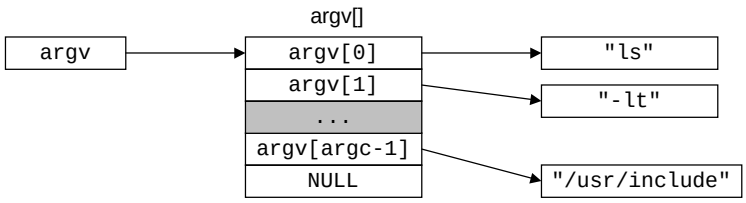


Figure 8.17: **Organization of an argument list.**

a null-terminated array of pointers, each of which points to an argument string. By convention `argv[0]` is the name of the executable object file. The list of environment variables is represented by a similar data structure, shown in Figure 8.18. The `envp` variable points to a null-terminated array of pointers to environment variable strings, each of which is a name-value pair of the form "`NAME=VALUE`".

After `execve` loads `filename`, it calls the startup code described in Section 7.9. The startup code sets up the stack and passes control to the main routine of the new program, which has a prototype of the form

```
int main(int argc, char **argv, char **envp);
```

or equivalently

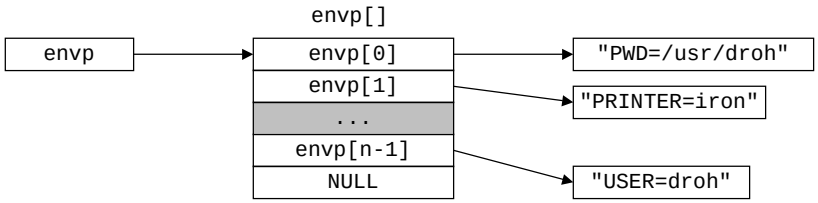


Figure 8.18: **Organization of an environment variable list.**

```
int main(int argc, char *argv[], char *envp[]);
```

When `main` begins executing on a Linux system, the user stack has the organization shown in Figure 8.19. Let's work our way from the bottom of the stack (the highest address) to the top (the lowest address). First

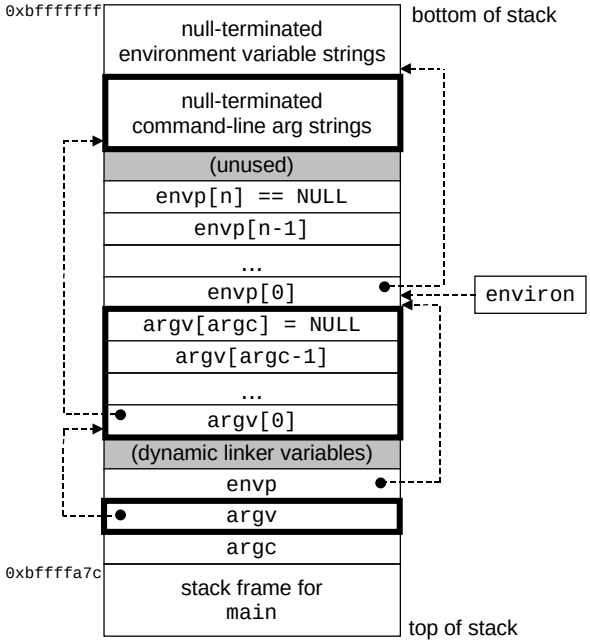


Figure 8.19: **Typical organization of the user stack when a new program starts.**

are the argument and environment strings, which are stored contiguously on the stack, one after the other without any gaps. These are followed further up the stack by a null-terminated array of pointers, each of which points to an environment variable string on the stack. The global variable `environ` points to the first of these pointers, `envp[0]`. The environment array is followed immediately by the null-terminated `argv[]` array, with each element pointing to an argument string on the stack. At the top of the stack are the three arguments to the `main` routine: (1) `envp`, which points the `envp[]` array, (2) `argv`, which points to the `argv[]` array, and (3) `argc`, which gives the number of non-null pointers in the `argv[]` array.

Unix provides several functions for manipulating the environment array.

```
#include <stdlib.h>

char *getenv(const char *name);
```
returns: ptr to `name` if exists, NULL if no match.

The `getenv` function searches the environment array for a string "`name=value`". If found, it returns a pointer to `value`, otherwise it returns `NULL`.

```
#include <stdlib.h>

int setenv(const char *name, const char *newvalue, int overwrite);
```
returns: 0 on success, -1 on error.

```
void unsetenv(const char *name);
```
returns: nothing.

If the environment array contains a string of the form "`name=oldvalue`" then `unsetenv` deletes it and `setenv` replaces `oldvalue` with `newvalue`, but only if `overwrite` is nonzero. If `name` does not exist, then `setenv` adds "`name=newvalue`" to the array.

Aside: Setting environment variables in Solaris systems
Solaris provides the `putenv` function in place of the `setenv` function. It provides no counterpart to the `unsetenv` function. **End Aside.**

Aside: Programs vs. processes.
This is a good place to stop and make sure you understand the distinction between a program and a process. A program is a collection of code and data; programs can exist as object modules on disk or as segments in an address space. A process is a specific instance of a program in execution; a program always runs in the context of some process. Understanding this distinction is important if you want to understand the `fork` and `execve` functions. The `fork` function runs the same program in a new child process that is a duplicate of the parent. The `execve` function loads and runs a new program in the context of the current process. While it overwrites the address space of the current process, it does *not* create a new process. The new program still has the same PID, and it inherits all of the file descriptors that were open at the time of the call to the `execve` function. **End Aside.**

Practice Problem 8.6:

Write a program, called `myecho`, that prints its command line arguments and environment variables. For example:

```
unix> ./myecho arg1 arg2
Command line arguments:
    argv[ 0]: myecho
    argv[ 1]: arg1
    argv[ 2]: arg2

Environment variables:
    envp[ 0]: PWD=/usr0/droh/ics/code/ecf
    envp[ 1]: TERM=emacs
```

```
  ...
envp[25]: USER=droh
envp[26]: SHELL=/usr/local/bin/tcsh
envp[27]: HOME=/usr0/droh
```

8.4.6 Using `fork` and `execve` to Run Programs

Programs such as Unix shells and Web servers (Chapter 12) make heavy use of the `fork` and `execve` functions. A shell is an interactive application-level program that runs other programs on behalf of the user. The original shell was the `sh` program, which was followed by variants such as `csh`, `tcsh`, `ksh`, and `bash`. A shell performs a sequence of *read/evaluate* steps, and then terminates. The read step reads a command line from the user. The evaluate step parses the command line and runs programs on behalf of the user.

Figure 8.20 shows the main routine of a simple shell. The shell print a command-line prompt, waits for the

code/ecf/shellex.c

```
1 #include "csapp.h"
2 #define MAXARGS   128
3
4 /* function prototypes */
5 void eval(char*cmdline);
6 int parseline(const char *cmdline, char **argv);
7 int builtin_command(char **argv);
8
9 int main()
10 {
11     char cmdline[MAXLINE]; /* command line */
12
13     while (1) {
14         /* read */
15         printf("> ");
16         Fgets(cmdline, MAXLINE, stdin);
17         if (feof(stdin))
18             exit(0);
19
20         /* evaluate */
21         eval(cmdline);
22     }
23 }
```

code/ecf/shellex.c

Figure 8.20: **The main routine for a simple shell program.**

user to type a command line on `stdin`, and then evaluates the command line.

Figure 8.21 shows the code that evaluates the command line. Its first task is to call the `parseline` function (Figure 8.22), which parses the space-separated command-line arguments and builds the `argv` vector that will eventually be passed to `execve`. The first argument is assumed to be either the name of a built-in shell command that is interpreted immediately, or an executable object file that will be loaded and run in the context of a new child process.

If the last argument is a "&" character, then `parseline` returns 1, indicating that the program should be executed in the *background* (the shell does not wait for it to complete). Otherwise it returns 0, indicating that the program should be run in the *foreground* (the shell waits for it to complete).

After parsing the command line, the `eval` function calls the `builtin_command` function, which checks whether the first command line argument is a built-in shell command. If so, it interprets the command immediately and returns 1. Otherwise, it returns 0. Our simple shell has just one built-in command, the `quit` command, which terminates the shell. Real shells have numerous commands, such as `pwd`, `jobs`, and `fg`.

If `builtin_command` returns 0, then the shell creates a child process and executes the requested program inside the child. If the user has asked for the program to run in the background, then the shell returns to the top of the loop and waits for the next command line. Otherwise the shell uses the `waitpid` function to wait for the job to terminate. When the job terminates, the shell goes on to the next iteration.

Notice that this simple shell is flawed because it does not reap any of its background children. Correcting this flaw requires the use of signals, which we describe in the next section.

8.5 Signals

To this point in our study of exceptional control flow, we have seen how hardware and software cooperate to provide the fundamental low-level exception mechanism. We have also seen how the operating system uses exceptions to support a higher-level form of exceptional control flow known as the context switch. In this section we will study a higher-level software form of exception, known as a Unix *signal*, that allows processes to interrupt other processes.

A *signal* is a message that notifies a process that an event of some type has occurred in the system. For example, Figure 8.23 shows the 30 different types of signals that are supported on Linux systems.

Each signal type corresponds to some kind of system event. Low-level hardware exceptions are processed by the kernel's exception handlers and would not normally be visible to user processes. Signals provide a mechanism for exposing the occurrence of such exceptions to user processes. For example, if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal (number 8). If a process executes an illegal instruction, the kernel sends it a SIGILL signal (number 4). If a process makes an illegal memory reference, the kernel sends it a SIGSEGV signal (number 11). Other signals correspond to higher-level software events in the kernel or in other user processes. For example, if you type a `ctrl-c` (i.e., press the `ctrl` key and the `c` key at the same time) while a process is running in the foreground, then the kernel sends a SIGINT (number 2) to the foreground process. A process can forcibly terminate another process by sending it a SIGKILL signal (number 9). When a child process terminates or stops, the kernel sends a SIGCHLD signal (number 17) to the parent.

code/ecf/shellex.c

```
1  /* eval - evaluate a command line */
2  void eval(char *cmdline)
3  {
4      char *argv[MAXARGS]; /* argv for execve() */
5      int bg;              /* should the job run in bg or fg? */
6      pid_t pid;           /* process id */
7
8      bg = parseline(cmdline, argv);
9      if (argv[0] == NULL)
10         return;   /* ignore empty lines */
11
12     if (!builtin_command(argv)) {
13         if ((pid = Fork()) == 0) {   /* child runs user job */
14             if (execve(argv[0], argv, environ) < 0) {
15                 printf("%s: Command not found.\n", argv[0]);
16                 exit(0);
17             }
18         }
19
20         /* parent waits for foreground job to terminate */
21         if (!bg) {
22             int status;
23             if (waitpid(pid, &status, 0) < 0)
24                 unix_error("waitfg: waitpid error");
25         }
26         else
27             printf("%d %s", pid, cmdline);
28     }
29     return;
30 }
31
32 /* if first arg is a builtin command, run it and return true */
33 int builtin_command(char **argv)
34 {
35     if (!strcmp(argv[0], "quit")) /* quit command */
36         exit(0);
37     if (!strcmp(argv[0], "&"))    /* ignore singleton & */
38         return 1;
39     return 0;                     /* not a builtin command */
40 }
```

code/ecf/shellex.c

Figure 8.21: `eval`**: evaluates the shell command line.**

code/ecf/shellex.c

```
1  /* parseline - parse the command line and build the argv array */
2  int parseline(const char *cmdline, char **argv)
3  {
4      char array[MAXLINE]; /* holds local copy of command line */
5      char *buf = array;   /* ptr that traverses command line */
6      char *delim;         /* points to first space delimiter */
7      int argc;            /* number of args */
8      int bg;              /* background job? */
9
10     strcpy(buf, cmdline);
11     buf[strlen(buf)-1] = ' ';  /* replace trailing '\n' with space */
12     while (*buf && (*buf == ' ')) /* ignore leading spaces */
13         buf++;
14
15     /* build the argv list */
16     argc = 0;
17     while ((delim = strchr(buf, ' '))) {
18         argv[argc++] = buf;
19         *delim = '\0';
20         buf = delim + 1;
21         while (*buf && (*buf == ' ')) /* ignore spaces */
22                buf++;
23     }
24     argv[argc] = NULL;
25
26     if (argc == 0)  /* ignore blank line */
27         return 1;
28
29     /* should the job run in the background? */
30     if ((bg = (*argv[argc-1] == '&')) != 0)
31         argv[--argc] = NULL;
32
33     return bg;
34 }
```

code/ecf/shellex.c

Figure 8.22: `parseline`**: parses a line of input for the shell.**

Number	Name	Default action	Corresponding event
1	SIGHUP	terminate	Terminal line hangup
2	SIGINT	terminate	Interrupt from keyboard
3	SIGQUIT	terminate	Quit from keyboard
4	SIGILL	terminate	Illegal instruction
5	SIGTRAP	terminate and dump core	Trace trap
6	SIGABRT	terminate and dump core	Abort signal from `abort` function
7	SIGBUS	terminate	Bus error
8	SIGFPE	terminate and dump core	Floating point exception
9	SIGKILL	terminate*	Kill program
10	SIGUSR1	terminate	User-defined signal 1
11	SIGSEGV	terminate and dump core	Invalid memory reference (seg fault)
12	SIGUSR2	terminate	User-defined signal 2
13	SIGPIPE	terminate	Wrote to a pipe with no reader
14	SIGALRM	terminate	Timer signal from `alarm` function
15	SIGTERM	terminate	Software termination signal
16	SIGSTKFLT	terminate	Stack fault on coprocessor
17	SIGCHLD	ignore	A child process has stopped or terminated
18	SIGCONT	ignore	Continue process if stopped
19	SIGSTOP	stop until next SIGCONT*	Stop signal not from terminal
20	SIGTSTP	stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	ignore	Urgent condition on socket
24	SIGXCPU	terminate	CPU time limit exceeded
25	SIGXFSZ	terminate	File size limit exceeded
26	SIGVTALRM	terminate	Virtual timer expired
27	SIGPROF	terminate	Profiling timer expired
28	SIGWINCH	ignore	Window size changed
29	SIGIO	terminate	I/O now possible on a descriptor.
30	SIGPWR	terminate	Power failure

Figure 8.23: **Linux signals.** Other Unix versions are similar. Notes: (1) *This signal can neither be caught nor ignored. (2) Years ago, main memory was implemented with a technology known as *core memory*. “Dumping core” is an historical term that means writing an image of the code and data memory segments to disk.

8.5.1 Signal Terminology

The transfer of a signal to a destination process occurs in two distinct steps:

- *Sending a signal.* The kernel *sends* (*delivers*) a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) the kernel has detected a system event such as a divide-by-zero error or the termination of a child process; (2) A process has invoked the `kill` function (discussed in the next section) to explicitly request the kernel to send a signal to the destination process. A process can send a signal to itself.
- *Receiving a signal.* A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal. The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*.

A signal that has been sent but not yet received is called a *pending signal*. At any point in time, there can be at most one pending signal of a particular type. If a process has a pending signal of type k, then any subsequent signals of type k sent to that process are *not* queued; they are simply discarded. A process can selectively *block* the receipt of certain signals. When a signal is blocked, it can be delivered, but the resulting pending signal will not be received until the process unblocks the signal.

A pending signal is received at most once. For each process, the kernel maintains the set of pending signals in the `pending` bit vector, and the set of blocked signals in the `blocked` bit vector. The kernel sets bit k in `pending` whenever a signal of type k is delivered and clears bit k in `pending` whenever a signal of type k is received.

8.5.2 Sending Signals

Unix systems provide a number of mechanisms for sending signals to processes. All of the mechanisms rely on the notion of a *process group*.

Process Groups

Every process belongs to exactly one *process group*, which is identified by a positive integer *process group ID*. The `getpgrp` function returns the process group ID of the current process.

```
#include <unistd.h>

pid_t getpgrp(void);
```

returns: process group ID of calling process

By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the `setpgid` function:

```
#include <unistd.h>

pid_t setpgid(pid_t pid, pid_t pgid);
```
returns: 0 on success, -1 on error.

The `setpgid` function changes the process group of process `pid` to `pgid`. If `pid` is zero, the PID of the current process is used. If `pgid` is zero, the PID of the process specified by `pid` is used for the process group ID. For example, if process 15213 is the calling process, then

```
setpgid(0, 0);
```

creates a new process group whose process group ID is 15213, and adds process 15213 to this new group.

Sending Signals With the `kill` Program

The `/bin/kill` program sends an arbitrary signal to another process. For example

```
unix> kill -9 15213
```

sends signal 9 (SIGKILL) to process 15213. A negative PID causes the signal to be sent to every process in process group PID. For example,

```
unix> kill -9 -15213
```

sends a SIGKILL signal to every process in process group 15213.

Sending Signals From the Keyboard

Unix shells use the abstraction of a *job* to represent the processes that are created as a result of evaluating a single command line. At any point in time, there is at most one foreground job and zero or more background jobs. For example, typing

```
unix> ls | sort
```

creates a foreground job consisting of two processes connected by a Unix pipe: one running the `ls` program, the other running the `sort` program.

The shell creates a separate process group for each job. Typically, the process group ID is taken from one of the parent processes in the job. For example, Figure 8.24 shows a shell with one foreground job and two background jobs. The parent process in the foreground job has a PID of 20 and a process group ID of 20. The parent process has created two children, each of which are also members of process group 20.

Typing `ctrl-c` at the keyboard causes a SIGINT signal to be sent to the shell. The shell catches the signal (see Section 8.5.3) and then sends a SIGINT to every process in the foreground process group. In the default case, the result is to terminate the foreground job. Similarly, typing `crtl-z` sends a SIGTSTP signal to the shell, which catches it and sends a SIGTSTP signal to every process in the foreground process group. In the default case, the result is to stop (suspend) the foreground job.

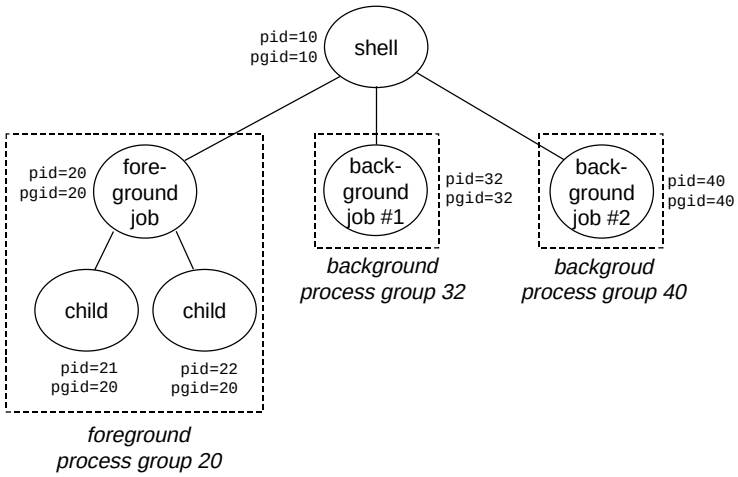


Figure 8.24: **Foreground and background process groups.**

Sending Signals With the `kill` Function

Processes send signals to other processes (including themselves) by calling the `kill` function.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

returns: 0 if OK, -1 on error

If `pid` is greater than zero, then the `kill` function sends signal number `sig` to process `pid`. If `pid` is less than zero, than `kill` sends signal `sig` to every process in process group `abs(pid)`. Figure 8.25 shows an example of a parent that uses the `kill` function to send a SIGKILL signal to its child.

Sending Signals With the `alarm` Function

A process can send SIGALRM signals to itself by calling the `alarm` function.

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

returns: remaining secs of previous alarm, or 0 if no previous alarm

The `alarm` function arranges for the kernel to send a SIGALRM signal to the calling process in `secs` seconds. If `secs` is zero, then no new alarm is scheduled. In any event, the call to `alarm` cancels any pending alarms, and returns the number of seconds remaining until any pending alarm was due to be delivered (had

code/ecf/kill.c

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6
7     /* child sleeps until SIGKILL signal received, then dies */
8     if ((pid = Fork()) == 0) {
9         Pause();  /* wait for a signal to arrive */
10        printf("control should never reach here!\n");
11        exit(0);
12    }
13
14    /* parent sends a SIGKILL signal to a child */
15    Kill(pid, SIGKILL);
16    exit(0);
17 }
```

code/ecf/kill.c

Figure 8.25: **Using the `kill` function to send a signal to a child.**

not this call to `alarm` cancelled it), or 0 if there were no pending alarms.

Figure 8.26 shows a program called `alarm` that arranges to be interrupted by a SIGALRM signal every second for five seconds. When the sixth SIGALRM is delivered it terminates. When we run the program in Figure 8.26, we get the following output: a "BEEP" every second for five seconds, followed by a "BOOM" when the program terminates.

```
unix> ./alarm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

Notice that the program in Figure 8.26 uses the `signal` function to install a *signal handler* function (`handler`) that is called asynchronously, interrupting the infinite `while` loop in `main`, whenever the process receives a SIGALRM signal. When the `handler` function returns, control passes back to `main`, which picks up where it was interrupted by the arrival of the signal. Installing and using signal handlers can be quite subtle, and is the topic of the next three sections.

8.5.3 Receiving Signals

When the kernel is returning from an exception handler and is ready to pass control to process p, it checks the set of unblocked pending signals (`pending & ~blocked`). If this set is empty (the usual case), then

code/ecf/alarm.c

```
1 #include "csapp.h"
2
3 void handler(int sig)
4 {
5     static int beeps = 0;
6
7     printf("BEEP\n");
8     if (++beeps < 5)
9         Alarm(1); /* next SIGALRM will be delivered in 1s */
10    else {
11        printf("BOOM!\n");
12        exit(0);
13    }
14 }
15
16 int main()
17 {
18    Signal(SIGALRM, handler); /* install SIGALRM handler */
19    Alarm(1); /* next SIGALRM will be delivered in 1s */
20
21    while (1) {
22        ;  /* signal handler returns control here each time */
23    }
24    exit(0);
25 }
```

code/ecf/alarm.c

Figure 8.26: **Using the `alarm` function to schedule periodic events.**

the kernel passes control to the next instruction (I_{next}) in the logical control flow of p.

However, if the set is nonempty, then the kernel chooses some signal k in the set (typically the smallest k) and forces p to *receive* signal k. The receipt of the signal triggers some *action* by the process. Once the process completes the action, then control passes back to the next instruction (I_{next}) in the logical control flow of p. Each signal type has a predefined *default action*, which is one of the following:

- The process terminates.
- The process terminates and dumps core.
- The process stops until restarted by a SIGCONT signal.
- The process ignores the signal.

Figure 8.23 shows the default actions associated with each type of signal. For example, the default action for the receipt of a SIGKILL is to terminate the receiving process. On the other hand, the default action for the receipt of a SIGCHLD is to ignore the signal. A process can modify the default action associated with a signal by using the `signal` function. The only exceptions are SIGSTOP and SIGKILL, whose default actions cannot be changed.

```
#include <signal.h>

typedef void handler_t(int)

handler_t *signal(int signum, handler_t *handler)
```
returns: ptr to previous handler if OK, SIG_ERR on error (does not set errno)

The `signal` function can change the action associated with a signal `signum` in one of three ways:

- If `handler` is SIG_IGN, then signals of type `signum` are ignored.
- If `handler` is SIG_DFL, then the action for signals of type `signum` reverts to the default action.
- Otherwise, `handler` is the address of a user-defined function, called a *signal handler*, that will be called whenever the process receives a signal of type `signum`. Changing the default action by passing the address of a handler to the `signal` function is known as *installing the handler*. The invocation of the handler is called *catching the signal*. The execution of the handler is referred to as *handling the signal*.

When a process catches a signal of type k, the handler installed for signal k is invoked with a single integer argument set to k. This argument allows the same handler function to catch different types of signals.

When the handler executes its `return` statement, control (usually) passes back to the instruction in the control flow where the process was interrupted by the receipt of the signal. We say “usually” because in some systems, interrupted system calls return immediately with an error. More on this in the next section.

Figure 8.27 shows a program that catches the SIGINT signal sent by the shell whenever the user types `ctrl-c` at the keyboard. The default action for SIGINT is to immediately terminate the process. In this example, we modify the default behavior to catch the signal, print a message, and then terminate the process.

code/ecf/sigint1.c

```
1 #include "csapp.h"
2
3 void handler(int sig) /* SIGINT handler */
4 {
5     printf("Caught SIGINT\n");
6     exit(0);
7 }
8
9 int main()
10 {
11     /* Install the SIGINT handler */
12     if (signal(SIGINT, handler) == SIG_ERR)
13         unix_error("signal error");
14
15     pause(); /* wait for the receipt of a signal */
16
17     exit(0);
18 }
```

code/ecf/sigint1.c

Figure 8.27: **A program that catches a SIGINT signal.**

The handler function is defined in lines 3–7. The main routine installs the handler in lines 12–13, and then goes to sleep until a signal is received (line 15). When the SIGINT signal is received, the handler runs, prints a message (line 5) and then terminates the process (line 6).

Practice Problem 8.7:

Write a program, called `snooze`, that takes a single command line argument, calls the `snooze` function from Problem 8.5 with this argument, and then terminates. Write your program so that the user can interrupt the snooze function by typing `ctrl-c` at the keyboard. For example:

```
unix> ./snooze 5
Slept for 3 of 5 secs.      User hits crtl-c after 3 seconds
unix>
```

8.5.4 Signal Handling Issues

Signal handling is straightforward for programs that catch a single signal and then terminate. However, subtle issues arise when a program catches multiple signals.

- *Pending signals can be blocked.* Unix signal handlers typically block pending signals of the type currently being processed by the handler. For example, suppose a process has caught a SIGINT signal and is currently running its SIGINT handler. If another SIGINT signal is sent to the process, then the SIGINT will become pending, but will not be received until after the handler returns.

- *Pending signals are not queued.* There can be at most one pending signal of any particular type. Thus, if two signals of type k are sent to a destination process while signal k is blocked because the destination process is currently executing a handler for signal k, then the second signal is simply discarded; it is not queued. The key idea is that the existence of a pending signal merely indicates that *at least* one signal has arrived.

- *System calls can be interrupted.* System calls such as `read`, `wait`, and `accept` that can potentially block the process for a long period of time are called *slow system calls*. On some systems, slow system calls that are interrupted when a handler catches a signal do not resume when the signal handler returns, but instead return immediately to the user with an error condition and `errno` set to EINTR.

Let's look more closely at the subtleties of signal handling, using a simple application that is similar in nature to real programs such as shells and Web servers. The basic structure is that a parent process creates some children that run independently for a while and then terminate. The parent must reap the children to avoid leaving zombies in the system. But we also want the parent to be free to do other work while the children are running. So we decide to reap the children with a SIGCHLD handler, instead of explicitly waiting for the children the terminate. (Recall that the kernel sends a SIGCHLD signal to the parent whenever one of its children terminates or stops.)

Figure 8.28 shows our first attempt. The parent installs a SIGCHLD handler, and then creates three children, each of which runs for 1 second and then terminates. In the meantime, the parent waits for a line of input from the terminal and then processes it. This processing is modeled by an infinite loop. When each child terminates, the kernel notifies the parent by sending it a SIGCHLD signal. The parent catches the SIGCHLD, reaps one child, does some additional cleanup work (modeled by the `sleep(2)` statement), and then returns.

The `signal1` program in Figure 8.28 seems fairly straightforward. But when we run it on our Linux system, we get the following output:

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
 <cr>
Parent processing input
```

From the output, we see that even though three SIGCHLD signals were sent to the parent, only two of these signals were received, and thus the parent only reaped two children. If we suspend the parent process, we see that indeed child process 10321 was never reaped and remains a zombie:

code/ecf/signal1.c

```
1  #include "csapp.h"
2
3  void handler1(int sig)
4  {
5      pid_t pid;
6
7      if ((pid = waitpid(-1, NULL, 0)) < 0)
8          unix_error("waitpid error");
9      printf("Handler reaped child %d\n", (int)pid);
10     Sleep(2);
11     return;
12 }
13
14 int main()
15 {
16     int i, n;
17     char buf[MAXBUF];
18
19     if (signal(SIGCHLD, handler1) == SIG_ERR)
20         unix_error("signal error");
21
22     /* parent creates children */
23     for (i = 0; i < 3; i++) {
24         if (Fork() == 0) {
25             printf("Hello from child %d\n", (int)getpid());
26             Sleep(1);
27             exit(0);
28         }
29     }
30
31     /* parent waits for terminal input and then processes it */
32     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
33         unix_error("read");
34
35     printf("Parent processing input\n");
36     while (1)
37         ;
38
39     exit(0);
40 }
```

code/ecf/signal1.c

Figure 8.28: `signal1`: This program is flawed because it fails to deal with the facts that signals can block, signals are not queued, and system calls can be interrupted.

```
 <ctrl-z>
Suspended
linux> ps
PID TTY STAT TIME COMMAND
...
10319  p5 T    0:03 signal1
10321  p5 Z    0:00 (signal1 <zombie>)
10323  p5 R    0:00 ps
```

What went wrong? The problem is that our code failed to account for the facts that signals can block and that signals are not queued. Here's what happened:

The first signal is received and caught by the parent. While the handler is still processing the first signal, the second signal is delivered and added to the set of pending signals. However, since SIGCHLD signals are blocked by the SIGCHLD handler, the second signal is not received. Shortly thereafter, while the handler is still processing the first signal, the third signal arrives. Since there is already a pending SIGCHLD, this third SIGCHLD signal is discarded. Sometime later, after the handler has returned, the kernel notices that there is a pending SIGCHLD signal and forces the parent to receive the signal. The parent catches the signal and executes the handler a second time. After the handler finishes processing the second signal, there are no more pending SIGCHLD signals, and there never will be, because all knowledge of the third SIGCHLD has been lost. The crucial lesson is that signals cannot be used to count the occurrence of events in other processes.

To fix the problem, we must recall that the existence of a pending signal only implies that at least one signal has been delivered since the last time the process received a signal of that type. So we must modify the SIGCHLD handler to reap as many zombie children as possible each time it is invoked. Figure 8.29 shows the modified SIGCHLD handler. When we run `signal2` on our Linux system, it now correctly reaps all of the zombie children:

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
 <cr>
Parent processing input
```

However, we are not done yet. If we run the `signal2` program on a Solaris system, it correctly reaps all of the zombie children. However, now the blocked `read` system call returns prematurely with an error, before we are able to type in our input on the keyboard:

```
solaris> ./signal2
Hello from child 18906
Hello from child 18907
Hello from child 18908
Handler reaped child 18906
Handler reaped child 18908
```

code/ecf/signal2.c

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5      pid_t pid;
6
7      while ((pid = waitpid(-1, NULL, 0)) > 0)
8          printf("Handler reaped child %d\n", (int)pid);
9      if (errno != ECHILD)
10         unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main()
16 {
17     int i, n;
18     char buf[MAXBUF];
19
20     if (signal(SIGCHLD, handler2) == SIG_ERR)
21         unix_error("signal error");
22
23     /* parent creates children */
24     for (i = 0; i < 3; i++) {
25         if (Fork() == 0) {
26             printf("Hello from child %d\n", (int)getpid());
27             Sleep(1);
28             exit(0);
29         }
30     }
31
32     /* parent waits for terminal input and then processes it */
33     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
34         unix_error("read error");
35
36     printf("Parent processing input\n");
37     while (1)
38         ;
39
40     exit(0);
41 }
```

code/ecf/signal2.c

Figure 8.29: `signal2`: An improved version of Figure 8.28 that correctly accounts for the facts that signals can block and are not queued. However it does not allow for the possibility that system calls can be interrupted.

```
Handler reaped child 18907
read: Interrupted system call
```

What went wrong? The problem arises because on this particular Solaris system, slow system calls such as `read` are not restarted automatically after they are interrupted by the delivery of a signal. Instead they return prematurely to the calling application with an error condition, unlike Linux systems, which restart interrupted system calls automatically.

In order to write portable signal handling code, we must allow for the possibility that system calls will return prematurely and then restart them manually when this occurs. Figure 8.30 shows the modification to `signal1` that manually restarts aborted `read` calls. The EINTR return code in `errno` indicates that the `read` system call returned prematurely after it was interrupted.

When we run our new `signal3` program on a Solaris system, the program runs correctly:

```
solaris> ./signal3
Hello from child 19571
Hello from child 19572
Hello from child 19573
Handler reaped child 19571
Handler reaped child 19572
Handler reaped child 19573
<cr>
Parent processing input
```

8.5.5 Portable Signal Handling

The differences in signal handling semantics from system to system — such as whether or not an interrupted slow system call is restarted or aborted prematurely — is an ugly aspect of Unix signal handling. To deal with this problem, the Posix standard defines the `sigaction` function, which allows users on Posix-compliant systems such as Linux and Solaris to clearly specify the signal-handling semantics they want.

```
#include <signal.h>

int sigaction(int signum, struct sigaction *act, struct sigaction *oldact);
                                                    returns: 0 if OK, -1 on error
```

The `sigaction` function is unwieldy because it requires the user to set the entries of a structure. A cleaner approach, originally proposed by Stevens [77], is to define a wrapper function, called `Signal`, that calls `sigaction` for us. Figure 8.31 shows the definition of `Signal`, which is invoked in the same way as the `signal` function. The `Signal` wrapper installs a signal handler with the following signal-handling semantics:

- Only signals of the type currently being processed by the handler are blocked.
- As with all signal implementations, signals are not queued.

code/ecf/signal3.c

```
1 #include "csapp.h"
2
3 void handler2(int sig)
4 {
5     pid_t pid;
6
7     while ((pid = waitpid(-1, NULL, 0)) > 0)
8         printf("Handler reaped child %d\n", (int)pid);
9     if (errno != ECHILD)
10        unix_error("waitpid error");
11    Sleep(2);
12    return;
13 }
14
15 int main() {
16    int i, n;
17    char buf[MAXBUF];
18    pid_t pid;
19
20    if (signal(SIGCHLD, handler2) == SIG_ERR)
21        unix_error("signal error");
22
23    /* parent creates children */
24    for (i = 0; i < 3; i++) {
25        pid = Fork();
26        if (pid == 0) {
27            printf("Hello from child %d\n", (int)getpid());
28            Sleep(1);
29            exit(0);
30        }
31    }
32
33    /* Manually restart the read call if it is interrupted */
34    while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
35        if (errno != EINTR)
36            unix_error("read error");
37
38    printf("Parent processing input\n");
39    while (1)
40        ;
41
42    exit(0);
43 }
```

code/ecf/signal3.c

Figure 8.30: `signal3`: An improved version of Figure 8.29 that correctly accounts for the fact that system calls can be interrupted.

code/src/csapp.c

```
1 handler_t *Signal(int signum, handler_t *handler)
2 {
3     struct sigaction action, old_action;
4
5     action.sa_handler = handler;
6     sigemptyset(&action.sa_mask); /* block sigs of type being handled */
7     action.sa_flags = SA_RESTART; /* restart syscalls if possible */
8
9     if (sigaction(signum, &action, &old_action) < 0)
10        unix_error("Signal error");
11    return (old_action.sa_handler);
12 }
```

code/src/csapp.c

Figure 8.31: `Signal`: A wrapper for `sigaction` that provides portable signal handling on Posix-compliant systems.

- Interrupted system calls are automatically restarted whenever possible.
- Once the signal handler is installed, it remains installed until `Signal` is called with a `handler` argument of either SIG_IGN or SIG_DFL. (Some older Unix systems restore the signal action to its default action after a signal has been processed by a handler.)

Figure 8.32 shows a version of the `signal2` program from Figure 8.29 that uses our `Signal` wrapper to get predictable signal handling semantics on different computer systems. The only difference is that we have installed the handler with a call to `Signal` rather than a call to `signal`. The program now runs correctly on both our Solaris and Linux systems, and we no longer need to manually restart interrupted `read` system calls.

8.6 Nonlocal Jumps

C provides a form of user-level exceptional control flow, called a *nonlocal jump*, that transfers control directly from one function to another currently executing function, without having to go through the normal call-and-return sequence. Nonlocal jumps are provided by the `setjmp` and `longjmp` functions.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
                                    returns: 0 from setjmp, nonzero from longjmps)
```

The `setjmp` function saves the current stack context in the `env` buffer, for later use by `longjmp`, and

code/ecf/signal4.c

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5      pid_t pid;
6
7      while ((pid = waitpid(-1, NULL, 0)) > 0)
8          printf("Handler reaped child %d\n", (int)pid);
9      if (errno != ECHILD)
10         unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main()
16 {
17     int i, n;
18     char buf[MAXBUF];
19     pid_t pid;
20
21     Signal(SIGCHLD, handler2); /* sigaction error-handling wrapper */
22
23     /* parent creates children */
24     for (i = 0; i < 3; i++) {
25         pid = Fork();
26         if (pid == 0) {
27             printf("Hello from child %d\n", (int)getpid());
28             Sleep(1);
29             exit(0);
30         }
31     }
32
33     /* parent waits for terminal input and then processes it */
34     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
35         unix_error("read error");
36
37     printf("Parent processing input\n");
38     while (1)
39         ;
40     exit(0);
41 }
```

code/ecf/signal4.c

Figure 8.32: `signal4`: A version of Figure 8.29 that uses our `Signal` wrapper to get portable signal-handling semantics.

returns a 0.

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
                                                                    never returns)
```

The `longjmp` function restores the stack context from the `env` buffer and then triggers a return from the most recent `setjmp` call that initialized `env`. The `setjmp` then returns with the nonzero return value `retval`.

The interactions between `setjmp` and `longjmp` can be confusing at first glance. The `setjmp` function is called once but returns *multiple times*: once when the `setjmp` is first called and the stack context is stored in the `env` buffer, and once for each corresponding `longjmp` call. On the other hand, the `longjmp` function is called once but never returns.

An important application of nonlocal jumps is to permit an immediate return from a deeply nested function call, usually as a result of detecting some error condition. If an error condition is detected deep in a nested function call, we can use a nonlocal jump to return directly to a common localized error handler instead of laboriously unwinding the call stack.

Figure 8.33 shows an example of how this might work. The `main` routine first calls `setjmp` to save the current stack context, and then calls function `foo`, which in turn calls function `bar`. If `foo` or `bar` encounter an error, they return immediately from the `setjmp` via a `longjmp` call. The nonzero return value of the `setjmp` indicates the error type, which can then be decoded and handled in one place in the code.

Another important application of nonlocal jumps is to branch out of a signal handler to a specific code location, rather than returning to the instruction that was interrupted by the arrival of the signal. For example, if a Web server attempts to send data to a browser that has unilaterally aborted the network connection between the client and the server, (e.g., as a result of the browser's user clicking the STOP button), the kernel will send a SIGPIPE signal to the server. The default action for the SIGPIPE signal is to terminate the process, which is clearly not a good thing for a server that is supposed to run forever. Thus, a robust Web server will install a SIGPIPE handler to catch these signals. After cleaning up, the SIGPIPE handler should jump to the code that waits for the next request from a browser, rather than returning to the instruction that was interrupted by the receipt of the SIGPIPE signal. Nonlocal jumps are the only way to handle this kind of error recovery.

Figure 8.34 shows a simple program that illustrates this basic technique. The program uses signals and nonlocal jumps to do a soft restart whenever the user types `ctrl-c` at the keyboard. The `sigsetjmp` and `siglongjmp` functions are versions of `setjmp` and `longjmp` that can be used by signal handlers.

The initial call to the `sigsetjmp` function saves the stack and signal context when the program first starts. The main routine then enters an infinite processing loop. When the user types ctrl-c, the shell sends a SIGINT signal to the process, which catches it. Instead of returning from the signal handler, which would pass back control back to the interrupted processing loop, the handler performs a nonlocal jump back to the

code/ecf/setjmp.c

```
#include "csapp.h"

jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    int rc;

    rc = setjmp(buf);
    if (rc == 0)
        foo();
    else if (rc == 1)
        printf("Detected an error1 condition in foo\n");
    else if (rc == 2)
        printf("Detected an error2 condition in foo\n");
    else
        printf("Unknown error condition in foo\n");
    exit(0);
}

/* deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

code/ecf/setjmp.c

Figure 8.33: **Nonlocal jump example.** This example shows the framework for using nonlocal jumps to recover from error conditions in deeply nested functions without having to unwind the entire stack.

code/ecf/restart.c

```
1 #include "csapp.h"
2
3 sigjmp_buf buf;
4
5 void handler(int sig)
6 {
7     siglongjmp(buf, 1);
8 }
9
10 int main()
11 {
12     Signal(SIGINT, handler);
13
14     if (!sigsetjmp(buf, 1))
15         printf("starting\n");
16     else
17         printf("restarting\n");
18
19     while(1) {
20         Sleep(1);
21         printf("processing...\n");
22     }
23     exit(0);
24 }
```

code/ecf/restart.c

Figure 8.34: **A program that uses nonlocal jumps to restart itself when the user types `ctrl-c`.**

beginning of the `main` program.

When we ran the program on our system, we got the following output:

```
unix> ./restart
starting
processing...
processing...
restarting          user hits ctrl-c
processing...
restarting          User hits ctrl-c
processing...
```

8.7 Tools for Manipulating Processes

Unix systems provide a number of useful tools for monitoring and manipulating processes.

STRACE: Prints a trace of each system call invoked by a program and its children. A fascinating tool for the curious student. Compile your program with `-static` to get a cleaner trace without a lot of output related to shared libraries.

PS: Lists processes (including zombies) currently in the system.

TOP: Prints information about the resource usage of current processes.

KILL: Sends a signal to a process. Useful for debugging programs with signal handlers and cleaning up wayward processes.

`/proc` **(Linux and Solaris)** : A virtual filesystem that exports the contents of numerous kernel data structures in an ASCII text form that can be read by user programs. For example, type "`cat /proc/loadavg`" to see the current load average on your Linux system.

8.8 Summary

Exceptional control flow occurs at all levels of a computer system. At the hardware level, exceptions are abrupt changes in the control flow that are triggered by events in the processor. At the operating system level, the kernel triggers abrupt changes in the control flows between different processes when it performs context switches. At the interface between the operating system and applications, applications can create child processes, wait for their child processes to stop or terminate, run new programs, and catch signals from other processes. The semantics of signal handling is subtle and can vary from system to system. However, mechanisms exist on Posix-compliant systems that allow programs to clearly specify the expected signal-handling semantics. Finally, at the application level, C programs can use nonlocal jumps to bypass the normal call/return stack discipline and branch directly from one function to another.

Bibliographic Notes

The Intel macro-architecture specification contains a detailed discussion of exceptions and interrupts on Intel processors [17]. Operating systems texts [66, 71, 79] contain additional information on exceptions, processes, and signals. The classic work by Stevens [72], while somewhat outdated, remains a valuable and highly readable description of how to work with processes and signals from application programs. Bovet and Cesati give a wonderfully clear description of the Linux kernel, including details of the process and signal implementations.

Homework Problems

Homework Problem 8.8 [Category 1]:

In this chapter, we have introduced some functions with unusual call and return behaviors: `setjmp`, `longjmp`, `execve`, and `fork`. Match each function with one of the following behaviors:

A. Called once, returns twice.

B. Called once, never returns.

C. Called once, returns one or more times.

Homework Problem 8.9 [Category 1]:

What is one possible output of the following program?

code/ecf/forkprob3.c

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int x = 3;
6
7     if (Fork() != 0)
8         printf("x=%d\n", ++x);
9
10    printf("x=%d\n", --x);
11    exit(0);
12 }
```

code/ecf/forkprob3.c

Homework Problem 8.10 [Category 1]:

How many “hello” output lines does this program print?

code/ecf/forkprob5.c

```
1 #include "csapp.h"
2
3 void doit()
4 {
5     if (Fork() == 0) {
6         Fork();
7         printf("hello\n");
8         exit(0);
9     }
10    return;
11 }
12
13 int main()
14 {
15    doit();
16    printf("hello\n");
17    exit(0);
18 }
```

code/ecf/forkprob5.c

Homework Problem 8.11 [Category 1]:

How many “hello” output lines does this program print?

code/ecf/forkprob6.c

```
1 #include "csapp.h"
2
3 void doit()
4 {
5     if (Fork() == 0) {
6         Fork();
7         printf("hello\n");
8         return;
9     }
10    return;
11 }
12
13 int main()
14 {
15    doit();
16    printf("hello\n");
17    exit(0);
18 }
```

code/ecf/forkprob6.c

Homework Problem 8.12 [Category 1]:

What is the output of the following program?

code/ecf/forkprob7.c

```
1 #include "csapp.h"
2 int counter = 1;
3
4 int main()
5 {
6     if (fork() == 0) {
7         counter--;
8         exit(0);
9     }
10    else {
11        Wait(NULL);
12        printf("counter = %d\n", ++counter);
13    }
14    exit(0);
15 }
```

code/ecf/forkprob7.c

Homework Problem 8.13 [Category 1]:

Enumerate all of the possible outputs of the program in Problem 8.4.

Homework Problem 8.14 [Category 2]:

Consider the following program:

code/ecf/forkprob2.c

```
1 #include "csapp.h"
2
3 void end(void)
4 {
5     printf("2");
6 }
7
8 int main()
9 {
10    if (Fork() == 0)
11        atexit(end);
12    if (Fork() == 0)
13        printf("0");
14    else
15        printf("1");
16    exit(0);
17 }
```

code/ecf/forkprob2.c

Determine which of the following outputs are possible. Note: The `atexit` function takes a pointer to a function and adds it to a list of functions (initially empty) that will be called when the `exit` function is called.

A. 112002

B. 211020

C. 102120

D. 122001

E. 100212

Homework Problem 8.15 [Category 2]:

Use `execve` to write a program, called `myls`, whose behavior is identical to the `/bin/ls` program. Your program should accept the same command line arguments, interpret the identical environment variables, and produce the identical output.

The `ls` program gets the width of the screen from the COLUMNS environment variable. If COLUMNS is unset, then `ls` assumes that the screen is 80 columns wide. Thus, you can check your handling of the environment variables by setting the COLUMNS environment to something smaller than 80:

```
unix> setenv COLUMNS 40
unix> ./myls
  ...output is 40 columns wide
unix> unsetenv COLUMNS
unix> ./myls
  ...output is now 80 columns wide
```

Homework Problem 8.16 [Category 3]:

Modify the program in Figure 8.15 so that

1. Each child terminates abnormally after attempting to write to a location in the read-only text segment.
2. The parent prints output that is identical (except for the PIDs) to the following:

```
child 12255 terminated by signal 11: Segmentation fault
child 12254 terminated by signal 11: Segmentation fault
```

Hint: Read the `man` pages for `wait(2)` and `psignal(3)`.

Homework Problem 8.17 [Category 3]:

Write your own version of the Unix `system` function:

```
int mysystem(char *command);
```

The `mysystem` function executes `command` by calling "`/bin/sh -c command`", and then returns after `command` has completed. If `command` exits normally (by calling the `exit` function or executing a `return` statement), then `mysystem` returns the `command` exit status. For example, if `command` terminates

by calling `exit(8)`, then `system` returns the value 8. Otherwise, if `command` terminates abnormally, then `mysystem` returns the status returned by the shell.

Homework Problem 8.18 [Category 1]:

One of your colleagues is thinking of using signals to allow a parent process to count events that occur in a child process. The idea is to notify the parent each time an event occurs by sending it a signal, and letting the parent's signal handler increment a global `counter` variable, which the parent can then inspect after the child has terminated. However, when he runs the test program in Figure 8.35 on his system, he discovers that when the parent calls `printf`, `counter` always has a value of 2, even though the child has sent five signals to the parent. Perplexed, he comes to you for help. Can you explain the bug?

code/ecf/counterprob.c

```
1 #include "csapp.h"
2
3 int counter = 0;
4
5 void handler(int sig)
6 {
7     counter++;
8     sleep(1); /* do some work in the handler */
9     return;
10 }
11
12 int main()
13 {
14     int i;
15
16     Signal(SIGUSR2, handler);
17
18     if (Fork() == 0) {  /* child */
19         for (i = 0; i < 5; i++) {
20             Kill(getppid(), SIGUSR2);
21             printf("sent SIGUSR2 to parent\n");
22         }
23         exit(0);
24     }
25
26     Wait(NULL);
27     printf("counter=%d\n", counter);
28     exit(0);
29 }
```

code/ecf/counterprob.c

Figure 8.35: **Counter program referenced in Problem 8.18.**

Homework Problem 8.19 [Category 3]:

Write a version of the `fgets` function, called `tfgets`, that times out after 5 seconds. The `tfgets`

function accepts the same inputs as `fgets`. If the user doesn't type an input line within 5 seconds, `tfgets` returns NULL. Otherwise it returns a pointer to the input line.

Homework Problem 8.20 [Category 4]:

Using the example in Figure 8.20 as a starting point, write a shell program that supports job control. Your shell should have the following features:

- The command line typed by the user consists of a `name` and zero or more arguments, all separated by one or more spaces. If `name` is a built-in command, the shell handles it immediately and waits for the next command line. Otherwise, the shell assumes that `name` is an executable file, which it loads and runs in the context of an initial child process (job). The process group ID for the job is identical to the PID of the child.
- Each job is identified by either a process ID (PID) or a job ID (JID), which is a small arbitrary positive integer assigned by the shell. JIDs are denoted on the command line by the prefix '%'. For example, "`%5`" denotes JID 5, and "`5`" denotes PID 5.
- If the command line ends with an ampersand, then the shell runs the job in the background. Otherwise, the shell runs the job in the foreground.
- Typing `ctrl-c` (`ctrl-z`) causes the shell to send a SIGINT (SIGTSTP) signal to every process in the foreground process group.
- The `jobs` built-in command lists all background jobs.
- The `bg <job>` built-in command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
- The `fg <job>` built-in command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground.
- The shell reaps all of its zombie children. If any job terminates because it receives a signal that was not caught, then the shell prints a message to the terminal with the job's PID and a description of the offending signal.

Figure 8.36 shows an example shell session.

```
unix> ./shell                          Run your shell program
> bogus
bogus: Command not found.                    Execve can't find executable
> foo 10
Job 5035 terminated by signal: Interrupt     User types ctrl-c
> foo 100 &
[1] 5036 foo 100 &
> foo 200 &
[2] 5037 foo 200 &
> jobs
[1] 5036 Running    foo 100 &
[2] 5037 Running    foo 200 &
> fg %1
Job [1] 5036 stopped by signal: Stopped      User types ctrl-z
> jobs
[1] 5036 Stopped    foo 100 &
[2] 5037 Running    foo 200 &
> bg 5035
5035: No such process
> bg 5036
[1] 5036 foo 100 &
> /bin/kill 5036
Job 5036 terminated by signal: Terminated
> fg %2                                      Wait for fg job to finish.
> quit
unix>                                       Back to the Unix shell
```

Figure 8.36: **Sample shell session for Problem 8.20.**

Chapter 9

Measuring Program Execution Time

One common question people ask is "How fast does Program X run on Machine Y?" Such a question might be raised by a programmer trying to optimize program performance, or by a customer trying to decide which machine to buy. In our earlier discussion of performance optimization (Chapter 5), we assumed this question could be answered with perfect accuracy. We were trying to establish the cycles per element (CPE) measure for programs down to two decimal places. This requires an accuracy of 0.1% for a procedure having a CPE of 10. In this chapter, we address this problem and discover that it is surprisingly complex.

You might expect that making near-perfect timing measurements on a computer system would be straightforward. After all, for a particular combination of program and data, the machine will execute a fixed sequence of instructions. Instruction execution is controlled by a processor clock that is regulated by a precision oscillator. There are many factors, however, that can vary from one execution of a program to another. Computers do not simply execute one program at a time. They continually switch from one process to another, executing some code on behalf of one process before moving on to the next. The exact scheduling of processor resources for one program depends on such factors as the number of users sharing the system, the network traffic, and the timing of disk operations. The access patterns to the caches depend not just on the references made by the program we are trying to measure, but on those of other processes executing concurrently. Finally, the branch prediction logic tries to guess whether branches will be taken or not based on past history. This history can vary from one execution of a program to another.

In this chapter, we describe two basic mechanisms computers use to record the passage of time—one based on a low frequency timer that periodically interrupts the processor and one based on a counter that is incremented every clock cycle. Application programmers can gain access to the first timing mechanism by calling library functions. Cycle timers can be accessed by library functions on some systems, but they require writing assembly code on others. We have deferred the discussion of program timing until now, because it requires understanding aspects of both the CPU hardware and the way the operating system manages process execution.

Using the two timing mechanisms, we investigate methods to get reliable measurements of program performance. We will see that timing variations due to context switching tend to be very large and hence must be eliminated. Variations caused by other factors such as cache and branch prediction are generally managed by evaluating program operation under carefully controlled conditions. Generally, we can get accurate measurements for durations that are either very short (less than around 10 millisecond) or very long (greater than

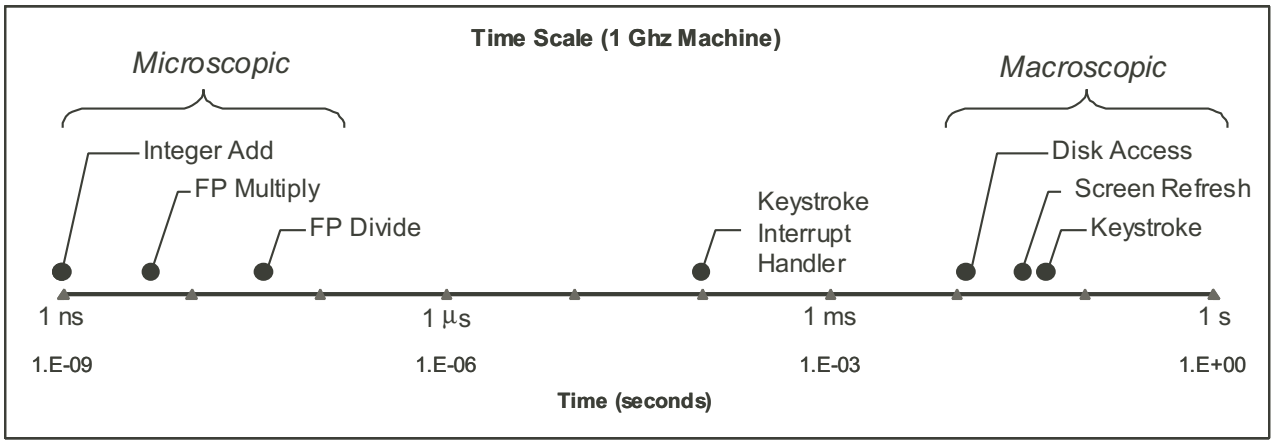


Figure 9.1: **Time Scale of Computer System Events.** The processor hardware works at a microscopic a time scale in which events having durations on the order of a few nanoseconds (ns). The OS must deal on a macroscopic time scale with events having durations on the order of a few milliseconds (ms).

around 1 second), even on heavily loaded machines. Times between around 10 milliseconds and 1 second require special care to measure accurately.

Much of the understanding of performance measurement is part of the folklore of computer systems. Different groups and individuals have developed their own techniques for measuring program performance, but there is no widely available body of literature on the subject. Companies and research groups concerned with getting highly accurate performance measurements often set up specially configured machines that minimize any sources of timing irregularity, such as by limiting access and by disabling many OS and networking services. We want methods that application programmers can use on ordinary machines, but there are no widely available tools for this. Instead, we will develop our own.

In this presentation we work through the issues systematically. We describe the design and evaluation of a number of experiments that helped us arrive at methods to achieve accurate measurements on a small set of systems. It is unusual to find a detailed experimental study in a book at this level. Generally, people expect the final answers, not a description of how those answers were determined. In this case, however, we cannot provide definitive answers on how to measure program execution time for an arbitrary program on an arbitrary system. There are too many variations of timing mechanisms, operating system behaviors, and runtime environment to have one single, simple solution. Instead, we anticipate that you will need to run your own experiments and develop your own performance measurement code. We hope that our case study will help you in this task. We summarize our findings in the form of a protocol that can guide your experiments.

9.1 The Flow of Time on a Computer System

Computers operate on two fundamentally different time scales. At a microscopic level, they execute instructions at a rate of one or more per clock cycle, where each clock cycle requires only around one nanosecond (abbreviated “ns”), or 10^{-9} seconds. On a macroscopic scale, the processor must respond to external events

that occur on time scales measured in milliseconds (abbreviated "ms"), or 10^{-3} seconds. For example, during video playback, the graphics display for most computers must be refreshed every 33 ms. A world-record typist can only type keystrokes at a rate of around one every 50 milliseconds. Disks typically require around 10 ms to initiate a disk transfer. The processor continually switches between these many tasks on a macroscopic time scale, devoting around 5 to 20 milliseconds to each task at a time. At this rate, the user perceives the tasks as being performed simultaneously, since a human cannot discern time durations shorter than around 100 ms. Within that time the processor can execute millions of instructions.

Figure 9.1 plots the durations of different event types on a logarithmic scale, with microscopic events having durations measured in nanoseconds and macroscopic events having durations measured in milliseconds. The macroscopic events are managed by OS routines that require around 5,000 to 200,000 clock cycles. These time ranges are measured in microseconds (abbreviated μs, where μ is the Greek letter "mu"). Although that may sound like a lot of computation, it is so much faster than the macroscopic events being processed that these routines place only a small load on the processor.

Practice Problem 9.1:

When a user is editing files with a real-time editor such as EMACS, every keystroke generates an interrupt signal. The operating system must then schedule the editor process to take the appropriate action for this keystroke. Suppose we had a system with a 1 GHz clock, and we had 100 users running EMACS typing at a rate of 100 words per minute. Assume an average of 6 characters per word. Assume also that the OS routine handling keystrokes requires, on average, 100,000 clock cycles per keystroke. What fraction of the processor load is consumed by all of the keystroke processing?

Note that this is a very pessimistic analysis of the load induced by keyboard usage. It's hard to imagine a real-life scenario with so many users typing this fast.

9.1.1 Process Scheduling and Timer Interrupts

External events such as keystrokes, disk operations, and network activity generate interrupt signals that make the operating system scheduler take over and possibly switch to a different process. Even in the absence of such events, we want the processor to switch from one process to another so that it will appear to the users as if the processor is executing many programs simultaneously. For this reason, computers have an external timer that periodically generates an interrupt signal to the processor. The spacing between these interrupt signals is called the *interval time*. When a timer interrupt occurs, the operating system scheduler can choose to either resume the currently executing process or to switch to a different process. This interval must be set short enough to ensure that the processor will switch between tasks often enough to provide the illusion of performing many tasks simultaneously. On the other hand, switching from one process to another requires thousands of clock cycles to save the state of the current process and to set up the state for the next, and hence setting the interval too short would cause poor performance. Typical timer intervals range between 1 and 10 milliseconds, depending on the processor and how it is configured.

Figure 9.2(a) illustrates the system's perspective of a hypothetical 150 ms of operation on a system with a 10 ms timer interval. During this period there are two active processes: A and B. The processor alternately executes part of process A, then part of B, and so on. As it executes these processes, it operates either in *user mode*, executing the instructions of the application program; or in *kernel mode*, performing operating system functions on behalf of the program, such as, handling page faults, input, or output. Recall that kernel

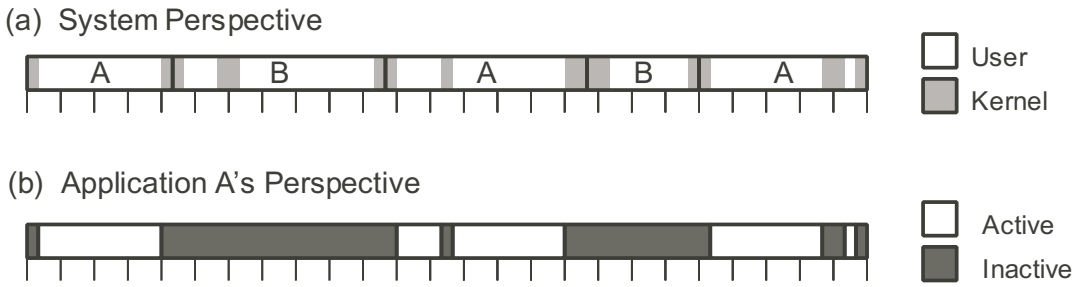


Figure 9.2: **System's vs. Applications View of Time.** The system switches from process to process, operating in either user or kernel mode. The application only gets useful computation done when its process is executing in user mode.

operation is considered part of each regular process rather than a separate process. The operating system scheduler is invoked every time there is an external event or a timer interrupt. The occurrences of timer interrupts are indicated by the tick marks in the figure. This means that there is actually some amount of kernel activity at every tick mark, but for simplicity we do not show it in the figure.

When the scheduler switches from process A to process B, it must enter kernel mode to save the state of process A (still considered part of process A) and to restore the state of process B (considered part of process B). Thus, there is kernel activity during each transition from one process to another. At other times, kernel activity can occur without switching processes, such as when a page fault can be satisfied by using a page that is already in memory.

9.1.2 Time from an Application Program's Perspective

From the perspective of an application program, the flow of time can be viewed as alternating between periods when the program is *active* (executing its instructions), and *inactive* (waiting to be scheduled by the operating system). It only performs useful computation when its process is operating in user mode. Figure 9.2(b) illustrates how program A would view the flow of time. It is active during the light-colored regions, when process A is executing in user mode; otherwise it is inactive.

As a way to quantify the alternations between active and inactive time periods, we wrote a program that continuously monitors itself and determines when there have been long periods of inactivity. It then generates a *trace* showing the alternations between periods of activity and inactivity. Details of this program are described later in the chapter. An example of such a trace is shown in Figure 9.3, generated while running on a Linux machine with a clock rate of around 550 MHz. Each period is labeled as either active ("A") or inactive "I"). The periods are numbered 0 to 9 for identification. For each period, the start time (relative to the beginning of the trace) and the duration are indicated. Times are expressed in both clock cycles and milliseconds. This trace shows a total of 20 time periods (10 active and 10 inactive) having a total duration of 66.9 ms. In this example, the periods of inactivity are fairly short, with the longest being 0.50 ms. Most of these periods of inactivity were caused by timer interrupts. The process was active for around 95.1% of the total time monitored. Figure 9.4 shows a graphical rendition of the trace shown in Figure 9.3. Observe the regular spacing of the boundaries between the activity periods indicated by the gray triangles. These

```
A0    Time        0    (0.00 ms),   Duration  3726508    (6.776448 ms)
I0    Time  3726508    (6.78 ms),   Duration   275025    (0.500118 ms)
A1    Time  4001533    (7.28 ms),   Duration        0    (0.000000 ms)
I1    Time  4001533    (7.28 ms),   Duration     7598    (0.013817 ms)
A2    Time  4009131    (7.29 ms),   Duration  5189247    (9.436358 ms)
I2    Time  9198378   (16.73 ms),   Duration   251609    (0.457537 ms)
A3    Time  9449987   (17.18 ms),   Duration  2250102    (4.091686 ms)
I3    Time 11700089   (21.28 ms),   Duration    14116    (0.025669 ms)
A4    Time 11714205   (21.30 ms),   Duration  2955974    (5.375275 ms)
I4    Time 14670179   (26.68 ms),   Duration   248500    (0.451883 ms)
A5    Time 14918679   (27.13 ms),   Duration  5223342    (9.498358 ms)
I5    Time 20142021   (36.63 ms),   Duration   247113    (0.449361 ms)
A6    Time 20389134   (37.08 ms),   Duration  5224777    (9.500967 ms)
I6    Time 25613911   (46.58 ms),   Duration   254340    (0.462503 ms)
A7    Time 25868251   (47.04 ms),   Duration  3678102    (6.688425 ms)
I7    Time 29546353   (53.73 ms),   Duration     8139    (0.014800 ms)
A8    Time 29554492   (53.74 ms),   Duration  1531187    (2.784379 ms)
I8    Time 31085679   (56.53 ms),   Duration   248360    (0.451629 ms)
A9    Time 31334039   (56.98 ms),   Duration  5223581    (9.498792 ms)
I9    Time 36557620   (66.48 ms),   Duration   247395    (0.449874 ms)
```

Figure 9.3: **Example Trace Showing Activity Periods.** From the perspective of an application program, processor operation alternates between periods when the program is actively executing (italicized) and when it is inactive. This trace shows a log of these periods for a program over a total duration of 66.9 ms. The program was active for 95.1% of this time.

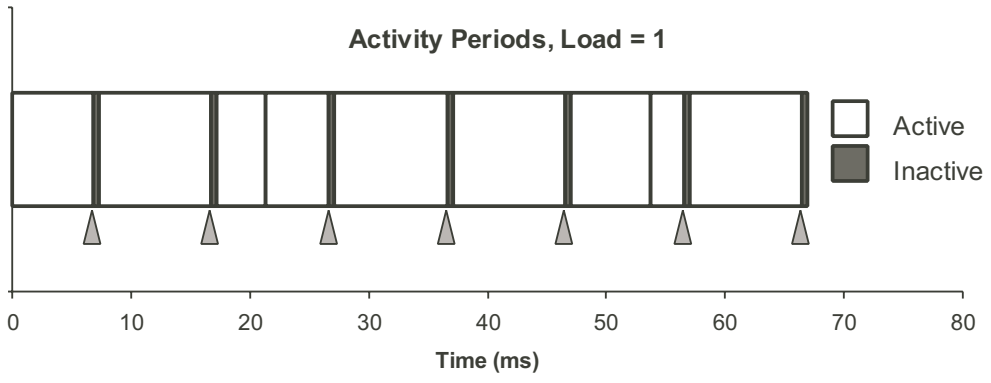


Figure 9.4: **Graphical Representation of Trace in Figure 9.3.** Timer interrupts are indicated with gray triangles.

```
A48   Time 191514104   (349.40 ms),   Duration  5224961   (9.532449 ms)
I48   Time 196739065   (358.93 ms),   Duration   247557   (0.451644 ms)
A49   Time 196986622   (359.38 ms),   Duration   858571   (1.566382 ms)
I49   Time 197845193   (360.95 ms),   Duration     8297   (0.015137 ms)
A50   Time 197853490   (360.97 ms),   Duration  4357437   (7.949733 ms)
I50   Time 202210927   (368.91 ms),   Duration  5718758  (10.433335 ms)
A51   Time 207929685   (379.35 ms),   Duration  2047118   (3.734774 ms)
I51   Time 209976803   (383.08 ms),   Duration     7153   (0.013050 ms)
A52   Time 209983956   (383.10 ms),   Duration  3170650   (5.784552 ms)
I52   Time 213154606   (388.88 ms),   Duration  5726129  (10.446783 ms)
A53   Time 218880735   (399.33 ms),   Duration  5217543   (9.518916 ms)
I53   Time 224098278   (408.85 ms),   Duration  5718135  (10.432199 ms)
A54   Time 229816413   (419.28 ms),   Duration  2359281   (4.304286 ms)
I54   Time 232175694   (423.58 ms),   Duration     7096   (0.012946 ms)
A55   Time 232182790   (423.60 ms),   Duration  2859227   (5.216390 ms)
I55   Time 235042017   (428.81 ms),   Duration  5718793  (10.433399 ms)
```

Figure 9.5: **Example Trace Showing Activity Periods on Loaded Machine.** When other active processes are present, the tracing process is inactive for longer periods of time. This trace shows a log of these periods for a program over a total duration of 89.8 ms. The process was active for 53.0% of this time.

boundaries are caused by timer interrupts.

Figure 9.5 shows a portion of a trace when there is one other active process sharing the processor. The graphical rendition of this trace is shown in Figure 9.6. Note that the time scales do not line up, since the portion of the trace we show in Figure 9.5 started at 349.4 ms into the tracing process. In this example we can see that while handling some of the timer interrupts, the OS also decides to switch context from one process to another. As a result, each process is only active around 50% of the time.

Practice Problem 9.2:

This problem concerns the interpretation of the section of the trace shown in Figure 9.5.

A. At what times during this portion of the trace did timer interrupts occur? (Some of these time points can be extracted directly from the trace, while others must be estimated by interpolation.)

B. Which of these occurred while the tracing process was active, and which while it was inactive?

C. Why are the longest periods of inactivity longer than the longest periods of activity?

D. Based on the pattern of active and inactive periods shown in this trace, what percent of the time would you expect the tracing process to be inactive when averaged over a longer time scale?

9.2 Measuring Time by Interval Counting

The operating system also uses the timer to record the cumulative time used by each process. This information provides a somewhat imprecise measure of program execution time. Figure 9.7 provides a graphic illustration of how this accounting works for the example of system operation shown in Figure 9.2. In this discussion, we refer to the period during which just one process executes as a *time segment*.

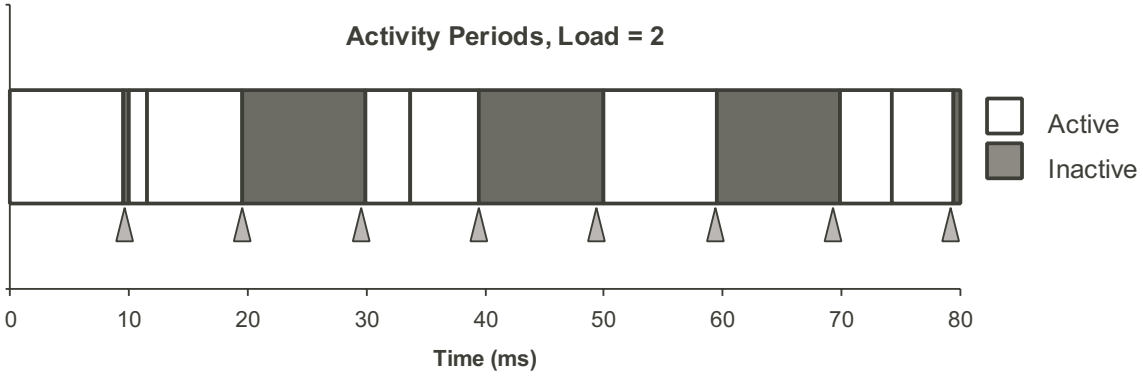


Figure 9.6: **Graphical Representation of Activity Periods for Trace in Figure 9.5.** Timer interrupts are indicated by gray triangles

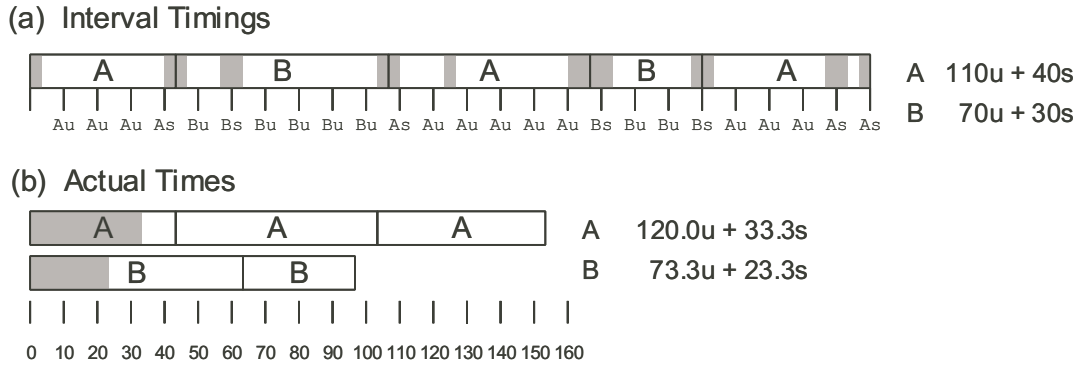


Figure 9.7: **Process Timing by Interval Counting.** With a timer interval of 10 ms, every 10 ms segment is assigned to a process as part of either its user (u) or system (s) time. This accounting provides only an approximate measure of program execution time.

9.2.1 Operation

The operating system maintains counts of the amount of user time and the amount of system time used by each process. When a timer interrupt occurs, the operating system determines which process was active and increments one of the counts for that process by the timer interval. It increments the system time if the system was executing in kernel mode, and the user time otherwise. The example shown in Figure 9.7(a) indicates this accounting for the two processes. The tick marks indicate the occurrences of timer interrupts. Each is labeled by the count that gets incremented: either `Au` or `As` for process A's user or system time, or `Bu` or `Bs` for process B's user or system time. Each tick mark is labeled according to the activity to its immediate left. The final accounting shows that process A used a total of 150 milliseconds: 110 of user time and 40 of system time. It shows that B used a total of 100 milliseconds: 70 of user time and 30 of system time.

9.2.2 Reading the Process Timers

When executing a command from the Unix shell, the user can prefix the command with the word "`time`" to measure the execution time of the command. This command uses the values computed using the accounting scheme described above. For example, to time the execution time of program `prog` with command line arguments `-n 17`, the user can simply type the command:

```
unix> time prog -n 17
```

After the program has executed, the shell will print a line summarizing the run time statistics, for example,

```
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
```

The first three numbers shown in this line are times. The first two show the seconds of user and system time. Observe how both of these show a 0 in the third decimal place. With a timer interval of 10 ms, all timings are multiples of hundredths of seconds. The third number is the total elapsed time, given in minutes and seconds. Observe that the system and user time sum to 2.49 seconds, less than half of the elapsed time of 6.52 seconds, indicating that the processor was executing other processes at the same time. The percentage indicates what fraction the combined user and system times were of the elapsed time, e.g., $(2.23 + 0.26)/6.52 = 0.381$. The remaining statistics summarize the paging and I/O behavior.

Programmers can also read the process timers by calling the library function `times`, declared as follows:

```
#include <sys/times.h>

struct tms {
   clock_t tms_utime; /* user time */
   clock_t tms_stime; /* system time */
   clock_t tms_cutime; /* user time of reaped children */
   clock_t tms_cstime; /* system time of reaped children */
};

clock_t times(struct tms *buf);
```

Returns: number of clock ticks elapsed since system started

These time measurements are expressed in terms of a unit called *clock ticks*. The defined constant `CLK_TCK` specifies the number of clock ticks per second. The data type `clock_t` is typically defined to be a long integer. The fields indicating child times give the accumulated times used by children that have terminated and have been reaped. Thus, `times` cannot be used to monitor the time used by any ongoing children. As a return value, `times` returns the total number of clock ticks that have elapsed since the system was started. We can therefore compute the total time (in clock ticks) between two different points in a program execution by making two calls to `times` and computing the difference of the return values.

The ANSI C standard also defines a function `clock` that measures the total time used by the current process:

```
#include <time.h>

clock_t clock(void);
```

Returns: total time used by process

Although the return value is declared to be the same type `clock_t` used with the `times` function, the two functions do not, in general, express time in the same units. To scale the time reported by `clock` to seconds, it should be divided by the defined constant `CLOCKS_PER_SEC`. This value need not be the same as the constant `CLK_TCK`.

9.2.3 Accuracy of Process Timers

As the example illustrated in Figure 9.7 shows, this timing mechanism is only approximate. Figure 9.7(b) shows the actual times used by the two processes. Process A executed for a total of 153.3 ms, with 120.0 in user mode and 33.3 in kernel mode. Process B executed for a total of 96.7 ms, with 73.3 in user mode and 23.3 in kernel mode. The interval accounting scheme makes no attempt to resolve time more finely than the timer interval.

Practice Problem 9.3:

What would the operating system report as the user and system times for the execution sequence illustrated below. Assume a 10 ms timer interval.

Practice Problem 9.4:

On a system with a timer interval of 10 ms, some segment of process A is recorded as requiring 70 ms, combining both system and user time. What are the minimum and maximum actual times used by this segment?

Practice Problem 9.5:

What would the counters record as the system and user times for the trace shown in Figure 9.3? How does this compare to the actual time during which the process was active?

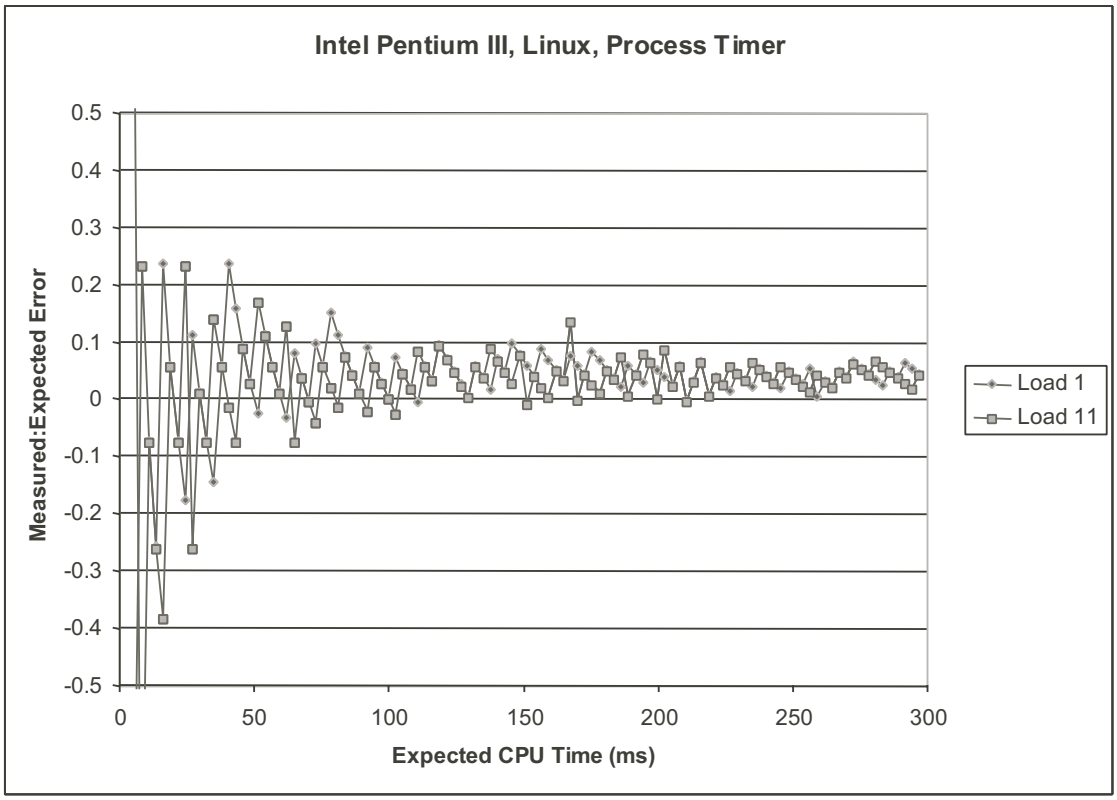


Figure 9.8: **Experimental Results for Measuring Interval Counting Accuracy.** The error is unacceptably high when measuring activities less than around 100 ms (10 timer intervals). Beyond this, the error rate is generally less than 10% regardless of whether running on lightly loaded (Load 1) or heavily loaded (Load 11) machine.

For programs that run long enough, (at least several seconds), the inaccuracies in this scheme tend to compensate for each other. The execution times of some segments are underestimated while those of others are overestimated. Averaged over a number of segments, the expected error approaches zero. From a theoretical perspective, however, there is no guaranteed bound on how far these measurements vary from the true run times.

To test the accuracy of this timing method, we ran a series of experiments that compared the time T_m measured by the operating system for a sample computation versus our estimate of what the time T_c would be if the system resources were dedicated solely to performing this computation. In general, T_c will differ from T_m for several reasons:

1. The inherent inaccuracies of the interval counting scheme can cause T_m to be either less or greater than T_c.

2. The kernel activity caused by the timer interrupt consumes 4 to 5% of the total CPU cycles, but these cycles are not accounted for properly. As can be seen in the trace illustrated in Figure 9.4, this activity finishes before the next timer interrupt and hence does not get counted explicitly. Instead, it simply

reduces the number of cycles available for the process executing during the next time interval. This will tend to increase T_m relative to T_c.

3. When the processor switches from one task to another, the cache tends to perform poorly for a transient period until the instructions and data for the new task get loaded into the cache. Thus the processor does not run as efficiently when switching between our program and other activities as it would if it executed our program continuously. This factor will tend to increase T_m relative to T_c.

We discuss how we can determine the value of T_c for our sample computation later in this chapter.

Figure 9.8 shows the results of this experiment running under two different loading conditions. The graphs show our measurements of the error rate, defined as the value of $(T_m - T_c)/T_c$ as a function of T_c. This error measure is negative when T_m underestimates T_c and is positive when T_m overestimates T_c. The two series show measurements taken under two different loading conditions. The series labeled "Load 1" shows the case where the process performing the sample computation is the only active process. The series labeled "Load 11" shows the case where 10 other processes are also attempting the same computation. The latter represents a very heavy load condition; the system is noticeably slow responding to keystrokes and other service requests. Observe the wide range of error values shown on this graph. In general, only measurements that are within $\pm$10% of the true value are acceptable, and hence we want only errors ranging from around -0.1 to $+0.1$.

Below around 100 ms (10 timer intervals), the measurements are not at all accurate due to the coarseness of the timing method. Interval counting is only useful for measuring relatively long computations—100,000,000 clock cycles or more. Beyond this, we see that the error generally ranges between 0.0 and 0.1, that is, up to 10% error. There is no noticeable difference between the two different loading conditions. Notice also that the errors have a positive bias; the average error for all measurements with $T_m \geq 100$ms is 1.04, due to the fact that the timer interrupts are consuming around 4% of the CPU time.

These experiments show that the process timers are useful only for getting approximate values of program performance. They are too coarse-grained to use for any measurement having duration of less than 100 ms. On this machine they have a systematic bias, overestimating computation times by an average of around 4%. The main virtue of this timing mechanism is that its accuracy does not depend strongly on the system load.

9.3 Cycle Counters

To provide greater precision for timing measurements, many processors also contain a timer that operates at the clock cycle level. This timer is a special register that gets incremented every single clock cycle. Special machine instructions can be used to read the value of the counter. Not all processors have such counters, and those that do vary in the implementation details. As a result, there is no uniform, platform-independent interface by which programmers can make use of these counters. On the other hand, with just a small amount of assembly code, it is generally easy to create a program interface for any specific machine.

9.3.1 IA32 Cycle Counters

All of the timings we have reported so far were measured using the IA32 cycle counter. With the IA32 architecture, cycle counters were introduced in conjunction with the "P6" microarchitecture (the PentiumPro and its successors). The cycle counter is a 64-bit, unsigned number. For a processor operating with a 1 GHz clock, this counter will wrap around from $2^{64} - 1$ to 0 only once every 1.8×10^{10} seconds, or every 570 years. On the other hand, if we consider only the low order 32 bits of this counter as an unsigned integer, this value will wrap around every 4.3 seconds. One can therefore understand why the IA32 designers decided to implement a 64-bit counter.

The IA32 counter is accessed with the `rdtsc` (for "read time stamp counter") instruction. This instruction takes no arguments. It sets register `%edx` to the high-order 32 bits of the counter and register `%eax` to the low-order 32 bits. To provide a C program interface, we would like to encapsulate this instruction within a procedure:

```
void access_counter(unsigned *hi, unsigned *lo);
```

This procedure should set location `hi` to the high-order 32 bits of the counter and `lo` to the low-order 32 bits. Implementing `access_counter` is a simple exercise in using the embedded assembly feature of GCC, as described in Section 3.15. The code is shown in Figure 9.9.

Based on this routine, we can now implement a pair of functions that can be used to measure the total number of cycles that elapse between any two time points:

```
#include "clock.h"

void start_counter();
double get_counter();
```

Returns: number of cycles since last call to `start_counter`

We return the time as a `double` to avoid the possible overflow problems of using just a 32-bit integer. The code for these two routines is also shown in Figure 9.9. It builds on our understanding of unsigned arithmetic to perform the double-precision subtraction and to convert the result to a `double`.

9.4 Measuring Program Execution Time with Cycle Counters

Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program. Typically, however, we are interested in measuring the time required to execute some particular piece of code. Our cycle counter routines compute the total number of cycles between a call to `start_counter` and a call to `get_counter`. They do not keep track of which process uses those cycles or whether the processor is operating in kernel or user mode. We must be careful when using such a measuring device to determine execution time. We investigate some of these difficulties and how they can be overcome.

As an example of code that uses the cycle counter, the routine in Figure 9.10 provides a way to determine the clock rate of a processor. Testing this function on several systems with parameter `sleeptime` equal

code/perf/clock.c

```
1 /* Initialize the cycle counter */
2 static unsigned cyc_hi = 0;
3 static unsigned cyc_lo = 0;
4
5
6 /* Set *hi and *lo to the high and low order bits  of the cycle counter.
7    Implementation requires assembly code to use the rdtsc instruction. */
8 void access_counter(unsigned *hi, unsigned *lo)
9 {
10     asm("rdtsc; movl %%edx,%0; movl %%eax,%1"   /* Read cycle counter */
11         : "=r" (*hi), "=r" (*lo)                /* and move results to */
12         : /* No input */                        /* the two outputs */
13         : "%edx", "%eax");
14 }
15
16 /* Record the current value of the cycle counter. */
17 void start_counter()
18 {
19     access_counter(&cyc_hi, &cyc_lo);
20 }
21
22 /* Return the number of cycles since the last call to start_counter. */
23 double get_counter()
24 {
25     unsigned ncyc_hi, ncyc_lo;
26     unsigned hi, lo, borrow;
27     double result;
28
29     /* Get cycle counter */
30     access_counter(&ncyc_hi, &ncyc_lo);
31
32     /* Do double precision subtraction */
33     lo = ncyc_lo - cyc_lo;
34     borrow = lo > ncyc_lo;
35     hi = ncyc_hi - cyc_hi - borrow;
36     result = (double) hi * (1 << 30) * 4 + lo;
37     if (result < 0) {
38         fprintf(stderr, "Error: counter returns neg value: %.0f\n", result);
39     }
40     return result;
41 }
```

code/perf/clock.c

Figure 9.9: **Code Implementing Program Interface to IA32 Cycle Counter** Assembly code is required to make use of the counter reading instruction.

code/perf/clock.c

```
1 /* Estimate the clock rate by measuring the cycles that elapse */
2 /* while sleeping for sleeptime seconds */
3 double mhz(int verbose, int sleeptime)
4 {
5     double rate;
6
7     start_counter();
8     sleep(sleeptime);
9     rate = get_counter() / (1e6*sleeptime);
10    if (verbose)
11        printf("Processor clock rate ~= %.1f MHz\n", rate);
12    return rate;
13 }
```

code/perf/clock.c

Figure 9.10: `mhz`**: Determines the clock rate of a processor.**

to 1 shows that it reports a clock rate within 1.0% of the rated performance for the processor. This example clearly shows that our routines measure elapsed time rather than the time used by a particular process. When our program calls `sleep`, the operating system will not resume the process until the sleep time of one second has expired. The cycles that elapse during that time are spent executing other processes.

9.4.1 The Effects of Context Switching

A naive way to measuring the run time of some procedure `P` is to simply use the cycle counter to time one execution of `P`, as in the following code:

```
1 double time_P()
2 {
3     start_counter();
4     P();
5     return get_counter();
6 }
```

This could easily yield misleading results if some other process also executes between the two calls to the counter routines. This is especially a problem if either the machine is heavily loaded, or if the run time for `P` is especially long. This phenomenon is illustrated in Figure 9.11. This figure shows the result of repeatedly measuring a program that computes the sum of an array of 131,072 integers. The times have been converted into milliseconds. Note that the run times are all over 36 ms, greater than the timer interval. Two trials were run, each measuring 18 executions of the exact same procedure. The series labeled “Load 1” indicates the run times on a lightly loaded machine, where this is the only process actively running. All of the measurements are within 3.4% of the minimum run time. The series labeled “Load 4” indicates the run times when three other processes making heavy use of the CPU and memory system are also running.

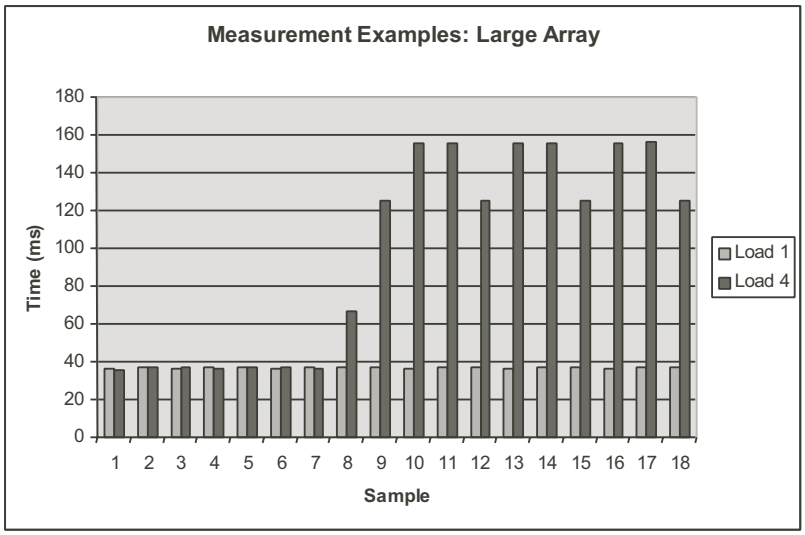


Figure 9.11: **Measurements of Long Duration Procedure under Different Loading Conditions** On a lightly loaded system, the results are consistent across samples, but on a heavily loaded system, many of the measurements overestimate the true execution time.

The first seven of these samples have times within 2% of the fastest Load 1 sample, but others range as much as 4.3 times greater.

As this example illustrates, context switching causes extreme variations in execution time. If a process is swapped out for an entire time interval it will fall behind by millions of instructions. Clearly, any scheme we devise to measure program execution times must avoid such large errors.

9.4.2 Caching and Other Effects

The effects of caching and branch prediction create smaller timing variations than does context switching. As an example, Figure 9.12 shows a series of measurements similar to those in Figure 9.11, except that the array is 4 times smaller, yielding execution times of around 8 ms. These execution times are shorter than the timer interval and therefore the executions are less likely to be affected by context switching. We see significant variations among the measurements—the slowest is 1.1 times slower the fastest, but none of these variations are as extreme as would be caused by context switching.

The variations shown in Figure 9.12 are due mainly to cache effects. The time to execute a block of code can depend greatly on whether or not the data and the instructions used by this code are present in the data and instruction caches at the beginning of execution.

As an example, we wrote two identical procedures, `procA` and `procB`, that are given a pointer of type `double *` and set the eight consecutive elements starting at this pointer to 0.0. We measured the number of clock cycles for various calls to these procedures with three different pointers: `b1`, `b2`, and `b3`. The call sequence and the resulting measurements are shown in Figure 9.13. The timings vary by almost a factor of 4, even though the calls perform identical computations. There were no conditional branches in this code, and hence we conclude that the variations must be due to cache effects.

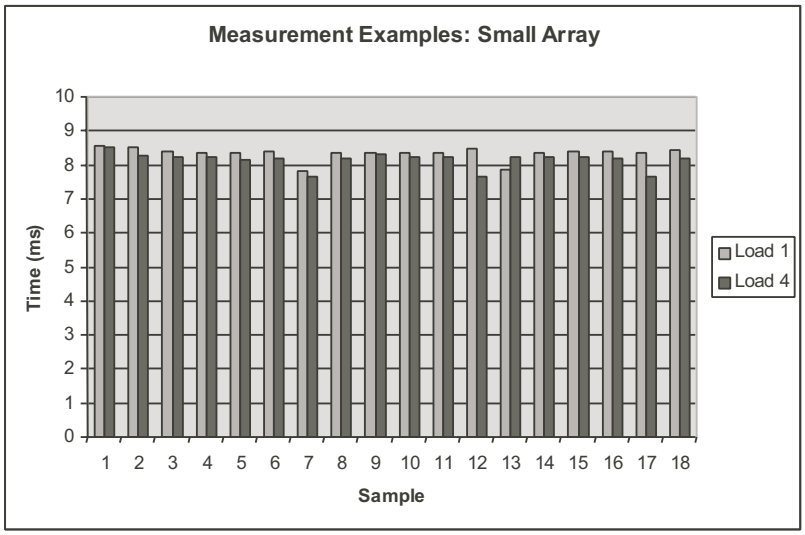


Figure 9.12: **Measurements of Short Duration Procedure under Different Loading Conditions** The variations are not as extreme as they were in Figure 9.11, but they are still unacceptably large.

Measurement	Call	Cycles
1	`procA(b1)`	399
2	`procA(b2)`	132
3	`procA(b3)`	134
4	`procA(b1)`	100
5	`procB(b1)`	317
6	`procB(b2)`	100

Figure 9.13: **Measurement Sequence with Identical Procedures Operating on Identical Data Sets.** The variations in these measurements are due to different miss conditions in the instruction and data caches.

Practice Problem 9.6:

Let c be the number of cycles that would be required by a call to `procA` or `procB` if there were no cache misses. For each computation, the cycles wasted due to cache misses can be apportioned between the different items needing to be brought into the cache:

- The instructions implementing the measurement code (e.g., `start_counter`, `get_counter`, and so on). Let the number of cycles for this be m.
- The instructions implementing the procedure being measured (`procA` or `procB`). Let the number of cycles for this be p.
- The data locations being updated (designated by `b1`, `b2`, or `b3`). Let the number of cycles for this be d.

Based on the measurements shown in Figure 9.13, give estimates of the values of c, m, p, and d.

Given the variations shown in these measurements, a natural question to ask is "Which one is right?" Unfortunately, the answer to this question is not simple. It depends on both the conditions under which our code will actually be used as well as the conditions under which we can get reliable measurements. One problem is that the measurements are not even consistent from one run to the next. The measurement table shown in Figure 9.13 show the data for just one testing run. In repeated tests, we have seen Measurement 1 range from 317 and 606, and Measurement 5 range from 301 to 326. On the other hand, the other four measurements only vary by at most a few cycles from one run to another.

Clearly Measurement 1 is an overestimate, because it includes the cost of loading the measurement code and data structures into cache. Furthermore, it is the most subject to wide variations. Measurement 5 includes the cost of loading `procB` into the cache. This is also subject to significant variations. In most real applications, the same code is executed repeatedly. As a result, the time to load the code into the instruction cache will be relatively insignificant. Our example measurements are somewhat artificial in that the effects of instruction cache misses were proportionally greater than what would occur in a real application.

To measure the time required by a procedure `P` where the effects of instruction cache misses are minimized we can execute the following code:

```
1 double time_P_warm()
2 {
3     P(); /* Warm up the cache */
4     start_counter();
5     P();
6     return get_counter();
7 }
```

Executing `P` once before starting the measurement will have the effect of bringing the code used by `P` into the instruction cache.

The code above also minimizes the effects of data cache misses, since the first execution of `P` will also have the effect of bringing the data accessed by `P` into the data cache. For procedures `procA` or `procB`, a measurement by `time_P_warm` would yield 100 cycles. This would be the right conditions to measure if we expect our code to access the same data repeatedly. For some applications, however, we would be more

likely to access new data with each execution. For example, a procedure that copies data from one region of memory to another would most likely be called under conditions where neither block is cached. Procedure `time_P_warm` would tend to underestimate the execution time for such a procedure. For `procA` or `procB`, it would yield 100 rather than the 132 to 134 measured when the procedure is applied to uncached data.

To force the timing code to measure the performance of a procedure where none of the data is initially cached, we can flush the cache of any useful data before performing the actual measurement. The following procedure does this for a system with caches of no more than 512KB:

code/perf/time_p.c

```
1 /* Number of bytes in the largest cache to be cleared */
2 #define CBYTES (1<<19)
3 #define CINTS (CBYTES/sizeof(int))
4
5 /* A large array to bring into cache */
6 static int dummy[CINTS];
7 volatile int sink;
8
9 /* Evict the existing blocks from the data caches */
10 void clear_cache()
11 {
12     int i;
13     int sum = 0;
14
15     for (i = 0; i < CINTS; i++)
16         dummy[i] = 3;
17     for (i = 0; i < CINTS; i++)
18         sum += dummy[i];
19     sink = sum;
20 }
```

code/perf/time_p.c

This procedure simply performs a computation over a very large array `dummy`, effectively evicting everything else from the cache. The code has several peculiar features to avoid common pitfalls. It both stores values into `dummy` and reads them back so that it will be cached regardless of the cache allocation policy. It performs a computation using array values and stores the result to a global integer (the declaration `volatile` indicates that any update to this variable must be performed), so that a clever optimizing compiler will not optimize away this part of the code.

With this procedure, we can get a measurement of `P` under conditions where its instructions are cached but its data is not by the following procedure:

```
1 double time_P_cold()
2 {
3     P(); /* Warm up data caches */
4     clear_cache(); /* Clear data caches */
5     start_counter();
```

```
6     P();
7     return get_counter();
8 }
```

Of course, even this method has deficiencies. On a machine with a unified L2 cache, procedure `clear_cache` will cause all instructions from `P` to be evicted. Fortunately, the instructions in the L1 instruction cache will remain. Procedure `clear_cache` also evicts much of the runtime stack from the cache, leading to an overestimate of the time required by `P` under more realistic conditions.

As this discussion shows, the effects of caching pose particular difficulties for performance measurement. Programmers have little control over what instructions and data get loaded into the caches and what gets evicted when new values must be loaded. At best, we can set up measurement conditions that somewhat match the anticipated conditions of our application by some combination of cache flushing and loading.

As mentioned earlier, the branch prediction logic also influences program performance, since the time penalty caused by branch instruction is much less when the branch direction and target are correctly predicted. This logic makes its predictions based on the past history of branch instructions that have been executed. When the system switches from one process to another, it initially makes predictions about branches in the new process based on those executed in the previous process. In practice, however, these effects create only minor performance variations from one execution of a program to another. The predictions depend most strongly on recent branches, and hence the influence by one process on another is very small.

9.4.3 The K-Best Measurement Scheme

Although our measurements using cycle timers are vulnerable to errors due to context switching, cache operation, and branch prediction, one important feature is that the errors will always cause overestimates of the true execution time. Nothing done by the processor can artificially speed up a program. We can exploit this property to get reliable measurements of execution times even when there are variances due to context switching and other effects.

Suppose we repeatedly execute a procedure and measure the number of cycles using either `time_P_warm` or `time_P_cold`. We record the K (e.g., 3) fastest times. If we find these measurements agree within some small tolerance ϵ (e.g., 0.1%), then it seems reasonable that the fastest of these represents the true execution time of the procedure. As an example, suppose for the runs shown in Figure 9.11 we set the tolerance to 1.0%. Then the fastest six measurements for Load 1 are within this tolerance, as are the fastest three for Load 4. We would therefore conclude that the run times are 35.98 ms and 35.89 ms, respectively. For the Load 4 case, we also see measurements clustered around 125.3 ms, and six around 155.8 ms, but we can safely discard these as overestimates.

We call this approach to measurement the "K-Best Scheme." It requires setting three parameters:

K: The number of measurements we require to be within some close range of the fastest.

ϵ: How close the measurements must be. That is, if the measurements in ascending order are labeled $v_1, v_2, \ldots, v_i, \ldots$, then we require $(1 + \epsilon)v_1 \geq v_K$.

M: The maximum number of measurements before we give up.

Our implementation performs a series of trials, and maintains an array of the K fastest times in sorted order. With each new measurement, it checks whether it is faster than the current one in array position K. If so, it replaces array element K and then performs a series of interchanges between adjacent array positions to move this value to the appropriate position in the array. This process continues until either the error criterion is satisfied, in which case we indicate that the measurements have "converged," or we exceed the limit M, in which case we indicate that the measurements failed to converge.

Experimental Evaluation

We conducted a series of experiments to test the accuracy of the K-best measurement scheme. Some issues we wished to determine were:

1. Does this scheme produce accurate measurements?
2. When and how quickly do the measurements converge?
3. Can the scheme determine the accuracy of its own measurements?

One challenge in designing such an experiment is to know the actual run times of the programs we are trying to measure. Only then can we determine the accuracy of our measurements. We know that our cycle timer gives accurate results as long as the computation we are measuring do not get interrupted. The likelihood of an interrupt is small for computations that are much shorter than the timer interval and when running on a lightly loaded machine. We exploit these properties to get reliable estimates of true run times.

As our object to measure, we used a procedure that repeatedly writes values to an array of 2,048 integers and then reads them back, similar to the code for `clear_cache`. By setting the number of repetitions r, we could create computations requiring a range of times. We first determined the *expected* run time of this procedure as a function of r, denoted $T(r)$, by timing it for r ranging from 1 to 10 (giving times ranging from 0.09 to 0.9 milliseconds), and performing a least squares fit to find a formula of the form $T(r) = mr+b$. By using small values of r, performing 100 measurements for each value of r, and running on a lightly loaded system we were able to get a very accurate characterization of $T(r)$. Our least squares analysis indicated that the formula $T(r) = 49273.4r + 166$ (in units of clock cycles) fits this data with a maximum error less than 0.04%. This gave us confidence in our ability to accurately predict the actual computation time for the procedure as a function of r.

We then measured performance using the K-best scheme with parameters $K = 3$, $\epsilon = 0.001$, and $M = 30$. We did this for a number of values of r to get expected run times in a range from 0.27 to 50 milliseconds. For each of the resulting measurements $M(r)$ we computed the measurement error $E_m(r)$ as $E_m(r) = (M(r) - T(r))/T(r)$. Figure 9.14 shows an experimental validation of the K-best scheme on an Intel Pentium III running Linux. In this figure we show the measurement error $E_m(r)$ as a function of $T(r)$, where we show $T(r)$ in units of milliseconds. Note that we show $E_m(r)$ on a logarithmic scale; each horizontal line represents an order of magnitude difference in measurement error. In order to be accurate within 1% we must have an error below 0.01. We do not attempt to show any errors smaller than 0.001 (i.e., 0.1%), since our testing setup does not provide high enough precision for this.

The three series indicate the errors under three different loading conditions. Observe that in all three cases the measurements for run times shorter than around 7.5 ms were very accurate. Thus, our scheme can be

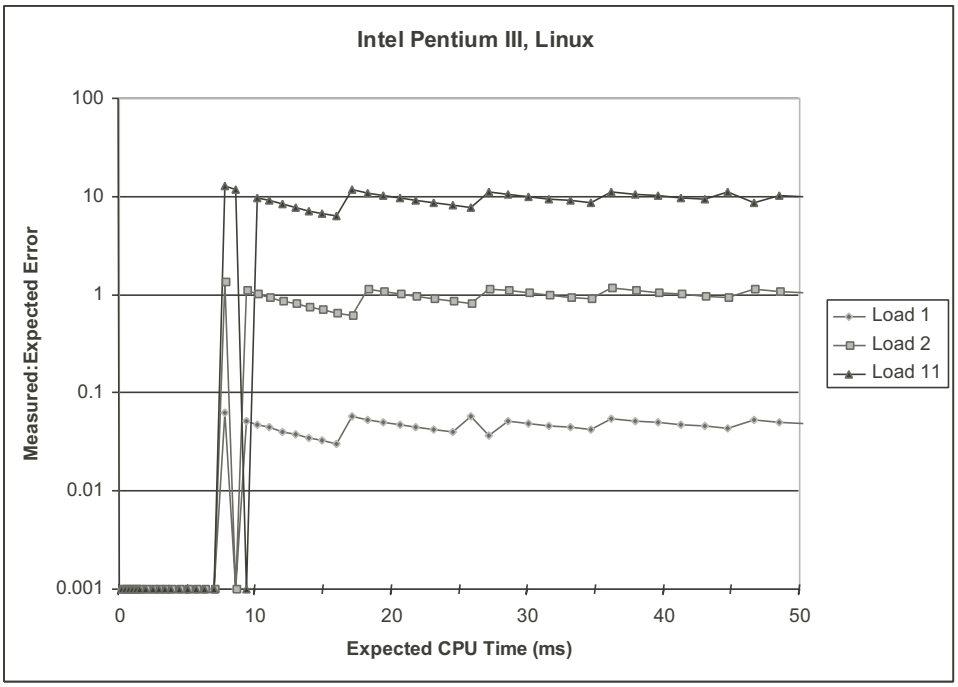


Figure 9.14: **Experimental Validation if K-Best Measurement Scheme on Linux System** We can consistently obtain very accurate measurements (around 0.1% error) for execution times up to around 8 ms. Beyond this, we encounter a systematic overestimate of around 4 to 6% on a lightly loaded machine and very poor results on a heavily loaded machine.

used to measure relatively short execution times even on a heavily loaded machine. Series "Load 1" indicates the case where there is only one active process. For execution times above 10 ms, the measurements T_m consistently overestimate the computation times T_c by around 4 to 6%. These overestimates are due to the time spent handling timer interrupts. They are consistent with the trace shown in Figure 9.3, showing that even on a lightly loaded machine, an application program can execute for only 95 to 96% of the time. Series "Load 2" and "Load 11" show the performance when other processes are actively executing. In both cases, the measurements become hopelessly inaccurate for execution times above around 7 ms. Note that an error of 1.0 means that T_m is twice T_c, while an error of 10.0 means that T_m is eleven times greater than T_c. Evidently, the operating system schedules each active process for one time interval. When n processes are active, each one only gets $1/n$th of the processor time.

From these results, we conclude that the K-best scheme provides accurate results only for very short computations. It is not really good enough for measuring execution times longer than around 7 ms, especially in the presence of other active processes.

Unfortunately, we found that our measurement program could not reliably determine whether or not it had obtained an accurate measurement. Our measurement procedure computes a prediction of its error as $E_p(r) = (v_k - v_1)/v_1$, where v_i is the i th smallest measurement. That is, it computes how well it achieves our convergence criterion. We found these estimates to be wildly optimistic. Even for the Load 11 case, where the measurements were off by a factor of 10, the program consistently estimated its error to be less than 0.001.

Setting the value of K

In our earlier experiments, we arbitrarily chose a value of 3 for the parameter K, determining the number of measurements we require to be within a small factor of the fastest in order to terminate. To more carefully evaluate the effect of this factor, we performed a series of measurements using values of K ranging from 1 to 5, as shown in Figure 9.15. We performed these measurements for execution times ranging up to 9 ms, since this is the upper limit of times for which our scheme can get useful results.

When we have $K = 1$, the procedure returns after making a single measurement. This can yield highly erratic results, especially when the machine is heavily loaded. If a timer interrupt happens to occur, the result is extremely inaccurate. Even without such a catastrophic event, the measurements will be subject to many sources of inaccuracy. Setting K to 2 greatly improves the accuracy. For execution times less than 5 ms, we consistently get accuracy better than 0.1%. Setting K even higher gives better results, both in consistency and accuracy, up to a limit of around 8 ms. These experiments show that our initial guess of $K = 3$ is a reasonable choice.

Compensating for Timer Interrupt Handling

The timer interrupts occur in a predictable way and cause a large systematic error in our measurements for execution times over around 7 ms. It would be good to remove this bias by subtracting from the measured run time for a program an estimate of the time spent handling timer interrupts. This requires determining two factors.

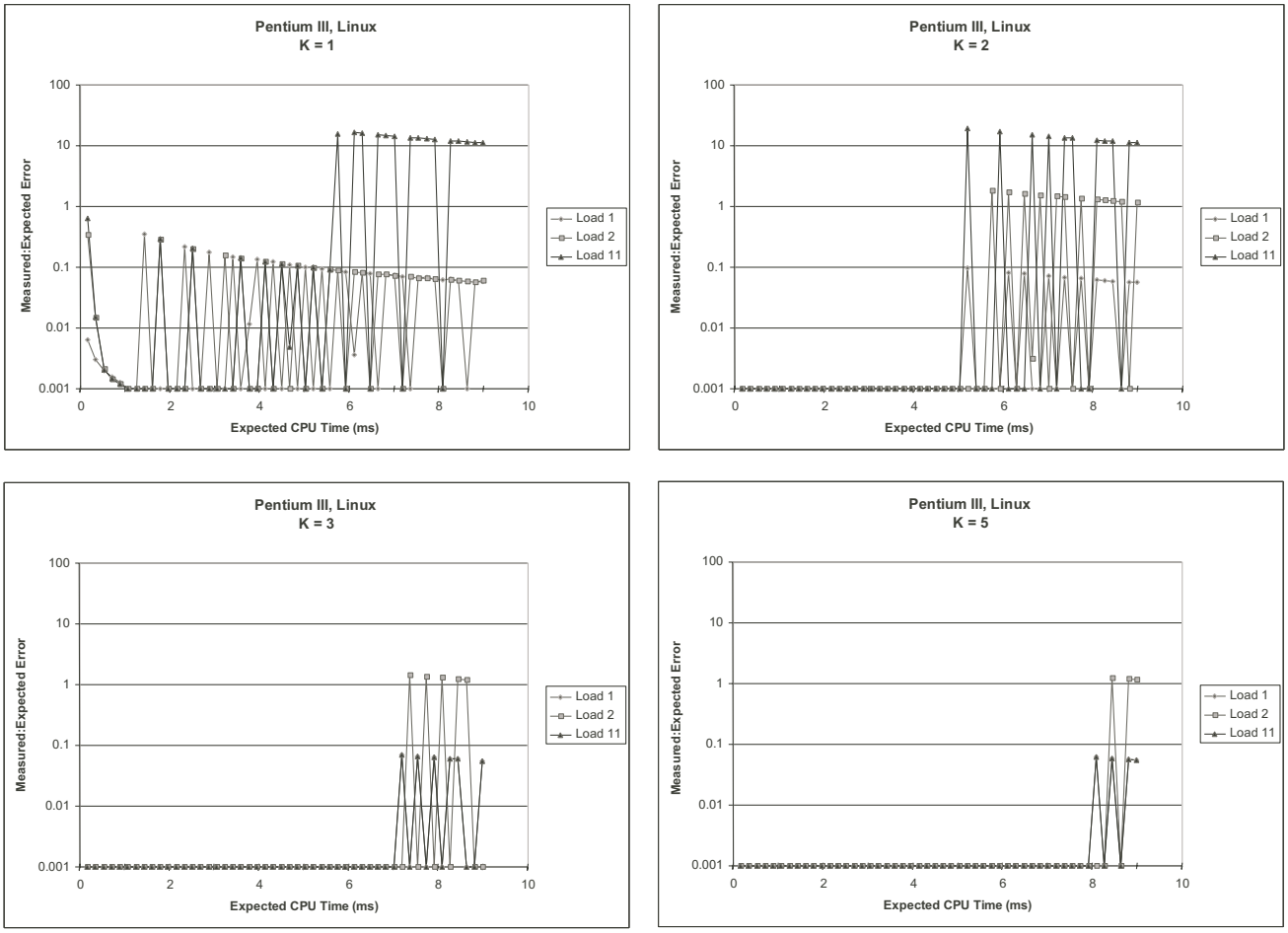


Figure 9.15: **Effectiveness of K-best scheme for different values of K.** K must be at least 2 to have reasonable accuracy. Values greater than 2 help on heavily loaded systems as the program times approach the timer interval.

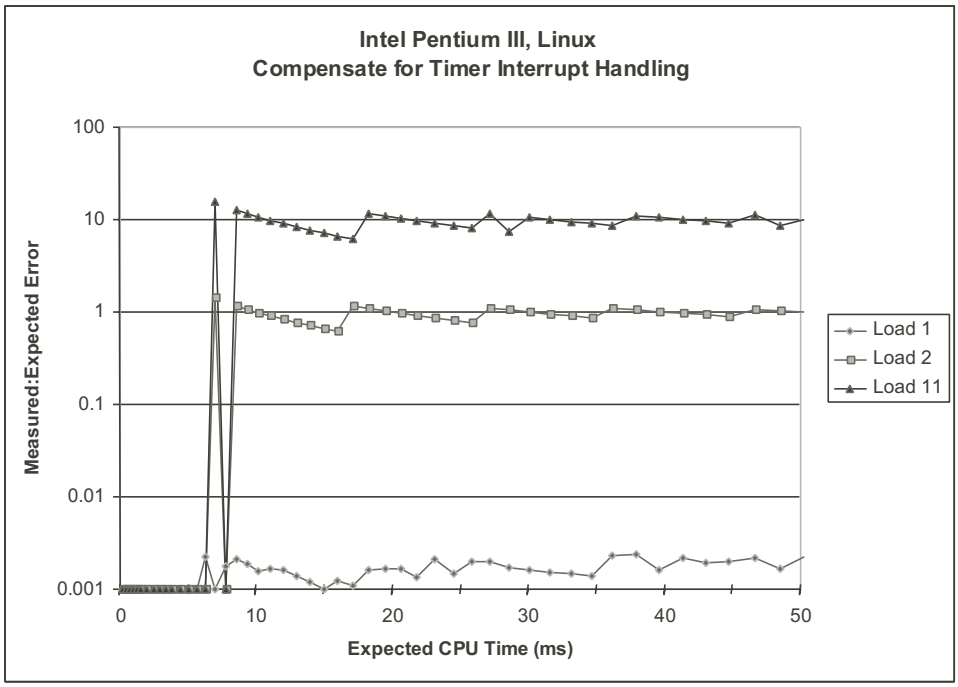


Figure 9.16: **Measurements with Compensation for Timer Interrupt Overhead** This approach greatly improves the accuracy of longer duration measurements on a lightly loaded machine.

1. We must determine how much time is required to handle a single timer interrupt. To preserve the property that we never underestimate the execution time of the procedure, we should determine the minimum number of clock cycles required to service a timer interrupt. That way we will never overcompensate.

2. We must determine how many timer interrupts occur during the period we are measuring.

Using a method similar to that used to generate the traces shown in Figures 9.3 and 9.5, we can detect periods of inactivity and determine their duration. Some of these will be due to timer interrupts, and some will be due to other system events. We can determine whether a timer interrupt has occurred by using the `times` procedure, since the value it returns will be increase one tick each time a timer interrupt occurs. We conducted such an evaluation for 100 periods of inactivity and found that the minimum timer interrupt processing period required 251,466 cycles. To determine the number of timer interrupts that occur during the program we are measuring, we simply call the `times` function twice—once before and once after the program, and then compute their difference.

Figure 9.16 shows the results obtained by this revised measurement scheme. As the figure illustrates, we can now get very accurate (within 1.0%) measurements on a lightly loaded machine, even for programs that execute for multiple time intervals. By removing the systematic error of timer interrupts, we now have a very reliable measurement scheme. On the other hand, we can see that this compensation does not help for programs running on heavily loaded machines.

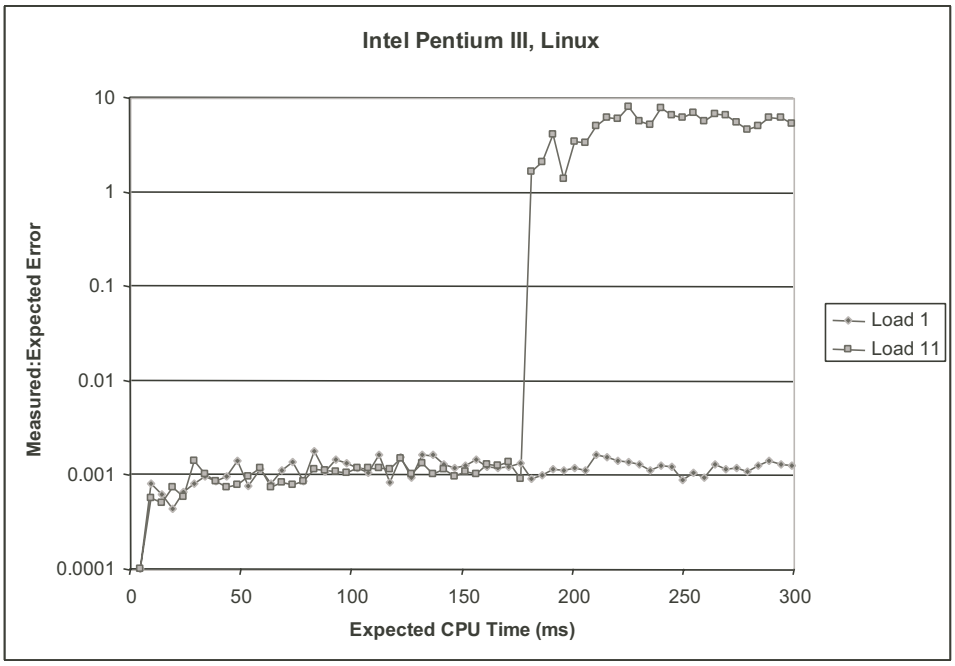


Figure 9.17: **Experimental Validation if K-Best Measurement Scheme on IA32/Linux System with Older Version of the Kernel.** On this system we could get more accurate measurements even for programs with longer execution times, especially on lightly loaded machine.

Evaluation on Other Machines

Since our scheme depends heavily on the scheduling policy of the operating system, we also ran experiments on three other system configurations:

1. Intel Pentium III running older version (2.0.36 vs. 2.2.16) of the Linux kernel.
2. Intel Pentium II running Windows-NT. Although this system uses an IA32 processor, the operating system is fundamentally different from Linux.
3. Compaq Alpha running Tru64 Unix. This uses a very different processor, but the operating system is similar to Linux.

As Figure 9.17 indicates, the performance characteristics under an older version of Linux are very different. On a lightly loaded machine, the measurements are within 0.2% accuracy for programs of almost arbitrary duration. We found that the processor spends only around 3500 cycles processing a timer interrupt with this version of Linux. Even on a heavily loaded machine, it will allow processes to run up to around 180 ms at a time. This experiment shows that the internal details of the operating system can greatly affect system performance and our ability to obtain accurate measurements.

Figure 9.18 shows the results on the Windows-NT system. Overall, the results are similar to those for the older Linux system. For short computations, or on a lightly loaded machine, we could get accurate

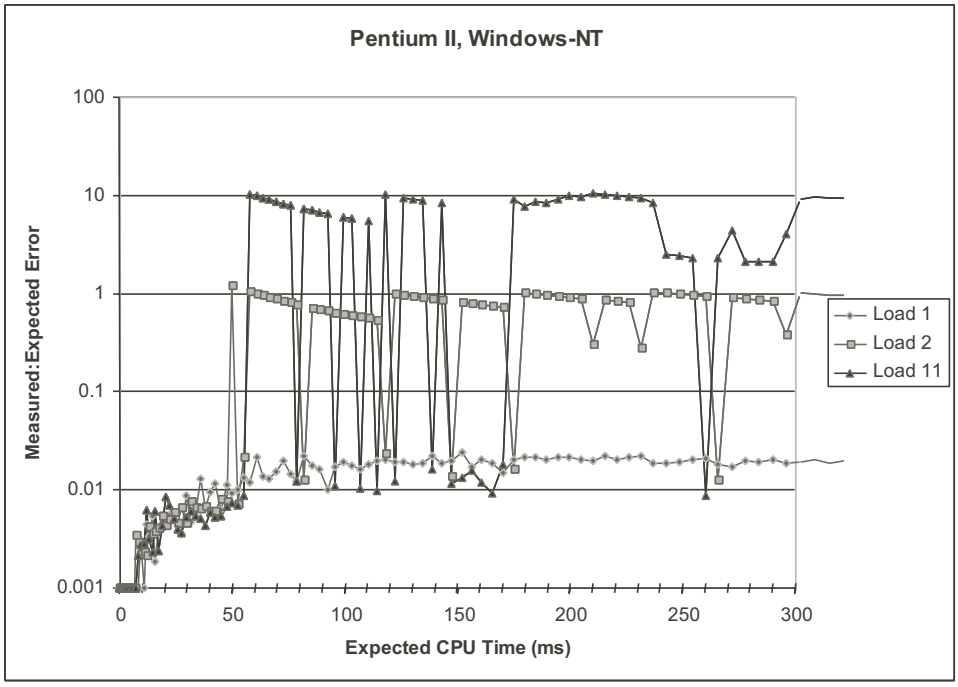


Figure 9.18: **Experimental Validation of K-Best Measurement Scheme on Windows-NT System.** On a lightly loaded system, we can consistently obtain accurate measurements (around 1.0% error). On a heavily loaded system, the accuracy becomes very poor for measurements longer than around 48 ms.

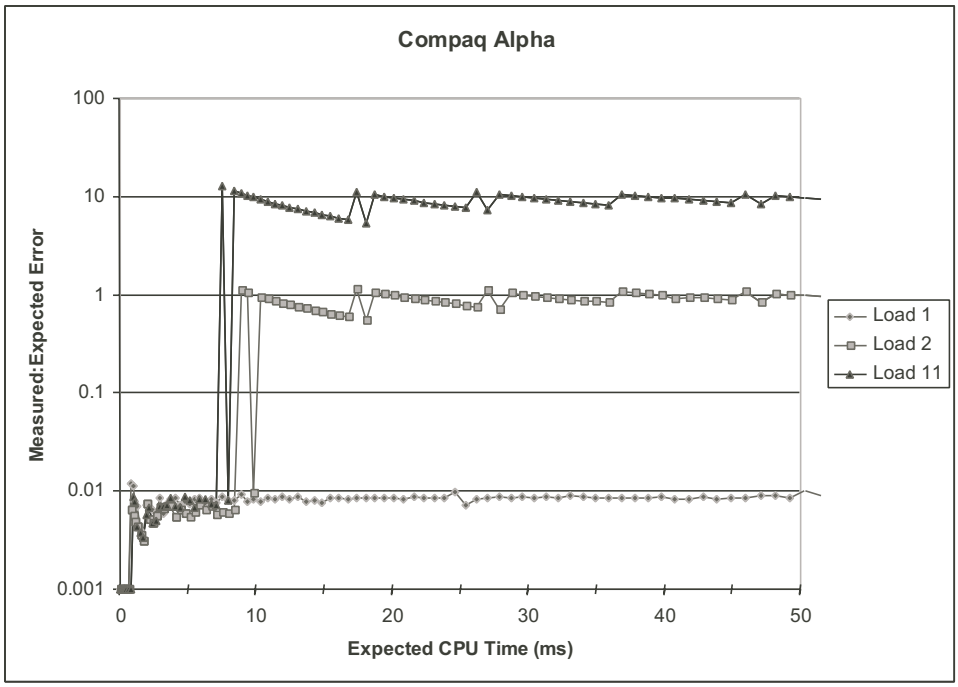


Figure 9.19: **Experimental Validation if K-Best Measurement Scheme on Compaq Alpha System.** For a lightly loaded system, we can consistently obtain accurate ($< 1.0\%$ error) measurements. For a heavily loaded system, durations beyond around 10 ms cannot be measured accurately.

measurements. In this case, our accuracies were around 0.01 (i.e., 1.0%), rather than 0.001. Still, this is good enough for most applications. In addition, our threshold between reliable and unreliable measurements on a heavily loaded machine was around 48 ms. One interesting feature is that we were sometimes able to get accurate measurements on a heavily loaded machine even for computations ranging up to 245 ms. Evidently, the NT scheduler will sometimes allow processes to remain active for longer durations, but we cannot rely on this property.

The Compaq Alpha results are shown in Figure 9.19. Again, we find that on a lightly loaded machine, programs of almost arbitrary duration can be measured with an error of less than 1%. On a heavily loaded machine, only programs with durations less than around 10 ms can be measured accurately.

Practice Problem 9.7:

Suppose we wish to measure a procedure that requires t milliseconds. The machine is heavily loaded and hence will not allow our measurement process to run more than 50 ms at a time.

A. Each trial involves measuring one execution of the procedure. What is the probability this trial will be allowed to run to completion without being swapped out, assuming it starts at some arbitrary point within the 50 ms time segment? Express your answer as a function of t, considering all possible values of t.

B. What is the expected number of trials required so that three of them are reliable measurements of the procedure, i.e., each runs within a single time segment? Express your answer as a function of t. What values do you predict for $t = 20$ and $t = 40$?

Observations

These experiments demonstrate that the K-best measurement scheme works fairly well on a variety of machines. On lightly loaded processors, it consistently gets accurate results on most machines, even for computations with long durations. Only the newer version of Linux incurs a sufficiently high timer interrupt overhead to seriously affect the measurement accuracy. For this system, compensating for this overhead greatly improves the measurement accuracy.

On heavily loaded machines, getting accurate measurements becomes difficult as execution times become longer. Most systems have some maximum execution time beyond which the measurement accuracy becomes very poor. The exact value of this threshold is highly system dependent, but typically ranges between 10 and 200 milliseconds.

9.5 Time-of-Day Measurements

Our use of the IA32 cycle counter provides high-precision timing measurements, but it has the drawback that it only works on IA32 systems. It would be good to have a more portable solution. We have seen that the library functions `times` and `clock` are implemented using interval counters and hence are not very accurate.

Another possibility is to use the library function `gettimeofday`. This function queries the *system clock* to determine the current date and time.

```
#include "time.h"

struct timeval {
   long tv_sec; /* Seconds */
   long tv_usec; /* Microseconds */
}

int gettimeofday(struct timeval *tv, NULL);
```

Returns: 0 for success, -1 for failure

The function writes the time into a structure passed by the caller that includes one field in units of seconds, and another field in units of microseconds. The first field encodes the total number of seconds that have elapsed since January 1, 1970. (This is the standard reference point for all Unix systems.) Note that the second argument to `gettimeofday` should simply be `NULL` on Linux systems, since it refers to an unimplemented feature for performing time zone correction.

Practice Problem 9.8:

On what date will the `tv_sec` field written by `gettimeofday` become negative on a 32-bit machine?

As shown in Figure 9.20, we can can use `gettimeofday` to create a pair of timer functions `start_timer` and `get_timer` that are similar to our cycle-timing functions, except that they measure time in seconds rather than clock cycles.

code/perf/tod.c

```
1 #include <sys/time.h>
2 #include <unistd.h>
3
4 static struct timeval tstart;
5
6 /* Record current time */
7 void start_timer()
8 {
9     gettimeofday(&tstart, NULL);
10 }
11
12 /* Get number of seconds since last call to start_timer */
13 double get_timer()
14 {
15     struct timeval tfinish;
16     long sec, usec;
17
18     gettimeofday(&tfinish, NULL);
19     sec = tfinish.tv_sec - tstart.tv_sec;
20     usec = tfinish.tv_usec - tstart.tv_usec;
21     return sec + 1e-6*usec;
22 }
```

code/perf/tod.c

Figure 9.20: **Timing Procedures Using Unix Time of Day Clock.** This code is very portable, but its accuracy depends on how the clock is implemented.

System	Resolution (μs)	Latency (μs)
Pentium II, Windows-NT	10,000	5.4
Compaq Alpha	977	0.9
Pentium III Linux	1	0.9
Sun UltraSparc	2	1.1

Figure 9.21: **Characteristics of `gettimeofday` Implementations.** Some implementations use interval counting, while others use cycle timers. This greatly affects the measurement precision.

The utility of this timing mechanism depends on how `gettimeofday` is implemented, and this varies from one system to another. Although the fact that the function generates a measurement in units of microseconds looks very promising, it turns out that the measurements are not always that precise. Figure 9.21 shows the result of testing the function on several different systems. We define the *resolution* of the function to be the minimum time value the timer can resolve. We computed this by repeatedly calling `gettimeofday` until the value written to the first argument changed. The resolution is then the number of microseconds by which it changed. As indicated in the table, some implementations can actually resolve times at a microsecond level, while others are much less precise. These variations occur, because some systems use cycle counters to implement the function, while others use interval counting. In the former case, the resolution can be very high—potentially higher than the 1 microsecond resolution provided by the data representation. In the latter case, the resolution will be poor—no better than what is provided by functions `times` and `clock`.

Figure 9.21 also shows the *latency* required by a call to `get_timer` on various systems. This property indicates the minimum time required for a call to the function. We computed this by repeatedly calling the function until one second had elapsed and dividing 1 by the number of calls. As can be seen, this function requires around 1-microsecond on most systems, and several microseconds on others. By comparison, our procedure `get_counter` requires only around 0.2 microseconds per call. In general, system calls involve more overhead than ordinary function calls. This latency also limits the precision of our measurements. Even if the data structure allowed expressing time in units with higher resolution, it is unclear how much more precisely we could measure time when each measurement incurs such a long delay.

Figure 9.22 shows the performance we get from an implementation of the K-best measurement scheme using `gettimeofday` rather than our own functions to access the cycle counter. We show the results on two different machines to illustrate the effect of the time resolution on accuracy. The measurements on a Windows-NT system show characteristics similar to those we found for Linux using `times` (Figure 9.8). Since `gettimeofday` is implemented using the process timers, the error can be negative or positive, and it is especially erratic for short duration measurements. The accuracy improves for longer durations, to the point where the error is less than 2.0% for durations greater than 200 ms. The measurements on a Linux system give results similar to those seen when making direct use of cycle counters. This can be seen by comparing the measurements to the Load 1 results in Figure 9.14 (without compensation) and in Figure 9.16 (with compensation). Using compensation, we can achieve better than 0.04% accuracy, even for measurements as long as 300 ms. Thus, `gettimeofday` performs just as well as directly accessing the cycle counter on this machine.

9.6 Putting it Together: An Experimental Protocol

We can summarize our experimental findings in the form of a protocol to determine how to answer the question "How fast does Program X run on Machine Y?"

- If the anticipated run times of X are long (e.g., greater than 1.0 second), then interval counting should work well enough and be less sensitive to processor load.
- If the anticipated run times of X are in a range of around 0.01 to 1.0 seconds, then it is essential to perform measurements on a lightly loaded system, and to use accurate, cycle-based timing. We should

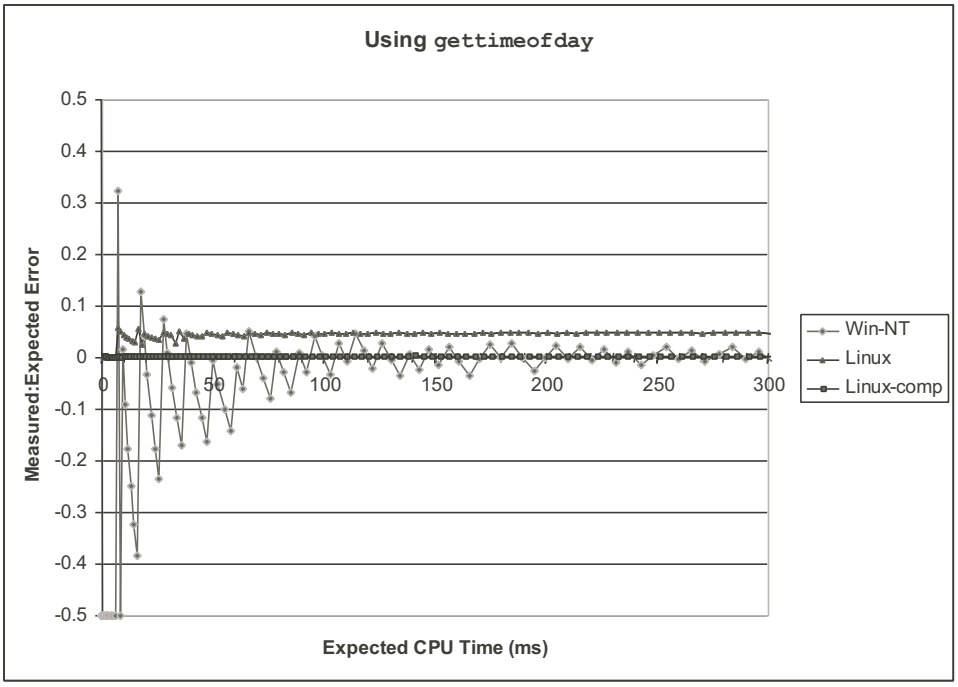


Figure 9.22: **Experimental Validation if K-Best Measurement Scheme Using `gettimeofday` Function.** Linux implements this function using cycle counters and hence achieve the same accuracy as do our own timing routines. Windows-NT implements this function using interval counting, and hence the accuracy is low, especially for small duration measurements.

perform tests of the `gettimeofday` library function to determine whether its implementation on machine Y is cycle based or interval based.

  - If the function is cycle based, then use it as the basis for the K-best timing function.
  - If the function is interval based, then we must find some method of using the machine's cycle counters. This may require assembly language coding.

- If the anticipated run times of X are less than around 0.01 second, then accurate measurements can be performed even on a heavily loaded system, as long as it is uses cycle-based timing. We then proceed in implementing a K-best timing function using either `gettimeofday` or by direct access to the machine's cycle counter.

9.7 Looking into the Future

There are several features that are being incorporated into systems that will have significant impact on performance measurements.

- *Process-specific cycle timing*. It is relatively easy for the operating system to manage the cycle counter so that it indicates the elapsed number of cycles for a specific process. All that is required is to store the count as part of the process' state. Then when the process is reactivated, the cycle counter is set to the value it had when the process was last deactivated, effectively freezing the counter while the process is inactive. Of course, the counter will still be affected by the overhead of kernel operation and by cache effects, but at least the effects of other processes will not be as severe. Already some systems support this feature. In terms of our protocol, this will allow us to use cycle-based timing to get accurate measurements of durations greater than around 0.01 second, even on heavily loaded systems.

- *Variable Rate Clocks*. In an effort to reduce power consumption, future systems will vary the clock rate, since power consumption is directly proportional to the clock rate. In that case, we will not have a simple conversion between clock cycles and nanoseconds. It even becomes difficult to know which unit should be used to express program performance. For a code optimizer, we gain more insight by counting cycles, but for someone implementing an application with real-time performance constraints, actual run times are more important.

9.8 Life in the Real World: An Implementation of the K-Best Measurement Scheme

We have created a library function `fcyc` that uses the K-best scheme to measure the number of clock cycles required by a function `f`.

```
#include "clock.h"
#include "fcyc.h"

typedef void (*test_funct)(int *);

double fcyc(test_funct f, int *params);
```

Returns: number of cycles used by `f` running `params`

The parameter `params` is a pointer to an integer. In general, it can point to an array of integers that form the parameters of the function being measured. For example, when measuring the lower-case conversion functions `lower1` and `lower2`, we pass as a parameter a pointer to a single `int`, which is the length of the string to be converted. When generating the memory mountain (Chapter 6, we pass a pointer to an array of size two containing the size and the stride.

There are a number of parameters that control the measurement, such as the values of K, ϵ, and M, and whether or not to clear the cache before each measurement. These parameters can be set by functions that are also in the library. See the file `fcyc.h` for details.

9.9 Summary

We have seen that computer systems have two fundamentally different methods of recording the passage of time. Timer interrupts occur at a rate that seems very fast when viewed on a macroscopic scale but very slow when viewed on a microscopic scale. By counting intervals, the system can get a very rough measure of program execution time. This method is only useful for long-duration measurements. Cycle counters are very fast, giving good measurements on a microscopic scale. For cycle counters that measure absolute time, the effects of context switching can induce error ranging from small (on a lightly loaded system) to very large (on a heavily loaded system). Thus, no scheme is ideal. It is important to understand the accuracy achievable on a particular system.

Through this effort to devise an accurate timing scheme and to evaluate its performance on a number of different systems, we have learned some important lessons:

- *Every system is different.* Details about the hardware, operating system, and library function implementations can have a significant effect on what kinds of programs can be measured and with what accuracy.
- *Experiments can be quite revealing.* We gained a great deal of insight into the operating system scheduler running simple experiments to generate activity traces. This led to the compensation scheme that greatly improves accuracy on a lightly loaded Linux system. Given the variations from one system to the next, and even from one release of the OS kernel to the next, it is important to be able to analyze and understand the many aspects of a system that affect its performance.
- *Getting accurate timings on heavily loaded systems is especially difficult.* Most systems researchers do all of their measurements on dedicated benchmark systems. They often run the system with many OS and networking features disabled to reduce sources of unpredictable activity. Unfortunately, ordinary programmers to not have this luxury. They must share the system with other users. Even on

heavily loaded systems, our K-best scheme is reasonably robust for measuring durations shorter than the timer interval.

- *The experimental setup must control some sources of performance variations.* Cache effects can greatly affect the execution time for a program. The conventional technique is to make sure that the cache is flushed of any useful data before the timing begins, or else that it is loaded with any data that would typically be in the cache initially.

Through a series of experiments, we were able to design and validate the K-best timing scheme, where we make repeated measurements until the fastest K are within some close range to each other. On some systems, we can make measurements using the library functions for finding the time of day. On other systems, we must access the cycle counters via assembly code.

Bibliographic Notes

There is surprisingly little literature on program timing. Stevens' Unix programming book [77] documents all of the different library functions for program timing. Wadleigh and Crawford's book on software optimization [81] describe code profiling and standard timing functions.

Homework Problems

Homework Problem 9.9 [Category 2]:

Determine the following based on the trace shown in Figure 9.3. Our program estimated the clock rate as 549.9 MHz. It then computed the millisecond timings in the trace by scaling the cycle counts. That is, for a time expressed in cycles as c, the program computed the millisecond timing as $c/549900$. Unfortunately, the program's method of estimating the clock rate is imperfect, and hence some of the millisecond timings are slightly inaccurate.

A. The timer interval for this machine is 10 ms. Which of the time periods above were initiated by a timer interrupt?

B. Based on this trace, what is the minimum number of clock cycles required by the operating system to service a timer interrupt?

C. From the trace data, and assuming the timer interval is exactly 10.0 ms, what can you infer as the value of the true clock rate?

Homework Problem 9.10 [Category 2]:

Write a program that uses library functions `sleep` and `times` to determine the approximate number of clock ticks per second. Try compiling the program and running it on multiple systems. Try to find two different systems that produce results that differ by at least a factor of two.

Homework Problem 9.11 [Category 1]:

We can use the cycle counter to generate activity traces such as was shown in Figures 9.3 and 9.5. Use the functions `start_counter` and `get_counter` to write a function:

```
#include "clock.h"

int inactiveduration(int thresh);
```

Returns: Number of inactive cycles

This function continually checks the cycle counter and detects when two successive readings differ by more than `thresh` cycles, an indication that the process has been inactive. Return the duration (in clock cycles) of that inactive period.

Homework Problem 9.12 [Category 1]:

Suppose we call function `mhz` (Figure 9.10) with parameter `sleeptime` equal to 2. The system has a 10 ms timer interval. Assume that `sleep` is implemented as follows. The processor maintains a counter that is incremented by one every time a timer interrupt occurs. When the system executes `sleep(`x`)`, the system schedules the process to be restarted when the counter reaches $t + 100\,x$, where t is the current value of the counter.

A. Let w denote the time that our process is inactive due to the call to `sleep`. Ignoring the various overheads of function calls, timer interrupts, etc., what range of values can w have?

B. Suppose a call to `mhz` yields 1000.0. Again ignoring the various overheads, what is the possible range of the true clock rate?

Chapter 10

Virtual Memory

Processes in a system share the CPU and main memory with other processes. However, sharing the main memory poses some special challenges. As demand on the CPU increases, processes slow down in some reasonably smooth way. But if too many processes need too much memory, then some of them will simply not be able to run. When a program is out of space, it is out of luck.

Memory is also vulnerable to corruption. If some process inadvertently writes to the memory used by another process, that process might fail in some bewildering fashion totally unrelated to the program logic.

In order to manage memory more efficiently and robustly, modern systems provide an abstraction of main memory known as *virtual memory (VM)*. Virtual memory is an elegant interaction of hardware exceptions, hardware address translation, main memory, disk files, and kernel software that provides each process with a large, uniform, and private address space. With one clean mechanism, virtual memory provides three important capabilities. (1) It uses main memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active areas in main memory, and transferring data back and forth between disk and memory as needed. (2) It simplifies memory management by providing each process with a uniform address space. (3) It protects the address space of each process from corruption by other processes.

Virtual memory is one of the great ideas in computer systems. A big reason for its success is that it works silently and automatically, without any intervention from the application programmer. Since virtual memory works so well behind the scenes, why would a programmer need to understand it? There are several reasons.

- *Virtual memory is central.* Virtual memory pervades all levels of computer systems, playing key roles in the design of hardware exceptions, assemblers, linkers, loaders, shared objects, files, and processes. Understanding virtual memory will help you better understand how systems work in general.
- *Virtual memory is powerful.* Virtual memory gives applications powerful capabilities to create and destroy chunks of memory, map chunks of memory to portions of disk files, and share memory with other processes. For example, did you know that you can read or modify the contents of a disk file by reading and writing memory locations? Or that you can load the contents of a file into memory without doing any explicit copying? Understanding virtual memory will help you harness its powerful capabilities in your applications.
- *Virtual memory is dangerous.* Applications interact with virtual memory every time they reference a

variable, dereference a pointer, or make a call to a dynamic allocation package such as `malloc`. If virtual memory is used improperly, applications can suffer from perplexing and insidious memory-related bugs. For example, a program with a bad pointer can crash immediately with a "Segmentation fault" or a "Protection fault", run silently for hours before crashing, or scariest of all, run to completion with incorrect results. Understanding virtual memory, and the allocation packages such as `malloc` that manage it, can help you avoid these errors.

This chapter looks at virtual memory from two angles. The first half of the chapter describes how virtual memory works. The second half describes how virtual memory is used and managed by applications. There is no avoiding the fact that VM is complicated, and the discussion reflects this in places. The good news is that if you work through the details, you will be able to simulate the virtual memory mechanism of small system by hand, and the virtual memory idea will be forever demystified. The second half builds on this understanding, showing you how to use and manage virtual memory in your programs. You will learn how to manage virtual memory via explicit memory mapping and calls to dynamic storage allocators such as the `malloc` package. You will also learn about a host of common memory-related errors in C programs and how to avoid them.

10.1 Physical and Virtual Addressing

The main memory of a computer system is organized as an array of M contiguous byte-sized cells. Each byte has a unique *physical address (PA)*. The first byte has an address of 0, the next byte an address of 1, the next byte an address of 2, and so on. Given this simple organization, the most natural way for a CPU to access memory would be to use physical addresses. We call this approach *physical addressing*. Figure 10.1 shows an example of physical addressing in the context of a load instruction that reads the word starting at physical address 4.

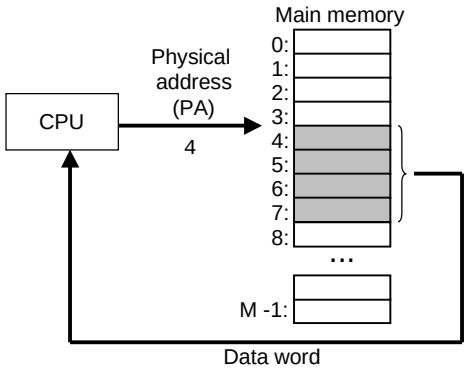


Figure 10.1: **A system that uses physical addressing.**

When the CPU executes the load instruction, it generates an effective physical address and passes it to main memory over the memory bus. The main memory fetches the four-byte word starting at physical address 4 and returns it to the CPU, which stores it in a register.

Early PCs used physical addressing, and systems such as digital signal processors, embedded microcontrollers, and Cray supercomputers continue to do so. However, modern processors designed for general-purpose computing use a form of addressing known as *virtual addressing* (Figure 10.2).

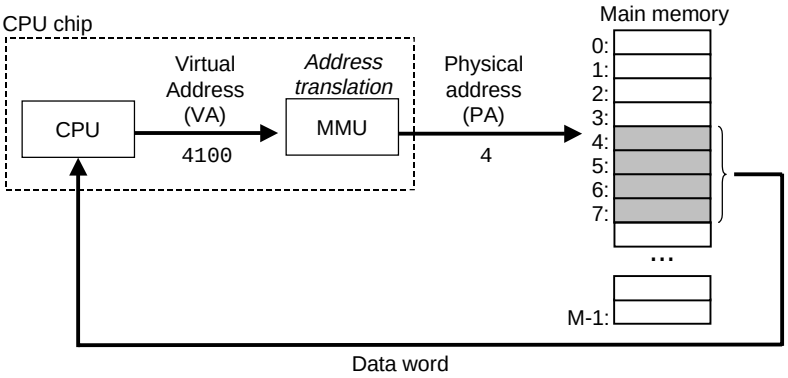


Figure 10.2: **A system that uses virtual addressing.**

With virtual addressing, the CPU accesses main memory by generating a *virtual address (VA)*, which is converted to the appropriate physical address before being sent to the memory. The task of converting a virtual address to a physical one is known as *address translation*. Like exception handling, address translation requires close cooperation between the CPU hardware and the operating system. Dedicated hardware on the CPU chip called the *memory management unit (MMU)* translates virtual addresses on the fly, using a look-up table stored in main memory whose contents are managed by the operating system.

10.2 Address Spaces

An *address space* is an ordered set of nonnegative integer addresses

$$\{0, 1, 2, \ldots\}.$$

If the integers in the address space are consecutive, then we say that it is a *linear address space*. To simplify our discussion, we will always assume linear address spaces. In a system with virtual memory, the CPU generates virtual addresses from an address space of $N = 2^n$ addresses called the *virtual address space*:

$$\{0, 1, 2, \ldots, N-1\}.$$

The size of an address space is characterized by the number of bits that are needed to represent the largest address. For example, a virtual address space with $N = 2^n$ addresses is called an n-bit address space. Modern systems typically support either 32-bit or 64-bit virtual address spaces.

A system also has a *physical address space* that corresponds to the M bytes of physical memory in the system:

$$\{0, 1, 2, \ldots, M-1\}.$$

M is not required to be a power of two, but to simplify the discussion we will assume that $M = 2^m$.

The concept of an address space is important because it makes a clean distinction between data objects (bytes) and their attributes (addresses). Once we recognize this distinction, then we can generalize and allow each data object to have multiple independent addresses, each chosen from a different address space. This is the basic idea of virtual memory. Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.

Practice Problem 10.1:

Complete the following table, filling in the missing entries and replacing each question mark with the appropriate integer. Use the following units: $K = 2^{10}$ (Kilo), $M = 2^{20}$ (Mega), $G = 2^{30}$ (Giga), $T = 2^{40}$ (Tera), $P = 2^{50}$ (Peta), or $E = 2^{60}$ (Exa).

# virtual address bits (n)	# virtual addresses (N)	Largest possible virtual address
8		
	$2^? = 64K$	
		$2^{32} - 1 = \ ?G - 1$
	$2^? = 256T$	
64		

10.3 VM as a Tool for Caching

Conceptually, a virtual memory is organized as an array of N contiguous byte-sized cells stored on disk. Each byte has a unique virtual address that serves as an index into the array. The contents of the array on disk are cached in main memory. As with any other cache in the memory hierarchy, the data on disk (the lower level) is partitioned into blocks that serve as the transfer units between the disk and the main memory (the upper level). VM systems handle this by partitioning the virtual memory into fixed-sized blocks called *virtual pages (VPs)*. Each virtual page is $P = 2^p$ bytes in size. Similarly, physical memory is partitioned into *physical pages (PPs)*, also P bytes in size. (Physical pages are also referred to as *page frames*.)

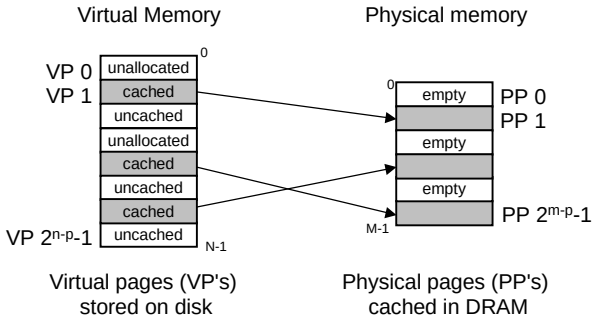


Figure 10.3: **How a VM system uses main memory as a cache.**

At any point in time, the set of virtual pages is partitioned into three disjoint subsets:

- **Unallocated:** Pages that have not yet been allocated (or created) by the VM system. Unallocated blocks do not have any data associated with them, and thus do not occupy any space on disk.

- **Cached:** Allocated pages that are currently cached in physical memory.
- **Uncached:** Allocated pages that are not cached in physical memory.

The example in Figure 10.3 shows a small virtual memory with 8 virtual pages. Virtual pages 0 and 3 have not been allocated yet, and thus do not yet exist on disk. Virtual pages 1, 4, and 6 are cached in physical memory. Pages 2, 3, 5, and 7 are allocated, but are not currently cached in main memory.

10.3.1 DRAM Cache Organization

To help us keep the different caches in the memory hierarchy straight, we will use the term *SRAM cache* to denote the L1 and L2 cache memories between the CPU and main memory, and the term *DRAM cache* to denote the VM system's cache that caches virtual pages in main memory.

The position of the DRAM cache in the memory hierarchy has a big impact on the way that it is organized. Recall that a DRAM is about 10 times slower than an SRAM and that disk is about 100,000 times slower than a DRAM. Thus, misses in DRAM caches are very expensive compared to misses in SRAM caches because DRAM cache misses are served from disk, while SRAM cache misses are usually served from DRAM-based main memory. Further, the cost of reading the first byte from a disk sector is about 100,000 times slower than reading successive bytes in the sector. The bottom line is that the organization of the DRAM cache is driven entirely by the enormous cost of misses.

Because of the large miss penalty and the expense of accessing the first byte, virtual pages tend to be large, typically four to eight KB. Due to the large miss penalty, DRAM caches are fully associative, that is, any virtual page can be placed in any physical page. The replacement policy on misses also assumes greater importance, because the penalty associated with replacing the wrong virtual page is so high. Thus, operating systems use much more sophisticated replacement algorithms for DRAM caches than the hardware does for SRAM caches. (These replacement algorithms are beyond our scope.) Finally, because of the large access time of disk, DRAM caches always use write-back instead of write-through.

10.3.2 Page Tables

As with any cache, the VM system must have some way to determine if a virtual page is cached somewhere in DRAM. If so, the system must determine which physical page it is cached in. If there is a miss, the system must determine where the virtual page is stored on disk, select a victim page in physical memory, and copy the virtual page from disk to DRAM, replacing the victim page.

These capabilities are provided by a combination of operating system software, address translation hardware in the MMU (memory management unit), and a data structure stored in physical memory known as a *page table* that maps virtual pages to physical pages. The address translation hardware reads the page table each time it converts a virtual address to a physical address. The operating system is responsible for maintaining the contents of the page table and transferring pages back and forth between disk and DRAM.

Figure 10.4 shows the basic organization of a page table. A page table is an array of *page table entries (PTEs)*. Each page in the virtual address space has a PTE at a fixed offset in the page table. For our purposes, we will assume that each PTE consists of a *valid bit* and an n-bit address field. The valid bit

indicates whether the virtual page is currently cached in DRAM. If the valid bit is set, the address field indicates the start of the corresponding physical page in DRAM where the virtual page is cached. If the valid bit is not set, then a null address indicates that the virtual page has not yet been allocated. Otherwise, the address points to the start of the virtual page on disk.

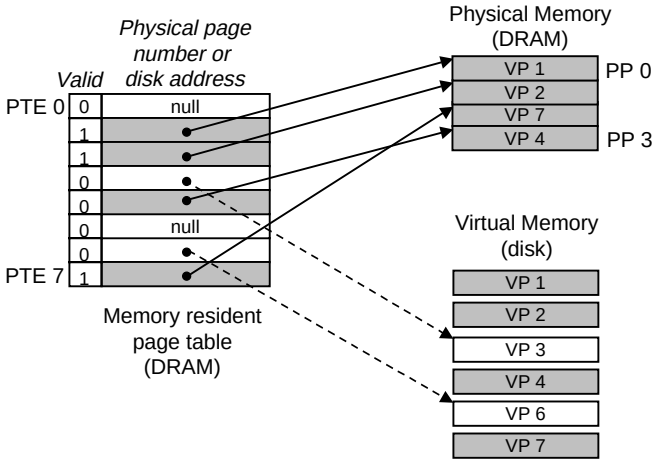


Figure 10.4: **Page table.**

The example in Figure 10.4 shows a page table for a system with 8 virtual pages and 4 physical pages. Four virtual pages (VP 1, VP 2, VP 4, and VP 7) are currently cached in DRAM. Two pages (VP 0 and VP 5) have not yet been allocated, and the rest (VP 3 and VP 6) have been allocated but are not currently cached. An important point to notice about Figure 10.4 is that because the DRAM cache is fully associative, any physical page can contain any virtual page.

Practice Problem 10.2:

Determine the number of page table entries (PTEs) that are needed for the following combinations of virtual address size (n) and page size (P).

n	$P = 2^p$	# PTEs
16	4K	
16	8K	
32	4K	
32	8K	

10.3.3 Page Hits

Consider what happens when the CPU reads a word of virtual memory contained in VP 2, which is cached in DRAM (Figure 10.5). Using a technique we will describe in detail in Section 10.6, the address translation hardware uses the virtual address as an index to locate PTE 2 and read it from memory. Since the valid bit is set, the address translation hardware knows that VP 2 is cached in memory. So it uses the physical memory

address in the PTE (which points to the start of the cached page in PP 0) to construct the physical address of the word.

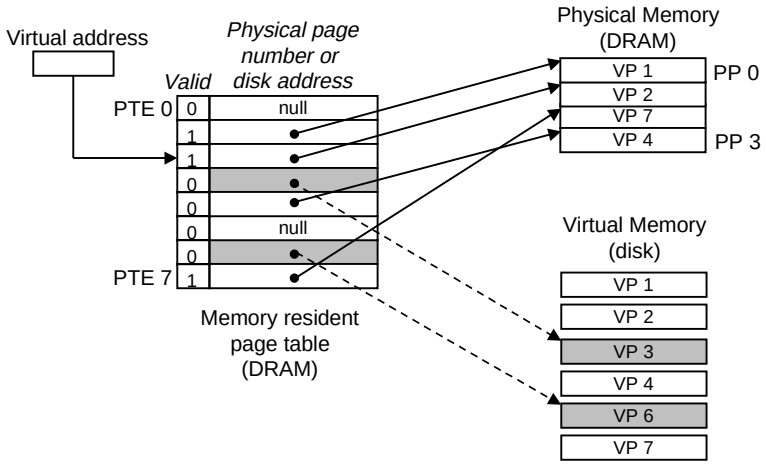


Figure 10.5: **VM page hit.** The reference to a word in VP 2 is a hit.

10.3.4 Page Faults

In virtual memory parlance, a DRAM cache miss is known as a *page fault*. Figure 10.6 shows the state of our example page table before the fault. The CPU has referenced a word in VP 3, which is not cached in DRAM. The address translation hardware reads PTE 3 from memory, infers from the valid bit that VP 3 is not cached, and triggers a page fault exception.

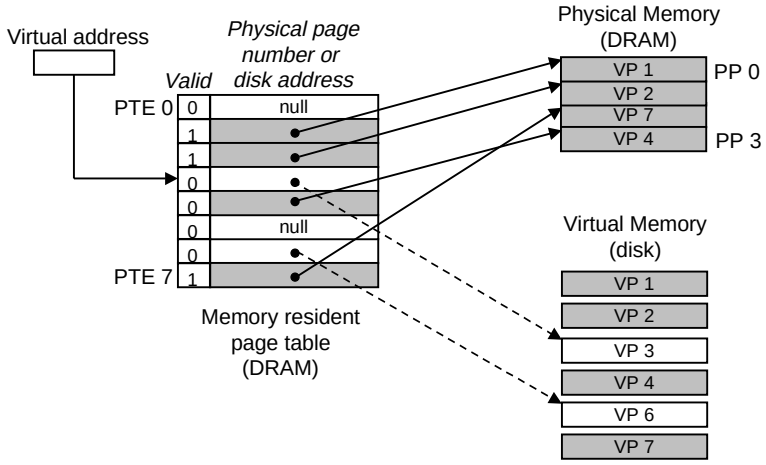


Figure 10.6: **VM page fault (before).** The reference to a word in VP 3 is a miss and triggers a page fault.

The page fault exception invokes a page fault exception handler in the kernel, which selects a victim page,

in this case VP 4 stored in PP 3. If VP 4 has been modified, then the kernel copies it back to disk. In either case, the kernel modifies the page table entry for VP 4 to reflect the fact that VP 4 is no longer cached in main memory.

Next the kernel copies VP 3 from disk to PP 3 in memory, updates PTE 3, and then returns. When the handler returns, it restarts the faulting instruction, which resends the faulting virtual address to the address translation hardware. But now, VP 3 is cached in main memory, and the page hit is handled normally by the address translation hardware, as we saw in Figure 10.5. Figure 10.7 shows the state of our example page table after the page fault.

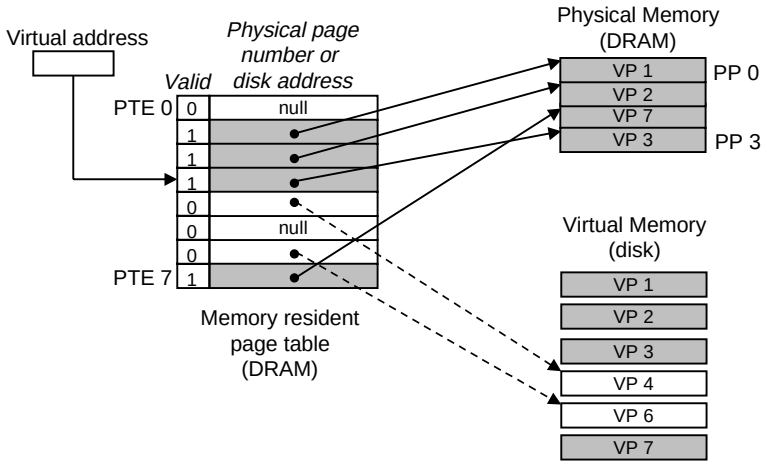


Figure 10.7: **VM page fault (after).** The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.

Virtual memory was invented in the early 1960s, long before the widening CPU-memory gap spawned SRAM caches. As a result, virtual memory systems use a different terminology from SRAM caches, even though many of the ideas are similar. In virtual memory parlance, blocks are known as pages. The activity of transferring a page between disk and memory is known as *swapping* or *paging*. Pages are *swapped in* (*paged in*) from disk to DRAM, and *swapped out* (*paged out*) from DRAM to disk. The strategy of waiting until the last moment to swap in a page, when a miss occurs, is known as *demand paging*. Other approaches, such as trying to predict misses and swap pages in before they are actually referenced, are possible. However, all modern systems use demand paging.

10.3.5 Allocating Pages

Figure 10.8 shows the effect on our example page table when the operating system allocates a new page of virtual memory, for example, as a result of calling `malloc`. In the example, VP 5 is allocated by creating room on disk and updating PTE 5 to point to the newly created page on disk.

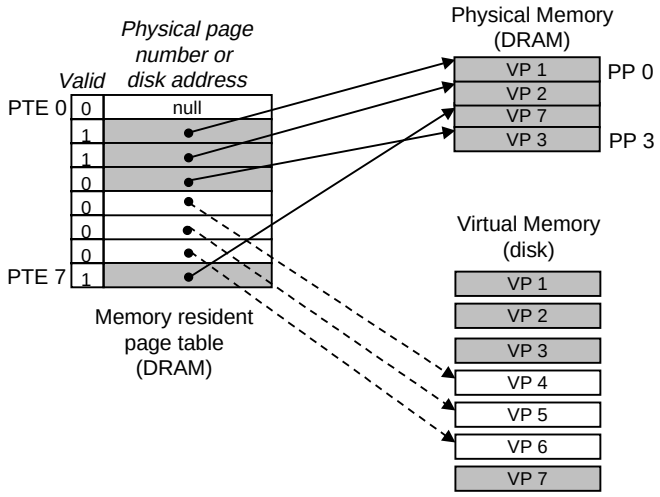


Figure 10.8: **Allocating a new virtual page.** The kernel allocates VP 5 on disk and points PTE 5 to this new location.

10.3.6 Locality to the Rescue Again

When many of us learn about the idea of virtual memory, our first impression is often that it must be terribly inefficient. Given the large miss penalties, we worry that paging will destroy program performance. In practice, virtual memory works pretty well because of our old friend *locality*.

Although the total number of pages that programs reference during an entire run might exceed the total size of physical memory, the principle of locality promises that at any point in time they will tend to work on a smaller set of *active pages* known as the *working set* or *resident set*. After an initial overhead where the working set is paged into memory, subsequent references to the working set result in hits, with no additional disk traffic.

As long as our programs have good temporal locality, virtual memory systems work quite well. But of course, not all programs exhibit good temporal locality. If the working set size exceeds the size of physical memory, then the program can produce an unfortunate situation known as *thrashing*, where pages are swapped in and out continuously. Although virtual memory is usually efficient, if a program's performance slows to a crawl, the wise programmer will consider the possibility that it is thrashing.

Aside: Counting page faults.
You can monitor the number of page faults (and lots of other information) with the Unix `getrusage` function.
End Aside.

10.4 VM as a Tool for Memory Management

In the last section we saw how virtual memory provides a mechanism for using the DRAM to cache pages from a typically larger virtual address space. Interestingly, some early systems such as the DEC PDP-11/70 supported a virtual address space that was *smaller* than the physical memory. Yet virtual memory was still a

useful mechanism because it greatly simplified memory management and provided a natural way to protect memory.

To this point we have assumed a single page table that maps a single virtual address space to the physical address space. In fact, operating systems provide a separate page table, and thus a separate virtual address space, for each process. Figure 10.9 shows the basic idea. In the example, the page table for process i maps VP 1 to PP 2 and VP 2 to PP 7. Similarly, the page table for process j maps VP 1 to PP 7 and VP 2 to PP 10. Notice that multiple virtual pages can be mapped to the same shared physical page.

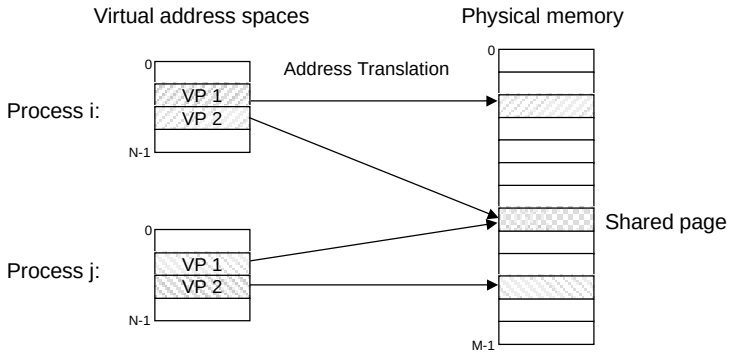


Figure 10.9: **How VM provides processes with separate address spaces.** The operating maintains a separate page table for each process in the system.

The combination of demand paging and separate virtual address spaces has a profound impact on the way that memory is used and managed in a system. In particular, VM simplifies linking and loading, the sharing of code and data, and allocating memory to applications.

10.4.1 Simplifying Linking

A separate address space allows each process to use the same basic format for its memory image, regardless of where the code and data actually reside in physical memory. For example, every Linux process uses the format shown in Figure 10.10.

The text section always starts at virtual address `0x08048000`, the stack always grows down from address `0xbfffffff`, shared library code always starts at address `0x40000000`, and the operating system code and data start always start at address `0xc0000000`. Such uniformity greatly simplifies the design and implementation of linkers, allowing them to produce fully linked executables that are independent of the ultimate location of the code and data in physical memory.

10.4.2 Simplifying Sharing

Separate address spaces provide the operating system with a consistent mechanism for managing sharing between user processes and the operating system itself. In general, each process has its own private code, data, heap, and stack areas that are not shared with any other process. In this case, the operating system creates page tables that map the corresponding virtual pages to disjoint physical pages.

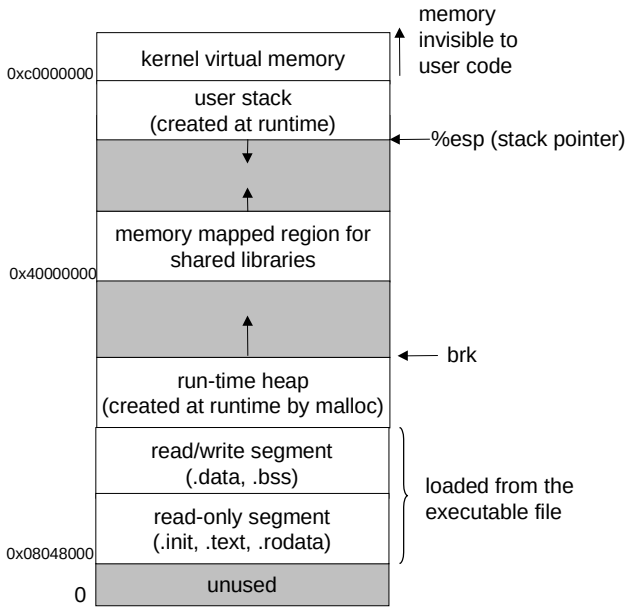


Figure 10.10: **The memory image of a Linux process.** Programs always start at virtual address `0x08048000`. The user stack always starts at virtual address `0xbfffffff`. Shared objects are always loaded in the region beginning at virtual address `0x40000000`.

However, in some instances it is desirable for processes to share code and data. For example, every process must call the same operating system kernel code, and every C program makes calls to routines in the standard C library such as `printf`. Rather than including separate copies of the kernel and standard C library in each process, the operating system can arrange for multiple processes to share a single copy of this code by mapping the appropriate virtual pages in different processes to the same physical pages.

10.4.3 Simplifying Memory Allocation

Virtual memory provides a simple mechanism for allocating additional memory to user processes. When a program running in a user process requests additional heap space (e.g., as a result of calling `malloc`), the operating system allocates an appropriate number, say k, of contiguous virtual memory pages, and maps them to k arbitrary physical pages located anywhere in physical memory. Because of the way page tables work, there is no need for the operating system to locate k contiguous pages of physical memory. The pages can be scattered randomly in physical memory.

10.4.4 Simplifying Loading

Virtual memory also makes it easy to load executable and shared object files into memory. Recall that the `.text` and `.data` sections in ELF executables are contiguous. To load these sections into a newly created process, the Linux loader allocates a contiguous chunk of virtual pages starting at address `0x08048000`,

marks them as invalid (i.e., not cached), and points their page table entries to the appropriate locations in the object file.

The interesting point is that the loader never actually copies any data from disk into memory. The data is paged in automatically and on demand by the virtual memory system the first time each page is referenced, either by the CPU when it fetches an instruction, or by an executing instruction when it references a memory location.

This notion of mapping a set of contiguous virtual pages to an arbitrary location in an arbitrary file is known as *memory mapping*. Unix provides a system call called `mmap` that allows application programs to do their own memory mapping. We will describe application-level memory mapping in more detail in Section 10.8.

10.5 VM as a Tool for Memory Protection

Any robust computer system must provide the means for the operating system to control access to the memory system. A user process should not be allowed to modify its read-only text section. It should not be allowed to read or modify any of the code and data structures in the kernel. It should not be allowed to read or write the private memory of other processes. And it should not be allowed to modify any virtual pages that are shared with other processes unless all parties explicitly allow it (via calls to explicit interprocess communication system calls).

As we have seen, providing separate virtual address spaces makes it easy to isolate the private memories of different processes. But the address translation mechanism can be extended in a natural way to provide even finer access control. Since the address translation hardware reads a PTE each time the CPU generates an address, it is straightforward to control access to the contents of a virtual page by adding some additional permission bits to the PTE. Figure 10.11 shows the general idea.

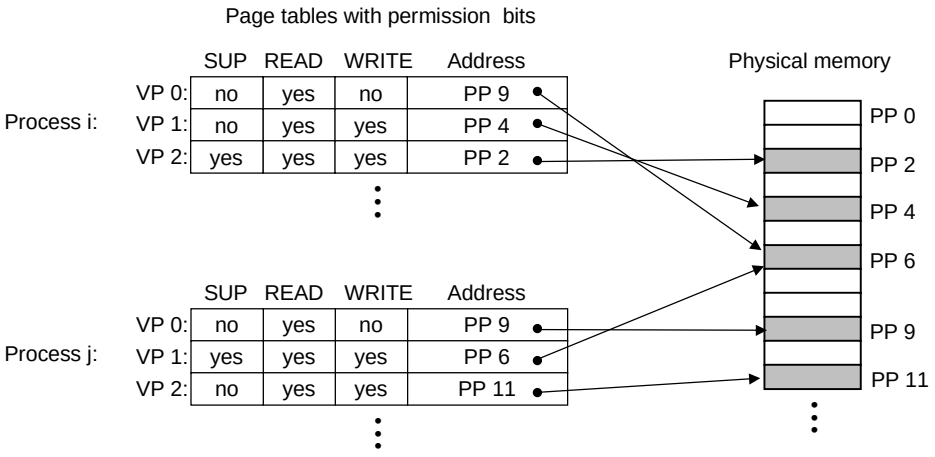


Figure 10.11: **Using VM to provide page-level memory protection.**

In this example, we have added three permission bits to each PTE. The SUP bit indicates whether processes must be running in kernel (supervisor) mode to access the page. Processes running in kernel mode can

Basic parameters	
Symbol	Description
$N = 2^n$	Number of addresses in virtual address space
$M = 2^m$	Number of addresses in physical address space
$P = 2^p$	Page size (bytes)

Components of a virtual address (VA)	
Symbol	Description
VPO	Virtual page offset (bytes)
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag

Components of a physical address (PA)	
Symbol	Description
PPO	Physical page offset (bytes)
PPN	Physical page number
CO	Byte offset within cache block
CI	Cache index
CT	Cache tag

Figure 10.12: **Summary of address translation symbols.**

access any page, but processes running in user mode are only allowed to access pages for which SUP is 0. The READ and WRITE bits control read and write access to the page. For example, if process i is running in user mode, then it has permission to read VP 0 and to read or write VP 1. However, it is not allowed to access VP 2.

If an instruction violates these permissions, then the CPU triggers a general protection fault that transfers control to an exception handler in the kernel. Unix shells typically report this exception as a "segmentation fault."

10.6 Address Translation

This section covers the basics of address translation. Our aim is to give you an appreciation of the hardware's role in supporting virtual memory, with enough detail so that you can work through some concrete examples by hand. However, keep in mind that we are omitting a number of details, especially related to timing, that are important to hardware designers, but are beyond our scope. For your reference, Figure 10.12 summarizes the symbols that we will using throughout this section.

Formally, address translation is a mapping between the elements of an N-element virtual address space (VAS) and an M-element physical address space (PAS),

$$\text{MAP: VAS} \rightarrow \text{PAS} \cup \emptyset$$

where

$$\begin{aligned} \text{MAP(A)} &= A' \text{ if data at virtual addr } A \text{ is present at physical addr } A' \text{ in PAS.} \\ &= \emptyset \text{ if data at virtual addr } A \text{ is not present in physical memory.} \end{aligned}$$

Figure 10.13 shows how the MMU uses the page table to perform this mapping. A control register in the CPU, the *page table base register (PTBR)* points to the current page table. The n-bit virtual address has two components: a p-bit *virtual page offset (VPO)* and an $(n - p)$-bit *virtual page number (VPN)*. The MMU uses the VPN to select the appropriate PTE. For example, VPN 0 selects PTE 0, VPN 1 selects VPN 1, and so on. The corresponding physical address is the concatenation of the *physical page number (PPN)* from the page table entry and the VPO from the virtual address. Notice that since the physical and virtual pages are both P bytes, the *physical page offset (PPO)* is identical to the VPO.

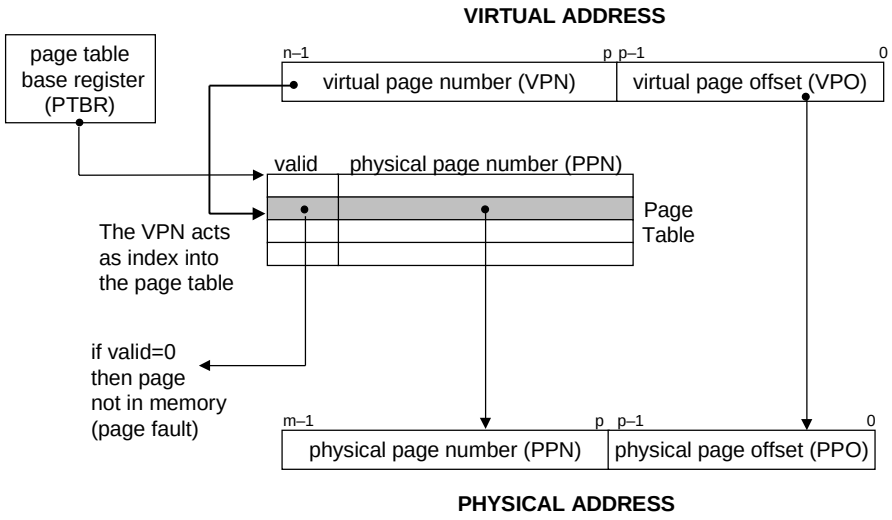


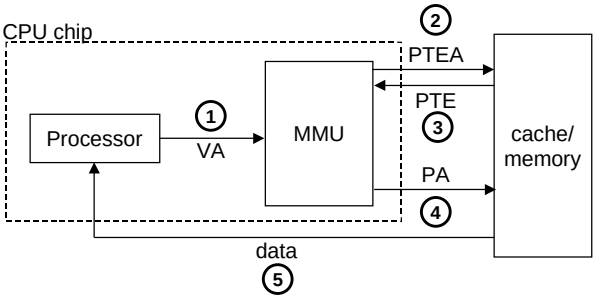
Figure 10.13: **Address translation with a page table.**

Figure 10.14(a) shows the steps that the CPU hardware performs when there is a page hit.

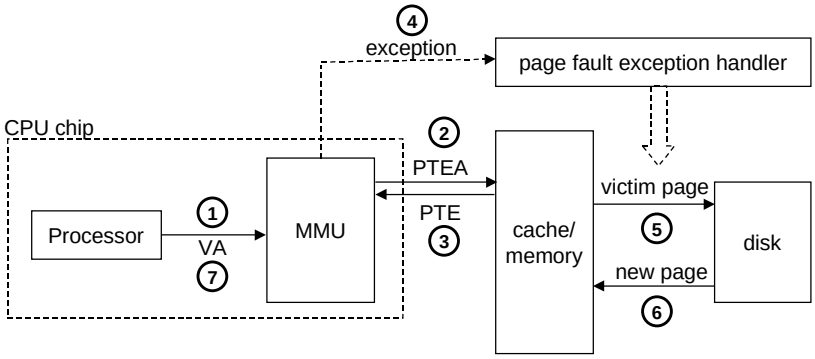
- *Step 1*: The processor generates a virtual address and sends it to the MMU.
- *Step 2*: The MMU generates the PTE address and requests it from the cache/main memory.
- *Step 3*: The cache/main memory returns the PTE to the MMU.
- *Step 3*: The MMU constructs the physical address and sends it to cache/main memory.
- *Step 4*: The cache/main memory returns the requested data word to the processor.

Unlike a page hit, which is handled entirely by hardware, handling a page fault requires cooperation between hardware and the operating system kernel (Figure 10.14(b)).

- *Steps 1 to 3*: The same as Steps 1 to 3 in Figure 10.14(a).



(a) Page hit.



(b) Page fault.

Figure 10.14: **Operational view of page hits and page faults.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

- *Step 4*: The valid bit in the PTE is zero, so the MMU triggers an exception, which transfers control in the CPU to a page fault exception handler in the operating system kernel.
- *Step 5*: The fault handler identifies a victim page in physical memory, and if that page has been modified, pages it out to disk.
- *Step 6*: The fault handler pages in the new page and updates the PTE in memory.
- *Step 7*: The fault handler returns to the original process, causing the faulting instruction to be restarted. The CPU resends the offending virtual address to the MMU. Because the virtual page is now cached in physical memory, there is a hit, and after the MMU performs the steps in Figure 10.14(b), the main memory returns the requested word to the processor

Practice Problem 10.3:

Given a 32-bit virtual address space and a 24-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P:

P	# VPN bits	# VPO bits	# PPN bits	# PPO bits
1 KB				
2 KB				
4 KB				
8 KB				

10.6.1 Integrating Caches and VM

In any system that uses both virtual memory and SRAM caches, there is the issue of whether to use virtual or physical addresses to access the cache. Although a detailed discussion of the tradeoffs is beyond our scope, most systems opt for physical addressing. With physical addressing it is straightforward for multiple processes to have blocks in the cache at the same time and to share blocks from the same virtual pages. Further, the cache does not have to deal with protection issues because access rights are checked as part of the address translation process.

Figure 10.15 shows how a physically-addressed cache might be integrated with virtual memory. The main idea is that the address translation occurs before the cache lookup. Notice that page table entries can be cached, just like any other data words.

10.6.2 Speeding up Address Translation with a TLB

As we have seen, every time the CPU generates a virtual address, the MMU must refer to a PTE in order the translate the virtual address into a physical address. In the worst case, this requires an additional fetch from memory, at a cost of tens to hundreds of cycles. If the PTE happens to be cached in L1, then the cost goes down to one or two cycles. However, many systems try to eliminate even this cost by including a small cache of PTEs in the MMU called a *translation lookaside buffer (TLB)*.

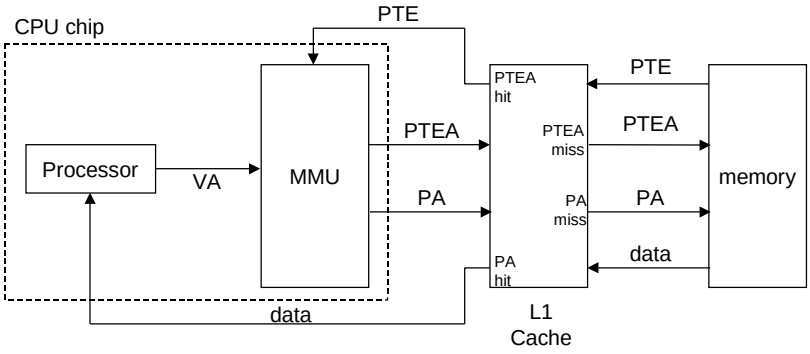


Figure 10.15: **Integrating VM with a physically-addressed cache.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

A TLB is a small, virtually-addressed cache where each line holds a block consisting of a single PTE. A TLB usually has a high degree of associativity. As shown in Figure 10.16, the index and tag fields that are used for set selection and line matching are extracted from the virtual page number in the virtual address. If the TLB has $T = 2^t$ sets, then the *TLB index (TLBI)* consists of the t least significant bits of the VPN, and the *TLB tag (TLBT)* consists of the remaining bits in the VPN.

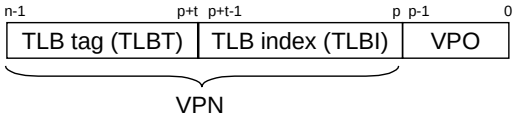


Figure 10.16: **Components of a virtual address that are used to access the TLB.**

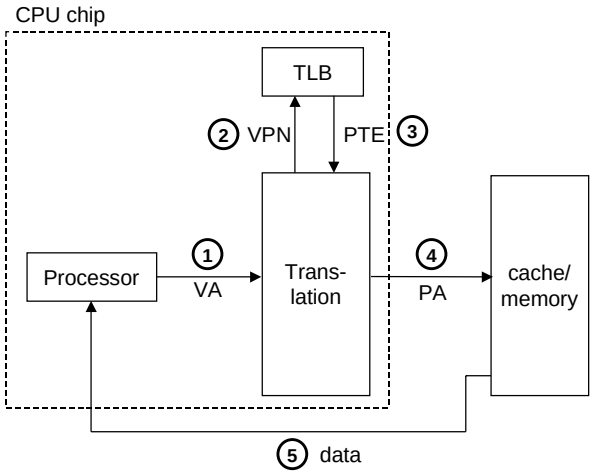
Figure 10.17(a) shows the steps involved when there is a TLB hit (the usual case). The key point here is that all of the address translation steps are performed inside the on-chip MMU, and thus are fast.

- *Step 1*: The CPU generates a virtual address.
- *Steps 2 and 3*: The MMU fetches the appropriate PTE from the TLB.
- *Step 4*: The MMU translates the virtual address to a physical address and sends it to the cache/main memory.
- *Step 5*: The cache/main memory returns the requested data word to the CPU.

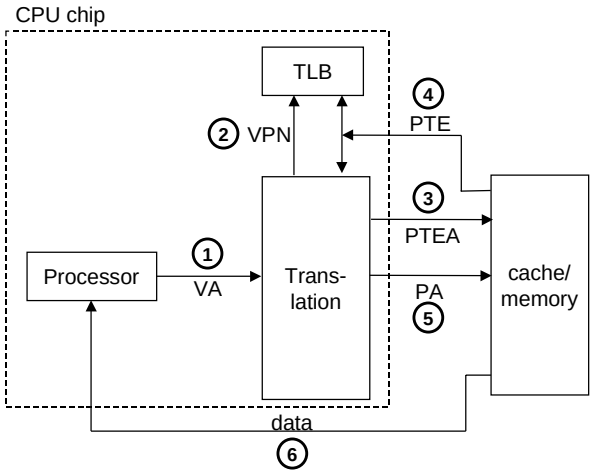
When there is a TLB miss, then the MMU must fetch the PTE from the L1 cache, as shown in Figure 10.17(b). The newly fetched PTE is stored in the TLB, possibly overwriting an existing entry.

10.6.3 Multi-level Page Tables

To this point we have assumed that the system uses a single page table to do address translation. But if we had a 32-bit address space, 4-KB pages, and a 4-byte PTE, then we would need a 4-MB page table resident



(a) TLB hit.



(b) TLB miss.

Figure 10.17: **Operational view of a TLB hit and miss.**

in memory at all times, even if the application referenced only a small chunk of the virtual address space. The problem is compounded for systems with 64-bit addresses spaces.

The common approach for compacting the page table is to a use a hierarchy of page tables instead. The idea is easiest to understand with a concrete example. Suppose the 32-bit virtual address space is partitioned into four-KB pages, and that page table entries are four bytes each. Suppose also that at this point in time the virtual address space has the following form: The first 2K pages of memory are allocated for code and data, the next 6K pages are unallocated, the next 1023 pages are also unallocated, and the next page is allocated for the user stack. Figure 10.18 shows how we might construct a two-level page table hierarchy for this virtual address space.

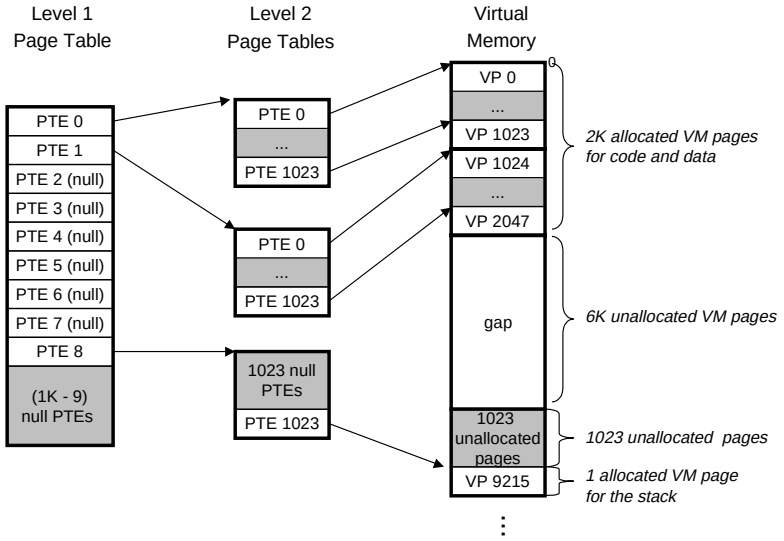


Figure 10.18: **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

Each PTE in the level-1 table is responsible for mapping a four-MB chunk of the virtual address space, where each chunk consists of 1024 contiguous pages. For example, PTE 0 maps the first chunk, PTE 1 the next chunk, and so on. Given that the address space is four GB, 1024 PTEs are sufficient to cover the entire space.

If every page in chunk i is unallocated, then level-1 PTE i is null. For example, in Figure 10.18, chunks 2–7 are unallocated. However, if at least one page in chunk i is allocated, then level-1 PTE i points to the base of a level-2 page table. For example, in Figure 10.18, all or portions of chunks 0, 1, and 8 are allocated, so their level-1 PTEs point to level-2 page tables.

Each PTE in a level-2 page table is responsible for mapping a 4-KB page of virtual memory, just as before when we looked at single-level page tables. Notice that with 4-byte PTEs, each level-1 and level-2 page table is 4K bytes, which conveniently is the same size as a page.

This scheme reduces memory requirements in two ways. First, if a PTE in the level-1 table is null, then the corresponding level-2 page table does not even have to exist. This represents a significant potential savings, since most of the 4-GB virtual address space for a typical program is unallocated. Second, only the level-1

table needs to be in main memory at all times. The level-2 page tables can be created and paged in and out by the VM system as they are needed, which reduces pressure on main memory. Only the most heavily used level-2 page tables need to be cached in main memory.

Figure 10.19 summarizes address translation with a k-level page table hierarchy. The virtual address is partitioned into k VPNs and a VPO. Each VPN i, $1 \leq i \leq k$, is an index into a page table at level i. Each PTE in a level-j table, $1 \leq j \leq k-1$, points to the base of some page table at level $j+1$. Each PTE in a level-k table contains either the PPN of some physical page or the address of a disk block. To construct the physical address, the MMU must access k PTEs before it can determine the PPN. As with a single-level hierarchy, the PPO is identical to the VPO.

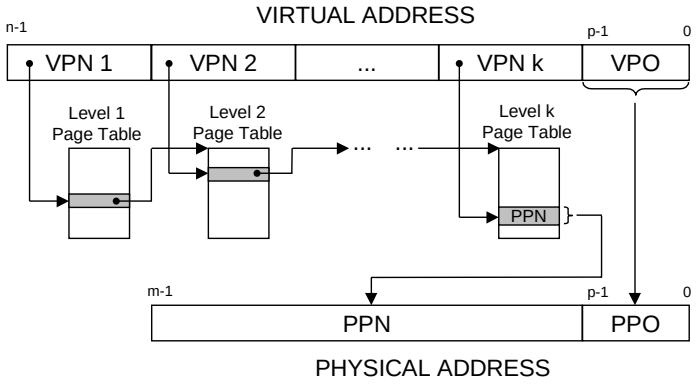


Figure 10.19: **Address translation with a k-level page table.**

Accessing k PTEs may seem expensive and impractical at first glance. However, the TLB comes to the rescue here by caching PTEs from the page tables at the different levels. In practice, address translation with multi-level page tables is not significantly slower than with single-level page tables.

10.6.4 Putting it Together: End-to-end Address Translation

In this section we put it all together with a concrete example of end-to-end address translation on a small system with a TLB and L1 d-cache. To keep things manageable, we make the following assumptions:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 14 bits wide ($n = 14$).
- Physical addresses are 12 bits wide ($m = 12$).
- The page size is 64 bytes ($P = 64$).
- The TLB is four-way set associative with 16 total entries.
- The L1 d-cache is physically-addressed and direct mapped, with a 4-byte line size and 16 total sets.

Figure 10.20 shows the formats of the virtual and physical addresses. Since each page is $2^6 = 64$ bytes, the low-order six bits of the virtual and physical addresses serve as the VPO and PPO respectively. The high-order eight bits of the virtual address serve as the VPN. The high-order six bits of the physical address serve as the PPN.

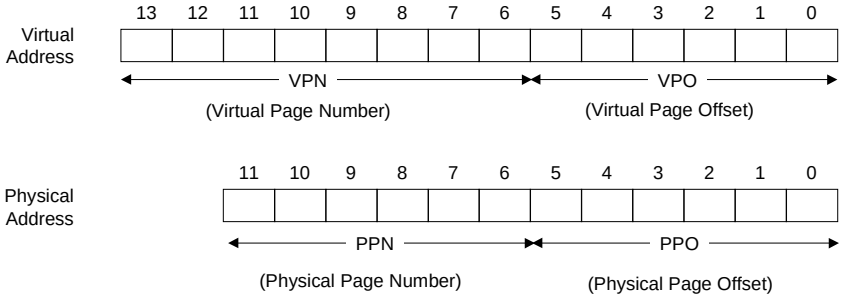
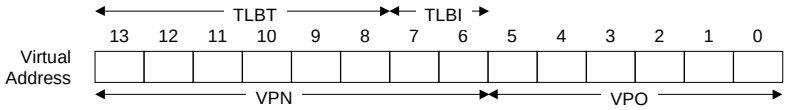


Figure 10.20: **Addressing for small memory system.** Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).

Figure 10.21 shows a snapshot of our little memory system, including the TLB (a), a portion of the page table (b), and the L1 cache (c). Above the figures of the TLB and cache, we have also shown how the bits of the virtual and physical addresses are partitioned by the hardware it accesses these devices.

- *TLB*: The TLB is virtually addressed using the bits of the VPN. Since the TLB has four sets, the two low-order bits of the VPN serve as the set index (TLBI). The remaining six high-order bits serve as the tag (TLBT) that distinguishes the different VPNs that might map to the same TLB set.
- *Page table.* The page table is a single-level design with a total of $2^8 = 256$ page table entries (PTEs). However, we are only interested in the first sixteen of these. For convenience, we have labelled each PTE with the VPN that indexes it; but keep in mind though that these VPNs are not part of the page table and not stored in memory. Also, notice that the PPN of each invalid PTE is denoted with a dash to reinforce the idea that whatever bit values might happen to be stored there are not meaningful.
- *Cache.* The direct-mapped cache is addressed by the fields in the physical address. Since each block is 4 bytes, the low-order 2 bits of the physical address serve as the block offset (CO). Since there are 16 sets, the next 4 bits serve as the set index (CI). The remaining 6 bits serve as the tag (CT).

Given this initial setup, lets see what happens when the CPU executes a load instruction that reads the byte at address `0x03d4`. (Recall that our hypothetical CPU reads one-byte words rather than four-byte words.) To begin this kind of manual simulation, we find it helpful to write down the bits in the virtual address, identify the various fields we will need, and determine their hex values. The hardware perform a similar task when it decodes the address.



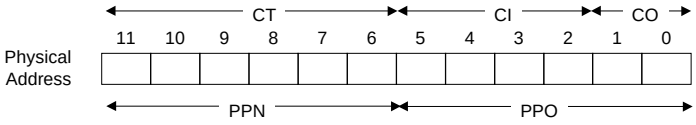
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

(a) TLB: Four sets, sixteen entries, four-way set associative.

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

(b) Page table: Only the first sixteen PTEs are shown.



Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

(c) Cache: 16 sets, four-byte blocks, direct mapped.

Figure 10.21: **TLB, page table, and cache for small memory system.** All values in the TLB, page table, and cache are in hexadecimal notation.

	TLBT						TLBI							
	0x03						0x03							
bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					

To begin, the MMU extracts the VPN (0x0F) from the virtual address and checks with the TLB to see if has cached a copy of PTE 0x0F from some previous memory reference. The TLB extracts the TLB index (0x03) and the TLB tag (0x3) from the VPN, hits on a valid match in the second entry of Set 0x3, and returns the cached PPN (0x0D) to the MMU.

If the TLB had missed, then the MMU would need to fetch the PTE from main memory. However, in this case we got lucky and had a TLB hit. The MMU now has everything it needs to form the physical address. It does this by concatenating the PPN (0x0D) from the PTE with the VPO (0x14) from the virtual address, which forms the physical address (0x354).

Next, the MMU sends the physical address to the cache, which extracts the cache offset CO (0x0), the cache set index CI (0x5), and the cache tag CT (0x0D) from the physical address.

	CT						CI				CO	
	0x0d						0x05				0x0	
bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

Since the tag in Set 0x5 matches CT, the cache detects a hit, reads out the data byte (0x36) at offset CO, and returns it to the MMU, which then passes it back to the CPU.

Other paths through the translation process are also possible. For example, if the TLB misses, then the MMU must fetch the PPN from a PTE in the page table. If the resulting PTE is invalid, then there is a page fault and the kernel must page in the appropriate page and rerun the load instruction. Another possibility is that the PTE is valid, but the necessary memory block misses in the cache.

Practice Problem 10.4:

Show how the example memory system in Section 10.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, physical address, and cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned". If there is a page fault, enter "–" for "PPN" and leave parts C and D blank.

Virtual address: 0x03d7

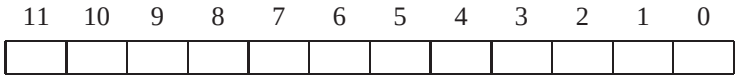
A. Virtual address format

13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit? (Y/N)	
Page fault? (Y/N)	
PPN	

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

10.7 Case Study: The Pentium/Linux Memory System

We conclude our discussion of caches and virtual memory with a case study of a real system: a Pentium-class system running Linux. Figure 10.22 gives the highlights of the Pentium memory system. The Pentium has a 32-bit (4 GB) address space. The *processor package* includes the CPU chip, a unified L2 cache, and a cache bus (backside bus) that connects them. The CPU chip proper contains four different caches: an instruction TLB, data TLB, L1 i-cache, and L1 d-cache. The TLBs are virtually addressed. The L1 and L2 caches are physically addressed. All caches in the Pentium (including the TLBs) are four-way set associative.

The TLBs cache 32-bit page table entries. The instruction TLB caches PTEs for the virtual addresses generated by the instruction fetch unit. The data TLB caches PTEs for the virtual instructions generated by instructions. The instruction TLB has 32 entries. The data TLB has 64 entries. The page size can be configured at start-up time as either 4 KB or 4 MB. Linux running on a Pentium uses 4-KB pages.

The L1 and L2 caches have 32-byte blocks. Each L1 caches is 16 KB in size and has 128 sets, each of which contains four lines. The L2 cache size can vary from a minimum of 128 KB to a maximum of 2 MB. A typical size is 512 KB.

10.7.1 Pentium Address Translation

This section discusses the address translation process on the Pentium. For your reference, Figure 10.23 summarizes the entire process, from the time the CPU generates a virtual address until a data word arrives

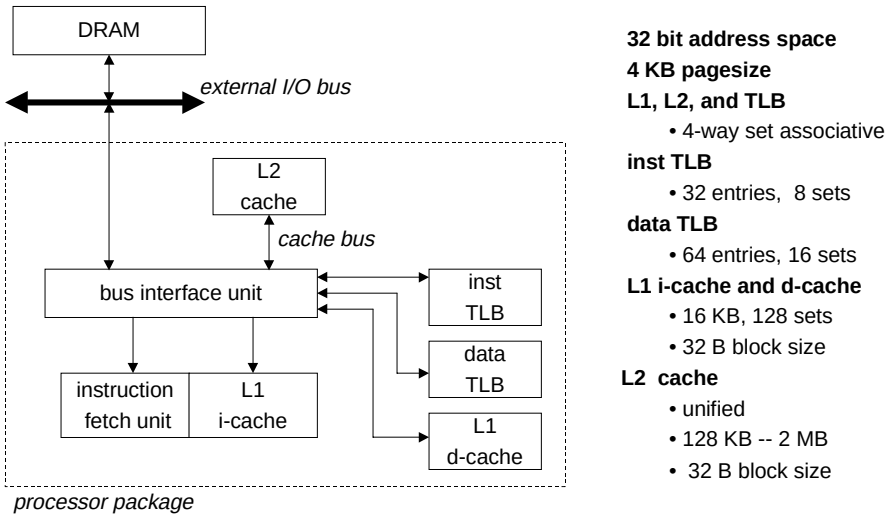


Figure 10.22: **The Pentium memory system.**

from memory.

Aside: Optimizing address translation.
In our discussion of address translation, we have described a sequential two-step process where the MMU (1) translates the virtual address to a physical address, and then (2) passes the physical address to the L1 cache. However, real hardware implementations use a neat trick that allows these steps to be partially overlapped, thus speeding up accesses to the L1 cache.

For example, a virtual address on a Pentium with 4-KB pages has 12 bits of VPO, and these bits are identical to the 12 bits of PPO in the corresponding physical address. Since the four-way set-associative physically-addressed L1 caches have 128 sets and 32-byte cache blocks, each physical address has five ($\log_2 32$) cache offset bits and seven ($\log_2 128$) index bits. These 12 bits fit exactly in the VPO of a virtual address, which is no accident! When the CPU needs a virtual address translated, it sends the VPN to the MMU and the VPO to the L1 cache. While the MMU is requesting a page table entry from the TLB, the L1 cache is busy using the VPO bits to find the appropriate set and read out the four tags and corresponding data words in that set. When the MMU gets the PPN back from the TLB, the cache is ready to try to match the PPN to one of these four tags.

This suggests the following question for you to ponder: What options do Intel engineers have if they want to increase the L1 cache size in future systems and still be able to use this trick? **End Aside.**

Pentium Page Tables

Every Pentium system uses the two-level page table shown in Figure 10.24. The level-1 table, known as the *page directory*, contains 1024 32-bit *page directory entries (PDEs)*, each of which points to one of 1024 level-2 page tables. Each page table contains 1024 32-bit *page table entries (PTEs)*, each of which points to a page in physical memory or on disk.

Each process has a unique page directory and set of page tables. When a Linux process is running, both the page directory and the page tables associated with allocated pages are all memory resident, although the Pentium architecture allows the page tables to be swapped in and out. The *page directory base register (PDBR)* points to the beginning of the page directory.

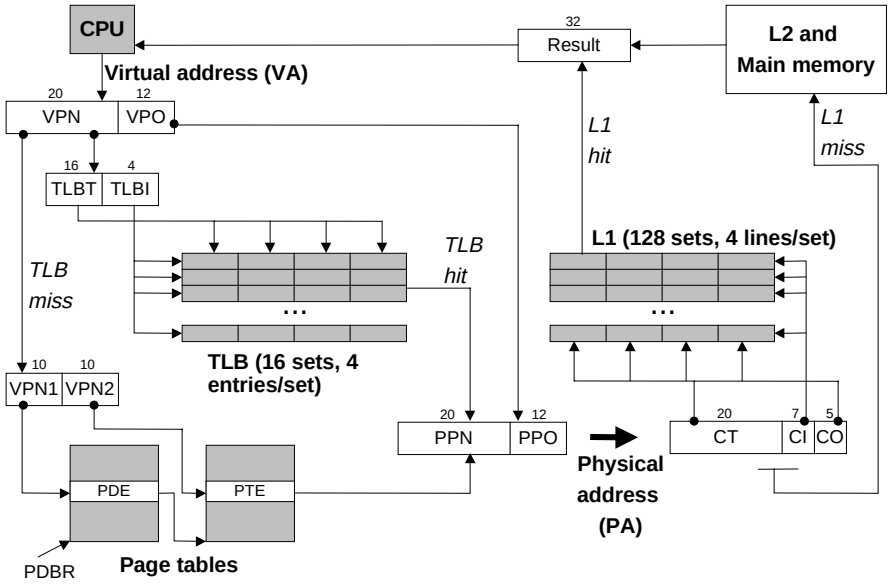


Figure 10.23: **Summary of Pentium address translation.**

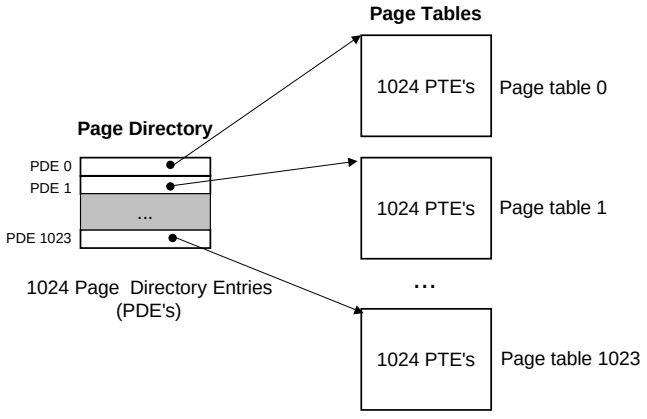
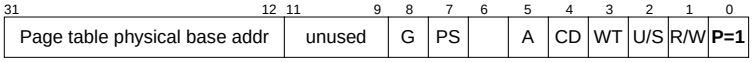


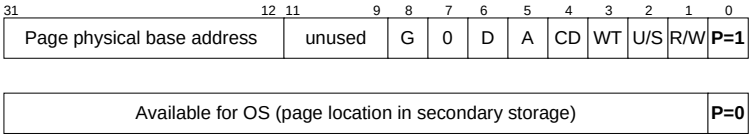
Figure 10.24: **Pentium multi-level page table.**

Figure 10.25(a) shows the format of a PDE. When $P = 1$ (which is always the case with Linux), the address field contains a 20-bit physical page number that points to the beginning of the appropriate page table. Notice that this imposes a 4-KB alignment requirement on page tables. Figure 10.25(b) shows the format



Field	Description
P	page table is present in physical memory (1) or not (0)
R/W	read-only or read-write access permission
U/S	user or supervisor mode (kernel mode) access permission
WT	write-through or write-back cache policy for this page table
CD	cache disabled (1) or enabled (0)
A	has the page been accessed? (set by MMU on reads and writes, cleared by software)
PS	page size 4K (0) or 4M (1)
G	global page (don't evict from TLB on task switch)
PT base addr	20 most significant bits of physical page table address

(a) Page Directory Entry (PDE).



Field	Description
P	page is present in physical memory (1) or not (0)
R/W	read-only or read/write access permission
U/S	user/supervisor mode (kernel mode) access permission
WT	write-through or write-back cache policy for this page
CD	cache disabled or enabled
A	reference bit (set by MMU on reads and writes, cleared by software)
D	dirty bit (set by MMU on writes, cleared by software)
G	global page (don't evict from TLB on task switch)
page base addr	20 most significant bits of physical page address

(b) Page Table Entry (PTE).

Figure 10.25: **Formats of Pentium page directory entry (PDE) and page table entry (PTE).**

of a PTE. When $P = 1$, the address field contains a 20-bit physical page number that points to the base of some page in physical memory. Again, this imposes a 4-KB alignment requirement on physical pages.

The PTE has two permission bits that control access to the page. The R/W bit determines whether the contents of a page are read/write or read/only. The U/S bit, which determines whether the page can be accessed in user mode, protects code and data in the operating system kernel from user programs.

As the MMU translates each virtual address, it also updates two other bits that can be used by the kernel's page fault handler. The MMU sets the A bit, which is known as a *reference bit*, each time a page is accessed. The kernel can use the reference bit to implement its page replacement algorithm. The MMU sets the D bit, or *dirty bit*, each time the page is written to. A page that has been modified is sometimes called a *dirty page*. The dirty bit tells the kernel whether or not it must write-back a victim page before it copies in a replacement page. The kernel can call a special kernel-mode instruction to clear the reference the dirty bits.

Aside: Execute permissions and buffer overflow attacks.
Notice that a Pentium page table entry lacks an execute permission bit to control whether the contents of a page can be executed. Buffer overflow attacks exploit this omission by loading and running code directly on the user stack (Section 3.13). If there were such an execute bit, then the kernel could eliminate the threat of such attacks by restricting execute privileges to the read-only code segment. **End Aside.**

Pentium Page Table Translation

Figure 10.26 shows how the Pentium MMU uses the two-level page table to translate a virtual address to a physical address. The 20-bit VPN is partitioned into two 10-bit chunks. VPN1 indexes a PDE in the page directory pointed at by the PDBR. The address in the PDE points to the base of some page table that is indexed by VPN2. The PPN in the PTE indexed by VPN2 is concatenated with the VPO to form the physical address.

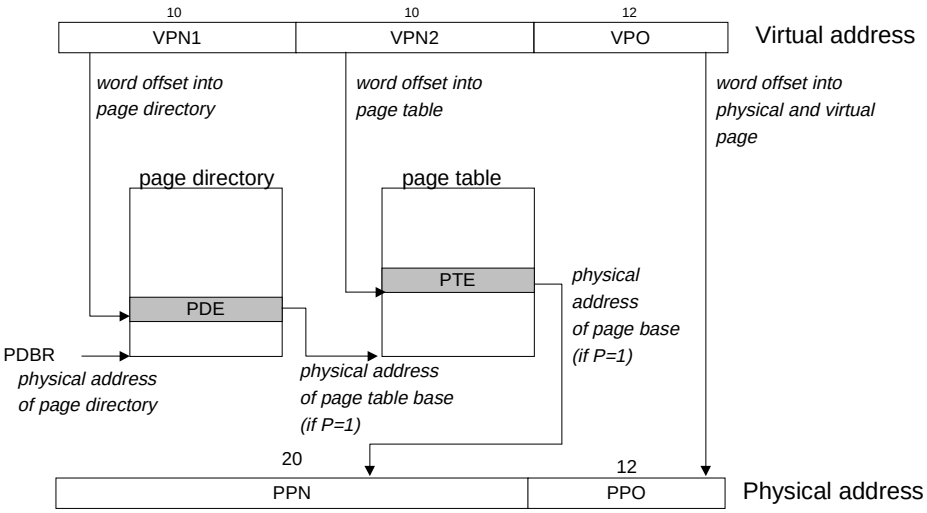


Figure 10.26: **Pentium page table translation.**

Pentium TLB Translation

Figure 10.27 summarizes the process of TLB translation in a Pentium system. If the PTE is cached in the set indexed by the TLBI (a TLB hit), then the PPN is extracted from this cached PTE and concatenated with the VPO to form the physical address. If the PTE is not cached, but the PDE is cached (a partial TLB hit), then

the MMU must fetch the appropriate PTE from memory before it can form the physical address. Finally, if neither the PDE or PTE is cached (a TLB miss), then the MMU must fetch both the PDE and the PTE from memory in order to form the physical address.

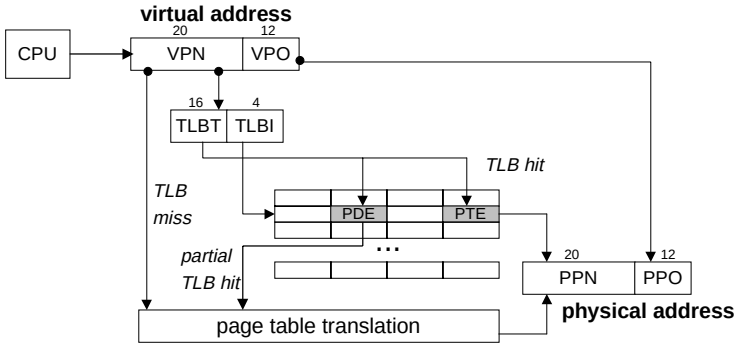


Figure 10.27: **Pentium TLB translation.**

10.7.2 Linux Virtual Memory System

A virtual memory system requires close cooperation between the hardware and the kernel software. While a complete description is beyond our scope, our aim in this section is to describe enough of the Linux virtual memory system to give you a sense of how a real operating system organizes virtual memory and how it handles page faults.

Linux maintains a separate virtual address space for each process of the form shown in Figure 10.28. We have seen this picture a number of times already, with its familiar code, data, heap, shared library, and stack segments. Now that we understand address translation, we can fill in some more details about the kernel virtual memory that lies above address `0xc0000000`.

The kernel virtual memory contains the code and data structures in the kernel. Some regions of the kernel virtual memory are mapped to physical pages that are shared by all processes. For example, each process shares the kernel's code and global data structures. Interestingly, Linux also maps a set of contiguous virtual pages (equal in size to the total amount of DRAM in the system) to the corresponding set of contiguous physical pages. This provides the kernel with a convenient way to access any specific location in physical memory, for example, when it needs to perform memory-mapped I/O operations on devices that are mapped to particular physical memory locations.

Other regions of kernel virtual memory contain data that differs for each process. Examples include page tables, the stack that the kernel uses when it is executing code in the context of the process, and various data structures that keep track of the current organization of the virtual address space.

Linux Virtual Memory Areas

Linux organizes the virtual memory as a collection of *areas* (also called *segments*). An area is a contiguous chunk of existing (allocated) virtual memory whose pages are related in some way. For example, the code

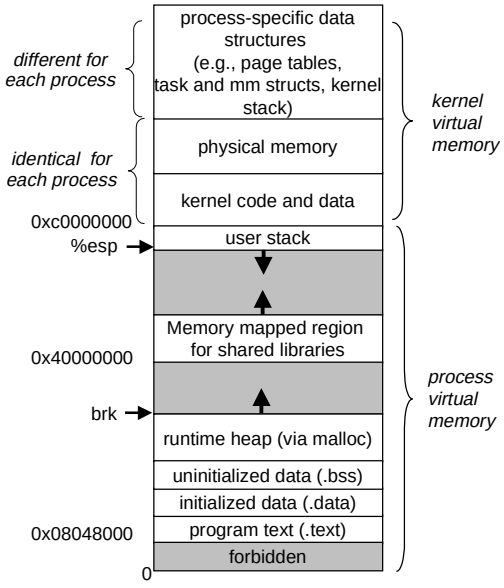


Figure 10.28: **The virtual memory of a Linux process.**

segment, data segment, heap, shared library segment, and user stack are all distinct areas. Each existing virtual page is contained in some area, and any virtual page that is not part of some area does not exist, and cannot be referenced by the process. The notion of an area is important because it allows the virtual address space to have gaps. The kernel does not keep track of virtual pages that do not exist, and such pages do not consume any additional resources in memory, on disk, or in the kernel itself.

Figure 10.29 highlights the kernel data structures that keep track of the virtual memory areas in a process. The kernel maintains a distinct task structure (`task_struct` in the source code) for each process in the system. The elements of the task structure either contain or point to all of the information that the kernel needs to run the process, (e.g., the PID, pointer to the user stack, name of the executable object file, and program counter).

One of the entries in the task structure points to an `mm_struct` that characterizes the current state of the virtual memory. The two fields of interest to us are `pgd`, which points to the base of the page directory table, and `mmap`, which points to a list of `vm_area_structs` (area structs), each of which characterizes an area of the current virtual address space. When the kernel runs this process, it stores `pgd` in the PDBR control register.

For our purposes, the area struct for a particular area contains the following fields:

- `vm_start`: Points to the beginning of the area.
- `vm_end`: Points to the end of the area.
- `vm_prot`: Describes the read/write permissions for all of the pages contained in the area.
- `vm_flags`: Describes (among other things) whether the pages in the area are shared with other processes or private to this process.

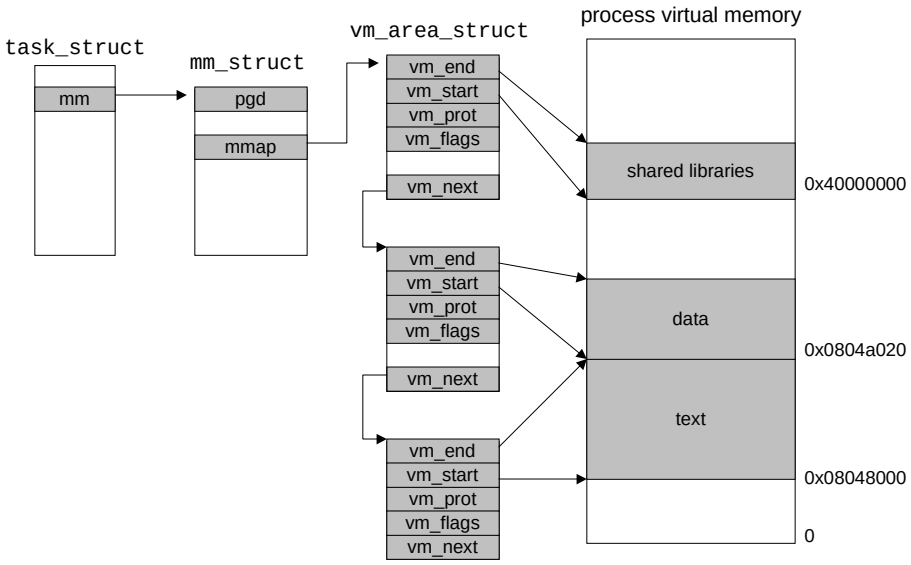


Figure 10.29: **How Linux organizes virtual memory.**

- `vm_next`: Points to the next area struct in the list.

Linux Page Fault Exception Handling

Suppose the MMU triggers a page fault while trying to translate some virtual address A. The exception results in a transfer of control to the kernel's page fault handler, which then performs the following steps:

1. Is virtual address A legal? In other words, does A lie within an area defined by some area struct? To answer this question, the fault handler searches the list of area structs, comparing A with the `vm_start` and `vm_end` in each area struct. If the instruction is not legal, then the fault handler triggers a segmentation fault, which terminates the process. This situation is labeled "1" in Figure 10.30.

   Because a process can create an arbitrary number of new virtual memory areas (using the `mmap` system call described later in Section 10.8), a sequential search of the list of area structs might be very costly. So in practice, Linux superimposes a tree on the list, using some fields that we have not shown, and performs the search on this tree.

2. Is the attempted memory access legal? In other words, does the process have permission to read or write the pages in this area? For example, was the page fault the result of a store instruction trying to write to a read-only page in the code segment? Is the page fault the result of a process running in user mode that is attempting to read a word from kernel virtual memory? If the attempted access is not legal, then the fault handler triggers a protection exception, which terminates the process. This situation is labeled "2" in Figure 10.30.

3. At this point, the kernel knows that the page fault resulted from a legal operation on a legal virtual

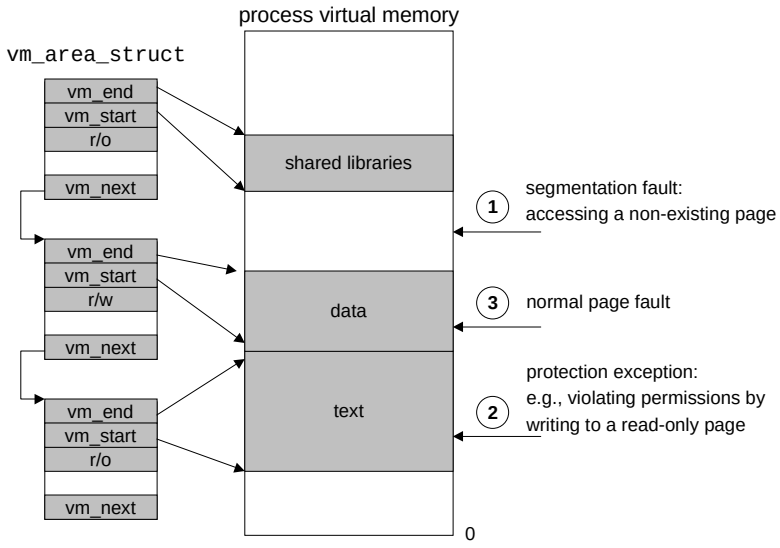


Figure 10.30: **Linux page fault handling.**

address. It handles the fault by selecting a victim page, swapping out the victim page if it is dirty, swapping in the new page, and updating the page table. When the page fault handler returns, the CPU restarts the faulting instruction, which sends A to the MMU again. This time, the MMU translates A normally, without generating a page fault.

10.8 Memory Mapping

Linux (along with other forms of Unix) initializes the contents of a virtual memory area by associating it with an *object* on disk, a process known as *memory mapping*. Areas can be mapped to one of two types of objects:

1. *Regular file in the Unix filesystem:* An area can be mapped to a contiguous section of a regular disk file, such as an executable object file. The file section is divided into page-sized pieces, with each piece containing the initial contents of a virtual page. Because of demand paging, none of these virtual pages is actually swapped into physical memory until the CPU first *touches* the page (i.e., issues a virtual address that falls within that page's region of the address space). If the area is larger than the file section, then the area is padded with zeros.

2. *Anonymous file:* An area can also be mapped to an anonymous file, created by the kernel, that contains all binary zeros. The first time the CPU touches a virtual page in such an area, the kernel finds an appropriate victim page in physical memory, swaps out the victim page if it is dirty, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident. Notice that no data is actually transferred between disk and memory. For this reason, pages in areas that are mapped to anonymous files are sometimes called *demand-zero pages*.

In either case, once a virtual page is initialized, it is swapped back and forth between a special *swap file* maintained by the kernel. The swap file is also known as the *swap space* or the *swap area*. An important point to realize is that at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes.

10.8.1 Shared Objects Revisited

The idea of memory mapping resulted from a clever insight that if the virtual memory system could be integrated into the conventional file system, then it could provide a simple and efficient way to load programs and data into memory.

As we have seen, the process abstraction promises to provide each process with its own private virtual address space that is protected from errant writes or reads by other processes. However, many processes have identical read-only text areas. For example, each process that runs the Unix shell program `tcsh` has the same text area. Further, many programs need to access identical copies of read-only run-time library code. For example, every C program requires functions from the standard C library such as `printf`. It would be extremely wasteful for each process to keep duplicate copies of these commonly used codes in physical memory. Fortunately, memory mapping provides us with a clean mechanism for controlling how objects are shared by multiple processes.

An object can be mapped into an area of virtual memory as either a *shared object* or a *private object*. If a process maps a shared object into an area of its virtual address space, then any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory. Further, the changes are also reflected in the original object on disk.

Changes made to an area mapped to a private object, on the other hand, are not visible to other processes, and any writes that the process makes to the area are *not* reflected back to the object on disk. A virtual memory area that a shared object is mapped into is often called a *shared area*. Similarly for a *private area*.

Suppose that process 1 maps a shared object into an area of its virtual memory, as shown in Figure 10.31(a). Now suppose that process 2 maps the same shared object into its address space (not necessarily at the same virtual address as process 1) as shown in Figure 10.31(b).

Since each object has a unique file name, the kernel can quickly determine that process 1 has already mapped this object and can point the page table entries in process 2 to the appropriate physical pages. The key point is that only a single copy of the shared object needs to be stored in physical memory, even though the object is mapped into multiple shared areas. For convenience, we have shown the physical pages as being contiguous, but of course this is not true in general.

Private objects are mapped into virtual memory using a clever technique known as *copy-on-write*. A private object begins life in exactly the same way as a shared object, with only one copy of the private object stored in physical memory. For example, Figure 10.32(a) shows a case where two processes have mapped a private object into different areas of their virtual memories, but share the same physical copy of the object. For each process that maps the private object, the page table entries for the corresponding private area are flagged as read-only, and the area struct is flagged as *private copy-on-write*. So long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory. However, as soon as a process attempts to write to some page in the private area, the write triggers

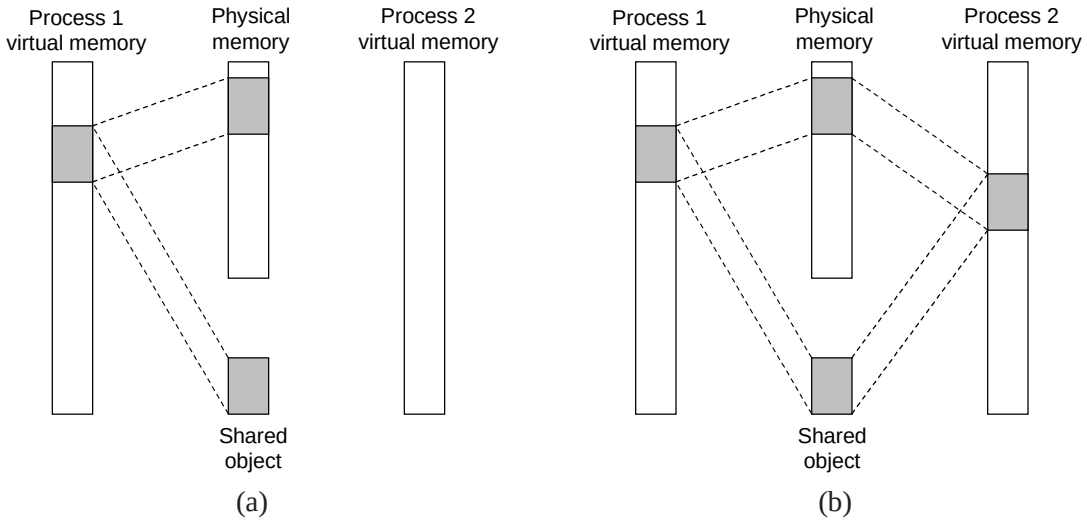


Figure 10.31: **A shared object.** (a) After process 1 maps the shared object. (b) After process 2 maps the same shared object. (Note that the physical pages are not necessarily contiguous.)

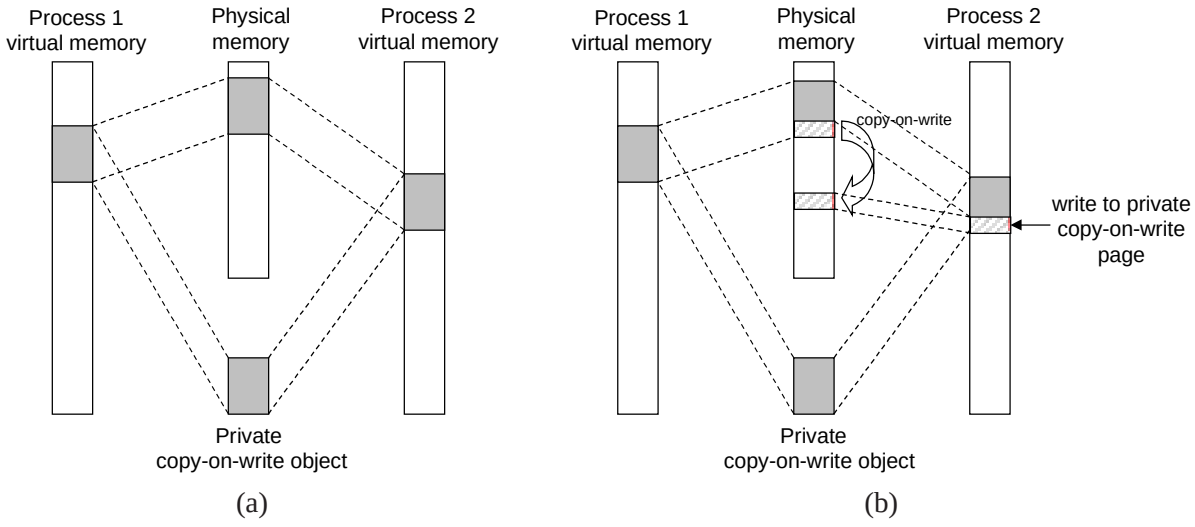


Figure 10.32: **A private copy-on-write object.** (a) After both processes have mapped the private copy-on-write object. (b) After process 2 writes to a page in the private area.

a protection fault.

When the fault handler notices that the protection exception was caused by the process trying to write to a page in a private copy-on-write area, it creates a new copy of the page in physical memory, updates the page table entry to point to the new copy, and then restores write permissions to the page, as shown in Figure 10.32(b). When the fault handler returns, the CPU reexecutes the write, which now proceeds normally on the newly created page.

By deferring the copying of the pages in private objects until the last possible moment, copy-on-write makes the most efficient use of scarce physical memory.

10.8.2 The `fork` Function Revisited

Now that we understand virtual memory and memory mapping, we can get a clear idea of how the `fork` function creates a new process with its own independent virtual address space.

When the `fork` function is called by the *current process*, the kernel creates various data structures for the *new process* and assigns it a unique PID. To create the virtual memory for the new process, it creates exact copies of the current process's `mm_struct`, area structs, and page tables. It flags each page in both processes as read-only, and flags each area struct in both processes as private copy-on-write.

When the `fork` returns in the new process, the new process now has an exact copy of the virtual memory as it existed when the fork was called. When either of the processes performs any subsequent writes, the copy-on-write mechanism creates new pages, thus preserving the abstraction of a private address space for each process.

10.8.3 The `execve` Function Revisited

Virtual memory and memory mapping also play key roles in the process of loading programs into memory. Now that we understand these concepts, we can understand how the `execve` function really loads and executes programs. Suppose that the program running in the current process makes the following call to `execve`:

```
Execve("a.out", NULL, NULL);
```

The `excecve` function loads and runs the program contained in the executable object file `a.out` within the current process, effectively replacing the current program with the `a.out` program. Loading and running `a.out` requires the following steps:

- *Delete existing user areas.* Delete the existing area structs in the user portion of the current process's virtual address.
- *Map private areas.* Create new area structs for the text, data, bss, and stack areas of the new program. All of these new areas are private copy-on-write The text and data areas are mapped to the text and data sections of the `a.out` file. The bss area is demand-zero, mapped to an anonymous file whose size is contained in `a.out`. The stack and heap area are also demand-zero, initially of zero-length. Figure 10.33 summarizes the different mappings of the private areas.

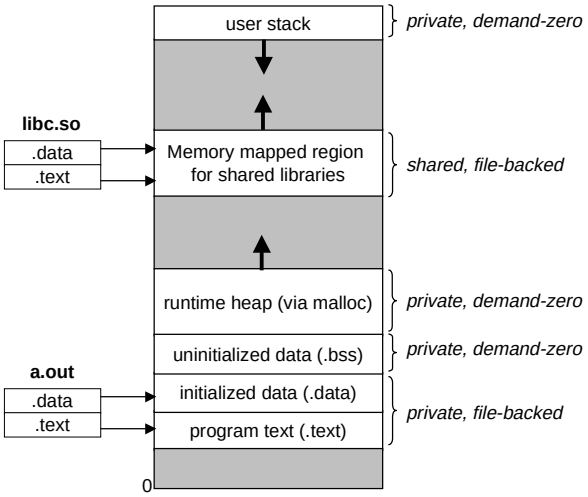


Figure 10.33: **How the loader maps the areas of the user address space.**

- *Map shared areas.* If the `a.out` program was linked with shared objects, such as the standard C library `libc.so`, then these objects are dynamically linked into the program, and then mapped into the shared region of the user's virtual address space.
- *Set the program counter (PC).* The last thing that `execve` does is to set the program counter in the current process's context to point to the entry point in the text area.

The next time this process is scheduled, it will begin execution from the entry point. Linux will swap in code and data pages as needed.

10.8.4 User-level Memory Mapping with the `mmap` Function

Unix processes can use the `mmap` function to create new areas of virtual memory and to map objects into these areas.

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t
offset);
```

returns: pointer to mapped area if OK, -1 on error

The `mmap` function asks the kernel to create a new virtual memory area, preferably one that starts at address `start`, and to map a contiguous chunk of the object specified by file descriptor `fd` to the new area. The contiguous object chunk has a size of `length` bytes and starts at an offset of `offset` bytes from the beginning of the file. The `start` address is merely a hint, and is usually specified as NULL. For our purposes, we will always assume a NULL start address. Figure 10.34 depicts the meaning of these arguments.

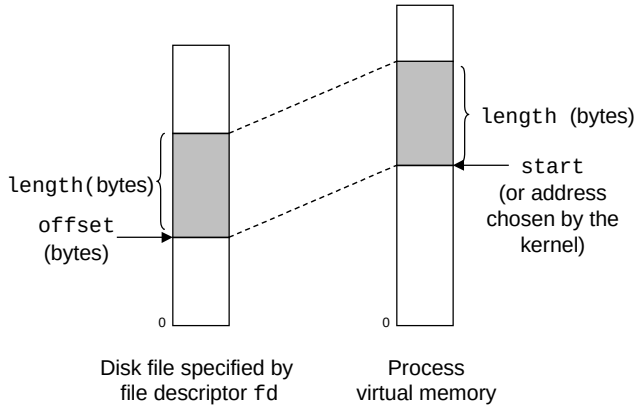


Figure 10.34: **Visual interpretation of `mmap` arguments.**

The `prot` argument contains bits that describe the access permissions of the newly mapped virtual memory area (i.e., the `vm_prot` bits in the corresponding area struct).

- PROT_EXEC: Pages in the area consist of instructions that may be executed by the CPU.
- PROT_READ: Pages in the area may be read.
- PROT_WRITE: Pages in the area may be written.
- PROT_NONE: Pages in the area cannot be accessed.

The `flags` argument consists of bits that describe the type of the mapped object. If the MAP_ANON flag bit is set and `fd` is NULL, then the backing store is an anonymous object and the corresponding virtual pages are demand-zero. MAP_PRIVATE indicates a private copy-on-write object, and MAP_SHARED indicates a shared object. For example,

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

asks the kernel to create a new read-only, private, demand-zero area of virtual memory containing `size` bytes. If the call is successful, then `bufp` contains the address of the new area.

The `munmap` function deletes regions of virtual memory.

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

returns: 0 if OK, -1 on error

The `munmap` function deletes the area starting at virtual address `start` and consisting of the next `length` bytes. Subsequent references to the deleted region result in segmentation faults.

Practice Problem 10.5:

Write a C program `mmapcopy.c` that uses `mmap` to copy an arbitrary-sized disk file to stdout. The name of the input file should be passed as a command line argument.

10.9 Dynamic Memory Allocation

While it is certainly possible to use the low-level `mmap` and `munmap` functions to create and delete areas of virtual memory, most C programs use a *dynamic memory allocator* when they need to acquire additional virtual memory at run time.

A dynamic memory allocator maintains an area of a process's virtual memory known as the *heap* (Figure 10.35). In most Unix systems, the heap is an area of demand-zero memory that begins immediately after the uninitialized `bss` area and grows upward (towards higher addresses). For each process, the kernel maintains a variable `brk` (pronounced "break") that points to the top of the heap.

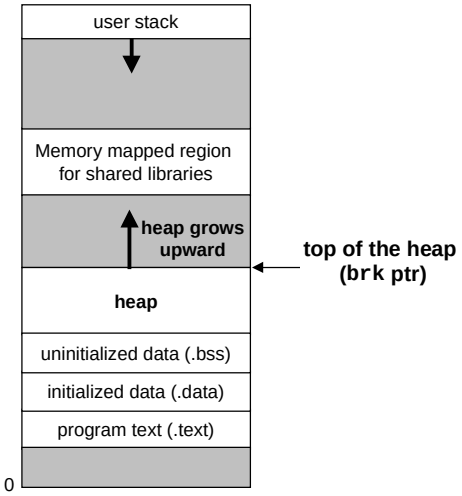


Figure 10.35: **The heap.**

An allocator maintains the heap as a collection of various sized *blocks*. Each block is a contiguous chunk of virtual memory that is either *allocated* or *free*. An allocated block has been explicitly reserved for use by the application. A free block is available to be allocated. A free block remains free until it is explicitly allocated by the application. An allocated block remains allocated until it is freed, either explicitly by the application, or implicitly by the memory allocator itself.

Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks.

Explicit allocators require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the `malloc` package. C programs allocate a block by calling the `malloc` function and free a block by calling the `free` function. The `new` and `free` calls in `C++` are comparable.

Implicit allocators, on the other hand, require the allocator to detect when an allocated block is no longer being used by the program and then free the block. Implicit allocators are also known as *garbage collectors*, and the process of automatically freeing unused allocated blocks is known as *garbage collection*. For example, higher-level languages such as Lisp, ML and Java rely on garbage collection to free allocated blocks.

The remainder of this section discusses the design and implementation of explicit allocators. We will discuss implicit allocators in Section 10.10. For concreteness, our discussion focuses on allocators that manage heap memory. However, students should be aware that memory allocation is a general idea that arises in a variety of contexts. For example, applications that do intensive manipulation of graphs will often use the standard allocator to acquire a large block of virtual memory, and then use an application-specific allocator to manage the memory within that block as the nodes of the graph are created and destroyed.

10.9.1 The `malloc` and `free` Functions

The C standard library provides an explicit allocator known as the `malloc` package. Programs allocate blocks from the heap by calling the `malloc` function.

```
#include <stdlib.h>

void *malloc(size_t size);
```
returns: ptr if OK, NULL on error

The `malloc` function returns a pointer to a block of memory of at least `size` bytes that is suitably aligned for any kind of data object that might be contained in the block. On the Unix systems that we are familiar with, `malloc` returns a block that is aligned to an 8-byte (double-word) boundary. The `size_t` type is defined as an `unsigned int`.

> **Aside: How big is a word?**
> Recall from our discussion of IA32 machine code in Chapter 3 that Intel refers to 4-byte objects as *double-words*. However, throughout this section we will assume that *words* are 4-byte objects and that *double-words* are 8-byte objects, which is consistent with conventional terminology. **End Aside.**

If `malloc` encounters a problem (e.g., the program requests a block of memory that is larger than the available virtual memory), then it returns NULL and sets `errno`. `Malloc` does not initialize the memory it returns. Applications that want initialized dynamic memory can use `calloc`, a thin wrapper around the `malloc` function that initializes the allocated memory to zero. Applications that want to change the size of a previously allocated block can use the `realloc` function.

Dynamic memory allocators such as `malloc` can allocate or deallocate heap memory explicitly by using the `mmap` and `munmap` functions, or they can use the `sbrk` function:

```
#include <unistd.h>

void *sbrk(int incr);
```
returns: old `brk` pointer on success, -1 on error

The `sbrk` function grows or shrinks the heap by adding `incr` to the kernel's `brk` pointer. If successful, it returns the old value of `brk`, otherwise it returns -1 and sets `errno` to ENOMEM. If `incr` is zero, then `sbrk` returns the current value of `brk`. Calling `sbrk` with a negative `incr` is legal but tricky because the return value (the old value of `brk`) points to `abs(incr)` bytes past the new top of the heap.

Programs free allocated heap blocks by calling the `free` function.

```
#include <stdlib.h>

void free(void *ptr);
                                                    returns: nothing
```

The `ptr` argument must point to the beginning of an allocated block that was obtained from `malloc`. If not, then the behavior of `free` is undefined. Even worse, since it returns nothing, `free` gives no indication to the application that something is wrong. As we shall see in Section 10.11, this can produce some baffling run-time errors.

Figure 10.36 shows how an implementation of `malloc` and `free` might manage a (very) small heap of 16 words for a C program. Each box represents a 4-byte word. The heavy-lined rectangles correspond to allocated blocks (shaded) and free blocks (unshaded). Initially the heap consists of a single 16-word double-word aligned free block.

- *Figure 10.36(a):* The program asks for a 4-word block. `Malloc` responds by carving out a 4-word block from the front of the free block and returning a pointer to the first word of the block.
- *Figure 10.36(b):* The program requests a 5-word block. `Malloc` responds by allocating a 6-word block from the front of the free block. In this example, `malloc` pads the block with an extra word in order to keep the free block aligned on a double-word boundary.
- *Figure 10.36(c):* The program requests a 6-word block and `malloc` responds by carving out a 6-word block from the free block.
- *Figure 10.36(d)* The program frees the 6-word block that was allocated in Figure 10.36(b). Notice that after the call to `free` returns, the pointer `p2` still points to the freed block. It is the responsibility of the application not to use `p2` again until it is reinitialized by a new call to `malloc`.
- *Figure 10.36(e):* The program requests a 2-word block. In this case, `malloc` allocates a portion of the block that was freed in the previous step and returns a pointer to this new block.

10.9.2 Why Dynamic Memory Allocation?

The most important reason that programs use dynamic memory allocation is that often they do not know the sizes of certain data structures until the program actually runs. For example, suppose we are asked to write a C program that reads a list of n ASCII integers, one integer per line, from `stdin` into a C array. The input consists of the integer n, followed by the n integers to be read and stored into the array. The simplest approach is to define the array statically with some hard-coded maximum array size:

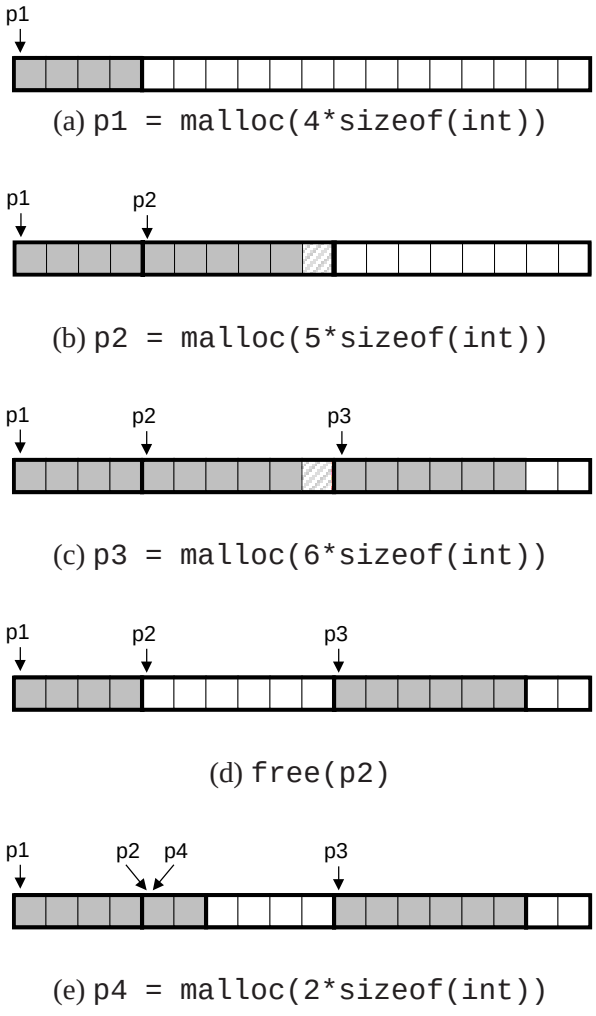


Figure 10.36: **Allocating and freeing blocks with `malloc`.** Each square corresponds to a word. Each heavy rectangle corresponds to a block. Allocated blocks are shaded. Free blocks are unshaded. Heap addresses increase from left to right.

```
1 #include "csapp.h"
2 #define MAXN 15213
3
4 int array[MAXN];
5
6 int main()
7 {
8     int i, n;
9
10    scanf("%d", &n);
11    if (n > MAXN)
12        app_error("Input file too big");
13    for (i = 0; i < n; i++)
14        scanf("%d", &array[i]);
15    exit(0);
16 }
```

Allocating arrays with hard-coded sizes like this is often a bad idea. The value of MAXN is arbitrary and has no relation to the actual amount of available virtual memory on the machine. Further, if the user of this program wanted to read a file that was larger than MAXN, the only recourse would be to recompile the program with a larger value of MAXN. While not a problem for this simple example, the presence of hard-coded array bounds can become a maintenance nightmare for large software products with millions of lines of code and numerous users.

A better approach is to allocate the array dynamically, at run time, after the value of n becomes known. With this approach, the maximum size of the array is limited only by the amount of available virtual memory.

```
1 #include "csapp.h"
2
3 int main()
4 {
5    int *array, i, n;
6
7    scanf("%d", &n);
8    array = (int *)Malloc(n * sizeof(int));
9    for (i = 0; i < n; i++)
10      scanf("%d", &array[i]);
11   exit(0);
12 }
```

Dynamic memory allocation is a useful and important programming technique. However, in order to use allocators correctly and efficiently, programmers need to have an understanding of how they work. We will discuss some of the gruesome errors that can result from the improper use of allocators in Section 10.11.

10.9.3 Allocator Requirements and Goals

Explicit allocators must operate within some rather stringent constraints.

- *Handling arbitrary request sequences.* An application can make an arbitrary sequence of allocate and free requests, subject to the constraint that each free request must correspond to a currently allocated block obtained from a previous allocate request. Thus the allocator cannot make any assumptions about the ordering of allocate and free requests. For example, the allocator cannot assume that all allocate requests are accompanied by a matching free, or that matching allocate and free requests are nested.

- *Making immediate responses to requests.* The allocator must respond immediately to allocate requests. Thus the allocator is not allowed to reorder or buffer requests in order to improve performance.

- *Using only the heap.* In order for the allocator to be scalable, any non-scalar data structures used by the allocator must be stored in the heap itself.

- *Aligning blocks (alignment requirement).* The allocator must align blocks in such a way that they can hold any type of data object. On most systems, this means that the block returned by the allocator is aligned on an eight-byte (double-word) boundary.

- *Not modifying allocated blocks.* Allocators can only manipulate or change free blocks. In particular, they are not allowed to modify or move blocks once they are allocated. Thus, techniques such as compaction of allocated blocks are not permitted.

Working within these constraints, the author of an allocator attempts to meet the often conflicting performance goals of maximizing throughput and memory utilization:

- *Goal 1: Maximizing throughput.* Given some sequence of n allocate and free requests

  $$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

  we would like to maximize an allocator's *throughput*, which is defined as the number of requests that it completes per unit time. For example, if an allocator completes 500 allocate requests and 500 free requests in 1 second, then its throughput is 1,000 operations per second. In general we can maximize throughput by minimizing the average time to satisfy allocate and free requests. As we'll see, it is not too difficult to develop allocators with reasonably good performance where the worst-case running time of an allocate request is linear in the number of free blocks and the running time of a free request is constant.

- *Goal 2: Maximizing memory utilization.* Naive programmers often incorrectly assume that virtual memory is an unlimited resource. In fact, the total amount of virtual memory allocated by all of the processes in a system is limited by the amount of swap space on disk. Good programmers realize that virtual memory is a finite resource that must be used efficiently. This is especially true for a dynamic memory allocator that might be asked to allocate and free large blocks of memory.

  There are a number of ways to characterize how efficiently an allocator uses the heap. In our experience, the most useful metric is *peak utilization*. As before, we are given some sequence of n allocate and free requests

  $$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

If an application requests a block of p bytes, then the resulting allocated block has a *payload* of p bytes. After request R_k has completed, let the *aggregate payload*, denoted P_k, be the sum of the payloads of the currently allocated blocks, and let H_k denote the current (monotonically nondecreasing) size of the heap.

Then the *peak utilization* over the first k requests, denoted by U_k, is given by

$$U_k = \frac{\max_{i \leq k} P_i}{H_k}.$$

The objective of the allocator then is to maximize the peak utilization U_{n-1} over the entire sequence. As we will see, there is a tension between maximizing throughput and utilization. In particular, it is easy to write an allocator that maximizes throughput at the expense of heap utilization. One of the interesting challenges in any allocator design is finding an appropriate balance between the two goals.

Aside: Relaxing the monotonicity assumption.
We could relax the monotonically nondecreasing assumption in our definition of U_k and allow the heap to grow up and down by letting H_k be the highwater mark over the first k requests. **End Aside.**

10.9.4 Fragmentation

The primary cause of poor heap utilization is a phenomenon known as *fragmentation*, which occurs when otherwise unused memory is not available to satisfy allocate requests. There are two forms of fragmentation: *internal fragmentation* and *external fragmentation*.

Internal fragmentation occurs when an allocated block is larger than the payload. This might happen for a number of reasons. For example, the implementation of an allocator might impose a minimum size on allocated blocks that is greater than some requested payload. Or, as we saw in Figure 10.36(b), the allocator might increase the block size in order to satisfy alignment constraints.

Internal fragmentation is straightforward to quantify. It is simply the sum of the differences between the sizes of the allocated blocks and their payloads. Thus, at any point in time, the amount of internal fragmentation depends only on the pattern of previous requests and the allocator implementation.

External fragmentation occurs when there *is* enough aggregate free memory to satisfy an allocate request, but no single free block is large enough to handle the request. For example, if the request in Figure 10.36(e) were for six words rather than two words, then the request could not be satisfied without requesting additional virtual memory from the kernel, even though there are six free words remaining in the heap. The problem arises because these six words are spread over two free blocks.

External fragmentation is much more difficult to quantify than internal fragmentation because it depends not only on the pattern of previous requests and the allocator implementation, but also on the pattern of *future* requests. For example, suppose that after k requests, all of the free blocks are exactly four words in size. Does this heap suffer from external fragmentation? The answer depends on the pattern of future requests. If all of the future allocate requests are for blocks that are smaller than four words, then there is no external fragmentation. On the other hand, if one or more requests ask for blocks larger than four words, then the heap does suffer from external fragmentation.

Since external fragmentation is difficult to quantify and impossible to predict, allocators typically employ heuristics that attempt to maintain small numbers of larger free blocks rather than large numbers of smaller free blocks.

10.9.5 Implementation Issues

The simplest imaginable allocator would organize the heap as a large array of bytes and a pointer `p` that initially points to the first byte of the array. To allocate `size` bytes, `malloc` would save the current value of `p` on the stack, increment `p` by `size`, and return the old value of `p` to the caller. `Free` would simply return to the caller without doing anything.

This naive allocator is an extreme point in the design space. Since each `malloc` and `free` execute only a handful of instructions, throughput would be extremely good. However, since the allocator never reuses any blocks, memory utilization would be extremely bad. A practical allocator that strikes a better balance between throughput and utilization must consider the following issues:

- *Free block organization:* How do we keep track of free blocks?
- *Placement:* How do we choose an appropriate free block in which to place a newly allocated block?
- *Splitting:* After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
- *Coalescing:* What do we do with a block that has just been freed?

The rest of this section looks at these issues in more detail. Since the basic techniques of placement, splitting, and coalescing cut across many different free block organizations, we will introduce them in the context of a simple free block organization known as an implicit free list.

10.9.6 Implicit Free Lists

Any practical allocator needs some data structure that allows it to distinguish block boundaries and to distinguish between allocated and free blocks. Most allocators embed this information in the blocks themselves. One simple approach is shown in Figure 10.37.

In this case, a block consists of a one-word *header*, the payload, and possibly some additional *padding*. The *header* encodes the block size (including the header and any padding) as well as whether the block is allocated or free. If we impose a double-word alignment constraint, then the block size is always a multiple of eight and the three low-order bits of the block size are always zero. Thus, we need to store only the 29 high-order bits of the block size, freeing the remaining three bits to encode other information. In this case, we are using the least significant of these bits (the *allocated bit*) to indicate whether the block is allocated or free. For example, suppose we have an allocated block with a block size of 24 (`0x18`) bytes. Then its header would be

```
0x00000018 | 0x1 = 0x00000019.
```

Similarly, a free block with a block size of 40 (`0x28`) bytes would have a header of

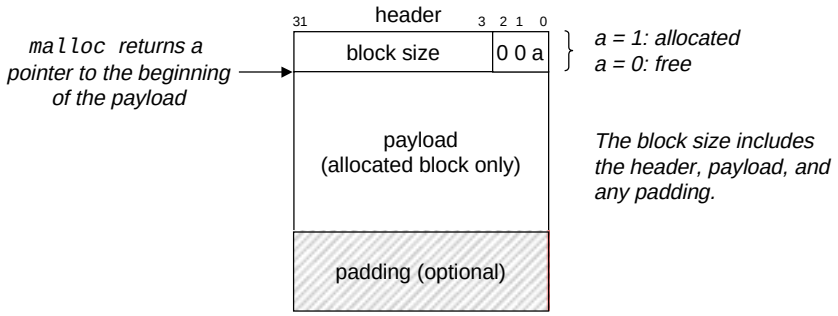


Figure 10.37: **Format of a simple heap block.**

```
0x00000028 | 0x0 = 0x00000028.
```

The header is followed by the payload that the application requested when it called `malloc`. The payload is followed by a chunk of unused padding that can be any size. There are a number of reasons for the padding. For example, the padding might be part of an allocator's strategy for combating external fragmentation. Or it might be needed to satisfy the alignment requirement.

Given the block format in Figure 10.37, we can organize the heap as a sequence of contiguous allocated and free blocks, as shown in Figure 10.38.

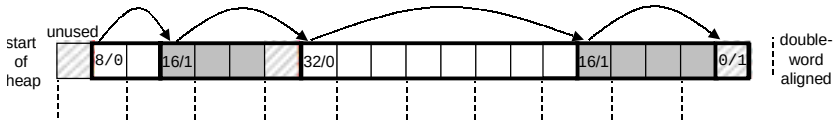


Figure 10.38: **Organizing the heap with an implicit free list.** Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

We call this organization an *implicit free list* because the free blocks are linked implicitly by the size fields in the headers. The allocator can indirectly traverse the entire set of free blocks by traversing *all* of the blocks in the heap. Notice that we need some kind of specially marked end block, in this example a terminating header with the allocated bit set and a size of zero. (As we will see in Section 10.9.12, setting the allocated bit simplifies the coalescing of free blocks.)

The advantage of an implicit free list is simplicity. A significant disadvantage is that the cost of any operation, such as placing allocated blocks, that requires a search of the free list will be linear in the *total* number of allocated and free blocks in the heap.

It is important to realize that the system's alignment requirement and the allocator's choice of block format impose a *minimum block size* on the allocator. No allocated or free block may be smaller than this minimum. For example, if we assume a double-word alignment requirement, then the size of each block must be a multiple of two words (8 bytes). Thus, the block format in Figure 10.37 induces a minimum block size of two words: one word for the header, and another to maintain the alignment requirement. Even if the application were to request a single byte, the allocator would still create a two-word block.

Practice Problem 10.6:

Determine the block sizes and header values that would result from the following sequence of `malloc` requests. Assumptions: (1) The allocator maintains double-word alignment, and uses an implicit free list with the block format from Figure 10.37. (2) Block sizes are rounded up to the nearest multiple of eight bytes.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(1)`		
`malloc(5)`		
`malloc(12)`		
`malloc(13)`		

10.9.7 Placing Allocated Blocks

When an application requests a block of k bytes, the allocator searches the free list for a free block that is large enough to hold the requested block. The manner in which the allocator performs this search is determined by the *placement policy*. Some common policies are *first fit*, *next fit*, and *best fit*.

First fit searches the free list from the beginning and chooses the first free block that fits. *Next fit* is similar to first fit, but instead of starting each search at the beginning of the list, it starts each search where the previous search left off. *Best fit* examines every free block and chooses the free block with the smallest size that fits.

An advantage of first fit is that it tends to retain large free blocks at the end of the list. A disadvantage is that it tends to leave "splinters" of small free blocks towards the beginning of the list, which will increase the search time for larger blocks. Next fit was first proposed by Knuth as an alternative to first fit, motivated by the idea that if we found a fit in some free block the last time, there is a good chance that the we will find a fit the next time in the remainder of the block. Next fit can run significantly faster than first fit, especially if the front of the list becomes littered with many small splinters. However, some studies suggest that next fit suffers from worse memory utilization than first fit. Studies have found that best fit generally enjoys better memory utilization than either first fit or next fit. However, the disadvantage of using best fit with simple free list organizations such as the implicit free list, is that it requires an exhaustive search of the heap. Later, we will look at more sophisticated segregated free list organizations that implement a best-fit policy without an exhaustive search of the heap.

10.9.8 Splitting Free Blocks

Once the allocator has located a free block that fits, it must make another policy decision about how much of the free block to allocate. One option is to use the entire free block. Although simple and fast, the main disadvantage is that it introduces internal fragmentation. If the placement policy tends to produce good fits, then some additional internal fragmentation might be acceptable.

However, if the fit is not good, then the allocator will usually opt to *split* the free block into two parts. The first part becomes the allocated block, and the remainder becomes a new free block. Figure 10.39 shows

how the allocator might split the eight-word free block in Figure 10.38 to satisfy an application's request for three words of heap memory.

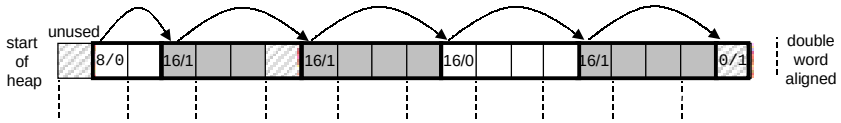


Figure 10.39: **Splitting a free block to satisfy a three-word allocation request.** Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

10.9.9 Getting Additional Heap Memory

What happens if the allocator is unable to find a fit for the requested block? One option is to try to create some larger free blocks by merging (coalescing) free blocks that are physically adjacent in memory (next section). However, if this does not yield a sufficiently large block, or if the free blocks are already maximally coalesced, then the allocator asks the kernel for additional heap memory, either by calling the `mmap` or `sbrk` functions. In either case, the allocator transforms the additional memory into one large free block, inserts the block into the free list, and then places the requested block in this new free block.

10.9.10 Coalescing Free Blocks

When the allocator frees an allocated block, there might be other free blocks that are adjacent to the newly freed block. Such adjacent free blocks can cause a phenomenon known as *false fragmentation*, where there is a lot of available free memory chopped up into small, unusable free blocks. For example, Figure 10.40 shows the result of freeing the block that was allocated in Figure 10.39. The result is two adjacent free blocks with payloads of three words each. As a result, a subsequent request for a payload of four words would fail, even though the aggregate size of the two free blocks is large enough to satisfy the request.

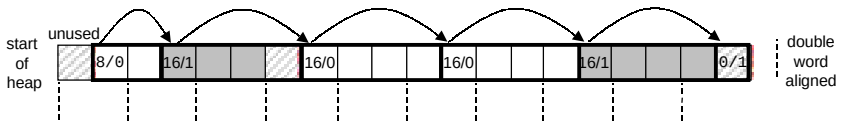


Figure 10.40: **An example of false fragmentation.** Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

To combat false fragmentation, any practical allocator must merge adjacent free blocks in a process known as *coalescing*. This raises an important policy decision about when to perform coalescing. The allocator can opt for *immediate coalescing* by merging any adjacent blocks each time a block is freed. Or it can opt for *deferred coalescing* by waiting to coalesce free blocks at some later time. For example, the allocator might defer coalescing until some allocation request fails, and then scan the entire heap, coalescing all free blocks.

Immediate coalescing is straightforward and can be performed in constant time, but with some request patterns it can introduce a form of thrashing where a block is repeatedly coalesced and then split soon thereafter. For example, in Figure 10.40 a repeated pattern of allocating and freeing a three-word block would introduce a lot of unnecessary splitting and coalescing. In our discussion of allocators, we will assume immediate coalescing, but you should be aware that fast allocators often opt for some form of deferred coalescing.

10.9.11 Coalescing with Boundary Tags

How does an allocator implement coalescing? Let us refer to the block we want to free as the *current block*. Then coalescing the next free block (in memory) is straightforward and efficient. The header of the current block points to the header of the next block, which can be checked to determine if the next block is free. If so, its size is simply added to the size of the current header and the blocks are coalesced in constant time.

But how would we coalesce the previous block? Given an implicit free list of blocks with headers, the only option would be to search the entire list, remembering the location of the previous block, until we reached the current block. With an implicit free list, this means that each call to `free` would require time linear in the size of the heap. Even with more sophisticated free list organizations, the search time would not be constant.

Knuth developed a clever and general technique, known as *boundary tags*, that allows for constant-time coalescing of the previous block. The idea, which is shown in Figure 10.41, is to add a *footer* (the boundary tag) at the end of each block, where the footer is a replica of the header. If each block includes such a footer, then the allocator can determine the starting location and status of the previous block by inspecting its footer, which is always one word away from the start of the current block.

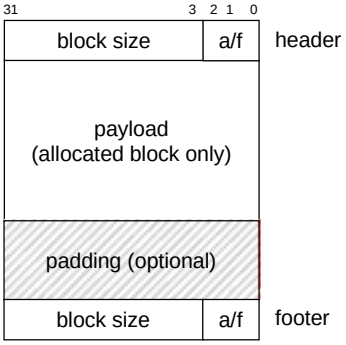


Figure 10.41: **Format of heap block that uses a boundary tag.**

Consider all the cases that can exist when the allocator frees the current block:

1. The previous and next blocks are both allocated.
2. The previous block is allocated and the next block is free.
3. The previous block is free and the next block is allocated.

4. The previous and next blocks are both free.

Figure 10.42 shows how we would coalesce each of the four cases.

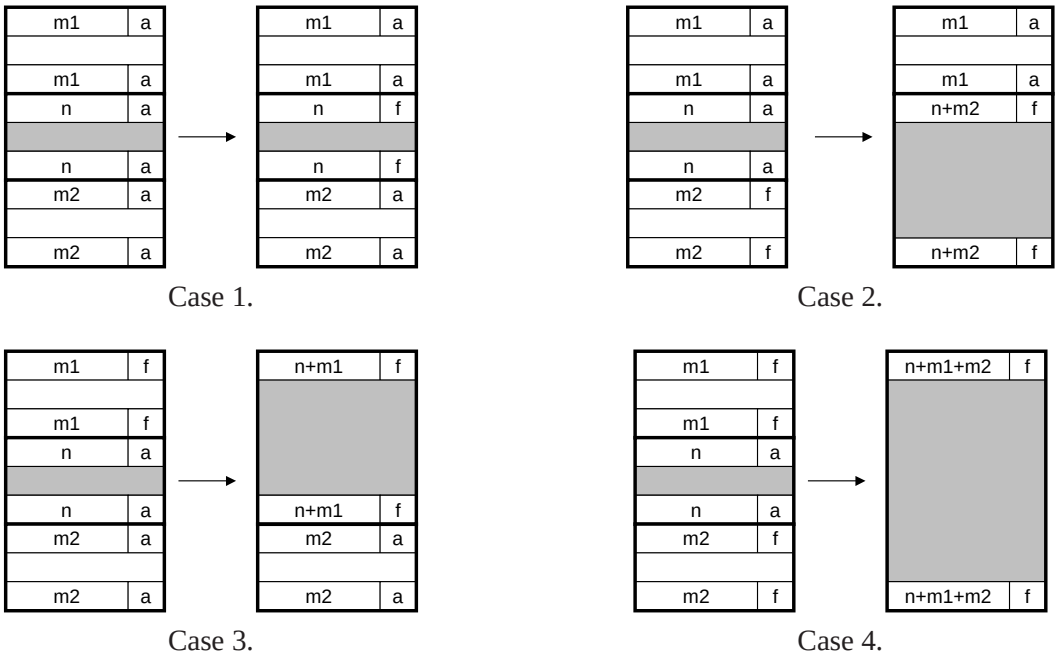


Figure 10.42: **Coalescing with boundary tags.** Case 1: prev and next allocated. Case 2: prev allocated, next free. Case 3: prev free, next allocated. Case 4: next and prev free.

In Case 1, both adjacent blocks are allocated and thus no coalescing is possible. So the status of the current block is simply changed from allocated to free. In Case 2, the current block is merged with the next block. The header of the current block and the footer of the next block are updated with the combined sizes of the current and next blocks. In Case 3, the previous block is merged with the current block. The header of the previous block and the footer of the current block are updated with the combined sizes of the two blocks. In Case 4, all three blocks are merged to form a single free block, with the header of the previous block and the footer of the next block updated with the combined sizes of the three blocks. In each case, the coalescing is performed in constant time.

The idea of boundary tags is a simple and elegant one that generalizes to many different types of allocators and free list organizations. However, there is a potential disadvantage. Requiring each block to contain both a header and a footer can introduce significant memory overhead if an application manipulates many small blocks. For example, if a graph application dynamically creates and destroys graph nodes by making repeated calls to `malloc` and `free`, and each graph node requires only a couple of words of memory, then the header and the footer will consume half of each allocated block.

Fortunately, there is a clever optimization of boundary tags that eliminates the need for a footer in allocated blocks. Recall that when we attempt to coalesce the current block with the previous and next blocks in memory, the size field in the footer of the previous block is only needed if the previous block is *free*. If we

were to store the allocated/free bit of the previous block in one of the excess low-order bits of the current block, then allocated blocks would not need footers, and we could use that extra space for payload. Note however, that free blocks still need footers.

Practice Problem 10.7:

Determine the minimum block size for each of the following combinations of alignment requirements and block formats. Assumptions: Implicit free list, zero-sized payloads are not allowed, and headers and footers are stored in four-byte words.

Alignment	Allocated block	Free block	Minimum block size (bytes)
Single-word	Header and footer	Header and footer	
Single-word	Header, but no footer	Header and footer	
Double-word	Header and footer	Header and footer	
Double-word	Header, but no footer	Header and footer	

10.9.12 Putting it Together: Implementing a Simple Allocator

Building an allocator is a challenging task. The design space is large, with numerous alternatives for block format, free list format, and placement, splitting, and coalescing policies. Another challenge is that you are often forced to program outside the safe, familiar confines of the type system, relying on the error-prone pointer casting and pointer arithmetic that is typical of low-level systems programming. While allocators do not require enormous amounts of code, they are subtle and unforgiving. Students familiar with higher-level languages such as C++ or Java often hit a conceptual wall when they first encounter this style of programming. To help you clear this hurdle, we will work through the implementation of a simple allocator based on an implicit free list with immediate boundary-tag coalescing.

General Allocator Design

Our allocator uses a model of the memory system provided by the `memlib.c` package shown in Figure 10.43. The purpose of the model is to allow us to run our allocator without interfering with the existing system-level `malloc` package. The `mem_init` function models the virtual memory available to the heap as a large, double-word aligned array of bytes. The bytes between `mem_start_brk` and `mem_brk` represent allocated virtual memory. The bytes following `mem_brk` represent unallocated virtual memory. The allocator requests additional heap memory by calling the `mem_sbrk` function, which has the same interface as the system's `sbrk` function, and the same semantics, except that it rejects requests to shrink the heap.

The allocator itself is contained in a source file (`malloc.c`) that users can compile and link into their applications. The allocator exports three functions to application programs:

```
1 int mm_init(void);
2 void *mm_malloc(size_t size);
3 void mm_free(void *bp);
```

code/vm/memlib.c

```
1 #include "csapp.h"
2
3 /* private global variables */
4 static void *mem_start_brk;  /* points to first byte of the heap */
5 static void *mem_brk;        /* points to last byte of the heap */
6 static void *mem_max_addr;   /* max virtual address for the heap */
7
8 /*
9  * mem_init - initializes the memory system model
10 */
11 void mem_init(int size)
12 {
13     mem_start_brk = (void *)Malloc(size); /* models available VM */
14     mem_brk = mem_start_brk;              /* heap is initially empty */
15     mem_max_addr = mem_start_brk + size;  /* max VM address for heap */
16 }
17
18 /*
19  * mem_sbrk - simple model of the the sbrk function. Extends the heap
20  *    by incr bytes and returns the start address of the new area. In
21  *    this model, the heap cannot be shrunk.
22  */
23 void *mem_sbrk(int incr)
24 {
25     void *old_brk = mem_brk;
26
27     if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
28         errno = ENOMEM;
29         return (void *)-1;
30     }
31     mem_brk += incr;
32     return old_brk;
33 }
```

code/vm/memlib.c

Figure 10.43: `memlib.c`: **Memory system model.**

The `mm_init` function initializes the allocator, returning 0 if successful and -1 otherwise. The `mm_malloc` and `mm_free` functions have the same interfaces and semantics as their system counterparts. The allocator uses the block format shown in Figure 10.41. The minimum block size is 16 bytes. The free list is organized as an implicit free list, with the invariant form shown in Figure 10.44.

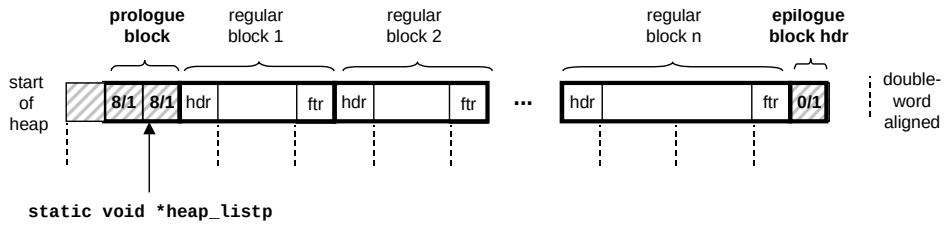


Figure 10.44: **Invariant form of the implicit free list.**

The first word is an unused padding word aligned to a double-word boundary. The padding is followed by a special *prologue block*, which is an eight-byte allocated block consisting of only a header and a footer. The prologue block is created during initialization and is never freed. Following the prologue block are zero or more regular blocks that are created by calls to `malloc` or `free`. The heap always ends with a special *epilogue block*, which is a zero-sized allocated block that consists of only a header. The prologue and epilogue blocks are tricks that eliminate the edge conditions during coalescing. The allocator uses a single private (`static`) global variable (`heap_listp`) that always points to the prologue block. (As a minor optimization, we could make it point to the next block instead of the prologue block.)

Basic Constants and Macros for Manipulating the Free List

Figure 10.45 shows some basic constants that we will use throughout the allocator code. Lines 2–5 define some basic size constants: the sizes of words (WSIZE) and double-words (DSIZE), the size of the initial free block and the default size for expanding the heap (CHUNKSIZE), and the number of overhead bytes consumed by the header and footer (OVERHEAD).

Manipulating the headers and footers in the free list can be troublesome because it demands extensive use of casting and pointer arithmetic. Thus, we find it helpful to define a small set of macros for accessing and traversing the free list (lines 10–26). The PACK macro (line 10) combines a size and an allocate bit and returns a value that can be stored in a header or footer.

The GET macro (line 13) reads and returns the word referenced by argument `p`. The casting here is crucial. The argument `p` is typically a (`void *`) pointer, which cannot be dereferenced directly. Similarly, the PUT macro (line 14) stores `val` in the word pointed at by argument `p`.

The GET_SIZE and GET_ALLOC macros (lines 17–18) return the size and allocated bit, respectively, from a header or footer at address `p`. The remaining macros operate on *block pointers* (denoted `bp`), that point to the first payload byte. Given a block pointer `bp`, the HDRP and FTRP macros (lines 21–22) return pointers to the block header and footer, respectively. The NEXT_BLKP and PREV_BLKP macros (lines 25–26) return the block pointers of the next and previous blocks, respectively.

The macros can be composed in various ways to manipulate the free list. For example, given a pointer `bp` to the current block, we could use the following line of code to determine the size of the next block in memory:

code/vm/malloc.c

```
1 /* Basic constants and macros */
2 #define WSIZE       4       /* word size (bytes) */
3 #define DSIZE       8       /* doubleword size (bytes) */
4 #define CHUNKSIZE  (1<<12)  /* initial heap size (bytes) */
5 #define OVERHEAD    8       /* overhead of header and footer (bytes) */
6
7 #define MAX(x, y) ((x) > (y)? (x) : (y))
8
9 /* Pack a size and allocated bit into a word */
10 #define PACK(size, alloc)  ((size) | (alloc))
11
12 /* Read and write a word at address p */
13 #define GET(p)       (*(size_t *)(p))
14 #define PUT(p, val)  (*(size_t *)(p) = (val))
15
16 /* Read the size and allocated fields from address p */
17 #define GET_SIZE(p)  (GET(p) & ~0x7)
18 #define GET_ALLOC(p) (GET(p) & 0x1)
19
20 /* Given block ptr bp, compute address of its header and footer */
21 #define HDRP(bp)       ((void *)(bp) - WSIZE)
22 #define FTRP(bp)       ((void *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
23
24 /* Given block ptr bp, compute address of next and previous blocks */
25 #define NEXT_BLKP(bp)  ((void *)(bp) + GET_SIZE(((void *)(bp) - WSIZE)))
26 #define PREV_BLKP(bp)  ((void *)(bp) - GET_SIZE(((void *)(bp) - DSIZE)))
```

code/vm/malloc.c

Figure 10.45: **Basic constants and macros for manipulating the free list.**

```
size_t size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
```

Creating the Initial Free List

Before calling `mm_malloc` or `mm_free`, the application must initialize the heap by calling the `mm_init` function (Figure 10.46). The `mm_init` function gets four words from the memory system and initializes

code/vm/malloc.c

```
1  int mm_init(void)
2  {
3      /* create the initial empty heap */
4      if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
5          return -1;
6      PUT(heap_listp, 0);                          /* alignment padding */
7      PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
8      PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
9      PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));   /* epilogue header */
10     heap_listp += DSIZE;
11
12     /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13     if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14         return -1;
15     return 0;
16 }
```

code/vm/malloc.c

Figure 10.46: `mm_init`: **Creates a heap with an initial free block.**

them to create the empty free list (lines 4–10). It then calls the `extend_heap` function (Figure 10.47), which extends the heap by CHUNKSIZE bytes and creates the initial free block. At this point, the allocator is initialized and ready to accept allocate and free requests from the application.

The `extend_heap` function is invoked in two different circumstances: (1) when the heap is initialized, and (2) when `mm_malloc` is unable to find a suitable fit. To maintain alignment, `extend_heap` rounds up the requested size to the nearest multiple of 2 words (8 bytes), and then requests the additional heap space from the memory system (lines 7–9).

The remainder of the `extend_heap` function (lines 12–17) is somewhat subtle. The heap begins on a double-word aligned boundary, and every call to `extend_heap` returns a block whose size is an integral number of double-words. Thus, every call to `mem_sbrk` returns a double-word aligned chunk of memory immediately following the header of the epilogue block. This header becomes the header of the new free block (line 12), and the last word of the chunk becomes the new epilogue block header (line 14). Finally, in the likely case that the previous heap was terminated by a free block, we call the `coalesce` function to merge the two free blocks and return the block pointer of the merged blocks (line 17).

code/vm/malloc.c

```
1 static void *extend_heap(size_t words)
2 {
3     char *bp;
4     size_t size;
5
6     /* Allocate an even number of words to maintain alignment */
7     size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8     if ((int)(bp = mem_sbrk(size)) < 0)
9         return NULL;
10
11    /* Initialize free block header/footer and the epilogue header */
12    PUT(HDRP(bp), PACK(size, 0));         /* free block header */
13    PUT(FTRP(bp), PACK(size, 0));         /* free block footer */
14    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* new epilogue header */
15
16    /* Coalesce if the previous block was free */
17    return coalesce(bp);
18 }
```

code/vm/malloc.c

Figure 10.47: `extend_heap`**: Extends the heap with a new free block.**

Freeing and Coalescing Blocks

An application frees a previously allocated block by calling the `mm_free` function (Figure 10.48), which frees the requested block (`bp`), and then merges adjacent free blocks using the boundary-tags coalescing technique described in Section 10.9.11.

The code in the `coalesce` helper function is a straightforward implementation of the four cases outlined in Figure 10.42. There is one somewhat subtle aspect. The free list format we have chosen — with its prologue and epilogue blocks that are always marked as allocated — allows us to ignore the potentially troublesome edge conditions where the requested block `bp` is at the beginning or end of the heap. Without these special blocks, the code would be messier, more error-prone, and slower because we would have to check for these rare edge conditions on each and every free request.

Allocating Blocks

An application requests a block of `size` bytes of memory by calling the `mm_malloc` function (Figure 10.49). After checking for spurious requests (lines 8–9), the allocator must adjust the requested block size to allow room for the header and the footer, and to satisfy the double-word alignment requirement. Lines 12–13 enforce the minimum block size of 16 bytes: eight (DSIZE) bytes to satisfy the alignment requirement, and eight more (OVERHEAD) for the header and footer. For requests over eight bytes (line 15), the general rule is to add in the overhead bytes and then round up to the nearest multiple of eight (DSIZE).

Once the allocator has adjusted the requested size, it searches the free list for a suitable free block (line 18).

code/vm/malloc.c

```
1  void mm_free(void *bp)
2  {
3      size_t size = GET_SIZE(HDRP(bp));
4
5      PUT(HDRP(bp), PACK(size, 0));
6      PUT(FTRP(bp), PACK(size, 0));
7      coalesce(bp);
8  }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) {            /* Case 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) {      /* Case 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size,0));
24         return(bp);
25     }
26
27     else if (!prev_alloc && next_alloc) {      /* Case 3 */
28         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
29         PUT(FTRP(bp), PACK(size, 0));
30         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
31         return(PREV_BLKP(bp));
32     }
33
34     else {                                     /* Case 4 */
35         size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
36             GET_SIZE(FTRP(NEXT_BLKP(bp)));
37         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
38         PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
39         return(PREV_BLKP(bp));
40     }
41 }
```

code/vm/malloc.c

Figure 10.48: `mm_free`**: Frees a block and uses boundary-tag coalescing to merge it with any adjacent free blocks in constant time.**

code/vm/malloc.c

```
1 void *mm_malloc(size_t size)
2 {
3     size_t asize;      /* adjusted block size */
4     size_t extendsize; /* amount to extend heap if no fit */
5     char *bp;
6
7     /* Ignore spurious requests */
8     if (size <= 0)
9         return NULL;
10
11    /* Adjust block size to include overhead and alignment reqs. */
12    if (size <= DSIZE)
13        asize = DSIZE + OVERHEAD;
14    else
15        asize = DSIZE * ((size + (OVERHEAD) + (DSIZE-1)) / DSIZE);
16
17    /* Search the free list for a fit */
18    if ((bp = find_fit(asize)) != NULL) {
19        place(bp, asize);
20        return bp;
21    }
22
23    /* No fit found. Get more memory and place the block */
24    extendsize = MAX(asize,CHUNKSIZE);
25    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26        return NULL;
27    place(bp, asize);
28    return bp;
29 }
```

code/vm/malloc.c

Figure 10.49: `mm_malloc`**: Allocates a block from the free list.**

If there is a fit, then the allocator places the requested block and optionally splits the excess (line 19), and then returns the address of the newly allocated block (line 20).

If the allocator cannot find a fit, then it extends the heap with a new free block (lines 24–26), places the requested block in the new free block and optionally splitting the block (line 27), and then return a pointer to the newly allocated block (line 28).

Practice Problem 10.8:

Implement a `find_fit` function for the simple allocator described in Section 10.9.12.

```
static void *find_fit(size_t asize)
```

Your solution should perform a first-fit search of the implicit free list.

Practice Problem 10.9:

Implement a `place` function for the example allocator.

```
static void place(void *bp, size_t asize)
```

Your solution should place the requested block at the beginning of the free block, splitting only if the size of the remainder would equal or exceed the minimum block size.

10.9.13 Explicit Free Lists

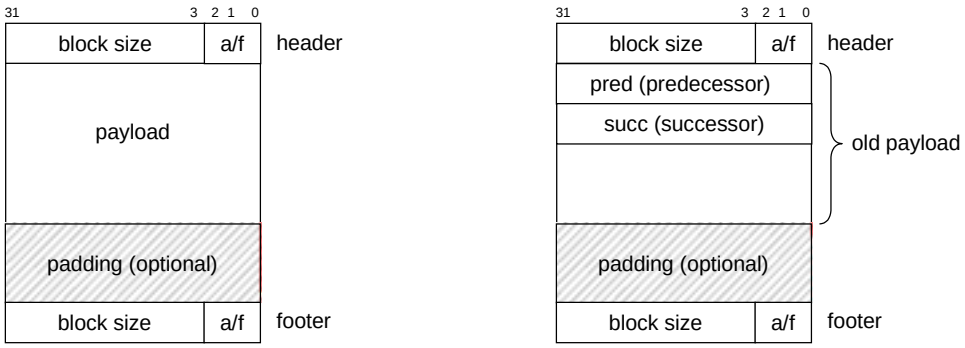
The implicit free list provides us with a simple way to introduce some basic allocator concepts. However, because block allocation is linear in the total number of heap blocks, the implicit free list is not appropriate for a general-purpose allocator (although it might be fine for a special-purpose allocator where the number of heap blocks is known beforehand to be small).

A better approach is to organize the free blocks into some form of explicit data structure. Since by definition the body of a free block is not needed by the program, the pointers that implement the data structure can be stored within the bodies of the free blocks. For example, the heap can be organized as a doubly-linked free list by including a `pred` (predecessor) and `succ` (successor) pointer in each free block, as shown in Figure 10.50.

Using a doubly-linked list instead of an implicit free list reduces the first fit allocation time from linear in the total number of blocks to linear in the number of *free* blocks. However, the time to free a block can be either linear or constant, depending on the policy we choose for ordering the blocks in the free list.

One approach is to maintain the list in *last-in first-out (LIFO)* order by inserting newly freed blocks at the beginning of the list. With a LIFO ordering and a first fit placement policy, the allocator inspects the most recently used blocks first. In this case, freeing a block can be performed in constant time. If boundary tags are used, then coalescing can also be performed in constant time.

Another approach is to maintain the list in *address order*, where the address of each block in the list is less than the address of its successor. In this case, freeing a block requires a linear-time search to locate the



(a) Allocated block (b) Free block

Figure 10.50: **Format of heap blocks that use doubly-linked free lists.**

appropriate predecessor. The trade-off is that address-ordered first fit enjoys better memory utilization than LIFO-ordered first fit, approaching the utilization of best fit.

A disadvantage of explicit lists in general is that free blocks must be large enough to contain all of the necessary pointers, as well as the header and possibly a footer. This results in a larger minimum block size, and potentially the degree of internal fragmentation.

10.9.14 Segregated Free Lists

As we have seen, an allocator that uses a single linked list of free blocks requires time linear in the number of free blocks to allocate a block. A popular approach for reducing the allocation time, known generally as *segregated storage*, is to maintain multiple free lists, where each list holds blocks that are roughly the same size.

The general idea is to partition the set of all possible block sizes into equivalence classes called *size classes*. There are many ways to define the size classes. For example, we might partition the block sizes by powers of two:

$$\{1\}, \{2\}, \{3, 4\}, \{5 – 8\}, \cdots, \{1025 – 2048\}, \{2049 – 4096\}, \{4097 – \infty\}$$

Or we might assign small blocks to their own size classes and partition large blocks by powers of two:

$$\{1\}, \{2\}, \{3\}, \cdots, \{1023\}, \{1024\}, \cdots, \{1025 – 2048\}, \{2049 – 4096\}, \{4097 – \infty\}$$

The allocator maintains an array of free lists, with one free list per size class, ordered by increasing size. When the allocator needs a block of size n, it searches the appropriate free list. If it cannot find a block that fits, it searches the next list, and so on.

The dynamic storage allocation literature describes dozens of variants of segregated storage that differ in how they define size classes, when they perform coalescing, when they request additional heap memory from the operating system, whether they allow splitting, and so forth. To give you a sense of what is possible, we will describe two of the basic approaches: *simple segregated storage* and *segregated fits*.

Simple Segregated Storage

With simple segregated storage, the free list for each size class contains same-sized blocks, each the size of the largest element of the size class. For example, if some size class is defined as $\{17 - 32\}$, then the free list for that class consists entirely of blocks of size 32.

To allocate a block of some given size, we check the appropriate free list. If the list is not empty, we simply allocate the first block in its entirety. Free blocks are never split to satisfy allocation requests. If the list is empty, the allocator requests a fixed-sized chunk of additional memory from the operating system (typically a multiple of the page size), divides the chunk into equal-sized blocks, and links the blocks together to form the new free list. To free a block, the allocator simply inserts the block at the front of the appropriate free list.

There are a number of advantages to this simple scheme. Allocating and freeing blocks are both fast constant-time operations. Further, the combination of the same-sized blocks in each chunk, no splitting, and no coalescing means that there is very little per-block memory overhead. Since each chunk has only same-sized blocks, the size of an allocated block can be inferred from its address. Since there is no coalescing, allocated blocks do not need an allocated/free flag in the header. Thus allocated blocks require no headers, and since there is no coalescing, they do not require any footers either. Since allocate and free operations insert and delete blocks at the beginning of the free list, the list need only be singly-linked instead of doubly-linked. The bottom line is that the only required field in any block is a one-word `succ` pointer in each free block, and thus the minimum block size is only one word.

A significant disadvantage is that simple segregated storage is susceptible to internal and external fragmentation. Internal fragmentation is possible because free blocks are never split. Worse, certain reference patterns can cause extreme external fragmentation because free blocks are never coalesced (Problem 10.10).

Researchers have proposed a crude form of coalescing to combat external fragmentation. The allocator keeps track of the number of free blocks in each memory chunk returned by the operating system. Whenever a chunk consists entirely of free blocks, the allocator removes the chunk from its current size class and makes it available for other size classes.

Practice Problem 10.10:

Describe a reference pattern that results in severe external fragmentation in an allocator based on simple segregated storage.

Segregated Fits

With this approach, the allocator maintains an array of free lists. Each free list is associated with a size class and is organized as some kind of explicit or implicit list. Each list contains potentially different-sized blocks whose sizes are members of the size class. There are many variants of segregated fits allocators. Here we describe a simple version.

To allocate a block, we determine the size class of the request and do a first-fit search of the appropriate free list for a block that fits. If we find one, then we (optionally) split it and insert the fragment in the appropriate free list. If we cannot find a block that fits, then we search the free list for the next larger size class. We

repeat until we find a block that fits. If none of free lists yields a block that fits, then we request additional heap memory from the operating system, allocate the block out of this new heap memory, and place the remainder in the largest size class. To free a block, we coalesce and place the result on the appropriate free list.

The segregated fits approach is a popular choice with production-quality allocators such as the GNU `malloc` package provided in the C standard library because it is both fast and memory efficient. Search times are reduced because searches are limited to particular parts of the heap instead of the entire heap. Memory utilization can improve because of the interesting fact that a simple first-fit search of a segregated free list approximates a best-fit search of the entire heap.

Buddy Systems

A *buddy system* is a special case of segregated fits where each size class is a power of two. The basic idea is that given a heap of 2^m words, we maintain a separate free list for each block size 2^k, where $0 \leq k \leq m$. Requested block sizes are rounded up to the nearest power of two. Originally, there is one free block of size 2^m words.

To allocate a block of size 2^k, we find the first available block of size 2^j, such that $k \leq j \leq m$. If $j = k$, then we are done. Otherwise we recursively split the block in half until $j = k$. As we perform this splitting, each remaining half (known as a *buddy*), is placed on the appropriate free list. To free a block of size 2^k, we continue coalescing with the free. When we encounter a allocated buddy, we stop the coalescing.

A key fact about buddy systems is that given the address and size of a block, it is easy to compute the address of its buddy. For example, a block of size 32 byes with address

```
xxx...x00000
```

has its buddy at address

```
xxx...x10000
```

In other words, the addresses of a block and its buddy differ in exactly one bit position.

The major advantage of a buddy system allocator is its fast searching and coalescing. The major disadvantage is that the power-of-two requirement on the block size can cause significant internal fragmentation. For this reason, buddy system allocators are not appropriate for general-purpose workloads. However, for certain application-specific workloads, where the block sizes are known in advance to be powers of two, buddy system allocators have a certain appeal.

10.10 Garbage Collection

With an explicit allocator such as the C `malloc` package, an application allocates and frees heap blocks by making calls to `malloc` and `free`. It is the application's responsibility to free any allocated blocks that it no longer needs.

Failing to free allocated blocks is a common programming error. For example, consider the following C function that allocates a block of temporary storage as part of its processing.

```
1 void garbage()
2 {
3     int *p = (int *)Malloc(15213);
4
5     return; /* array p is garbage at this point */
6 }
```

Since `p` is no longer needed by the program, it should have been freed before `foo` returned. Unfortunately, the programmer has forgotten to free the block. It remains allocated for the lifetime of the program, needlessly occupying heap space that could be used to satisfy subsequent allocation requests.

A *garbage collector* is a dynamic storage allocator that automatically frees allocated blocks that are no longer needed by the program. Such blocks are known as *garbage* and hence the term garbage collector. The process of automatically reclaiming heap storage is known as *garbage collection*. In a system that supports garbage collection, applications explicitly allocate heap blocks but never explicitly free them. In the context of a C program, the application calls `malloc`, but never calls `free`. Instead, the garbage collector periodically identifies the garbage blocks and makes the appropriate calls to `free` to place those blocks back on the free list.

Garbage collection dates back to Lisp systems developed by McCarthy at MIT in the early 1960s. It is an important part of modern language systems such as Java, ML, Perl, and Mathematica, and it remains an active and important area of research. The literature describes an amazing number of approaches for garbage collection. We will limit our discussion to McCarthy's original *Mark&Sweep* algorithm, which is interesting because it can be built on top of an existing `malloc` package to provide garbage collection for C and C++ programs.

10.10.1 Garbage Collector Basics

A garbage collector views memory as a directed *reachability graph* of the form shown in Figure 10.51. The nodes of the graph are partitioned into a set of *root nodes* and a set of *heap nodes*. Each heap node corresponds to an allocated block in the heap. A directed edge $p \rightarrow q$ means that some location in block p points to some location in block q. Root nodes correspond to locations not in the heap that contain pointers into the heap. These locations can be registers, variables on the stack, or global variables in the read-write data area of virtual memory.

We say that a node p is *reachable* if there exists a directed path from any root node to p. At any point in time, the unreachable nodes correspond to garbage that can never be used again by the application. The role of a garbage collector is to maintain some representation of the reachability graph and periodically reclaim the unreachable nodes by freeing them and returning them to the free list.

Garbage collectors for languages like ML and Java, which exert tight control over how applications create and use pointers, can maintain an exact representation of the reachability graph, and thus can reclaim all garbage. However, collectors for languages like C and C++ cannot in general maintain exact representations

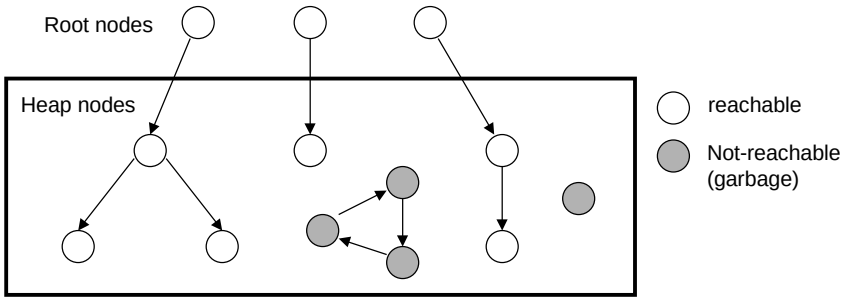


Figure 10.51: **A garbage collector's view of memory as a directed graph.**

of the reachability graph. Such collectors are known as *conservative garbage collectors*. They are conservative in the sense that each reachable block is correctly identified as reachable, while some unreachable nodes might be incorrectly identified as reachable.

Collectors can provide their service on demand, or they can run as separate threads in parallel with the application, continuously updating the reachability graph and reclaiming garbage. For example, consider how we might incorporate a conservative collector for C programs into an existing `malloc` package, as shown in Figure 10.52.

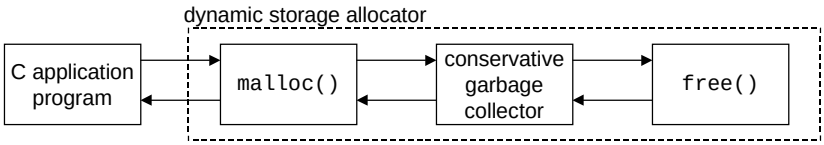


Figure 10.52: **Integrating a conservative garbage collector and a C `malloc` package.**

The application calls `malloc` in the usual manner whenever it needs heap space. If `malloc` is unable to find a free block that fits, then it calls the garbage collector in hopes of reclaiming some garbage to the free list. The collector identifies the garbage blocks and returns them to the heap by calling the `free` function. The key idea is that the collector calls `free` instead of the application. When the call to the collector returns, `malloc` tries again to find a free block that fits. If that fails, then it can ask the operating system for additional memory. Eventually `malloc` returns a pointer to the requested block (if successful) or the NULL pointer (if unsuccessful).

10.10.2 Mark&Sweep Garbage Collectors

A Mark&Sweep garbage collector consists of a *mark phase*, which marks all reachable and allocated descendents of the root nodes, followed by a *sweep phase*, which frees each unmarked allocated block. Typically, one of the spare low-order bits in the block header is used to indicate whether a block is marked or not.

Our description of Mark&Sweep will assume the following functions, where `ptr` is defined as `typedef void *ptr`.

- `ptr isPtr(ptr p)`: If `p` points to some word in an allocated block, returns a pointer `b` to the beginning of that block. Returns NULL otherwise.
- `int blockMarked(ptr b)`: Returns `true` if block `b` is already marked.
- `int blockAllocated(ptr b)`: Returns `true` if block `b` is allocated.
- `void markBlock(ptr b)`: Marks block `b`.
- `int length(b)`: Returns the length in words (excluding the header) of block `b`.
- `void unmarkBlock(ptr b)`: Changes the status of block `b` from marked to unmarked.
- `ptr nextBlock(ptr b)`: Returns the successor of block b in the heap.

The mark phase calls the `mark` function shown in Figure 10.53(a) once for each root node. The `mark` function returns immediately if `p` does not point to an allocated and unmarked heap block. Otherwise, it marks the block and calls itself recursively on each word in block. Each call to the `mark` function marks any unmarked and reachable descendents of some root node. At the end of the mark phase, any allocated block that is not marked is guaranteed to be unreachable, and hence garbage that can be reclaimed in the sweep phase.

```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}
```

```
void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
         unmarkBlock(b);
      else if (blockAllocated(b))
         free(b);
      b = nextBlock(b);
   }
   return;
}
```

Figure 10.53: **Pseudo-code for the `mark` and `sweep` functions.**

The sweep phase is a single call to the `sweep` function shown in Figure 10.53(b). The `sweep` function iterates over each block in the heap, freeing any unmarked allocated blocks (i.e., garbage) that it encounters.

Figure 10.54 shows a graphical interpretation of Mark&Sweep for a small heap. Block boundaries are indicated by heavy lines. Each square corresponds to a word of memory. Each block has a one-word header, which is either marked or unmarked.

Initially, the heap in Figure 10.53 consists of six allocated blocks, each of which is unmarked. Block 3 contains a pointer to block 1. Block 4 contains pointers to blocks 3 and 6. The root points to block 4. After the mark phase, blocks 1,3, 4, and 6 are marked because they are reachable from the root. Blocks 2 and 5 are unmarked because they are unreachable. After the sweep phase, the two unreachable blocks are reclaimed to the free list.

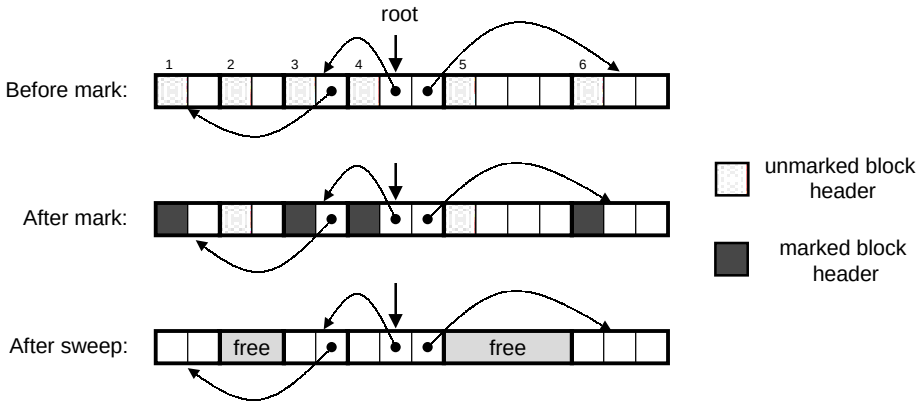


Figure 10.54: **Mark and sweep example.** Note that the arrows in this example denote memory references, and not free list pointers.

10.10.3 Conservative Mark&Sweep for C Programs

Mark&Sweep is an appropriate approach for garbage collecting C programs because it works in place without moving any blocks. However, the C language poses some interesting challenges for the implementation of the `isPtr` function.

First, C does not tag memory locations with any type information. Thus, there is no obvious way for `isPtr` to determine if its input parameter `p` is a pointer or not. Second, even if we were to know that `p` was a pointer, there would be no obvious way for `isPtr` to determine whether `p` points to some location in the payload of an allocated block.

One solution to the latter problem is to maintain the set of allocated blocks as a balanced binary tree that maintains the invariant that all blocks in the left subtree are located at smaller addresses and all blocks in the right subtree are located in larger addresses. As shown in Figure 10.55, this requires two additional fields (`left` and `right`) in the header of each allocated block. Each field points to the header of some allocated block.

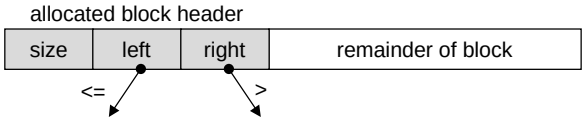


Figure 10.55: **Left and right pointers in a balanced tree of allocated blocks.**

The `isPtr(ptr p)` function uses the tree to perform a binary search of the allocated blocks. At each step, it relies on the size field in the block header to determine if `p` falls within the extent of the block.

The balanced tree approach is correct in the sense that it is guaranteed to mark all of the nodes that are reachable from the roots. This is a necessary guarantee, as application users would certainly not appreciate having their allocated blocks prematurely returned to the free list. However, it is conservative in the sense that it may incorrectly mark blocks that are actually unreachable, and thus it may fail to free some garbage.

While this does not affect the correctness of application programs, it can result in unnecessary external fragmentation.

The fundamental reason that Mark&Sweep collectors for C programs must be conservative is that the C language does not tag memory locations with type information. Thus, scalars like `ints` or `floats` can masquerade as pointers. For example, suppose that some reachable allocated block contains an `int` in its payload whose value happens to correspond to an address in the payload of some other allocated block b. There is no way for the collector to infer that the data is really an `int` and not a pointer. Thus the allocator must conservatively mark block b as reachable, when in fact it might not be.

10.11 Common Memory-related Bugs in C Programs

Managing and using virtual memory can be a difficult and error-prone task for the C programmers. Memory-related bugs are among the most frightening because they often manifest themselves at a distance, in both time and space, from the source of the bug. Write the wrong data to the wrong location, and your program can run for hours before it finally fails in some distant part of the program. We conclude our discussion of virtual memory with a discussion of some of the common memory-related bugs.

10.11.1 Dereferencing Bad Pointers

As we learned in Section 10.7.2, there are large holes in the virtual address space of a process that are not mapped to any meaningful data. If we attempt to dereference a pointer into one of these holes, the operating system will terminate our program with a segmentation exception. Also, some areas of virtual memory are read-only. Attempting to write to one of these areas terminates the program with a protection exception.

A common example of dereferencing a bad pointer is the classic `scanf` bug. Suppose we want to use `scanf` to read an integer from `stdin` into a variable. The correct way to do this is to pass `scanf` a format string and the *address* of the variable:

```
scanf("%d", &val)
```

However, it is easy for new C programmers (and experienced ones too!) to pass the *contents* of `val` instead of its address:

```
scanf("%d", val)
```

In this case, `scanf` will interpret the contents of `val` as an address and attempt to write a word to that location. In the best case, the program terminates immediately with an exception. In the worst case, the contents of `val` correspond to some valid read/write area of virtual memory, and we overwrite memory, usually with disastrous and baffling consequences much later.

10.11.2 Reading Uninitialized Memory

While `.bss` memory locations (such as uninitialized global C variables) are always initialized to zeros by the loader, this is not true for heap memory. A common error is to assume that heap memory is initialized to zero:

```
1 /* return y = Ax */
2 int *matvec(int **A, int *x, int n)
3 {
4     int i, j;
5
6     int *y = (int *)Malloc(n * sizeof(int));
7
8     for (i = 0; i < n; i++)
9         for (j = 0; j < n; j++)
10            y[i] += A[i][j] * x[j];
11    return y;
12 }
```

In this example, the programmer has incorrectly assumed that vector `y` has been initialized to zero. A correct implementation would zero `y[i]` between lines 8 and 9, or use `calloc`.

10.11.3 Allowing Stack Buffer Overflows

As we saw in Section 3.13, a program has a *buffer overflow bug* if it writes to a target buffer on the stack without the size of the input string. For example, the following function has a buffer overflow bug because the `gets` function copies an arbitrary length string to the buffer. To fix this, we would need to the use the `fgets` function, which limits the size of the input string.

```
1 void bufoverflow()
2 {
3     char buf[64];
4
5     gets(buf); /* here is the stack buffer overflow bug */
6     return;
7 }
```

10.11.4 Assuming that Pointers and the Objects they Point to Are the Same Size

One common mistake is to assume that pointers to objects are the same size as the objects they point to:

```
1 /* Create an nxm array */
2 int **makeArray1(int n, int m)
3 {
4     int i;
5     int **A = (int **)Malloc(n * sizeof(int));
6
7     for (i = 0; i < n; i++)
8         A[i] = (int *)Malloc(m * sizeof(int));
9     return A;
10 }
```

The intent here is to create an array of n pointers, each of which points to an array of m `int`s. However, because the programmer has written `sizeof(int)` instead of `sizeof(int *)` in line 5, the code

actually creates an array of `int`s. This code will run fine on machines where `int`s and pointers to `int`s are the same size.

But if we run this code on a machine like the Alpha, where a pointer is larger than an `int`, then the loop in lines 7 and 8 will write past the end of the `A` array. Since one of these words will likely be the boundary tag footer of the allocated block, we may not discover the error until we free the block much later in the program, at which point the coalescing code in the allocator will fail dramatically and for no apparent reason. This is an insidious example of the kind of "action at a distance" that is so typical of memory-related programming bugs.

10.11.5 Making Off-by-one Errors

Off-by-one errors are another common source of overwriting bugs:

```
1 /* Create an nxm array */
2 int **makeArray2(int n, int m)
3 {
4     int i;
5     int **A = (int **)Malloc(n * sizeof(int));
6
7     for (i = 0; i <= n; i++)
8         A[i] = (int *)Malloc(m * sizeof(int));
9     return A;
10 }
```

This is another version of the program in the previous section. Here we have created an n-element array of pointers in line 5, but then tried to initialize $n + 1$ of its elements in lines 7 and 8, in the process overwriting some memory that follows the `A` array.

10.11.6 Referencing a Pointer Instead of the Object it Points to

If we are not careful about the precedence and associativity of C operators, then we incorrectly manipulate a pointer instead of the object it points to. For example, consider the following function, whose purpose is to remove the first item in a binary heap of `*size` items, and then reheapify the remaining `*size - 1` items.

```
1 int *binheapDelete(int **binheap, int *size)
2 {
3     int *packet = binheap[0];
4
5     binheap[0] = binheap[*size - 1];
6     *size--; /* this should be (*size)-- */
7     heapify(binheap, *size, 0);
8     return(packet);
9 }
```

In line 3, the intent is to decrement the integer value pointed to by the `size` pointer (e.g., `(*size)--`). However, because the unary `--` and `*` operators have the same precedence and associate from right to left, the code in line 6 actually decrements the pointer itself instead of the integer value that it points to. If we are lucky, the program will crash immediately; but more likely we will be left scratching our heads when the program produces an incorrect answer much later in its execution. The moral here is to use parentheses whenever in doubt about precedence and associativity. For example, in line 6 we could have clearly stated our intent by using the expression `(*size)--`.

10.11.7 Misunderstanding Pointer Arithmetic

Another common mistake is to forget that arithmetic operations on pointers are performed in units that are the size of the objects they point to, which are not necessarily bytes. For example, the intent of the following function is to scan an array of `int`s and return a pointer to the first occurrence of `val`.

```
1 int *search(int *p, int val)
2 {
3     while (*p && *p != val)
4         p += sizeof(int); /* should be p++ */
5     return p;
6 }
```

However, because line 4 increments the pointer by four (the number of bytes in an integer) each time through the loop, the function incorrectly scans every fourth integer in the array.

10.11.8 Referencing Non-existent Variables

Naive C programmers who do not understand the stack discipline will sometimes reference local variables that are no longer valid, as in the following example:

```
1 int *stackref ()
2 {
3     int val;
4
5     return &val;
6 }
```

This function returns a pointer (say `p`) to a local variable on the stack and then pops its stack frame. Although `p` still points to a valid memory address, it no longer points to a valid variable. When other functions are called later in the program, the memory will be reused for their stack frames. Later, if the program assigns some value to `*p`, then it might actually be modifying an entry in another function's stack frame, with potentially disastrous and baffling consequences.

10.11.9 Referencing Data in Free Heap Blocks

A similar error is to reference data in heap blocks that have already been freed. For example, consider the following example, which allocates an integer array `x` in line 6, prematurely frees block `x` in line 12, and then later references it in line 14.

```
1 int *heapref(int n, int m)
2 {
3     int i;
4     int *x, *y;
5
6     x = (int *)Malloc(n * sizeof(int));
7
8     /* ... */   /* other calls to malloc and free go here */
9
10    free(x);
11
12    y = (int *)Malloc(m * sizeof(int));
13    for (i = 0; i < m; i++)
14        y[i] = x[i]++;  /* oops! x[i] is a word in a free block */
15
16    return y;
17 }
```

Depending on the pattern of `malloc` and `free` calls that occur between lines 6 and 10, when the program references `x[i]` in line 14, the array `x` might be part of some other allocated heap block and have been overwritten. As with many memory-related bugs, the error will only become evident later in the program when we notice that the values in `y` are corrupted.

10.11.10 Introducing Memory Leaks

Memory leaks are slow, silent killers that occur when programmers inadvertently create garbage in the heap by forgetting to free allocated blocks. For example, the following function allocates a heap block `x` and then returns without freeing it.

```
1 void leak(int n)
2 {
3     int *x = (int *)Malloc(n * sizeof(int));
4
5     return;  /* x is garbage at this point */
6 }
```

If `foo` is called frequently, then the heap will gradually fill up with garbage, in the worst case consuming the entire virtual address space. Memory leaks are particularly serious for programs such as daemons and servers, which by definition never terminate.

10.12 Summary

In this chapter, we have looked at how virtual memory works, how it is used by the system for functions such as loading programs, mapping shared libraries, and providing processes with private protected address spaces. We have also looked at a myriad of ways that virtual memory can be used and misused by application programs.

A key lesson is that even though virtual memory is provided automatically by the system, it is a finite memory resource that must be managed wisely by the application. As we learned from our study of dynamic storage allocators, managing virtual memory resources can involve subtle time and space trade-offs. Another key lesson is that it is easy to make memory-related errors from C programs. Bad pointer values, freeing already free blocks, improper casting and pointer arithmetic, and overwriting heap structures are just a few of the many ways we can get in trouble. In fact, the nastiness of memory-related errors was an important motivation for Java, which tightly controls access to the virtual memory by eliminating the ability to take addresses of variables, and by taking complete control of the dynamic storage allocator.

Bibliographic Notes

Kilburn and his colleagues published the first description of virtual memory [39]. Architecture texts contain additional details about the hardware's role in virtual memory [31]. Operating systems texts contain additional information about the operating system's role [66, 79, 71].

Knuth wrote the classic work on storage allocation in 1968 [40]. Since that time there has been a tremendous amount of work in the area. Wilson, Johnstone, Neely, and Boles have written a beautiful survey and performance evaluation of explicit allocators [84]. The general comments in the text about the throughput and utilization of different allocator strategies are taken from this survey. Jones and Lins provide a comprehensive survey of garbage collection [34]. Kernighan and Ritchie [37] show the complete code for a simple allocator based on an explicit free list with a block size and successor pointer in each free block. The code is interesting in that it uses unions to eliminate a lot of the complicated pointer arithmetic, but at the expense of a linear-time (rather than constant-time) free operation.

Ben Zorn's *Dynamic Storage Allocation Repository* at `www.cs.colorado.edu/~zorn/DSA.html` is a handy resource. It includes sections on debugging tools for detecting memory-related errors and implementations of `malloc`/`free` and garbage collectors.

Homework Problems

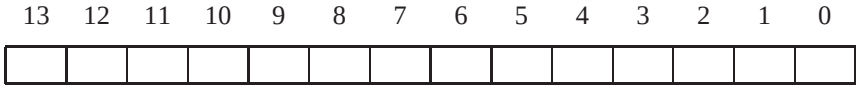
Homework Problem 10.11 [Category 1]:

In the following series of problems, you are to show how the example memory system in Section 10.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, the physical address, and the cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache Byte returned". If there is a page fault, enter "–" for "PPN" and leave parts C and D

blank.

Virtual address: `0x027c`

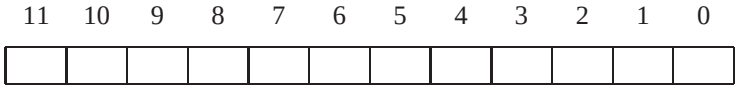
A. Virtual address format



B. Address translation

Parameter	Value
VPN	
TLB Index	
TLB Tag	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	

C. Physical address format



D. Physical memory reference

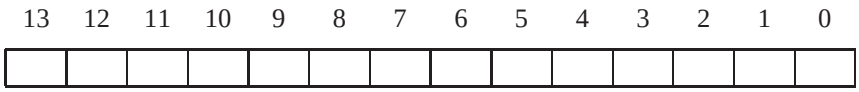
Parameter	Value
Byte offset	
Cache Index	
Cache Tag	
Cache Hit? (Y/N)	
Cache Byte returned	

Homework Problem 10.12 [Category 1]:

Repeat Problem 10.11 for the following address:

Virtual address: `0x03a9`

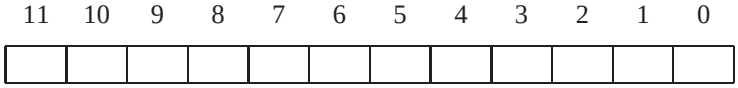
A. Virtual address format



B. Address translation

Parameter	Value
VPN	
TLB Index	
TLB Tag	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	

C. Physical address format



D. Physical memory reference

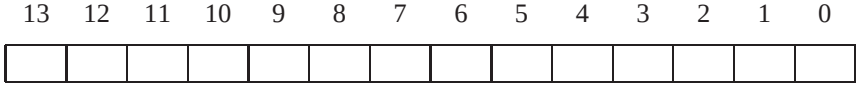
Parameter	Value
Byte offset	
Cache Index	
Cache Tag	
Cache Hit? (Y/N)	
Cache Byte returned	

Homework Problem 10.13 [Category 1]:

Repeat Problem 10.11 for the following address:

Virtual address: `0x0040`

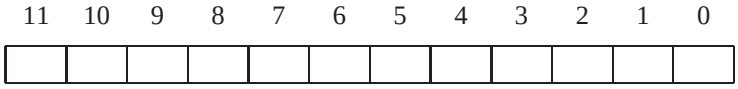
A. Virtual address format



B. Address translation

Parameter	Value
VPN	
TLB Index	
TLB Tag	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	
Cache Index	
Cache Tag	
Cache Hit? (Y/N)	
Cache Byte returned	

Homework Problem 10.14 [Category 2]:

Given an input file `hello.txt` that consists of the string `"Hello, world!\n"`, write a C program that uses `mmap` to change the contents of `hello.txt` to `"Jello, world!\n"`.

Homework Problem 10.15 [Category 1]:

Determine the block sizes and header values that would result from the following sequence of `malloc` requests. Assumptions: (1) The allocator maintains double-word alignment, and uses an implicit free list with the block format from Figure 10.37. (2) Block sizes are rounded up to the nearest multiple of eight bytes.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(3)`		
`malloc(11)`		
`malloc(20)`		
`malloc(21)`		

Homework Problem 10.16 [Category 1]:

Determine the minimum block size for each of the following combinations of alignment requirements and block formats. Assumptions: Explicit free list, four-byte `pred` and `succ` pointers in each free block, zero-sized payloads are not allowed, and headers and footers are stored in a four-byte words.

Alignment	Allocated block	Free block	Minimum block size (bytes)
Single-word	Header and footer	Header and footer	
Single-word	Header, but no footer	Header and footer	
Double-word	Header and footer	Header and footer	
Double-word	Header, but no footer	Header and footer	

Homework Problem 10.17 [Category 3]:

Develop a version of the allocator in Section 10.9.12 that performs a next-fit search instead of a first-fit search.

Homework Problem 10.18 [Category 3]:

The allocator in Section 10.9.12 requires both a header and a footer for each block in order to perform constant-time coalescing. Modify the allocator so that free blocks require a header and footer, but allocated blocks require only a header.

Homework Problem 10.19 [Category 1]:

You are given three groups of statements relating to memory management and garbage collection below. In each group, only one statement is true. Your task is to indicate the statement that is true.

1. (a) In a buddy system, up to 50% of the space can be wasted due to internal fragmentation.
   (b) The first-fit memory allocation algorithm is slower than the best-fit algorithm (on average).
   (c) Deallocation using boundary tags is fast only when the list of free blocks is ordered according to increasing memory addresses.
   (d) The buddy system suffers from internal fragmentation, but not from external fragmentation.

2. (a) Using the first-fit algorithm on a free list that is ordered according to decreasing block sizes results in low performance for allocations, but avoids external fragmentation.
   (b) For the best-fit method, the list of free blocks should be ordered according to increasing memory addresses.
   (c) The best-fit method chooses the largest free block into which the requested segment fits.
   (d) Using the first-fit algorithm on a free list that is ordered according to increasing block sizes is equivalent to using the best-fit algorithm.

3. Mark-and-sweep garbage collectors are called conservative if
   (a) they coalesce freed memory only when a memory request cannot be satisfied,
   (b) they treat everything that looks like a pointer as a pointer,
   (c) they perform garbage collection only when they run out of memory,
   (d) they do not free memory blocks forming a cyclic list.

Homework Problem 10.20 [Category 4]:

Write your own version of `malloc` and `free` and compare its running time and space utilization to the version of `malloc` provided in the standard C library.

Part III

Interaction and Communication Between Programs

Chapter 11

Concurrent Programming with Threads

A *thread* is a unit of execution, associated with a process, with its own thread ID, stack, stack pointer, program counter, condition codes, and general-purpose registers. Multiple threads associated with a process run concurrently in the context of that process, sharing its code, data, heap, shared libraries, signal handlers, and open files.

Programming with threads instead of conventional processes is increasingly popular because threads are less expensive (in terms of overhead) than processes and because they provide a trivial mechanism for sharing global data. For example, a high-performance Web server might assign a separate thread for each open connection to a Web browser, with each thread sharing a single in-memory cache of frequently requested Web pages.

Another important factor in the popularity of threads is the adoption of the standard *Pthreads* (Posix threads) interface for manipulating threads from C programs. The benefit of threads has been known for some time, but their use was hindered because each computer vendor developed its own incompatible threads package. As a result, threaded programs written for one platform would not run on other platforms. The adoption of Pthreads in 1995 has improved this situation immensely. Posix threads are available on most Unix systems.

Unfortunately, the ease with which threads share global data also makes them vulnerable to subtle and baffling errors. Bugs in threaded programs are especially scary because they are usually not easily repeatable. In this chapter, we will show you the basics of threaded programs, discuss some of the tricky ways that they can fail if you are not careful, and give you tips for avoiding these errors.

11.1 Basic Thread Concepts

To this point, we have worked with the traditional view of a process shown in Figure 11.1. In this view, a process consists of the code and data in the user's virtual memory, along with some state maintained by the kernel known as the *process context*. The code and data includes the program's text, data, runtime heap, shared libraries, and the stack. The process context can be partitioned into two different kinds of state: *program context* and *kernel context*. The program context resides in the processor, and includes the contents of the general-purpose registers, various condition codes registers, the stack pointer, and the program counter. The kernel context resides in kernel data structures, and consists of items such as the

process ID, the data structures that characterize the organization of the virtual memory, and information about open files, installed signal handlers, and the extent of the heap.

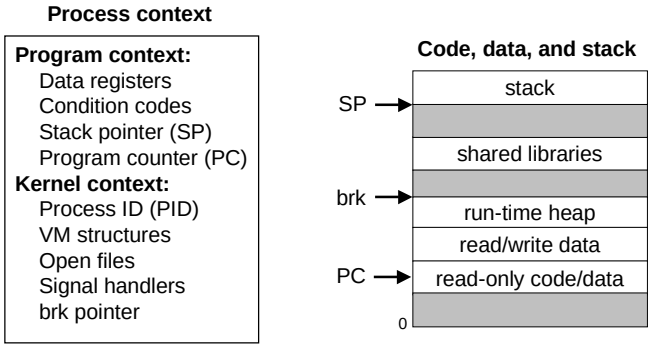


Figure 11.1: **Traditional view of a process.**

If we rearrange the items in Figure 11.1, then we get the alternative view of a process shown in Figure 11.2. Here, a process consists of a thread, which consists of a stack and the program context (which we will call the *thread context*), plus the kernel context and the program code and data (minus the stack, of course).

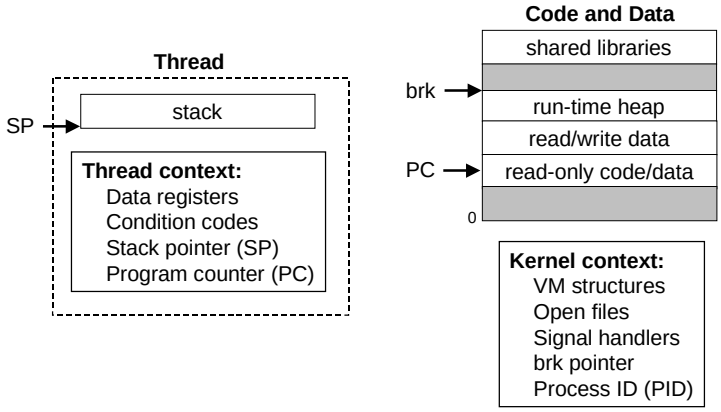


Figure 11.2: **Alternative view of a process.**

The interesting point about Figure 11.2 is that it treats the process as an execution unit with a very small amount of state that runs in the context of a much larger amount of state. Given this view, we can now extend our notion of process to include multiple threads that share the same code, data, and kernel context, as shown in Figure 11.3. Each thread associated with a process has its own stack, registers, condition codes, stack pointer, and program counter. Since there are now multiple threads, we will also add an integer *thread ID (TID)* to each thread context.

The execution model for multiple threads is similar in some ways to the execution model for multiple processes. Consider the example in Figure 11.4. Each process begins life as a single thread called the *main thread*. At some point, the main thread creates a *peer thread* and from this point in time the two threads run concurrently (i.e., their logical flows overlap in time). Eventually, control passes to the peer thread via a

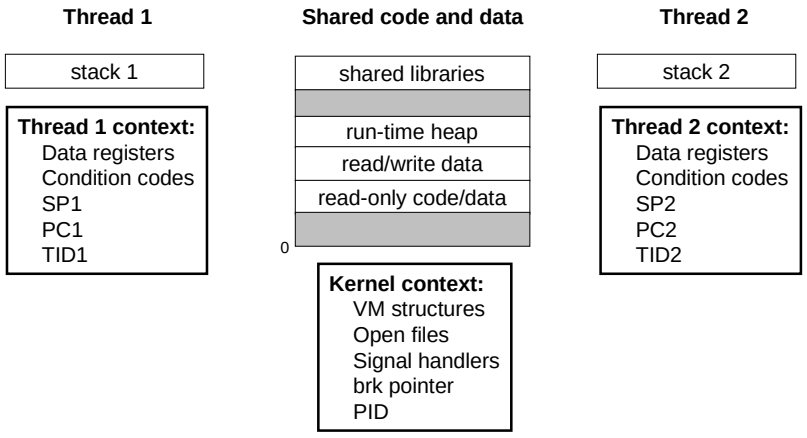


Figure 11.3: **Associating multiple threads with a process.**

context switch, because the main thread executes a slow system call such as `read` or `sleep`, or because it is interrupted by the system's interval timer. The peer thread executes for awhile before control passes back to the main thread, and so on.

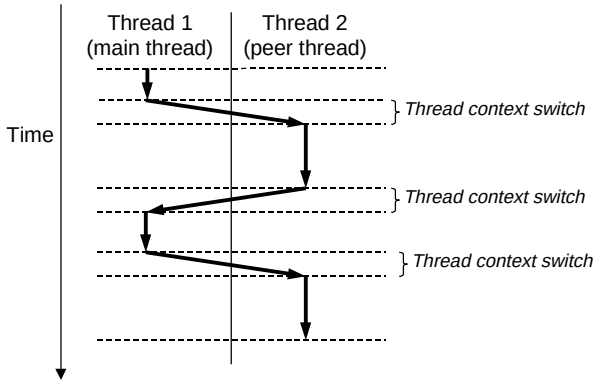


Figure 11.4: **Concurrent thread execution.**

Thread execution differs from processes in some important ways. Because a thread context is much smaller than a process context, a thread context switch is faster than a process context switch. Another difference is that threads, unlike processes, are not organized in a rigid parent-child hierarchy. The threads associated with a process form a pool of peers, independent of which threads were created by which other threads. The main thread is distinguished from other threads only in the sense that it is always the first thread to run in the process. The main impact of this notion of a pool of peers is that a thread can kill any of its peers, or wait for any of its peers to terminate. Further, each peer can read and write the same shared data.

11.2 Thread Control

Pthreads defines about 60 functions that allow C programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state. However, most threaded programs use only a small subset of the functions defined in the interface.

Figure 11.5 shows a simple Pthreads program called `hello.c`. In this program, the main thread creates a peer thread and then waits for it to terminate. The peer thread prints "`hello, world!\n`" and terminates. When the main thread detects that the peer thread has terminated, it terminates itself (and the entire process) by calling `exit`.

code/threads/hello.c

```
1 #include "csapp.h"
2
3 void *thread(void *vargp);
4
5 int main()
6 {
7     pthread_t tid;
8
9     Pthread_create(&tid, NULL, thread, NULL);
10    Pthread_join(tid, NULL);
11    exit(0);
12 }
13
14 /* thread routine */
15 void *thread(void *vargp)
16 {
17    printf("Hello, world!\n");
18    return NULL;
19 }
```

code/threads/hello.c

Figure 11.5: `hello.c`**: The Pthreads "hello, world" program.**

This is the first threaded program we have seen, so let's dissect it carefully. Line 3 is the prototype for the thread routine `thread`. The Pthreads interface mandates that each thread routine has a single `(void *)` input argument and returns a single `(void *)` output value. If you want to pass multiple arguments to a thread routine, then you can put the arguments into a structure and pass a pointer to the structure. Similarly, if you want the thread routine to return multiple arguments, you can return a pointer to a structure.

Line 5 marks the beginning of the `main` routine, which runs in the context of the main thread. In line 7, the main routine declares a single local variable `tid`, which will be used to store the thread ID of the peer thread. In line 9, the main thread creates a new peer thread by calling the `pthread_create` function.[1]

When the call to `pthread_create` returns, the main thread and the newly created thread are running

[1] We are actually calling an error-handling wrapper, which were introduced in Section 8.3 and described in detail in Appendix A.

concurrently, and `tid` contains the ID of the new thread. In line 10, the main thread waits for the newly created thread to terminate. Finally, in line 11, the main thread terminates itself and the entire process by calling `exit`.

Lines 15–19 define the thread routine, which in this case simply prints a string then terminates by executing the `return` statement in line 18.

11.2.1 Creating Threads

Threads create other threads by calling the `pthread_create` function.

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
```
returns: 0 if OK, non-zero on error

The `pthread_create` function creates a new thread and runs the *thread routine* `f` in the context of the new thread and with an input argument of `arg`. The `attr` argument can be used to change the default attributes of the newly created thread. However, changing these attributes is beyond our scope, and in our examples, we will always call `pthread_create` with a NULL `attr` argument.

When `pthread_create` returns, argument `tid` contains the ID of the newly created thread. The new thread can determine its own thread ID by calling the `pthread_self` function.

```
#include <pthread.h>

pthread_t pthread_self(void);
```
returns: thread ID of caller

11.2.2 Terminating Threads

A thread terminates in one of the following ways:

- The thread terminates *implicitly* when its top-level thread routine returns.
- The thread terminates *explicitly* by calling the `pthread_exit` function, which returns a pointer to the return value `thread_return`. If the main thread calls `pthread_exit`, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value of `thread_return`.

```
#include <pthread.h>

int pthread_exit(void *thread_return);
```
returns: 0 if OK, non-zero on error

- Some peer thread calls the Unix `exit` function, which terminates the process and all threads associated with the process.
- Another peer thread terminates the current thread by calling the `pthread_cancel` function with the ID of the current thread.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```
returns: 0 if OK, non-zero on error

11.2.3 Reaping Terminated Threads

Threads wait for other threads to terminate by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```
returns: 0 if OK, non-zero on error

The `pthread_join` function blocks until thread `tid` terminates, assigns the `(void *)` pointer returned by the thread routine to the location pointed to by `thread_return`, and then *reaps* any memory resources held by the terminated thread.

Notice that, unlike the Unix `wait` function, the `pthread_join` function can only wait for a specific thread to terminate. There is no way to instruct `pthread_wait` to wait for an arbitrary thread to terminate. This can complicate our code by forcing us to use other less intuitive mechanisms to detect process termination. Indeed some have argued convincingly that this represents a bug in the specification [77].

11.2.4 Detaching Threads

At any point in time, a thread is *joinable* or *detached*. A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread. In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.

By default, threads are created joinable. In order to avoid memory leaks, each joinable thread should either be explicitly reaped by another thread, or detached by a call to the `pthread_detach` function.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```
returns: 0 if OK, non-zero on error

The `pthread_detach` function detaches the joinable thread `tid`. Threads can detach themselves by calling `pthread_detach` with an argument of `pthread_self()`.

Even though some of our examples will use joinable threads, there are good reasons to use detached threads in real programs. For example, a high-performance Web server might create a new peer thread each time it receives a connection request from a Web browser. Since each connection is handled independently by a separate thread, it is unnecessary and indeed undesirable for the server to explicitly wait for each peer thread to terminate. In this case, each peer thread should detach itself before it begins processing the request so that its memory resources can be reclaimed after it terminates.

Practice Problem 11.1:

A. The program in Figure 11.6 has a bug. The thread is supposed to sleep for one second and then print a string. However, when we run it, nothing prints. Why?

B. You can fix this bug by replacing the `exit` function in line 9 with one of two different Pthreads function calls. Which ones?

code/threads/hellobug.c

```
1 #include "csapp.h"
2 void *thread(void *vargp);
3
4 int main()
5 {
6     pthread_t tid;
7
8     Pthread_create(&tid, NULL, thread, NULL);
9     exit(0);
10 }
11
12 /* thread routine */
13 void *thread(void *vargp)
14 {
15     Sleep(1);
16     printf("Hello, world!\n");
17     return NULL;
18 }
```

code/threads/hellobug.c

Figure 11.6: **Buggy program for Problem 11.1.**

11.3 Shared Variables in Threaded Programs

From a programmer's perspective, one of the attractive aspects of threads is the ease with which multiple threads can share the same program variables. However, in order to use threads correctly, we must have a clear understanding of what we mean by sharing and how it works.

There are some basic questions to work through in order to understand whether a variable in a C program is shared or not: (1) What is the underlying memory model for threads? (2) Given this model, how are instances of the variable mapped to memory? (3) And finally, how many threads reference each of these instances? The variable is *shared* if and only if multiple threads reference some instance of the variable.

To keep our discussion of sharing concrete, we will use the program in Figure 11.7 as a running example. Although somewhat contrived, it is nonetheless useful to study because it illustrates a number of subtle points about sharing. The example program consists of a main thread that creates two peer threads. The main thread passes a unique ID to each peer thread, which uses the id to print a personalized message, along with a count of the total number of times that the thread routine has been invoked. Here is the output when we run it on our system:

```
unix> ./sharing
[0]: Hello from foo (cnt=1)
[1]: Hello from bar (cnt=2)
```

11.3.1 Threads Memory Model

A pool of concurrent threads runs in the context of a process. Each thread has its own separate thread context, which includes a thread ID, stack, stack pointer, program counter, condition codes, and general-purpose register values. Each thread shares the rest of the process context with the other threads. This includes the entire user virtual address space, which consists of read-only text (code), read/write data, the heap, and any shared library code and data areas. The threads also share the same set of open files and the same set of installed signal handlers.

In an operational sense, it is impossible for one thread to read or write the register values of another thread. On the other hand, any thread can access any location in the shared virtual memory. If some thread modifies a memory location, then every other thread will eventually see the change if it reads that location. Thus, registers are never shared, while virtual memory is always shared.

The memory model for the separate thread stacks is not as clean. These stacks are contained in the stack area of the virtual address space, and are *usually* accessed independently by their respective threads. We say *usually* rather than *always*, because different thread stacks are not protected from other threads. So if a thread somehow manages to acquire a pointer to another thread's stack, then it can read and write any part of that stack. Our example program shows an example of this in line 29, where the peer threads reference the contents of the main thread's stack indirectly through the global `ptr` variable.

11.3.2 Mapping Variables to Memory

C variables in threaded programs are mapped to virtual memory according to their storage classes.

code/threads/sharing.c

```
1 #include "csapp.h"
2 #define N 2
3
4 char **ptr;  /* global variable */
5
6 void *thread(void *vargp);
7
8 int main()
9 {
10     int i;
11     pthread_t tid;
12     char *msgs[N] = {
13         "Hello from foo",
14         "Hello from bar"
15     };
16
17     ptr = msgs;
18
19     for (i = 0; i < N; i++)
20         Pthread_create(&tid, NULL, thread, (void *)i);
21     Pthread_exit(NULL);
22 }
23
24 void *thread(void *vargp)
25 {
26     int myid = (int)vargp;
27     static int cnt = 0;
28
29     printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
30 }
```

code/threads/sharing.c

Figure 11.7: **Example program that illustrates different aspects of sharing.**

- *Global variables.* A *global variable* is any variable declared outside of a function. At run-time, the read/write area of virtual memory contains exactly one instance of each global variable that can be referenced by any thread.

  For example, the global `ptr` variable in line 4 has one run-time instance in the read/write area of virtual memory. When there is only one instance of a variable, we will denote the instance by simply using the variable name, in this case `ptr`.

- *Local automatic variables.* A *local automatic variable* is one that is declared inside a function without the `static` attribute. At run-time, each thread's stack contains its own instances of any local automatic variables. This is true even if multiple threads execute the same thread routine.

  For example, there is one instance of the local variable `tid`, and it resides on the stack of the main thread. We will denote this instance as `tid.m`. As another example, there are two instances of the local variable `myid`, one instance on the stack of peer thread 0, and the other on the stack of peer thread 1. We will denote these instances as `myid.p0` and `myid.p1` respectively.

- *Local static variables.* A *local static variable* is one that is declared inside a function with the `static` attribute. As with global variables, the read/write area of virtual memory contains exactly one instance of each local static variable declared in a program.

  For example, even though each peer thread in our example program declares `cnt` in line 27, at run-time there is only one instance of `cnt` residing in the read/write area of virtual memory. Each peer thread reads and writes this instance.

11.3.3 Shared Variables

A variable v is shared if and only if one of its instances is referenced by more than one thread. For example, variable `cnt` in our example program is shared because it has only one run-time instance, and this instance is referenced by both peer threads. On the other hand, `myid` is not shared because each of its two instances is referenced by exactly one thread. However, it is important to realize that local automatic variables such as `msgs` can also be shared.

Practice Problem 11.2:

A. Using the analysis from Section 11.3, fill each entry in the following table with "Yes" or "No" for the example program in Figure 11.7. In the first column, the notation $v.t$ denotes an instance of variable v residing on the local stack for thread t, where t is either `m` (main thread), `p0` (peer thread 0), or `p1` (peer thread 1).

Variable instance	Referenced by main thread?	Referenced by peer thread 0 ?	Referenced by peer thread 1?
`ptr`			
`cnt`			
`i.m`			
`msgs.m`			
`myid.p0`			
`myid.p1`			

B. Given the analysis in Part A, which of the variables `ptr`, `cnt`, `i`, `msgs`, and `myid` are shared?

11.4 Synchronizing Threads with Semaphores

Shared variables can be convenient, but they introduce the possibility of a new class of *synchronization errors* that we have not encountered yet. Consider the `badcnt.c` program in Figure 11.8 that creates two threads, each of which increments a shared counter variable called `cnt`.

Since each thread increments the counter NITERS times, we might expect its final value to be $2 \times$ NITERS. However, when we run `badcnt.c` on our system, we not only get wrong answers, we get different answers each time!

```
unix> ./badcnt
BOOM! ctr=198841183

unix>  ./badcnt
BOOM! ctr=198261801

unix>  ./badcnt
BOOM! ctr=198269672
```

So what went wrong? To understand the problem clearly, we need to study the assembly code for the counter loop, as shown in Figure 11.9. We will find it helpful to partition the loop code for thread i into five parts:

- H_i: The block of instructions at the head of the loop.
- L_i: The instruction that loads the shared variable `cnt` into register `%eax`$_i$, where `%eax`$_i$ denotes the value of register `%eax` in thread i.
- U_i: The instruction that updates (increments) `%eax`$_i$.
- S_i: The instruction that stores the updated value of `%eax`$_i$ back to the shared variable `cnt`.
- T_i: The block of instructions at the tail of the loop.

Notice that the head and tail manipulate only local stack variables, while L_i, U_i, and S_i manipulate the contents of the shared counter variable.

11.4.1 Sequential Consistency

When the two peer threads in `badcnt.c` run concurrently on a single CPU, the instructions are completed one after the other in some order. Thus, each concurrent execution defines some total ordering (or interleaving) of the instructions in the two threads. When we reason about concurrent execution, the *only* assumption we can make about the total ordering of instructions is that it is *sequentially consistent*. That is, instructions

code/threads/badcnt.c

```
1  #include "csapp.h"
2
3  #define NITERS 100000000
4
5  void *count(void *arg);
6
7  /* shared variable */
8  unsigned int cnt = 0;
9
10 int main()
11 {
12     pthread_t tid1, tid2;
13
14     Pthread_create(&tid1, NULL, count, NULL);
15     Pthread_create(&tid2, NULL, count, NULL);
16
17     Pthread_join(tid1, NULL);
18     Pthread_join(tid2, NULL);
19
20     if (cnt != (unsigned)NITERS*2)
21         printf("BOOM! cnt=%d\n", cnt);
22     else
23         printf("OK cnt=%d\n", cnt);
24     exit(0);
25 }
26
27 /* thread routine */
28 void *count(void *arg)
29 {
30     int i;
31
32     for (i=0; i<NITERS; i++)
33         cnt++;
34     return NULL;
35 }
```

code/threads/badcnt.c

Figure 11.8: `badcnt.c`: **An improperly synchronized counter program.**

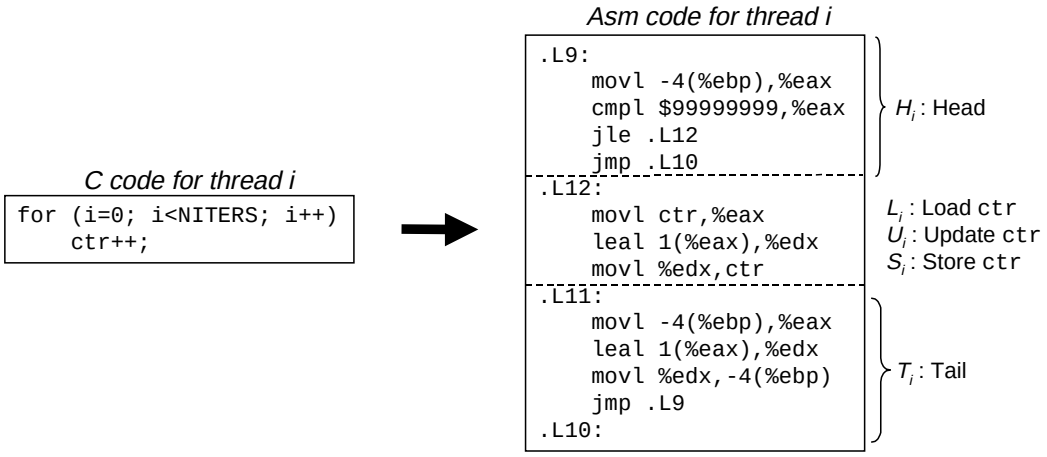


Figure 11.9: **IA32 assembly code for the counter loop in** `badcnt.c`**.**

can be interleaved in any order, so long as the instructions for each thread execute in program order. For example, the ordering

$$H_1, H_2, L_1, L_2, U_1, U_2, S_1, S_2, T_1, T_2$$

is sequentially consistent, while the ordering

$$H_1, H_2, U_1, L_2, L_1, U_2, S_1, S_2, T_1, T_2$$

is not sequentially consistent because U_1 executes before L_1. Unfortunately not all sequentially consistent orderings are created equal. Some will produce correct results, but others will not, and there is no way for us to predict whether the operating system will choose a correct ordering for our threads. For example, Figure 11.10(a) shows the step-by-step operation of a correct instruction ordering. After each thread has updated the shared variable `cnt`, its value in memory is 2, which is the expected result.

Step	Thread	Instr	%eax$_1$	%eax$_2$	cnt
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	1	U_1	1	–	0
4	1	S_1	1	–	1
5	2	H_2	–	–	1
6	2	L_2	–	1	1
7	2	U_2	–	2	1
8	2	S_2	–	2	2
9	2	T_2	–	2	2
10	1	T_1	1	–	2

(a) Correct ordering

Step	Thread	Instr	%eax$_1$	%eax$_2$	cnt
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	1	U_1	1	–	0
4	2	H_2	–	–	0
5	2	L_2	–	0	0
6	1	S_1	1	–	1
7	1	T_1	1	–	1
8	2	U_2	–	1	1
9	2	S_2	–	1	1
10	2	T_2	–	1	1

(b) Incorrect ordering

Figure 11.10: **Sequentially-consistent orderings for the first loop iteration in** `badcnt.c`**.**

On the other hand, the ordering in Figure 11.10(b) produces an incorrect value for `cnt`. The problem occurs because thread 2 loads `cnt` in step 5, after thread 1 loads `cnt` in step 2, but before thread 1 stores its updated value in step 6. Thus each thread ends up storing an updated counter value of 1.

We can clarify this idea of correct and incorrect instruction orderings with the help of a formalism known as a *progress graph*, which we introduce in the next section.

Practice Problem 11.3:

Which of the following instruction orderings for `badcnt.c` are sequentially consistent?

A. $H_1, H_2, L_1, L_2, U_1, U_2, S_2, S_1, T_2, T_1$.

B. $H_1, H_2, L_2, U_2, S_2, U_1, T_2, L_1, S_1, T_1$.

C. $H_2, L_2, H_1, L_1, U_1, S_1, U_2, S_2, T_2, T_1$.

D. $H_2, H_1, L_2, L_1, S_2, U_1, U_2, S_1, T_2, T_1$.

Practice Problem 11.4:

Complete the table for the following sequentially consistent ordering of `badcnt.c`.

Step	Thread	Instr	`%eax`$_1$	`%eax`$_2$	`cnt`
1	1	H_1	–	–	0
2	1	L_1			
3	2	H_2			
4	2	L_2			
5	2	U_2			
6	2	S_2			
7	1	U_1			
8	1	S_1			
9	1	T_1			
10	2	T_2			

Does this ordering result in a correct value for `cnt`?

11.4.2 Progress Graphs

A *progress graph* models the execution of n concurrent threads as a trajectory through an n-dimensional Cartesian space. Each axis k corresponds to the progress of thread k. Each point $(I_1, I_2, \ldots, I_n)$ represents the state where thread k, ($k = 1, \ldots, n$) has completed instruction I_k. The origin of the graph corresponds to the *initial state* where none of the threads has yet completed an instruction.

Figure 11.11 shows the 2-dimensional progress graph for the first loop iteration of the `badcnt.c` program. The horizontal axis corresponds to thread 1, the vertical axis to thread 2. Point (L_1, S_2) corresponds to the state where thread 1 has completed L_1 and thread 2 has completed S_2.

A progress graph models instruction execution as a *transition* from one state to another. A transition is represented as a directed edge from one point to an adjacent point. Figure 11.12 shows the legal transitions

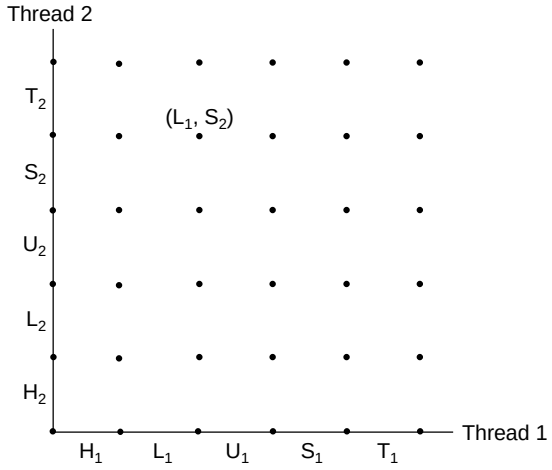


Figure 11.11: **Progress graph for the first loop iteration of** `badcnt.c`**.**

in a 2-dimensional progress graph. For the single-processor systems that we are concerned about, where instructions complete one at a time in sequentially-consistent order, legal transitions move to the right (an instruction in thread 1 completes) or up (an instruction in thread 2 completes). Programs never run backwards, so transitions that move down or to the left are not legal.

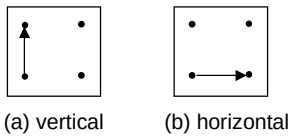


Figure 11.12: **Legal transitions in a progress graph.**

The execution history of a program is modeled as a *trajectory*, or sequence of transitions, through the state space. Figure 11.13 shows the trajectory that corresponds to the instruction ordering

$$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2.$$

For thread i, the instructions (L_i, U_i, S_i) that manipulate the contents of the shared variable `cnt` constitute a *critical section* (with respect to shared variable `cnt`) that should not be interleaved with the critical section of the other thread. The intersection of the two critical sections defines a region of the state space known as an *unsafe region*. Figure 11.14 shows the unsafe region for the variable `cnt`. Notice that the unsafe region abuts, but does not include, the states along its perimeter. For example, states (H_1, H_2) and (S_1, U_2) abut the unsafe region, but are not a part of it.

A trajectory that skirts the unsafe region is known as a *safe trajectory*. Conversely, a trajectory that touches any part of the unsafe region is an *unsafe trajectory*. Figure 11.15 shows examples of safe and unsafe trajectories through the state space of our example `badcnt.c` program. The upper trajectory skirts the unsafe region along its left and top sides, and thus is safe. The lower trajectory crosses the unsafe region with one of its diagonal transitions, and thus is unsafe.

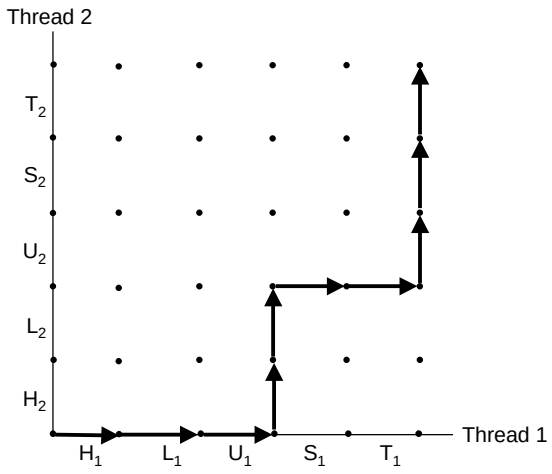


Figure 11.13: **An example trajectory.**

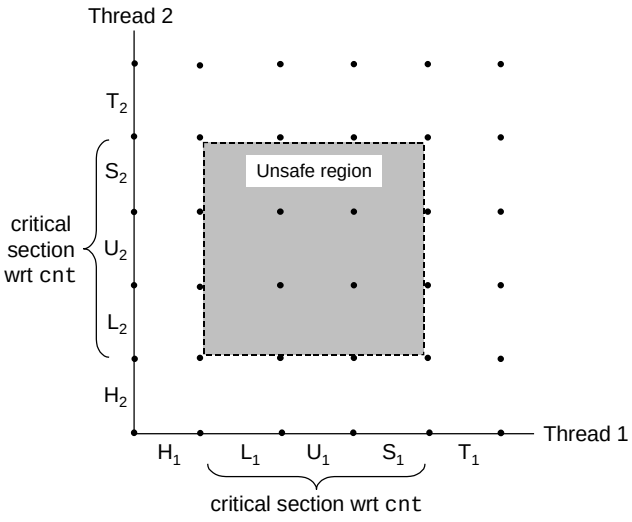


Figure 11.14: **Critical sections and unsafe regions.**

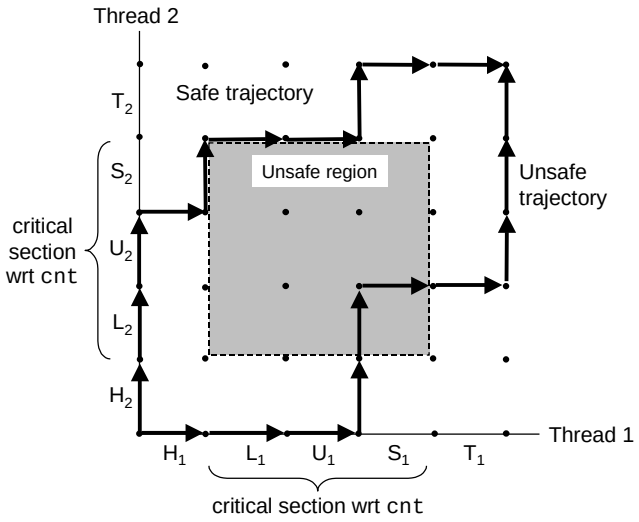


Figure 11.15: **Safe and unsafe trajectories.**

Any safe trajectory will correctly update the shared counter. Conversely, any unsafe trajectory will produce either a predictably wrong result or a result that cannot be predicted. The latter case arises when the trajectory crosses the lower right-hand corner of the region with a diagonal transition from state (U_1, H_2) to (S_1, L_2). If thread 1 stores its updated value of the counter variable before thread 2 loads it, then the result will be correct, otherwise it will be incorrect. Since we cannot predict the ordering of load and store operations, we consider this case, as well as the symmetric case where the trajectory crosses the upper left-hand corner of the unsafe region, to be incorrect.

The bottom line is that in order to guarantee correct execution of our example threaded program — and indeed any concurrent program that shares global data structures – we must somehow *synchronize* the threads so that they always have a safe trajectory. Dijkstra proposed a solution to this problem in a classic paper that introduced the fundamental idea of a *semaphore*.

11.4.3 Protecting Shared Variables with Semaphores

A *semaphore*, s, is a global variable with a nonnegative integer value that can only be manipulated by two special operations, called P and V:

- $P(s)$: `while (s <= 0); s--;`
- $V(s)$: `s++;`

The names P and V come from the Dutch *Proberen* (to test) and *Verhogen* (to increment). The P operation waits for the semaphore s to become nonzero, and then decrements it. The V operation increments s. The test and decrement operations in P occur indivisibly, in the sense that once the predicate $s <= 0$ becomes false, the decrement occurs without interruption. The increment operation in V also occurs indivisibly, in that it loads, increments, and stores the semaphore without interruption.

The definitions of P and V ensure that a running program can never enter a state where a properly initialized semaphore has a negative value. This property, known as the *semaphore invariant*, provides a powerful tool for controlling the trajectories of concurrent programs so that they avoid unsafe regions.

The basic idea is to associate a semaphore s, initially 1, with each shared variable (or related set of shared variables) and then surround the corresponding critical section with $P(s)$ and $V(s)$ operations.[2]

For example, the progress graph in Figure 11.16 shows how we would use semaphores to properly synchronize our example counter program. In the figure, each state is labeled with the value of semaphore s in that state. The crucial idea is that this combination of P and V operations creates a collection of states, called a *forbidden region*, where $s < 0$. Because of the semaphore invariant, no feasible trajectory can include one of the states in the forbidden region. And since the forbidden region completely encloses the unsafe region, no feasible trajectory can touch any part of the unsafe region. Thus, every feasible trajectory is safe, and regardless of the ordering of the instructions at runtime, the program correctly increments the counter.

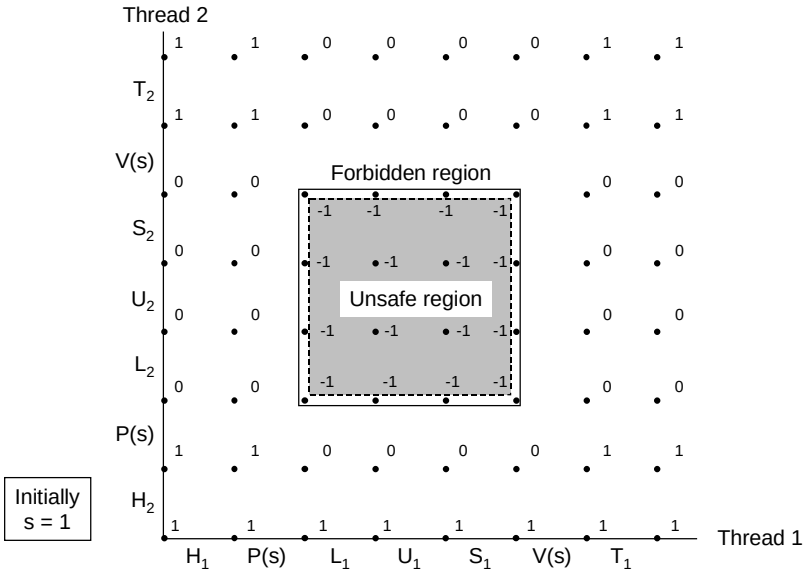


Figure 11.16: **Safe sharing with semaphores.** The states where $s < 0$ define a *forbidden region* that surrounds the unsafe region.

11.4.4 Posix Semaphores

The Posix standard defines a number of functions for manipulating semaphores. This section describes the three basic functions, `sem_init`, `sem_wait` (P), and `sem_post` (V).

A program initializes a semaphore by calling the `sem_init` function.

[2] A semaphore that is used in this way to protect shared variables is called a *binary semaphore* because its value is always 0 or 1.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```
returns: 0 if OK, -1 on error

The `sem_init` function initializes semaphore `sem` to `value`. Each semaphore must be initialized before it can be used. If `sem` is being used to synchronize concurrent threads associated with the same process, then `pshared` is zero. If `sem` is being used to synchronize concurrent processes (not discussed here), then `pshared` is nonzero. We use Posix semaphores only in the context of concurrent threads, so `pshared` is 0 in all of our examples.

Programs perform P and V operations on semaphore `sem` by calling the `sem_wait` and `sem_post` functions, respectively.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```
returns: 0 if OK, -1 on error

Figure 11.17 shows a version of the threaded counter program from Figure 11.8, called `goodcnt.c`, that uses semaphore operations to properly synchronize access to the shared counter variable. The code follows directly from Figure 11.16, so there are just a few aspects of it to point out:

- First, in a convention dating back to Dijkstra's original semaphore paper, a binary semaphore used for safe sharing is often called a *mutex* because it provides each thread with *mutually exclusive access* to the shared data. We have followed this convention in our code.
- Second, the `Sem_init`, `P`, and `V` functions are Unix-style error-handling wrappers for the `sem_init`, `sem_wait`, and `sem_post` functions, respectively.

11.4.5 Signaling With Semaphores

We saw in the previous section how semaphores can be used for sharing. But semaphores can also be used for *signaling*. In this scenario, a thread uses a semaphore operation to notify another thread when some condition in the program state becomes true. A classic example is the `producer-consumer` interaction shown in Figure 11.18. A producer thread and a consumer thread share a buffer with n slots. The producer thread repeatedly produces new items and inserts them in the buffer. The consumer thread repeatedly removes items from the buffer and then consumes them. Other variants allow different combinations of single and multiple producers and consumers.

Producer-consumer interactions are common in real systems. For example, in a multimedia system, the producer might encode video frames while the consumer decodes and renders them on the screen. The purpose of the buffer is to reduce jitter in the video stream caused by data-dependent differences in the

code/threads/goodcnt.c

```
 1 #include "csapp.h"
 2
 3 #define NITERS 10000000
 4
 5 void *count(void *arg);
 6
 7 /* shared variables */
 8 unsigned int cnt; /* counter */
 9 sem_t sem;        /* semaphore */
10
11 int main()
12 {
13     pthread_t tid1, tid2;
14
15     Sem_init(&sem, 0, 1);
16
17     Pthread_create(&tid1, NULL, count, NULL);
18     Pthread_create(&tid2, NULL, count, NULL);
19
20     Pthread_join(tid1, NULL);
21     Pthread_join(tid2, NULL);
22
23     if (cnt != (unsigned)NITERS*2)
24         printf("BOOM! cnt=%d\n", cnt);
25     else
26         printf("OK cnt=%d\n", cnt);
27     exit(0);
28 }
29
30 /* thread routine */
31 void *count(void *arg)
32 {
33     int i;
34
35     for (i=0; i<NITERS; i++) {
36         P(&sem);
37         cnt++;
38         V(&sem);
39     }
40     return NULL;
41 }
```

code/threads/goodcnt.c

Figure 11.17: `goodcnt.c`: **A properly synchronized version of** `badcnt.c`**.**

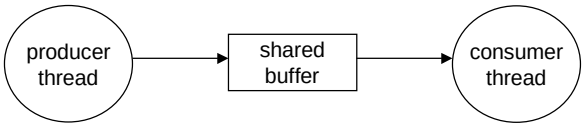


Figure 11.18: **Producer-consumer model.**

encoding and decoding times for individual frames. The buffer provides a reservoir of slots to the producer and a reservoir of encoded frames to the consumer. Another common example is the design of graphical user interfaces. The producer detects mouse and keyboard events and inserts them in the buffer. The consumer removes the events from the buffer in some priority-based manner and paints the screen.

Figure 11.19 outlines how we would use Posix semaphores to synchronize the producer and consumer threads in a simple producer-consumer program where the buffer can hold at most one item.

We use two semaphores, `empty` and `full` to characterize the state of the buffer. The `empty` semaphore indicates that the buffer contains no valid items. It is initialized to the initial number of available empty buffer slots (1). The `full` semaphore indicates that the buffer contains a valid item. It is initialized to the initial number of valid items (0).

The producer thread produces an item (in this case a simple integer), then waits for the buffer to be empty with a P operation on semaphore `empty`. After the producer writes the item to the buffer, it informs the consumer that there is now a valid item with a V operation on `full`. Conversely, the consumer thread waits for a valid item with a P operation on `full`. After reading the item, it signals that the buffer is empty with a V operation on `empty`.

The impact at run-time is that the producer and consumer ping-pong back and forth, as shown in Figure 11.21.

11.5 Synchronizing Threads with Mutex and Condition Variables

As an alternative to P and V operations on semaphores, Pthreads provides a family of synchronization operations on *mutex* and *condition variables*. In general, we prefer semaphores over their Pthreads counterparts because they are more elegant and simpler to reason about. However, there are some useful synchronization patterns, such as timeout waiting, that are impossible to implement with semaphores. Thus, it is worthwhile to have some facility with the Pthreads operations.

In the previous section, we learned that semaphores can be used for both sharing *and* signaling. Pthreads, on the other hand, provides one set of functions (based on mutex variables) for sharing, and another set (based on condition variables) for signaling.

11.5.1 Mutex Variables

A *mutex* is synchronization variable that is used like a binary semaphore to protect the access to shared variables. There are three basic operations defined on a mutex.

code/threads/prodcons.c

```
1 #include "csapp.h"
2
3 #define NITERS 5
4
5 void *producer(void *arg), *consumer(void *arg);
6
7 struct {
8   int buf;           /* shared variable */
9   sem_t full, empty; /* semaphores */
10 } shared;
11
12
13 int main()
14 {
15   pthread_t tid_producer, tid_consumer;
16
17   /* initialize the semaphores */
18   Sem_init(&shared.empty, 0, 1);
19   Sem_init(&shared.full,  0, 0);
20
21   /* create threads and wait for them to finish */
22   Pthread_create(&tid_producer, NULL, producer, NULL);
23   Pthread_create(&tid_consumer, NULL, consumer, NULL);
24   Pthread_join(tid_producer, NULL);
25   Pthread_join(tid_consumer, NULL);
26
27   exit(0);
28 }
```

code/threads/prodcons.c

Figure 11.19: **Producer-consumer program: Main routine.** One producer thread and one consumer thread manipulate a 1-item buffer. Initially, `empty == 1` and `full == 0`.

code/threads/prodcons.c

```
1  /* producer thread */
2  void *producer(void *arg)
3  {
4    int i, item;
5
6    for (i=0; i<NITERS; i++) {
7      /* produce item */
8      item = i;
9      printf("produced %d\n", item);
10
11     /* write item to buf */
12     P(&shared.empty);
13     shared.buf = item;
14     V(&shared.full);
15   }
16   return NULL;
17 }
18
19 /* consumer thread */
20 void *consumer(void *arg)
21 {
22   int i, item;
23
24   for (i=0; i<NITERS; i++) {
25     /* read item from buf */
26     P(&shared.full);
27     item = shared.buf;
28     V(&shared.empty);
29
30     /* consume item */
31     printf("consumed %d\n", item);
32   }
33   return NULL;
34 }
```

code/threads/prodcons.c

Figure 11.20: **Producer-consumer program: Producer and consumer threads.**

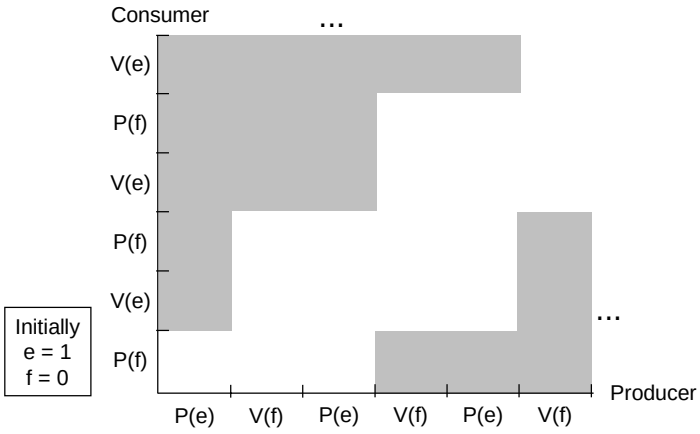


Figure 11.21: **Progress graph for** `prodcons.c`.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

return: 0 if OK, nonzero on error

A mutex must be initialized before it can be used, either at run-time by calling the `pthread_init` function, or at compile-time:

```
1     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

For our purposes, the `attr` argument in `pthread_init` will always be NULL and can be safely ignored.

The `pthread_mutex_lock` function performs a P operation and the `pthread_mutex_unlock` function performs a V operation. Completing the call to `pthread_mutex_lock` is referred to as *acquiring the mutex*, and completing the call to `pthread_mutex_unlock` is referred to as *releasing the mutex*. At any point in time, at most one thread can hold the mutex.

11.5.2 Condition Variables

Condition variables are synchronization variables that are used for signaling. Pthreads defines three basic operations on condition variables.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

return: 0 if OK, nonzero on error

A condition variable `cond` must be initialized before it is used, either by calling `pthread_cond_init` or at compile-time:

```
1     pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

For our purposes the `attr` argument will always be NULL and can be safely ignored.

A thread waits for some program condition associated with the condition variable `cond` to become true by calling `pthread_cond_wait`. In order to guarantee that the call to `pthread_cond_wait` is indivisible with respect to other instances of `pthread_cond_wait` and `pthread_cond_signal`, Pthreads requires that a mutex variable `mutex` be associated with the condition variable `cond`, and that a call to `pthread_mutex_wait` must always be protected by that mutex:

```
1    Pthread_mutex_lock(&mutex);
2    Pthread_cond_wait(&cond, &mutex);
3    Pthread_mutex_unlock(&mutex);
```

The call to `pthread_cond_wait` releases the mutex and suspends the current thread until `cond` becomes true. At this point, we say that the current thread is *waiting* on condition variable `cond`.

Later, someother thread indicates that the condition associated with `cond` has become true by making a call to the `pthread_cond_signal` function:

```
1    Pthread_cond_signal(&cond);
```

If there are any threads waiting on condition `cond`, then the call to `pthread_cond_signal` sends a *signal* that wakes up exactly one of them.

The thread that wakes up as a result of the signal reacquires the mutex and then returns from its call to `pthread_cond_wait`.

If no threads are currently waiting on condition `cond`, then nothing happens. Thus, like Unix signals, and unlike Posix semaphores, *Pthreads signals are not queued*, which makes them much harder to reason about than semaphore operations.

Aside: Pthreads signals vs. Unix signals
Pthreads signals are totally unrelated to the Unix signals that we learned about in Sectionsec:ecf:signals. Unix signals have been around since the early days of Unix. Pthreads is a more modern development dating from the mid 1990s. It is unfortunate that the Pthreads standards group decided to use the same term, but the terminology is fixed and we must accept it. In this chapter, we are only dealing with Pthreads signals. **End Aside.**

11.5.3 Barrier Synchronization

In general, we find that synchronizing with mutex and condition variables is more difficult and cumbersome than with semaphores. However, a barrier is an example of a synchronization pattern that can be expressed quite elegantly with operations on mutex and condition variables.

A *barrier* is a function, `void barrier(void)`, that returns to the caller only when every other thread has also called `barrier`. Barriers are essential in concurrent programs whose threads must progress at roughly the same rate. For example, we use threads in our research to implement parallel simulations of earthquakes. The duration of the earthquake (say 60 seconds) is broken up into thousands of tiny timesteps. Each thread runs on a separate processor and models the propagation of seismic waves through some chunk of the earth, first for timestep 1, then for timestep 2, and so on. In order to get consistent results, each thread must finish simulating timestep k before the others can begin simulating timestep $k + 1$. We guarantee this by placing a barrier between the execution of each timestep.

Our barrier implementation uses a signaling variant called `pthread_cond_broadcast` that wakes up *every* thread currently waiting on condition variable `cond`.

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

returns: 0 if OK, nonzero on error

Figure 11.22 shows the code for a simple barrier package based on mutex and condition variables.

The barrier package uses 4 global variables that are defined in lines 2–7. The variables are declared with the `static` attribute so that they are not visible to other object modules. `Cond` is a condition variable and `mutex` is its associated mutex variable. `Nthreads` records the number of threads involved in the barrier, and `barriercnt` keeps track of the number of threads that have called the `barrier` function.

The `barrier_init` function in lines 9–14 is called once by the main thread, before it creates any other threads.

The mutex that surrounds the body of the `barrier` function guarantees that it is executed indivisibly and in some total order by each thread. Thus, once the current thread has acquired the mutex in line 18, there are only two possibilities.

1. If the current thread is the last to enter the barrier, then it clears the barrier count for the next time (line 20), and wakes up all of the other threads (line 21). We know that all of the other threads are asleep, waiting on a signal, because the current thread is the last to enter the barrier.
2. If the current thread is not the last thread, then it goes to sleep and releases the mutex (line 24) so that other threads can enter the barrier.

11.5.4 Timeout Waiting

Sometimes when we write a concurrent program, we are only willing to wait a finite amount of time for a particular condition to become true. Since a P operation can block indefinitely, this kind of *timeout waiting* is not possible to implement with semaphore operations. However, Pthreads provides this capability, in the form of the `pthread_cond_timedwait` function.

code/threads/barrier.c

```
#include "csapp.h"

static pthread_mutex_t mutex;
static pthread_cond_t cond;

static int nthreads;
static int barriercnt = 0;

void barrier_init(int n)
{
    nthreads = n;
    Pthread_mutex_init(&mutex, NULL);
    Pthread_cond_init(&cond, NULL);
}

void barrier()
{
    Pthread_mutex_lock(&mutex);
    if (++barriercnt == nthreads) {
        barriercnt = 0;
        Pthread_cond_broadcast(&cond);
    }
    else
        Pthread_cond_wait(&cond, &mutex);
    Pthread_mutex_unlock(&mutex);
}
```

code/threads/barrier.c

Figure 11.22: `barrier.c`**: A simple barrier synchronization package.**

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           struct timespec *abstime);
                                                  returns: 0 if OK, ETIMEDOUT if timeout
```

The `pthread_cond_timedwait` function behaves like the `pthread_cond_wait` function, except that it returns with an error code of ETIMEDOUT once the value of the system clock exceeds the absolute time value in `abstime`. Figure 11.23 shows a handy routine that a thread can use to build the `abstime` argument each time it calls `pthread_cond_timedwait`:

code/threads/maketimeout.c

```
 1 #include "csapp.h"
 2
 3 struct timespec *maketimeout(struct timespec *tp, int secs)
 4 {
 5     struct timeval now;
 6
 7     gettimeofday(&now, NULL);
 8     tp->tv_sec = now.tv_sec + secs;
 9     tp->tv_nsec = now.tv_usec * 1000;
10     return tp;
11 }
```

code/threads/maketimeout.c

Figure 11.23: `maketimeout`**: Builds a timeout structure for** `pthread_cond_timedwait`**.**

For a simple example of timeout waiting in a threaded program, suppose we want to write a *beeping timebomb* that waits at most 5 seconds for the user to hit the return key, printing out "BEEP" every second. If the user doesn't hit the return key in time, then the program explodes by printing "BOOM!". Otherwise, it prints "Whew!" and exits. Figure 11.24 shows a threaded timebomb that is based on the `pthread_cond_timedwait` function.

The main timebomb thread locks the mutex and then creates a peer thread that calls `getchar`, which blocks the thread until the user hits the return key. When `getchar` returns, the peer thread signals the main thread that the user has hit the return key, and then terminates. Notice that since the main thread locked the mutex before creating the peer thread, the peer thread cannot acquire the mutex and signal the main thread until the main thread releases the mutex by calling `pthread_cond_timedwait`.

Meanwhile, after the main thread creates the peer thread, it waits up to one second for the peer thread to terminate. If `pthread_cond_timedwait` does not time out, then the main thread knows that the peer thread has terminated, so it prints "Whew!" and exits. Otherwise, it beeps and waits for another second. This continues until it has waited a total of 5 seconds, at which point the loop terminates, the main thread explodes by printing "Boom!", and then exits.

code/threads/timebomb.c

```
 1 #include "csapp.h"
 2
 3 #define TIMEOUT 5
 4
 5 void *thread(void *vargp);
 6 struct timespec *maketimeout(struct timespec *tp, int secs);
 7
 8 pthread_cond_t cond;
 9 pthread_mutex_t mutex;
10 pthread_t tid;
11
12 int main()
13 {
14     int i, rc;
15     struct timespec timeout;
16
17     Pthread_cond_init(&cond, NULL);
18     Pthread_mutex_init(&mutex, NULL);
19
20     Pthread_mutex_lock(&mutex);
21     Pthread_create(&tid, NULL, thread, NULL);
22     for (i=0; i<TIMEOUT; i++) {
23         printf("BEEP\n");
24         rc = pthread_cond_timedwait(&cond, &mutex,
25                                     maketimeout(&timeout, 1));
26         if (rc != ETIMEDOUT) {
27             printf("WHEW!\n");
28             exit(0);
29         }
30     }
31     printf("BOOM!\n");
32     exit(0);
33 }
34
35 /* thread routine */
36 void *thread(void *vargp)
37 {
38     getchar();
39     Pthread_mutex_lock(&mutex);
40     Pthread_cond_signal(&cond);
41     Pthread_mutex_unlock(&mutex);
42     return NULL;
43 }
```

code/threads/timebomb.c

Figure 11.24: `timebomb.c`: **A beeping timebomb that explodes unless the user hits a key within 5 seconds.**

11.6 Thread-safe and Reentrant Functions

When we program with threads, we must be careful to write functions that are thread-safe. A function is *thread-safe* if and only if it will always produce correct results when called repeatedly within multiple concurrent threads. If a function is not thread-safe, then it is said to be *thread-unsafe*. We can identify four (non-disjoint) classes of thread-unsafe functions:

1. *Failing to protect shared variables.* We have already encountered this problem with the `count` function in Figure 11.8 that increments an unprotected global counter variable.

   This class of thread-unsafe function is relatively easy to make thread-safe: protect the shared variables with the appropriate synchronization operations (e.g., P and V functions or Pthreads lock and unlock functions). An advantage is that it does not require any changes in the calling program. A disadvantage is that the synchronization operations will slow down the function.

2. *Relying on state across multiple function invocations.* A pseudo-random number generator is a good example of this class of thread-unsafe function. Consider the `rand` package from [37]:

   code/threads/rand.c

   ```
   1 unsigned int next = 1;
   2
   3 /* rand - return pseudo-random integer on 0..32767 */
   4 int rand(void)
   5 {
   6     next = next*1103515245 + 12345;
   7     return (unsigned int)(next/65536) % 32768;
   8 }
   9
  10 /* srand - set seed for rand() */
  11 void srand(unsigned int seed)
  12 {
  13     next = seed;
  14 }
   ```

   code/threads/rand.c

   The `rand` function is thread-unsafe because the result of the current invocation depends on an intermediate result from the previous iteration. When we call `rand` repeatedly from a single thread after seeding it with a call to `srand`, we can expect a repeatable sequence of numbers. However, this assumption no longer holds if multiple threads are calling `rand`.

   The only way to make a function such as `rand` thread-safe is to rewrite it so that it does not use any static data, relying instead on the caller to pass the state information in arguments. The disadvantage is that the programmer is now forced to change the code in the calling routine as well. In a large program where there are potentially hundreds of different call sites, making such modifications could be non-trivial and error-prone.

3. *Returning a pointer to a static variable.* Some functions, such as `gethostbyname`, compute a result in a `static` structure and then return a pointer to that structure. If we call such functions from concurrent threads, then disaster is likely as results being used by one thread are suddenly overwritten by another thread.

   There are two ways to deal with this class of thread-unsafe function. One option is to rewrite the function so that the caller passes the address of the structure to store the results in. This eliminates all shared data, but it requires the programmer to change the code in the caller as well.

   If the thread-unsafe function is difficult or impossible to modify (e.g., it is linked from a library), then another option is to use what we call the *lock-and-copy* approach. The idea is to associate a mutex with the thread-unsafe function. At each call site, lock the mutex, call the thread-unsafe function, dynamically allocate memory for the result, copy the result returned by the function to this memory, and then unlock the mutex. An attractive variant is to define a thread-safe wrapper function that performs the lock-and-copy, and then replace all calls to the thread-unsafe function with calls to the wrapper.

4. *Calling thread-unsafe functions.* If a function f calls a thread-unsafe function, then f is thread-unsafe, too.

Thread-safety can be a confusing issue because there is no simple comprehensive rule for distinguishing thread-safe functions from thread-unsafe ones. Although every thread-unsafe function references shared variables (or calls other functions that are thread-unsafe), not every function that references shared data is thread-unsafe. As we have seen, it all depends on how the function uses the shared variables.

11.6.1 Reentrant Functions

There is an important class of thread-safe functions, known as *reentrant functions*, that are characterized by the property that they do not reference any shared data when they are called by multiple threads. Although the terms *thread-safe* and *reentrant* are sometimes incorrectly used as synonyms, there is a clear technical distinction that is worth preserving. Reentrant functions are typically more efficient than non-reentrant thread-safe functions because they require no synchronization operations. Furthermore, as we have seen, sometimes the only way to convert a thread-unsafe function into a thread-safe one is to rewrite it so that it is reentrant.

Figure 11.25 shows the set relationships between reentrant, thread-safe, and thread-unsafe functions. The set of all functions is partitioned into the disjoint sets of thread-safe and thread-unsafe functions. The set of reentrant functions is a proper subset of the thread-safe functions.

Is it possible to inspect the code of some function and declare *a priori* that it is reentrant? Unfortunately, it depends. If all function arguments are passed by value (i.e., no pointers) and all data references are to local automatic stack variables (i.e., no references to static or global variables), then the function is *explicitly reentrant*, in the sense that we can assert its reentrancy regardless of how it is called.

However, if we loosen our assumptions a bit and allow some parameters in our otherwise explicitly reentrant function to be passed by reference (that is, we allow them to pass pointers) then we have an *implicitly reentrant* function, in the sense that it is only reentrant if the calling threads are careful to pass pointers to

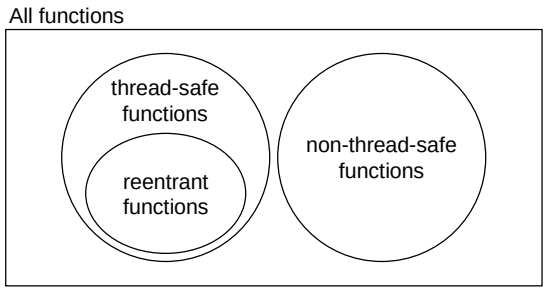


Figure 11.25: **Relationships between the sets of reentrant, thread-safe, and non-thread-safe functions.**

non-shared data. In the rest of the book, we will use the term *reentrant* to include both explicit and implicit reentrant functions, but it is important to realize that reentrancy is sometimes a property of both the caller and the callee.

To understand the distinctions between thread-unsafe, thread-safe, and reentrant functions more clearly, let's consider different versions of our `maketimeout` function from Figure 11.23. We will start with the function in Figure 11.26, a thread-unsafe function that returns a pointer to a static variable.

code/threads/maketimeout_u.c

```
1 #include "csapp.h"
2
3 struct timespec *maketimeout_u(int secs)
4 {
5     static struct timespec timespec;
6     struct timeval now;
7
8     gettimeofday(&now, NULL);
9     timespec.tv_sec = now.tv_sec + secs;
10    timespec.tv_nsec = now.tv_usec * 1000;
11    return &timespec;
12 }
```

code/threads/maketimeout_u.c

Figure 11.26: `maketimeout_u`**: A version of** `maketimeout` **that is not thread-safe.**

Figure 11.27 shows how we might use the lock-and-copy approach to create a thread-safe (but not reentrant) wrapper that the calling program can invoke instead of the original thread-unsafe function.

Finally, we can go all out and rewrite the unsafe function so that it is reentrant, as shown in Figure 11.28. Notice that the calling thread now has the responsibility of passing an address that points to unshared data.

code/threads/maketimeout_t.c

```
1 #include "csapp.h"
2 struct timespec *maketimeout_u(int secs);
3
4 static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
5
6 struct timespec *maketimeout_t(int secs)
7 {
8     struct timespec *sp; /* shared */
9     struct timespec *up = Malloc(sizeof(struct timespec)); /* unshared */
10
11    Pthread_mutex_lock(&mutex);
12    sp = maketimeout_u(secs);
13    *up = *sp;    /* copy the shared struct to the unshared one */
14    Pthread_mutex_unlock(&mutex);
15    return up;
16 }
```

code/threads/maketimeout_t.c

Figure 11.27: `maketimeout_t`**: A version of** `maketimeout` **that is thread-safe but not reentrant.**

code/threads/maketimeout_r.c

```
1 #include "csapp.h"
2
3 struct timespec *maketimeout_r(struct timespec *tp, int secs)
4 {
5     struct timeval now;
6
7     gettimeofday(&now, NULL);
8     tp->tv_sec = now.tv_sec + secs;
9     tp->tv_nsec = now.tv_usec * 1000;
10    return tp;
11 }
```

code/threads/maketimeout_r.c

Figure 11.28: `maketimeout_r`**: A version of** `maketimeout` **that is reentrant and thread-safe.**

11.6.2 Thread-safe Library Functions

Most Unix functions and the functions defined in the standard C library (such as `malloc`, `free`, `realloc`, `printf`, and `scanf`) are thread-safe, with only a few exceptions. Figure 11.29 lists the common exceptions. (See [77] for a complete list.)

Thread-unsafe function	Thread-unsafe class	Unix thread-safe version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	(none)
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

Figure 11.29: **Common thread-unsafe library functions.**

The `asctime`, `ctime`, and `localtime` functions are commonly used functions for converting back and forth between different time and data formats. The `gethostbyname`, `gethostbyaddr`, and `inet_nota` functions are commonly used network programming functions that we will encounter in the next chapter.

With the exception of `rand`, all of these thread-unsafe functions are of the class-3 variety that return a pointer to a static variable. If we need to call one of these functions in a threaded program, the simplest approach is to lock-and-copy as in Figure 11.27.

The disadvantage is that the additional synchronization will slow down the program. Further, this approach will not work for a class-2 thread-unsafe function such as `rand` that relies on static state across calls. Therefore, Unix systems provide reentrant versions of most thread-unsafe functions that end with the "`_r`" suffix. Unfortunately, these functions are poorly documented and the interfaces differ from system to system. Thus, the "`_r`" interface should not be used unless absolutely necessary.

11.7 Other Synchronization Errors

Even if we have managed to make our functions thread-safe, our programs can still suffer from subtle synchronization errors such as races and deadlocks.

11.7.1 Races

A *race* occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y. Races usually occur because programmers assume that threads will take some particular trajectory through the execution state space, forgetting the golden rule that threaded programs must work correctly for any feasible trajectory.

An example is the easiest way to understand the nature of races. Consider the simple program in Figure 11.30. The main thread creates four peer threads and passes a pointer to a unique integer ID to each one.

Each peer thread copies the ID passed in its argument to a local variable (line 22), and then prints a message containing the ID.

code/threads/race.c

```
1 #include "csapp.h"
2
3 #define N 4
4
5 void *thread(void *vargp);
6
7 int main()
8 {
9     pthread_t tid[N];
10    int i;
11
12    for (i = 0; i < N; i++)
13        Pthread_create(&tid[i], NULL, thread, &i);
14    for (i = 0; i < N; i++)
15        Pthread_join(tid[i], NULL);
16    exit(0);
17 }
18
19 /* thread routine */
20 void *thread(void *vargp)
21 {
22    int myid = *((int *)vargp);
23
24    printf("Hello from thread %d\n", myid);
25    return NULL;
26 }
```

code/threads/race.c

Figure 11.30: **A program with a race.**

It looks simple enough, but when we run this program on our system, we get the following incorrect result:

```
unix> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

The problem is caused by a race between each peer thread and the main thread. Can you spot the race?

Here is what happens. When the main thread creates a peer thread in line 13, it passes a pointer to the local stack variable i. At this point the race is on between the next call to `pthread_create` in line 13 and the dereferencing and assignment of the argument in line 22. If the peer thread executes line 22 before the main thread executes line 13, then the `myid` variable gets the correct ID. Otherwise it will contain the ID of some

other thread. The scary thing is that whether we get the correct answer depends on how the kernel schedules the execution of the threads. On our system it fails, but on other systems it might work correctly, leaving the programmer blissfully unaware of a serious bug.

To eliminate the race, we can dynamically allocate a separate block for each integer ID, and pass the thread routine a pointer to this block, as shown in Figure 11.31 (lines 13-15). Notice that the thread routine must free the block in order to avoid a memory leak.

code/threads/norace.c

```
 1 #include "csapp.h"
 2
 3 #define N 4
 4
 5 void *thread(void *vargp);
 6
 7 int main()
 8 {
 9     pthread_t tid[N];
10     int i, *ptr;
11
12     for (i = 0; i < N; i++) {
13         ptr = Malloc(sizeof(int));
14         *ptr = i;
15         Pthread_create(&tid[i], NULL, thread, ptr);
16     }
17     for (i = 0; i < N; i++)
18         Pthread_join(tid[i], NULL);
19     exit(0);
20 }
21
22 /* thread routine */
23 void *thread(void *vargp)
24 {
25     int myid = *((int *)vargp);
26
27     Free(vargp);
28     printf("Hello from thread %d\n", myid);
29     return NULL;
30 }
```

code/threads/norace.c

Figure 11.31: **A correct version the program in Figure 11.30 without a race.**

When we run this program on our system, we now get the correct result:

```
unix> ./norace
Hello from thread 0
Hello from thread 1
```

```
Hello from thread 2
Hello from thread 3
```

We will use a similar technique in Chapter 12 when we discuss the design of threaded network servers.

Practice Problem 11.5:

In Figure 11.31, we might be tempted to free the allocated memory block immediately after line 15 in the main thread, instead of freeing it in the peer thread. But this would be a bad idea. Why?

Practice Problem 11.6:

A. In Figure 11.31, we eliminated the race by allocating a separate block for each integer ID. Outline a different approach that does not call the `malloc` or `free` functions.

B. What are the advantages and disadvantages of this approach?

11.7.2 Deadlocks

Semaphores introduce the potential for a nasty kind of runtime error, called *deadlock*, where a collection of threads are blocked, waiting for a condition that will never be true. The progress graph is an invaluable tool for understanding deadlock. For example, Figure 11.32 shows the progress graph for a pair of threads that use two semaphores for sharing. From this graph, we can glean some important insights about deadlock:

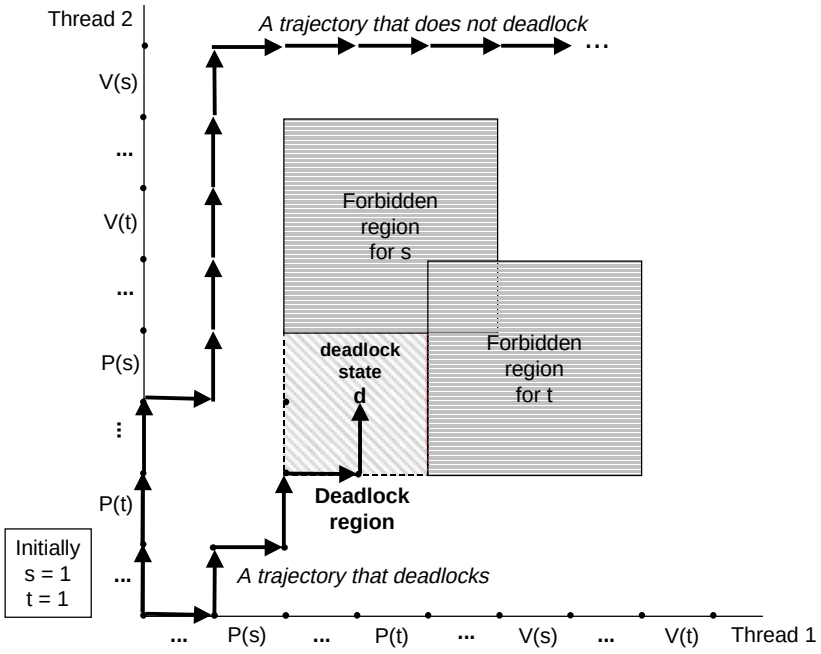


Figure 11.32: **Progress graph for a program that can deadlock.**

- The programmer has incorrectly ordered the P and V operations such that the forbidden regions for the two semaphores overlap. If some execution trajectory happens to reach the `deadlock state` d, then no further progress is possible because the overlapping forbidden regions block progress in every legal direction. In other words, the program is deadlocked because each thread is waiting for the other to do a V operation that will never occur.

- The overlapping forbidden regions induce a set of states, called the *deadlock region*. If a trajectory happens to touch a state in the deadlock region, then deadlock is inevitable. Trajectories can enter deadlock regions, but they can never leave.

- Deadlock is an especially difficult problem, because it is not always predictable. Some lucky execution trajectories will skirt the deadlock region, while others will be trapped by it. Figure 11.32 shows an example of each. The implications for a programmer are somewhat scary. You might run the same program 1000 times without any problem, but then the next time it deadlocks. Or the program might work fine on one machine but deadlock on another. Worst of all, the error is often not repeatable because different executions have different trajectories.

Practice Problem 11.7:

Consider the following program, which uses a pair of semaphores for sharing.

```
Initially: s = 1, t = 0.

Thread 1:       Thread 2:
   P(s);           P(s);
   V(s);           V(s);
   P(t);           P(t);
   V(t);           V(t);
```

A. Does the program always deadlock?

B. What simple change to the initial semaphore values will fix the deadlock?

11.8 Summary

Threads are a popular and useful tool for introducing concurrency in programs. Threads are typically more efficient than processes, and it is much easier to share data between threads than between processes. However, the ease of sharing introduces the possibility of synchronization errors that are difficult to diagnose.

Programmers writing threaded programs must be careful to protect shared data with the appropriate synchronization mechanisms. Functions called by threads must be thread-safe. Races and deadlocks must be avoided. In sum, the wise programmer approaches the design of threaded programs with great care and not a little trepidation.

Bibliographic Notes

Semaphore operations were proposed by Dijkstra [22]. The progress graph concept was introduced by Coffman [15] and later formalized by Carson and Reynolds [9]. The book by Butenhof [8] contains a good description of the Posix threads interface. The paper by Andrew Birrell [4] is an excellent introduction to the general principles of threads programming and its pitfalls.

Homework Problems

Homework Problem 11.8 [Category 2]:

Write a version of `hello.c` (Figure 11.5) that reads a command line argument n, and then creates and reaps n joinable peer threads.

Homework Problem 11.9 [Category 3]:

Generalize the producer-consumer program in Figure 11.19 to manipulate a circular buffer with a capacity of BUFSIZE integer items. The producer generates NITEMS integer items: $0, 1, \ldots,$ NITEMS-1. For each item, it works for a while (i.e., `Sleep(rand()%MAXRAND)`), produces the item by printing a message, and then inserts the item at the rear of the buffer. The consumer repeatedly removes an item from the front of the buffer, works for a while, and then consumes the item by printing a message. For example:

```
unix> ./prodconsn
produced 0
produced 1
consumed 0
produced 2
consumed 1
consumed 2
produced 3
produced 4
consumed 3
consumed 4
```

Homework Problem 11.10 [Category 3]:

Write a barrier synchronization package, with the same interface as the package in Figure 11.22, that uses semaphores instead of Pthreads mutex and condition variables. Write a driver program `barriermain.c` that tests your barrier routine. The driver accepts a command line argument n, calls the `barrier_init` function, and then creates n threads that repeatedly synchronize by printing a message and calling the `barrier`. For example,

```
unix> ./barrier 2
1026: hello from barrier 0
2051: hello from barrier 0
1026: hello from barrier 1
2051: hello from barrier 1
```

```
1026: hello from barrier 2
2051: hello from barrier 2
1026: hello from barrier 3
2051: hello from barrier 3
1026: hello from barrier 4
2051: hello from barrier 4
```

Homework Problem 11.11 [Category 3]:

Implement a threaded version of the C `fgets` function, called `tfgets`, that times out and returns a NULL pointer if it does not receive an input line on stdin within 5 seconds.

- Your function should be implemented in a package called `tfgets-thread.c`.
- Your solution may use any Pthreads function.
- Your solution may not use the Unix `sleep` or `alarm` functions.
- Test your solution using the following driver program:

code/threads/tfgets-main.c

```
1  #include "csapp.h"
2
3  char *tfgets(char *s, int size, FILE *stream);
4
5  int main()
6  {
7      char buf[MAXLINE];
8
9      if (tfgets(buf, MAXLINE, stdin) == NULL)
10         printf("BOOM!\n");
11     else
12         printf("%s", buf);
13
14     exit(0);
15 }
```

code/threads/tfgets-main.c

Homework Problem 11.12 [Category 3]:

For an interesting contrast in concurrency models, implement `tfgets` using processes, signals, and nonlocal jumps instead of threads.

- Your function should be implemented in a package called `tfgets-proc.c`.
- Your solution may not use the Unix `alarm` function.

- Test your solution using the driver program from Problem 11.11.

Chapter 12

Network Programming

Network applications are everywhere. Any time you browse the Web, send an email message, or pop up an X window, you are using a network application. Interestingly, all network applications are based on the same basic programming model, have similar overall logical structures, and rely on the same programming interface.

Network applications rely on many of the concepts that we have already learned in our study of systems. For example, processes, signals, threads, reentrancy, byte ordering, memory mapping, and dynamic storage allocation all play important roles. There are new concepts to master as well. We will need to understand the basic client-server programming model and how to write client-server programs that use the services provided by the Internet. Since Unix models network devices as files, we will also need a deeper understanding of Unix file I/O. At the end, we will tie all of these ideas together by developing a small but functional Web server that can serve both static and dynamic content with text and graphics to real Web browsers.

12.1 Client-Server Programming Model

Every network application is based on the *client-server model*. With this model, an application consists of a *server* process and one or more *client* processes. A server manages some *resource*, and it provides some *service* for its clients by manipulating that resource.

For example, a Web server manages a set of disk files that it retrieves for clients. An FTP server manages a set of disk files that it stores and retrieves for clients. An X server manages a bit-mapped display, which is paints for clients, and a keyboard and mouse, which it reads for clients. The X server is interesting because it is always close to the user while the client can be far away. Thus proximity plays no role in the definitions of clients and servers, even though we often think of servers as being remote and clients being local.

The fundamental operation in the client-server model is the *transaction* depicted in Figure 12.1.

A transaction consists of four steps:

1. When a client needs service, it initiates a transaction by sending a *request* to the server. For example, when a Web browser needs a file, it sends a request to a Web server.

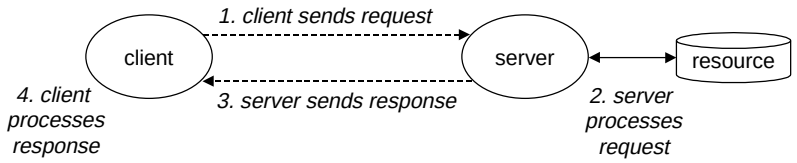


Figure 12.1: **A client-server transaction.**

2. The server receives the request, interprets it, and manipulates its resource in the appropriate way. For example, when a Web server receives a request from a browser, it reads a disk file.

3. The server sends a *response* to the client, and then waits for the next request. For example, a Web server sends the file back to a client.

4. The client receives the response and manipulates it. For example, after a Web browser receives a page from the server, it displays it on the screen.

Aside: Client-server transactions vs database transactions.
Client-server transactions are *not* database transactions and do not share any of their properties. In this context, a transaction simply connotes a sequence of steps by a client and server. **End Aside.**

It is important to realize that clients and servers are processes, and not machines, or *hosts* as they are often called in this context. A single host can run many different clients and servers concurrently, and a client and server transaction can be on the same or different hosts. The client-server model is the same, regardless of the mapping of clients and servers to hosts.

12.2 Networks

Clients and servers often run on separate hosts and communicate using the hardware and software resources of a *computer network*. Networks are sophisticated systems and we can only hope to scratch the surface here. Our aim is to give you a a workable mental model from a progammer's perspective.

To a host, a network is just another I/O device that serves as a source and sink for data, as shown in Figure 12.2. An adapter plugged into an expansion slot on the I/O bus provides the physical interface to the network. Data received from the network is copied from the adapter across the I/O and memory buses into memory, typically by a DMA transfer. Similarly, data can also be copied from memory to the network.

Physically, a network is a hierarchical system that is organized by geographical proximity. At the lowest level is a LAN (Local Area Network) that spans a building or a campus. The most popular LAN technology by far is *ethernet*, which was developed in the mid-1970's at Xerox PARC. Ethernet has proven to be remarkably resilient, evolving from 3 Mb/s transfer rates, to 10 Mb/s, to 100 Mb/s, and more recently to 1 Gb/s.

An *ethernet segment* consists of some wires (usually twisted pairs of wires) and a small box called a *hub*, as shown in Figure 12.3. Ethernet segments typically span small areas, such as a room or a floor in a building. Each wire has the same maximum bit bandwidth, typically 100 Mb/s or 1 Gb/s. One end is attached to an

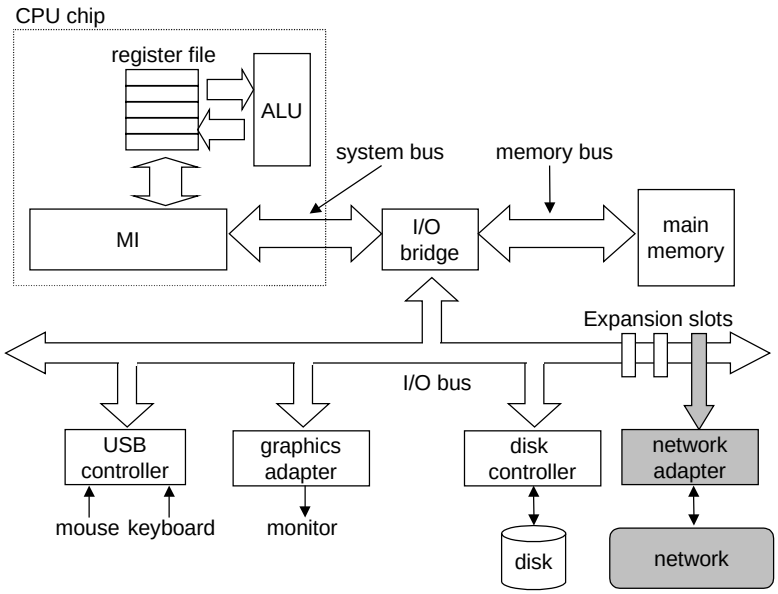


Figure 12.2: **Hardware organization of a network host.**

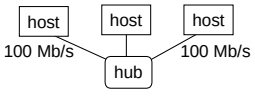


Figure 12.3: **Ethernet segment.**

adapter on a host, and the other end is attached to a *port* on the hub. A hub slavishly copies every bit that it receives on each port to every other port. Thus every host sees every bit.

Each ethernet adapter has a globally unique 48-bit address that is stored in a persistent memory on the adapter. A host can send a chunk of bits called a *frame* to any other host on the segment. Each frame includes some fixed number of *header* bits that identify the source and destination of the frame and the frame length, followed by a *payload* of data bits. Every host adapter sees the frame, but only the destination host actually reads it.

Multiple ethernet segments can be connected into larger LANs, called *bridged ethernets*, using a set of wires and small boxes called *bridges*, as shown in Figure 12.4. Bridged ethernets can span entire buildings or campuses. In a bridged ethernet, some wires connect bridges to bridges, and others connect bridges to

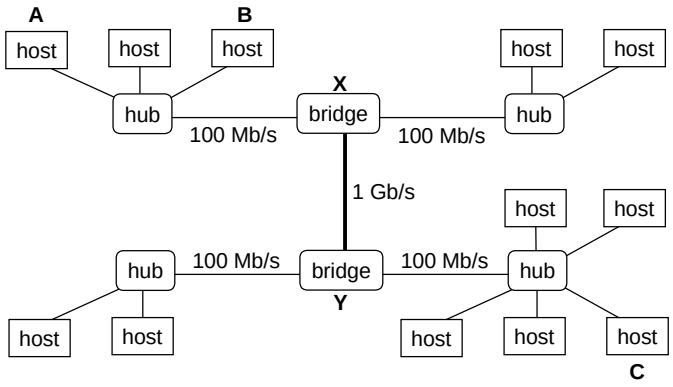


Figure 12.4: **Bridged ethernet segments.**

hubs. The bandwidths of the wires can be different. In our example, the bridge-bridge wire has a 1 Gb/s bandwidth, while the four hub-bridge wires have bandwidths of 100 Mb/s.

Bridges make better use of the available wire bandwidth than hubs. Using a clever distributed algorithm, they automatically learn over time which hosts are reachable from which ports, and then selectively copy frames from one port to another only when it is necessary. For example, if host A sends a frame to host B, which is on the segment, then bridge X will throw away the frame when it arrives at its input port, thus saving bandwidth on the other segments. However, if host A sends a frame to host C on a different segment, then bridge X will copy the frame only to the port connected to bridge Y, which will copy the frame only to the port connected to bridge C's segment.

To simplify our pictures of LANs, we will draw the hubs and bridges and the wires that connect them as a single horizontal line, as shown in Figure 12.5.

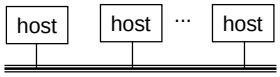


Figure 12.5: **Conceptual view of a LAN.**

At a higher level in the hierarchy, multiple incompatible LANs can be connected by specialized computers called *routers* to form an *internet* (interconnected network).

Aside: Internet vs. internet.
We will always use lower-case *internet* to denote the general concept, and upper-case *Internet* to denote a specific implementation, namely the global IP Internet. **End Aside.**

Each router has an adapter (port) for each network that it is connected to. Routers can also connect high-speed point-to-point phone connections, which are examples of networks known as WANs (Wide-Area Networks), so called because they span larger geographical areas than LANs. In general, routers can be used to build internets from arbitrary collections of LANs and WANs. For example, Figure 12.6 shows an example internet with a pair of LANs and WANs connected by three routers.

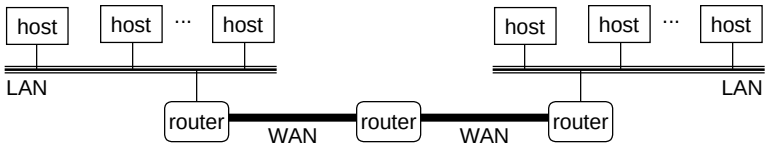


Figure 12.6: **A small internet.** Two LANs and two WANs are connected by three routers.

The crucial property of an internet is that it can consist of different LANs and WANs with radically different and incompatible technologies. Each host is physically connected to every other host, but how is it possible for some *source host* to send data bits to another *destination host* across all of these incompatible networks?

The solution is a layer of *protocol software* running on each host and router that smooths out the differences between the different networks. This software implements a *protocol* that governs how hosts and routers cooperate in order to transfer data. The protocol must provide two basic capabilities:

- *Naming scheme.* Different LAN technologies have different and incompatible ways of assigning addresses to hosts. The internet protocol smooths these differences by defining a uniform format for host addresses. Each host is then assigned at least one of these *internet addresses* that uniquely identifies it.
- *Delivery mechanism.* Different networking technologies have different and incompatible ways of encoding bits on wires and of packaging these bits into frames. The internet protocol smoothes these differences by defining a uniform way to bundle up data bits into discrete chunks called *packets*. A packet consists of a *header*, which contains the packet size and addresses of the source and destination hosts, and a *payload*, which contains data bits sent from the source host.

Figure 12.7 shows an example of how hosts and routers use the internet protocol to transfer data across incompatible LANs.

The example internet consists of two LANs connected by a router. A client running on host A, which is attached to LAN1, sends a sequence of data bytes to a server running on host B, which is attached to LAN2. There are eight basic steps:

1. The client on host A invokes a system call that copies the data from the client's virtual address space into a kernel buffer.

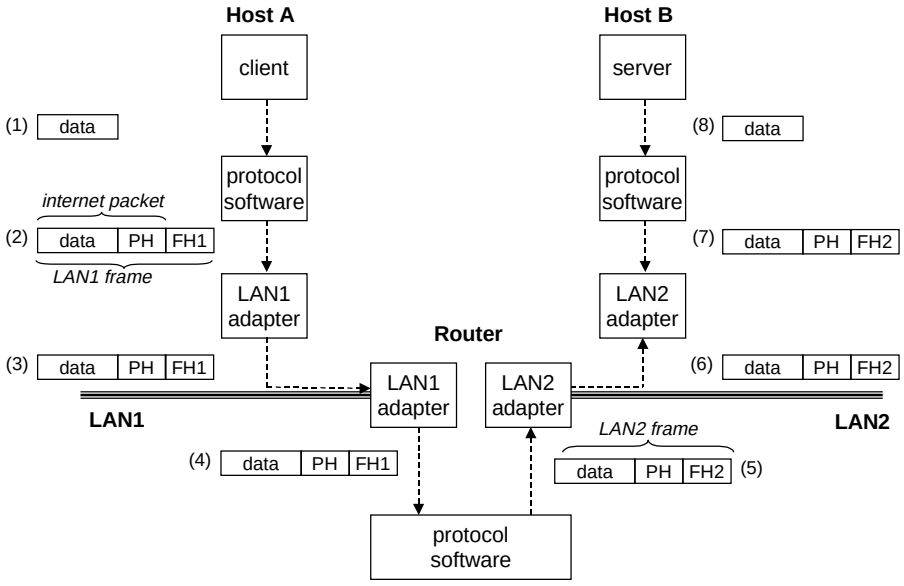


Figure 12.7: **How data travels from one host to another on an internet.** Key: PH: internet packet header, FH1: frame header for LAN1, FH2: frame header for LAN2.

2. The protocol software on host A creates a LAN1 frame by appending an internet header and a LAN1 frame header to the data. The internet header is addressed to internet host B. The LAN1 frame header is addressed to the router. It then passes the frame to the adapter. Notice that the payload of the LAN1 frame is an internet packet, whose payload is the actual user data. This kind of *encapsulation* is one of the fundamental insights of internetworking.

3. The LAN1 adapter copies the frame to the network.

4. When the frame reaches the router, the router's LAN1 adapter reads it from the wire and passes it to the protocol software.

5. The router fetches the destination internet address from the internet packet header and uses this as an index into a routing table to determine where to forward the packet, which in this case is LAN2. The router then strips off the old LAN1 frame header, prepends a new LAN2 frame header addressed to host B, and passes the resulting frame to the adapter.

6. The router's LAN2 adapter copies the frame to the network.

7. When the frame reaches host B, its adapter reads the frame from the wire and passes it to the protocol software.

8. Finally, the protocol software on host B strips off the packet header and frame header. The protocol software will eventually copy the resulting data into the server's virtual address space when the server invokes a system call that reads the data.

Of course, we are glossing over many difficult issues here. What if different networks have different maximum frame sizes? How do routers know where to forward frames? How are routers informed when the the

network topology changes? What if packet gets lost? Nonetheless, our example captures the essence of the internet idea, and encapsulation is key.

12.3 The Global IP Internet

The global IP Internet is the most famous and successful implementation of an internet. It has existed in one form or another since 1970. While the internal architecture of the Internet is complex and constantly changing, the organization of client-server applications has remained remarkably stable since the early 1980s. Figure 12.8 shows the basic hardware and software organization of an Internet client-server application.

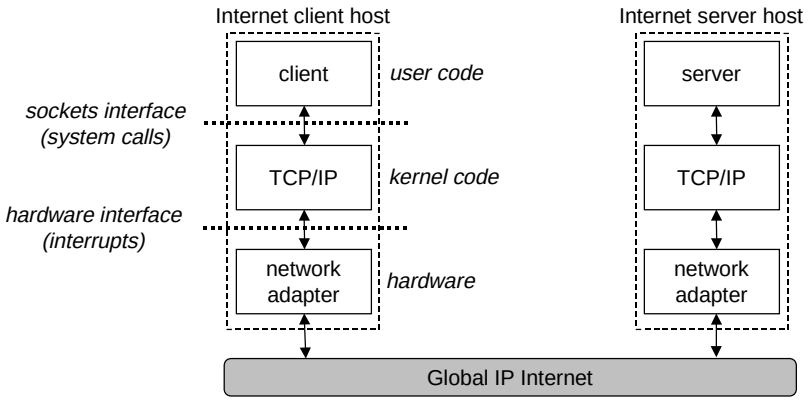


Figure 12.8: **Hardware and software organization of an Internet application.**

Each Internet host runs software that implements the *TCP/IP* protocol (Transmission Control Protocol/Internet Protocol), which is supported by almost every modern computer system. Internet clients and servers communicate using a mix of *sockets interface* functions and Unix file I/O functions. (We will describe Unix file I/O in Section 12.4 and the sockets interface in Section 12.5.) The sockets functions are typically implemented as system calls that trap into the kernel and call various kernel-mode functions in TCP/IP.

Aside: Berkeley sockets.
The sockets interface was developed by researchers at University of California at Berkeley in the early 1980s. For this reason, it is still often referred to as *Berkeley sockets*. The Berkeley researchers developed the sockets interface to work with any underlying protocol. The first implementation was for TCP/IP, which they included in the Unix 4.2BSD kernel and distributed to numerous universities and labs. This was one of the most important events in the history of the Internet. Almost overnight, thousands of people had access to TCP/IP and its source codes. It generated tremendous excitement and sparked a flurry of new research in networking and internetworking. **End Aside.**

TCP/IP is actually of family of protocols, each of which contributes different capabilities. For example, the IP protocol provides the basic naming scheme and a delivery mechanism that can send packets known as *datagrams* from one Internet host to any another host. The IP mechanism is unreliable in the sense that it makes no effort to recover if datagrams are lost or duplicated in the network. UDP (Unreliable Datagram Protocol) extends IP slightly so that packets can be transfered from process to process, rather than host to

host. TCP is a complex protocol that builds on IP to provide reliable full-duplex connections between processes. To simplify our discussion, we will treat TCP/IP as a single monolithic protocol. We will not discuss its inner workings, and we will only discuss some of the basic capabilities that TCP and IP provide to application programs. We will not discuss UDP.

From a programmer's perspective, we can think of the Internet as a worldwide collection of hosts with the following properties:

- Hosts are mapped to a set of 32-bit *IP addresses.*
- The set of IP addresses is mapped to a set of identifiers called *Internet domain names.*
- A process on one host communicates with a process on another host over a *connection.*

The next three sections discuss these fundamental ideas in more detail.

12.3.1 IP Addresses

An IP address is an unsigned 32-bit integer. Network programs store IP addresses in the *IP address structure* shown in Figure 12.9.

netinet/in.h

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

netinet/in.h

Figure 12.9: **IP address structure.**

Aside: Why store the scalar IP address in a structure?
Storing a scalar address in a structure is an unfortunate historical artifact from the early Berkeley 4.xBSD implementations of the sockets interface. It would make more sense to define a scalar type for IP addresses, but it is too late to change now because of the enormous installed base of applications. **End Aside.**

Because Internet hosts can have different host byte orders, TCP/IP defines a uniform *network byte order* (which is a big-endian byte order) for any integer data item, such as an IP address, that is carried across the network in a packet header. Addresses in IP address structures are always stored in big-endian network byte order, even if the host byte order is little-endian. Unix provides the following functions for converting between network and host byte order.

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
                                                    both return: value in network byte order
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
                                                    both return: value in host byte order
```

The `hotnl` function converts a 32-bit integer from host byte to network byte order. The `ntohl` function converts a 32-bit integer from network byte order to host byte order. The `htons` and `ntohs` functions perform corresponding conversions for 16-bit integers.

IP addresses are typically presented to humans in a form known as *dotted-decimal notation*, where each byte is represented by its decimal value and separated from the other bytes by a period. For example, `128.2.194.242` is the dotted-decimal representation of the address `0x8002c2f2`. You can use the Linux HOSTNAME command to determine the dotted-decimal address of your own host:

```
linux> hostname -i
128.2.194.242
```

Internet programs convert back and forth between IP addresses and dotted-decimal strings using the `inet_aton` and `inet_ntoa` functions:

```
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
                                                    returns: 1 if OK, 0 on error
char *inet_ntoa(struct in_addr in);
                                                    returns: pointer to a dotted-decimal string
```

The `inet_aton` function converts a dotted-decimal string (`cp`) to an IP address in network byte order (`inp`). Similarly, the `inet_ntoa` function converts an IP address in network byte order to its corresponding dotted-decimal string. Notice that a call to `inet_aton` passes a pointer to a structure, while a call to `inet_ntoa` passes the structure itself.

Aside: What do `ntoa` and `aton` mean?
The `"n"` denotes *network* representation. The `"a"` denotes *application* representation. **End Aside.**

Practice Problem 12.1:

Complete the following table.

Hex address	Dotted-decimal address
0x0	
0xffffffff	
0xef000001	
	205.188.160.121
	64.12.149.13
	205.188.146.23

Practice Problem 12.2:

Write a program `hex2dd.c` that converts its hex argument to a dotted-decimal string and prints the result. For example,

```
unix> ./hex2dd 0x8002c2f2
128.2.194.242
```

Practice Problem 12.3:

Write a program `dd2hex.c` that converts its dotted-decimal argument to a hex number and prints the result. For example,

```
unix> ./dd2hex 128.2.194.242
0x8002c2f2
```

12.3.2 Internet Domain Names

Internet clients and servers use IP addresses when they communicate with each other. However, large integers are difficult for people to remember, so the Internet also defines a separate set of more human-friendly *domain names* as well as a mechanism that maps the set of domain names to the set of IP addresses. A domain name is a sequence of words (letters, numbers, and dashes) separated by periods. For example,

```
kittyhawk.cmcl.cs.cmu.edu
```

The set of domain names forms a hierarchy and each domain name encodes its position in the hierarchy. An example is the easiest way to understand this. Figure 12.10 shows a portion of the domain name hierarchy. The hierarchy is represented as a tree. The nodes of the tree represent domain names that are formed by the path back to the root. Sub-trees are referred to as *subdomains*. The first level in the hierarchy is an unnamed root node. The next level is a collection of *first-level domain names* that are defined by a non-profit organization called ICANN (Internet Corporation for Assigned Names and Numbers). Common first-level domains include `com`, `edu`, `gov`, `org`, and `net`.

At the next level are *second-level* domain names such as `cmu.edu`, which are assigned on a first-come first-serve basis by various authorized agents of ICANN. Once an organization has received a second-level domain name, then it is free to create any other new domain name within its subdomain.

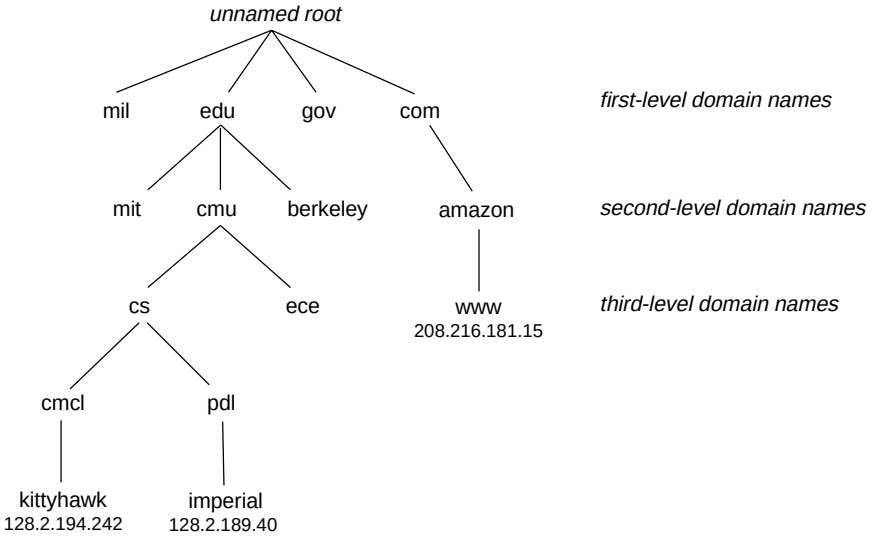


Figure 12.10: **Subset of the Internet domain name hierarchy.**

The Internet defines a mapping between the set of domain names and the set of IP addresses. Until 1988, this mapping was maintained manually in a single text file called `hosts.txt`. Since then, the mapping has been maintained in a distributed world-wide database known as *DNS* (Domain Naming System). The DNS database consists of millions of the *host entry structures* shown in Figure 12.11, each of which defines the mapping between a set of domain names (an official name and a list of aliases) and a set of IP addresses. In a mathematical sense, we can think of each host entry as an equivalence class of domain names and IP addresses.

netdb.h

```
/* DNS host entry structure */
struct hostent {
    char   *h_name;        /* official domain name of host */
    char   **h_aliases;    /* null-terminated array of domain names */
    int    h_addrtype;     /* host address type (AF_INET) */
    int    h_length;       /* length of an address, in bytes */
    char   **h_addr_list;  /* null-terminated array of in_addr structs */
};
```

netdb.h

Figure 12.11: **DNS host entry structure.**

Internet applications retrieve arbitrary host entries from the DNS database by calling the `gethostbyname` and `gethostbyaddr` functions.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
                                        returns: non-NULL pointer if OK, NULL pointer on error with h_errno set
struct hostent *gethostbyaddr(const char *addr, int len, 0);
                                        returns: non-NULL pointer if OK, NULL pointer on error with h_errno set
```

The `gethostbyname` returns the host entry associated with the domain name `name`. The `gethostbyaddr` function returns the host entry associated with the IP address `addr`. The second argument gives the length in bytes of an IP address, which for the current Internet is always four bytes. For our purposes, the third argument is always zero.

We can explore some of the properties of the DNS mapping with the `hostinfo` program in Figure 12.12, which reads a domain name or dotted-decimal address from the command line and displays the corresponding host entry. (We are actually calling error handling wrappers, which were introduced in Section 8.3 and described in detail in Appendix A.)

Each Internet host has the locally-defined domain name `localhost`, which always maps to the *loopback address* `127.0.0.1`:

```
unix> ./hostinfo localhost
official hostname: localhost
alias: localhost.localdomain
address: 127.0.0.1
```

The `localhost` name provides a convenient and portable way to reference clients and servers that are running on the same machine, which can be especially useful for debugging. We can use HOSTNAME to determine the real domain name of our local host:

```
unix> hostname
kittyhawk.cmcl.cs.cmu.edu
```

In the simplest case there is a one-to-one mapping between a domain name and an IP address:

```
unix> ./hostinfo kittyhawk.cmcl.cs.cmu.edu
official hostname: kittyhawk.cmcl.cs.cmu.edu
address: 128.2.194.242
```

However, in some cases, multiple domain names are mapped to the same IP address:

```
unix> ./hostinfo cs.mit.edu
official hostname: EECS.MIT.EDU
alias: cs.mit.edu
address: 18.62.1.6
```

In the most general case, multiple domain names can be mapped to multiple IP addresses:

```
unix> ./hostinfo www.aol.com
```

code/net/hostinfo.c

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     char **pp;
6     struct in_addr addr;
7     struct hostent *hostp;
8
9     if (argc != 2) {
10        fprintf(stderr, "usage: %s <domain name or dotted-decimal>\n",
11                argv[0]);
12        exit(0);
13    }
14
15    if (inet_aton(argv[1], &addr) != 0)
16        hostp = Gethostbyaddr((const char *)&addr, sizeof(addr), AF_INET);
17    else
18        hostp = Gethostbyname(argv[1]);
19
20    printf("official hostname: %s\n", hostp->h_name);
21
22    for (pp = hostp->h_aliases; *pp != NULL; pp++)
23        printf("alias: %s\n", *pp);
24
25    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
26        addr.s_addr = *((unsigned int *)*pp);
27        printf("address: %s\n", inet_ntoa(addr));
28    }
29    exit(0);
30 }
```

code/net/hostinfo.c

Figure 12.12: HOSTINFO**: Retrieves and prints a DNS host entry.**

```
official hostname: aol.com
alias: www.aol.com
address: 205.188.160.121
address: 64.12.149.13
address: 205.188.146.23
```

Finally, we notice that some valid domain names are not mapped to any IP address:

```
unix> ./hostinfo edu
Gethostbyname error: No address associated with name
unix> ./hostinfo cmcl.cs.cmu.edu
Gethostbyname error: No address associated with name
```

Practice Problem 12.4:

Compile the HOSTINFO program from Figure 12.12. Then run `hostinfo aol.com` three times in a row on your system.

A. What do you notice about the ordering of the IP addresses in the three host entries?

B. How might this ordering be useful?

12.3.3 Internet Connections

Internet clients and servers communicate by sending and receiving streams of bytes over *connections*. A connection is *point-to-point* in the sense that it connects a pair of processes. It is *full-duplex* in the sense that data can flow in both directions at the same time. And it is *reliable* in the sense that — barring some catastrophic failure such as a cable cut by a careless backhoe operator — the stream of bytes sent by the source process is eventually received by the destination process in the same order it was sent.

A *socket* is an endpoint of a connection. Each socket has a corresponding *socket address* that consists of an Internet address and an 16-bit integer *port*, and is denoted by `address:port`. The port in the client's socket address is assigned automatically by the kernel when the client makes a connection request, and is known as an *ephemeral port*. However, the port in the server's socket address is typically some *well-known port* that is associated with the service. For example, Web servers typically use port 80, and email servers use port 25. On Unix machines, the file `/etc/services` contains a comprehensive list of the services provided on that machine, along with their well-known ports.

A connection is uniquely identified by the socket addresses of its two endpoints. This pair of socket addresses is known as a *socket pair* and is denoted by the tuple

```
(cliaddr:cliport, servaddr:servport)
```

where `cliaddr` is the client's IP address, `cliport` is the client's port, `servaddr` is the server's IP address, and `servport` is the server's port. For example, Figure 12.13 shows a connection between a Web client and a Web server.

In this example, the Web client's socket address is

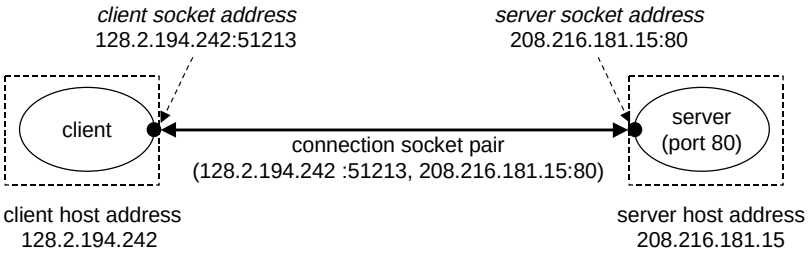


Figure 12.13: **Anatomy of an Internet connection**

`128.2.194.242:51213`

where port `51213` is an ephemeral port assigned by the kernel. The Web server's socket address is

`208.216.181.15:80`

where port `80` is the well-known port associated with Web services. Given these client and server socket addresses, the connection between the client and server is uniquely identified by the socket pair

```
(128.2.194.242:51213, 1208.216.181.15:80).
```

In Section 12.5 we will learn how C programs use the sockets interface to establish connections between clients and servers. But since sockets are modeled in Unix as files, we must first develop an understanding of Unix file I/O, which is the topic of the next section.

12.4 Unix file I/O

A Unix *file* is a sequence of n bytes

$$B_0, B_1, \ldots, B_k, \ldots, B_{n-1}.$$

All I/O devices, such as networks, disks, and terminals, are modeled as files, and all input and output is performed by reading and writing the appropriate files. This elegant mapping of devices to files allows Unix to export a simple, low-level application interface, known as *Unix I/O*, that enables all input and output to be performed in a uniform and consistent way.

> **Aside: Standard I/O and Unix I/O.**
> The familiar, higher-level I/O routines in the C standard library, such as `printf` and `scanf`, are all implemented using the lower-level Unix I/O functions. **End Aside.**

An application announces its intention to access an I/O device by asking the kernel to *open* the corresponding file. The kernel returns a small non-negative integer, called a *descriptor*, that identifies the file in all subsequent operations on the file. The kernel keeps track of all information about the open file; the application keeps keep track of only the descriptor.

The kernel maintains a *file position* k, initially 0, for each open file. An application can set the current file position k explicitly by performing a *seek* operation.

A *read* operation copies $m > 0$ bytes from the file to memory, starting at the current file position k, and then incrementing k by m. A read operation with $k \geq n$ triggers a condition known as *end-of-file* (EOF), which can be detected by the application. Notice that there is no explicit "EOF character" at the end of a file. Similarly, a *write* operation copies $m > 0$ bytes from memory to a file, starting at the current file position k, and then updating k.

When an application is finished reading and writing the file, it informs the kernel by asking it to *close* the file. The kernel frees the structures it created when the file was opened and restores the descriptor to a pool of available descriptors. The next file that is opened is guaranteed to receive the smallest available descriptor in the pool. When a process terminates for any reason, the kernel closes all open files, and frees their memory resources.

By convention, each process created by a Unix shell begins life with three open files: *standard input* (descriptor 0), *standard output* (descriptor 1), and *standard error* (descriptor 2). The system header file `unistd.h` defines the following constants,

```
#define STDIN_FILENO  0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

which for clarity can be used instead of explicit descriptor values.

12.4.1 The `read` and `write` Functions

Applications perform input and output by calling the `read` and `write` functions, respectively.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
                                    returns: number of bytes read if OK, 0 on EOF, -1 on error

ssize_t write(int fd, const void *buf, size_t count);
                                    returns: number of bytes written if OK, -1 on error
```

The `read` function copies at most `count` bytes from the current file position of descriptor `fd` to memory location `buf`. A return value of -1 indicates an error, and a return value of 0 indicates EOF. Otherwise, the return value indicates the number of bytes that were actually transferred.

The `write` function copies at most `count` bytes from memory location `buf` to the current file position of descriptor `fd`. Figure 12.14 shows a program that uses `read` and `write` calls to copy the standard input to the standard output, one byte at a time.

code/net/cpstdin.c

```
1 #include "csapp.h"
2
3 int main(void)
4 {
5     char c;
6
7     /* copy stdin to stdout, one byte at a time */
8     while(Read(STDIN_FILENO, &c, 1) != 0)
9         Write(STDOUT_FILENO, &c, 1);
10    exit(0);
11 }
```

code/net/cpstdin.c

Figure 12.14: **Copies standard input to standard output.**

12.4.2 Robust File I/O With the `readn` and `writen` Functions.

In some situations, `read` and `write` transfer fewer bytes than the application requests. Such *short counts* do not indicate an error, and can occur for a number of reasons:

- *Encountering end-of-file on reads.* If the file contains only 20 more bytes and we are reading in 50-byte chunks, then the current `read` will return a short count of 20. The next `read` will signal EOF (end-of-file) by returning a short count of zero.
- *Reading text lines from a terminal.* If the open file is associated with a terminal (i.e., a keyboard and display), then the `read` function will transfer the next text line.
- *Reading and writing network sockets.* If the open file corresponds to a network socket, then internal buffering constraints and long network delays can cause `read` and `write` to return short counts.

Robust applications in general, and network applications in particular, must anticipate and deal with short counts. In Figure 12.14 we skirted this issue by transferring one byte at a time. While technically correct, this approach is grossly inefficient because it requires $2n$ system calls. Instead, you should use the `readn` and `writen` functions from W. Richard Stevens's classic network programming text [77].

```
#include "csapp.h"

ssize_t readn(int fd, void *buf, size_t count);
ssize_t writen(int fd, void *buf, size_t count);
```

both return: number of bytes read (0 if EOF) or written, -1 on error

The code for these functions is shown in Figure 12.15. The `readn` function returns a short count only when the input operation extends past the end of file. Other short counts are handled by repeatedly invoking `read` until `count` bytes have been transferred. The `writen` function never returns a short count.

code/src/csapp.c

```
ssize_t readn(int fd, void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nread;
    char *ptr = buf;

    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        ptr += nread;
    }
    return (count - nleft);    /* return >= 0 */
}
```

code/src/csapp.c

code/src/csapp.c

```
ssize_t writen(int fd, const void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nwritten;
    const char *ptr = buf;

    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0;    /* and call write() again */
            else
                return -1;       /* errorno set by write() */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return count;
}
```

code/src/csapp.c

Figure 12.15: `readn` **and** `writen`**: Robust versions of** `read` **and** `write` Adapted from [77].

Notice that the routines manually restart `read` or `write` if they are interrupted by the return from an application signal handler (lines 9-10). Manual restarts are unnecessary on Unix systems, which automatically restart interrupted `read` and `write` calls. However, other systems such as Solaris do not restart interrupted system calls, and on these systems we must manually restart them.

12.4.3 Robust Input of Text Lines Using the `readline` Function

A *text line* is a sequence of ASCII characters terminated by a newline character. (The newline character is the same as the ASCII line feed character (LF) and has a numeric value of `0x0a`.) Many network applications, such as Web clients and servers, communicate using sequences of text lines. For these programs you should use the `readline` function [77] whenever you input a text line.

```
#include "csapp.h"

ssize_t readline(int fd, void *buf, size_t maxlen);
                                    returns: number of bytes read (0 if EOF), -1 on error
```

The `readline` function has the same semantics as the `fgets` function in the C Standard I/O library. It reads the next text line from file `fd` (including the terminating newline character), copies it to memory location `buf`, and terminates the text line with the null character. `Readline` reads at most `maxlen-1` bytes, leaving room for the terminating zero. If the text line is longer than `maxlen-1` bytes, then `readline` simply returns the first `maxlen-1` bytes of the line. Figure 12.16 shows the code for the `readline` package. It is somewhat subtle and needs to be studied carefully.

The `my_read` function copies the next character in the file to location `ptr`. It returns -1 on error (with `errno`) set appropriately, 0 on EOF, and 1 otherwise. Notice that `my_read` is a `static` function, and thus is not visible to applications.

To improve efficiency, `my_read` maintains a `static` buffer that it refreshes in MAXLINE-sized blocks. Variable `read_ptr` points to the buffer byte to return to the caller, and variable `read_cnt` is the number of bytes in the buffer that have yet to be returned to the caller. The function initiates a new block-read operation each time `read_cnt` drops to zero (line 6).

The `readline` function calls `my_read` at most `maxlen-1` times, terminating either when it encounters a newline character (line 30), when `my_read` returns EOF (line 35), or when `my_read` indicates an error (line 40).

12.4.4 The `stat` Function

An application retrieves information about disk files by calling the `stat` function.

code/src/csapp.c

```
1  static ssize_t my_read(int fd, char *ptr)
2  {
3      static int read_cnt = 0;
4      static char *read_ptr, read_buf[MAXLINE];
5
6      if (read_cnt <= 0) {
7      again:
8          if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
9              if (errno == EINTR)
10                 goto again;
11             return -1;
12         }
13         else if (read_cnt == 0)
14             return 0;
15         read_ptr = read_buf;
16     }
17     read_cnt--;
18     *ptr = *read_ptr++;
19     return 1;
20 }
21
22 ssize_t readline(int fd, void *buf, size_t maxlen)
23 {
24     int n, rc;
25     char c, *ptr = buf;
26
27     for (n = 1; n < maxlen; n++) {  /* notice that loop starts at 1 */
28         if ( (rc = my_read(fd, &c)) == 1) {
29             *ptr++ = c;
30             if (c == '\n')
31                 break;          /* newline is stored, like fgets() */
32         }
33         else if (rc == 0) {
34             if (n == 1)
35                 return 0;       /* EOF, no data read */
36             else
37                 break;          /* EOF, some data was read */
38         }
39         else
40             return -1;  /* error, errno set by read() */
41     }
42     *ptr = 0;   /* null terminate like fgets() */
43     return n;
44 }
```

code/src/csapp.c

Figure 12.16: `readline` **package: Reads a text line from a descriptor.** Adapted from [77].

```
#include <unistd.h>
#include <sys/stat.h>

int stat(const char *filename, struct stat *buf);
```
returns: 0 if OK, -1 on error

The `stat` function takes as input a filename, such as `/usr/dict/words`, and fills in the members of a `stat` structure shown in Figure 12.17. We will need the `st_mode` and `st_size` members of the `stat` structure when we discuss Web servers in Section 12.7. The `st_mode` member encodes both the file type and the file protection bits. The `st_size` member contains the file size in bytes. The meaning of the other members is beyond our scope.

statbuf.h (included by sys/stat.h)

```
/* file info returned by the stat function */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

statbuf.h (included by sys/stat.h)

Figure 12.17: **The `stat` structure.**

A Unix system recognizes a number of different file types. For example, a *regular file* contains some sort of binary or text data. To the kernel there is no difference between text files and binary files. A *directory file* contains information about other files. And a *socket* is a file that is used to communicate with another process across a network. Unix provides macro predicates for determining the file type. Figure 12.18 shows a subset. Each file type macro takes an `st_mode` member as its argument.

Macro	Description
S_ISREG()	Is this a regular file?
S_ISDIR()	Is this a directory file?

Figure 12.18: **Some macros for determining the type of file.** Defined in `sys/stat.h`

The protection bits in `st_mode` can be tested using the bit masks in Figure 12.19. For example, the following code fragment checks if the current process has permission to read a file:

`st_mode` mask	Description
S_IRUSR	User (owner) can read this file
S_IWUSR	User (owner) can write this file
S_IXUSR	User (owner) can execute this file
S_IRGRP	Group members can read this file
S_IWGRP	Group members can write this file
S_IXGRP	Group members can execute this file
S_IROTH	Others (anyone) can read this file
S_IWOTH	Others (anyone) can write this file
S_IXOTH	Others (anyone) can execute this file

Figure 12.19: **Masks for checking protection bits.** Defined in `sys/stat.h`

```
1     if (S_ISREG(stat.st_mode) && (stat.st_mode & S_IRUSR))
2         printf("This is a regular file that I can read\n");
```

12.4.5 The `dup2` Function

The Unix shell provides an I/O redirection operator that allows users to redirect the standard output to a disk file. For example,

```
unix> ls >foo
```

writes the standard output of the `ls` program to the file `foo`. As we shall see in Section 12.7, a Web server performs a similar kind of redirection when it runs a CGI program on behalf of the client. One way to accomplish I/O redirection is to use the `dup2` function.

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

returns: nonnegative descriptor if OK, -1 on error

The `dup2` function duplicates descriptor `oldfd`, assigns it to descriptor `newfd`, and returns `newfd`. If `newfd` was already open, then `dup2` closes `newfd` before it duplicates `oldfd`.

For each process, the kernel maintains a *descriptor table* that is indexed by the process's open descriptors. The entry for an open descriptor points to a *file table entry* that consists of, among other things, the current file position and a *reference count* of the number of descriptor entries that currently point to it. The file table entry in turn points to an *i-node* table entry that characterizes the physical location of the file on disk, and contains most of the information in the `stat` structure, including the `st_mode` and `st_size` members.

Typically there is a one-to-one mapping between descriptors and files. For example, suppose we have the situation in Figure 12.20 where descriptor 1 (stdout) corresponds to file A (say a terminal), while descriptor 4 corresponds to file B (say a disk). The reference counts for the A and B are both equal to 1.

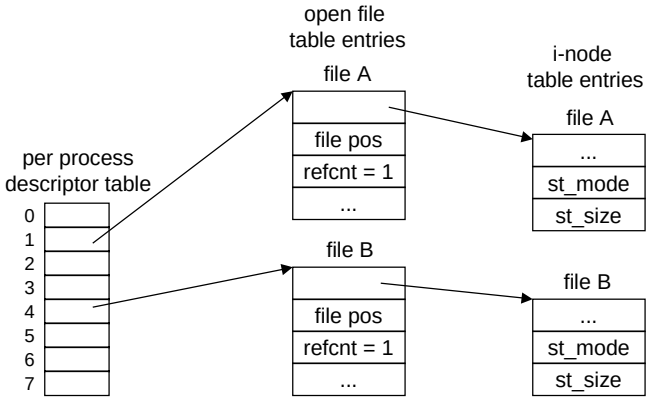


Figure 12.20: **Kernel data structures before** `dup2(4,1)`

The `dup2` function allows multiple descriptors to be associated with the same file. For example, Figure 12.21 shows the situation after calling `dup2(4,1)`. Both descriptors now correspond to file B, file A has been closed, and the reference count for file B has been incremented. From this point on, any data that is written to standard output is redirected to file B.

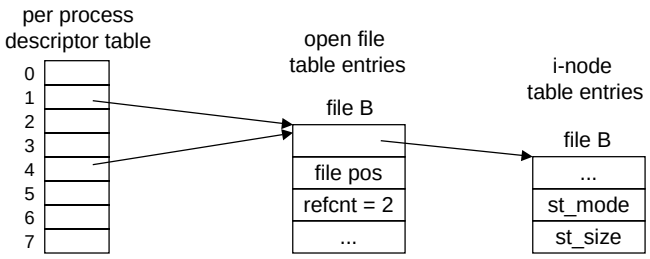


Figure 12.21: **Kernel data structures after** `dup2(4,1)`

12.4.6 The `close` Function

A process informs the kernel that is finished reading and writing a file by calling the `close` function.

```
#include <unistd.h>

int close(int fd);
```

returns: zero if OK, -1 on error

The kernel does not delete the associated file table entry unless the reference count is zero. For example,

suppose we have the situation in Figure 12.21, where descriptors 1 and 4 both point to the same file table entry. If we were to close descriptor 1, then we could still perform input and output on descriptor 4.

Closing a closed descriptor is an error, but unfortunately programmers rarely check for this error. In Section 12.6.2 we will see that threaded programs that close already closed descriptors can suffer from a subtle race condition that sometimes causes a thread to catastrophically close another thread's open descriptor. The bottom line: always check return codes, even for seemingly innocuous functions such as `close`.

12.4.7 Other Unix I/O Functions

Unix provides two additional I/O functions, `open` and `lseek`. The `open` function creates new files and opens existing files. In each case, it returns a descriptor that can be used by other Unix file I/O routines. We will not describe `open` in any more depth because a clear understanding requires numerous details about Unix file systems that are not relevant to network programming.

The `lseek` function modifies the current file position. Since it is illegal to change the current file position of a socket, we will not discuss this function either.

12.4.8 Unix I/O vs. Standard I/O

The ANSII C standard defines a set of input and output functions, known as the *standard I/O library*, that provide a higher-level and more convenient alternative to the underlying Unix I/O functions. Functions such as `fopen`, `fclose`, `fseek`, `fread`, `fwrite`, `fgetc`, `fputc`, `fgets`, `fputs`, `fscanf`, and `fprintf` are commonly used standard I/O functions.

The standard I/O functions are the method of choice for input and output on disk and terminal devices. And in fact, most C programmers use these functions exclusively, never bothering with the the lower-level Unix I/O functions. Unfortunately, standard I/O poses some tricky problems when we attempt to use it for input and output on network sockets.

The standard I/O models a file as a *stream*, which is a higher-level abstraction of a file descriptor. Like descriptors, streams can be full-duplex, so a program can perform input and output on the same stream. However, there are restrictions on full-duplex streams that interact badly with restrictions on sockets:

- *Restriction 1:* An input function cannot follow an output function without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`. For efficiency reasons, standard I/O streams are buffered. Each stream has its own buffer. The first call to a standard I/O input function reads a large block of data from the disk, and stores it in a buffer in main memory. Subsequent requests to read from the stream are served from the buffer rather than disk. The `fflush` function empties the buffer associated with a stream. The latter three functions use the Unix I/O `lseek` function to reset the current file position.
- *Restriction 2:* An output function cannot follow an input function without an intervening call to `fseek`, `fsetpos`, or `rewind`, unless the input function encounters an end-of-file.

These restrictions pose a problem for network applications because it is illegal to use the `lseek` function on a network socket. The first restriction on stream I/O can be worked around by a discipline of flushing

the buffer before every input operation. The only way to work around the second restriction is to open two streams on the same open socket descriptor, one for reading and one for writing.

```
1     FILE *fpin, *fpout;
2
3     fpin = fdopen(sockfd, "r");
4     fpout = fdopen(sockfd, "w");
```

However, this approach has problems as well, because it requires the application to call `fclose` on both streams in order to free the memory resources associated with each stream and avoid a memory leak:

```
1     fclose(fpin);
2     fclose(fpout);
```

Each of these operations attempts to close the same underlying socket descriptor, so the second `close` operation will fail. While this is not necessarily a fatal error in a sequential program, closing the same descriptor twice in a threaded program is a recipe for disaster. Thus, we recommend avoiding the standard I/O functions for input and output on network sockets. Use the robust `readn`, `writen`, and `readline` functions instead.

12.5 The Sockets Interface

The *sockets interface* is a set of functions that are used in conjunction with the Unix file I/O functions to build network applications. It has been implemented on most modern systems, including Linux and the other Unix variants, Windows, and Macintosh systems. Figure 12.22 gives an overview of the sockets interface in the context of a typical client-server transaction. You should use this picture as road map when we discuss the individual functions.

12.5.1 Socket Address Structures

From the perspective of the Unix kernel, a socket is an endpoint for communication. From the perspective of a Unix program, a socket is an open file with a corresponding descriptor.

Internet socket addresses are stored in 16-byte structures of the type `sockaddr_in` shown in Figure 12.23. For Internet applications, the `sin_family` member is AF_INET, the `sin_port` member is a 16-bit port number, and the `sin_addr` member is a 32-bit IP address. The IP address and port number are always stored in network (big-endian) byte order.

Aside: What does the `_in` suffix mean?
The `_in` suffix is short for *internet*, not *input*. **End Aside.**

Aside: Why do we need that `sockaddr` structure?
The generic `sockaddr` structure in Figure 12.23 is an unfortunate historical artifact that confuses many programmers. The sockets interface was designed in the early 1980's to work with any type of underlying network protocol,

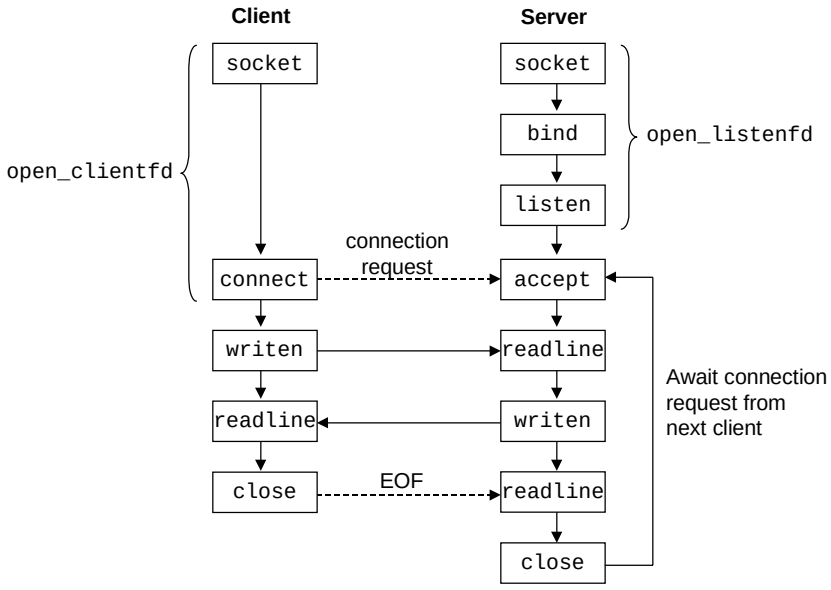


Figure 12.22: **Overview of the sockets interface.**

sockaddr: socketbits.h (included by socket.h). sockaddr_in: netinit/in.h

```
/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    unsigned short  sa_family;    /* protocol family */
    char            sa_data[14];  /* address data.  */
};

/* Internet-style socket address structure */
struct sockaddr_in  {
    unsigned short  sin_family;  /* address family (always AF_INET) */
    unsigned short  sin_port;    /* port number in network byte order */
    struct in_addr  sin_addr;    /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

sockaddr: socketbits.h (included by socket.h). sockaddr_in: netinit/in.h

Figure 12.23: **Socket address structures.** The `in_addr` struct is shown in Figure 12.9.

each of which was expected to define its own 16-byte protocol-specific `sockaddr_xx` socket address structure. No one at the time had any inkling that TCP/IP would become so dominant. **End Aside.**

The `connect`, `bind`, and `accept` functions require a pointer to protocol-specific socket address structure. The problem faced by the designers of the sockets interface was how to define these functions to accept any kind of socket address structure. Today we would use the generic `void *` pointer, which did not exist in C at that time. The solution was to define sockets functions to expect a pointer to a generic `sockaddr` structure, and then require applications to cast pointers to protocol-specific structures to this generic structure.

To simplify our code examples, we will follow Stevens's lead and define the following type

```
1     typedef struct sockaddr SA;
```

that we use whenever we need to cast a protocol-specific structure to a generic one.

12.5.2 The `socket` Function

Clients and servers use the `socket` function to create a *socket descriptor*.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

returns: nonnegative descriptor if OK, -1 on error

In our codes, we will always call the `socket` function with the following arguments:

```
1     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

where AF_INET indicates that we are using the Internet, and SOCK_STREAM indicates that the socket will be an endpoint for an Internet connection. The `sockfd` descriptor returned by `socket` is only partially opened and cannot yet be used for reading and writing. How we finish opening the socket depends on whether we are a client or a server.

12.5.3 The `connect` Function

A client establishes a connection with a server by calling the `connect` function.

```
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen );
```

returns: 0 if OK, -1 on error

The `connect` function attempts to establish an Internet connection with the server at socket address `serv_addr`, where `addrlen` is `sizeof(sockaddr_in)`. The `connect` function blocks until either the connection is successfully established, or an error occurs. If successful, the `sockfd` descriptor is now ready for reading and writing, and the resulting connection is characterized by the socket pair

```
(x:y, serv_addr.sin_addr:serv_addr.sin_port)
```

where `x` is the client's IP address and `y` is the ephemeral port that uniquely identifies the client process on the client host.

Figure 12.24 shows our `open_clientfd` helper function that a client uses to establish a connection with a server running on host `hostname` and listening for connection requests on the well-known port `port`. It returns a file descriptor that is ready for input and output using Unix file I/O.

code/src/csapp.c

```
1  int open_clientfd(char *hostname, int port)
2  {
3      int clientfd;
4      struct hostent *hp;
5      struct sockaddr_in serveraddr;
6
7      clientfd = Socket(AF_INET, SOCK_STREAM, 0);
8
9      /* fill in the server's IP address and port */
10     hp = Gethostbyname(hostname);
11     bzero((char *) &serveraddr, sizeof(serveraddr));
12     serveraddr.sin_family = AF_INET;
13     bcopy((char *)hp->h_addr,
14           (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
15     serveraddr.sin_port = htons(port);
16
17     /* establish a connection with the server */
18     Connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr));
19
20     return clientfd;
21 }
```

code/src/csapp.c

Figure 12.24: `open_clientfd`**: helper function that establishes a connection with a server.**

After creating the socket descriptor (line 11), we retrieve the DNS host entry for the server (line 14) and copy the first IP address in the host entry (which is already in network byte order) to the server's socket address structure (lines 17-18). After initializing the socket address structure with the server's well-known port number in network byte order (line 19), we initiate the connect request to the server (line 22). When `connect` returns, we return the socket descriptor to the client, which can immediately begin using Unix I/O operations to communicate with the server.

12.5.4 The `bind` Function

The remaining functions — `bind`, `listen`, and `accept` — are used by servers to establish connections with clients.

```
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
                                                    returns: 0 if OK, -1 on error
```

The `bind` function tells the kernel to associate the server's socket address in `my_addr` with the socket descriptor `sockfd`. The `addrlen` argument is `sizeof(sockaddr_in)`.

12.5.5 The `listen` Function

Clients are active entities that initiate connection requests. Servers are passive entities that wait for connection requests from clients. By default, the kernel assumes that a descriptor created by the `socket` function corresponds to an *active socket* that will live on the client end of a connection. A server calls the `listen` function to tell the kernel that the descriptor will be used by a server instead of a client.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
                                                    returns: 0 if OK, -1 on error
```

The `listen` function converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients. The `backlog` argument is a hint about the number of outstanding connection requests that the kernel should queue up before it starts to refuse requests. The exact meaning of the `backlog` argument requires an understanding of TCP/IP that is beyond our scope. We will typically set it to a large value, such as 1024.

Figure 12.25 shows our `open_listenfd` helper function that opens and returns a listening socket ready to receive client connection requests on the well-known port `port`. After we create the `listenfd` socket descriptor (line 11), we use the `setsockopt` function (not described here) to configure the server so that it can be terminated and restarted immediately (lines 14-15). By default, a restarted server will deny connection requests from clients for approximately 30 seconds, which seriously hinders debugging.

In lines 20-23, we initialize the server's socket address structure in preparation for calling the `bind` function. In this case, we have used the INADDR_ANY wild card address to tell the kernel that this server will accept requests to any of the IP addresses for this host (line 22), and to well-known port `port` (line 23). Notice that we use the `htonl` and `htons` functions to convert the IP address and port number from host byte order to network byte order. Finally, we convert `listenfd` to a listening descriptor (line 27) and return it to the caller.

code/src/csapp.c

```
1  int open_listenfd(int port)
2  {
3      int listenfd;
4      int optval;
5      struct sockaddr_in serveraddr;
6
7      /* create a socket descriptor */
8      listenfd = Socket(AF_INET, SOCK_STREAM, 0);
9
10     /* eliminates "Address already in use" error from bind. */
11     optval = 1;
12     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
13                (const void *)&optval , sizeof(int));
14
15     /* listenfd will be an endpoint for all requests to port
16        on any IP address for this host */
17     bzero((char *) &serveraddr, sizeof(serveraddr));
18     serveraddr.sin_family = AF_INET;
19     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
20     serveraddr.sin_port = htons((unsigned short)port);
21     Bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr));
22
23     /* make it a listening socket ready to accept connection requests */
24     Listen(listenfd, LISTENQ);
25
26     return listenfd;
27 }
```

code/src/csapp.c

Figure 12.25: `open_listenfd`**: helper function that opens and returns a listening socket.**

12.5.6 The `accept` Function

Servers wait for connection requests from clients by calling the `accept` function.

```
#include <sys/socket.h>

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
                                    returns: nonnegative connected descriptor if OK, -1 on error
```

The `accept` function waits for a connection request from a client to arrive on the listening descriptor `listenfd`, then fills in the client's socket address in `addr`, and returns a *connected descriptor* that can be used to communicate with the client using Unix I/O functions.

The distinction between a listening descriptor and a connected descriptor can be confusing when we first encounter the `accept` function. The listening descriptor serves as an endpoint for client connection requests. It is typically created once and exists for the lifetime of the server. The connected descriptor is the endpoint of the connection that is established between the client and the server. It is created each time the server accepts a connection request and exists only as long as it takes the server to service a client.

Figure 12.26 outlines the roles of the listening and connected descriptors. In Step 1, the server calls `accept`, which waits for a connection request to arrive on the listening descriptor, which for concreteness we will assume is descriptor 3 (recall that descriptors 0–2 are reserved for the standard files).

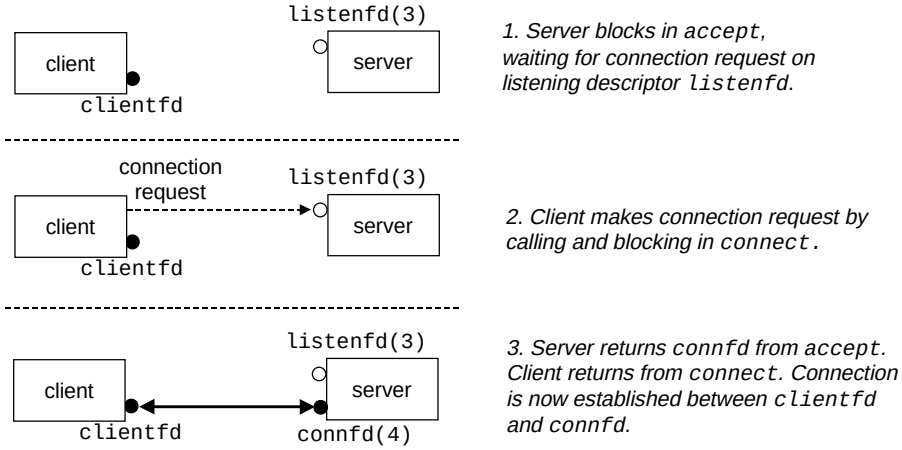


Figure 12.26: **The roles of the listening and connected descriptors.**

In Step 2, the client calls the `connect` function, which sends a connection request to `listenfd`. In Step 3, the `accept` function opens a new connected descriptor `connfd` (which we will assume is descriptor 4), establishes the connection between `clientfd` and `connfd`, and then returns `connfd` to the application. The client also returns from the `connect`, and from this point, the client and server can pass data back and forth by reading and writing `clientfd` and `connfd` respectively.

12.5.7 Example Echo Client and Server

The best way to learn the sockets interface is to study example code. Figure 12.27 shows the code for an echo client. After establishing a connection with the server (line 15), the client enters a loop that repeatedly reads a text from standard input (line 17), sends the text line to the server (line 18), reads the echo line from the server (line 19), and prints the result to standard output (line 20). The loop terminates when `fgets` encounters end-of-file on standard input, either because the user typed `ctrl-d` at the keyboard, or because it has exhausted the text lines in a redirected input file.

code/net/echoclient.c

```
1  #include "csapp.h"
2
3  int main(int argc, char **argv)
4  {
5      int clientfd, port;
6      char *host, buf[MAXLINE];
7
8      if (argc != 3) {
9          fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
10         exit(0);
11     }
12     host = argv[1];
13     port = atoi(argv[2]);
14
15     clientfd = open_clientfd(host, port);
16
17     while (Fgets(buf, MAXLINE, stdin) != NULL) {
18         Writen(clientfd, buf, strlen(buf));
19         Readline(clientfd, buf, MAXLINE);
20         Fputs(buf, stdout);
21     }
22
23     Close(clientfd);
24     exit(0);
25 }
```

code/net/echoclient.c

Figure 12.27: **Echo client main routine.**

After the loop terminates, the client closes the descriptor (line 23). This result in an end-of-file notification being sent to the server, which it detects when it receives a return code of zero from its `readline` function. After closing its descriptor, the client terminates (line 24). Since the client's kernel automatically closes all open descriptors when a process terminates, the `close` in line 23 is not necessary. However, it is good programming practice to explicitly close any descriptors we have opened.

Figure 12.28 shows the main routine for the echo server. After opening the listening descriptor (line 18), it enters an infinite loop. Each iteration waits for a connection request from a client (line 21), prints the

domain name and IP address of the connected client (lines 23-27), and calls the `echo` function that services the client (line 29). When the echo routine returns, the main routine closes the connected descriptor (line 30). Once the client and server have closed their respective descriptors, the connection is terminated.

code/net/echoserveri.c

```
1 #include "csapp.h"
2
3 void echo(int connfd);
4
5 int main(int argc, char **argv)
6 {
7     int listenfd, connfd, port, clientlen;
8     struct sockaddr_in clientaddr;
9     struct hostent *hp;
10    char *haddrp;
11
12    if (argc != 2) {
13        fprintf(stderr, "usage: %s <port>\n", argv[0]);
14        exit(0);
15    }
16    port = atoi(argv[1]);
17
18    listenfd = open_listenfd(port);
19    while (1) {
20        clientlen = sizeof(clientaddr);
21        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
22
23        /* determine the domain name and IP address of the client */
24        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
25                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
26        haddrp = inet_ntoa(clientaddr.sin_addr);
27        printf("server connected to %s (%s)\n", hp->h_name, haddrp);
28
29        echo(connfd);
30        Close(connfd);
31    }
32 }
```

code/net/echoserveri.c

Figure 12.28: **Iterative echo server main routine.**

Figure 12.29 shows the code for the `echo` routine, which repeatedly reads and writes lines of text until the `readline` function encounters end-of-file in line 8.

code/net/echo.c

```
1 #include "csapp.h"
2
3 void echo(int connfd)
4 {
5     size_t n;
6     char buf[MAXLINE];
7
8     while((n = Readline(connfd, buf, MAXLINE)) != 0) {
9         printf("server received %d bytes\n", n);
10        Writen(connfd, buf, n);
11    }
12 }
```

code/net/echo.c

Figure 12.29: `echo` **function that reads and echos text lines.**

12.6 Concurrent Servers

The echo server in Figure 12.28 is known as an *iterative server* because it can only service one client at a time. The disadvantage of iterative servers is that a slow client can preclude every other client from being serviced. For a real server that might be expected to service hundreds or thousands of clients per second, it is unacceptable to allow one slow client to deny service to the others.

A better approach is to build a *concurrent server* that can service multiple clients concurrently. In this section, we will investigate alternative concurrent server designs based on processes and threads.

12.6.1 Concurrent Servers Based on Processes

A concurrent server based on processes accepts connection requests in the parent and forks a separate child process to service each client. For example, suppose we have two clients and a server that is listening for connection requests on a listening descriptor 3. Now suppose that the server accepts a connection request from client 1 and returns connected descriptor 4, as shown in Figure 12.30.

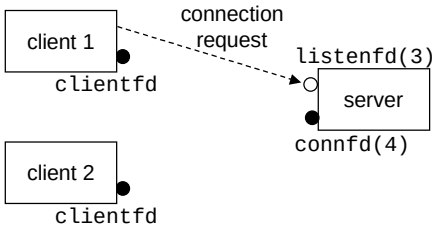


Figure 12.30: **Server accepts connection request from client.**

After accepting the connection request, the server forks a child, which gets a complete copy of the server's

descriptor table. The child closes its copy of listening descriptor 3 and the parent closes its copy of connected descriptor 4, since they will not be needed. This gives us the situation in Figure 12.31, where the child process is busy servicing the client.

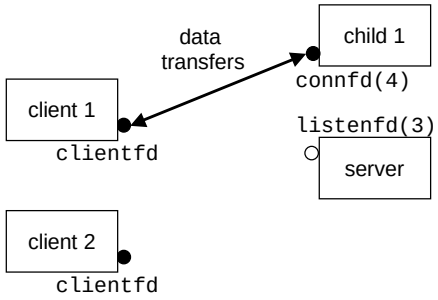


Figure 12.31: **Server forks a child process to service the client.**

Now suppose that after the parent creates the child for client 1, it accepts a new connection request from client 2 and returns a new connected descriptor (say 5), as shown in Figure 12.32.

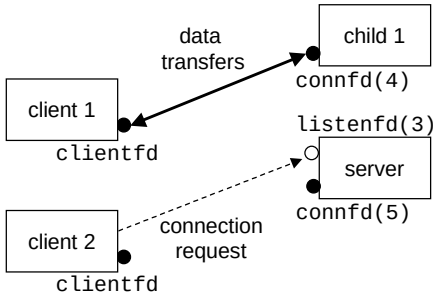


Figure 12.32: **Server accepts another connection request.**

The parent forks another child, which begins servicing its client using connected descriptor 5, as shown in Figure 12.33. At this point, the parent is waiting for the next connection request and the two children are servicing their respective clients.

Figure 12.34 shows the code for a concurrent echo server based on processes. The `echo` function in line 25 is defined in Figure 12.29. There are several points to make about this server.

- Since servers typically run for long periods of time, we must include a SIGCHLD handler that reaps zombie children (lines 8–14). Since SIGCHLD signals are blocked while the SIGCHLD handler is executing, and since Unix signals are not queued, the SIGCHLD handler must be prepared to reap multiple zombie children.
- Notice that the parent and the child close their respective copies of `connfd` (lines 39 and 36 respectively). This especially important for the parent, which must close its copy of the connected descriptor to avoid a memory leak that will eventually crash the system.

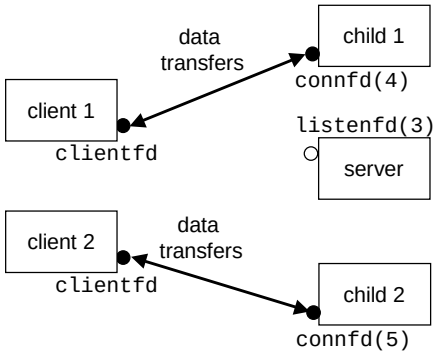


Figure 12.33: **Server forks another child to service the new client.**

- Because of the reference count in the socket's file table entry (Figure 12.21), the socket will not be closed until both the parent's and child's copies of `connfd` are closed.

Discussion

Of the concurrent-server designs that we will study in this section, process-based designs are by far the simplest to write and debug. Processes provide a clean sharing model where descriptors are shared and user address spaces are not. As long as we remember to reap zombie children and close the parent's connected descriptor each time we create a new child, the children run independently of each other and the parent and can be debugged in isolation.

Process-based designs do have disadvantages though. If a particular service requires processes to share state information such as a memory-resident file cache, performance statistics that are aggregated across all processes, or aggregate request logs, then we must use explicit IPC mechanisms such as FIFO's, System V shared memory, or System V semaphores (none of which are discussed here). Another disadvantage is that process-based servers tend to be slower than other designs because the overhead for process control and IPC is relatively high. Nonetheless, the simplicity of process-based designs provides a powerful attraction.

Practice Problem 12.5:

After the parent closes the connected descriptor in line 39 of the concurrent server in Figure 12.34, the child is still able to communicate with the client using its copy of the descriptor. Why?

Practice Problem 12.6:

If we were to delete line 36 of Figure 12.34 that closes the connected descriptor, the code would still be correct, in the sense that there would be no memory leak. Why?

12.6.2 Concurrent Servers Based on Threads

Another approach to building concurrent servers is to use threads instead of processes. There are several advantages to using threads. First, threads have less run time overhead than processes. We would expect a

code/net/echoserverp.c

```
1  #include "csapp.h"
2
3  void echo(int connfd);
4
5  /* SIGCHLD signal handler */
6  void handler(int sig)
7  {
8      pid_t pid;
9      int stat;
10
11     while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
12         ;
13     return;
14 }
15
16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd, port, clientlen;
19     struct sockaddr_in clientaddr;
20
21     if (argc != 2) {
22         fprintf(stderr, "usage: %s <port>\n", argv[0]);
23         exit(0);
24     }
25     port = atoi(argv[1]);
26
27     Signal(SIGCHLD, handler);
28
29     listenfd = open_listenfd(port);
30     while (1) {
31         clientlen = sizeof(clientaddr);
32         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
33         if (Fork() == 0) {
34             Close(listenfd); /* child closes its listening socket */
35             echo(connfd);    /* child services client */
36             Close(connfd);   /* child closes connection with client */
37             exit(0);         /* child exits */
38         }
39         Close(connfd); /* parent closes connected socket (important!) */
40     }
41 }
```

code/net/echoserverp.c

Figure 12.34: **Concurrent echo server based on processes.**

server based on threads to have better throughput (measured in clients serviced per second) than one based on processes. Second, because all threads share the same global variables and heap variables, it is much easier for threads to share state information.

The major disadvantage of using threads is that the same memory model that makes it easy to share data structures also makes it easy to share data structures unintentionally and incorrectly. As we learned in Chapter 11, shared data must be protected, functions called from threads must be reentrant, and race conditions must be avoided.

The threaded echo server in Figure 12.35 illustrates some of the subtle issues that can arise. The overall structure is similar to the process-based design. The main thread repeatedly waits for a connection request (line 22) and then creates a peer thread to handle the request (line 23).

The first issue we encounter is how to pass the connected descriptor to the peer thread when we call `pthread_create`. The obvious approach is to pass a pointer to the descriptor:

```
1     connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
2     Pthread_create(&tid, NULL, thread, &connfd);
```

and then let the peer thread dereference the pointer and assign it to a local variable.

```
1 void *thread(void *vargp)
2 {
3     int connfd = *((int *)vargp);
4     /* ... */
5     return NULL;
6 }
```

However, this would be wrong because it introduces a race between the assignment statement in the peer thread and the `accept` statement in the main thread. If the assignment statement completes before the next `accept`, then the local `connfd` variable in the peer thread gets the correct descriptor value. However, if the assignment completes *after* the `accept`, then the local `confd` variable in the peer thread gets the descriptor number of the *next* connection. The unhappy result is two threads are now performing input and output on the same descriptor. In order to avoid the potentially deadly race, we must assign each connected descriptor returned by `accept` to its own dynamically allocated memory block, as shown in lines 21-22.

Now consider the thread routine in lines 28-38. To avoid memory leaks, we must detach the thread so that its memory resources will be reclaimed when it terminates (line 32), and we must free the memory block that was allocated by the main thread (line 33). Finally, the thread routine calls the `echo_r` function (line 35) before terminating in line 37.

So why do we call `echo_r` instead of the trusty `echo` function? The `echo` function calls the `readline` function (Figure 12.16, which in turn calls the `my_read` function (Figure 12.16), which maintains three `static` variables, and thus is not reentrant. Since `my_read` is not reentrant, neither are `readline` or `echo`.

To build a correct threaded echo server, we must use a reentrant version of `echo` called `echo_r`, which is based on the `readline_r` function, a reentrant version of the `readline` function developed by Stevens [77].

code/net/echoservert.c

```
1 #include "csapp.h"
2
3 void echo_r(int connfd);
4 void *thread(void *vargp);
5
6 int main(int argc, char **argv)
7 {
8     int listenfd, *connfdp, port, clientlen;
9     struct sockaddr_in clientaddr;
10    pthread_t tid;
11
12    if (argc != 2) {
13        fprintf(stderr, "usage: %s <port>\n", argv[0]);
14        exit(0);
15    }
16    port = atoi(argv[1]);
17
18    listenfd = open_listenfd(port);
19    while (1) {
20        clientlen = sizeof(clientaddr);
21        connfdp = Malloc(sizeof(int));
22        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
23        Pthread_create(&tid, NULL, thread, connfdp);
24    }
25 }
26
27 /* thread routine */
28 void *thread(void *vargp)
29 {
30    int connfd = *((int *)vargp);
31
32    Pthread_detach(pthread_self());
33    Free(vargp);
34
35    echo_r(connfd); /* reentrant version of echo() */
36    Close(connfd);
37    return NULL;
38 }
```

code/net/echoservert.c

Figure 12.35: **Concurrent echo server based on threads.**

```
#include "csapp.h"

ssize_t readline_r(Rline *rptr);                    returns: number of bytes read (0 if EOF), -1 on error
```

The `readline` function takes as input an `Rline` structure shown in Figure 12.36. The first three members correspond to the arguments that users pass to `readline`. The next three members correspond to the static variables that `my_read` uses for buffering.

code/include/csapp.h

```
1 typedef struct {
2     int read_fd;        /* caller's descriptor to read from */
3     char *read_ptr;     /* caller's buffer to read into */
4     size_t read_maxlen; /* max bytes to read */
5
6     /* next three are used internally by the function */
7     int rl_cnt;         /* initialize to 0 */
8     char *rl_bufptr;    /* initialize to rl_buf */
9     char rl_buf[MAXBUF];/* internal buffer */
10 } Rline;
```

code/include/csapp.h

Figure 12.36: `Rline` **structure used by** `readline_r` **and initialized by** `readline_rinit`**.**

The `Rline` structure is initialized by the `readline_rinit` function in Figure 12.37, which saves the user arguments and initializes the internal buffering information.

code/src/csapp.c

```
1 void readline_rinit(int fd, void *ptr, size_t maxlen, Rline *rptr)
2 {
3     rptr->read_fd = fd;         /* save caller's arguments */
4     rptr->read_ptr = ptr;
5     rptr->read_maxlen = maxlen;
6
7     rptr->rl_cnt = 0;           /* and init our counter & pointer */
8     rptr->rl_bufptr = rptr->rl_buf;
9 }
```

code/src/csapp.c

Figure 12.37: `readline_rint`**: Initialization function for** `readline_r`**.**

Figure 12.38 shows the code for the `readline_r` package. The only difference between `readline_r` and `readline` is that `readline_r` calls `my_read_r` instead of `my_read` in line 28. The `my_read_r` function is similar to the original `my_read` function, except that it references members of the `Rline` struct instead of static variables.

code/src/csapp.c

```
1  static ssize_t my_read_r(Rline *rptr, char *ptr)
2  {
3      if (rptr->rl_cnt <= 0) {
4      again:
5          rptr->rl_cnt = read(rptr->read_fd, rptr->rl_buf,
6                              sizeof(rptr->rl_buf));
7          if (rptr->rl_cnt < 0) {
8              if (errno == EINTR)
9                  goto again;
10             else
11                 return(-1);
12         }
13         else if (rptr->rl_cnt == 0)
14             return(0);
15         rptr->rl_bufptr = rptr->rl_buf;
16     }
17     rptr->rl_cnt--;
18     *ptr = *rptr->rl_bufptr++ & 255;
19     return(1);
20 }
21
22 ssize_t readline_r(Rline *rptr)
23 {
24     int n, rc;
25     char c, *ptr = rptr->read_ptr;
26
27     for (n = 1; n < rptr->read_maxlen; n++) {
28         if ( (rc = my_read_r(rptr, &c)) == 1) {
29             *ptr++ = c;
30             if (c == '\n')
31                 break;
32         } else if (rc == 0) {
33             if (n == 1)
34                 return(0);      /* EOF, no data read */
35             else
36                 break;          /* EOF, some data was read */
37         } else
38             return(-1); /* error */
39     }
40     *ptr = 0;
41     return(n);
42 }
```

code/src/csapp.c

Figure 12.38: `readline_r` **package: Reentrant version of** `readline`**.** Adapted from [77].

Given the reentrant `readline_r` function, we can now create a reentrant version of `echo` (Figure 12.39) that calls `readline_r` instead of `readline`.

code/net/echo_r.c

```
1 #include "csapp.h"
2
3 void echo_r(int connfd)
4 {
5     size_t n;
6     char buf[MAXLINE];
7     Rline rline;
8
9     readline_rinit(connfd, buf, MAXLINE, &rline);
10    while((n = Readline_r(&rline)) != 0) {
11        printf("server received %d bytes\n", n);
12        Writen(connfd, buf, n);
13    }
14 }
```

code/net/echo_r.c

Figure 12.39: `echo_r`**: Reentrant version of** `echo`

Discussion

Threaded designs are attractive because they promise better performance than process-based designs. But the performance gain can come with a steep price in complexity. Unlike processes, which share almost nothing, threads share almost everything. Because of this, it easy to write incorrect threaded programs that suffer from races, unprotected shared variables, and non-reentrant functions. These bugs are extremely difficult to find because they are usually non-deterministic, and thus not easily repeatable. Nothing is scarier to a programmer than a random non-repeatable bug.

The subtle issues involved in threading our simple echo server are clear evidence of the potential complexity of threaded designs. Nonetheless, if a process-based design would be unacceptably slow for a particular application, then we might need to opt for a threaded design.

12.7 Web Servers

So far we have discussed network programming in the context of a simple echo server. In this section, we will show you how to use the basic ideas of network programming, Unix I/O, and Unix processes to build your own your small, but functional Web server.

12.7.1 Web Basics

Web clients and servers interact using a text-based application-level protocol known as *HTTP (Hypertext Transfer Protocol)*. HTTP is a simple protocol. A Web client (known as a *browser*) opens an Internet connection to a server and requests some *content*. The server responds with the requested content and then closes the connection. The browser reads the content and displays it on the screen.

What makes the Web so different from conventional file retrieval services such as FTP? The main reason is that the content can be written in a programming language known as *HTML (Hypertext Markup Language)*. An HTML program (page) contains instructions (tags) that tell the browser how to to display various text and graphical objects in the page. For example,

```
<b> Make me bold! </b>
```

tells the browser to print the text between the `<b>` and `</b>` tags in boldface type. However, the real power of HTML is that a page can contain pointers (hyperlinks) to content stored on remote servers anywhere in the Internet. For example,

```
<a href="http://www.cmu.edu/index.html">Carnegie Mellon</a>
```

tells the browser to highlight the text object "Carnegie Mellon" and to create a hyperlink to an HTML file called `index.html` that is stored on the CMU Web server. If the user clicks on the highlighted text object, the browser requests the corresponding HTML file from the CMU server and displays it.

12.7.2 Web Content

To Web clients and servers, *content* is a sequence of bytes with an associated *MIME* (Multipurpose Internet Mail Extensions) type. Figure 12.40 shows some common MIME types.

MIME type	Description
text/html	HTML page
text/plain	Unformatted text
application/postscript	Postscript document
image/gif	Binary image encoded in GIF format
image/jpg	Binary image encoded in JPG format

Figure 12.40: **Example MIME types.**

Web servers provide content to clients in two different ways:

- Fetch a disk file and return its contents to the client. The disk file is known as *static content* and the process of returning the file to the client is known as *serving static content*.
- Run an executable file and return its output to the client. The output produced by the executable at runtime is known as *dynamic content*, and the process of running the program and returning its output to the client is known as *serving dynamic content*.

Thus, every piece of content returned by a Web server is associated with some file that it manages. Each of these files has a unique name known as a *URL* (Universal Resource Locator). For example, the URL

```
http://www.aol.com:80/index.html
```

identifies an HTML file called `/index.html` on Internet host `www.aol.com` that is managed by a Web server listening on port 80. The port number is optional and defaults to well-known port 80.

URLs for executable files can include program arguments after the filename. A '?' character separates the filename from the arguments, and each argument is separated by a '&' character. For example, the URL

```
http://kittyhawk.cmcl.cs.cmu.edu:8000/cgi-bin/adder?15000&213
```

identifies an executable called `/cgi-bin/adder` that will be called with two argument strings: `15000` and `213`.

Clients and servers use different parts of the URL during a transaction. For example, a client uses the prefix

```
http://www.aol.com:80
```

to determine what kind of server to contact, where the server is, and what port it is listening on. The server uses the suffix

```
/index.html
```

to find the file on its filesystem, and to determine whether the request is for static or dynamic content. There are several important points to understand about how servers interpret the suffix of a URL:

- There are no standard rules for determining whether a URL refers to static or dynamic content. Each server has its own rules for the files that it manages. A common approach is to identify a set of directories, such as `cgi-bin`, where all executables must reside.
- The initial '/' in the suffix does *not* denote the Unix root directory. Rather is denotes the home directory for whatever kind of content is being requested. For example, a server might by configured so that all static content is stored in directory `/usr/httpd/html` and all dynamic content is stored in directory `/usr/httpd/cgi-bin`.
- The minimal URL suffix is the '/' character, which all servers expand to some default home page such as `/index.html`. This explains why it is possible to fetch the home page of a site by simply typing a domain name to the browser. The browser appends the missing '/' to the URL and passes it to the server, which expands the '/' to some default file name.

12.7.3 HTTP Transactions

Since HTTP is based on text lines transmitted over Internet connections, we can use the Unix TELNET program to conduct transactions with any Web server on the Internet. The TELNET program is very handy for debugging servers that talk to clients with text lines over connections. For example, Figure 12.41 uses TELNET to request the home page from the AOL Web server.

```
 1 unix> telnet www.aol.com 80          Client: open connection to server
 2 Trying 205.188.146.23...              Telnet prints 3 lines to the terminal
 3 Connected to aol.com.
 4 Escape character is '^]'.
 5 GET / HTTP/1.1                        Client: request line
 6 host: www.aol.com                     Client: required HTTP/1.1 header
 7                                       Client: empty line terminates headers.
 8 HTTP/1.0 200 OK                       Server: response line
 9 MIME-Version: 1.0                     Server: followed by five response headers
10 Date: Mon, 08 Jan 2001 04:59:42 GMT
11 Server: NaviServer/2.0 AOLserver/2.3.3
12 Content-Type: text/html               Server: expect HTML in the response body
13 Content-Length: 42092                 Server: expect 42,092 bytes in the response body
14                                       Server: empty line terminates response headers
15 <html>                                Server: first HTML line in response body
16 ...                                   Server: 766 lines of HTML not shown.
17 </html>                               Server: last HTML line in response body
18 Connection closed by foreign host.    Server: closes connection
19 unix>                                Client: closes connection and terminates
```

Figure 12.41: **An HTTP transaction that serves static content.**

In line 1 we run TELNET from a Unix shell and ask it to open a connection to the AOL Web server. TELNET prints three lines of output to the terminal, opens the connection, and then waits for us to enter text (line 5). Each time we enter a text line and hit the `enter` key, TELNET reads the line, appends carriage return and line feed characters (`"\r\n"` in C notation), and sends the line to the server. This is consistent with the HTTP standard, which requires every text line to be terminated by a carriage return and line feed pair. To initiate the transaction, we enter an HTTP request (lines 5-7). The server replies with an HTTP response (lines 8-17) and then closes the connection (line 18).

HTTP Requests

An *HTTP request* consists of a *request line* (line 5), followed by zero or more *request headers* (line 6), followed by an empty text line that terminates the list of headers (line 7). A request line has the form

```
<method> <uri> <version>
```

HTTP supports a number of different *methods*, including GET, POST, OPTIONS, HEAD, PUT, DELETE, and TRACE). We will only discuss the workhorse GET method, which according to one study accounts for over 99% of HTTP requests [75]. The GET method instructs the server to generate and return the content identified by the *URI* (Uniform Resource Identifier). The URI is the suffix of the corresponding URL that includes the file name and optional arguments. [1]

[1] Actually, this is only true when a browser requests content. If a proxy server requests content, then the URI must be the complete URL.

The `<version>` field in the request line indicates the HTTP version that the request conforms to. The current version is HTTP/1.1 [25]. HTTP/1.0 is a previous version from 1996 that is still in use [3]. HTTP/1.1 defines additional headers that provide support for advanced features such as caching and security, as well as a (seldom used) mechanism that allows a client and server to perform multiple transactions over the same *persistent connection*. In practice, the two versions are compatible because HTTP/1.0 clients and servers simply ignore unknown HTTP/1.1 headers.

In sum, the request line in line 5 asks the server to fetch and return the HTML file `/index.html`. It also informs the server that the remainder of the request will be in HTTP/1.1 format.

Request headers provide additional information to the server, such as the brand name of the browser or the MIME types that the browser understands. Request headers have the form

```
<header name>: <header data>
```

For our purposes, the only header we need to be concerned with is the `Host` header (line 5), which is required in HTTP/1.1 requests, but not in HTTP/1.0 requests. The `Host` header is only used by *proxy caches*, which sometimes serve as intermediaries between a browser and the *origin server* that manages the requested file. Multiple proxies can exist between a client and an origin server in a so-called *proxy chain*. The data in the `Host` header, which identifies the domain name of the origin server, allows a proxy in the middle of a proxy chain to determine if it might have a locally cached copy of the requested content.

Continuing with our example in Figure 12.41, the empty text line in line 6 (generated by hitting `enter` on our keyboard) terminates the headers and instructs the server to send the requested HTML file.

HTTP Responses

HTTP responses are similar to HTTP requests. An *HTTP response* consists of a *response line* (line 8), followed by zero or more *response headers* (lines 9-13), followed by an empty line that terminates the headers (line 14), followed by the *response body* (lines 15-17).

A response line has the form

```
<version>  <status code> <status message>
```

The version field describes the HTTP version that the response conforms to. The *status code* is a 3-digit positive integer that indicates the disposition of the request. The *status message* gives the English equivalent of the error code. Figure 12.42 lists some common status codes and their corresponding messages.

The response headers in lines 9-13 provide additional information about the response. The two most important headers are `Content-Type` (line 12), which tells the client the MIME type of the content in the response body, and `Content-Length` (line 13), which indicates its size in byte.

The empty text line in line 11 that terminates the response headers is followed by the request body, which contains the requested content.

Status code	Status Message	Description
200	OK	Request was handled without error.
301	Moved permanently	Content has moved to the hostname in the Location header.
400	Bad request	Request could not be understood by the server.
403	Forbidden	Server lacks permission to access the requested file.
404	Not found	Server could not find the requested file.
501	Not implemented	Server does not support the request method.
505	HTTP version not supported	Server does not support version in request.

Figure 12.42: **Some HTTP status codes.**

12.7.4 Serving Dynamic Content

If we stop to think for a moment how a server might provide dynamic content to a client, certain questions arise. For example, how does the client pass any program arguments to the server? How does the server pass these arguments to the child process that it creates? How does the server pass other information to the child that it might need to generate the content? Where does the child send its output? These questions are addressed by a de facto standard called *CGI (Common Gateway Interface)*.

How Does the Client Pass Program Arguments to the Server?

Arguments for GET requests are passed in the URI. Each argument is separated by a '&' character. Spaces are not allowed in arguments and must be denoted with the `%20` string. Similar encodings exist for other special characters.

> **Aside: Passing arguments in HTTP POST requests.**
> Arguments for HTTP POST requests are passed in the request body rather than the URI. **End Aside.**

How Does the Server Pass Arguments to the Child?

After a server receives a request such as

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

it calls `fork` to creates a child process and calls `execve` to run the `/cgi-bin/adder` program in the context of the child. The `adder` program is often referred to as *CGI program* because it obeys the rules of the CGI standard. And since many CGI programs are written as Perl scripts, CGI programs are often called *CGI scripts*.

Before the call to `execve`, the child process sets the CGI environment variable QUERY_STRING to `15000&213`, which the `adder` program can reference at runtime using the Unix `getenv` function.

How Does the Server Pass Other Information to the Child?

CGI defines a number of other environment variables that a CGI program can expect to be set when it runs. Figure 12.43 shows a subset.

Environment variable	Description
SERVER_PORT	Port that the parent is listening on
REQUEST_METHOD	GET or POST
REMOTE_HOST	Domain name of client
REMOTE_ADDR	Dotted-decimal IP address of client
CONTENT_TYPE	POST only: MIME type of the request body
CONTENT_LENGTH	POST only: Size in bytes of the request body

Figure 12.43: **Examples of CGI environment variables.**

Where Does the Child Send its Output?

A CGI program prints dynamic content to the standard output. Before the child process loads and runs the CGI program, it uses the Unix `dup2` function to redirect standard output to the connected descriptor that is associated with the client. Thus, anything that the CGI program writes to standard output goes directly to the client.

> **Aside: Passing arguments to HTTP POST requests.**
> For POST requests, the child would also need to redirect standard input to the connected descriptor. The CGI program would then read the arguments in the request body from standard input. **End Aside.**

Notice that since the parent does not know the type or size of the content that the child generates, the child is responsible for generating the `Content-type` and `Content-length` response headers, as well as the empty line that terminates the headers.

Figure 12.44 shows a simple CGI program that sums its two arguments and returns an HTML file with the result to the client. Figure 12.45 shows an HTTP transaction that serves dynamic content from the `adder` program.

> **Practice Problem 12.7**:
>
> In Section 12.4.8, we warned about the dangers of using the C standard I/O functions in servers. Yet the CGI program in Figure 12.44 is able to use standard I/O without any problems. Why?

12.8 Putting it Together: The TINY Web Server

We will conclude our discussion of network programming by developing a small but functioning Web server called TINY. TINY is an interesting program. It combines many of the ideas that we have learned about concurrency, Unix I/O, the sockets interface, and HTTP in only 250 lines of code. While it lacks the functionality, robustness, and security of a real server, it is powerful enough to serve both static and dynamic content to real Web browsers. We encourage you to study it and implement it yourself. It is quite exciting (even for the authors!) to point a real browser at your own server and watch it display a complicated Web page with text and graphics.

code/net/tiny/cgi-bin/adder.c

```
#include "csapp.h"

int main(void) {
    char *buf, *p;
    char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
    int n1=0, n2=0;

    /* extract the two arguments */
    if ((buf = getenv("QUERY_STRING")) != NULL) {
        p = strchr(buf, '&');
        *p = '\0';
        strcpy(arg1, buf);
        strcpy(arg2, p+1);
        n1 = atoi(arg1);
        n2 = atoi(arg2);
    }

    /* make the response body */
    sprintf(content, "Welcome to add.com: ");
    sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
    sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
            content, n1, n2, n1 + n2);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* generate the HTTP response */
    printf("Content-length: %d\r\n", strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
    fflush(stdout);
    exit(0);
}
```

code/net/tiny/cgi-bin/adder.c

Figure 12.44: **CGI program that sums two integers.**

```
1 unix> telnet kittyhawk.cmcl.cs.cmu.edu 8000   Client: open connection
2 Trying 128.2.194.242...
3 Connected to kittyhawk.cmcl.cs.cmu.edu.
4 Escape character is '^]'.
5 GET /cgi-bin/adder?15000&213 HTTP/1.0   Client: request line
6                                         Client: empty line terminates headers
7 HTTP/1.0 200 OK                         Server: response line
8 Server: Tiny Web Server                 Server: identify server
9 Content-length: 115                     Adder: expect 115 bytes in response body
10 Content-type: text/html                Adder: expect HTML in response body
11                                        Adder: empty line terminates headers
12 Welcome to add.com: THE Internet addition portal.  Adder: first HTML line
13 <p>The answer is: 15000 + 213 = 15213   Adder: second HTML line in response body
14 <p>Thanks for visiting!                 Adder: third HTML line in response body
15 Connection closed by foreign host.      Server: closes connection
16 unix>                                  Client: closes connection and terminates
```

Figure 12.45: **An HTTP transaction that serves dynamic HTML content.**

The TINY `main` Routine

Figure 12.46 shows TINY's main routine. TINY is an iterative server that listens for connection requests on the port that is passed in the command line. After opening a listening socket (line 28) by calling the `open_listenfd` function from Figure 12.46, TINY executes the typical infinite server loop, repeatedly accepting a connection request (line 31) and performing a transaction (line 32).

The `doit` Function

The `doit` function in Figure 12.47 handles one HTTP transaction. First, we read and parse the request line (lines 9-10). Notice that we are using the robust `readline` function from Figure 12.16 to read the request line.

TINY only supports the GET method. If the client requests another method (such as POST), we send it an error message and return to the main routine (lines 11-15), which then closes the connection and awaits the next connection request. Otherwise, we read and (as we shall see) ignore any request headers (line 16).

Next, we parse the URI into a filename and a possibly empty CGI argument string, and we set a flag that indicates whether the request is for static or dynamic content (line 19). If the file does not exist on disk, we immediately send an error message to the client and return (lines 20-24).

Finally, if the request is for static content (lines 26), we verify that the file is a regular file (i.e., not a directory file or a FIFO) and that we have read permission (line 27). If so, we serve the static content (line 32) to the client. Similarly, if the request is for dynamic content (line 34), we verify that the file is executable (line 35), and if so we go ahead and serve the dynamic content (line 40).

code/net/tiny/tiny.c

```
1  /*
2   * tiny.c - A simple HTTP/1.0 Web server that uses the GET method
3   *          to serve static and dynamic content.
4   */
5  #include "csapp.h"
6
7  void doit(int fd);
8  void read_requesthdrs(int fd);
9  int parse_uri(char *uri, char *filename, char *cgiargs);
10 void serve_static(int fd, char *filename, int filesize);
11 void get_filetype(char *filename, char *filetype);
12 void serve_dynamic(int fd, char *filename, char *cgiargs);
13 void clienterror(int fd, char *cause, char *errnum,
14                  char *shortmsg, char *longmsg);
15
16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd, port, clientlen;
19     struct sockaddr_in clientaddr;
20
21     /* check command line args */
22     if (argc != 2) {
23         fprintf(stderr, "usage: %s <port>\n", argv[0]);
24         exit(1);
25     }
26     port = atoi(argv[1]);
27
28     listenfd = open_listenfd(port);
29     while (1) {
30         clientlen = sizeof(clientaddr);
31         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
32         doit(connfd);
33         Close(connfd);
34     }
35 }
```

code/net/tiny/tiny.c

Figure 12.46: **The TINY Web server.**

code/net/tiny/tiny.c

```
1  void doit(int fd)
2  {
3      int is_static;
4      struct stat sbuf;
5      char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
6      char filename[MAXLINE], cgiargs[MAXLINE];
7
8      /* read request line and headers */
9      Readline(fd, buf, MAXLINE);
10     sscanf(buf, "%s %s %s\n", method, uri, version);
11     if (strcasecmp(method, "GET")) {
12        clienterror(fd, method, "501", "Not Implemented",
13                    "Tiny does not implement this method");
14         return;
15     }
16     read_requesthdrs(fd);
17
18     /* parse URI from GET request */
19     is_static = parse_uri(uri, filename, cgiargs);
20     if (stat(filename, &sbuf) < 0) {
21         clienterror(fd, filename, "404", "Not found",
22                     "Tiny couldn't find this file");
23         return;
24     }
25
26     if (is_static) { /* serve static content */
27         if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
28             clienterror(fd, filename, "403", "Forbidden",
29                         "Tiny couldn't read the file");
30             return;
31         }
32         serve_static(fd, filename, sbuf.st_size);
33     }
34     else { /* serve dynamic content */
35         if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
36             clienterror(fd, filename, "403", "Forbidden",
37                         "Tiny couldn't run the CGI program");
38             return;
39         }
40         serve_dynamic(fd, filename, cgiargs);
41     }
42 }
```

code/net/tiny/tiny.c

Figure 12.47: TINY `doit`**: Handles one HTTP transaction.**

The `clienterror` Function

TINY lacks many of the robustness features of a real server. However it does check for some obvious errors and reports them to the client. The `clienterror` function in Figure 12.48 sends an HTTP response to the client with the appropriate status code and status message in the response line, along with an HTML file in the response body that explains the error to the browser's user.

code/net/tiny/tiny.c

```
1  void clienterror(int fd, char *cause, char *errnum,
2                   char *shortmsg, char *longmsg)
3  {
4      char buf[MAXLINE], body[MAXBUF];
5
6      /* build the HTTP response body */
7      sprintf(body, "<html><title>Tiny Error</title>");
8      sprintf(body, "%s<body bgcolor=""ffffff"">\r\n", body);
9      sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10     sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11     sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);
12
13     /* print the HTTP response */
14     sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15     Writen(fd, buf, strlen(buf));
16     sprintf(buf, "Content-type: text/html\r\n");
17     Writen(fd, buf, strlen(buf));
18     sprintf(buf, "Content-length: %d\r\n\r\n", strlen(body));
19     Writen(fd, buf, strlen(buf));
20     Writen(fd, body, strlen(body));
21 }
```

code/net/tiny/tiny.c

Figure 12.48: TINY `clienterror`**: Sends an error message to the client.**

Recall that an HTML response should indicate the size and type the content in the body. Thus, we have opted to build the HTML content as a single string (lines 7-11) so that we can easily determine its size (line 18). Also, notice that we are using the robust `writen` function from Figure 12.15 for all output.

The `read_requesthdrs` Function

TINY does not use any of the information in the request headers. It simply reads and ignores them by calling the `read_requesthdrs` function in Figure 12.49. Notice that the empty text line that terminates the request headers consists of a carriage return and line feed pair, which we check for in line 6.

code/net/tiny/tiny.c

```
1 void read_requesthdrs(int fd)
2 {
3     char buf[MAXLINE];
4
5     Readline(fd, buf, MAXLINE);
6     while(strcmp(buf, "\r\n"))
7         Readline(fd, buf, MAXLINE);
8     return;
9 }
```

code/net/tiny/tiny.c

Figure 12.49: TINY `read_requesthdrs`**: Reads and ignores request headers.**

The `parse_uri` Function

TINY assumes that the home directory for static content is the current Unix directory `'.'`, and that the home directory for executables is `./cgi-bin`. Any URI that contains the string `cgi-bin` is assumed to denote a request for dynamic content. The default file name is `./home.html`.

The `parse_uri` function in Figure 12.50 implements these policies. It parses the URI into a filename and an optional CGI argument string. If the request is for static content (line 5) we clear the CGI argument string (line 6), and then convert the URI into a relative Unix pathname such as `./index.html` (lines 7-8). If the URI ends with a `'/'` character (line 9), then we append the default file name (lines 9). On the other hand, if the request is for dynamic content (line 13), we extract any CGI arguments (line 14-20) and convert the remaining portion of the URI to a relative Unix file name (lines 21-22).

The `serve_static` Function

TINY serves 4 different types of static content: HTML files, unformatted text files, and images encoded in GIF and JPG formats. These file types account for the majority of static content served over the Web.

The `serve_static` function in Figure 12.51 sends an HTTP response whose body contains the contents of a local file. First, we determine the file type by inspecting the suffix in the filename (line 7), and then send the response line and response headers to the client (lines 6-12). Notice that we are using the `writen` function from Figure 12.15 for all output on the descriptor. Notice also that a blank line terminates the headers (line 12).

Next, we send the response body by copying the contents of the requested file to the connected descriptor `fd` (lines 15-19). The code here is somewhat subtle and needs to be studied carefully.

Line 15 opens `filename` for reading and gets its descriptor. In line 16, the Unix `mmap` function maps the requested file to a virtual memory area. Recall from our discussion of `mmap` in Section 10.8 that the call to `mmap` maps the first `filesize` bytes of file `srcfd` to a private read-only area of virtual memory that starts at address `srcp`.

Once we have mapped the file to memory, we no longer need its descriptor, so we close the file (line 17).

code/net/tiny/tiny.c

```
1  int parse_uri(char *uri, char *filename, char *cgiargs)
2  {
3      char *ptr;
4
5      if (!strstr(uri, "cgi-bin")) {  /* static content */
6          strcpy(cgiargs, "");
7          strcpy(filename, ".");
8          strcat(filename, uri);
9          if (uri[strlen(uri)-1] == '/')
10             strcat(filename, "home.html");
11         return 1;
12     }
13     else {  /* dynamic content */
14         ptr = index(uri, '?');
15         if (ptr) {
16             strcpy(cgiargs, ptr+1);
17             *ptr = '\0';
18         }
19         else
20             strcpy(cgiargs, "");
21         strcpy(filename, ".");
22         strcat(filename, uri);
23         return 0;
24     }
25 }
```

code/net/tiny/tiny.c

Figure 12.50: TINY `parse_uri`**: Parses an HTTP URI.**

code/net/tiny/tiny.c

```
1  void serve_static(int fd, char *filename, int filesize)
2  {
3      int srcfd;
4      char *srcp, filetype[MAXLINE], buf[MAXBUF];
5
6      /* send response headers to client */
7      get_filetype(filename, filetype);
8      sprintf(buf, "HTTP/1.0 200 OK\r\n");
9      sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10     sprintf(buf, "%sContent-length: %d\n", buf, filesize);
11     sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
12     Writen(fd, buf, strlen(buf));
13
14     /* send response body to client */
15     srcfd = Open(filename, O_RDONLY, 0);
16     srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
17     Close(srcfd);
18     Writen(fd, srcp, filesize);
19     Munmap(srcp, filesize);
20 }
21
22 /*
23  * get_filetype - derive file type from file name
24  */
25 void get_filetype(char *filename, char *filetype)
26 {
27     if (strstr(filename, ".html"))
28         strcpy(filetype, "text/html");
29     else if (strstr(filename, ".gif"))
30         strcpy(filetype, "image/gif");
31     else if (strstr(filename, ".jpg"))
32         strcpy(filetype, "image/jpg");
33     else
34         strcpy(filetype, "text/plain");
35 }
```

code/net/tiny/tiny.c

Figure 12.51: TINY `serve_static`**: Serves static content to a client.**

Failing to do this would introduce a potentially fatal memory leak.

Line 18 performs the actual transfer of the file to the client. The `writen` function copies the `filesize` bytes starting at location `srcp` (which of course is mapped to the requested file) to the client's connected descriptor. Finally, line 19 frees the mapped virtual memory area. This is important to avoid a potentially fatal memory leak.

The `serve_dynamic` Function

TINY serves any type of dynamic content by forking a child process, and then running a CGI program in the context of the child.

The `serve_dynamic` function in Figure 12.52 begins by sending a response line indicating success to the client (lines 6-7), along with an informational `Server` header (lines 8-9). The CGI program is responsible for sending the rest of the response. Notice that this is not as robust as we might wish, since it doesn't allow for the possibility that the CGI program might encounter some error.

code/net/tiny/tiny.c

```
1  void serve_dynamic(int fd, char *filename, char *cgiargs)
2  {
3      char buf[MAXLINE];
4
5      /* return first part of HTTP response */
6      sprintf(buf, "HTTP/1.0 200 OK\r\n");
7      Writen(fd, buf, strlen(buf));
8      sprintf(buf, "Server: Tiny Web Server\r\n");
9      Writen(fd, buf, strlen(buf));
10
11     if (Fork() == 0) { /* child */
12         /* real server would set all CGI vars here */
13         setenv("QUERY_STRING", cgiargs, 1);
14         Dup2(fd, STDOUT_FILENO);         /* redirect output to client */
15         Execve(filename, NULL, environ); /* run CGI program */
16     }
17     Wait(NULL); /* parent reaps child */
18 }
```

code/net/tiny/tiny.c

Figure 12.52: TINY `serve_dynamic`**: Serves dynamic content to a client.**

After sending the first part of the response, we fork a new child process (line 11). The child initializes the QUERY_STRING environment variable with the CGI arguments from the request URI (line 13). Notice that a real server would set the other CGI environment variables here as well. For brevity, we have omitted this step.

Next, the child redirects the child's standard output to the connected file descriptor (line 14), and then loads and runs the CGI program (line 15). Since the CGI program runs in the context of the child, it has access to

the same open descriptors and environment variables that existed before the call to the `execve` function. Thus, everything that the CGI program writes to standard output goes directly to the client process, without any intervention from the parent process.

Meanwhile, the parent blocks in a call to `wait`, waiting to reap the child when it terminates (line 17).

Practice Problem 12.8:

A. Is the TINY `doit` routine reentrant? Why or why not?

B. If not, how would you make it reentrant?

12.9 Summary

In this chapter we have learned some basic concepts about network applications. Network applications use the client-server model, where servers perform services on behalf of their clients. The Internet provides network applications with two key mechanisms: (1) A unique name for each Internet host, and (2) a mechanism for establishing a connection to a server running on any of those hosts. Clients and servers establish connections by using the sockets interface, and they communicate over these connections using standard Unix file I/O functions.

There are two basic design options for servers. An iterative server handles one request at a time. A concurrent server can handle multiple requests concurrently. We investigated two designs for concurrent servers, one that forks a new process for each request, the other that creates a new thread for each request. Other designs are possible, such as using the Unix `select` function to explicitly manage the concurrency, or avoiding the per-connection overhead by pre-forking a set of child processes to handle connection requests.

Finally, we studied the design and implementation of a simple but functional Web server. In a few lines of code, it ties together many important systems concepts such as Unix I/O, memory mapping, concurrency, the sockets interface, and the HTTP protocol.

Bibliographic Notes

The official source information for the Internet is contained in a set of freely-available numbered documents known as *RFCs* (Requests for Comments). A searchable index of RFCs is available from

`http://www.rfc-editor.org/rfc.html`

RFCs are typically written for developers of Internet infrastructure, and thus are usually too detailed for the casual reader. However, for authoritative information, there is no better source.

There are many texts on computer networking [41, 55, 80]. The great technical writer W. Richard Stevens developed a whole series of classic texts on such topics as advanced Unix programming [72], the Internet protocols [73, 74, 75], and Unix network programming [77, 76]. Serious students of Unix systems programming will want to study all of them. Tragically, Stevens died in 1999. His contributions will be greatly missed.

The authoritative list of MIME types is maintained at

```
ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types
```

The HTTP/1.1 protocol is documented in RFC 2616.

Homework Problems

Homework Problem 12.9 [Category 2]:

Modify the `cpstdinbuf` program in Figure 12.14 so that it uses `readn` and `writen` to copy standard input to standard output, MAXBUF bytes at a time.

Homework Problem 12.10 [Category 2]:

A. Modify TINY so that it echos every request line and request header.

B. Use your favorite browser to make a request to TINY for static content. Capture the output from TINY in a file.

C. Inspect the output from TINY to determine the the version of HTTP your browser uses.

D. Consult the HTTP/1.1 standard in RFC 2616 to determine the meaning of each header in the HTTP request from your browser. You can obtain RFC 2616 from `www.rfc-editor.org/rfc.html`.

Homework Problem 12.11 [Category 2]:

Extend TINY to so that it serves MPG video files. Check your work using a real browser.

Homework Problem 12.12 [Category 2]:

Modify TINY so that its reaps CGI children inside a SIGCHLD handler instead of explicitly waiting for them to terminate.

Homework Problem 12.13 [Category 2]:

Modify TINY so that when it serves static content, it copies the requested file to the connected descriptor using `malloc`, `read`, and `write`, instead of `mmap` and `write`.

Homework Problem 12.14 [Category 2]:

A. Write an HTML form for the CGI `adder` function in Figure 12.44. Your form should include two text boxes that users will fill in with the two numbers they want to add together. Your form should also request content using the GET method.

B. Check your work by using a real browser to request the form from TINY, submit the filled in form to TINY, and then display the the dynamic content generated by `adder`.

Homework Problem 12.15 [Category 2]:

Extend TINY to support the HTTP HEAD method. Check your work using TELNET as a Web client.

Homework Problem 12.16 [Category 3]:

Extend TINY so that it serves dynamic contest requested by the HTTP POST method. Check your work using your favorite Web browser.

Homework Problem 12.17 [Category 3]:

Build a concurrent TINY server based on processes.

Homework Problem 12.18 [Category 3]:

Build a concurrent TINY server based on threads.

Homework Problem 12.19 [Category 4]:

Build your own concurrent Web proxy cache.

Appendix A

Error handling

A.1 Introduction

Programmers should *always* check the error codes returned by system-level functions. There are many subtle ways that things can go wrong, and it only makes sense to use the status information that the kernel is able to provide us. Unfortunately, programmers are often reluctant to do error checking because it clutters their code, turning a single line of code into a multi-line conditional statement. Error checking is also confusing because different functions indicate errors in different ways.

We were faced with a similar problem when writing this text. On the one hand, we would like our code examples to be concise and simple to read. On the other hand, we do not want to give students the wrong impression that it is OK to skip error checking. To resolve these issues, we have adopted an approach based on *error-handling wrappers* that was pioneered by W. Richard Stevens in his classic network programming text [77].

The idea is that given some base system-level function `foo`, we define a wrapper function `Foo` with identical arguments, but with the first letter capitalized. The wrapper calls the base function and checks for errors. If it detects an error, the wrapper prints an informative message and terminates the process. Otherwise it returns to the caller. Notice that if there are no errors, the wrapper behaves exactly like the base function. Put another way, if a program runs correctly with wrappers, it will run correctly if we lower-case the first letter of each wrapper and recompile.

The wrappers are packaged in a single source file (`csapp.c`) that is compiled and linked into each program. A separate header file (`csapp.h`) contains the function prototypes for the wrappers.

This appendix gives a tutorial on the different kinds of error-handling in Unix systems and gives examples of the different styles of error-handling wrappers. For reference, we also include the complete sources for the `csapp.h` and `csapp.c` files.

A.2 Error handling in Unix systems

The systems-level function calls that we will encounter in this book use three different styles for returning errors: *Unix-style*, *Posix-style*, and *DNS-style*.

Unix-style error handling

Functions such as `fork` and `wait` that were developed in the early days of Unix (as well as some older Posix functions) overload the function return value with both error codes *and* useful results. For example, when the Unix-style `wait` function encounters an error (e.g., there is no child process to reap) it returns -1 and sets the global variable `errno` to an error code that indicate the cause of the error. If `wait` completes successfully, then it returns the useful result, which is the PID of the reaped child. Unix-style error-handling code is typically of the form:

```
1     if ((pid = wait(NULL)) < 0) {
2         fprintf(stderr, "wait error: %s\n", strerror(errno));
3         exit(0);
4     }
```

The `strerror` function returns a text description for a particular value of `errno`.

Posix-style error handling

Many of the newer Posix functions such as Pthreads use the return value only to indicate success (0) or failure (nonzero). Any useful results are returned in function arguments that are passed by reference. We refer to this approach as *Posix-style error handling*. For example, the Posix-style `pthread_create` function indicates success or failure with its return value and returns the ID of the newly created thread (the useful result) by reference in its first argument. Posix-style error-handling code is typically of the form:

```
1     if ((retcode = pthread_create(&tid, NULL, thread, NULL)) != 0) {
2         fprintf(stderr, "pthread_create error: %s\n", strerror(retcode));
3         exit(0);
4     }
5
```

DNS-style error handling

The `gethostbyname` and `gethostbyaddr` functions that retrieve DNS (Domain Name System) host entries have yet another approach for returning errors. These functions return a NULL pointer on failure and set the global `h_errno` variable. DNS-style error handling is typically of the form:

```
1     if ((p = gethostbyname(name)) == NULL) {
2         fprintf(stderr, "gethostbyname error: %s\n:", hstrerror(h_errno));
3         exit(0);
4     }
```

The `hstrerror` function returns a text description for a particular value of `h_errno`.

Summary of error-reporting functions

Thoughout this book, we use the following error-reporting functions to accomodate different error-handling styles.

```
#include "csapp.h"

void unix_error(char *msg);
void posix_error(int code, char *msg);
void dns_error(char *msg);
void app_error(char *msg);
```

return: nothing

As their names suggest, the `unix_error`, `posix_error`, and `dns_error` functions report Unix-style errors, Posix-style, and DNS-style errors and then terminate. The `app_error` function is included as a convenience for application errors. It simply prints its input and then terminates. Figure A.1 shows the code for the error reporting functions.

A.3 Error-handling wrappers

Here are some examples of the different error-handling wrappers.

Unix-style error-handling wrappers

Figure A.2 shows the wrapper for the Unix-style `wait` function. If the `wait` returns with an error, the wrapper prints an informative message and then exits. Otherwise, it returns a PID to the caller.

Figure A.3 shows the wrapper for the Unix-style `kill` function. Notice that this function, unlike `Wait`, returns `void` on success.

Posix-style error-handling wrappers

Figure A.4 shows the wrapper for the Posix-style `pthread_mutex_lock` function. Like most Posix-style functions, it does not overload useful results with error return codes, so the wrapper returns `void` on success.

One exception is the Posix-style `pthread_cond_timedwait` which returns an error code of ETIMEDOUT if the call times out. Since this particular return code is useful to applications, the wrapper passes it back to the caller, as shown in Figure A.5.

code/src/csapp.c

```
1 void unix_error(char *msg) /* unix-style error */
2 {
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4     exit(0);
5 }
6
7 void posix_error(int code, char *msg) /* posix-style error */
8 {
9     fprintf(stderr, "%s: %s\n", msg, strerror(code));
10    exit(0);
11 }
12
13 void dns_error(char *msg) /* dns-style error */
14 {
15    fprintf(stderr, "%s: %s\n", msg, hstrerror(h_errno));
16    exit(0);
17 }
18
19 void app_error(char *msg) /* application error */
20 {
21    fprintf(stderr, "%s\n", msg);
22    exit(0);
23 }
```

code/src/csapp.c

Figure A.1: **Error-reporting functions.**

code/src/csapp.c

```
1 pid_t Wait(int *status)
2 {
3     pid_t pid;
4
5     if ((pid  = wait(status)) < 0)
6         unix_error("Wait error");
7     return pid;
8 }
```

code/src/csapp.c

Figure A.2: **Wrapper for Unix-style `wait` function.**

code/src/csapp.c

```
1 void Kill(pid_t pid, int signum)
2 {
3     int rc;
4
5     if ((rc = kill(pid, signum)) < 0)
6         unix_error("Kill error");
7 }
```

code/src/csapp.c

Figure A.3: **Wrapper for Unix-style** `kill` **function.**

code/src/csapp.c

```
1 void Pthread_mutex_lock(pthread_mutex_t *mutex)
2 {
3     int rc;
4
5     if ((rc = pthread_mutex_lock(mutex)) != 0)
6         posix_error(rc, "Pthread_mutex_lock error");
7 }
```

code/src/csapp.c

Figure A.4: **Wrapper for Posix-style** `pthread_mutex_lock` **function.**

code/src/csapp.c

```
1 int Pthread_cond_timedwait(pthread_cond_t *cond,
2                            pthread_mutex_t *mutex,
3                            struct timespec *abstime)
4 {
5     int rc = pthread_cond_timedwait(cond, mutex, abstime);
6
7     if ((rc != 0) && (rc != ETIMEDOUT))
8         posix_error(rc, "Pthread_cond_timedwait error");
9     return rc;
10 }
```

code/src/csapp.c

Figure A.5: **Wrapper for Posix-style** `pthread_cond_timedwait` **function.**

DNS-style error-handling wrappers

Figure A.6 shows the error-handling wrapper for the DNS-style `gethostbyname` function.

code/src/csapp.c

```
1 struct hostent *Gethostbyname(const char *name)
2 {
3     struct hostent *p;
4
5     if ((p = gethostbyname(name)) == NULL)
6         dns_error("Gethostbyname error");
7     return p;
8 }
```

code/src/csapp.c

Figure A.6: **Wrapper for DNS-style `gethostbyname` function.**

A.4 The csapp.h header file

code/include/csapp.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <setjmp.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <errno.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* Simplifies calls to bind(), connect(), and accept() */
typedef struct sockaddr SA;

/* External variables */
extern int h_errno;    /* defined by BIND for DNS errors */
extern char **environ; /* defined by libc */

/* Misc constants */
#define MAXLINE  8192  /* max text line length */
#define MAXBUF   8192  /* max I/O buffer size */
#define LISTENQ  1024  /* second argument to listen() */

/* Our own error-handling functions */
void unix_error(char *msg);
void posix_error(int code, char *msg);
void dns_error(char *msg);
void app_error(char *msg);

/* Process control wrappers */
pid_t Fork(void);
void Execve(const char *filename, char *const argv[], char *const envp[]);
pid_t Wait(int *status);
pid_t Waitpid(pid_t pid, int *iptr, int options);
void Kill(pid_t pid, int signum);
```

```
47 unsigned int Sleep(unsigned int secs);
48 void Pause(void);
49 unsigned int Alarm(unsigned int seconds);
50 void Setpgid(pid_t pid, pid_t pgid);
51 pid_t Getpgrp();
52
53 /* Sigaction wrapper */
54 typedef void handler_t(int);
55 handler_t *Signal(int signum, handler_t *handler);
56
57 /* Unix I/O wrappers */
58 int Open(const char *pathname, int flags, mode_t mode);
59 ssize_t Read(int fd, void *buf, size_t count);
60 ssize_t Write(int fd, const void *buf, size_t count);
61 off_t Lseek(int fildes, off_t offset, int whence);
62 void Close(int fd);
63 int Select(int  n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
64            struct timeval *timeout);
65 void Dup2(int fd1, int fd2);
66
67 /* Memory mapping wrappers */
68 void *Mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
69 void Munmap(void *start, size_t length);
70
71 /* Standard I/O wrappers */
72 void Fclose(FILE *fp);
73 FILE *Fdopen(int fd, const char *type);
74 char *Fgets(char *ptr, int n, FILE *stream);
75 FILE *Fopen(const char *filename, const char *mode);
76 void Fputs(const char *ptr, FILE *stream);
77 size_t Fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
78 void Fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
79
80 /* Dynamic storage allocation wrappers */
81 void *Malloc(size_t size);
82 void *Calloc(size_t nmemb, size_t size);
83 void Free(void *ptr);
84
85 /* Thread control wrappers */
86 void Pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
87                     void * (*routine)(void *), void *argp);
88 void Pthread_join(pthread_t tid, void **thread_return);
89 void Pthread_cancel(pthread_t tid);
90 void Pthread_detach(pthread_t tid);
91 void Pthread_exit(void *retval);
92 pthread_t Pthread_self(void);
93
94 /* Semaphore wrappers */
95 void Sem_init(sem_t *sem, int pshared, unsigned int value);
96 void P(sem_t *sem);
```

```
97 void V(sem_t *sem);
98
99 /* Mutex wrappers */
100 void Pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
101 void Pthread_mutex_lock(pthread_mutex_t *mutex);
102 void Pthread_mutex_unlock(pthread_mutex_t *mutex);
103
104 /* Condition variable wrappers */
105 void Pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
106 void Pthread_cond_signal(pthread_cond_t *cond);
107 void Pthread_cond_broadcast(pthread_cond_t *cond);
108 void Pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
109 int Pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
110                            struct timespec *abstime);
111
112 /* Sockets interface wrappers */
113 int Socket(int domain, int type, int protocol);
114 void Setsockopt(int s, int level, int optname, const void *optval, int optlen);
115 void Bind(int sockfd, struct sockaddr *my_addr, int addrlen);
116 void Listen(int s, int backlog);
117 int Accept(int s, struct sockaddr *addr, int *addrlen);
118 void Connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
119
120 /* DNS wrappers */
121 struct hostent *Gethostbyname(const char *name);
122 struct hostent *Gethostbyaddr(const char *addr, int len, int type);
123
124 /* Stevens's socket I/O functions (UNP, Sec 3.9) */
125 ssize_t readn(int fd, void *vptr, size_t n);
126 ssize_t writen(int fd, const void *vptr, size_t n);
127 ssize_t readline(int fd, void *vptr, size_t maxlen); /* non-reentrant */
128
129 /*
130  * Stevens's reentrant readline_r package
131  */
132 /* struct used by readline_r */
133 typedef struct {
134     int read_fd;        /* caller's descriptor to read from */
135     char *read_ptr;     /* caller's buffer to read into */
136     size_t read_maxlen; /* max bytes to read */
137
138     /* next three are used internally by the function */
139     int rl_cnt;         /* initialize to 0 */
140     char *rl_bufptr;    /* initialize to rl_buf */
141     char rl_buf[MAXBUF];/* internal buffer */
142 } Rline;
143
144 void readline_rinit(int fd, void *ptr, size_t maxlen, Rline *rptr);
145 ssize_t readline_r(Rline *rptr);
146
```

```
147 /* Wrappers for Stevens's socket I/O helpers */
148 ssize_t Readn(int fd, void *vptr, size_t n);
149 void Writen(int fd, void *vptr, size_t n);
150 ssize_t Readline(int fd, void *vptr, size_t maxlen);
151 ssize_t Readline_r(Rline *);
152 void Readline_rinit(int fd, void *ptr, size_t maxlen, Rline *rptr);
153
154 /* Our own client/server helper functions */
155 int open_clientfd(char *hostname, int portno);
156 int open_listenfd(int portno);
```

code/include/csapp.h

A.5 The csapp.c source file

code/src/csapp.c

```
#include "csapp.h"

/**************************
 * Error-handling functions
 **************************/
void unix_error(char *msg) /* unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}

void posix_error(int code, char *msg) /* posix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(code));
    exit(0);
}

void dns_error(char *msg) /* dns-style error */
{
    fprintf(stderr, "%s: %s\n", msg, hstrerror(h_errno));
    exit(0);
}

void app_error(char *msg) /* application error */
{
    fprintf(stderr, "%s\n", msg);
    exit(0);
}

/****************************************
 * Wrappers for Unix process control functions
 ****************************************/

pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}

void Execve(const char *filename, char *const argv[], char *const envp[])
{
    if (execve(filename, argv, envp) < 0)
        unix_error("Execve error");
```

```
47 }
48
49 pid_t Wait(int *status)
50 {
51     pid_t pid;
52
53     if ((pid  = wait(status)) < 0)
54         unix_error("Wait error");
55     return pid;
56 }
57
58 pid_t Waitpid(pid_t pid, int *iptr, int options)
59 {
60     pid_t retpid;
61
62     if ((retpid  = waitpid(pid, iptr, options)) < 0)
63         unix_error("Waitpid error");
64     return(retpid);
65 }
66
67 handler_t *Signal(int signum, handler_t *handler)
68 {
69     struct sigaction action, old_action;
70
71     action.sa_handler = handler;
72     sigemptyset(&action.sa_mask); /* block sigs of type being handled */
73     action.sa_flags = SA_RESTART; /* restart syscalls if possible */
74
75     if (sigaction(signum, &action, &old_action) < 0)
76         unix_error("Signal error");
77     return (old_action.sa_handler);
78 }
79
80 void Kill(pid_t pid, int signum)
81 {
82     int rc;
83
84     if ((rc = kill(pid, signum)) < 0)
85         unix_error("Kill error");
86 }
87
88 void Pause()
89 {
90     (void)pause();
91     return;
92 }
93
94 unsigned int Sleep(unsigned int secs)
95 {
96     unsigned int rc;
```

```
 97
 98     if ((rc = sleep(secs)) < 0)
 99         unix_error("Sleep error");
100     return rc;
101 }
102
103 unsigned int Alarm(unsigned int seconds) {
104     return alarm(seconds);
105 }
106
107 void Setpgid(pid_t pid, pid_t pgid) {
108     int rc;
109
110     if ((rc = setpgid(pid, pgid)) < 0)
111         unix_error("Setpgid error");
112     return;
113 }
114
115 pid_t Getpgrp(void) {
116     return getpgrp();
117 }
118
119 /********************************
120  * Wrappers for Unix I/O routines
121  ********************************/
122
123 int Open(const char *pathname, int flags, mode_t mode)
124 {
125     int rc;
126
127     if ((rc = open(pathname, flags, mode))  < 0)
128         unix_error("Open error");
129     return rc;
130 }
131
132 ssize_t Read(int fd, void *buf, size_t count)
133 {
134     ssize_t rc;
135
136     if ((rc = read(fd, buf, count)) < 0)
137         unix_error("Read error");
138     return rc;
139 }
140
141 ssize_t Write(int fd, const void *buf, size_t count)
142 {
143     ssize_t rc;
144
145     if ((rc = write(fd, buf, count)) < 0)
146         unix_error("Write error");
```

```
147     return rc;
148 }
149
150 off_t Lseek(int fildes, off_t offset, int whence)
151 {
152     off_t rc;
153
154     if ((rc = lseek(fildes, offset, whence)) < 0)
155         unix_error("Lseek error");
156     return rc;
157 }
158
159 void Close(int fd)
160 {
161     int rc;
162
163     if ((rc = close(fd)) < 0)
164         unix_error("Close error");
165 }
166
167 int Select(int  n, fd_set *readfds, fd_set *writefds,
168            fd_set *exceptfds, struct timeval *timeout)
169 {
170     int rc;
171
172     if ((rc = select(n, readfds, writefds, exceptfds, timeout)) < 0)
173         unix_error("Select error");
174     return rc;
175 }
176
177 void Dup2(int fd1, int fd2)
178 {
179     if (dup2(fd1, fd2) == -1)
180         unix_error("dup2 error");
181 }
182
183 /************************************
184  * Wrapper for memory mapping functions
185  ************************************/
186 void *Mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
187 {
188     void        *ptr;
189
190     if ((ptr = mmap(addr, len, prot, flags, fd, offset)) == ((void *) -1))
191         unix_error("mmap error");
192     return(ptr);
193 }
194
195 void Munmap(void *start, size_t length)
196 {
```

```
197     if (munmap(start, length) < 0)
198         unix_error("munmap error");
199 }
200
201 /*********************************************
202  * Wrappers for dynamic storage allocation functions
203  *********************************************/
204
205 void *Malloc(size_t size)
206 {
207     void *p;
208
209     if ((p  = malloc(size)) == NULL)
210         unix_error("Malloc error");
211     return p;
212 }
213
214 void *Calloc(size_t nmemb, size_t size)
215 {
216     void *p;
217
218     if ((p = calloc(nmemb, size)) == NULL)
219         unix_error("Calloc error");
220     return p;
221 }
222
223 void Free(void *ptr)
224 {
225     free(ptr);
226 }
227
228 /*********************************************************
229  * Error-handling wrappers for the Standard I/O functions.
230  *********************************************************/
231 void Fclose(FILE *fp)
232 {
233     if (fclose(fp) != 0)
234         unix_error("Fclose error");
235 }
236
237 FILE *Fdopen(int fd, const char *type)
238 {
239     FILE *fp;
240
241     if ((fp = fdopen(fd, type)) == NULL)
242         unix_error("Fdopen error");
243
244     return fp;
245 }
246
```

```
247 char *Fgets(char *ptr, int n, FILE *stream)
248 {
249     char *rptr;
250
251     if (((rptr = fgets(ptr, n, stream)) == NULL) && ferror(stream))
252         app_error("Fgets error");
253
254     return rptr;
255 }
256
257 FILE *Fopen(const char *filename, const char *mode)
258 {
259     FILE *fp;
260
261     if ((fp = fopen(filename, mode)) == NULL)
262         unix_error("Fopen error");
263
264     return fp;
265 }
266
267 void Fputs(const char *ptr, FILE *stream)
268 {
269     if (fputs(ptr, stream) == EOF)
270         unix_error("Fputs error");
271 }
272
273 size_t Fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
274 {
275     size_t n;
276
277     if (((n = fread(ptr, size, nmemb, stream)) < nmemb) && ferror(stream))
278         unix_error("Fread error");
279     return n;
280 }
281
282 void Fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
283 {
284     if (fwrite(ptr, size, nmemb, stream) < nmemb)
285         unix_error("Fwrite error");
286 }
287
288
289 /*********************************************
290  * Wrappers for Pthreads thread control functions
291  ********************************************/
292
293 void Pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
294                     void * (*routine)(void *), void *argp)
295 {
296     int rc;
```

```
297
298     if ((rc = pthread_create(tidp, attrp, routine, argp)) != 0)
299         posix_error(rc, "Pthread_create error");
300 }
301
302 void Pthread_cancel(pthread_t tid) {
303     int rc;
304
305     if ((rc = pthread_cancel(tid)) != 0)
306         posix_error(rc, "Pthread_cancel error");
307 }
308
309 void Pthread_join(pthread_t tid, void **thread_return) {
310     int rc;
311
312     if ((rc = pthread_join(tid, thread_return)) != 0)
313         posix_error(rc, "Pthread_join error");
314 }
315
316 void Pthread_detach(pthread_t tid) {
317     int rc;
318
319     if ((rc = pthread_detach(tid)) != 0)
320         posix_error(rc, "Pthread_detach error");
321 }
322
323 void Pthread_exit(void *retval) {
324     pthread_exit(retval);
325 }
326
327 pthread_t Pthread_self(void) {
328     return pthread_self();
329 }
330
331 /*********************************************************
332  * Wrappers for Pthreads mutex and condition variable functions
333  *********************************************************/
334
335 void Pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)
336 {
337     int rc;
338
339     if ((rc = pthread_mutex_init(mutex, attr)) != 0)
340         posix_error(rc, "Pthread_mutex_init error");
341 }
342
343 void Pthread_mutex_lock(pthread_mutex_t *mutex)
344 {
345     int rc;
346
```

```
347     if ((rc = pthread_mutex_lock(mutex)) != 0)
348         posix_error(rc, "Pthread_mutex_lock error");
349 }
350
351 void Pthread_mutex_unlock(pthread_mutex_t *mutex)
352 {
353     int rc;
354
355     if ((rc = pthread_mutex_unlock(mutex)) != 0)
356         posix_error(rc, "Pthread_mutex_unlock error");
357 }
358
359 void Pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr)
360 {
361     int rc;
362
363     if ((rc = pthread_cond_init(cond, attr)) != 0)
364         posix_error(rc, "Pthread_cond_init error");
365 }
366
367 void Pthread_cond_signal(pthread_cond_t *cond)
368 {
369     int rc;
370
371     if ((rc = pthread_cond_signal(cond)) != 0)
372         posix_error(rc, "Pthread_cond_signal error");
373 }
374
375 void Pthread_cond_broadcast(pthread_cond_t *cond)
376 {
377     int rc;
378
379     if ((rc = pthread_cond_broadcast(cond)) != 0)
380         posix_error(rc, "Pthread_cond_broadcast error");
381 }
382
383 void Pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
384 {
385     int rc;
386
387     if ((rc = pthread_cond_wait(cond, mutex)) != 0)
388         posix_error(rc, "Pthread_cond_wait error");
389 }
390
391 int Pthread_cond_timedwait(pthread_cond_t *cond,
392                            pthread_mutex_t *mutex,
393                            struct timespec *abstime)
394 {
395     int rc = pthread_cond_timedwait(cond, mutex, abstime);
396
```

```
397     if ((rc != 0) && (rc != ETIMEDOUT))
398         posix_error(rc, "Pthread_cond_timedwait error");
399     return rc;
400 }
401
402 /*******************************
403  * Wrappers for Posix semaphores
404  *******************************/
405
406 void Sem_init(sem_t *sem, int pshared, unsigned int value)
407 {
408     if (sem_init(sem, pshared, value) < 0)
409         unix_error("Sem_init error");
410 }
411
412 void P(sem_t *sem)
413 {
414     if (sem_wait(sem) < 0)
415         unix_error("P error");
416 }
417
418 void V(sem_t *sem)
419 {
420     if (sem_post(sem) < 0)
421         unix_error("V error");
422 }
423
424 /****************************
425  * Sockets interface wrappers
426  ****************************/
427
428 int Socket(int domain, int type, int protocol)
429 {
430     int rc;
431
432     if ((rc = socket(domain, type, protocol)) < 0)
433         unix_error("Socket error");
434     return rc;
435 }
436
437 void Setsockopt(int s, int level, int optname, const void *optval, int optlen)
438 {
439     int rc;
440
441     if ((rc = setsockopt(s, level, optname, optval, optlen)) < 0)
442         unix_error("Setsockopt error");
443 }
444
445 void Bind(int sockfd, struct sockaddr *my_addr, int addrlen)
446 {
```

```
447     int rc;
448
449     if ((rc = bind(sockfd, my_addr, addrlen)) < 0)
450         unix_error("Bind error");
451 }
452
453 void Listen(int s, int backlog)
454 {
455     int rc;
456
457     if ((rc = listen(s,  backlog)) < 0)
458         unix_error("Listen error");
459 }
460
461 int Accept(int s, struct sockaddr *addr, int *addrlen)
462 {
463     int rc;
464
465     if ((rc = accept(s, addr, addrlen)) < 0)
466         unix_error("Accept error");
467     return rc;
468 }
469
470 void Connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
471 {
472     int rc;
473
474     if ((rc = connect(sockfd, serv_addr, addrlen)) < 0)
475         unix_error("Connect error");
476 }
477
478 /************************
479  * DNS interface wrappers
480  ************************/
481
482 struct hostent *Gethostbyname(const char *name)
483 {
484     struct hostent *p;
485
486     if ((p = gethostbyname(name)) == NULL)
487         dns_error("Gethostbyname error");
488     return p;
489 }
490
491 struct hostent *Gethostbyaddr(const char *addr, int len, int type)
492 {
493     struct hostent *p;
494
495     if ((p = gethostbyaddr(addr, len, type)) == NULL)
496         dns_error("Gethostbyaddr error");
```

```
497     return p;
498 }
499
500
501 /*********************************
502  * Client/server helper functions
503  *********************************/
504 /*
505  * open_clientfd - open connection to server at <hostname, port>
506  *   and return a socket descriptor ready for reading and writing.
507  */
508 int open_clientfd(char *hostname, int port)
509 {
510     int clientfd;
511     struct hostent *hp;
512     struct sockaddr_in serveraddr;
513
514     clientfd = Socket(AF_INET, SOCK_STREAM, 0);
515
516     /* fill in the server's IP address and port */
517     hp = Gethostbyname(hostname);
518     bzero((char *) &serveraddr, sizeof(serveraddr));
519     serveraddr.sin_family = AF_INET;
520     bcopy((char *)hp->h_addr,
521           (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
522     serveraddr.sin_port = htons(port);
523
524     /* establish a connection with the server */
525     Connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr));
526
527     return clientfd;
528 }
529
530 /*
531  * open_listenfd - open and return a listening socket on port
532  */
533 int open_listenfd(int port)
534 {
535     int listenfd;
536     int optval;
537     struct sockaddr_in serveraddr;
538
539     /* create a socket descriptor */
540     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
541
542     /* eliminates "Address already in use" error from bind. */
543     optval = 1;
544     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
545                (const void *)&optval , sizeof(int));
546
```

```
547     /* listenfd will be an endpoint for all requests to port
548        on any IP address for this host */
549     bzero((char *) &serveraddr, sizeof(serveraddr));
550     serveraddr.sin_family = AF_INET;
551     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
552     serveraddr.sin_port = htons((unsigned short)port);
553     Bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr));
554 
555     /* make it a listening socket ready to accept connection requests */
556     Listen(listenfd, LISTENQ);
557 
558     return listenfd;
559 }
560 
561 /*****************************************
562  * I/O helper functions (from Stevens UNP)
563  *****************************************/
564 ssize_t readn(int fd, void *buf, size_t count)
565 {
566     size_t nleft = count;
567     ssize_t nread;
568     char *ptr = buf;
569 
570     while (nleft > 0) {
571         if ((nread = read(fd, ptr, nleft)) < 0) {
572             if (errno == EINTR)
573                 nread = 0;      /* and call read() again */
574             else
575                 return -1;      /* errno set by read() */
576         }
577         else if (nread == 0)
578             break;              /* EOF */
579         nleft -= nread;
580         ptr += nread;
581     }
582     return (count - nleft);    /* return >= 0 */
583 }
584 
585 ssize_t writen(int fd, const void *buf, size_t count)
586 {
587     size_t nleft = count;
588     ssize_t nwritten;
589     const char *ptr = buf;
590 
591     while (nleft > 0) {
592         if ((nwritten = write(fd, ptr, nleft)) <= 0) {
593             if (errno == EINTR)
594                 nwritten = 0;    /* and call write() again */
595             else
596                 return -1;       /* errorno set by write() */
```

```
597         }
598         nleft -= nwritten;
599         ptr += nwritten;
600     }
601     return count;
602 }
603
604 static ssize_t my_read(int fd, char *ptr)
605 {
606     static int read_cnt = 0;
607     static char *read_ptr, read_buf[MAXLINE];
608
609     if (read_cnt <= 0) {
610     again:
611         if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
612             if (errno == EINTR)
613                 goto again;
614             return -1;
615         }
616         else if (read_cnt == 0)
617             return 0;
618         read_ptr = read_buf;
619     }
620     read_cnt--;
621     *ptr = *read_ptr++;
622     return 1;
623 }
624
625 ssize_t readline(int fd, void *buf, size_t maxlen)
626 {
627     int n, rc;
628     char c, *ptr = buf;
629
630     for (n = 1; n < maxlen; n++) {  /* notice that loop starts at 1 */
631         if ( (rc = my_read(fd, &c)) == 1) {
632             *ptr++ = c;
633             if (c == '\n')
634                 break;          /* newline is stored, like fgets() */
635         }
636         else if (rc == 0) {
637             if (n == 1)
638                 return 0;       /* EOF, no data read */
639             else
640                 break;          /* EOF, some data was read */
641         }
642         else
643             return -1;  /* error, errno set by read() */
644     }
645     *ptr = 0;   /* null terminate like fgets() */
646     return n;
```

```
}

/*
 * readline_r: Stevens's reentrant readline package
 * (mentioned but not defined in UNP 23.5)
 */

static ssize_t my_read_r(Rline *rptr, char *ptr)
{
    if (rptr->rl_cnt <= 0) {
    again:
        rptr->rl_cnt = read(rptr->read_fd, rptr->rl_buf,
                            sizeof(rptr->rl_buf));
        if (rptr->rl_cnt < 0) {
            if (errno == EINTR)
                goto again;
            else
                return(-1);
        }
        else if (rptr->rl_cnt == 0)
            return(0);
        rptr->rl_bufptr = rptr->rl_buf;
    }
    rptr->rl_cnt--;
    *ptr = *rptr->rl_bufptr++ & 255;
    return(1);
}

ssize_t readline_r(Rline *rptr)
{
    int n, rc;
    char c, *ptr = rptr->read_ptr;

    for (n = 1; n < rptr->read_maxlen; n++) {
        if ( (rc = my_read_r(rptr, &c)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return(0);      /* EOF, no data read */
            else
                break;          /* EOF, some data was read */
        } else
            return(-1); /* error */
    }
    *ptr = 0;
    return(n);
}

```

```
697 /*
698  * readline_rinit - initialization function for readline_r
699  */
700 void readline_rinit(int fd, void *ptr, size_t maxlen, Rline *rptr)
701 {
702     rptr->read_fd = fd;         /* save caller's arguments */
703     rptr->read_ptr = ptr;
704     rptr->read_maxlen = maxlen;
705 
706     rptr->rl_cnt = 0;           /* and init our counter & pointer */
707     rptr->rl_bufptr = rptr->rl_buf;
708 }
709 
710 /*************************************************
711  * Error-handling wrappers for Stevens's I/O helpers
712  ************************************************/
713 
714 ssize_t Readn(int fd, void *ptr, size_t nbytes)
715 {
716     ssize_t n;
717 
718     if ((n = readn(fd, ptr, nbytes)) < 0)
719         unix_error("Readn error");
720     return n;
721 }
722 
723 void Writen(int fd, void *ptr, size_t nbytes)
724 {
725     if (writen(fd, ptr, nbytes) != nbytes)
726         unix_error("Writen error");
727 }
728 
729 ssize_t Readline(int fd, void *ptr, size_t maxlen)
730 {
731     ssize_t n;
732 
733     if ((n = readline(fd, ptr, maxlen)) < 0)
734         unix_error("Readline error");
735     return n;
736 }
737 
738 ssize_t Readline_r(Rline *rptr)
739 {
740     ssize_t n;
741 
742     if ( (n = readline_r(rptr)) == -1)
743         unix_error("readline_r error");
744     return(n);
745 }
746 
```

```
747 void Readline_rinit(int fd, void *ptr, size_t maxlen, Rline *rptr)
748 {
749     readline_rinit(fd, ptr, maxlen, rptr);
750 }
```

code/src/csapp.c

Appendix B

Solutions to Practice Problems

B.1 Intro

B.2 Representing and Manipulating Information

Problem 2.1 Solution: [Pg. 24]

Converting between binary and hexadecimal is not very exciting, but it is an important skill. Like many skills, it can only be gained by practice.

Decimal	Binary	Hexadecimal
0	00000000	`00`
55	00110111	`37`
136	10001000	`88`
243	11110011	`F3`
82	01010010	`52`
172	10101100	`AC`
231	11100111	`E7`
167	10100111	`A7`
62	00111110	`3E`
188	10111100	`BC`

Problem 2.2 Solution: [Pg. 32]

This problem tests your understanding of the byte representation of data and the two different byte orderings.

A. Little endian: `78` Big endian: `12`

B. Little endian: `78 56` Big endian: `12 34`

C. Little endian: `78 56 34` Big endian: `12 34 56`

Note that on a little-endian machine we enumerate bytes starting from the most signficant byte and working toward the least, while on the big-endian machine we enumerate bytes starting from the least signficant byte and working toward the most.

Problem 2.3 Solution: [Pg. 32]

This problem is another chance to practice hexadecimal to binary conversion. It also gets you thinking about integer and floating-point representations. We will explore these representations in more detail later in this chapter.

A. Using the notation of the example in the text, we write the two strings as

```
   0   0   3   5   4   3   2   1
00000000001101010100001100100001
           ********************
     4   A   5   5   0   C   8   4
01001010010101010000110010000100
```

B. With the second word shifted two positions relative to the first we find a sequence with 21 matching bits.

C. We find all bits of the integer embedded in the floating point number, except for the most signficant bit having value 1. Such is the case for the example in the text as well. In addition the floating-point number has some nonzero high-order bits that do not match those of the integer.

Problem 2.4 Solution: [Pg. 33]

It prints `41 42 43 44 45 46`. Recall also that the library routine `strlen` does not count the terminating null character, and so `show_bytes` printed only through the character '`F`.'

Problem 2.5 Solution: [Pg. 36]

This problem is a drill to help you become more familiar with Boolean operations.

Operation	Result
a	$[01101001]$
b	$[01010101]$
~a	$[10010110]$
~b	$[10101010]$
a & b	$[01000001]$
a | b	$[01111101]$
a ^ b	$[00111100]$

Problem 2.6 Solution: [Pg. 37]

This procedure relies on the fact that EXCLUSIVE-OR is commutative and associative, and that $a \hat{} a = 0$ for any a. We will see in Chapter 5 that the code does not work correctly when the two pointers `x` and `y` are equal, that is, they point to the same location.

Step	*x	*y
Initially	a	b
Step 1	$a \hat{} b$	b
Step 2	$a \hat{} b$	$(a \hat{} b) \hat{} b = (b \hat{} b) \hat{} a = a$
Step 3	$(a \hat{} b) \hat{} a = (a \hat{} a) \hat{} b = b$	a

Problem 2.7 Solution: [Pg. 38]

Here are the expressions:

A. `x | ~0xFF`

B. `x ^ 0xFF`

C. `x & ~0xFF`

These expressions are typical of the kind commonly found in performing low-level bit operations. The expression `~0xFF` creates a mask where the 8 least-significant bits equal 0 and the rest equal 1. Observe that such a mask will be generated regardless of the word size. By contrast, the expression `0xFFFFFF00` would only work on a 32-bit machine.

Problem 2.8 Solution: [Pg. 38]

These problems help you think about the relation between Boolean operations and typical masking operations. Here is the code:

```
/* Bit Set */
int bis(int x, int m)
{
  int result = x | m;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  int result = x & ~m;
  return result;
}
```

It is easy to see that `bis` is equivalent to Boolean OR—a bit is set in `z` if either this bit is set in `x` or it is set in `m`.

The `bic` operation is a bit more subtle. We want to set a bit of `z` to 0 if the corresponding bit of `m` equals 1. If we complement the mask giving `~m`, then we want to set a bit of `z` to 0 if the corresponding bit of the complemented mask equals 0. We can do this with the AND operation.

Problem 2.9 Solution: [Pg. 39]

This problem highlights the relation between bit-level Boolean operations and logic operations in C.

Expression	Value	Expression	Value
`x & y`	`0x02`	`x && y`	`0x01`
`x | y`	`0xF7`	`x || y`	`0x01`
`~x | ~y`	`0xFD`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	`0x01`

Problem 2.10 Solution: [Pg. 40]

The expression is `!(x ^ y)`.

That is `x^y` will be zero if and only if every bit of `x` matches the corresponding bit of `y`. We then exploit the ability of `!` to determine whether a word contains any nonzero bit.

There is no real reason to use this expression rather than simply writing `x == y`, but it demonstrates some of the nuances of bit-level and logical operations.

Problem 2.11 Solution: [Pg. 40]

This problem is a drill to help you understand the different shift operations.

`x`	`x << 3`	`x >> 2` (Logical)	`x >> 2` (Arithmetic)
`0xF0`	`0x80`	`0x3C`	`0xFC`
`0x0F`	`0x78`	`0x03`	`0x03`
`0xCC`	`0x60`	`0x33`	`0xF3`
`0x55`	`0xA8`	`0x15`	`0x15`

Problem 2.12 Solution: [Pg. 43]

In general, working through examples for very small word sizes is a very good way to understand computer arithmetic.

The unsigned values correspond to those in Figure 2.1.

For the two's complement values, hex digits `0` through `7` have a most significant bit of 0, yielding non-negative values, while while hex digits `8` through `F`, have a most significant bit of 1, yielding a negative value.

$\overline{x}$ (Hex)	$B2U_4(\overline{x})$	$B2T_4(\overline{x})$
`0`	0	0
`3`	3	3
`8`	8	-8
`A`	10	-6
`F`	15	-1

Problem 2.13 Solution: [Pg. 45]

The functions $T2U$ and $U2T$ are very peculiar from a mathematical perspective. It is important to understand how they behave.

We solve this problem by simply reordering the rows according to the two's complement value, and then list the unsigned value as the result of the function application.

x	$T2U_4(x)$
-8	8
-6	10
-1	15
0	0
3	3

Problem 2.14 Solution: [Pg. 46]

This exercise tests your understanding of Equation 2.4.

For the first eight entries, the values of x are negative and $T2U_4(x) = x + 2^4$. For the remaining eight entries, the values of x are nonnegative and $T2U_4(x) = x$.

Problem 2.15 Solution: [Pg. 52]

The effect of truncation is fairly intuitive for unsigned numbers, but not for two's complement numbers. This exercise lets you explore its properties using very small word sizes.

Hex		Unsigned		Two's Complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	0	0	0
3	3	3	3	3	3
8	0	8	0	-8	0
A	2	10	2	-6	2
F	7	15	7	-1	-1

As Equation 2.7 states, the effect of this truncation on unsigned values is to simply to find their residue, modulo 8. The effect of the truncation on signed values is a bit more complex. According to Equation 2.8, we first compute the modulo 8 residue of the argument. This will give values 0–7 for arguments 0–7, and also for arguments -8–-1. Then we apply function $U2T_3$ to these residues, giving two repetitions of the sequences 0–3 and -4–-1.

Problem 2.16 Solution: [Pg. 52]

This problem was designed to demonstrate how easily bugs can arise due to the implicit casting from signed to unsigned. It seems quite natural to pass parameter `length` as an unsigned, since one would never want to use a negative length. The stopping criterion `i <= length-1` also seems quite natural. But combining these two yields an unexpected outcome!

Since parameter `length` is unsigned, the computation $0-1$ is performed using unsigned arithmetic, which is equivalent to modular addition. The result is then $UMax_{32}$ (assuming a 32-bit machine). The $\leq$ comparison is also performed using an unsigned comparison, and since any 32-bit number is less than or equal to $UMax_{32}$, the comparison always holds! Thus, the code attempts to access invalid elements of array `a`.

The code can be fixed by either declaring `length` to be an `int`, or by changing the test of the `for` loop to be `i < length`.

Problem 2.17 Solution: [Pg. 56]

This problem is a simple demonstration of arithmetic modulo 16. The easiest way to solve it is to convert the hex pattern into its unsigned decimal value. For nonzero values of x, we must have $(-_4^{\mathrm{u}}\, x) + x = 16$. Then we convert the complemented value back to hex.

x		$-_4^{\mathrm{u}}\, x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
3	3	13	D
8	8	8	8
A	10	6	6
F	15	1	1

Problem 2.18 Solution: [Pg. 58]

This problem is an exercise to make sure you understand two's complement addition.

x	y	$x + y$	$x +_4^{\mathrm{t}} y$	Case
-16 [10000]	-11 [10101]	-27	5 [00101]	1
-16 [10000]	-16 [10000]	-32	0 [00000]	1
-8 [11000]	7 [00111]	-1	-1 [11111]	2
-2 [11110]	5 [00101]	3	3 [00011]	3
8 [01000]	8 [01000]	16	-16 [10000]	4

Problem 2.19 Solution: [Pg. 60]

This problem helps you understand two's complement negation using a very small word size.

For $w = 4$, we have $TMin_4 = -8$. So -8 is its own additive inverse, while other values are negated by integer negation.

x		$-_4^{\mathrm{t}}\, x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
3	3	-3	D
8	-8	-8	8
A	-6	6	6
F	-1	1	1

The bit patterns are the same as for unsigned negation.

Problem 2.20 Solution: [Pg. 63]

This problem is an exercise to make sure you understand two's complement multiplication.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	6	[110]	2	[010]	12	[001100]	4	[100]
Two's Comp.	−2	[110]	2	[010]	−4	[111100]	−4	[100]
Unsigned	1	[001]	7	[111]	7	[000111]	7	[111]
Two's Comp.	1	[001]	−1	[111]	−1	[111111]	7	[111]
Unsigned	7	[111]	7	[111]	49	[110001]	1	[001]
Two's Comp.	−1	[111]	−1	[111]	1	[000001]	1	[001]

Problem 2.21 Solution: [Pg. 64]

In Chapter 3, we will see many examples of the `leal` instruction in action. The instruction is provided to support pointer arithmetic, but the C compiler often uses it as a way to perform multiplication by small constants.

For each value of k, we can compute two multiples: 2^k (when `b` is 0) and $2^k + 1$ (when `b` is `a`. Thus, we can compute multiples 1, 2, 3, 4, 5, 8, and 9.

Problem 2.22 Solution: [Pg. 65]

We have found that students find this exercise looks difficult when working directly with assembly code. Formulating it in the manner we have shown in `optarith` can help clarify the behavior.

We can see that `M` is 15; `x*M` is computed as `(x<<4)-x`.

We can see that `N` is 4; a bias value of 3 is added when `y` is negative, and the right shift is by 2.

Problem 2.23 Solution: [Pg. 68]

Understanding fractional binary representations is an important step to understanding floating-point encodings. This exercise lets you try out some simple examples.

Fractional Value	Binary Rep.	Decimal Rep.
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$	0.011	0.375
$\frac{23}{16}$	1.0111	1.4375
$\frac{77}{32}$	10.1101	2.40625
$\frac{11}{8}$	1.011	1.375
$\frac{45}{8}$	101.101	5.625
$\frac{49}{16}$	11.0001	3.0625

One simple way to think about fractional binary representations is to represent a number as a fraction of the form $\frac{x}{2^k}$. We can write this in binary using the binary representation of x, with the binary point inserted k

positions from the right. As an example, for $\frac{23}{16}$, we have $23_10 = 10111_2$. We then put the binary point 4 positions from the right to get 1.0111_2.

Problem 2.24 Solution: [Pg. 68]

In most cases, the limited precision of floating-point numbers is not a major problem, because the *relative* error of the computation is still fairly low. In this example, however, the system was sensitive to the *absolute* error.

A. We can see that $x - 0.1$ has binary representation:

$$0.00000000000000000000000001100[1100]\cdots_2$$

Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is simply $2^{-20} \times \frac{1}{10}$, which is around 9.54×10^{-8}.

B. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$.

C. $0.343 \times 2000 \approx 687$.

Problem 2.25 Solution: [Pg. 73]

Working through floating point representations for very small word sizes helps clarify how IEEE floating point works. Note especially the transition between denormalized and normalized values.

Bits	e	E	f	M	V
0 00 00	0	0	0	0	0
0 00 01	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
0 00 10	0	0	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$
0 00 11	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$
0 01 00	1	0	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$
0 01 01	1	0	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$
0 01 10	1	0	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$
0 01 11	1	0	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01	2	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$
0 10 10	2	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$
0 10 11	2	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	NaN
0 11 10	—	—	—	—	NaN
0 11 11	—	—	—	—	NaN

Problem 2.26 Solution: [Pg. 74]

This exercise helps you think about what numbers cannot be represented exactly in floating point.

The number has binary representation 1 followed by n 0's followed by 1, giving value $2^{n+1} + 1$.

When $n = 23$, the value is $2^{24} + 1 = 16,777,217$.

Problem 2.27 Solution: [Pg. 78]

In general it is better to use a library macro rather than inventing your own code. This code seems to work on a variety of machines, however.

We assume that the value `1e400` overflows to infinity.

code/data/ieee.c

```
1 #define POS_INFINITY 1e400
2 #define NEG_INFINITY (-POS_INFINITY)
3 #define NEG_ZERO (-1.0/POS_INFINITY)
```

code/data/ieee.c

Problem 2.28 Solution: [Pg. 79]

Exercises such as this one help you develop your ability to reason about floating point operations from a programmer's perspective. Make sure you understand each of the answers.

A. `x == (int)(float) x`
No. For example when `x` is $TMax$.

B. `x == (int)(double) x`
Yes, since `double` has greater precision and range than `int`.

C. `f == (float)(double) f`
Yes, since `double` has greater precision and range than `float`.

D. `d == (float) d`
No. For example when `d` is `1e40`, we will get $+\infty$ on the right.

E. `f == -(-f)`
Yes, since a floating-point number is negated by simply inverting its sign bit.

F. `2/3 == 2/3.0`
No, the left hand value will be the integer value `0`, while the right hand value will be the floating-point approximation of $\frac{2}{3}$.

G. `(d >= 0.0) || ((d*2) < 0.0)`
Yes, since multiplication is monotonic.

H. `(d+f)-d == f`
No, for example when `d` is $+\infty$ and `f` is 1, the left hand side will be NaN, while the right hand side will be 1.

B.3 Machine Level Representation of C Programs

Problem 3.1 Solution: [Pg. 101]

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
`%eax`	`0x100`	Register
`0x104`	`0xAB`	Absolute address
`$0x108`	`0x108`	Immediate
`(%eax)`	`0xFF`	Address `0x100`
`4(%eax)`	`0xAB`	Address `0x104`
`9(%eax,%ecx)`	`0x11`	Address `0x10C`
`260(%ecx,%edx)`	`0x13`	Address `0x108`
`0xFC(,%ecx,4)`	`0xFF`	Address `0x100`
`(%eax,%edx,4)`	`0x11`	Address `0x10C`

Problem 3.2 Solution: [Pg. 104]

Reverse engineering is a good way to understand systems. In this case, we want to reverse the effect of the C compiler to determine what C code gave rise to this assembly code. The best way is to run a "simulation," starting with values `x`, `y`, and `z` at the locations designated by pointers `xp`, `yp`, and `zp`, respectively. We would then get the following behavior:

```
1   movl 8(%ebp),%edi     xp
2   movl 12(%ebp),%ebx    yp
3   movl 16(%ebp),%esi    zp
4   movl (%edi),%eax      x
5   movl (%ebx),%edx      y
6   movl (%esi),%ecx      z
7   movl %eax,(%ebx)      *yp = x
8   movl %edx,(%esi)      *zp = y
9   movl %ecx,(%edi)      *xp = z
```

From this we can generate the following C code:

code/asm/decode1-ans.c

```
1 void decode1(int *xp, int *yp, int *zp)
2 {
3     int tx = *xp;
4     int ty = *yp;
5     int tz = *zp;
6
7     *yp = tx;
8     *zp = ty;
9     *xp = tz;
10 }
```

code/asm/decode1-ans.c

Problem 3.3 Solution: [Pg. 106]

This exercise demonstrates the versatility of the `leal` instruction and gives you more practice in deciphering the different operand forms. Note that although the operand forms are classified as type "Memory" in Figure 3.3, no memory access occurs.

Expression	Result
`leal 6(%eax), %edx`	$6 + x$
`leal (%eax,%ecx), %edx`	$x + y$
`leal (%eax,%ecx,4), %edx`	$x + 4y$
`leal 7(%eax,%eax,8), %edx`	$7 + 9x$
`leal 0xA(,$ecx,4), %edx`	$10 + 4y$
`leal 9(%eax,%ecx,2), %edx`	$9 + x + 2y$

Problem 3.4 Solution: [Pg. 106]

This problem gives you a chance to test your understanding of operands and the arithmetic instructions.

Instruction	Destination	Value
`addl %ecx,(%eax)`	`0x100`	`0x100`
`subl %edx,4(%eax)`	`0x104`	`0xA8`
`imull $16,(%eax,%edx,4)`	`0x10C`	`0x110`
`incl 8(%eax)`	`0x108`	`0x14`
`decl %ecx`	`%ecx`	`0x0`
`subl %edx,%eax`	`%eax`	`0xFD`

Problem 3.5 Solution: [Pg. 107]

This exercise gives you a chance to generate a little bit of assembly code. The solution code was generated by GCC. By loading parameter `n` in register `%ecx`, it can then use byte register `%cl` to specify the shift amount for the `sarl` instruction.

```
1   movl 12(%ebp),%ecx    Get x
2   movl 8(%ebp),%eax     Get n
3   sall $2,%eax          x <<= 2
4   sarl %cl,%eax         x >>= n
```

Problem 3.6 Solution: [Pg. 108]

This instruction is used to set register `%edx` to 0, exploiting the property that $x \hat{} x = 0$ for any x. It corresponds to the C statement `i = 0`.

This is an example of an assembly language *idiom*—a fragment of code that is often generated to fulfill a special purpose. Recognizing such idioms is one step in becoming proficient at reading assembly code.

Problem 3.7 Solution: [Pg. 113]

This example requires you to think about the different comparison and set instructions. A key point to note is that by casting the value on one side of a comparison to `unsigned`, the comparison is performed as if both sides are unsigned, due to implicit casting.

```
1 char ctest(int a, int b, int c)
2 {
3   char t1 =         a <             b;
4   char t2 =         b <  (unsigned)    a;
5   char t3 = (short) c >= (short)    a;
6   char t4 = (char) a != (char)    c;
7   char t5 =         c >             b;
8   char t6 =         a >             0;
9   return t1 + t2 + t3 + t4 + t5 + t6;
10 }
```

Problem 3.8 Solution: [Pg. 116]

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `jbe` instruction has as target `0x8048d1c` + `0xda`. As the original disassembled code shows, this is `0x8048cf8`.

```
8048d1c:  76 da           jbe    8048cf8
8048d1e:  eb 24           jmp    8048d44
```

B. According to the annotation produced by the disassembler, the jump target is at absolute address `0x8048d44`. According to the byte encoding, this must be at an address `0x54` bytes beyond that of the `mov` instruction. Subtracting these gives address `0x8048cf0`, as confirmed by the disassembled code:

```
8048cee:  eb 54           jmp    8048d44
8048cf0:  c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

C. The target is at offset `000000cb` relative to `0x8048907` (the address of the `nop` instruction). Summing these gives address `0x80489d2`.

```
8048902:  e9 cb 00 00 00  jmp    80489d2
8048907:  90              nop
```

D. An indirect jump is denoted by instruction code `ff 25`. The address from which the jump target is to be read is encoded explicitly by the following 4 bytes. Since the machine is little endian, these are given in reverse order as `e0 a2 04 08`.

```
80483f0:  ff 25 e0 a2 04  jmp    *0x804a2e0
80483f5:  08
```

Problem 3.9 Solution: [Pg. 119]

Annotating assembly code and writing C code that mimics its control flow are good first steps in understanding assembly language programs. This problem gives you practice for an example with simple control flow. It also gives you a chance to examine the implementation of logical operations.

A. *code/asm/simple-if.c*

```
1 void cond(int a, int *p)
2 {
3   if (p == 0)
4     goto done;
5   if (a <= 0)
6     goto done;
7   *p += a;
8  done:
9 }
```

code/asm/simple-if.c

B. The first conditional branch is part of the implementation of the `||` expression. If the test for `p` being nonnull fails, the code will skip the test of `a > 0`.

Problem 3.10 Solution: [Pg. 120]

The code generated when compiling loops can be tricky to analyze, because the compiler can perform many different optimizations on loop code, and because it can be difficult to match program variables with registers. We start practicing this skill with a fairly simple loop.

A. The register usage can be determined by simply looking at how the arguments get fetched.

Register Usage		
Register	Variable	Initially
%esi	x	x
%ebx	y	y
%ecx	n	n

B. The *body-statement* portion consists of lines 4 to 6 in the C code and lines 6 to 8 in the assembly code. The *test-expr* portion is on line 7 in the C code. In the assembly code, it is implemented by the instructions on lines 9 to 14 as well as the branch condition on line 15.

C. The annotated code is as follows.

```
   Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
1    movl 8(%ebp),%esi      Put x in %esi
2    movl 12(%ebp),%ebx     Put y in %ebx
3    movl 16(%ebp),%ecx     Put n in %ecx
```

```
 4   .p2align 4,,7
 5 .L6:                      loop:
 6   imull %ecx,%ebx           y *= n
 7   addl %ecx,%esi            x += n
 8   decl %ecx                 n--
 9   testl %ecx,%ecx           Test n
10   setg %al                  n > 0
11   cmpl %ecx,%ebx            Compare y:n
12   setl %dl                  y < n
13   andl %edx,%eax            (n > 0) & (y < n)
14   testb $1,%al              Test least significant bit
15   jne .L6                    If != 0, goto loop
```

Note the somewhat strange implementation of the test expression. Apparently, the compiler recognizes that the two predicates `(n > 0)` and `(y < n)` can only evaluate to 0 or 1, and hence the branch condition need only test the least significant byte of their AND. The compiler could have been more clever and used the `testb` instruction to perform the AND operation.

Problem 3.11 Solution: [Pg. 125]

This is another chance to practice deciphering loop code. The C compiler has done some interesting optimizations.

A. The register usage can be determined by looking at how the arguments get fetched, and how registers are initialized.

Register Usage		
Register	Variable	Initially
%eax	a	a
%ebx	b	b
%ecx	i	0
%edx	result	a

B. The *test-expr* occurs on line 5 of the C code and on line 10 and the jump condition of line 11 in the assembly code. The *body-statement* occurs on lines 6 through 8 of the C code and on lines 7 to 9 of the assembly code. The compiler has detected that the initial test of the while loop will always be true, since `i` is initialized to 0, which is clearly less than 256.

C. The annotated code is as follows

```
1   movl 8(%ebp),%eax          Put a in %eax
2   movl 12(%ebp),%ebx         Put b in %ebx
3   xorl %ecx,%ecx             i = 0
4   movl %eax,%edx             result = a
5   .p2align 4,,7
  a in %eax, b in %ebx, i in %ecx, result in %edx
```

```
 6 .L5:                          loop:
 7   addl %eax,%edx                result += a
 8   subl %ebx,%eax                a -= b
 9   addl %ebx,%ecx                i += b
10   cmpl $255,%ecx                Compare i:255
11   jle .L5                       If <= goto loop
12   movl %edx,%eax                Set result as return value
```

D. The equivalent `goto` code is as follows

```
 1 int loop_while_goto(int a, int b)
 2 {
 3   int i = 0;
 4   int result = a;
 5  loop:
 6   result += a;
 7   a -= b;
 8   i += b;
 9   if (i <= 255)
10     goto loop;
11   return result;
12 }
```

Problem 3.12 Solution: [Pg. 127]

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces you to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops. It also demonstrates an optimization technique known as *code motion*, where a computation is moved out of a loop when it can be determined that its result will not change within the loop.

A. We can see that `result` must be in register `%eax`. It gets set to 0 initially and it is left in `%eax` at the end of the loop as a return value. We can see that `i` is held in register `%edx`, since this register is used as the basis for two conditional tests.

B. The instructions on lines 2 and 4 set `%edx` to `n-1`.

C. The tests on lines 5 and 12 require `i` to be nonnegative.

D. Variable `i` gets decremented by instruction 4.

E. Instructions 1, 6, and 7 cause `x*y` to be stored in register `%edx`.

F. Here is the original code:

```
1 int loop(int x, int y, int n)
2 {
3   int result = 0;
4   int i;
5   for (i = n-1; i >= 0; i = i-x) {
6     result += y * x;
7   }
8   return result;
9 }
```

Problem 3.13 Solution: [Pg. 131]

This problem gives you a chance to reason about the control flow of a switch statement. Answering the questions requires you to combine information from several places in the assembly code:

1. Line 2 of the assembly code adds 2 to `x` to set the lower range of the cases to 0. That means that the minimum case label is -2.
2. Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 6. This implies that the maximum case label is $-2 + 6 = 4$.
3. In the jump table, we see that the second entry (case label -1) has the same destination (`.L10`) as the jump instruction on line 4, indicating the default case behavior. Thus, case label -1 is missing in the switch statement body.
4. In the jump table, we see that the fifth and sixth entries have the same destination. These correspond to case labels 2 and 3.

From this reasoning we conclude:

A. The case labels in the switch statement body had values -2, 0, 1, 2, 3, and 4.

B. The case with destination `.L8` had labels 2 and 3.

Problem 3.14 Solution: [Pg. 135]

This is another example of an assembly code idiom. At first it seems quite peculiar—a `call` instruction with no matching `ret`. Then we realize that it is not really a procedure call after all.

A. `%eax` is set to the address of the `popl` instruction.

B. This is not a true subroutine call, since the control follows the same ordering as the instructions and the return address is popped from the stack.

C. This is the only way in IA32 to get the value of the program counter into an integer register.

Problem 3.15 Solution: [Pg. 136]

This problem makes concrete the discussion of register usage conventions. Registers `%edi`, `%esi`, and `%ebx` are callee save. The procedure must save them on the stack before altering their values and restore them before returning. The other three registers are caller save. They can be altered without affecting the behavior of the caller.

Problem 3.16 Solution: [Pg. 139]

Being able to reason about how functions use the stack is a critical part of understanding compiler-generated code. As this example illustrates, the compiler allocates a significant amount of space that never gets used.

A. We started with `%esp` having value `0x800040`. Line 2 decrements this by 4, giving `0x80003C`, and this becomes the new value of `%ebp`.

B. We can see how the two `leal` instructions compute the arguments to pass to `scanf`. Since arguments are pushed in reverse order, we can see that `x` is at offset -4 relative to `%ebp` and `y` is at offset -8. The addresses are therefore `0x800038` and `0x800034`.

C. Starting with the original value of `0x800040`, line 2 decremented the stack pointer by 4. Line 4 decremented it by 24, and line 5 decremented it by 4. The three pushes decremented it by 12, giving an overall change of 44. Thus, at line 11 `%esp` equals `0x800014`.

D. The stack frame has the following structure and contents:

```
          +----------+
0x80003C  | 0x800060 | <-- %ebp
          +----------+
0x800038  |     0x53 | (x)
          +----------+
0x800034  |     0x46 | (y)
          +----------+
0x800030  |          |
          +----------+
0x80002C  |          |
          +----------+
0x800028  |          |
          +----------+
0x800024  |          |
          +----------+
0x800020  |          |
          +----------+
0x80001C  | 0x800038 |
          +----------+
0x800018  | 0x800034 |
          +----------+
0x800014  | 0x300070 | <-- %esp
          +----------+
```

E. Byte addresses `0x800020` through `0x800033` are unused.

Problem 3.17 Solution: [Pg. 143]

This exercise tests your understanding of data sizes and array indexing. Observe that a pointer of any kind is four bytes long. The GCC implementation of `long double` uses 12 bytes to store each value, even though the actual format requires only 10 bytes.

Array	Element Size	Total Size	Start Address	Element i
S	2	28	x_S	$x_S + 2i$
T	4	12	x_T	$x_T + 4i$
U	4	24	x_U	$x_U + 4i$
V	12	96	x_V	$x_V + 12i$
W	4	16	x_W	$x_W + 4i$

Problem 3.18 Solution: [Pg. 145]

This problem is a variant of the one shown for integer array `E`. It is important to understand the difference between a pointer and the object being pointed to. Since data type `short` requires two bytes, all of the array indices are scaled by a factor of two. Rather than using `movl` as before, we now use `movw`.

Expression	Type	Value	Assembly
`S+1`	`short *`	$x_S + 2$	`leal 2(%edx),%eax`
`S[3]`	`short`	$Mem[x_S + 6]$	`movw 6(%edx),%ax`
`&S[i]`	`short *`	$x_S + 2i$	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	$Mem[x_S + 8i + 2]$	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short *`	$x_S + 2i - 10$	`leal -10(%edx,%ecx,2),%eax`

Problem 3.19 Solution: [Pg. 147]

This problem requires you to work through the scaling operations to determine the address computations, and to apply the formula for row-major indexing. The first step is to annotate the assembly to determine how the address references are computed:

```
1  movl 8(%ebp),%ecx              Get i
2  movl 12(%ebp),%eax             Get j
3  leal 0(,%eax,4),%ebx           4*j
4  leal 0(,%ecx,8),%edx           8*i
5  subl %ecx,%edx                 7*i
6  addl %ebx,%eax                 5*j
7  sall $2,%eax                   20*j
8  movl mat2(%eax,%ecx,4),%eax    mat2[(20*j + 4*i)/4]
9  addl mat1(%ebx,%edx,4),%eax    + mat1[(4*j + 28*i)/4]
```

From this we can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this we can determine that `mat1` has 7 columns, while `mat2` has 5, giving `M` $= 5$ and `N` $= 7$.

Problem 3.20 Solution: [Pg. 150]

This exercise requires you to study assembly code to understand how it has been optimized. This is an important skill for improving program performance. By adjusting your source code, you can have an effect on the efficiency of the generated machine code.

Here is an optimized version of the C code:

```
1 /* Set all diagonal elements to val */
2 void fix_set_diag_opt(fix_matrix A, int val)
3 {
4   int *Aptr = &A[0][0] + 255;
5   int cnt = N-1;
6   do {
7     *Aptr = val;
8     Aptr -= (N+1);
9     cnt--;
10  } while (cnt >= 0);
11 }
```

The relation to the assembly code can be seen via the following annotations:

```
1   movl 12(%ebp),%edx    Get val
2   movl 8(%ebp),%eax     Get A
3   movl $15,%ecx         i = 0
4   addl $1020,%eax       Aptr = &A[0][0] + 1020/4
5   .p2align 4,,7
6 .L50:                 loop:
7   movl %edx,(%eax)      *Aptr = val
8   addl $-68,%eax        Aptr -= 68/4
9   decl %ecx             i--
10  jns .L50              if i >= 0 goto loop
```

Observe how the assembly code program starts at the end of the array and works backward. It decrements the pointer by 68 ($= 17 \cdot 4$), since array elements `A[i-1][i-1]` and `A[i][i]` are spaced `N+1` elements apart.

Problem 3.21 Solution: [Pg. 155]

This problem gets you to think about structure layout and the code used to access structure fields. The structure declaration is a variant of the example shown in the text. It shows that nested structures are allocated by embedding the inner structures within the outer ones.

A. The layout of the structure is as follows:

Offset	0	4	8	12
Contents	p	s.x	s.y	next

B. 16 bytes

C. As always, we start by annotating the assembly code:

```
1   movl 8(%ebp),%eax          Get sp
2   movl 8(%eax),%edx          Get sp->s.y
3   movl %edx,4(%eax)          Copy to sp->s.x
4   leal 4(%eax),%edx          Get &(sp->s.x)
5   movl %edx,(%eax)           Copy to sp->p
6   movl %eax,12(%eax)         sp->next = p
```

From this, we can generate C code as follows:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = sp->s.y;
    sp->p     = &(sp->s.x);
    sp->next  = sp;
}
```

Problem 3.22 Solution: [Pg. 159]

This is a very tricky problem. It raises the need for puzzle-solving skills as part of reverse engineering to new heights. It shows very clearly that unions are simply a way to associate multiple names (and types) with a single storage location.

A. The layout of the union is as follows. As the figure illustrate, the union can have either its "`e1`" interpretation, having fields `e1.p` and `e1.y`, or it can have its "`e2`" interpretation, having fields `e2.x` and `e2.next`.

Offset	0	4
Contents	`e1.p`	`e1.y`
	`e2.x`	`e2.next`

B. 8 bytes

C. As always, we start by annotating the assembly code. In our annotations, we show multiple possible interpretations for some of the instructions, and then indicate which interpretation later gets discarded. For example, line 2 could be interpreted as either getting element `e1.y` or `e2.next`. In line 3, we see that the value gets used in an indirect memory reference, for which only the second interpretation of line 2 is possible.

```
1   movl 8(%ebp),%eax          Get up
2   movl 4(%eax),%edx          up->e1.y (no) or up->e2.next
3   movl (%edx),%ecx           up->e2.next->e1.p or up->e2.next->e2.x (no)
4   movl (%eax),%eax           up->e1.p (no) or up->e2.x
5   movl (%ecx),%ecx           *(up->e2.next->e1.p)
6   subl %eax,%ecx             *(up->e2.next->e1.p) - up->e2.x
7   movl %ecx,4(%edx)          Store in up->e2.next->e1.y
```

From this, we can generate C code as follows:

```
void proc (union ele *up)
{
    up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
}
```

Problem 3.23 Solution: [Pg. 162]

Understanding structure layout and alignment is very important for understanding how much storage different data structures require and for understanding the code generated by the compiler for accessing structures. This problem lets you work out the details of some example structures.

A. `struct P1 { int i; char c; int j; char d; };`

i	c	j	d	Total	Alignment
0	4	8	12	16	4

B. `struct P2 { int i; char c; char d; int j; };`

i	c	d	j	Total	Alignment
0	4	5	8	12	4

C. `struct P3 { short w[3]; char c[3] };`

w	c	Total	Alignment
0	6	10	2

D. `struct P4 { short w[3]; char *c[3] };`

w	c	Total	Alignment
0	8	20	4

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

a	p	Total	Alignment
0	32	36	4

Problem 3.24 Solution: [Pg. 170]

This problem covers a wide range of topics: stack frames, string representations, ASCII code, and byte ordering. It demonstrates the dangers of out-of-bounds memory references and the basic ideas behind buffer overflow.

A. Stack at line 7.

```
+-------------+
| 08 04 86 43 |  Return Address
+-------------+
| bf ff fc 94 |  Saved %ebp <-- %ebp
+-------------+
|             |  buf[4-7]
+-------------+
|             |  buf[0-3]
+-------------+
|             |
+-------------+
|             |
+-------------+
| 00 00 00 01 |  Saved %esi
+-------------+
| 00 00 00 02 |  Saved %ebx
+-------------+
```

B. Stack after line 10 (showing only words that are modified).

```
+-------------+
| 08 04 86 00 |  Return Address
+-------------+
| 31 30 39 38 |  Saved %ebp <-- %ebp
+-------------+
| 37 36 35 34 |  buf[4-7]
+-------------+
| 33 32 31 30 |  buf[0-3]
+-------------+
```

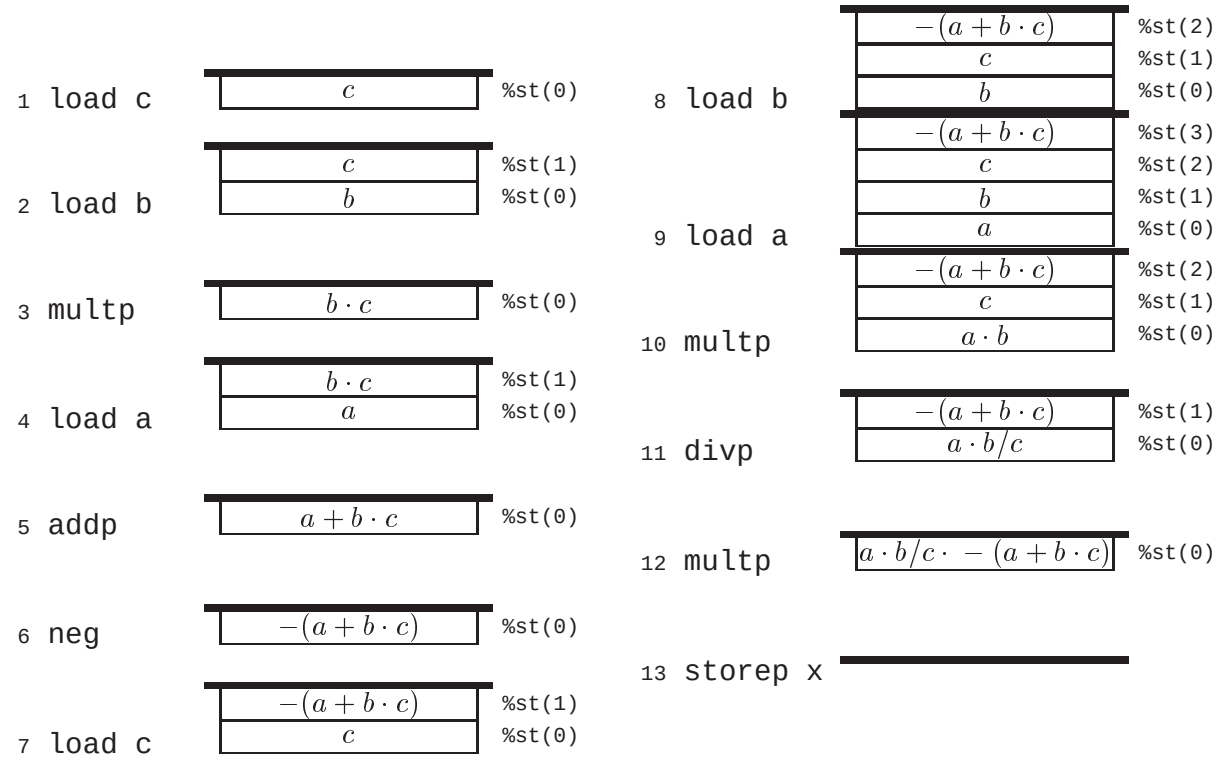
C. The program is attempting to return to address `0x08048600`. The low-order byte was overwritten by the terminating null character.

D. The saved value of register `%ebp` was changed to `0x31303938`, and this will be loaded into the register before `getline` returns. The other saved registers are not affected, since they are saved on the stack at lower addresses than `buf`.

E. The call to `malloc` should have had `strlen(buf)+1` as its argument, and it should also check that the returned value is non-null.

Problem 3.25 Solution: [Pg. 178]

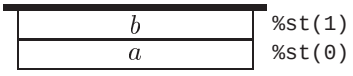
This problem gives you a chance to try out the recursive procedure described in 3.14.3.



Problem 3.26 Solution: [Pg. 179]

This code is similar to that generated by the compiler for selecting between two values based on the outcome of a test.

```
    test %eax,%eax
```




```
    jne L11
```


```
    fstp %st(0)

    jmp L9

L11:
```

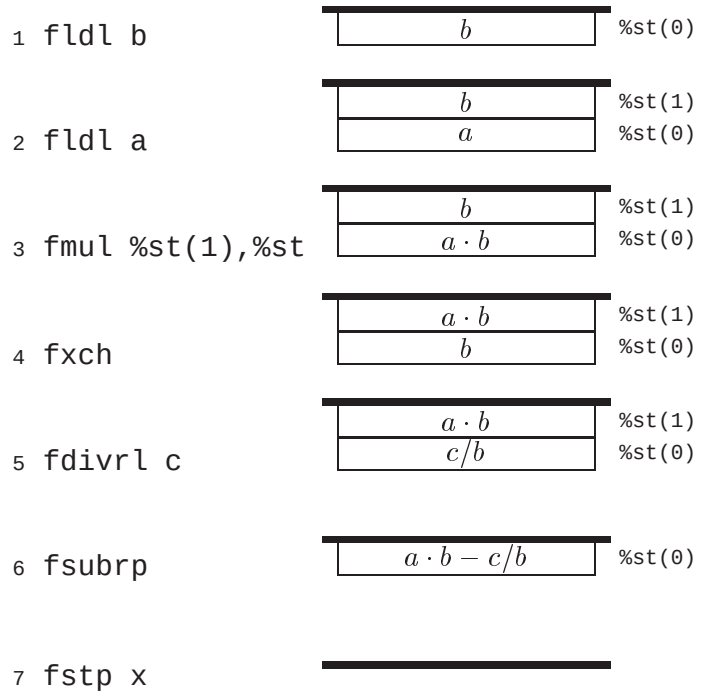

```
    fstp %st(1)

L9:
```

The resulting top of stack value is `x ? a : b`.

Problem 3.27 Solution: [Pg. 182]

Floating-point code is tricky, with all the different conventions about popping operands, the order of the arguments, etc. This problem gives you a chance to work through some specific cases in complete detail.



This code computes the expression `x = a*b - c/b`.

Problem 3.28 Solution: [Pg. 184]

This problem requires you to think about the different operand types and sizes in floating-point code.

code/asm/fpfunct2-ans.c

```
1 double funct2(int a, double x, float b, float i)
2 {
3     return a/(x+b) - (i+1);
4 }
```

code/asm/fpfunct2-ans.c

Problem 3.29 Solution: [Pg. 186]

Insert the following code between lines 4 and 5:

```
1   cmpb $1,%ah        Test if comparison outcome is <
```

Problem 3.30 Solution: [Pg. 191]

code/asm/asmprobs-ans.c

```
1 int ok_smul(int x, int y, int *dest)
2 {
3     long long prod = (long long) x * y;
4     int trunc = (int) prod;
5
6     *dest = trunc;
7     return (trunc == prod);
8 }
```

code/asm/asmprobs-ans.c

B.4 Processor Architecture

B.5 Optimizing Program Performance

Problem 5.1 Solution: [Pg. 205]

This problem illustrates some of the subtle effects of memory aliasing.

As the commented code below shows, the effect will be to set the value at `xp` to zero.

```
1     *xp = *xp + *xp; /* 2x */
2     *xp = *xp - *xp; /* 2x-2x = 0 */
3     *xp = *xp - *xp; /* 0-0 = 0 */
```

This example illustrates that our intuition about program behavior can often be wrong. We naturally think of the case where `xp` and `yp` are distinct but overlook the possibility that they might be equal. Bugs often arise due to conditions the programmer does not anticipate.

Problem 5.2 Solution: [Pg. 216]

This is a simple exercise, but it is important to recognize that the four statements of a `for` loop—initial, test, update, and body—get executed different numbers of times.

Code	`min`	`max`	`incr`	`square`
A.	1	91	90	90
B.	91	1	90	90
C.	1	1	90	90

Problem 5.3 Solution: [Pg. 238]

As we found in Chapter 3, reverse engineering from assembly code to C code provides useful insights into the compilation process. The following code shows the form for general data and combining operation.

```
1 void combine5px8(vec_ptr v, data_t *dest)
2 {
```

```
 3     int length = vec_length(v);
 4     int limit = length - 3;
 5     data_t *data = get_vec_start(v);
 6     data_t x = IDENT;
 7     int i;
 8
 9     /* Combine 8 elements at a time */
10     for (i = 0; i < limit; i+=8) {
11         x = x OPER data[0]
12             OPER data[1]
13             OPER data[2]
14             OPER data[3]
15             OPER data[4]
16             OPER data[5]
17             OPER data[6]
18             OPER data[7];
19         data += 8;
20     }
21
22     /* Finish any remaining elements */
23     for (; i < length; i++) {
24         x = x OPER data[0];
25         data++;
26     }
27     *dest = x;
28 }
```

Our handwritten pointer code is able to eliminate loop variable `i` by computing an ending value for the pointer. This is another example of where a human can often see transformations that are overlooked by the compiler.

Problem 5.4 Solution: [Pg. 246]

Spilled values are generally stored in the local stack frame. They therefore have a negative offset relative to `%ebp`. We can see such a reference at line 12 in the assembly code.

A. Variable `limit` has been spilled to the stack.

B. It is at offset -8 relative to `%ebp`.

C. This value is only required to determine whether the `jl` instruction closing the loop should be taken. If the branch prediction logic predicts the branch as taken, then the next iteration can proceed before the loop test has completed. Therefore, the comparison instruction is not part of the critical path determining the loop performance. Furthermore, since this variable is not altered within the loop, having it on the stack does not require any additional store operations.

Problem 5.5 Solution: [Pg. 252]

This problem demonstrates the need to be careful when using conditional moves. They require evaluating a value for the source operand, even when this value is not used.

This code always dereferences `xp` (instruction B2). This will cause a null pointer reference in the case where `xp` is zero.

Problem 5.6 Solution: [Pg. 260]

This problem requires you to analyze the potential load-store interactions in a program.

A. It will set each element `a[`i`]` to $i + 1$, for $0 \leq i \leq 998$.

B. It will set each element `a[`i`]` to 0, for $1 \leq i \leq 999$.

C. In the second case, the load of one iteration depends on the result of the store from the previous iteration. Thus, there is a write/read dependency between successive iterations.

D. It will give a CPE of 5.00, since there are no dependencies between stores and subsequent loads.

Problem 5.7 Solution: [Pg. 266]

Amdahl's Law is best understood by working through some examples. This one requires you to look at Equation 5.1 from an unusual perspective.

This problem is a simple application of the equation. You are given $S = 2$ and $\alpha = .8$, and you must then solve for k:

$$\begin{aligned} 2 &= \frac{1}{(1-0.8)+0.8/k} \\ 0.4 + 1.6/k &= 1.0 \\ k &= 2.67 \end{aligned}$$

B.6 The Memory Hierarchy

Problem 6.1 Solution: [Pg. 280]

The idea here is to minimize the number of address bits by minimizing the aspect ratio $\max(r, c)/\min(r, c)$. In other words, the squarer the array, the fewer the address bits.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1	4	4	2	2	2
16×4	4	4	2	2	2
128×8	16	8	4	3	4
512×4	32	16	5	4	5
1024×4	32	32	5	5	5

Problem 6.2 Solution: [Pg. 287]

The point of this little drill is to make sure you understand the relationship between cylinders and tracks. Once you have that straight, just plug and chug:

$$\begin{aligned} \textit{Disk capacity} &= \frac{\textit{512 bytes}}{\textit{sector}} \times \frac{\textit{400 sectors}}{\textit{track}} \times \frac{\textit{10,000 tracks}}{\textit{surface}} \times \frac{\textit{2 surfaces}}{\textit{platter}} \times \frac{\textit{2 platters}}{\textit{disk}} \\ &= \textit{8,192,000,000 bytes} \\ &= \textit{8.192 GB.} \end{aligned}$$

Problem 6.3 Solution: [Pg. 289]

This solution to this problem is a straightforward application of the formula for disk access time. The average rotational latency (in ms) is

$$\begin{aligned} T_{avg\,rotation} &= \textit{1/2} \times T_{max\,rotation} \\ &= \textit{1/2} \times (\textit{60 secs / 15,000 RPM}) \times \textit{1000 ms/sec} \\ &\approx \textit{2 ms}. \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\,transfer} &= (\textit{60 secs / 15,000 RPM}) \times \textit{1/500 sectors/track} \times \textit{1000 ms/sec} \\ &\approx \textit{0.008 ms}. \end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned} T_{access} &= T_{avg\,seek} + T_{avg\,rotation} + T_{avg\,transfer} \\ &= \textit{8 ms} + \textit{2 ms} + \textit{0.008 ms} \\ &\approx \textit{10 ms}. \end{aligned}$$

Problem 6.4 Solution: [Pg. 298]

To create a stride-1 reference pattern, the loops must be permuted so that the rightmost indices change most rapidly.

```
1  int sumarray3d(int a[N][N][N])
2  {
3      int i, j, k, sum = 0;
4
5      for (k = 0; k < N; k++) {
6          for (i = 0; i < N; i++) {
7              for (j = 0; j < N; j++) {
8                  sum += a[k][i][j];
9              }
10         }
11     }
12     return sum;
13 }
```

This is an important idea. Make sure you understand why this particular loop permutation results in a stride-1 access pattern.

Problem 6.5 Solution: [Pg. 298]

The key to solving this problem is to visualize how the array is laid out in memory and then analyze the reference patterns. Function `clear1` accesses the array using a stride-1 reference pattern and thus clearly has the best spatial locality. Function `clear2` scans each of the N structs in order, which is good, but within each struct it hops around in a non-stride-1 pattern at the following offsets from the beginning of the struct: $0, 12, 4, 16, 8, 20$. So `clear2` has worse spatial locality than `clear1`. Function `clear3` not only hops around within each struct, but it also hops from struct to struct. So `clear3` exhibits worse spatial locality than `clear2` and `clear1`.

Problem 6.6 Solution: [Pg. 306]

The solution is a straightforward application of the definitions of the various cache parameters in Figure 6.26. Not very exciting, but you need to understand how the cache organization induces these partitions in the address bits before you can really understand how caches work.

	m	C	B	E	S	t	s	b
1.	32	1024	4	1	256	22	8	2
2.	32	1024	8	4	32	24	5	3
3.	32	1024	32	32	1	27	0	5

Problem 6.7 Solution: [Pg. 312]

The padding eliminates the conflict misses. Thus $3/4$ of the references are hits.

Problem 6.8 Solution: [Pg. 313]

Sometimes, understanding why something is a bad idea helps you understand why the alternative is a good idea. Here, the bad idea we are looking at is indexing the cache with the high order bits instead of the middle bits.

A. With high-order bit indexing, each contiguous array chunk consists of 2^t blocks, where t is the number of tag bits. Thus, the first 2^t contiguous blocks of the array would map to Set 0, the next 2^t blocks would map to Set 1, and so on.

B. For a direct-mapped cache where $(S, E, B, m) = (512, 1, 32, 32)$, the cache capacity is 512 32-byte blocks, and there are $t = 18$ tag bits in each cache line. Thus, the first 2^{18} blocks in the array would map to Set 0, the next 2^{18} blocks to Set 1. Since our array consists of only $4096/32 = 512$ blocks, all of the blocks in the array map to Set 0. Thus the cache will hold at most 1 array block at any point in time, even though the array is small enough to fit entirely in the cache. Clearly, using high-order bit indexing makes poor use of the cache.

Problem 6.9 Solution: [Pg. 316]

The two low order bits are the block offset (CO), followed by three bits of set index (CI), with the remaining bits serving as the tag (CT).

12	11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

Problem 6.10 Solution: [Pg. 317]

Address: `0x0E34`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x0
Cache set index (CI)	0x5
Cache tag (CT)	0x71
Cache hit? (Y/N)	Y
Cache byte returned	0xB

Problem 6.11 Solution: [Pg. 318]

Address: `0x0DD5`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	0	1	0	1
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x1
Cache set index (CI)	0x5
Cache tag (CT)	0x6E
Cache hit? (Y/N)	N
Cache byte returned	–

Problem 6.12 Solution: [Pg. 318]

Address: `0x1FF4`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Cache block offset	0x0
Cache set index	0x1
Cache tag	0xFF
Cache hit? (Y/N)	N
Cache byte returned	–

Problem 6.13 Solution: [Pg. 318]

This problem is a sort of inverse version of Problems 6.9–6.12 that requires you to work backwards from the contents of the cache to derive the addresses that will hit in a particular set. In this case, Set 3 contains one valid line with a tag of `0x32`. Since there is only one valid line in the set, four addresses will hit. These addresses have the binary form `0 0110 0100 11xx`. Thus, the four hex addresses that hit in Set 3 are:

`0x064C`, `0x064D`, `0x064E`, and `0x064F`.

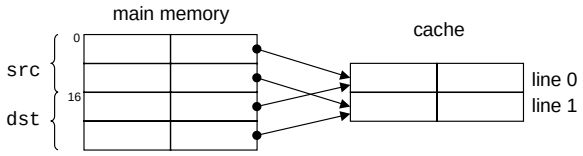


Figure B.1: **Figure for Problem 6.14.**

Problem 6.14 Solution: [Pg. 324]

A. The key to solving this problem is to visualize the picture in Figure B.1. Notice that each cache line holds exactly one row of the array, that the cache is exactly large enough to hold one array, and that for all i, row i of `src` and `dst` maps to the same cache line. Because the cache is too small to hold both arrays, references to one array keep evicting useful lines from the other array. For example, the write to `dst[0][0]` evicts the line that was loaded when we read `src[0][0]`. So when we next read `src[0][1]`we have a miss.

dst array		
	col 0	col 1
row 0	m	m
row 1	m	m

src array		
	col 0	col 1
row 0	m	m
row 1	m	h

B. When the cache is 32 bytes, it is large enough to hold both arrays. Thus the only misses are the initial cold misses.

dst array		
	col 0	col 1
row 0	m	h
row 1	m	h

src array		
	col 0	col 1
row 0	m	h
row 1	m	h

Problem 6.15 Solution: [Pg. 325]

Each 16-byte cache line holds two contiguous `algae_position` structures. Each loop visits these structures in memory order, reading one integer element each time. So the pattern for each loop is: miss, hit, miss, hit, and so on. Notice that for this problem, we could have predicted the miss rate without actually enumerating the total number of reads and misses.

A. What is the total number of read accesses? 512 reads.

B. What is the total number of read accesses that miss in the cache? 256 misses.

C. What is the miss rate? $256/512 = 50\%$.

Problem 6.16 Solution: [Pg. 326]

The key to this problem is noticing that the cache can only hold $1/2$ of the array. So the column-wise scan of the second half of the array evicts the lines that were loaded during the scan of the first half. For example, reading the first element of `grid[16][0]` evicts the line that was loaded when we read elements from `grid[0][0]`. This line also contained `grid[0][1]`). So when we begin scanning the next column, the reference to the first element of `grid[0][1]` misses.

A. What is the total number of read accesses? 512 reads.

B. What is the total number of read accesses that miss in the cache? 256 misses.

C. What is the miss rate? $256/512 = 50\%$.

D. What would the miss rate be if the cache were twice as big? If the cache were twice as big, it could hold the entire `grid` array. The only misses would be the initial cold misses, and the miss rate would be $1/4 = 25\%$.

Problem 6.17 Solution: [Pg. 326]

This loop has a nice stride-1 reference pattern, and thus the only misses are the initial cold misses.

A. What is the total number of read accesses? 512 reads.

B. What is the total number of read accesses that miss in the cache? 128 misses.

C. What is the miss rate? $128/512 = 25\%$.

D. What would the miss rate be if the cache were twice as big? Increasing the cache size by any amount would not change the miss rate, since cold misses are unavoidable.

Problem 6.18 Solution: [Pg. 331]

This problem is just a sanity check to make sure you been following the discussion. Stride corresponds to spatial locality. Working set size corresponds to temporal locality.

Problem 6.19 Solution: [Pg. 331]

A. The peak throughput from L1 is about 1000 MB/s and the clock frequency is about 500 MHz. Thus it takes roughly $500/1000 \times 4 = 2$ cycles to access a word from L1.

B. To estimate the L2 access time, we need to identify a region on the memory mountain where each reference is missing in L1 and then hitting in L2. In particular, we want a region where (1) the working set is too big for L1 but fits in L2 (e.g., 256 bytes) and (2) the stride exceeds the line size (e.g., a stride of 16 words). From the memory mountain graph, observe that the effective throughput in the region (size=256, stride=16) is about 300 MB/s. Thus, we estimate that it takes about $500/300 \times 4 \approx 7$ cycles to read a word from L2.

C. To estimate the main memory access time, we look at the point on the mountain with the largest stride and working set size, where every reference is missing in both L1 and L2. From the graph, the read throughput in the region (size=8M, stride=16) is about 80 MB/s. Thus, we estimate that it takes about $500/80 \times 4 \approx 25$ cycles to read a word from main memory.

B.7 Linking

Problem 7.1 Solution: [Pg. 357]

The purpose of this problem is to help you understand the relationship between linker symbols and C variables and functions. Notice that the C local variable `temp` does *not* have a symbol table entry.

Symbol	`swap.o .symtab` entry?	Symbol type	Module where defined	Section
`buf`	yes	extern	`main.o`	`.data`
`bufp0`	yes	global	`swap.o`	`.data`
`bufp1`	yes	global	`swap.o`	`.bss`
`swap`	yes	global	`swap.o`	`.text`
`temp`	no	—	—	—

Problem 7.2 Solution: [Pg. 360]

This is a simple drill that checks your understanding of the rules that a Unix linker uses when it resolves global symbols that are defined in more than one module. Understanding these rules can help you avoid some nasty programming bugs.

A. The linker chooses the strong symbol defined in Module 1 over the weak symbol defined in Module 2 (Rule 2):

   (a) `REF(main.1) --> DEF(main.1)`

   (b) `REF(main.2) --> DEF(main.1)`

B. This is an ERROR, because each module defines a strong symbol `main` (Rule 1).

C. The linker chooses the strong symbol defined in Module 2 over the weak symbol defined in Module 1 (Rule 2):

   (a) `REF(x.1) --> DEF(x.2)`

   (b) `REF(x.2) --> DEF(x.2)`

Problem 7.3 Solution: [Pg. 365]

Placing static libraries in the wrong order on the command line is a common source of linker errors that confuses many programmers. However, once you understand how linkers use static libraries to resolve references, it's pretty straightforward. This little drill checks your understanding of this idea:

A. `gcc p.o libx.a`

B. `gcc p.o libx.a liby.a`

C. `gcc p.o libx.a liby.a libx.a`

Problem 7.4 Solution: [Pg. 369]

This problem concerns the disassembly listing in Figure 7.10. Our purpose here is to give you some practice reading disassembly listings and to check your understanding of PC-relative addressing.

A. The hex address of the relocated reference in line 5 is `0x80483bb`.

B. The hex value of the relocated reference in line 5 is `0x9`. Remember that the disassembly listing shows the value of the reference in little-endian byte order.

C. The key observation here is that no matter where the linker locates the `.text` section, the distance between the reference and the `swap` function is always the same. Thus, because the reference is a PC-relative address, its value will be `0x9`, regardless of where the linker locates the `.text` section.

Problem 7.5 Solution: [Pg. 374]

How C programs actually start up is a mystery to most programmers. These questios check your understanding of this startup process. You can answer them by referrig to the C startup code in Figure 7.14:

A. Every program needs a `main` function, because the C startup code, which is common to every C program, jumps to a function called `main`.

B. If `main` terminates with a `return` statement, then control passes back to the startup routine, which returns control to the operating system by calling `_exit`. The same behavior occurs if the user omits the `return` statement. If `main` terminates with a call to `exit`, then `exit` eventually returns control to the operating system by calling `_exit`. The net effect is the same in all three cases: when `main` has finished, control passes back to the operating system.

B.8 Exceptional Control Flow

Problem 8.1 Solution: [Pg. 408]

In our example program in Figure 8.13, the parent and child execute disjoint sets of instructions. However, in this program, the parent and child execute non-disjoint sets of instructions, which is possible because the parent and child have identical code segments. This can be a difficult conceptual hurdle. So be sure you understand the solution to this problem.

A. What is the output of the child process? The key idea here is that the child executes both `printf` statements. After the `fork` returns, it executes the `printf` in line 8. Then it falls out of the `if` statement and executes the `printf` in line 9. Here is the output produced by the child:

```
printf1: x=2
printf2: x=1
```

B. What is the output of the parent process? The parent executes only the `printf` in line 9:

```
printf2: x=0
```

Problem 8.2 Solution: [Pg. 408]

This program has the same process hierarchy as the program in Figure 8.14(c). There are a total of four processes, each of which prints a single “hello” line. Thus, the program prints four “hello” lines.

Problem 8.3 Solution: [Pg. 408]

This program has the same process hierarchy as the program in Figure 8.14(c). There are four processes. Each process prints one “hello” line in `doit` and one “hello” line in `main` after it returns from `doit`. Thus, the program prints a total of eight “hello” lines.

Problem 8.4 Solution: [Pg. 411]

A. Each time we run this program, it generates six output lines.

B. The ordering of the output lines will vary from system to system, depending on the how the kernel interleaves the instructions of the parent and the child. In general, any topological sort of the following graph is a valid ordering:

```
          --> ''0'' --> ''2'' --> ''Bye''      parent process
         /
''Hello''
         \
          --> ''1'' --> ''Bye''                child process
```

For example, when we run the program on our system, we get the following output:

```
unix> ./waitprob1
Hello
0
1
Bye
2
Bye
```

In this case, the parent runs first, printing "`Hello`" in line 8 and "`0`" in line 10. The call to `wait` blocks because the child has not yet terminated, so the kernel does a context switch and passes control to the child, which prints "`1`" in line 10 and "`Bye`" in line 16, and then terminates with an exit status of 2 in line 17. After the child terminates, the parent resumes, printing the child's exit status in line 14 and "`Bye`" in line 16.

Problem 8.5 Solution: [Pg. 415]

code/ecf/snooze.c

```
1 unsigned int snooze(unsigned int secs) {
2     unsigned int rc = sleep(secs);
3     printf("Slept for %u of %u secs.\n", secs - rc, secs);
4     return rc;
5 }
```

code/ecf/snooze.c

Problem 8.6 Solution: [Pg. 417]

code/ecf/myecho.c

```
1 #include "csapp.h"
2
3 int main(int argc, char *argv[], char *envp[])
4 {
5     int i;
```

```
6
7     printf("Command line arguments:\n");
8     for (i=0; argv[i] != NULL; i++)
9         printf("    argv[%2d]: %s\n", i, argv[i]);
10
11    printf("\n");
12    printf("Environment variables:\n");
13    for (i=0; envp[i] != NULL; i++)
14        printf("    envp[%2d]: %s\n", i, envp[i]);
15
16    exit(0);
17 }
```

code/ecf/myecho.c

Problem 8.7 Solution: [Pg. 429]

The `sleep` function returns prematurely whenever the sleeping process receives a signal that is not ignored. But since the default action upon receipt of a SIGINT is to terminate the process (Figure 8.23), we must install a SIGINT handler to allow the `sleep` function to return. The handler simply catches the SIGNAL and returns control to the `sleep` function, which then returns immediately.

code/ecf/snooze.c

```
1 #include "csapp.h"
2
3 /* SIGINT handler */
4 void handler(int sig)
5 {
6     return; /* catch the signal and return */
7 }
8
9 unsigned int snooze(unsigned int secs) {
10    unsigned int rc = sleep(secs);
11    printf("Slept for %u of %u secs.\n", secs - rc, secs);
12    return rc;
13 }
14
15 int main(int argc, char **argv) {
16
17    if (argc != 2) {
18        fprintf(stderr, "usage: %s <secs>\n", argv[0]);
19        exit(0);
20    }
21
22    if (signal(SIGINT, handler) == SIG_ERR) /* install SIGINT handler */
23        unix_error("signal error\n");
24    (void)snooze(atoi(argv[1]));
25    exit(0);
26 }
```

code/ecf/snooze.c

B.9 Measuring Program Performance

Problem 9.1 Solution: [Pg. 451]

At first, it seems ridiculous to interrupt the CPU and execute 100,000 cycles just to deal with a single keystroke. When you work through the numbers, however, it becomes clear that the overall load on the CPU will be fairly small.

100 WPM corresponds to 10 keystrokes per second. The total number of cycles used per second by the 100 typists will be $10 \times 10^2 \times 10^5 = 10^8$, i.e., 10% of the total cycles the processor can supply.

Problem 9.2 Solution: [Pg. 454]

This problem requires careful study of the trace, and an anticipation of the type of pattern that will arise.

A. They occur every 9.98–9.99ms: *358.93*, *368.91*, 378.89, *388.88*, 398.86, *408.85*, 418.83, *428.81*. Note that the ones that are not italicized were determined by adding 9.98 to the preceding time.

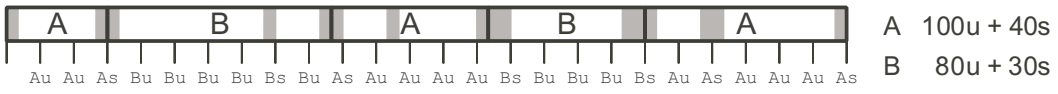
B. The italicized times shown above. They caused a new period of inactivity.

C. The inactive times include the time spent servicing two interrupts in addition to the time spent executing the other process.

D. Our process is active for around 9.5ms every 20.0 ms, i.e., 47.5% of the time.

Problem 9.3 Solution: [Pg. 457]

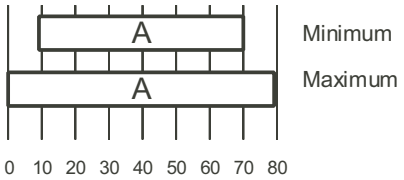
This problem involves simply labeling the execution sequence according to the process that is executing, and determining whether the process is in user or kernel mode.



Problem 9.4 Solution: [Pg. 457]

This is an interesting thought problem. It helps you reason about the range of possible times that can lead to a given interval count.

The following diagram illustrates the two cases:



For the minimum case, the segment started just before the interrupt at time 10 and finished right as the interrupt at time 70 occurred, giving a total time of just over 60ms. For the maximum case, the segment started right after the interrupt at time 0 and continued until just before the interrupt at time 80, giving a total time of just under 80ms.

Problem 9.5 Solution: [Pg. 457]

This problem requires thinking about how well the accounting scheme works. The seven timer interrupts occur while the process is active. This would give a user time of 70ms and a system time of 0ms. In the actual trace, the process ran for 63.7ms in user mode and 3.3ms in kernel mode. The counter overestimated the true execution time by $70/(63.7 + 3.3) = 1.04$X.

Problem 9.6 Solution: [Pg. 465]

This problem requires reasoning about the different sources of delay in a program and under what conditions these sources will apply.

From these measurements we get:

$$\begin{aligned} c + m + p + d &= 399 \\ c + d &= 133 \pm 1 \\ c + p &= 317 \end{aligned}$$

From this we conclude that $c = 100$, $d \approx 33$, $p = 217$, and $m = 49$.

Problem 9.7 Solution: [Pg. 475]

This problem requires applying probability theory to a simple model of process scheduling. It demonstrates that obtaining accurate measurements becomes very difficult as the times approach the process time limit.

A. For $t \leq 50$, the probability of running in one segment is $1 - t/50$. For $t > 50$, the probability is 0.

B. For $t \geq 50$, we will never get any trial that executes within a single process segment. For $t < 50$, the probability of success is $p = (50 - t)/50$, and hence we would expect $3/p = 150/(50 - t)$ trials. For $t = 20$ we expect to require 5 trials, while for $t = 40$ we expect 15.

Problem 9.8 Solution: [Pg. 476]

This is the UNIX version of the Y2K problem. Some people predict total disaster when the clock wraps around. Just as with Y2K, we believe these fears are unwarranted.

This will occur 2^{31} seconds after January 1, 1970, i.e., on January 19, 2038, at 3:14AM.

B.10 Virtual Memory

Problem 10.1 Solution: [Pg. 488]

This problem gives you some appreciation for the sizes of different address spaces. At one point in time, a 32-bit address space seemed impossibly large. But now there are database and scientific applications that need more, and you can expect this trend to continue. At some point in your lifetime, expect to find yourself complaining about the cramped 64-bit address space on your personal computer!

# address bits (n)	# unique addresses (N)	Largest address
8	$2^8 = 256$	$2^8 - 1 = 255$
16	$2^{16} = 64K$	$2^{16} - 1 = 64K - 1$
32	$2^{32} = 4G$	$2^{32} - 1 = 4G - 1$
48	$2^{48} = 256T$	$2^{48} = 256T - 1$
64	$2^{64} = 16,384P$	$2^{64} - 1 = 16,384P - 1$

Problem 10.2 Solution: [Pg. 490]

Since each virtual page is $P = 2^p$ bytes, there are a total of $2^n/2^p = 2^{n-p}$ possible pages in the system, each of which needs a page table entry (PTE).

n	$P = 2^p$	# PTE's
16	4K	16
16	8K	8
32	4K	1M
32	8K	512K

Problem 10.3 Solution: [Pg. 500]

You need to understand this kind of problem cold in order to understand address translation. Here is how to solve the first subproblem: We are given $n = 32$ virtual address bits and $m = 24$ physical address bits. A page size of $P = 1KB$ means we need $\log_2(1K) = 10$ bits for both the VPO and PPO (Recall that the VPO and PPO are identical). The remaining address bits are the VPN and PPN respectively.

P	# VPN bits	# VPO bits	# PPN bits	# PPO bits
1 KB	22	10	14	10
2 KB	21	11	13	11
4 KB	20	12	12	12
8 KB	19	13	11	13

Problem 10.4 Solution: [Pg. 507]

Doing a few of these manual simulations is a great way to firm up your understanding of address translation. You might find it helpful to write out all the bits in the addresses, and then draw boxes around the different bit fields, such as VPN, TLBI, etc. In this particular problem, there are no misses of any kind: the TLB has a copy of the PTE and the cache has a copy of the requested data words. See Problems 10.11, 10.12, and 10.13 for some different combinations of hits and misses.

```
A. 00 0011 1101 0111

B.     VPN:            0xf
       TLBI:           0x3
       TLBT:           0x3
       TLB hit?        Y
       page fault?     N
       PPN:            0xd

C.     0011 0101 0111

D.     CO:             0x3
       CI:             0x5
       CT:             0xd
       cache hit?      Y
       cache byte?     0x1d
```

Problem 10.5 Solution: [Pg. 522]

Solving this problem will give you a good feel for the idea of memory mapping. Try it your yourself. We haven't discussed the `open`, `fstat`, or `write` functions, so you'll need to read their man pages to see how they work.

code/vm/mmapcopy.c

```
 1 #include "csapp.h"
 2
 3 /*
 4  * mmapcopy - uses mmap to copy file fd to stdout
 5  */
 6 void mmapcopy(int fd, int size)
 7 {
 8     char *bufp; /* ptr to memory mapped VM area */
 9
10     bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
11     Write(1, bufp, size);
12     return;
13 }
14
15 /* mmapcopy driver */
16 int main(int argc, char **argv)
```

```
17 {
18     struct stat stat;
19     int fd;
20
21     /* check for required command line argument */
22     if (argc != 2) {
23         printf("usage: %s <filename>\n", argv[0]);
24         exit(0);
25     }
26
27     /* copy the input argument to stdout */
28     fd = Open(argv[1], O_RDONLY, 0);
29     fstat(fd, &stat);
30     mmapcopy(fd, stat.st_size);
31     exit(0);
32 }
```

code/vm/mmapcopy.c

Problem 10.6 Solution: [Pg. 531]

This problem touches on some core ideas such as alignment requirements, minimum block sizes, and header encodings. The general approach for determining the block size is to round the sum of the requested payload and the header size to nearest multiple of the alignment requirement (in this case eight bytes). For example, the block size for the `malloc(1)` request is $4 + 1 = 5$ rounded up to eight. The block size for the `malloc(13)` request is $13 + 4 = 17$ rounded up to 24.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(1)`	8	`0x9`
`malloc(5)`	16	`0x11`
`malloc(12)`	16	`0x11`
`malloc(13)`	24	`0x19`

Problem 10.7 Solution: [Pg. 535]

The minimum block size can have a significant effect on internal fragmentation. Thus, it is good to understand the minimum block sizes associated with different allocator designs and alignment requirements. The tricky part is to realize that the same block can be allocated or free at different points in time. Thus, the minimum block size is the maximum of the minimum allocated block size and the minimum free block size. For example, in the last subproblem, the minimum allocated block size is a four-byte header and a one-byte payload rounded up to eight bytes. The minimum free block size is a four-byte header and four-byte footer, which is already a multiple of eight and doesn't need to be rounded. So the minimum block size for this allocator is eight bytes.

Alignment	Allocated block	Free block	Minimum block size (bytes)
Single-word	Header and footer	Header and footer	12
Single-word	Header, but no footer	Header and footer	8
Double-word	Header and footer	Header and footer	16
Double-word	Header, but no footer	Header and footer	8

Problem 10.8 Solution: [Pg. 543]

There is nothing very tricky here. But the solution requires you to understand how the rest of our simple implicit-list allocator works and how to manipulate and traverse blocks.

code/vm/malloc.c

```
1 static void *find_fit(size_t asize)
2 {
3     void *bp;
4
5     /* first fit search */
6     for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
7         if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
8             return bp;
9         }
10    }
11    return NULL; /* no fit */
12 }
```

code/vm/malloc.c

Problem 10.9 Solution: [Pg. 543]

The is another warm-up exercise to help you become familiar with allocators. Notice that for this allocator the minimum block size is 16 bytes. If the remainder of the block after splitting would be greater than or equal to the minimum block size, then we go ahead and split the block (lines 6 to 10). The only tricky part here is to realize that you need to place the new allocated block (lines 6 and 7) before moving to the next block (line 8).

code/vm/malloc.c

```
1 static void place(void *bp, size_t asize)
2 {
3     size_t csize = GET_SIZE(HDRP(bp));
4
5     if ((csize - asize) >= (DSIZE + OVERHEAD)) {
6         PUT(HDRP(bp), PACK(asize, 1));
7         PUT(FTRP(bp), PACK(asize, 1));
8         bp = NEXT_BLKP(bp);
9         PUT(HDRP(bp), PACK(csize-asize, 0));
10        PUT(FTRP(bp), PACK(csize-asize, 0));
11    }
12    else {
13        PUT(HDRP(bp), PACK(csize, 1));
14        PUT(FTRP(bp), PACK(csize, 1));
15    }
16 }
```

code/vm/malloc.c

Problem 10.10 Solution: [Pg. 545]

Here is one pattern that will cause external fragmentation: The application makes numerous allocation and free requests to the first size class, followed by numerous allocation and free requests to the second size class, followed by numerous allocation and free requests to the third size class, and so on. For each size class, the allocator creates a lot of memory that is never reclaimed because the allocator doesn't coalesce, and because the application never requests blocks from that size class again.

B.11 Concurrent Programming with Threads

Problem 11.1 Solution: [Pg. 569]

This is your first exposure to the many synchronization problems that can arise in threaded programs.

A. The problem is that the main thread calls `exit` without waiting for the peer thread to terminate. The `exit` call terminates the entire process, including any threads that happen to be running. So the peer thread is being killed before it has a chance to print its output string.

B. We can fix the bug by replacing the `exit` function with either `pthread_exit`, which waits for outstanding threads to terminate before it terminates the process, or `pthread_join` which explicitly reaps the peer thread.

Problem 11.2 Solution: [Pg. 572]

The main idea here is that stack variables are private while global and static variables are shared. Static variables such as `cnt` are a little tricky because the sharing is limited to the functions within their scope, in this case the thread routine.

A. Here is the table:

Variable instance	Referenced by main thread?	Referenced by peer thread 0 ?	Referenced by peer thread 1?
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

Notes:

`ptr:` A global variable that is written by the main thread and read by the peer threads.

`cnt:` A static variable with only one instance in memory that is read and written by the two peer threads.

`i.m`: A local automatic variable stored on the stack of the main thread. Even though its value is passed to the peer threads, the peer threads never reference it on the stack, and thus it is not shared.

`msgs.m`: A local automatic variable stored on the main thread's stack and referenced indirectly through `ptr` by both peer threads.

`myid.0` **and** `myid.1`: Instances of a local automatic variable residing on the stacks of peer threads 0 and 1 respectively.

B. Variables `ptr`, `cnt`, and `msgs` are referenced by more than one thread, and thus are shared.

Problem 11.3 Solution: [Pg. 576]

A. Sequentially consistent.

B. Not sequentially consistent because U_1 executes before L_1.

C. Sequentially consistent.

D. Not sequentially consistent because S_2 executes before U_2.

Problem 11.4 Solution: [Pg. 576]

The important idea here is that sequential consistency is not enough to guarantee correctness. Programs must explicitly synchronize accesses to shared variables.

Step	Thread	Instr	$\texttt{\%eax}_1$	$\texttt{\%eax}_2$	ctr
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	2	H_2	–	–	0
4	2	L_2	–	0	0
5	2	U_2	–	1	0
6	2	S_2	–	1	1
7	1	U_1	1	–	1
8	1	S_1	1	–	1
9	1	T_1	1	–	1
10	2	T_2	1	–	1

Variable `cnt` has a final incorrect value of 1.

Problem 11.5 Solution: [Pg. 599]

If we free the block immediately after the call to `pthread_create` in line 15, then we will introduce a new race, this time between the call to `free` in the main thread, and the assignment statement in line 25 of the thread routine.

Problem 11.6 Solution: [Pg. 599]

A. Another approach is to pass the integer `i` directly, rather than passing a pointer to `i`:

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, (void *)i);
```

In the thread routine, we cast the argument back to an `int` and assign it to `myid`:

```
int myid = (int) vargp;
```

B. The advantage is that it reduces overhead by eliminating the calls to `malloc` and `free`. A significant disadvantage is that it assumes that pointers are at least as large as `int`s. While this assumption is true for all modern systems, it might not be true for legacy or future systems.

Problem 11.7 Solution: [Pg. 600]

A. This program always deadlocks because the initial state is within the deadlock region..

B. To eliminate the deadlock, initaliaize the binary semaphore `t` to 1 instead of 0.

B.12 Network Programming

Problem 12.1 Solution: [Pg. 613]

Hex address	Dotted decimal address
0x0	0.0.0.0
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdbc9217	205.188.146.23

Problem 12.2 Solution: [Pg. 614]

code/net/hex2dd.c

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     struct in_addr inaddr;  /* addr in network byte order */
6     unsigned int   addr;    /* addr in host byte order */
7
```

```
 8     if (argc != 2) {
 9         fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
10         exit(0);
11     }
12     sscanf(argv[1], "%x", &addr);
13     inaddr.s_addr = htonl(addr);
14     printf("%s\n", inet_ntoa(inaddr));
15
16     exit(0);
17 }
```

code/net/hex2dd.c

Problem 12.3 Solution: [Pg. 614]

code/net/dd2hex.c

```
 1 #include "csapp.h"
 2
 3 int main(int argc, char **argv)
 4 {
 5     struct in_addr inaddr;  /* addr in network byte order */
 6     unsigned int   addr;    /* addr in host byte order */
 7
 8     if (argc != 2) {
 9         fprintf(stderr, "usage: %s <dotted-decimal>\n", argv[0]);
10         exit(0);
11     }
12
13     if (inet_aton(argv[1], &inaddr) == 0)
14         app_error("inet_aton error");
15     addr = ntohl(inaddr.s_addr);
16     printf("0x%x\n", addr);
17
18     exit(0);
19 }
```

code/net/dd2hex.c

Problem 12.4 Solution: [Pg. 618]

Each time we request the host entry for `aol.com`, the list of corresponding Internet addresses is returned in a different, round-robin order.

```
unix> ./hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13

unix>> ./hostinfo aol.com
```

```
official hostname: aol.com
address: 64.12.149.13
address: 205.188.146.23
address: 205.188.160.121

unix>> ./hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13
```

The different ordering of the addresses in different DNS queries is known as *DNS round-robin*. It can be used to load-balance requests to a heavily used domain name.

Problem 12.5 Solution: [Pg. 640]

When the parent forks the child, it gets a copy of the connected descriptor and the reference count for the associated file table in incremented from 1 to 2. When the parent closes its copy of the descriptor, the reference count is decremented from 2 to 1. Since the kernel will not close a file until the reference counter in its file table goes to zero, the child's end of the connection stays open.

Problem 12.6 Solution: [Pg. 640]

When a process terminates for any reason, the kernel closes all open descriptors. Thus, the child's copy of the connected file descriptor will be closed automatically when the child exits.

Problem 12.7 Solution: [Pg. 652]

The reason that standard I/O works in CGI programs is that we never have to explicitly close the standard input and output streams. When the child exits, the kernel will close streams and their associated file descriptors automatically.

Problem 12.8 Solution: [Pg. 662]

A. The `doit` function is not reentrant, because it and its subfunctions use the non-reentrant `readline` function.

B. To make Tiny reentrant, we must replace all calls to `readline` with its reentrant counterpart `readline_r`, being carefull to call `readline_rinit` in `doit` before the first call to `readline_r`.

Bibliography

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] V. Bala, E. Duesterwald, and S. Banerjiia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2000.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol - HTTP/1.0. RFC 1945, 1996.

[4] A. Birrell. An introduction to programming with threads. Technical Report Report 35, Digital Systems Research Center, 1989.

[5] F. P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, 1979.

[6] A. Demke Brown and T. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, October 2000.

[7] B. R. Buck and J.K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–324, June 2000.

[8] D. Butenhof. *Programming with Posix Threads*. Addison-Wesley, 1997.

[9] S. Carson and P. Reynolds. The geometry of semaphore programs. *ACM Transactions on Programming Languages and Systems*, 9(1):25–53, 1987.

[10] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA)*, pages 70–79, January 1999.

[11] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), June 1994.

[12] S. Chen, P. Gibbons, and T. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD Conference*. ACM, May 2001.

[13] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, May 1999.

[14] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[15] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971.

[16] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, 14(10):48–54, October 1981.

[17] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1999. Order Number 243190.
Also available at `http://developer.intel.com/`.

[18] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Order Number 243191.
Also available at `http://developer.intel.com/`.

[19] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, March 2000.

[20] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA)*, Atlanta, GA, May 1999. IEEE.

[21] B. Davis, B. Jacob, and T. Mudge. The new DRAM interfaces: SDRAM, RDRAM, and variants. In *Proceedings of the Third International Symposium on High Performance Computing (ISHPC)*, Tokyo, Japan, October 2000.

[22] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[23] C. Ding and K. Kennedy. Improving cache performance of dynamic applications through data and computation reorganizations at run time. In *Proceedings of the 1999 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241. ACM, May 1999.

[24] M. W. Eichen and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November, 1988. In *IEEE Symposium on Research in Security and Privacy*, 1989.

[25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol - HTTP/1.1. RFC 2616, 1999.

[26] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, October 1998.

[27] G. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11), November 2000.

[28] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2), February 1995.

[29] L. Gwennap. New algorithm improves branch prediction. *Microprocessor Report*, 9(4), March 1995.

[30] S. P. Harbison and G. L. Steele, Jr. *C, A Reference Manual*. Prentice-Hall, 1995.

[31] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan-Kaufmann, San Francisco, 1996.

[32] Intel. *Tool Interface Standards Portable Formats Specification, Version 1.1*, 1993. Order number 241597.
Also available at `http://developer.intel.com/vtune/tis.htm`.

[33] F. Jones, B. Prince, R. Norwood, J. Hartigan, W. Vogley, C. Hart, and D. Bondurant. A new era of fast dynamic RAMs. *IEEE Spectrum*, pages 43–39, October 1992.

[34] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[35] M. Kaashoek, D. Engler, G. Ganger, H. Briceo, R. Hunt, D. Maziers, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles (SOSP)*, October 1997.

[36] R. Katz. *Contemporary Logic Design*. Addison-Wesley, 1993.

[37] B. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[38] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[39] T. Kilburn, B. Edwards, M. Lanigan, and F. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962.

[40] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition*. Addison-Wesley, 1973.

[41] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2000.

[42] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, April 1991.

[43] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[44] J. R. Levine. *Linkers and Loaders*. Morgan-Kaufmann, San Francisco, 1999.

[45] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the 2000 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 157–168. ACM, June 2000.

[46] J. L. Lions. Ariane 5 Flight 501 failure. Technical report, European Space Agency, July 1996. Available as http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

[47] S. Macguire. *Writing Solid Code*. Microsoft Press, 1993.

[48] J. Markoff. Microsoft caught in 'dirty tricks' vs. AOL. *New York Times*, August 16 1999.

[49] E. Marshall. Fatal error: How Patriot overlooked a Scud. *Science*, page 1347, March 13 1992.

[50] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, March 1986.

[51] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, October 1992.

[52] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, 1997.

[53] M. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, 2001.

[54] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1998 ACM SIGMOD Conference*. ACM, June 1988.

[55] L. Peterson and B. Davies. *Computer Networks: A Systems Approach, Third Edition*. Morgan-Kaufmann, 1999.

[56] S. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan-Kaufmann, 1990.

[57] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[58] J. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 1996.

[59] D. Ritchie. The evolution of the Unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(6 Part 2):1577–1593, October 1984.

[60] D. Ritchie. The development of the C language. In *Proceedings of the Second History of Programming Languages Conference*, Cambridge, MA, April 1993.

[61] D. Ritchie and K. Thompson. The Unix time-sharing system. *Communications of the ACM*, 17(7):365–367, July 1974.

[62] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, Washington, August 1997.

[63] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[64] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, School of Computer Science, Carnegie Mellon University, 1999.

[65] B. Shriver and B. Smith. *Anatomy of a High-Performance Processor*. IEEE Computer Society, 1998.

[66] A. Silberschatz and P. Galvin. *Operating Systems Concepts, Fifth Edition*. John Wiley & Sons, 1998.

[67] R. Skeel. Roundoff error and the Patriot missile. *SIAM News*, 25(4):11, July 1992.

[68] A. Smith. Cache memories. *ACM Computing Surveys*, 14(3), September 1982.

[69] E. H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, 1988.

[70] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

[71] W. Stallings. *Operating Systems: Internals and Design Principles, Fourth Edition*. Prentice Hall, 2000.

[72] W. Richard Stevens. *Advanced Programming in the Unix Environment*. Addison-Wesley, 1992.

[73] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison-Wesley, 1994.

[74] W. Richard Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, 1995.

[75] W. Richard Stevens. *TCP/IP Illustrated: TCP for Transactions, HTTP, NNTP and the Unix domain protocols*, volume 3. Addison-Wesley, 1996.

[76] W. Richard Stevens. *Unix Network Programming: Interprocess Communications, Second Edition*, volume 2. Prentice-Hall, 1998.

[77] W. Richard Stevens. *Unix Network Programming: Networking APIs, Second Edition*, volume 1. Prentice-Hall, 1998.

[78] T. Stricker and T. Gross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179, San Antonio, TX, February 1997. IEEE.

[79] A. Tanenbaum. *Modern Operating Systems, Second Edition*. Prentice Hall, 2001.

[80] A. Tannenbaum. *Computer Networks, Third Edition*. Prentice-Hall, 1996.

[81] K. P. Wadleigh and I. L. Crawford. *Software Optimization for High-Performance Computing: Creating Faster Applications*. Prentice-Hall, 2000.

[82] J. F. Wakerly. *Digital Design Principles and Practices, Third Edition*. Prentice-Hall, 2000.

[83] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2), April 1965.

[84] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, Kinross, Scotland, 1995.

[85] M. Wolf and M. Lam. A data locality algorithm. In *Conference on Programming Language Design and Implementation (SIGPLAN)*, pages 30–44, June 1991.

[86] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer*, August 2000.

[87] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–26, October 1997.

Index

Chapter 4

Processor Architecture

Modern microprocessors are among the most complex systems ever created by humans. A single silicon chip, roughly the size of a fingernail, can contain a complete, high-performance processor, large cache memories, and the logic required to interface it to external devices. In terms of performance, the processors implemented on a single chip today dwarf the room-sized supercomputers that cost over $10 million just 20 years ago. Even the embedded processors found in everyday appliances such as cell phones, personal digital assistants, and handheld game systems are far more powerful than the early developers of computers ever envisioned.

Thus far, we have only viewed computer systems down to the level of machine-language programs. We have seen that a processor must execute a sequence of instructions, where each instruction performs some primitive operation, such as adding two numbers. An instruction is encoded in binary form as a sequence of one or more bytes. The instructions supported by a particular processor and their byte-level encodings are known as its *instruction-set architecture* (ISA). Different "families" of processors, such as Intel IA32, IBM/Motorola PowerPC, and Sun Microsystems SPARC have different ISAs. A program compiled for one type of machine will not run on another. On the other hand, there are many different models of processors within a single family. Each manufacturer produces processors of ever-growing performance and complexity, but the different models remain compatible at the ISA level. Popular families, such as IA32, have processors supplied by multiple manufacturers. Thus, the ISA provides a conceptual layer of abstraction between compiler writers, who need only know what instructions are permitted and how they are encoded, and processor designers, who must build machines that execute those instructions.

In this chapter, we take a brief look at the design of processor hardware. We study the way a hardware system can execute the instructions of a particular ISA. This view will give you a better understanding of how computers work and the technological challenges faced by computer manufacturers. One important concept is that the actual way a modern processor operates can be quite different from the model of computation implied by the ISA. The ISA model would seem to imply *sequential* instruction execution, where each instruction is fetched and executed to completion before the next one begins. By executing different parts of multiple instructions simultaneously, the processor can achieve higher performance than if it executed just one instruction at a time. Special mechanisms are used to make sure the processor computes the same results as it would with sequential execution. This idea of using clever tricks to improve performance while maintaining the functionality of a simpler and more abstract model is well known in computer science.

Examples include the use of caching in Web browsers and information retrieval data structures such as balanced binary trees and hash tables.

Chances are you will never design your own processor. This is a task for experts working at fewer than 100 companies worldwide. Why, then, should you learn about processor design?

- *It is intellectually interesting.* There is an intrinsic value in learning how things work. It is especially interesting to learn the inner workings of a system that is such a part of the daily lives of computer scientists and engineers and yet remains a mystery to many. Processor design embodies many of the principles of good engineering practice. It requires creating as simple a structure as possible to perform a complex task.
- *Understanding how the processor works aids in understanding how the overall computer system works.* In Chapter 6, we will look at the memory system and the techniques used to create an image of a very large memory with a very fast access time. Seeing the processor side of the processor-memory interface will make this presentation more complete.
- *Although few people design processors, many design hardware systems containing processors.* This has become commonplace as processors are embedded into real-world systems such as automobiles and appliances. Embedded system designers must understand how processors work, because these systems are generally designed and programmed at a lower level of abstraction than is the case for desktop systems.
- *You just might work on a processor design.* Although the number of companies producing microprocessors is small, the design teams working on those processors are already large and growing. There can be over 800 people involved in the different aspects of a major processor design.

In this chapter, we start by defining a simple instruction set that we use as a running example for our processor implementations. We call this the "Y86" instruction set, because it was inspired by the IA32 instruction set, which is colloquially referred to as "X86." Compared with IA32, the Y86 instruction set has fewer data types, instructions, and addressing modes. It also has a simpler byte-level encoding. Still, it is sufficiently complete to allow us to write simple programs manipulating integer data. Designing a processor to implement Y86 requires us to face many of the challenges faced by processor designers.

We then provide some background on digital hardware design. We describe the basic building blocks used in a processor and how they are connected together and operated. This presentation builds on our discussion of Boolean algebra and bit-level operations from Chapter 2. We also introduce a simple language, HCL (for "Hardware Control Language") to describe the control portions of hardware systems. We will later use this language to describe our processor designs. Even if you already have some background in logic design, read this section to understand our particular notation.

As a first step in designing a processor, we present a functionally correct, but somewhat impractical, Y86 processor based on *sequential* operation. This processor executes a complete Y86 instruction on every clock cycle. The clock must run slowly enough to allow an entire series of actions to complete within one cycle. Such a processor could be implemented, but its performance would be well below what could be achieved for this much hardware.

With the sequential design as a basis, we then apply a series of transformations to create a *pipelined* processor. This processor breaks the execution of each instruction into five steps, each of which is handled

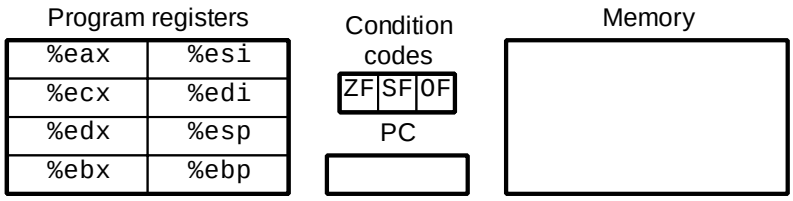


Figure 4.1: **Y86 programmer-visible state.** As with IA32, programs for Y86 access and modify the program registers, the condition code, the program counter (PC), and the memory.

by a separate section or *stage* of the hardware. Instructions progress through the stages of the pipeline, with one instruction entering the pipeline on each clock cycle. As a result, the processor can be executing the different steps of up to five instructions simultaneously. Making this processor preserve the sequential behavior of the Y86 ISA requires handling a variety of *hazard* conditions, where the location or operands of one instruction depend on those of other instructions that are still in the pipeline.

We have devised a variety of tools for studying and experimenting with our processor designs. These include an assembler for Y86, a simulator for running Y86 programs on your machine, and simulators for two sequential and one pipelined processor design. The control logic for these designs is described by files in HCL notation. By editting these files and recompiling the simulator, you can alter and extend the simulation behavior. A number of exercises are provided that involve implementing new instructions and modifying how the machine processes instructions. Testing code is provided to help you evaluate the correctness of your modifications. These exercises will greatly aid your understanding of the material and will give you an appreciation for the many different design alternatives faced by processor designers.

4.1 The Y86 Instruction Set Architecture

As Figure 4.1 illustrates, each instruction in a Y86 program can read and modify some part of the processor state. This is referred to as the *programmer-visible* state, where the "programmer" in this case is either someone writing programs in assembly code or a compiler generating machine-level code. We will see in our processor implementations that we do not need to represent and organize this state in exactly the manner implied by the ISA, as long as we can make sure that machine-level programs appear to have access to the programmer-visible state. The state for Y86 is similar to that for IA32. There are eight *program registers*: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, and `%ebp`. Each of these stores a word. Register `%esp` is used as a stack pointer by the push, pop, call, and return instructions. Otherwise, the registers have no fixed meanings or values. There are three single-bit *condition codes*: `ZF`, `SF`, and `OF`, storing information about the effect of the most recent arithmetic or logical instruction. The program counter (PC) holds the address of the instruction currently being executed. The *memory* is conceptually a large array of bytes, holding both program and data. Y86 programs reference memory locations using *virtual addresses*. A combination of hardware and operating system software translates these into the actual, or *physical* addresses indicating where the values are actually stored in memory. We will study virtual memory in more detail in Chapter 10. For now, we can think of the virtual memory system as providing Y86 programs with an image of a monolithic byte array.

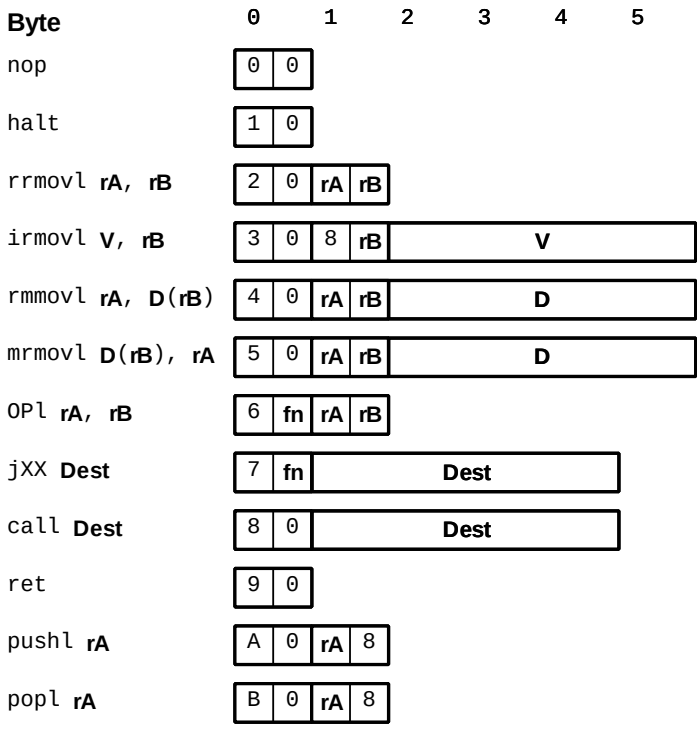


Figure 4.2: **Y86 instruction set.** Instruction encodings range between 1 and 6 bytes. An instruction consists of a one-byte instruction specifier, possibly a one-byte register specifier, and possibly a four-byte constant word. Field `fn` specifies a particular integer operation (`OPl`) or a particular branch condition (`jXX`). All numeric values are shown in hexadecimal.

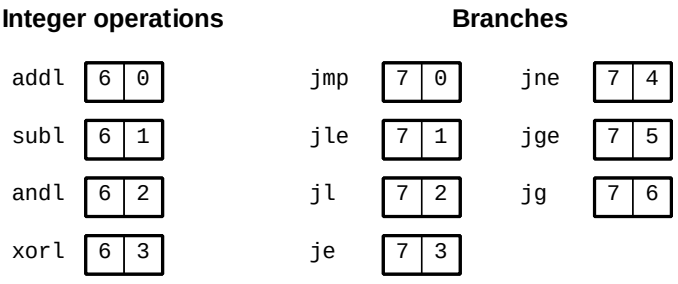


Figure 4.3: **Function codes for Y86 instruction set.** The codes specify a particular integer operation or branch condition. These instructions are shown as `OPl` and `jXX` in Figure 4.2.

Figure 4.2 gives a concise description of the individual instructions in the Y86 ISA. We use this instruction set as a target for our processor implementations. The set of Y86 instructions is largely a subset of the IA32 instruction set. It includes only four-byte integer operations; it has fewer addressing modes; and it includes a smaller set of operations. Since we only use four-byte data, we refer to these as "words." In this figure, we show the assembly code representation of the instructions on the left and the byte encodings on the right. The assembly code is similar to the GAS representation of IA32 programs.

Here are some further details about the different Y86 instructions.

- The IA32 `movl` instruction is split into four different instructions: `irmovl`, `rrmovl`, `mrmovl`, and `rmmovl`, explicitly indicating the form of the source and destination. The source is either immediate (`i`), register (`r`), or memory (`m`). It is designated by the first character in the instruction name. The destination is either register (`r`) or memory (`m`). It is designated by the second character in the instruction name. Explicitly identifying the four types of data transfer will prove helpful when we decide how to implement them.

  The memory references for the two memory movement instructions have a simple base and displacement format. We do not support the second index register or any scaling of the register value in the address computation.

  As with IA32, we do not allow direct transfers from one memory location to another. In addition, we do not allow a transfer of immediate data to memory.

- There are four integer operation instructions, shown in Figure 4.2 as `OPl`. These are `addl`, `subl`, `andl`, and `xorl`. They operate only on register data, whereas IA32 also allows operations on memory data. These instructions set the three condition codes `ZF`, `SF`, and `OF` (zero, sign, and overflow).

- The seven jump instructions (shown in Figure 4.2 as `jXX`) are `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`. Branches are taken according to the type of branch and the settings of the condition codes. The branch conditions are the same as with IA32 (Figure 3.11).

- The `call` instruction pushes the return address on the stack and jumps to the destination address. The `ret` instruction returns from such a call.

- The `pushl` and `popl` instructions implement push and pop, just as they do in IA32.

- The `halt` instruction stops instruction execution. IA32 has a comparable instruction, called `hlt`. IA32 application programs are not permitted to use this instruction, since it causes the entire system to stop. We use `halt` in our Y86 programs to stop the simulator.

Figure 4.2 also shows the byte-level encoding of the instructions. Each instruction requires between one and six bytes, depending on which fields are required. Every instruction has an initial byte identifying the instruction type. This byte is split into two four-bit parts: the high-order or *code* part, and the low-order or *function* part. As you can see in Figure 4.2, code values range from `0` to hexadecimal `B`. The function values are significant only for the cases where a group of related instructions share a common code. These are given in Figure 4.3, showing the specific encodings of the integer operation and branch instructions.

As shown in Figure 4.4, each of the eight program registers has an associated *register identifier* (ID) ranging from 0 to 7. The numbering of registers in Y86 matches what is used in IA32. The program registers are

Number	Register name
0	%eax
1	%ecx
2	%edx
3	%ebx
6	%esi
7	%edi
4	%esp
5	%ebp
8	No register

Figure 4.4: **Y86 program register identifiers.** Each of the eight program registers has an associated identifer (ID) ranging from 0 to 7. ID 8 in a register field of an instruction indicates the absence of a register operand.

stored within the CPU in a *register file*, a small random-access memory where the register IDs serve as addresses. ID value 8 is used in the instruction encodings and within our hardware designs when we need to indicate that no register should be accessed.

Some instructions are just one byte long, but those that require operands have longer encodings. First, there can be an additional *register specifier byte*, specifying either one or two registers. These register fields are called rA and rB in Figure 4.2. As the assembly code versions of the instructions show, they can specify the registers used for data sources and destinations, as well as the base register used in an address computation, depending on the instruction type. Instructions that have no register operands, such as branches and `call`, do not have a register specifier byte. Those that require just one register operand (`irmovl`, `pushl`, and `popl`) have the other register specifier set to value 8. This convention will prove useful in our processor implementation.

Some instructions require an additional four-byte *constant word*. This word can serve as the immediate data for `irmovl`, the displacement for `rmmovl` and `mrmovl` address specifiers, and the destination of branches and calls. Note that branch and call destinations are given as absolute addresses, rather than using the PC-relative addressing seen in IA32. Processors use PC-relative addressing to give more compact encodings of branch instructions and to allow code to be copied from one part of memory to another without the need to update all of the branch target addresses. Since we are more concerned with simplicity in our presentation, we use absolute addressing. As with IA32, all integers have a little-endian encoding. When the instruction is written in disassembled form, these bytes appear in reverse order.

As an example, let us generate the byte encoding of the instruction `rmmovl %esp,0x12345(%edx)` in hexadecimal. From Figure 4.2 we can see that `rmmovl` has initial byte `40`. We can also see that source register `%esp` should be encoded in the rA field, and base register `%edx` should be encoded in the rB field. Using the register numbers in Figure 4.4, we get a register specifier byte of `42`. Finally, the displacement is encoded in the four-byte constant word. We first pad `0x12345` with leading 0s to fill out four bytes, giving a byte sequence of `00 01 23 45`. We write this in byte-reversed order as `45 23 01 00`. Combining these we get an instruction encoding of `404245230100`.

One important property of any instruction set is that the byte encodings must have a unique interpretation.

An arbitrary sequence of bytes either encodes a unique instruction sequence or is not a legal byte sequence. This property holds for Y86, because every instruction has a unique combination of code and function in its initial byte, and given this byte, we can determine the length and meaning of any additional bytes. This property ensures that a processor can execute an object code program without any ambiguity about the meaning of the code. Even if the code is embedded within other bytes in the program, we can readily determine the instruction sequence as long as we start from the first byte in the sequence. On the other hand, if we do not know the starting position of a code sequence, we cannot reliably determine how to split the sequence into individual instructions. This causes problems for disassemblers and other tools that attempt to extract machine-level programs directly from object code byte sequences.

Practice Problem 4.1:

Determine the byte encoding of the Y86 instruction sequence that follows. The line "`.pos 0x100`" indicates that the starting address of the object code should be `0x100`.

```
.pos 0x100  # Start generating code at address 0x100
    irmovl $15,%ebx
    rrmovl %ebx,%ecx
loop:
    rmmovl %ecx,-3(%ebx)
    addl   %ebx,%ecx
    jmp loop
```

Practice Problem 4.2:

For each byte sequence listed, determine the Y86 instruction sequence it encodes. If there is some invalid byte in the sequence, show the instruction sequence up to that point and indicate where the invalid value occurs. For each sequence, we show the starting address, then a colon, and then the byte sequence.

A. `0x100:3083fcffffff40630008000010`
B. `0x200:a06880080200001030830a00000090`
C. `0x300:50540700000000f0b018`
D. `0x400:6113730004000010`
E. `0x500:6362a080`

Aside: Comparing IA32 to Y86 Instruction Encodings
Compared with the instruction encodings used in IA32, the encoding of Y86 is much simpler but also less compact. The register fields only occur in fixed positions in all Y86 instructions, whereas they are packed into various positions in the different IA32 instructions. We use a four-bit encoding of registers, even though there are only eight possible registers. IA32 uses just 3 bits. Thus, IA32 can pack a push or pop instruction into just one byte, with a 5-bit field indicating the instruction type and the remaining 3 bits for the register specifier. IA32 can encode constant values in 1, 2, or 4 bytes, whereas Y86 always requires 4 bytes. **End Aside.**

Aside: RISC and CISC Instruction Sets
IA32 is sometimes labeled as a "complex instruction set computer" (CISC—pronounced "sisk"), and is deemed to be the opposite of ISAs that are classified as "reduced instruction set computers" (RISC—pronounced "risk").

Historically, CISC machines came first, having evolved from the earliest computers. By the early 1980s, instruction sets for mainframe and minicomputers had grown quite large, as machine designers incorporated new instructions to support high-level tasks, such as manipulating circular buffers, peforming decimal arithmetic, and evaluating polynomials. The first microprocessors appeared in the early 1970s and had limited instruction sets, because the integrated circuit technology then posed severe constraints on what could be implemented on a single chip. Microprocessors evolved quickly and, by the early 1980s, were following the path of increasing instruction-set complexity set by mainframes and minicomputers. The 80x86 family took this path, evolving into IA32. Even IA32 continues to evolve as new classes of instructions are added to support the processing required by multimedia applications.

The RISC design philosophy developed in the early 1980s as an alternative to these trends. A group of hardware and compiler experts at IBM, strongly influenced by the ideas of IBM researcher John Cocke, recognized that they could generate efficient code for a much simpler form of instruction set. In fact, many of the high-level instructions that were being added to instruction sets were very difficult to generate with a compiler and were seldom used. A simpler instruction set could be implemented with much less hardware and could be organized in an efficient pipeline structure, similar to those described later in this chapter. IBM did not commercialize this idea until many years later, when it developed the Power and PowerPC ISAs.

The RISC concept was further developed by Professors David Patterson, of the University of California at Berkeley, and John Hennessy, of Stanford University. Patterson gave the name RISC to this new class of machines, and CISC to the existing class, since there had previously been no need to have a special designation for a nearly universal form of instruction set.

Comparing CISC with the original RISC instruction sets, we find the following general characteristics:

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [19] is over 700 pages long.	Many fewer instructions. Typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings. IA32 instructions can range from 1 to 15 bytes.	Fixed length encodings. Typically all instructions are encoded as four bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifi er can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacment addressing.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logial operations only use register operands. Memory referencing is only allowed by *load* instructions, reading from memory into a register, and *store* instructions, writing from a register to memory. This convention is referred to as a *load/store architecture*.
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions that store the test result in a normal register are used for conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addressess. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

The Y86 instruction set includes attributes of both CISC and RISC instruction sets. On the CISC side, it has condition codes, variable-length instructions, and stack-intensive procedure linkages. On the RISC side, it uses a load-store architecture and a regular encoding. It can be viewed as taking a CISC instruction set (IA32) and simplifying it by applying some of the principles of RISC. **End Aside.**

Aside: The RISC Versus CISC Controversy

Through the 1980s, battles raged in the computer architecture community regarding the merits of RISC versus CISC instruction sets. Proponents of RISC claimed they could get more computing power for a given amount of hardware through a combination of streamlined instruction set design, advanced compiler technology, and pipelined processor implementation. CISC proponents countered that fewer CISC instruction were required to perform a given task, and so their machines could achieve higher overall performance.

Major companies introduced RISC processor lines, including Sun Microsystems (SPARC), IBM and Motorola (PowerPC), and Digital Equipment Corporation (Alpha).

In the early 1990s, the debate diminished as it became clear that neither RISC nor CISC in their purest forms were better than designs that incorporated the best ideas of both. RISC machines evolved and introduced more instructions, many of which take multiple cycles to execute. RISC machines today have hundreds of instructions in their repertoire, hardly fi tting the name "reduced instruction set machine." The idea of exposing implementation

IA32 code

```
     int Sum(int *Start, int Count)
 1 Sum:
 2   pushl %ebp
 3   movl %esp,%ebp
 4   movl 8(%ebp),%ecx    ecx = Start
 5   movl 12(%ebp),%edx   edx = Count
 6   xorl %eax,%eax       sum = 0
 7   testl %edx,%edx
 8   je .L34
 9 .L35:
10   addl (%ecx),%eax     add *Start to sum
11   addl $4,%ecx         Start++
12   decl %edx            Count--
13   jnz .L35             Stop when 0
14 .L34:
15   movl %ebp,%esp
16   popl %ebp
17   ret
```

Y86 code

```
     int Sum(int *Start, int Count)
 1 Sum:    pushl %ebp
 2   rrmovl %esp,%ebp
 3   mrmovl 8(%ebp),%ecx       ecx = Start
 4   mrmovl 12(%ebp),%edx      edx = Count
 5   irmovl $0, %eax           sum = 0
 6   andl   %edx,%edx
 7   je     End
 8 Loop:   mr-
movl (%ecx),%esi        get *Start
 9   addl %esi,%eax            add to sum
10   irmovl $4,%ebx
11   addl %ebx,%ecx            Start++
12   irmovl $-1,%ebx
13   addl %ebx,%edx            Count-
                               -
14   jne    Loop               Stop when 0
15 End:
16   popl %ebp
17   ret
```

Figure 4.5: **Comparison of Y86 and IA32 assembly programs.** The Sum function computes the sum of an integer array. The Y86 code differs from the IA32 mainly in that it may require multiple instructions to perform what can be done with a single IA32 instruction.

artifacts to machine-level programs proved to be short-sighted. As new processor models were developed using more advanced hardware structures, many of these artifacts became irrelevant, but they still remained part of the instruction set. Still, the core of RISC design is an instruction set that is well-suited to execution on a pipelined machine.

More recent CISC machines also take advantage of high-performance pipeline structures. As we will discuss in Section 5.7, they fetch the CISC instructions and dynamically translate them into a sequence of simpler, RISC-like operations. For example, an instruction that adds a register to memory is translated into three operations: one to read the original memory value, one to perform the addition, and a third to write the sum to memory. Since the dynamic translation can generally be performed well in advance of the actual instruction execution, the processor can sustain a very high execution rate.

Marketing issues, apart from technological ones, have also played a major role in determining the success of different instruction sets. By maintaining compatibility with its existing processors, Intel with IA32 made it easy to keep moving from one generation of processor to the next. As integrated circuit technology improved, Intel and other IA32 processor manufacturers could overcome the ineffi ciencies created by the original 8086 instruction-set design, using RISC techniques to produce performance comparable to the best RISC machines. In the areas of desktop and laptop computing, IA32 has achieved total domination.

RISC processors have done very well in the market for *embedded processors*, controlling such systems as cellular telephones, automobile brakes, and Internet appliances. In these applications, saving on cost and power is more important than maintaining backward compatibility. In terms of the number of processors sold, this is a very large and growing market. **End Aside.**

Figure 4.5 shows IA32 and Y86 assembly code for the following C function:

```
int Sum(int *Start, int Count)
{
  int sum = 0;
  while (Count) {
    sum += *Start;
    Start++;
    Count--;
  }
  return sum;
}
```

The IA32 code was generated by the C compiler GCC. The Y86 code is essentially the same, except that Y86 sometimes requires two instructions to accomplish what can be done with a single IA32 instruction. If we had written the program using array indexing, however, the conversion to Y86 code would be more difficult, since Y86 does not have scaled addressing modes.

Figure 4.6 shows an example of a complete program file written in Y86 assembly code. The program contains both data and instructions. Directives indicate where to place code or data and how to align it. The program specifies issues such as stack placement, data initialization, program initialization, and program termination.

In this program, words beginning with "`.`" are *assembler directives* telling the assembler to adjust the address at which it is generating code or to insert some words of data. The directive `.pos 0` (line 2) indicates that the assembler should begin generating code starting at address 0. This is the starting point of all Y86 programs. The next two instructions (lines 3 and 4) initialize the stack and frame pointers. We can see that the label `Stack` is declared at the end of the program (line 39), to indicate address `0x100` using a `.pos` directive (line 38). Our stack will therefore start at this address and grow downward.

Lines 8 to 12 of the program declare an array of four words, having values `0xd`, `0xc0`, `0xb00`, and `0xa000`. The label `array` denotes the start of this array, and is aligned on a four-byte boundary (using the `.align` directive). Lines 14 to 19 show a "main" procedure that calls the function `Sum` on the four-word array and then halts.

As this example shows, writing a program in Y86 requires the programmer to perform tasks we ordinarily assign to the compiler, linker, and run-time system. Fortunately, we only do this for small programs for which simple mechanisms suffice.

Figure 4.7 shows the result of assembling the code shown in Figure 4.6 by an assembler we call YAS. The assembler output is in ASCII format to make it more readable. On lines of the assembly file that contain instructions or data, the object code contains an address, followed by the values of between 1 and 6 bytes.

We have implemented an instruction set simulator we call YIS. Running on our sample object code, it generates the following output:

```
Stopped in 46 steps at PC = 0x3a.  Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x000000f8
```

code/arch/y86-code/asum.ys

```
1  # Execution begins at address 0
2          .pos 0
3  init:   irmovl Stack, %esp      # Set up Stack pointer
4          irmovl Stack, %ebp      # Set up base pointer
5          jmp Main                # Execute main program
6
7  # Array of 4 elements
8          .align 4
9  array:  .long 0xd
10         .long 0xc0
11         .long 0xb00
12         .long 0xa000
13
14 Main:   irmovl $4,%eax
15         pushl %eax      # Push 4
16         irmovl array,%edx
17         pushl %edx      # Push array
18         call Sum        # Sum(array, 4)
19         halt
20
21         # int Sum(int *Start, int Count)
22 Sum:    pushl %ebp
23         rrmovl %esp,%ebp
24         mrmovl 8(%ebp),%ecx     # ecx = Start
25         mrmovl 12(%ebp),%edx    # edx = Count
26         irmovl $0, %eax         # sum = 0
27         andl   %edx,%edx
28         je     End
29 Loop:   mrmovl (%ecx),%esi      # get *Start
30         addl %esi,%eax          # add to sum
31         irmovl $4,%ebx          #
32         addl %ebx,%ecx          # Start++
33         irmovl $-1,%ebx         #
34         addl %ebx,%edx          # Count--
35         jne    Loop             # Stop when 0
36 End:
37         popl %ebp
38         ret
39         .pos 0x100
40 Stack:  # The stack goes here
```

code/arch/y86-code/asum.ys

Figure 4.6: **Sample program written in Y86 assembly code.** The Sum function is called to compute the sum of a 4-element array.

code/arch/y86-code/asum.yo

```
                    | # Execution begins at address 0
  0x000:            |         .pos 0
  0x000: 308400010000 | init:   irmovl Stack, %esp      # Set up Stack pointer
  0x006: 308500010000 |         irmovl Stack, %ebp      # Set up base pointer
  0x00c: 7024000000   |         jmp Main                # Execute main program
                    |
                    | # Array of 4 elements
  0x014:            |         .align 4
  0x014: 0d000000     | array:  .long 0xd
  0x018: c0000000     |         .long 0xc0
  0x01c: 000b0000     |         .long 0xb00
  0x020: 00a00000     |         .long 0xa000
                    |
  0x024: 308004000000 | Main:   irmovl $4,%eax
  0x02a: a008         |         pushl %eax      # Push 4
  0x02c: 308214000000 |         irmovl array,%edx
  0x032: a028         |         pushl %edx      # Push array
  0x034: 803a000000   |         call Sum        # Sum(array, 4)
  0x039: 10           |         halt
                    |
                    |         # int Sum(int *Start, int Count)
  0x03a: a058         | Sum:    pushl %ebp
  0x03c: 2045         |         rrmovl %esp,%ebp
  0x03e: 501508000000 |         mrmovl 8(%ebp),%ecx     # ecx = Start
  0x044: 50250c000000 |         mrmovl 12(%ebp),%edx    # edx = Count
  0x04a: 308000000000 |         irmovl $0, %eax         # sum = 0
  0x050: 6222         |         andl   %edx,%edx
  0x052: 7374000000   |         je     End
  0x057: 506100000000 | Loop:   mrmovl (%ecx),%esi      # get *Start
  0x05d: 6060         |         addl %esi,%eax          # add to sum
  0x05f: 308304000000 |         irmovl $4,%ebx          #
  0x065: 6031         |         addl %ebx,%ecx          # Start++
  0x067: 3083ffffffff |         irmovl $-1,%ebx         #
  0x06d: 6032         |         addl %ebx,%edx          # Count--
  0x06f: 7457000000   |         jne    Loop             # Stop when 0
  0x074:            | End:
  0x074: b058         |         popl %ebp
  0x076: 90           |         ret
  0x100:            |         .pos 0x100
  0x100:            | Stack:  # The stack goes here
```

code/arch/y86-code/asum.yo

Figure 4.7: **Output of YAS assembler.** Each line includes a hexadecimal address and between 1 and 6 bytes of object code.

```
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000

Changes to memory:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
```

The simulator only prints out words that change during simulation, either in registers or in memory. The original values (here they are all 0) are shown on the left, and the final values are shown on the right. We can see in this output that register `%eax` contains `0xabcd`, the sum of the four-element array passed to subroutine `Sum`. In addition, we can see that the stack, which starts at address `0x100` and grows downward, has been used, causing changes to memory at addresses `0xf0` through `0xfc`.

Practice Problem 4.3:

Write Y86 code to implement a recursive sum function `rSum`, based on the following C code:

```
int rSum(int *Start, int Count)
{
    if (Count <= 0)
        return 0;
    return *Start + rSum(Start+1, Count-1);
}
```

You might find it helpful to compile the C code on an IA32 machine and then translate the instructions to Y86.

Practice Problem 4.4:

The `pushl` instruction both decrements the stack pointer by 4 and writes a register value to memory. It is not totally clear what the processor should do with the instruction `pushl %esp`, since the register being pushed is being changed by the same instruction. Two conventions are possible: (1) push the original value of `%esp`, or (2) push the decremented value of `%esp`.

Let's resolve this issue by doing the same thing an IA32 processor would do. We could try reading the Intel documentation on this instruction, but a simpler approach is to conduct an experiment on an actual machine. The C compiler would not normally generate this instruction, so we must use hand-generated assembly code for this task. As described in Section 3.15, the best way to insert small amounts of assembly code into a C program is to use the `asm` feature of GCC. Here is a test program we have written. Rather than attempting to read the `asm` declaration, you will find it easiest to read the assembly code in the comment preceding it.

```
int pushtest()
{
    int rval;
    /* Insert the following assembly code:
       movl %esp,%eax   # Save stack pointer
```

```
       pushl %esp       # Push stack pointer
       popl  %edx       # Pop it back
       subl  %edx,%eax  # 0 or 4
       movl  %eax,rval  # Set as return value
    */
  asm("movl %%esp,%%eax;pushl %%esp;popl %%edx;subl %%edx,%%eax;movl %%eax,%0"
        : "=r" (rval)
        : /* No Input */
        : "%edx", "%eax");
    return rval;
}
```

In our experiments, we find that the function `pushtest` returns 0. What does this imply about the behavior of the instruction `pushl %esp` under IA32?

Practice Problem 4.5:

A similar ambiguity occurs for the instruction `popl %esp`. It could either set `%esp` to the value read from memory or to the incremented stack pointer. As with Practice Problem 4.4, let us run an experiment to determine how an IA32 machine would handle this instruction and then design our Y86 machine to follow the same convention.

```
int poptest(int tval)
{
    int rval;
    /* Insert the following assembly code:
       pushl tval     # Save tval on stack
       movl %esp,%edx # Save stack pointer
       popl %esp      # Pop to stack pointer.
       movl %esp,rval # Set popped value as return value
       movl %edx,%esp # Restore original stack pointer
    */
    asm("pushl %1; movl %%esp,%%edx; popl %%esp; movl %%esp,%0; movl %%edx,%%esp"
        : "=r" (rval)
        : "r" (tval)
        : "%edx");
    return rval;
}
```

We find this function always returns `tval`, the value passed to it as its argument. What does this imply about the behavior of `popl %esp`? What other Y86 instruction would have the exact same behavior?

4.2 Logic Design and the Hardware Control Language HCL

In hardware design, electronic circuits are used to compute functions on bits and to store bits in different kinds of memory elements. Most contemporary circuit technology represents different bit values as high or

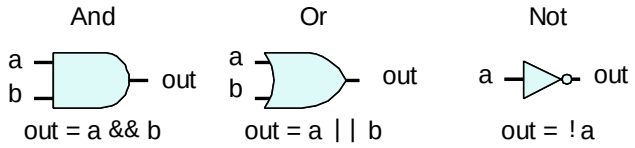


Figure 4.8: **Logic gate types.** Each gate generates output equal to some Boolean function of its inputs.

low voltages on signal wires. In current technology, logic value 1 is represented by a high voltage of around 1.0 volt, while logic value 0 is represented by a low voltage of around 0.0 volts. Three major components are required to implement a digital system: combinational logic to compute functions on the bits, memory elements to store bits, and clock signals to regulate the updating of the memory elements.

In this section, we provide a brief description of these different components. We also introduce HCL (for "hardware control language"), the language that we use to describe the control logic of the different processor designs. We only describe HCL informally here. A complete reference for HCL can be found in Appendix A.

Aside: Modern Logic Design
At one time hardware designers created circuit designs by drawing schematic diagrams of logic circuits (first with paper and pencil and later with computer graphics terminals). Nowadays, most designs are expressed in a *hardware description language* (HDL), a textual notation that looks similar to a programming language but that is used to describe hardware structures rather than program behaviors. The most commonly used languages are Verilog, having a syntax similar to C, and VHDL, having a syntax similar to the Ada programming language. These languages were originally designed for expressing simulation models of digital circuits. In the mid-1980s, researchers developed *logic synthesis* programs that could generate efficient circuit designs from HDL descriptions. There are now a number of commercial synthesis programs, and this has become the dominant technique for generating digital circuits. This shift from hand-designed circuits to synthesized ones can be likened to the shift from writing programs in assembly code to writing them in a high-level language and having a compiler generate the machine code. **End Aside.**

4.2.1 Logic Gates

Logic gates are the basic computing elements for digital circuits. They generate an output equal to some Boolean function of the bit values at their inputs. Figure 4.8 shows the standard symbols used for Boolean functions AND, OR, and NOT. HCL expressions are shown below the gates for the Boolean operations. As you can see, we adopt the syntax for logic operators in C (Section 2.1.9): && for AND, || for OR, and ! for NOT. We use these instead of the bit-level C operators &, |, and ~, because logic gates operate on single-bit quantities, not entire words.

Logic gates are always active. If some input to a gate changes, then within some small amount of time, the output will change accordingly.

4.2.2 Combinational Circuits and HCL Boolean Expressions

By assembling a number of logic gates into a network, we can construct computational blocks known as *combinational circuits*. Two restrictions are placed on how the networks are constructed:

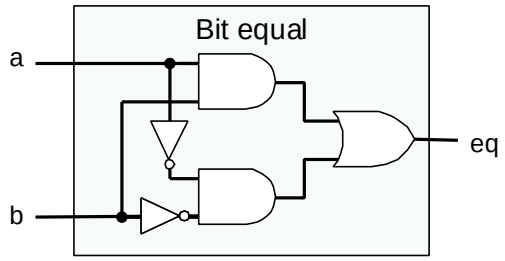


Figure 4.9: **Combinational circuit to test for bit equality.** The output will equal 1 when both inputs are 0, or both are 1.

- The outputs of two or more logic gates cannot be connected together. Otherwise the two could try to drive the wire in opposite directions, possibly causing an invalid voltage or a circuit malfunction.
- The network must be *acyclic*. That is, there cannot be a path through a series of gates that forms a loop in the network. Such loops can cause ambiguity in the function computed by the network.

Figure 4.9 shows an example of a simple combinational circuit that we will find useful. It has two inputs, a and b. It generates a single output eq, such that the output will equal 1 if either a and b are both 1 (detected by the upper AND gate) or are both 0 (detected by the lower AND gate). We write the function of this network in HCL as:

```
bool eq = (a && b) || (!a && !b);
```

This code simply defines the bit-level (denoted by data type `bool`) signal eq as a function of inputs a and b. As this example shows HCL uses C-style syntax, with '=' associating a signal name with an expression. Unlike C, however, we do not view this as performing a computation and assigning the result to some memory location. Instead, it is simply a way to give a name to an expression.

Practice Problem 4.6:

Write an HCL expression for a signal xor, equal to the EXCLUSIVE-OR of inputs a and b. What is the relation between the signals xor and eq defined above?

Figure 4.10 shows another example of a simple but useful combinational circuit known as a *multiplexor*. A multiplexor selects a value from among a set of different data signals, depending on the value of a control input signal. In this single-bit multiplexor, the two data signals are the input bits a and b, while the control signal is the input bit s. The output will equal a when s is 1, and it will equal b when s is 0. In this circuit, we can see that the two AND gates determine whether to pass their respective data inputs to the OR gate. The upper AND gate passes signal b when s is 0 (since the other input to the gate is !s), while the lower AND gate passes signal a when s is 1. Again, we can write an HCL expression for the output signal, using the same operations as are present in the combinational circuit:

```
bool out = (s && a) || (!s && b);
```

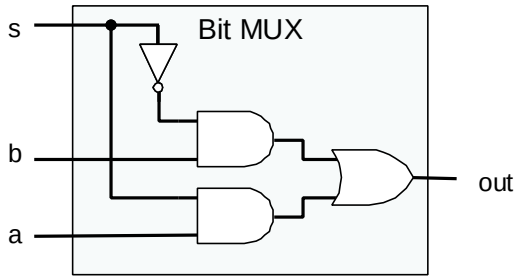


Figure 4.10: **Single-bit multiplexor circuit.** The output will equal input a if the control signal s is 1 and will equal input b when s is 0.

Our HCL expressions demonstrate a clear parallel between combinational logic circuits and logical expressions in C. They both use Boolean operations to compute functions over their inputs. Several differences between these two ways of expressing computation are worth noting:

- Since a combinational circuit consists of a series of logic gates, it has the property that the outputs continually respond to changes in the inputs. If some input to the circuit changes, then after some delay, the outputs will change accordingly. In contrast, a C expression is only evaluated when it is encountered during the execution of a program.
- Logical expressions in C allow arguments to be arbitrary integers, interpreting 0 as FALSE and anything else as TRUE. In contrast, our logic gates only operate over the bit values 0 and 1.
- Logical expressions in C have the property that they might only be partially evaluated. If the outcome of an AND or OR operation can be determined by just evaluating the first argument, then the second argument will not be evaluated. For example, with the C expression:

  ```
  (a && !a) && func(b,c)
  ```

  the function `func` will not be called, because the expression `(a && !a)` evaluates to 0. In contrast, combinational logic does not have any partial evaluation rules. The gates simply respond to changes on their inputs.

4.2.3 Word-Level Combinational Circuits and HCL Integer Expressions

By assembling large networks of logic gates, we can construct combinational circuits that compute much more complex functions. Typically, we design circuits that operate on data *words*. These are groups of bit-level signals that represent an integer or some control pattern. For example, our processor designs will contain numerous words, with word sizes ranging between 4 and 32 bits, representing integers, addresses, instruction codes, and register identifiers.

Combinational circuits to perform word-level computations are constructed using logic gates to compute the individual bits of the output word, based on the individual bits of the input word. For example, Figure 4.11 shows a combinational circuit that tests whether two 32-bit words A and B are equal. That is, the output will

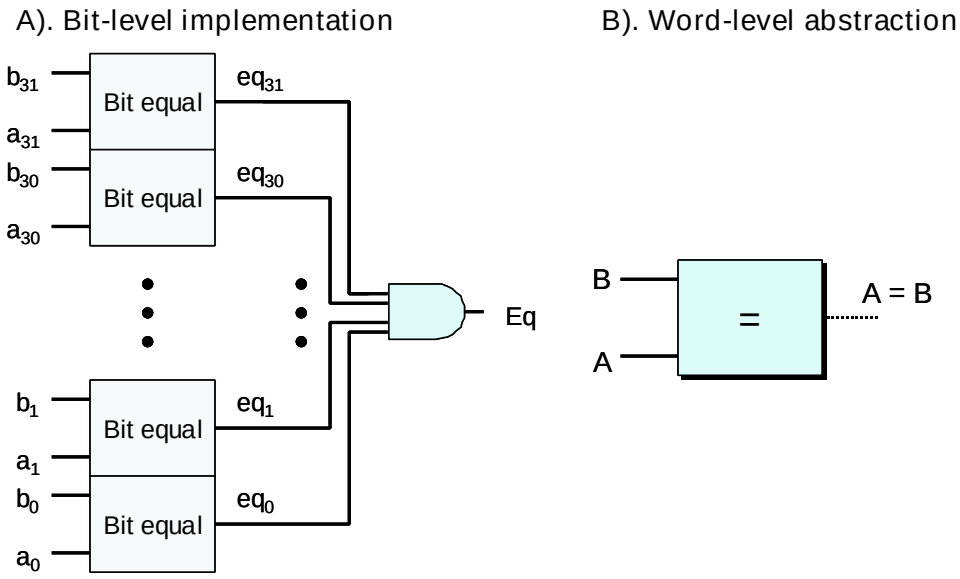


Figure 4.11: **Word-level equality test circuit.** The output will equal 1 when each bit from word A equals its counterpart from word B. Word-level equality is one of the operations in HCL.

equal 1 if and only if each bit of A equals the corresponding bit of B. This circuit is implemented using 32 of the single-bit equality circuits shown in Figure 4.9. The outputs of these single-bit circuits are combined with an AND gate to form the circuit output.

In HCL, we will declare any word-level signal as an `int`, without specifying the word size. This is done for simplicity. In a full-featured hardware description language, every word can be declared to have a specific number of bits. HCL allows words to be compared for equality, and so the functionality of the circuit shown in Figure 4.11 can be expressed at the word level as

```
bool Eq = (A == B);
```

where arguments A and B are of type `int`. Note that we use the same syntax conventions as in C, where '=' denotes assignment, while '==' denotes the equality operator.

As is shown on the right side of Figure 4.11, we will draw word-level circuits using medium-thickness lines to represent the set of wires carrying the individual bits of the word, and we will show the resulting Boolean signal as a dashed line.

Practice Problem 4.7:

Suppose you want to implement a word-level equality circuit using the EXCLUSIVE-OR circuits from Practice Problem 4.6 rather than from bit-level equality circuits. Design such a circuit for a 32-bit word consisting of 32 bit-level EXCLUSIVE-OR circuits and two additional logic gates.

Figure 4.12 shows the circuit for a word-level multiplexor. This circuit generates a 32-bit word Out equal to one of the two input words, A or B, depending on the control input bit s. The circuit consists of 32

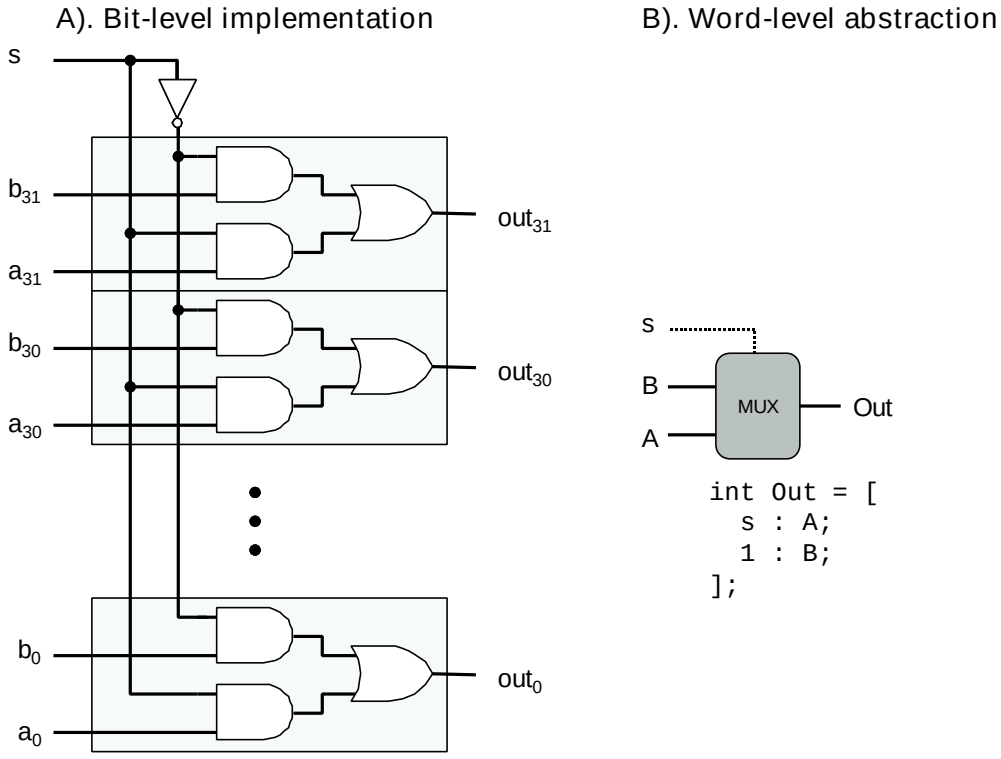


Figure 4.12: **Word-level multiplexor circuit.** The output will equal input word A when the control signal s is 1, and it will equal B otherwise. Multiplexors are described in HCL using case expressions.

identical subcircuits, each having a structure similar to the bit-level multiplexor from Figure 4.10. Rather than simply replicating the bit-level multiplexor 32 times, the word-level version reduces the number of inverters by generating `!s` once and reusing it at each bit position.

We will use many forms of multiplexors in our processor designs. They allow us to select a word from a number of sources depending on some control condition. Multiplexing functions are described in HCL using *case expressions*. A case expression has the following general form:

$$
\begin{array}{l}
\texttt{[} \\
\quad \begin{array}{lcl} select_1 & : & expr_1 \\ select_2 & : & expr_2 \\ & \vdots & \\ select_k & : & expr_k \end{array} \\
\texttt{]}
\end{array}
$$

The expression contains a series of cases, where each case i consists of a Boolean expression $select_i$, indicating when this case should be selected, and an integer expression $expr_i$, indicating the resulting value.

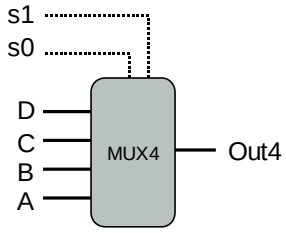
Unlike the switch statement of C, we do not require the different selection expressions to be mutually exclusive. Logically, the selection expressions are evaluated in sequence, and the case for the first one yielding 1 is selected. For example, the word-level multiplexor of Figure 4.12 can be described in HCL as:

```
int Out = [
        s: A;
        1: B;
];
```

In this code, the second selection expression is simply `1`, indicating that this case should be selected if no prior one has been. This is the way to specify a default case in HCL. Nearly all case expressions end in this manner.

Allowing nonexclusive selection expressions makes the HCL code more readable. An actual hardware multiplexor must have mutually exclusive signals controlling which input word should be passed to the output, such as the signals `s` and `!s` in Figure 4.12. To translate an HCL case expression into hardware, a logic synthesis program would need to analyze the set of selection expressions and resolve any possible conflicts by making sure that only the first matching case would be selected.

The selection expressions can be arbitrary Boolean expressions, and there can be an arbitrary number of cases. This allows case expressions to describe blocks where there are many choices of input signals with complex selection criteria. For example, consider the following diagram of a four-way multiplexor:

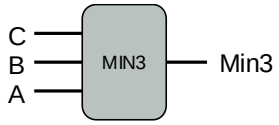


This circuit selects from among the four input words A, B, C, and D based on the control signals s1 and s0, treating the controls as a two-bit binary number. We can express this in HCL using Boolean expressions to describe the different combinations of control bit patterns:

```
int Out4 = [
        !s1 && !s0 : A; # 00
        !s1        : B; # 01
        s1 && !s0  : C; # 10
        1          : D; # 11
];
```

The comments on the right (any text starting with # and running for the rest of the line is a comment) show which combination of s1 and s0 will cause the case to be selected. Observe that the selection expressions can sometimes be simplified, since only the first matching case is selected. For example, the second expression can be written !s1, rather than the more complete !s1 && s0, since the only other possibility having s1 equal to 0 was given as the first selection expression.

As a final example, suppose we want design a logic circuit that finds the minimum value among a set of words A, B, and C, diagrammed as follows:



We can express this using an HCL case expression as

```
int Min3 = [
        A <= B && A <= C : A;
        B <= A && B <= C : B;
        1                : C;
];
```

Practice Problem 4.8:

Write HCL code describing a circuit that for word inputs A, B, and C selects the *median* of the three values. That is, the output equals the word lying between the minimum and maximum of the three inputs.

Combinational logic circuits can be designed to perform many different types of operations on word-level data. The detailed design of these is beyond the scope of our presentation. One important combinational circuit, known as an *arithmetic/logic unit* (ALU), is diagrammed at an abstract level in Figure 4.13. This circuit has three inputs: two data inputs labeled A and B, and a control input. Depending on the setting of the control input, the circuit will perform different arithmetic or logical operations on the data inputs. Observe that the four operations diagrammed for this ALU correspond to the four different integer operations supported by the Y86 instruction set, and the control values match the function codes for these instructions (Figure 4.3). Note also the ordering of operands for subtraction, where the A input is subtracted from the B input. This ordering is chosen in anticipation of the ordering of arguments in the subl instruction.

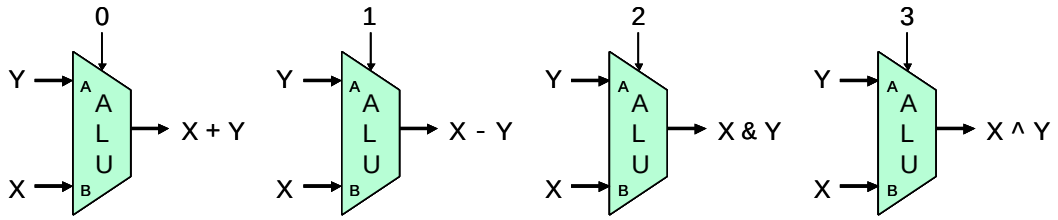
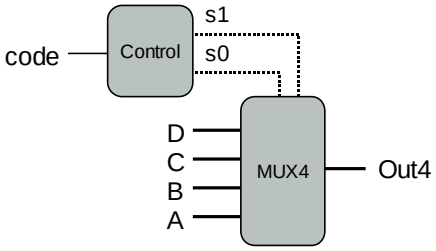


Figure 4.13: **Arithmetic/logic unit (ALU).** Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

4.2.4 Set Membership

In our processor designs, we will find many examples where we want to compare one signal against a number of possible matching signals, such as to test whether the code for some instruction being processed matches some category of instruction codes. As a simple example, suppose we want to generate the signals s1 and s0 for the four-way multiplexor of Figure 4.12 by selecting the high- and low-order bits from a two-bit signal code, as follows:



In this circuit, the two-bit signal code would then control the selection among the four data words A, B, C, and D. We can express the generation of signals s1 and s0 using equality tests based on the possible values of code:

```
bool s1 = code == 2 || code == 3;

bool s0 = code == 1 || code == 3;
```

A more concise expression can be written that expresses the property that s1 is 1 when code is in the set $\{2, 3\}$, and s0 is 1 when code is in the set $\{1, 3\}$:

```
bool s1 = code in { 2, 3 };

bool s0 = code in { 1, 3 };
```

The general form of a set membership test is

$$iexpr \quad \texttt{in} \quad \{iexpr_1, iexpr_2, \ldots, iexpr_k\}$$

where both the value being tested, $expr$, and the candidate matches, $iexpr_1$ through $iexpr_k$, are all integer expressions.

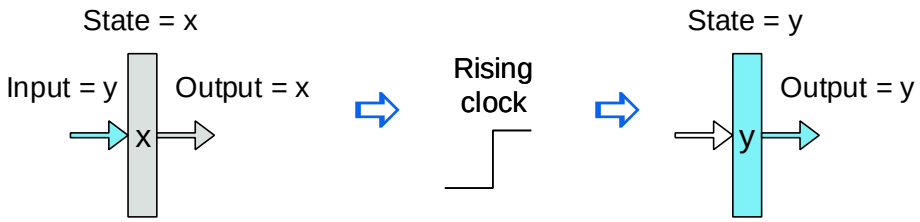


Figure 4.14: **Register operation.** The register outputs remain held at the current register state until the clock signal rises. When the clock rises, the values at the register inputs are captured to become the new register state.

4.2.5 Memory and Clocking

Combinational circuits, by their very nature, do not store any information. Instead, they simply react to the signals at their inputs, generating outputs equal to some function of the inputs. To create *sequential circuits*, that is, systems that have state and perform computations on that state, we must introduce devices that store information represented as bits. We consider two classes of memory devices:

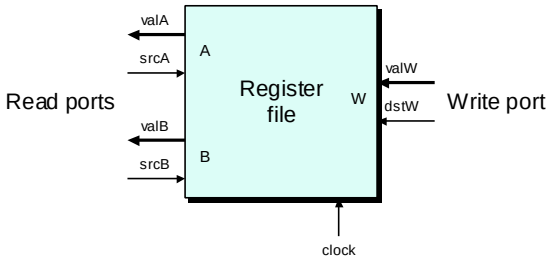
Clocked registers (or simply *registers*) store individual bits or words. A clock signal controls the loading of the register with the value at its input.

Random-access memories (or simply *memories*) store multiple words, using an address to select which word should be read or written. Examples of random-access memories include (1) the virtual memory system of a processor, where a combination of hardware and operating system software make it appear to a processor that it can access any word within a large address space; and (2) the register file, where register identifiers serve as the addresses. In an IA32 or Y86 processor, the register file holds the eight program registers (`%eax`, `%ecx`, etc.).

As we can see, the word "register" means two slightly different things when speaking of hardware versus machine-language programming. In hardware, a register is directly connected to the rest of the circuit by its input and output wires. In machine-level programming, the registers represent a small collection of addressable words in the CPU, where the addresses consist of register IDs. These words are generally stored in the register file, although we will see that the hardware can sometimes pass a word directly from one instruction to another to avoid the delay of first writing and then reading the register file. When necessary to avoid ambiguity, we will call the two classes of registers "hardware registers" and "program registers," respectively.

Figure 4.14 gives a more detailed view of a hardware register and how it operates. For most of the time, the register remains in a fixed state (shown as x), generating an output equal to its current state. Signals propagate through the combinational logic preceding the register, creating a new value for the register input (shown as y), but the register output remains fixed as long as the clock is low. As the clock rises, the input signals are loaded into the register as its next state (y), and this becomes the new register output until the next rising clock edge. A key point is that the registers serve as barriers between the combinational logic in different parts of the circuit. Values only propagate from a register input to its output once every clock cycle at the rising clock edge.

The following diagram shows a typical register file:



This register file has two *read ports*, named A and B, and one *write port*, named W. Such a *multiported* random-access memory allows multiple read and write operations to take place simultaneously. In the register file diagrammed, the circuit can read the values of two program registers and update the state of a third. Each port has an address input, indicating which program register should be selected, and a data output or input giving a value for that program register. The addresses are register identifiers, using the encoding shown in Figure 4.4. The two read ports have address inputs srcA and srcB (short for "source A" and "source B") and data outputs valA and valB (short for "value A" and "value B"). The write port has address input dstW (short for "destination W"), and data input valW (short for "value W").

Although the register file is not a combinational circuit (since it has internal storage), reading words from it operates in the same manner as a block of combinational logic having the addresses as inputs and the data as outputs. When either srcA or srcB is set to some register ID, then after some delay, the value stored in the corresponding program register will appear on either valA or valB. For example, setting srcA to 3 will cause the value of program register %ebx to be read, and this value will appear on output valA.

The writing of words to the register file is controlled by a clock signal in a manner similar to the loading of values into a clocked register. Every time the clock rises, the value on input valW is written to the program register indicated by the register ID on input dstW. When dstW is set to the special ID value 8, no program register is written.

4.3 Sequential Y86 Implementations

Now we have the components required to implement a Y86 processor. As a first step, we describe a processor called SEQ (for "sequential" processor). On each clock cycle, SEQ performs all the steps required to process a complete instruction. This would require a very long cycle time, however, and so the clock rate would be unacceptably low. Our purpose in developing SEQ is to provide a first step toward our ultimate goal of implementing an efficient, pipelined processor.

4.3.1 Organizing Processing into Stages

In general, processing an instruction involves a number of operations. We organize them in a particular sequence of stages, attempting to make all instructions follow a uniform sequence, even though the instructions differ greatly in their actions. The detailed processing at each step depends on the particular instruction being executed. Creating this framework will allow us to design a processor that makes best use