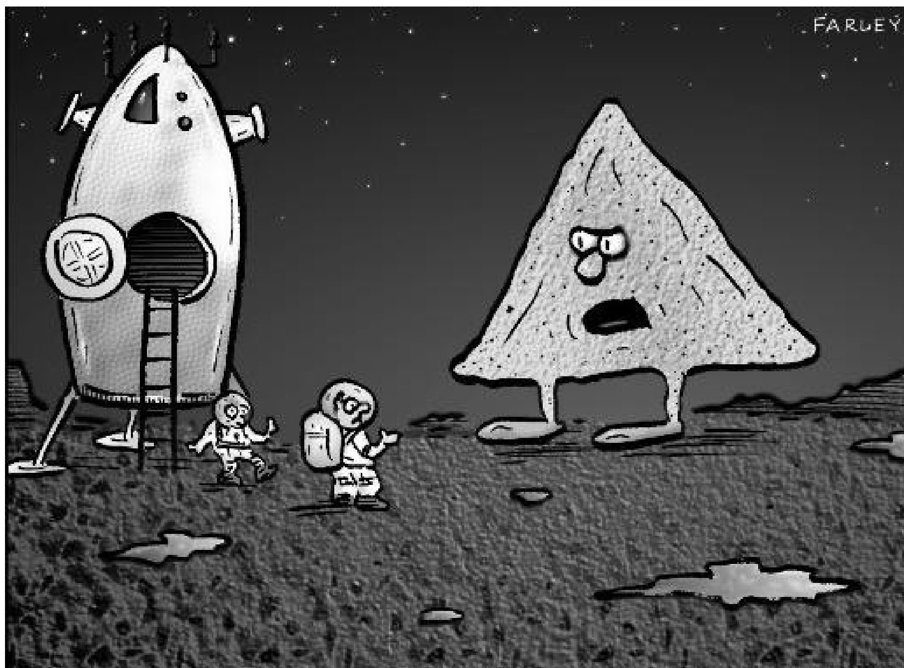
Travaux pratiques de système de quatrième année
Utilisation du système Nachos

Année scolaire 2002-2003



© Copyright 1994, David Farley, voir http://www.cs.washington.edu/homes/tom/nachos/

Isabelle Puaut
Ivan Leplumey

Chapitre 1

Introduction

Historique

NACHOS est un système d'exploitation à vocation pédagogique conçu à l'origine à l'université de Berkeley [1]. Son objectif est de permettre à des étudiants d'appréhender le fonctionnement interne d'un système d'exploitation. Pour ce faire, il dispose des mécanismes utilisés dans les systèmes commerciaux, tout en adoptant une conception d'une approche plus aisée.

Ce logiciel a été modifié en 1999-2000 lors d'un projet de développement de la quatrième année option informatique (équivalent maîtrise) de l'INSA de Rennes, par un groupe de 8 étudiants. Ce groupe était constitué de Matthieu Gabriac, Julien Gloaguen, Jérôme Le Dorze, Aurélien Letort, Antoine Mahé, Freddy Perraud, Anthony Remazeilles et Chloé Rispal, et a été encadré par Isabelle Puaut. Le résultat de ce travail est un système NACHOS adapté aux objectifs pédagogiques de l'enseignement de système en option informatique à l'INSA. Nous utilisons dans tout le document le terme NACHOS pour désigner les modifications au NACHOS d'origine effectuées à l'INSA de Rennes. Une liste exhaustive des modifications apportées à NACHOS est donnée dans le fichier README accompagnant les sources.

Objet et contenu du document

Ce document a pour but de faciliter la compréhension de NACHOS et se propose de rentrer progressivement dans les détails de sa mise en œuvre. Il est important de noter que son contenu décrit le résultat final attendu suite à la réalisation des travaux pratiques. Certaines parties détaillées ici ne sont pas dans les fichiers sources qui vous seront fournis, et seront à réaliser pendant les travaux pratiques.

Le document est organisé comme suit. La structure interne de NACHOS est présentée dans le chapitre 2. Les sujets de TP sont donnés dans le chapitre 3.

1. Toutes les informations relatives à NACHOS pourront être trouvées sur la home page NACHOS à l'URL *http://www.cs.berkeley.edu/˜tea/nachos*

Bugs, suggestions d'améliorations

Si vous trouvez un bug dans le code source de NACHOS ou dans sa documentation, ou que vous avez envie de proposer des améliorations ou extensions au code source, merci de le signaler (e-mail: puaut@irisa.fr / leplumey@insa-rennes.fr).

Chapitre 2

Une introduction à Nachos

Nous présentons dans le paragraphe 2.1 l'organisation générale du système, chacun de ses modules étant détaillé dans la suite. Le paragraphe 2.2 est consacré aux composants matériels, alors que le paragraphe 2.3 présente les pilotes de périphériques. Le paragraphe 2.4 explique le fonctionnement du cœur du noyau (gestion des tâches légères – threads – et de leur synchronisation). Les paragraphes suivants portent successivement sur le fonctionnement des pilotes de périphériques (paragraphe 2.3) et sur le système de gestion des fichiers (paragraphe 2.6). Les mécanismes de gestion de la mémoire virtuelle sont détaillés dans le paragraphe 2.7, le dernier paragraphe expliquant les étapes de réalisation et d'exécution d'un programme utilisateur avec NACHOS.

2.1 Présentation de la structure du système

Dans cette section, nous présentons les grandes composantes qui constituent le système NACHOS et ceci sans rentrer dans les détails de leur mise en œuvre.

2.1.1 Nachos est un processus UNIX

L'objectif recherché lors de la construction de NACHOS était de permettre à des étudiants de se familiariser à investissement modéré avec les concepts de base des systèmes d'exploitation. Pour ce faire, NACHOS ne s'exécute pas directement au dessus du matériel. À la place, il utilise un matériel (processeur, périphériques) *émulé*, l'ensemble formé par le matériel émulé, le noyau NACHOS et les applications s'exécutant au sein d'un même processus UNIX (cf figure 2.1).

L'intérêt d'une telle approche est double. D'une part, le développement du noyau est simplifié, car la matériel émulé est volontairement très simple. D'autre part, sa mise au point est simplifiée, puisque une erreur de programmation ne cause pas l'arrêt brutal de la machine comme dans un système d'exploitation réel, et que les outils de mise au point classiques (par exemple *gdb*) peuvent être utilisés.

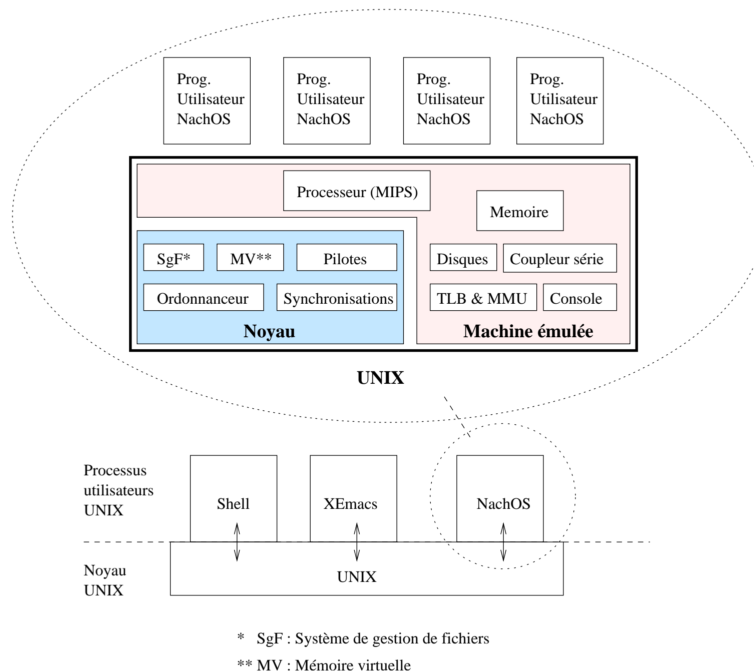


FIG. 2.1 – *Structure interne de* NACHOS

NACHOS est développé en $C++$, en utilisant uniquement les caractéristiques de base du langage (encapsulation).

2.1.2 Émulation du matériel

Une autre caractéristique essentielle de NACHOS concerne l'utilisation de matériel *émulé par logiciel* à la place du matériel réel. Ainsi, dans NACHOS, le système ne se sert pas directement du matériel de la machine hôte (Sparc/Solaris) mais utilise des émulations de composants matériels par logiciel (matérialisés par des objets $C++$). Ces derniers sont plus simples car leurs fonctionnalités, bien que réalistes, répondent seulement à nos besoins.

Les principaux éléments matériels émulés sont un processeur MIPS sur lequel vont s'exécuter les programmes utilisateurs, un sablier (timer), des disques, un coupleur série, une console, une unité de gestion de la mémoire (MMU - Memory Management Unit) et un cache de traduction d'adresse (TLB - Translation Look-Aside Buffer) associé à la MMU.

2.1.3 Modules principaux du noyau

La structure interne de NACHOS met en évidence un découpage en modules, qui apparaît dans la structure des répertoires utilisée lors du développement. Les principaux modules sont :

- le *noyau* qui comprend les fonctionnalités vitales du système : taches légères (threads), primitives de synchronisation entre threads, ordonnancement,

- les *pilotes* des périphériques émulés (disque, console)
- la gestion de la *mémoire virtuelle*,
- la *gestion de fichiers.*

NACHOS est écrit en C++ et les différentes classes du noyau sont répartis selon la structure de répertoires suivante :

- *kernel* : classes du noyau,
- *machine* : classes d'émulation du matériel,
- *vm* : classes de gestion de la mémoire virtuelle,
- *handler* : classes des pilotes de périphériques,
- *filesys* : classes du système de gestion des fichiers,
- *utility* : classes utilitaires (listes, etc.),
- *test* : programmes utilisateurs,
- *bin* : programmes et scripts de conversions de formats de fichiers exécutables.

Noyau de Nachos (répertoire *kernel*)

Le noyau de NACHOS gère un ensemble de processus légers (threads), partageant le même espace d'adressage et s'exécutant de manière quasi-parallèle. Nous nommons *processus* l'ensemble formé d'un espace d'adressage et des threads se le partageant.

L'ordonnancement des threads est réalisé selon une politique de type FIFO (First In, First Out), et ne gère aucune priorité. La synchronisation au sein du noyau se fait à l'aide de sémaphores, de variables de condition et de verrous. Il n'y a pas à la base de partage de temps (time-slicing).

Lorsqu'un thread est bloqué sur une primitive de synchronisation, un changement de contexte se produit et l'unité centrale est allouée au premier thread présent dans la file des prêts gérée par l'ordonnançeur. Si aucun thread n'est prêt, le système patiente jusqu'à l'arrivée d'une interruption.

Pilotes de périphériques (répertoire *handler*)

Les pilotes de périphériques constituent la couche logicielle qui permet de faire fonctionner les composants matériels émulés. Ils permettent de passer des opérations asynchrones proposées par les émulations (et par le matériel normalement) à des opérations synchrones. Ceci se fait par l'utilisation des interruptions et des primitives de synchronisation (sémaphores et verrous). Les pilotes proposent des méthodes bloquantes qui rendent la main une fois que l'opération est complètement effectuée.

Système de gestion des fichiers (répertoire *filesys*)

NACHOS dispose d'un système de fichiers. Les fonctionnalités proposées sont :

- la création et suppression de répertoires,
- la création et suppression de fichiers,
- la lecture et l'écriture dans les fichiers

Il est possible de copier un programme utilisateur, qui existe sous la forme d'un fichier UNIX, dans le système de fichiers de NACHOS. Ce programme pourra alors être chargé puis exécuté.

Gestion de la mémoire virtuelle (répertoire *vm*)

NACHOS dispose d'une gestion complète de la mémoire virtuelle : on utilise la traduction d'adresses d'une part et le va-et-vient de pages entre la mémoire et le disque d'autre part.

La traduction d'adresses consiste à transformer à l'exécution, avec l'aide d'un support matériel (MMU et TLB) les adresses *virtuelles* manipulées par les programmes utilisateurs en *adresses réelles* dans la mémoire de la machine. La traduction d'adresses utilise des tables stockées en mémoire. L'utilisation d'une zone de *swap* (ou zone d'échange) sur disque permet, lorsque la mémoire est entièrement utilisée, de libérer des pages physiques en plaçant leur contenu sur disque. Le défaut de page, mécanisme inverse, intervient pour charger une page présente dans la zone d'échange en mémoire réelle.

2.1.4 Programmes utilisateur (répertoire *test*)

Les programmes utilisateur sont écrits en C. Des appels système, permettant de communiquer avec le noyau, ainsi que des routines diverses, regroupées dans une bibliothèque liée aux programmes utilisateurs, peuvent être utilisés. On ne dispose pas cependant de toutes les fonctionnalités des librairies C classiques et encore moins de tous les appels système disponibles sous UNIX (NACHOS reste un mini-système).

La conception du système, avec l'émulation d'une machine MIPS sur laquelle vont être exécutés les programmes utilisateur, impose à ces derniers d'être compilés en code MIPS. Ceci est réalisé par un compileur croisé (cross-compilateur), en l'occurrence ici gcc, qui va générer du code MIPS exécutable sous NACHOS.

On peut répartir les appels système en plusieurs groupes suivant leurs fonctions :

- les processus et threads,
- les fichiers,
- la synchronisation entre threads,
- la communication série,
- la console.

2.1.5 Paramétrage du système

Fichier de configuration (répertoire *kernel*, fichier nachos.cfg)

Il est possible de modifier certaines paramètres de NACHOS entre deux exécutions sans avoir besoin de générer à nouveau l'exécutable. Ceci est réalisé à l'aide d'un fichier de configuration qui comporte la valeur de certains paramètres tels que le nombre de pages de mémoire de la machine émulée, le nom du programme à lancer au démarrage du système, le formattage du disque au démarrage, etc. Le format du fichier de configuration est détaillé dans le paragraphe 2.9.6.

Statistiques

Un module de statistiques permet de visualiser l'effet de certains paramètres du système (par exemple, cache de traduction d'adresses) sur ses performances. Les données sont collectées au cours du fonctionnement de NACHOS et portent sur les temps *virtuels* d'exécution des processus, sur les accès mémoires (défauts de TLB, défauts de page, nombre d'instructions exécutées), sur les accès au disque et à la console. Toutes les statistiques sont données processus par processus, les statistiques des différents threads du processus étant cumulées.

2.2 La machine émulée (répertoire *machine*)

2.2.1 Le processeur MIPS (fichiers *machine.cc, machine.h*)

Le processeur émulé par NACHOS correspond à une architecture MIPS. Le processeur dispose en interne de registres entiers, de registres flottants simple précision, et d'un code code condition pour les opérations flottantes. L'objet *machine* de la classe *Machine* exporte plusieurs méthodes, permettant de lancer le processeur et d'inspecter son état (en particulier lecture et écriture des registres entiers et flottants):

- **void Run()**, qui exécute le programme utilisateur chargé en mémoire ;
- **int ReadRegister(int num)**, qui renvoie le contenu du registre entier *num*;
- **void WriteRegister(int num, int value)**, qui écrit la valeur *value* dans le registre entier *num*.
- **int ReadFPRegister(int num)**, qui renvoie la valeur du registre flottant *num*.
- **void WriteFPRegister(int num, int value)**, qui écrit la valeur *value* dans le registre flottant *num*.
- **char ReadCC(void)**, qui renvoie le contenu du code condition flottant.
- **void WriteCC(char val)**, qui initialise le contenu du code condition flottant avec la valeur *val*.

En outre, les constantes suivantes contiennent les codes correspondants aux registres du processeur :

- *StackReg* : pointeur de pile
- *RetAddrReg* : adresse de retour de procédure
- *PCReg* : compteur ordinal
- *NextPCReg* : pointeur vers la prochaine instruction
- *BadVAddrReg* : adresse virtuelle ayant causé une exception
- *NumTotalRegs* : nombre de registres entiers du processeur
- *NumFPRegs* : nombre de registres flottants du processeur
- *Table* : pointeur vers la table des pages
- *NumPage* : nombre de pages de cette table

Certains registres entiers ont un rôle particulier lors des appels système pour le passage de paramètres (voir paragraphe 2.5).

2.2.2 Le contrôleur d'interruptions (fichiers *interrupt.cc, interrupt.h*)

Le contrôleur d'interruptions a pour rôle de définir si on doit ou non prendre en compte les interruptions qui sont générées par le matériel. L'objet *interrupt* de la classe *Interrupt* correspond à ce contrôleur, et il exporte trois méthodes :

- **IntStatus SetLevel(IntStatus level)**, permet d'autoriser ou d'interdire les interruptions, et rend en résultat l'état précédent vis à vis des interruprions,
- **void Enable()**, autorise les interruptions,
- **IntStatus getLevel()**, renvoie l'état courant vis à vis des interruptions.

Le type *IntStatus* est un type énuméré définissant la validité des interruptions et qui contient les valeurs :

- *IntOff* : les interruptions sont interdites,
- *IntOn* : les interruptions sont autorisées.

2.2.3 Le coupleur série (fichiers *ACIA.h, ACIA.cc*)

NACHOS possède une émulation de coupleur série qui permet d'envoyer ou recevoir des caractères via une liaison série. Le coupleur série est accessible au moyen de l'objet *interface*, instance de la classe *ACIA*. Comme son nom l'indique, cet objet constitue l'interface du coupleur série.

Les registres de données

Le registre *outputRegister* (resp. *inputRegister*) constitue le registre de données en émission (resp. réception) et contient le caractère à envoyer (resp. recevoir). Les registres de données sont accessibles via les méthodes suivantes :

- **void PutChar(char)** place le caractère passé en paramètre, dans le registre *outputRegister*.
- **char GetChar()** retourne le caractère contenu dans le registre *inputRegister* s'il est plein, 0 sinon.

Les registres de données sont initialisés à 0. Le coupleur série émet une interruption en réception lorsque le registre *inputRegister* devient plein et une interruption en émission lorsque le registre *outputRegister* devient vide.

Les registres d'état

Les registres *ouputStateRegister* et *inputStateRegister* indiquent l'état des registres de données. Ils sont accessibles en lecture uniquement, à travers les méthodes suivantes :

- **RegStatus GetOutputStateReg()** qui retourne le contenu du registre *outputStateRegister*, donc l'état du registre de données en émission *ouputRegister*.
- **RegStatus GetInputStateReg()** qui retourne le contenu du registre *inputStateRegister*, donc l'état du registre de données en réception *inputRegister*.

Le registre de contrôle

Le registre de contrôle *mode* détermine le mode de fonctionnement du coupleur. Ce dernier peut fonctionner par test d'état/attente active ou par interruptions. **void SetWorkingMode(int mod)**, méthode de la classe ACIA, permet de modifier le registre de contrôle *mode*, la méthode GetWorkingMode permet de le lire. Le paramètre *mod* peut prendre les trois valeurs suivantes :

- *BUSY_WAITING* : pour fonctionner en attente active;
- *EM_INTERRUPT* : pour autoriser les interruptions en émission;
- *REC_INTERRUPT* : pour autoriser les interruptions en réception.

Un masque peut être réalisé à l'aide de ces différentes constantes. Par exemple, pour autoriser simultanément les interruptions en émission et en réception, un ou logique est effectué entre *EM_INTERRUPT* et *REC_INTERRUPT*.

2.2.4 Le disque (fichiers *disk.cc*, *disk.h*)

Le disque émulé contient *NUM_TRACKS* pistes, chacune étant composé de *SECTORS_PER_TRACKS* secteurs. Chaque secteur a une taille de *SECTOR_SIZE* octets. Deux méthodes permettent d'accéder au contenu du disque :

- **void ReadRequest(int sectorNumber, char* data)** lit un secteur sur le disque,
- **void WriteRequest(int sectorNumber, char* data)** écrit un secteur sur le disque.

Ces deux méthodes retournent immédiatement, sans attendre la fin de l'entrée/sortie disque. La simulation du temps pris par la lecture ou écriture sur le disque, et la synchronisation pour attendre la fin du transfert, sont gérés par le pilote du disque (voir 2.3.3, 14).

2.2.5 La console (fichiers *console.cc, console.h*)

La machine émulée contient une console, qui permet d'afficher des caractères à l'écran et de recevoir des caractères par le clavier. La console est gérée par son pilote (voir 2.3.2, page 14). La console émulée permet :

- l'affichage d'un caractère par la méthode **void PutChar(char ch)**;
- la lecture d'un caractère au clavier par la méthode **char GetChar()**.

Lors d'un affichage, ou d'une écriture, de manière identique au disque, la fin de l'opération est spécifiée par une interruption demandée par la console.

2.2.6 Le TLB (Translation Look-aside Buffer) (fichiers *tlb.cc, tlb.h*)

Le TLB est un cache stockant en permanence les dernières correspondances pages virtuelles - pages réelles. Il permet une traduction d'adresse plus rapide qu'une recherche via la MMU (voir 2.2.7) et, dans le cas où la page virtuelle ne se trouve pas dans le TLB, la traduction d'adresse est effectuée par la MMU. Dans NACHOS, l'objet *tlb* de la classe *TLB* émule ce cache de traduction.

Le TLB est une structure participant à la gestion de la mémoire virtuelle. La compréhension de cet élément n'est pas nécessaire pour effectuer les premiers travaux pratiques. Une description plus complète est donnée dans le paragraphe 2.7.1, page 24.

2.2.7 La MMU (Memory Management Unit, fichiers *mmu.cc, mmu.h*)

La MMU émulée dans NACHOS a pour but de simuler une traduction d'adresse. L'objet *mmu* de la classe *MMU* correspond à ce composant, et son unique méthode **ExceptionType Translate(int virtualPage,int* physicalPage,bool* writing)** permet de récupérer la page physique associée à une page virtuelle. Sommairement, il existe deux cas :

- la page recherchée n'est pas en mémoire physique, et la MMU déclenche un défaut de page (routine s'occupant d'aller charger la page désirée en mémoire physique)

- la page est en mémoire physique, et la MMU retourne le numéro de la page physique correspondante.

De même que le TLB, la MMU participe à la gestion de la mémoire virtuelle, et peut être oubliée lors des premiers travaux pratiques. Son rôle est détaillé plus amplement dans la section 2.7.1, page 26.

2.3 Les pilotes de périphériques (répertoire *handler*)

Les pilotes (handlers) de périphériques permettent de gérer les périphériques émulés par NACHOS (coupleur série, console, disque). Ils représentent les seuls moyens d'accès aux périphériques qui leur sont associés. De plus, les handlers assurent la synchronisation des entrées/sorties et gèrent le partage des périphériques en cadre multi-thread.

2.3.1 Le pilote du coupleur série (fichiers *ACIA_handler.cc, ACIA_handler.h*)

Ce pilote assure la synchronisation des envois et des réceptions de messages. Un objet *Handler* issu de la classe *ACIA_handler* est instancié lors du démarrage du système.

Description de la classe *ACIA_handler*

Les membres de cette classe sont :

- *send_buffer* et *receive_buffer* qui constituent respectivement les tampons en émission et en réception,
- *emission_finished* et *reception_finished* qui sont les sémaphores utilisés pour la synchronisation;
- *ind_send* et *ind_rec* qui sont les indices de remplissage du tampon en réception et en émission.

Deux méthodes sont exportées :

- **void tty_send(char* buff)**, qui assure l'envoi des chaînes de caractères passées en paramètre, via le tampon en émission. Cette primitive est *non bloquante.*
- **void tty_receive(char* buff,int lg)**, qui assure la réception de chaines de caractères via le tampon en réception et renvoie la chaîne lue. Cette primitive est *bloquante.*

Ces deux méthodes seront appelées lors des appels système *TtySend* et *TtyReceive.* Les méthodes *tty_send* et *tty_receive* peuvent être implémentées pour fonctionner de manière synchrone (avec attente active) ou de manière asynchrone (avec interruptions). Dans le second

cas, les deux méthodes suivantes constituant les routines de traitement d'interruptions seront utilisées :

- **void interrupt_send()** est la routine de traitement d'interruption en émission responsable de l'envoi d'un caractère. Elle est systématiquement appelée lorsque le registre de données en émission devient vide et les interruptions en émission sont autorisées.
- **void interrupt_receive()** est la routine de traitement d'interruption en réception responsable de la réception d'un caractère. Elle est systématiquement appelée lorsque le registre de données en réception devient plein et les interruptions en réception sont autorisées.

Les routines de traitement d'interruptions détectent également la fin de l'envoi ou d'une réception de message.

2.3.2 Le pilote de la console (fichiers *synchcons.cc, synchcons.h*)

Il existe un pilote chargé de la synchronisation des routines d'écriture et de lecture via la console. L'objet *synchcons* issu de la classe *SynchConsole* assure ces opérations dans la console grâce à deux méthodes :

- **void PutString(char *buffer,int size)**
- **void GetString(char *buffer,int size)**

La console est un matériel asynchrone (une requête retourne immédiatement) ; le pilote rend l'utilisation de la console synchrone, en attendant la fin de l'opération désirée (signalée par une interruption). De plus, le pilote doit contrôler qu'il n'y a qu'une opération d'écriture, et une opération de lecture à la fois. Les méthodes du pilote de console sont appelées lors des appels systèmes *Write(chaine*,longueur,ConsoleOutPut)* et *Read(chaine*,longueur,ConsoleInPut)* qui permettent respectivement d'écrire et de lire une chaîne de caractères sur la console. Il existe également dans la bibliothèque de NACHOS des routines (*printf,writestr,...*) permettant d'écrire vers la console.

2.3.3 Le pilote du disque (fichiers *synchdisk.cc, synchdisk.h*)

Cette classe permet de synchroniser les accès au disque. Elle possède 3 attributs : un pointeur sur le disque dont les accès doivent être synchronisés, un sémaphore pour attendre la fin des entrées/sorties, un verrou pour assurer qu'une seule requête de lecture ou d'écriture ne peut être effectuée à la fois (le verrou est pris - respectivement relâché - au début - resp. à la fin - des deux méthodes d'accès au disque).

- **void ReadSector(int sectorNumber, char* data)**, lecture synchrone d'un secteur du disque,

- **void WriteSector(int sectorNumber, char* data)**, écriture synchrone d'un secteur sur le disque,
- **void RequestDone()**, permet de libérer un thread en attente de la fin de son accès disque.

2.4 Le noyau de Nachos (répertoire *kernel*)

2.4.1 Fonctionnement interne du noyau

NACHOS est un système d'exploitation pour des processus ayant des espaces d'adressage séparés, chaque processus étant parallèle (multi-thread). Trois objets de NACHOS sont à la base de son fonctionnement :

- **Scheduler** qui gère l'ordonnancement des threads.
- **Thread** qui contient les données nécessaires à la gestion des threads.
- **Addrspace** qui mémorise les données relatives aux différents espaces d'adressage.

Objet Scheduler (fichiers *scheduler.cc, scheduler.h*)

Une instance de la classe **Scheduler** est utilisée dans NACHOS. Elle sert à gérer la file des prêts (*readyList* et à effectuer le changement de contexte entre les threads. Le thread élu qui possède l'unité centrale, ne fait pas partie de la file des prêts, mais est référencé par le pointeur global **currentThread**.

La classe Scheduler exporte trois méthodes :

- **ReadyToRun(Thread *thread)** qui ajoute un thread en queue de liste des prêts.
- **FindNextToRun(void)** qui renvoie et enlève le premier thread de la liste des prêts.
- **Run(Thread *nextThread)** alloue le processeur au thread passé en paramètre (en général le résultat de **FindNextToRun**).

Objet Thread (fichiers *thread.cc, thread.h*)

L'objet **Thread** encapsule un thread. Il contient le contexte du thread auquel il est associé. Du fait de l'utilisation d'un processeur MIPS émulé et du fait que le noyau s'exécute directement sur la machine hôte, le contexte d'un thread n'est pas limité au contexte de la machine MIPS émulée (registres de la machine MIPS). À la place, son contexte est séparé en deux composantes :

- Le *contexte utilisateur*, constitué de l'état des registres de la machine MIPS : *userRegisters*
- *Le* contexte noyau*, constitué de l'état de la machine hôte (SPARC), représenté par les variables d'état* machineState et le pointeur de pile *kernelStackTop*.

Les principales méthodes de la classe Thread sont :

- **Join(int Idthread)**, qui bloque le thread appelant jusqu'à la terminaison du thread *Idthread*.
- **Yield(void)**, qui met le thread appelant en queue de file des prêts.
- **Sleep(void)**, qui endort le thread appelant jusqu'à ce qu'on le réveille explicitement.
- **Finish(void)**, qui termine le thread appelant et planifie sa destruction.
- **SaveUserState(void)**, qui sauvegarde l'état des registres utilisateurs (MIPS).
- **RestoreUserState(void)**, qui restaure l'état des registres utilisateurs (MIPS).
- **initKernelContext(int arg)**, qui crée le contexte noyau de l'objet **Thread** (allocation et initialisation). Cette méthode est privée à la classe Thread.
- **void* newThread(char *name,VoidFunctionPtr func, int arg);**, qui crée un nouveau thread dans l'espace d'adressage courant, alloue et initialise son contexte noyau et utilisateur et l'insère dans la file des prêts.
- **Exec(char* filename,Thread *masterThread)**, qui prend en argument un objet Thread créé au préalable et se charge d'initialiser un nouvel espace d'adressage pour son exécution. L'espace d'adressage est initialisé à partir du fichier exécutable de nom *filename*. La méthode *Exec* effectue les mêmes actions que *newThread* à la création d'espace d'adressage près.

Objet Addrspace (fichiers *addrspace.cc, addrspace.h*)

À chaque objet *Thread* est associé un objet *Addrspace* (exception faite du premier thread créé au lancement du système). Tous les threads se partageant le même espace d'adressage pointent sur le même objet *Addrspace*. Le constructeur de l'objet *Addrspace* charge l'exécutable en mémoire et crée un espace mémoire associé. Ces notions seront vues au cours du second semestre et détaillées dans les TPs associés.

Changement de contexte

Le changement de contexte entre deux threads (que les threads appartiennent ou non au même espace d'adressage) est effectué dans la procédure *Run* du *Scheduler*. L'objectif du changement de contexte est de sauvegarder le contexte du thread appelant dans l'objet thread correspondant, et de restaurer celui du nouveau thread élu.

Dans un système d'exploitation s'exécutant sur machine nue, les applications et le système d'exploitation s'exécutent sur le même type d'architecture. De ce fait, seul le contexte relatif à cette architecture (registres du processeur essentiellement) doivent être sauvegardés. Ici, le contexte est un peu différent, car les applications s'exécutent sur un processeur émulé (MIPS) alors que le noyau s'exécute directement sur la machine hôte (SPARC). De ce fait, deux contextes doivent être gérés lors d'un changement de contexte : le contexte utilisateur

(MIPS) et le contexte noyau (SPARC). La sauvegarde/restauration du contexte noyau vous est directement fournie (fonction bas-niveau *SWITCH*). Vous n'aurez pas à gérer le contexte noyau lors des travaux pratiques.

Deux cas peuvent se présenter au retour de la fonction *SWITCH*:

- C'est la première fois que le thread élu prend la main. D'après le fonctionnement de la fonction *initKernelContext*, c'est la fonction *StartThreadExecution* qui est exécutée. Celle-ci doit donc initialiser le contexte utilisateur avant de lancer la machine MIPS.
- L'élu avait déjà eu la main au préalable. Il se retrouve donc dans la fonction *Run* du *Scheduler*, après l'appel à *SWITCH*. La procédure *Run* doit alors restaurer le contexte utilisateur.

2.4.2 Outils de synchronisation (fichiers *synch.cc, synch.h*)

Trois types de synchronisation sont définis dans NACHOS : les sémaphores, les verrous et les variables de condition. Toutes les méthodes relatives aux outils de synchronisation sont atomiques. Comme on est sur un système monoprocesseur, l'atomicité est mise en œuvre simplement en interdisant les interruptions.

Sémaphore

Contient un entier et une file d'attente.

- **void P()** Fait attendre le thread appelant jusqu'à ce que la valeur du sémaphore soit positive, puis décrémente cette dernière.
- **void V()** Incrémente la valeur du sémaphore, ce qui peut entraîner la libération d'un thread bloqué sur ce sémaphore s'il en existe un.

Verrou

Ce mécanisme d'exclusion mutuelle possède un booléen et une file d'attente.

- **void Acquire()** Le thread s'approprie le verrou s'il est libre sinon il se bloque dans la file.
- **void Release()** Cette méthode est exécutable seulement par le détenteur du verrou pour relâcher le verrou. Elle met son état à libre puis regarde dans la file du verrou si un éventuel thread attend l'accès. Dans ce cas, elle retire ce thread de la file du verrou et le remet dans la file des prêts.

Variable de condition

Nécessite un verrou qui lui est propre. Toutes les opérations sur cette variable de condition doivent être effectuées lorsque le thread courant a acquis le verrou. Chaque variable de

condition possède sa propre file d'attente ce qui permet de mettre des threads en attente de la condition pour accéder au verrou.

- **void Wait(Lock *conditionLock)** Le thread relâche le verrou et se place dans l'ensemble d'attente de la condition. Il s'endort. Il doit redemander l'accès au verrou à son réveil.
- **void Signal(Lock *conditionLock)** Le thread possesseur du verrou réveille un des threads de l'ensemble d'attente de la condition (le met dans la file des prêts) et lui donne ainsi la possibilité de redemander l'accès au verrou.
- **void Broadcast(Lock *conditionLock)** Même effet que signal mais sur tous les threads de l'ensemble d'attente.

2.5 Fonctionnement des appels systèmes

2.5.1 Fonctionnement global

Les programmes utilisateur ont accès à une liste d'appels système qui sont caractérisés dans le code MIPS généré par l'instruction *syscall*, dont le rôle est de générer une exception. Cette liste d'appels est décrite dans le fichier *syscall.h*, et le code MIPS chargé de spécifier le numéro de l'appel et de créer l'exception système se trouve dans le fichier *start.s* (ce fichier assembleur, une fois compilé, est lié avec les programmes utilisateur). Le type d'appel système est spécifié dans le registre *r2* de la machine MIPS, et les paramètres de l'appel sont stockés automatiquement lors de la compilation dans les registres *r4*, *r5*, *r6* et *r7* selon leur nombre. La valeur de retour de l'appel, si il y en a une, est placée dans le registre *r2*.

L'exception générée par l'instruction *syscall* est récupérée au niveau d'une routine de traitement d'exceptions située dans le fichier *exception.cc*, qui va alors rediriger l'appel, en fonction de son type, vers les pilotes de périphériques ou vers le noyau du système.

2.5.2 Exemple de déroulement d'un appel système

Nous détaillons ici le cheminement d'un appel système dans NACHOS. Prenons pour exemple un programme utilisateur dont l'effet est d'écrire le chiffre 5 à l'écran :

```
int main() {
     WriteInt(5);
}
```

Lors de la compilation de ce code source, *WriteInt(5)* est traduite en un appel au code assembleur portant l'étiquette *WriteInt* dans le fichier *start.s* :

```
        .globl WriteInt
        .ent WriteInt
WriteInt:
```

```
addiu $2,$0,SC_WriteInt
syscall
j     $31
.end WriteInt
```

Le code généré pour l'appel à *WriteInt* a placé la valeur 5 dans le registre *r4*.

Lors du décodage des instructions de l'application par la machine MIPS, l'instruction *syscall* provoque l'appel de la routine de traitement d'exceptions. Le code de l'exception, ici *SC_WriteInt*, permet à cette routine de savoir quelle est l'action à réaliser. Dans notre cas, nous allons demander au pilote de console d'écrire la chaîne correspondant à 5 (voir code ci-dessous).

```
// lecture du nombre à afficher depuis le registre 4
num = machine->ReadRegister(4);

// conversion du nombre en chaîne de caractères
sprintf(ch,"%i",num);
size = strlen(ch);

// envoi de la chaîne obtenue au pilote de console
synchcons->PutString(ch,size);

// écriture du code de retour dans le registre 2
machine->WriteRegister(2,0);
```

2.6 Système de gestion des fichiers (répertoire *filesys*)

Le système de gestion des fichiers propose des fonctions afin d'organiser un ensemble de fichiers dans une arborescence de style UNIX. Chaque fichier contient un en-tête (*fileheader*), stocké sur disque, décrivant où trouver les données du fichier sur le disque. Chaque répertoire est vu comme un fichier contenant le nom des fichiers contenus dans le répertoire et les adresses disques de leurs *fileheaders*. Le système de fichiers NACHOS possède sur disque : (i) une carte des secteurs libres du disque (classe BitMap) ; (ii) le répertoire racine du système de gestion de fichiers. Ces données sont stockées respectivement dans les secteurs 0 et 1 du disque. Lorsque des opérations sont effectuées sur l'un d'eux, les changements sont immédiatement enregistrés sur le disque si l'opération réussit, sinon aucun changement n'est enregistré.

Le système de gestion des fichiers gère les accès concurrents aux fichiers, organise la hiérarchie des répertoires et permet la création de gros fichiers et de fichiers de tailles extensibles. La figure 2.2 montre une vue synthétique de l'imbrication des différentes structures nécessaires au système de fichier en mémoire.

2.6.1 Classe *FileHeader* (fichiers *filehdr.cc, filehdr.h*)

Les objets du système de fichiers (répertoires et fichiers) possèdent une structure d'en-tête contenant les informations suivantes : taille de l'objet, nombre de blocs qu'il utilise, table des secteurs utilisés. La structure est soit enregistrée sur le disque, soit contenue dans une instance de la classe *FileHeader* quand le fichier/répertoire est ouvert. Les méthodes exportées par la classe *FileHeader* sont listées ci-dessous.

- **bool Allocate(BitMap *bitMap, int fileSize)** : initialise la structure d'en-tête de l'objet et alloue de la place sur le disque pour ses données,
- **void Deallocate(BitMap *bitMap)** : désalloue les blocs de données de l'objet sur disque,
- **void FetchFrom(int sectorNumber)** : initialise la structure d'en-tête d'un objet par lecture sur disque,
- **void WriteBack(int sectorNumber)** : écrit sur le disque les modifications relatives à la structure d'en-tête,
- **int ByteToSector(int offset)** : renvoie le numéro de secteur correspondant à une donnée contenue au déplacement *offset*,
- **int FileLength()** : renvoie le taille de l'objet en octets,
- **void Print()** : affiche le contenu du fichier,
- **bool isDir()** : permet de savoir si un objet est un répertoire ou non,
- **void SetFile()** : mémorise qu'un objet est de type fichier,
- **void SetDir()** : mémorise qu'un objet est de type répertoire.

2.6.2 Classe *FileSystem* (fichiers *filesys.cc, filesys.h*)

Ce module offre des routines pour initialiser le système de fichiers, pour créer, ouvrir ou effacer des fichiers et des répertoires :

- **FileSystem(bool format)**, initialise le système de fichiers. Si *format* est vrai, le disque est vidé et on initialise le système à un répertoire vide et la bitmap en conséquence.
- **bool Create(char *name, int initialSize)**, crée un fichier dans le système. Cette méthode vérifie que le fichier n'existe pas déjà et que son nom est valide, alloue de la place sur le disque pour l'en-tête et pour les données du fichier, ajoute le fichier dans la table du répertoire concerné et met à jour les modifications effectuées sur le disque.
- **OpenFile* Open(char *name)**, ouvre un fichier pour une lecture ou une écriture. On ramène l'en-tête concerné en mémoire.
- **bool Remove(char *name)**, efface un fichier du système, retire le fichier du répertoire qui le contient, efface son en-tête et ses données, met à jour les changements au niveau de la bitmap et du répertoire.

- **void List()**, affiche les contenus de la bitmap, du répertoire de base et pour chacun de ses fichiers l'en-tête et les données.
- **bool Mkdir(char *)**, crée un nouveau répertoire, vérifie que le nom est valide et que le répertoire n'existe pas déjà, alloue de la place pour l'en-tête et pour les données et met à jour les changements sur le disque.
- **bool Rmdir(char *)**, efface un répertoire après avoir vérifié qu'il existe et qu'il est vide.
- **char* decompname(char * name, char *head, char *tail)**, est utilisé pour gérer la hiérarchie des répertoires. *decompname* prend en argument un nom de fichier et le décompose en deux parties, retournées dans *head* et *tail*. Si le paramètre est *name=/dir1/dir2/fic* la fonction renvoie *dir1* dans head et */dir2/fic* dans *tail*.
- **int FindDir(char *name)**, *name* est le nom absolu du fichier et *FindDir* renvoie le secteur du dernier répertoire contenant le fichier. Par exemple pour *name=/dir1/dir2/fic* on obtient le numéro de secteur de dir2 et name devient fic. Cette méthode utilise la précédente et est utilisée dans toutes les méthodes qui nécessitent de modifier le système de fichier ou de vérifier la validité d'un nom.
- **OpenFile *GetFreeMapFile()**, récupère la bitmap.
- **OpenFile *GetDirFile()**, renvoie un *Openfile** sur le fichier du répertoire racine.

2.6.3 Classe *Directory* (fichiers *directory.cc, directory.h*)

La classe *Directory* est une table dont chaque entrée correspond à un fichier et contient le numéro de secteur sur lequel est sauvée sa structure d'en-tête. Cette classe contient les méthodes permettant de garder la cohérence du contenu des répertoires entre le disque et la mémoire.

- **void FetchFrom(OpenFile *file)**, lit le contenu du répertoire depuis le disque,
- **void WriteBack(OpenFile *file)**, écrit les modifications concernant le répertoire sur le disque,
- **Findindex(char *name)**, retourne l'index de *name* dans la table du répertoire,
- **int Find(char *name)**, retourne le numéro de secteur du *FileHeader* de *name*,
- **bool Add(char *name, int newSector)**, rajoute un fichier de nom *name*, de secteur d'en-tête *newSector* au répertoire,
- **bool Remove(char *name)**, efface un fichier du répertoire,
- **void List()**, liste tous les fichiers du répertoire,
- **void Print()**, liste tous les fichiers du répertoire, l'emplacement de leurs structures d'en-tête et le contenu de chaque fichier,
- **bool empty()**, retourne vrai si le répertoire est vide,

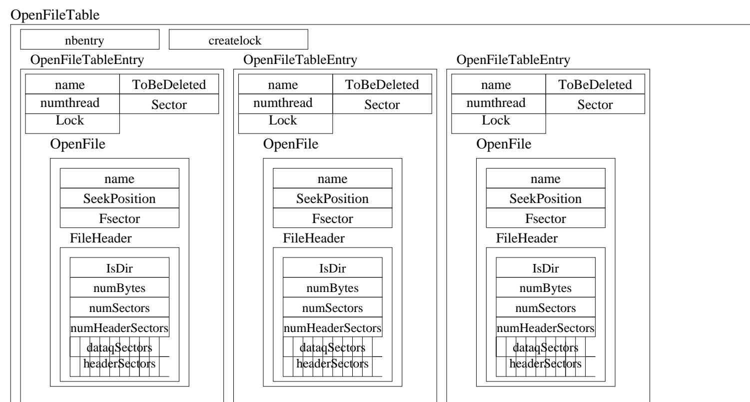


FIG. 2.2 – *Table des fichiers ouverts*

2.6.4 Classe *OpenFile* (fichiers *openfile.cc, openfile.h*)

Cette classe permet, pour un fichier ouvert, de mémoriser les informations sur son utilisation. Lorsqu'un fichier est ouvert, son *FileHeader* est situé en mémoire. Un *OpenFile* contient le nom du fichier qu'il représente, son *FileHeader*, ainsi que la position courante dans le fichier.

- **OpenFile(int f)** réalise l'ouverture du fichier correspondant au numéro de secteur passé en argument,
- **~Openfile()** réalise la fermeture du fichier,
- **void Seek(int position)** modifie la position courante dans le fichier,
- **int ReadAt(char *into, int numBytes, int position)** réalise la lecture de *numBytes* octets à partir de *position* et les place dans le tampon *into*. Le contenu du fichier n'est pas conservé en mémoire (pas de gestion de cache disque).
- **int WriteAt(char *from, int numBytes, int position)**, réalise l'écriture de *numBytes* octets à partir de *position* depuis le tampon *from*. Le contenu de la zone mémoire écrite est transférée immédiatement sur le disque, de manière synchrone.
- **int Length()** retourne la taille du fichier en octets.
- **FileHeader* GetFileHeader()** retourne la structure d'en-tête associée au fichier,
- **char* GetName()** retourne le nom du fichier
- **void SetName(char*)** modifie le nom du fichier.
- **bool isDir()** retourne vrai si le fichier est un répertoire et faux sinon.

2.6.5 Classes *OpenFileTable* et *OpenFileTableEntry* (fichiers *oftable.cc, oftable.h*)

La classe *OpenFileTable* maintient une table des fichiers ouverts afin de gérer la synchronisation de leurs accès. Chaque fois qu'un thread ouvre un fichier, il faut vérifier dans la

table qu'un autre thread ne l'ait pas déjà ouvert pour gérer les accès concurrents de lecture/écriture. Chaque entrée (classe *OpenFileTableEntry*) dans la table contient le nom du fichier, un *OpenFile* du fichier, le nombre de threads qui ont ouvert ce fichier, un verrou pour la synchronisation , le numéro de secteur du *FileHeader*, ainsi qu'un booléen *ToBeDeleted* qui indique que le fichier doit être détruit quand tous les threads l'auront fermé. *OpenFileTable* possède les méthodes suivantes:

- **OpenFile * Open(char *name)**, crée une entrée dans la table pour un fichier qui n'est pas encore ouvert.
- **void Close(char *name)**, est appelée lorsqu'un thread ferme le fichier, décrémente le nombre de threads qui ont le fichier ouvert. Si ce nombre devient nul, il retire le fichier de la table des fichiers ouverts.
- **void Lock(char *name)**, bloque le verrou du fichier afin d'effectuer une écriture exclusive.
- **void Release(char *name)**, relâche le verrou pour permettre des opérations de lecture ou d'écriture sur le fichier après avoir effectué les opérations sur le disque.
- **bool Remove(char *name)**, efface un fichier du répertoire et positionne *ToBeDeleted* à true (aucune nouvelle ouverture du fichier n'est permise, mais le fichier ne sera effectivement détruit que quand tous les threads l'auront fermé).
- **int next_entry()**, pour obtenir la prochaine entrée valide de la table.
- **int findl(char *name)**, pour trouver un fichier dans la table.

2.7 Gestion de la mémoire virtuelle (répertoires *vm, machine*)

2.7.1 Mécanisme de traduction d'adresses

On utilise le mécanisme de traduction d'adresse, qui introduit une séparation entre les adresses virtuelles, que manipule le processeur lors de l'interprétation des programmes, et les adresses physiques qui sont transmises à la mémoire centrale.

Accès à la mémoire (répertoire *machine*, fichier *translate.cc*)

La classe *machine* fournit deux méthodes d'accès à la mémoire virtuelle:

ReadMem(int addr, int size, int *value):

- *addr* : adresse virtuelle de la lecture
- *size* : nombre d'octets à lire (1, 2 ou 4)
- *value* : valeur lue

WriteMem(int addr, int size, int *value):

- *addr* : adresse virtuelle de l'écriture
- *size* : nombre d'octets à écrire (1, 2 ou 4)

- *value* : valeur à écrire

Ces deux fonctions font appel à la méthode :

Translate(int virtAddr, int* physAddr, int size, bool writing)

- *virtAdd* : adresse virtuelle à traduire
- *physAddr* : adresse physique correspondante
- *size* : nombre d'octets de l'accès
- *writing* : type de l'accès (lecture/écriture)

pour transformer l'adresse virtuelle en adresse physique et ainsi pouvoir accéder aux valeurs voulues dans la mémoire de la machine. Cette dernière calcule tout d'abord la page virtuelle associée à l'adresse virtuelle ainsi que le déplacement dans cette page. Elle fait ensuite appel au cache de traduction d'adresses (cf. 2.7.1) pour obtenir la page réelle associée à cette page virtuelle. L'adresse réelle est alors obtenue grâce à la formule : *AdresseRéelle = NuméroPageRéelle * TaillePage + Déplacement*

La fonction de traduction renvoie une exception correspondant au résultat de l'opération :

- *NoException* : la traduction d'adresses s'est bien déroulée,
- *PageFaultException* : la traduction n'est pas valide (i.e. adresse virtuelle invalide),
- *ReadOnlyException* : une écriture à été tentée sur une page protégée en écriture (code),
- *BusErrorException* : la traduction a rendu une adresse physique invalide,
- *AddressErrorException* : la traduction n'était pas alignée sur une adresse paire (lire 4 octets à l'adresse 3).

Le cache de traduction : Classe *TLB* (rép. *machine*, fichier *tlb.cc, tlb.h*)

Le TLB (Translation Look-aside Buffer) est un cache qui conserve les traductions les plus utilisées (au sens de l'algorithme LRU). Ce cache est la première structure utilisée pour passer d'une adresse virtuelle à une adresse réelle. La composition d'une entrée de ce cache est présentée fig. 2.3.

virtualPage	physicalPage	valid	dirty	readonly

FIG. 2.3 – *Structure d'une entrée du TLB*

La signification des différents éléments qui composent une entrée est la suivante :

- *virtualPage* est un numéro de page virtuelle,
- *physicalPage* est le numéro de page réelle correspondant à la page virtuelle,
- *valid* est un bit qui détermine si l'entrée est valide ou non,
- *dirty* est un bit qui signale si la page a été modifiée,
- *readonly* est un bit permettant de connaître les droits en écriture de la page.

Ce cache est totalement associatif. L'ensemble de ses entrées valides est examiné pour savoir si la correspondance est présente ou non.

Si la traduction était absente du TLB, plusieurs opérations doivent être effectuées pour le mettre à jour :

- Appel à la MMU (Memory Management Unit) pour effectuer la traduction
- Mise à jour pour l'entrée concernée des bits *use* et *dirty* à jour dans la TPR (Table des Pages Réelles) par rapport à ceux du TLB

La correspondance récupérée est placée dans le cache à la place de la plus ancienne traduction. L'index de cette entrée est donné par l'algorithme LRU (*lru.cc, lru.h*) (Least Recent Use). Ce dernier se compose d'une liste de n index (n=taille du cache). Le 1er élément est celui qui a servi le moins récemment, le dernier est celui qui a servi en dernier. Deux méthodes sont offertes par l'algorithme LRU :

- **WantOldest()** : renvoie le premier élément de la liste ou un index libre
- **Use(int i)** : place l'index i en fin de liste

Il reste ensuite à :

- Vérifier que l'accès est en accord avec les bits de protection de la page
- Prévenir l'algorithme LRU que cette page est utilisée (elle devient la plus récente)
- Vérifier que l'adresse réelle est dans l'intervalle [0..TailleMémoirePhysique]

L'objet *TLB* comporte quelques autres méthodes :

- **Flush()** : Invalide toutes les entrées du cache (lors d'un changement de contexte)
- **Invalid(i)** : Invalide l'entrée *i* du cache

Table des pages virtuelles : Classes *TranslationTable*, *PageTableEntry*, *BookTableEntry* (rép. *vm*, fichiers *translationtable.cc, translationtable.h*)

Les tables des pages virtuelles (classe *TranslationTable*) de chaque processus sont utilisées en second recours dans le mécanisme de traduction d'adresse. La composition d'une entrée de cette table (classe *PageTableEntry*) est présentée figure 2.4. L'indexation de la table se fait à l'aide du numéro de page virtuelle.

physicalPage	pageDisk	valid	swap	io	readOnly

FIG. 2.4 – *Structure d'une entrée de la table des pages virtuelles*

La signification des différents éléments qui composent une entrée est la suivante :

- *physicalPage* est le numéro de la page réelle correspondant à la page virtuelle,
- *pageDisk* est le numéro de page sur le disque de swap lorsque la page en question y est placée,

- le bit *valid* détermine la validité de l'entrée dans la table,
- le bit *swap* permet de savoir, lorsque la page n'est pas valide (*valid* à 0), si la page est sur la zone de swap (zone modifiable) ou dans le fichier exécutable (zone non modifiable). Sa valeur est respectivement à 1 et 0,
- le bit *io* indique si la page est déjà concernée par une traduction d'adresse entraînant une entrée/sortie disque,
- le bit *readOnly* permet de savoir si la page est protégée en écriture, il prend dans ce cas la valeur 1.

Les différentes méthodes d'accès aux données des tables des pages virtuelles font partie de la classe *TranslationTable* contenue dans les fichiers *translationtable.cc, translationtable.h.* Leurs noms commencent par le préfixe *set* ou *set* suivant que l'on veut lire ou écrire des attributs des entrées de tables et sont suivis du nom de la valeur à lire ou modifier. Par exemple, pour lire la valeur du bit *swap* on utilise la méthode **bool getBitSwap(int virtualPage)** et pour écrire une nouvelle valeur dans *pageDisk*, on utilise la méthode **void setPageDisk(int virtualPage,short int pageDisk)**.

Les tables des pages à un seul niveau sont structurées sous la forme d'un tableau de descripteurs de pages virtuelles. Les entrées (classe *PageTableEntry*) sont indexées directement en utilisant le numéro de la page virtuelle.

Dans le cas où la structure des tables se compose de deux niveaux, la mise en oeuvre se fait par une structure composée d'une table des livres (classe *BookTableEntry*) qui contient des références vers des tables des pages. Pour trouver dans le second niveau l'entrée nécessaire à la traduction d'adresse, on découpe le numéro de page virtuelle dont on dispose en deux parties. La première indexe dans la table des livres une référence vers la table des pages concernée, la seconde indexe cette table des pages pour trouver l'entrée voulue. La figure 2.5 présente la structure des tables à deux niveaux.

L'unité de gestion de mémoire : Classe *MMU* (rép. *machine, fichiers* mmu.cc, mmu.h*)*

Son rôle est de fournir la page réelle associée à une page virtuelle de façon à permettre la mise à jour du TLB. Pour cela, la MMU vérifie que le numéro de la page virtuelle recherchée est bien valide par rapport au contenu de la table de traduction (cf. 2.7.1), puis consulte cette dernière pour y récupérer les informations.

Deux cas se présentent ensuite :

- Soit la page était swappée sur disque (bit *swap* à 1) et donc absente de la mémoire : la MMU déclenche un défaut de page, qui est récupéré dans le noyau de NACHOS par le gestionnaire de défaut de pages (*pageFaultManager.cc, pageFaultManager.h*) qui charge la page en mémoire et renvoie le numéro de la page réelle qui lui est ainsi associée.

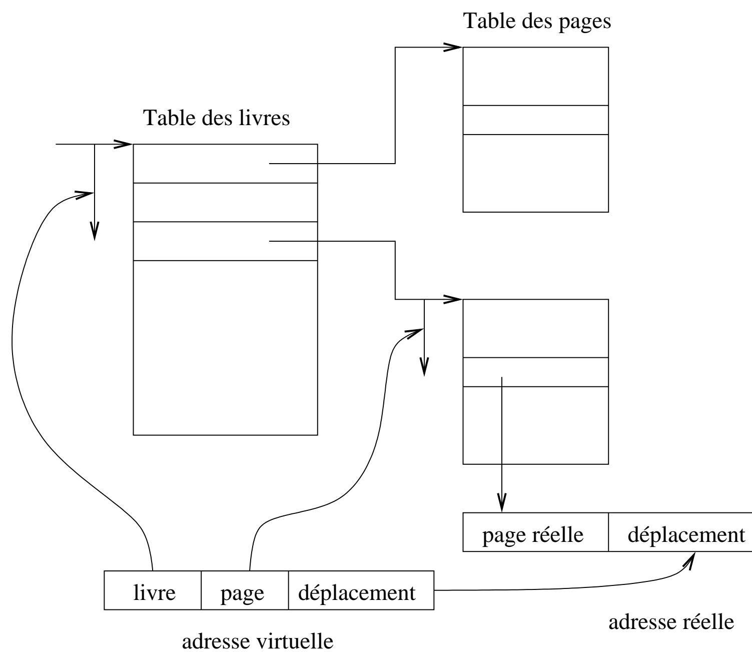


FIG. 2.5 – *Structure d'une table des pages à deux niveaux*

virtualPage	owner	free	use	system	dirty

FIG. 2.6 – *Structure d'une entrée dans la table des pages réelles*

– Soit la page était déjà en mémoire, et la MMU renvoie au TLB les informations dont il a besoin (numéro de la page réelle avec son bit de protection).

Table des pages réelles

La figure 2.6 montre la structure d'une entrée de la table des pages réelles. La signification des différents éléments qui composent une entrée est la suivante :

– *virtualPage* est le numéro de la page virtuelle correspondant à cette page,

– *owner* est un pointeur sur le processus propriétaire de la page,

– *use* indique si la page a été référencée récemment,

– *dirty* indique si son contenu a été modifié,

– *system* signale que la page est en cours de traitement par la voleur de page,

– *free* indique si la page est libre ou non.

2.7.2 Mécanisme de swap

Voleur de pages : Classe RealMemManager (rép. *vm*, fichiers *mem.cc, mem.h*)

Le voleur de pages est appelé lorsque l'on souhaite allouer une nouvelle page mais qu'aucune page libre n'est présente dans la mémoire réelle. Le principe consiste alors à réquisitionner la page d'un autre processus, à la placer dans la zone d'échange sur le disque et à donner la page libérée au processus demandeur.

Pour choisir une page, le voleur fait le tour des entrées de la table des pages réelles de manière circulaire. La page réquisitionnée est choisie suivant certains critères. Elle ne doit pas avoir été référencée récemment (bit *use* à 0) et ne doit pas être marquée comme étant une page système (bit *system* à 0). Pour pouvoir faire la différence entre les pages récemment référencées et les autres, l'algorithme remet à zéro le bit de référence (bit *use*). Ainsi, les pages non utilisées entre deux tours de table du voleur de page, pourront être réquisitionnées puisqu'elles apparaîtront comme non référencées.

Le principe de fonctionnement du voleur de pages semble simple mais il est nécessaire de faire attention à certains problèmes dûs au parallélisme. En effet, si une page réquisitionnée est marquée comme ayant été modifiée, l'algorithme provoque sa recopie sur la zone d'échange. Or l'entrée/sortie est bloquante puisque le pilote de disque utilise des synchronisations. Le processus qui a besoin d'une page de mémoire est placé dans une file d'attente, son exécution est suspendue. Ceci impose à l'algorithme d'être ré-entrant puisque rien n'empêche un autre processus, pendant la suspension du premier, de vouloir lui aussi une nouvelle page et de provoquer l'appel au voleur de pages. A un instant donné, l'algorithme peut chercher une page pour plusieurs processus.

Un autre problème se pose lorsque toutes les pages réelles sont marquées comme étant *système* : c'est à dire qu'il y a autant de voleurs de pages de lancés que de pages dans la mémoire, toutes les pages sont en cours de recopie vers la zone de swap. Il faut dans ce cas suspendre le processus appelant tant que la situation n'évolue pas. Ensuite quand des pages ne sont plus *système*, il faut vérifier que des pages ne sont pas libres. En effet, il se peut qu'entre temps un processus soit terminé et ait libéré ses pages dans la mémoire réelle.

Routine de traitement de défauts de page : Classe PageFaultManager (rép. *vm*, fichiers *pageFaultManager.cc, pageFaultManager.h*)

La routine de traitement de défauts de page est appelé par la MMU quand une demande d'accès à une page virtuelle provoque un défaut de page (i.e. la page n'est pas en mémoire). L'unique méthode de cet objet est : **Exception PageFault(int vpn)** où *vpn* est la page virtuelle en cause.

Cette méthode consulte d'abord le bit *swap* de cette page, pour savoir où est le contenu de la page à charger :

- bit *swap*=1 : la page est dans la zone de swap (page de pile ou de données)

– bit *swap*=0 : la page est à charger dans l'exécutable (page de code ou page de données pas encore chargée)
– bit *swap*=0 et *pageDisk*=-1 : la page n'existe pas (premier accès à une page de pile)

Dans le premier cas, on demande au gestionnaire de swap de charger cette page, dont l'offset est connu grâce au champ *pageDisk* de la table des pages, puis au gestionnaire de mémoire réelle de donner une page réelle libre où l'on transfère alors la page.

Dans les deux autres cas, les opérations à effectuer sont identiques sauf pour l'initialisation de la page :

– page de code ou de données : initialisation par chargement depuis l'exécutable
– page de pile : pas de chargement (page vierge)

Dans tous les cas, la table des pages est mise à jour et la méthode renvoie une exception rendant compte du résultat de l'opération.

2.7.3 Format des fichiers exécutables

Les fichiers sources sont compilés en code MIPS par un compilateur croisé qui génère le format d'exécutable COFF (Common Object File Format). L'exécutable est ensuite tranformé dans un format beaucoup plus simple, appelé NOFF. Ce format de fichier est décrit dans le fichier *noff.h* du répertoire *bin* (voir TP gestion de mémoire virtuelle).

2.8 Les fichiers utilitaires (répertoire *utility*)

NACHOS possède des outils de déboguage, de gestion de listes et de gestion de bitmaps. Cette partie décrit les routines et les accès à ces outils.

2.8.1 Les routines de déboguage (fichiers *utility.h, utility.cc*)

NACHOS offre certaines routines permettant d'afficher des messages facilitant le déboguage des fonctions et méthodes système lors de leur implémentation. Il est possible de sélectionner le 'type' de message de déboguage que l'on désire systématiquement afficher. Chaque type de message est identifié par un drapeau (flag). Une chaîne de caractères interne au système recense tous les 'flags' des messages à afficher. Cette chaîne est construite à partir de la ligne de commande lors du lancement de NACHOS. Voici la liste des flags prédéfinis dans NACHOS :

– '+' – tous types de messages ;
– 't' – threads ;
– 's' – sémaphores, locks, et conditions ;
– 'i' – gestionnaires d'interruptions ;
– 'm' – émulation de la machine ;
– 'd' – émulation du disque ;

– 'f' – système de fichier ;
– 'a' – espaces d'adressage ;
– 'x' – gestion de mémoire virtuelle (VM);
– 'c' – configuration du système ;

Cette liste peut être complétée par d'autres flags suivant vos besoins. Les routines d'aide au déboguage sont les suivantes :

– **void DebugInit(char* flaglist)** initialise la liste des flags positionnés à partir de la chaîne de caractères *flaglist*.
– **bool DebugIsEnabled(char flag)** retourne vrai si *flag* appartient à la chaîne de caractères contenant les 'flags' des messages autorisés à être affichés.
– **void DEBUG(char flag, char *format, ...)** permet d'afficher un message via la console si *flag* correspond à un flag autorisé. La chaîne de caractères passée en paramètre doit être au format standard d'un printf. Les paramètres à afficher sont passés en paramètre à la fonction DEBUG.
– **ASSERT(condition)** permet d'afficher un message d'erreur lorsque la condition passée en paramètre n'est pas vérifiée. Ce message contient la ligne et le fichier depuis lesquels ASSERT a été appelé.

2.8.2 Définition de pointeurs de fonctions (fichier *utility.h*)

Certaines méthodes et fonctions du système ont pour paramètres des fonctions. Il apparaît alors utile de définir un type identifiant une fonction avec argument et un second type identifiant une fonction sans argument. Les lignes suivantes définissent ces types:

– **typedef void (*VoidFunctionPtr)(int arg)**
– **typedef void (*VoidNoArgFunctionPtr)()**

Avec de tels pointeurs de fonctions, l'appel se fait simplement et de la manière suivante:

– (*VoidFunctionPtr)(arg)
– (*VoidNoArgFunctionPtr)()

2.8.3 Les listes (fichiers *list.h, synchlist.cc, synchlist.h*)

NACHOS propose une structure de listes ainsi que les accès classiques aux listes et leurs éléments. Ces listes sont définies dans la classe modèle *List*. Les objets issus de *List* ne chaînent pas directement entre eux les éléments de la liste. Chaque élément est associé à un descripteur d'élément défini dans la classe modèle *ListElement*. Un descripteur contient les informations nécessaires au chaînage, un pointeur vers un élément et sa priorité dans la liste. Dans le cas d'une liste triée, la priorité de l'élément détermine sa place dans la liste, les éléments les

plus prioritaires étant placés en tête de liste. Les constructeurs, destructeurs et accès liste disponibles sont décris ci-dessous:

- **List()** est le constructeur de la classe *List* et permet d'initialiser une liste a vide.
- **˜List()** est le destructeur de la classe *List*. Il efface les descripteurs d'éléments mais pas les éléments eux-mêmes. En effet, chaque élément peut être chaîné dans des listes distinctes ; il n'est donc pas souhaitable de les effacer de la mémoire lors de la destruction d'une liste.
- **void Prepend(void *item)** insère l'élément pointé par *item* en tête de liste.
- **void Append(void *item)** insère l'élément pointé par *item* en queue de liste.
- **void *Remove()** efface le descripteur de tête de la liste.
- **void Mapcar(VoidFunctionPtr func)** applique la fonction pointée par *func* à chaque élément de la liste.
- **bool IsEmpty()** retourne vrai si la liste est vide, faux sinon.
- **void SortedInsert(void *item, Priority sortKey)** insère un élément dans la liste par ordre de priorité croissante.
- **void *SortedRemove(Priority *keyPtr)** permet d'effacer l'élément de tête d'une liste triée. La valeur de la clé de l'élément effacé est contenu à l'adresse *keyPtr* passée en paramètre.
- **bool Search(void *item)** retourne vrai si l'élément pointé par *item* est contenu dans la liste.
- **void RemoveItem(void *item)** efface l'élément pointé par *item* s'il est contenu dans la liste.

Le type de la clé des éléments d'une liste n'est pas prédéfini. La classe *ListElement* est une classe template. Il faut donc préciser le type de la clé lors de son instanciation ou définir de nouveaux types de listes de la façon suivante : typedef List<int> Listint; pour définir une liste donc la clé sera un entier.

2.8.4 Les bitmaps : Classe BitMap (fichiers *bitmap.cc, bitmap.h*)

NACHOS contient également des objets bitmap définis dans la classe *BitMap*. Ce sont des tableaux de bits pouvant être indépendamment à 0 ou à 1. Leur utilisation est particulièrement intéressante dans la gestion de la mémoire et du disque. Les accès aux objets *BitMap* sont les suivants :

- **BitMap(int nitems)**, le constructeur, alloue l'espace mémoire nécessaire pour le stockage des bits. Cette allocation se fait par bloc de 32 bits. Tous les bits sont initialisés à 0.
- **void Mark(int which)** met le bit de rang *which* à 1.

- **void Clear(int which)** met le bit de rang *which* à 0.
- **bool Test(int which)** retourne vrai si le bit de rang *which* est à 1, faux sinon.
- **int Find()** retourne le rang du premier bit à 0 et l'affecte à 1. En d'autres termes, cette fonction trouve et alloue un bit. Si aucun bit n'est à 0, retourne -1.
- **int NumClear()** retourne le nombre de bits à 0.
- **void Print()** affiche le contenu d'un bitmap.
- **void FetchFrom(OpenFile *file)** initialise le contenu d'un bitmap à partir d'un fichier NACHOS. *file* est le pointeur vers le fichier.
- **void WriteBack(OpenFile *file)** affecte le contenu d'un bitmap à partir d'un fichier NACHOS. *file* est le pointeur vers le fichier.

2.9 Mon premier programme Nachos

Cette section présente tout d'abord les appels systèmes disponibles dans NACHOS et les fonctions de la librairie NACHOS. Nous présentons ensuite comment développer un programme utilisateur, le compiler et l'exécuter sous NACHOS.

2.9.1 Appels système disponibles (répertoire *test*, fichiers *start.s*)

L'ensemble des primitives système disponibles depuis un programme utilisateur dans NACHOS sont définies dans les fichiers *syscall.h* et *start.s*. Sauf mention contraire, en cas d'échec, chacune de ces primitives système renvoie -1.

Les primitives disponibles sont :

Gestion de processus légers (threads).

- **void Halt()**: arrête NACHOS et affiche les statistiques de la session à l'écran.
- **int SysTime()**: renvoie le temps passé dans NACHOS au moment de l'appel.
- **void Exit(int status)**: sort du programme utilisateur (*status* = 0 signifie une sortie normale)
- **int Exec(char *name)**: exécute le fichier NACHOS passé en paramètre et retourne l'identificateur d'espace d'adresse.
- **int Join(SpaceId id)**: attend la fin du programme *id*, et renvoie son statut de sortie.
- **int newThread(void (*func)())**: crée un nouveau thread qui exécutera la procédure *func* dans le même espace d'adressage que celui de son père.
- **void Yield()**: commute de thread prêt, dans le même ou vers un autre espace d'adressage (relâchement volontaire du processeur).
- **void PError(char *mess)**: affiche la dernière erreur rencontrée dans l'exécution d'une primitive système, avec en entête le message personnalisé *mess*.

Accès aux fichiers.

- **int Create(char *name,int size)**: crée le fichier NACHOS *name* de taille *size.*
- **OpenFileId Open(char *name)**: ouvre le fichier *name* et renvoie l'identificateur associé à ce fichier.
- **int Write(char *buffer, int size, OpenFileId id)**: écrit *size* octets depuis *buffer* vers le fichier de descripteur *id.*
- **int Read(char *buffer, int size, OpenFileId id)**: lit *size* octets depuis le fichier *id* vers *buffer* et renvoie le nombre de caractères lus, qui peut être inférieur au nombre demandé dans le cas d'un fichier trop petit.
- **int Close(OpenFileId id)**: ferme le fichier identifié par *id.*
- **int Remove(char* name)**: efface le fichier de nom *name.*
- **int Mkdir(char* name)**: crée un nouveau répertoire de nom *name.*
- **int Rmdir(char* name)**: détruit le répertoire de nom *name.*

Synchronisation.

- **SemId SemCreate(int count)**: crée un sémaphore initialisé à la valeur *count* et retourne son identificateur.
- **int SemDestroy(SemId sema)**: détruit un sémaphore spécifié par son identificateur *sema.*
- **void V(SemId sema)**: effectue l'opération V sur le sémaphore *sema.*
- **void P(SemId sema)**: effectue l'opération P sur le sémaphore *sema.*
- **LockId LockCreate()**: crée un verrou et retourne son identificateur.
- **int LockDestroy(LockId id)**: détruit le verrou spécifié par son identificateur *id.*
- **int LockAcquire(LockId id)**: acquisition du verrou d'identificateur *id.*
- **int LockRelease(LockId id)**: relâche le verrou d'identificateur *id.*
- **CondId CondCreate()**: crée une nouvelle variable de condition.
- **int CondDestroy(CondId id)**: détruit la variable de condition d'identificateur *id*
- **int CondWait(CondId cond,LockId lock)**: met dans l'ensemble d'attente associée de la condition *cond* le thread courant, qui relâche le verrou *lock.*
- **int CondSignal(CondId cond,LockId lock)**: le thread possesseur du verrou *lock* réveille un des threads de l'ensemble d'attente de la condition *cond* et lui donne ainsi la possibilité de redemander l'accès au verrou.
- **int CondBroadcast(CondId cond,LockId lock)**: idem mais sur tous les threads de l'ensemble d'attente.

Accès au coupleur série.

- **int TtySend(char *mess)**: envoie le message *mess*, terminé par le caractère nul, via la ligne série. La valeur de retour indique le nombre de caractères effectivement envoyés.
- **int TtyReceive(char *mess,int length)**: reçoit un message *mess* de longueur maximale *length* par la ligne série. La valeur de retour spécifie le nombre de caractères effectivement reçus.

Accès à la console

- **void WriteInt(int val)**: affiche l'entier *val* sur la console.
- **int ReadInt()**: lit un entier depuis la console.

2.9.2 Fonctions de la librairie Nachos (répertoire *test*, fichiers *libnachos.cc, libnachos.h*)

L'ensemble de la librairie NACHOS est décrite dans les fichiers *libnachos.c* et *libnachos.h*. Ces fonctions sont directement accessibles depuis les programmes utilisateur et offrent des facilités supplémentaires pour la programmation par rapport aux seuls appels système. Les fonctions à disposition sont les suivantes :

Opérations sur les processus légers :

- **int threadExec(void (*func)())**: crée un nouveau thread qui exécutera la procédure *func* dans le même espace d'adressage que celui de son père. Cette fonction fait appel à l'appel système *newThread* en gérant correctement la terminaison du thread.

Opérations d'entrées/sortie :

- **void writestr(char *c)** : écrit la chaîne de caractère *c* sur la sortie standard.
- **void newline()** : saute une ligne sur la sortie standard.
- **void writeln(char *c)** : écrit la chaîne de caractère *c*, puis passe à la ligne.
- **void printf(char*format,...)** : affiche la chaîne *format*, qui peut comprendre des références à des variables: %c pour un caractère, %s pour une chaîne, %d ou %i pour un entier, %x pour une chaîne en hexa, %f pour un flottant. Les variables sont spécifiées dans les paramètres suivants du printf. Enfin, la chaîne formatée peut contenir des caractères spéciaux comme \n(retour à la ligne) ou \t (tabulation).

Opérations sur les chaînes de caractères :

- **int strcmp(const char *s1, const char *s2)** : compare les 2 chaînes *s1* et *s2* et renvoie un entier supérieur, égal ou inférieur à zéro suivant que *s1* est supérieur, égal

ou inférieur à *s2*.

- **char* strcpy(char *dst, const char *src)** : copie la chaîne *src* (terminée par un 0) dans *dst*, et renvoie *dst* si la copie a marché.
- **size_t strlen(const char *s)** : renvoie la taille de la chaîne *s* sous le format *size_t*, qui est équivalent au type *int*.
- **char* strcat(char *dst, const char *src)** : concatène la chaîne *src* à la fin de la chaîne *dst*. Renvoie *dst* en cas de succès.
- **int toupper(int c)** : renvoie la majuscule correspondant au caractère(casté en *int*) passé en paramètre.
- **int tolower(int c)** : renvoie la minuscule correspondant au caractère(casté en *int*) passé en paramètre.
- **int atoi(const char *str)** : convertit la chaîne *str* (par exemple "1024") en entier (sur cet exemple 1024).
- **void bcopy(const void *s1, void *s2, size_t n)** : copie *n* caractères de la chaîne *s2* vers la chaîne *s1*.
- **int bcmp(const void *s1, const void *s2, size_t n)** : compare les *n* premiers caractères des chaînes *s1* et *s2*, puis renvoie un entier supérieur, égal ou inférieur à zéro suivant que *s1* est supérieur, égal ou inférieur à *s2*.
- **void bzero(void *s, size_t n)** : met les *n* premiers caractères de la chaîne *s* à 0.

Opérations sur les emplacements mémoire :

- **int memcmp(const void *s1, const void *s2, size_t n)** : compare les *n* premiers octets de 2 zones mémoire. La valeur retournée est égale à zéro si les deux zones sont identiques, -1 si la zone *s1* est inférieure à la zone *s2*, 1 sinon.
- **void* memcpy(void *s1, const void *s2, size_t n)** : copie les n premiers octets de *s2* vers *s1*. Renvoie *s1* si la fonction a marché.
- **void* memset(void *s, int c, size_t n)** : affecte la valeur *c* aux *n* premiers octets à partir de l'adresse *s*.

2.9.3 Compilation d'un programme utilisateur

Un programme utilisateur, écrit en langage C, est compilé au moyen d'un compilateur croisé pour obtenir du code MIPS. Le code obtenu est sous format *.coff*, et pour que le chargement de l'exécutable soit plus aisé, le code est modifié au moyen d'un script, pour obtenir un exécutable, format *.noff*.

Il est possible d'effectuer cette compilation automatiquement, au moyen du *Makefile* du répertoire *test*. Pour cela, il suffit d'enregistrer le programme dans ce répertoire, de rajouter

les lignes suivantes (avec prog.c, le programme à compiler):

```
prog.o: prog.c
        $(CC) $(CFLAGS) -c prog.c
prog  : prog.o start.o libnachos.o
        $(LD) $(LDFLAGS) start.o libnachos.o prog.o  -o prog.coff
        ../bin/coff2noff prog.coff prog
```

2.9.4 Compilation de Nachos

Pour compiler NACHOS, il suffit d'utiliser le fichier *Makefile* qui se trouve à la racine des sources. Si vous changer l'ensemble des fichiers à compiler, ou seulement leur nom ou leur emplacement, le fichier *Makefile.common* doit être modifié en conséquence. La compilation génère un fichier exécutable nommé *nachos* à la racine de l'arborescence des sources NACHOS.

2.9.5 Exécution d'un programme Nachos (répertoire *kernel*, fichiers *main.cc*, *system.cc*)

L'exécutable du système d'exploitation, qui se nomme nachos, a la caractéristique de proposer à l'utilisateur un ensemble de paramètres qui permettent d'initialiser le système. Ces directives d'exécution se partagent en deux classes : certaines peuvent être fournies sur la ligne de commande, en même temps que l'exécutable NACHOS, d'autres sont définies dans un fichier de configuration.

Les directives fournies sur la ligne de commande sont les suivantes:

-d : permet d'afficher à l'écran les messages de déboguage, dont le type est spécifié. Ainsi

```
nachos -d f
```

affiche ceux concernant le système de fichiers. Si cette directive est suivi de +, ou bien s'il n'est pas suivi par un type de message, tous les messages de déboguage sont affichés. L'ensemble des types de messages sont définis dans la section 2.8.1, page 29.

-s : déclenche une exécution "pas à pas" d'un programme utilisateur. Entre chaque instruction est alors affiché le contenu de tous les registres de la machine, ainsi que les interruptions en attente. Appuyer sur entrée entraîne l'exécution d'une nouvelle instruction.

-x : permet de lancer dès l'initialisation du système un programme utilisateur, qui est dans le système de fichiers UNIX. Il faut pour cela donner le nom relatif de ce programme par rapport au répertoire courant.

-c : permet de tester la console. Il est possible de spécifier l'entrée et la sortie standard. Par défaut, le clavier et le terminal sont les domaines d'entrée-sortie.

-z : affiche un message de copyright de NACHOS.

-f <nomfich>: utilise le fichier de configuration <nomfich> à la place du fichier de configuration par défaut nachos.cfg.

Il est évidemment possible de combiner lors du lancement plusieurs directives.

2.9.6 Configuration de Nachos (répertoire *utility*, fichiers *config.cc, config.h*)

NACHOS possède aussi un fichier de configuration, dont l'intérêt est de permettre de modifier des paramètres du système au lancement de l´exécutable, sans avoir besoin de recompiler tout le système. Ce fichier de configuration est le fichier *nachos.cfg* par défaut (il doit se trouver à l'endroit à partir duquel on lance NACHOS). Le nom de ce fichier peut être modifié en utilisant l'option -f. Il y est ainsi possible de déterminer la structure du système et d'effectuer des opérations, avant le lancement du système, sur le système de fichiers. Pour spécifier un paramètre, la syntaxe est la suivante :

`<nom de paramètre> = <valeur du paramètre>`

Structure interne de Nachos

Les différents éléments qui peuvent être spécifiés par l'utilisateur sont :

`TimeSharing = <entier>`

Un partage de temps est effectué.

`AddressDump = [ 0 | 1 ]`

Permet de récupérer dans un fichier nommé *adresse* tous les adresses virtuelles qui ont été adressées lors de l'exécution d'un ou plusieurs programmes utilisateurs.

`NumPhysPages  = <entier>`

Le nombre de pages physiques dans la mémoire.

`PageTableSize = <entier>`

La taille de la table des pages.

`TableType = <TableType>`

La table des pages peut être à un seul niveau (*SingleLevel*), ou à deux niveaux (*DualLevel*).

`ExistTLB = [ 0 | 1 ]`

L'utilisation (1) ou non (0) du TLB.

`TLBSize = <entier>`

Le nombre d'entrée du TLB.

`ProcessorFrequency = <entier>`

Précise la fréquence du processeur MIPS émulé (en MHz). Sert dans les statistiques à regarder l'impact de la fréquence du processeur sur les performances des applications, à performances de périphériques équivalentes.

`SectorSize = <entier>`

Précise la taille d'un secteur du disque (en octets). Doit être une puissance de deux.

`PageSize = <entier>`

Précise la taille d'une page en mémoire physique (en octets). Doit être une puissance de deux. Dans la mise en œuvre actuelle du système de pagination, la taille d'une page doit être identique à la taille d'un secteur.

`useACIA =  [ 0 | 1 ]`

Le coupleur série est nécessaire (1) ou non (0). Si ce n'est pas le cas celui-ci n'est pas instancié, ce qui permet de rendre NACHOS plus rapide (Comparez les temps de scrutation du tampon du coupleur, au temps correspondant à une entrée sortie pour comprendre l'intérêt de ce paramètre).

`TargetMachineName = <nom>`

Le nom de la machine cible, lors de la communication via le coupleur série.

`NumPortLoc = <entier>`

Le numéro de port local (>1024) d'enregistrement du service réseau pour la simulation de la communication par le port série.

`NumPortDist = <entier>`

Le numéro de port distant (>1024) du service réseau pour la simulation de la communication par le port série.

Gestion du système de fichiers

Les opérations sur le système de fichiers sont effectuées avant le lancement du système, et de ce fait ces instructions ne sont pas prises en compte dans les statistiques.

`ProgramToRun = <nom>`

Le nom du programme utilisateur à lancer. Attention, il faut indiquer le chemin relatif par rapport à la racine du système de fichier NACHOS.

`FormatDisk =  [ 0 | 1 ]`

Permet d'effectuer le formattage du disque.

`ListDir =  [ 0 | 1 ]`

Affiche toute l'arborescence du système de fichiers, avec la position des fichiers.

`PrintFileSyst = [ 0 | 1 ]`

Affiche tout le contenu du système de fichiers.

`FileToPrint = <position Nachos>`

Affiche le contenu d'un fichier de NACHOS.

`FileToCopy =  <position Unix> <position Nachos>`

Copie un fichier Unix dans le système de fichier NACHOS. Les deux noms à donner sont les noms relatifs, respectivement par rapport au répertoire courant UNIX, et par rapport à la racine du système NACHOS.

`FileToRemove = <position Nachos>`

Retire le fichier spécifié du système.

`NumDirEntries = <entier>`

Correspond au nombre d'entrées par répertoire.

`DirToMake = <position Nachos>`

Crée un répertoire dans l'arborescence de NACHOS.

`DirToRemove = <position Nachos>`

Efface un répertoire de NACHOS.
Les paramètres ont des valeurs par défaut (voir le constructeur de la classe *Config*).

Chapitre 3

Sujets de travaux pratiques

3.1 Avant propos

3.1.1 Vue d'ensemble des sujets

Les travaux pratiques sont décomposés en quatre parties, les trois dernières reposant sur la première :

- Réalisation du noyau NACHOS (partage du processeur entre threads, outils de synchronisation, mécanisme d'appel système)
- Gestion d'entrées/sorties via la réalisation d'un pilote série
- Amélioration du système de gestion de fichiers de NACHOS
- Gestion de mémoire virtuelle, via la mise en place d'un système complet de gestion de la mémoire virtuelle (chargement des pages à la demande, stockage des objets dans une zone d'échanges).

Passez voir les enseignants si vous ne parvenez pas à la fin d'un TP pour ne pas être bloqués pour les TPs suivants.

3.1.2 Consignes lors de la réalisation des TPs

- Pour vous y retrouver, et permettre aux enseignants de repérer votre code au milieu du code existant, encadrez votre code par les directives *ifdef/endif* suivantes, en n'oubliant pas d'encadrer aussi les zones de commentaires :

```
#ifndef ETUDIANTS_TP_n
  <le vieux code que vous voulez changer pour le TP numero n>
#endif ETUDIANTS_TP_n
#ifdef ETUDIANTS_TP_n
  <le beau code que vous voulez ajouter a la place>
#endif ETUDIANTS_TP_n
```

Cette consigne est **IMPERATIVE**. Elle servira aux enseignants à récupérer automatiquement votre code parmi le code de NACHOS.

- Vous ne devez pas modifier le répertoire machine, ni les parties gérant le contexte noyau des threads.
- Pour mettre aux points votre système, vous disposez d'un système d'affichage de traces. N'hésitez pas non plus à utiliser un débogueur, comme par exemple *gdb* (une aide en ligne est disponible avec *gdb*)

3.2 Mise en place de Nachos

Pour commencer à utiliser Nachos, recopier le contenu du répertoire :

/home/clyde/d1/insa4/commun/Nachos

dans votre répertoire personnel. Il est conseillé de ne pas recopier le sous répertoire *doc*, qui contient une documentation en-ligne de Nachos.

Une version *html* du manuel de référence est disponible sous *doc/html* (fichier d'entrée *index.html*. Une version imprimée du manuel de référence est aussi disponible dans les salles de TP.

3.3 Questionnaire d'exploration de Nachos

Sont consignées ici quelques questions pour vous aider à rentrer dans le code de NACHOS. Les réponses aux questions s'obtiennent en lisant (et relisant) ce document et le code source de NACHOS, et les pages *html* générées automatiquement à partir du code source de NACHOS. Le moyen de plus simple pour explorer le source est d'utiliser la commande *grep* qui recherche une chaîne dans un fichier ou ensemble de fichiers (faire *man grep*).

Mécanisme d'appel système

Examiner l'exécution d'un programme utilisateur qui nécessite un appel au système. On pourra par exemple regarder le fichier *test/hello.c*, qui se contente d'afficher un message sur la console.

1. Lister les fichiers empruntés lors de l'exécution du programme
2. Lister les méthodes appelées et leur rôle

Gestion de threads et de processus

1. Quelle variable est utilisée pour mémoriser la liste des threads prêts à s'exécuter ? Est-ce que le thread élu appartient à cette liste ? Comment accéder à ce thread ?

2. A quoi sert la variable *Alive* ? Quelle est la différence avec le champ *readyList* de l'objet scheduler ?
3. Comment se comportent les routines de gestion de listes vis à vis de l'allocation de mémoire ? Est-ce qu'elles se chargent d'allouer/désallouer les objets chaînés dans la liste ? Pourquoi ?
4. A quel endroit est placé un objet thread quand il est bloqué sur un sémaphore ?
5. Comment faire en sorte qu'on ne soit pas interrompu lors de la manipulation des structures de données du noyau ?
6. A quoi sert la méthode *Run* de l'objet scheduler ? Quel est le rôle des variables *userRegisters*, *kernelStackTop*, *machineState* et *kernelStack* de l'objet thread ? Que doit faire la méthode *Run* de ces variables ? Quand doit-on appeler les méthodes *SaveUserState* et *RestoreUserState* d'un objet thread ?
7. Comment initialiser la variable *aspace* lors de la création d'un nouveau thread par la méthode *newThread* ?

Environnement de développement

1. Quels sont les outils offerts par NACHOS pour la mise au point des programmes utilisateur ? Comment par exemple visualiser toutes les opérations effectuées par la machine MIPS émulée ?
2. Peut-on utiliser l'utilitaire *gdb* pour mettre au point le code de NACHOS ? Le lancer et visualiser le contenu de différentes variables du noyau.
3. Peut-on utiliser *gdb* pour mettre au point les programmes utilisateur ? Pourquoi ?

3.4 TP 1 : Gestion de processus et synchronisation (10h)

Ce premier TP nécessite d'analyser uniquement le contenu des répertoires *kernel* et *test* et de lire la documentation associée.

Il est important de souligner qu'avec la version de NACHOS qui vous est livrée au départ, bien qu'il soit possible de compiler le noyau, il n'est pas encore possible d'exécuter des programmes. Ce sera possible (pour des programmes non multithreadés, comme par exemple le programme *hello*) d'exécuter des programmes dès que les outils de synchronisation seront développés (ces outils sont nécessaires pour les pilotes de périphériques).

3.4.1 Gestion de processus légers

Dans la version des sources qui vous a été livrée, un seul programme utilisateur s'exécute à un moment donné, et il est composé d'un seul flot d'exécution (thread). L'objectif de cette première partie du TP est de supporter plusieurs threads. Pour cela, vous devrez écrire ou compléter les méthodes suivantes :

- méthode newThread de la classe Thread (§ 2.4.1), dont l'objectif est de créer un nouveau thread. On initialisera les différents champs de la classe Thread. On utilisera la méthode *StackAllocate* de la classe *AddrSpace* pour allouer une nouvelle pile, et on utilisera la méthode initKernelThread de la classe Thread pour initialiser son contenu (contexte noyau et utilisateur). Ensuite, le thread sera inséré dans la file des prêts.
- méthode Finish de la classe Thread, dont l'objectif est de marquer le thread comme étant détruit (flag *threadToBeDestroyed*) et de l'enlever de la file des prêts. La destruction proprement dite n'est pas effectuée tout de suite, car on est encore en train d'utiliser sa pile; elle sera effectuée dans l'ordonnançeur (méthode Run) après le changement de contexte.
- méthode Run de la classe Scheduler, dont l'objectif est de donner la main à un thread. Cette méthode devra sauvegarder le contexte du thread ayant l'unité centrale (objet currentThread) et restaurer celui du thread à élire. La sauvegarde/restauration du contexte noyau est déjà écrit; il suffit d'appeler la fonction SWITCH (cette fonction effectue le changement de pile).

On testera les débordements de pile lors du changement de contexte entre threads.

3.4.2 Outils de synchronisation

L'objectif de cette seconde partie est d'implanter des outils de synchronisation (voir § 2.4.2). Le travail demandé est le suivant :

1. Créer le code nécessaire à l'introduction de tous les appels systèmes nécessaires à l'introduction des outils de synchronisation. Pour ce faire, consulter le paragraphe 2.5 qui

décrit le fonctionnement des appels système sous NACHOS. Les fichiers concernés sont les suivants :

- fichier assembleur MIPS *start.s*, qui est lié avec les programmes utilisateur et utilise l'instruction MIPS *syscall* pour exécuter un point d'entrée du noyau (voir paragraphe 2.5)
- fichier *exception.cc*, qui intercepte tous les appels aux noyau et selon le numéro de l'appel (contenu du registre MIPS *r2*) appelle la méthode mettant en œuvre l'appel).

2. Compléter le fichier *sync.cc* pour écrire le code des sémaphores et verrous (locks). L'atomicité des primitives de synchronisation sera assurée par interdiction des interruptions pendant les primitives.
3. Tester de manière intensive votre code via l'écriture :
   - d'un programme de rendez-vous entre 3 threads
   - d'un programme établissant un schéma de synchronisation producteur/consommateur

On fera en sorte de mettre en attente le premier thread créé dans le cas ou il se termine le premier. On le bloquera sur un sémaphore jusqu'à ce que tous les autres threads soient terminés.

3.4.3 Espaces d'adressage séparés

Dans la version de NACHOS dont vous disposez, un seul processus (multi-threadé si vous avez terminé la première partie de ce TP) s'exécute. Ce qui est demandé ici est de permettre l'exécution de plusieurs processus ayant des espaces d'adressage séparés (ayant chacune leur table des pages privées). On écrira pour cela le corps des méthodes *SaveState* et *RestoreState* de la classe *AddrSpace*, chargées de sauvegarder/restaurer le contexte d'un thread spécifique à son espace d'adressage (gestion des champs *Table* et *pageTableSize* de l'objet *machine*).

Dès que vous aurez terminé cette partie du TP, vous pourrez utiliser le *shell* fourni avec NACHOS.

3.4.4 Bonus

Ajout à NACHOS de deux des trois points suivants :

- implantation des variables de condition et démonstration de leur fonctionnement par un programme de test
- gestion des threads par priorité
- ajout de paramètres aux programmes utilisateur

3.5 TP 2 : Gestion d'entrées/sorties caractères (4 h)

Ce TP nécessitera une intervention au niveau du répertoire *handler* (fichiers *ACIA_handler.h* et *ACIA_handler.cc*).

3.5.1 Construction d'un pilote série

L'objectif de ce TP est de gérer l'envoi et la réception de message via une liaison série. Il s'agit de créer un pilote (ou handler) de coupleur série, capable d'émettre et de recevoir de manière asynchrone des lignes de caractères. Pour cela nous utiliserons le coupleur série. La synchronisation sera effectuée en utilisant le noyau de synchronisation écrit dans le TP précédent.

3.5.2 Description du pilote de coupleur série

Un pilote de coupleur série est un dispositif capable d'échanger des messages. Un message est une séquence d'au plus *lmax* caractères ASCII, terminée par le caractère de code ASCII 0. Les opérations offertes par le pilote permettront d'émettre et de recevoir une ligne (voir § 2.3.1). Le pilote gérera le coupleur par attente active dans un premier temps puis par interruptions dans un second temps. Le pilote est représenté par l'objet *handler*, instanciation de la classe *ACIA_handler*.

3.5.3 Les procédures d'émission et de réception

Avec attente active. La procédure d'émission gère l'envoi du tampon passé en paramètre, caractère par caractère, en procédant à une attente active sur le registre de données en émission.

La procédure de réception gère la réception du message, caractère par caractère, en procédant à une attente active sur le registre de données en réception.

Sous interruptions. La procédure d'émission remplit le tampon d'émission (si ce dernier est libre), puis demande les interruptions en émission. C'est ensuite la routine de traitement d'interruptions en émission qui entretient le transfert de la ligne. Lorsque ce dernier a transféré le dernier caractère, il effectue un V sur le sémaphore d'accès au tampon d'émission pour le libérer et permettre l'exécution de la requête en émission suivante.

En réception, la routine de traitement d'interruptions stocke tous les caractères reçus dans le tampon de réception (si ce dernier est libre) jusqu'à la fin du message. A la réception du dernier caractère de la ligne, la routine de traitement d'interruption fait un V sur le sémaphore d'accès au tampon de réception. La procédure de réception vide alors le contenu du tampon de réception pour le transmettre à l'appelant. Enfin elle libère le tampon en réception pour permettre l'exécution de la requête en réception suivante.

3.5.4 Travail demandé

Le travail demandé est le suivant:

1. Implémenter dans le fichier *handler/ACIA_handler.cc* :
   - le constructeur de la classe *ACIA_handler* qui réalise l'initialisation du coupleur;
   - les méthodes **tty_send** et **tty_receive** de manière à fonctionner en attente active, en prévoyant le partage du coupleur série entre plusieurs threads.
2. Ecrire un programme de test avec un émetteur s'exécutant sur une machine et un consommateur s'exécutant sur une autre machine.
3. Les exécuter sur deux machines distantes.
4. Reprendre la première partie de manière à fonctionner sous interruptions. Implémenter les méthodes **interrupt_send** et **interrupt_receive**.
5. Reprendre la troisième étape et comparer les deux modes de fonctionnement à l'aide des statistiques obtenues.

3.5.5 Annexe

Le fichier de configuration. Veillez bien à mettre à jour les champs *useACIA* et *TargetMachineName* dans le fichier de configuration */kernel/nachos.cfg*. Attention! l'utilisation du coupleur série induit une nette diminution de la vitesse d'exécution du système. Il est judicieux de copier les fichiers test sans que le système utilise le coupleur série, puis de redémarrer le système en mettant à jour les champs *useACIA*, et *FormatDisk*.

3.6 TP 3 : Amélioration du système de gestion de fichiers de Nachos (6h)

Pour ce TP, il sera nécessaire de prendre connaissance du pilote de disque (répertoire *handler*, fichiers *synchdisk.h* et *synchdisk.cc*). Une intervention sera nécessaire dans le répertoire *filesys*.

3.6.1 Système de fichiers initial

Le code d'un système de fichiers initial vous est donné. Il comprend le disque émulé (fichiers *disk.cc* et *disk.h*), des méthodes d'accès à ce disque (fichiers *synchdisk.cc* et *synchdisk.h*). Il permet la création de gros fichiers et de fichiers extensibles mais il possède des limitations, notamment il n'y a pas de synchronisation : un thread peut lire un fichier pendant qu'un autre le modifie ou bien 2 threads peuvent modifier le même fichier en même temps ce qui peut conduire à des absurdités. Il n'y a pas non plus de hiérarchie des répertoires mais un seul répertoire qui contient tous les fichiers. C'est à vous de l'améliorer pour obtenir une version qui corresponde à la description qui a été donnée dans ce document, comme détaillé ci-dessous.

3.6.2 Synchronisation

Mettre en place une synchronisation lecteur/rédacteur lors des accès aux fichiers (sans blocage à l'ouverture). Lors de la destruction d'un fichier, celui-ci doit être retiré de son répertoire mais ses données restent sur le disque jusqu'à ce que tous les threads l'aient fermé. Vos modifications doivent respecter les contraintes suivantes :

- Chaque thread possède une instance de la classe *openFile* qui fournit une *position propre* dans le fichier, ainsi 2 threads peuvent lire ou écrire dans le même fichier en même temps sans interférence.
- Les opérations du système de fichiers doivent être *atomiques*. Par exemple si un thread lit un fichier qu'un autre modifie, il doit voir soit toutes les modifications, soit aucune. Par contre, si l'appel de la lecture se fait après que l'écriture soit finie, il doit voir toutes les modifications. Il est naturel de vouloir verrouiller un fichier au niveau de *WriteAt* et *ReadAt* mais la première faisant parfois appel à la seconde, on peut provoquer un inter-blocage. Il faut donc verrouiller au niveau des appels système.
- Quand un thread demande la destruction d'un fichier, les threads qui avaient ouvert ce fichier peuvent encore lire ou écrire dedans jusqu'à ce qu'ils le ferment. Par contre, une fois que la requête de suppression a été effectuée, aucun thread ne peut l'ouvrir. Un fichier peut être détruit sans être ouvert.

Pour mettre en place cette synchronisation, vous devez maintenir une table des fichiers ouverts. L'interface de la classe OpenFileTable vous est donnée (fichier oftable.h).

3.6.3 Hiérarchie des répertoires

Dans le système initial, tous les fichiers sont dans le même répertoire. Vous devez écrire les méthodes *FindDir*, *Mkdir*, *Rmdir* et modifier *Open*, *Create* et *Remove* de *Filesys* afin que le système possède une hiérarchie de répertoires. L'arborescence doit être de type UNIX avec des noms complets du type */dir1/dir2/fic* (ne pas oublier le premier / qui représente le répertoire racine). Vous n'avez pas à gérer la maintenance d'un répertoire courant ni les droits sur les différents fichiers.

3.6.4 Bonus

Gérer un cache en mémoire (de taille bornée) contenant les secteurs de fichiers les plus utilisés. On regardera en utilisant le module de statistiques l'influence sur les performances de programmes.

3.7 TP 4 : Gestion de mémoire virtuelle (10h)

L'objectif de ce TP est de réaliser les deux éléments centraux d'un système de gestion de mémoire virtuelle :

- la routine de traitement des défauts de page qui charge en mémoire à la demande les pages depuis le disque,
- un algorithme de remplacement de page, appelé également *voleur de page*.

On étudiera pour ce dernier TP le contenu du répertoire *vm*. Le système de gestion de mémoire virtuelle doit correspondre à la description donnée au paragraphe 2.7.

3.7.1 Chargement des programmes à la demande

Jusqu'à présent, le code et les données des programmes étaient chargés en mémoire dès leur lancement. Il s'agit ici de changer la méthode de chargement (constructeur de l'objet *AddrSpace* (voir § 2.4.1) de manière à ne pas allouer les pages en mémoire dès le chargement, mais plutôt de déclencher un défaut de page lors de leur premier accès, en mettant le bit *valid* de la table de traduction d'adresses à 0. Les pages de code seront protégées contre l'écriture.

3.7.2 Routine traitement des défauts de pages

Ecrire la routine de traitement des défauts de page (méthode *PageFaultManager* de la classe *PageFault*). Dans un premier temps, on supposera qu'il existe toujours une page de libre en mémoire réelle, et qu'il n'y a qu'un seul thread qui s'exécute dans le programme que l'on exécute. Les actions à réaliser par la routine sont les suivantes :

1. recherche d'une page libre en mémoire réelle (on utilisera pour cela le gestionnaire de mémoire physique - classe *RealMemManager* du fichier *mem.cc*).
2. charger la page manquante depuis le disque (le numéro de secteur sera trouvé dans la table de traduction d'adresses, et aura été initialisée au chargement du programme en mémoire).
3. modifier la table de traduction d'adresses et la table des pages réelles en conséquence.

3.7.3 Algorithme de remplacement de page

Lorsque la mémoire physique est pleine et qu'un processus veut charger en mémoire une page qui est absente de la mémoire physique, la routine de traitement des défauts de page appelle un algorithme de remplacement de page, qui réquisitionne une page à un processus, en la recopiant sur disque si nécessaire. On vous demande ici d'implanter l'algorithme de remplacement de page qui a été présenté dans le paragraphe 2.7. Ceci sera réalisé en complétant la méthode *clock_algorithm* de la classe *RealMemManager*.

Les structures de données et constantes utilisées par l'algorithme de remplacement de page sont :

- la table des pages réelles
- le nombre de pages réelles, et la taille d'une page, qui sont accessibles via l'objet de configuration *cfg* (champs *cfg->NumPhysPages* et *cfg->PageSize*)
- la table de traduction d'adresses du processus courant, accessible via l'objet machine (champ *machine->Table*).
- les objets modélisant le tlb (objet *tlb*) et le gestionnaire de swap (objet *swap*)
- la mémoire de la machine, accessible via l'objet machine (champ *machine->mainMemory*)

3.7.4 Utilisation du TLB

Configurez NACHOS pour qu'il utilise le TLB (le code vous est fourni. Examinez l'impact du TLB sur les performances du système.

3.7.5 Bonus

On pourra implanter une des fonctionnalités suivantes :

- table des pages inversées à la place des tables des pages telles qu'elles sont prévues dans NACHOS
- segments de mémoire partagés par plusieurs processus
- un mécanisme de copy-on-write

3.7.6 Trucs et astuces

Nous énumérons ci-dessous quelques trucs vous permettant d'éviter des problèmes lors de la mise au point de votre système de gestion de la mémoire virtuelle :

- Un processus P1 qui recopie une page réelle X sur disque perd la main sur l'I/O disque. Pour éviter qu'un autre processus P2 veuille aussi recopier cette même page X sur disque suite à son propre appel au voleur de page, on a introduit le bit *system* associé à chaque page en mémoire réelle.
- Le fait qu'une page ait une entrée dans le TLB n'exclut pas que cette page soit réquisitionnée par le voleur de page pour être recopiée sur disque. A chaque fois qu'une page est réquisitionnée, n'oubliez pas d'invalider son entrée dans le TLB.
- Quand un processus P1 perd la main sur une I/O disque, le processus P2 qui prend la main peut accéder au voleur de page : il faut donc faire attention à ce que P1 retrouve son contexte quand il reprendra la main. Pour cela, utiliser une *variable locale local_i_clock* pour le parcours de la table des pages réelles dans l'algorithme de remplacement de page par chaque processus, tout en maintenant une variable globale *i_clock*.

- Soientt deux threads *T1* et *T2*. Si *T2* déclenche un défaut de page pendant que *T1* est en train de résoudre ce même défaut de page, il faut que *T2* se bloque pendant la résolution du défaut de page. Pour ce faire, on utilisera le bit *IO* présent dans la table de traduction d'adresses.

Table des matières

1 Introduction 3

2 Une introduction à Nachos 5

2.1 Présentation de la structure du système 5

2.1.1 Nachos est un processus UNIX 5

2.1.2 Émulation du matériel 6

2.1.3 Modules principaux du noyau 6

2.1.4 Programmes utilisateur (répertoire *test*) 8

2.1.5 Paramétrage du système 9

2.2 La machine émulée (répertoire *machine*) 9

2.2.1 Le processeur MIPS (fichiers *machine.cc, machine.h*) 9

2.2.2 Le contrôleur d'interruptions (fichiers *interrupt.cc, interrupt.h*) 10

2.2.3 Le coupleur série (fichiers *ACIA.h, ACIA.cc*) 10

2.2.4 Le disque (fichiers *disk.cc*, *disk.h*) 12

2.2.5 La console (fichiers *console.cc, console.h*) 12

2.2.6 Le TLB (Translation Look-aside Buffer) (fichiers *tlb.cc, tlb.h*) 12

2.2.7 La MMU (Memory Management Unit, fichiers *mmu.cc, mmu.h*) . . . 12

2.3 Les pilotes de périphériques (répertoire *handler*) 13

2.3.1 Le pilote du coupleur série (fichiers *ACIA_handler.cc, ACIA_handler.h*) 13

2.3.2 Le pilote de la console (fichiers *synchcons.cc, synchcons.h*) 14

2.3.3 Le pilote du disque (fichiers *synchdisk.cc, synchdisk.h*) 14

2.4 Le noyau de Nachos (répertoire *kernel*) 15

2.4.1 Fonctionnement interne du noyau 15

2.4.2 Outils de synchronisation (fichiers *synch.cc, synch.h*) 17

2.5 Fonctionnement des appels systèmes 18

2.5.1 Fonctionnement global 18

2.5.2 Exemple de déroulement d'un appel système 18

2.6 Système de gestion des fichiers (répertoire *filesys*) 19

2.6.1 Classe *FileHeader* (fichiers *filehdr.cc, filehdr.h*) 20

2.6.2 Classe *FileSystem* (fichiers *filesys.cc, filesys.h*) 20

2.6.3 Classe *Directory* (fichiers *directory.cc, directory.h*) . . . 21
2.6.4 Classe *OpenFile* (fichiers *openfile.cc, openfile.h*) . . . 22
2.6.5 Classes *OpenFileTable* et *OpenFileTableEntry* (fichiers *oftable.cc, oftable.h*) . . . 22
2.7 Gestion de la mémoire virtuelle (répertoires *vm, machine*) . . . 23
2.7.1 Mécanisme de traduction d'adresses . . . 23
2.7.2 Mécanisme de swap . . . 28
2.7.3 Format des fichiers exécutables . . . 29
2.8 Les fichiers utilitaires (répertoire *utility*) . . . 29
2.8.1 Les routines de déboguage (fichiers *utility.h, utility.cc*) . . . 29
2.8.2 Définition de pointeurs de fonctions (fichier *utility.h*) . . . 30
2.8.3 Les listes (fichiers *list.h, synchlist.cc, synchlist.h*) . . . 30
2.8.4 Les bitmaps : Classe BitMap (fichiers *bitmap.cc, bitmap.h*) . . . 31
2.9 Mon premier programme Nachos . . . 32
2.9.1 Appels système disponibles (répertoire *test*, fichiers *start.s*) . . . 32
2.9.2 Fonctions de la librairie Nachos (répertoire *test*, fichiers *libnachos.cc, libnachos.h*) . . . 34
2.9.3 Compilation d'un programme utilisateur . . . 35
2.9.4 Compilation de Nachos . . . 36
2.9.5 Exécution d'un programme Nachos (répertoire *kernel*, fichiers *main.cc, system.cc*) . . . 36
2.9.6 Configuration de Nachos (répertoire *utility*, fichiers *config.cc, config.h*) 37

3 Sujets de travaux pratiques **41**
3.1 Avant propos . . . 41
3.1.1 Vue d'ensemble des sujets . . . 41
3.1.2 Consignes lors de la réalisation des TPs . . . 41
3.2 Mise en place de Nachos . . . 42
3.3 Questionnaire d'exploration de Nachos . . . 42
3.4 TP 1 : Gestion de processus et synchronisation (10h) . . . 44
3.4.1 Gestion de processus légers . . . 44
3.4.2 Outils de synchronisation . . . 44
3.4.3 Espaces d'adressage séparés . . . 45
3.4.4 Bonus . . . 45
3.5 TP 2 : Gestion d'entrées/sorties caractères (4 h) . . . 46
3.5.1 Construction d'un pilote série . . . 46
3.5.2 Description du pilote de coupleur série . . . 46
3.5.3 Les procédures d'émission et de réception . . . 46
3.5.4 Travail demandé . . . 47

3.5.5 Annexe 47
3.6 TP 3 : Amélioration du système de gestion de fichiers de Nachos (6h) 48
3.6.1 Système de fichiers initial 48
3.6.2 Synchronisation 48
3.6.3 Hiérarchie des répertoires 49
3.6.4 Bonus 49
3.7 TP 4 : Gestion de mémoire virtuelle (10h) 50
3.7.1 Chargement des programmes à la demande 50
3.7.2 Routine traitement des défauts de pages 50
3.7.3 Algorithme de remplacement de page 50
3.7.4 Utilisation du TLB 51
3.7.5 Bonus 51
3.7.6 Trucs et astuces 51

Index **54**

Index

Symboles

-c 36
-d 36
-f 36
-s 36
-x 36
-z 36
~List() 31
~Openfile 22

A

ACIA 10
ACIA_handler 13
Acquire 17
Add 21
AddressDump 37
AddressErrorException 24
Addrspace 15, 16
adresses physiques 23
adresses virtuelles 23
Allocate 20
appels systèmes MIPS 18
Append 31
ASSERT 30
asynchrone 14
atoi 35

B

bcmp 35
bcopy 35
BitMap 31
bitmap
- BitMap 31
- Clear 32
- FetchFrom 32
- Find 32
- Mark 31
- NumClear 32
- Print 32
- Test 32
- WriteBack 32

bitmaps 31
BookTableEntry 25
Broadcast 18
BusErrorException 24
BUSY_WAITING 11
ByteToSector 20
bzero 35

C

cache associatif 25
cache de traduction 24
Clear 32
Close 23, 33
COFF 29
compilation 35
CondBroadcast 33
CondCreate 33
CondDestroy 33
CondSignal 33
CondWait 33
console 12
constantes
- BUSY_WAITING 11
- EM_INTERRUPT 11
- IntOff 10

- IntOn 10
- REC_INTERRUPT 11

coupleur série 10, 13
Create 20, 33
currentThread 15

D

défaut de page 12
Deallocate 20
DEBUG 30
DebugInit 30
DebugIsEnabled 30
decompname 21
directives 36
Directory 21
DirToMake 39
DirToRemove 39
dirty 24, 27
disque 12

E

EM_INTERRUPT 11
emission_finished 13
empty 21
Enable 10
Exec 16, 32
ExistTLB 37
Exit 32

F

FetchFrom 20, 21, 32
fichier de configuration 9, 37
fichiers

- ACIA.cc 10
- ACIA.h 10
- ACIA_handler.cc 13
- ACIA_handler.h 13
- addrspace.cc 16
- addrspace.h 16
- bitmap.cc 31
- bitmap.h 31
- config.cc 37
- config.h 37
- console.cc 12
- console.h 12
- directory.cc 21
- directory.h 21
- disk.cc 12
- disk.h 12
- exception.cc 18
- filehdr.cc 20
- filehdr.h 20
- filesys.cc 20
- filesys.h 20
- interrupt.cc 10
- interrupt.h 10
- libnachos.cc 34
- libnachos.h 34
- list.h 30
- machine.cc 9
- machine.h 9
- main.cc 36
- mem.cc 28
- mem.h 28
- mmu.cc 12, 26
- mmu.h 12, 26
- nachos.cfg 9
- oftable.cc 22
- oftable.h 22
- openfile.cc 22
- openfile.h 22
- pageFaultManager.cc 26
- pageFaultManager.h 26
- scheduler.cc 15
- scheduler.h 15
- start.s 18
- synch.cc 17
- synch.h 17
- synchcons.cc 14

- synchcons.h 14
- synchlist.cc 30
- synchlist.h 30
- syscall.h 18
- system.cc 36
- thread.cc 15
- thread.h 15
- tlb.cc 12, 24
- tlb.h 12, 24
- translate.cc 23
- translationtable.cc 25
- translationtable.h 25
- utility.cc 29
- utility.h 29, 30

file des prêts 15
FileHeader 20
FileLength 20
FileSystem 20
FileToCopy 39
FileToPrint 39
FileToRemove 39
Find 21, 32
FindDir 21
Findindex 21
findl 23
FindNextToRun 15
Finish 16
Flush 25
Format des fichiers exécutables 29
FormatDisk 38
free 27

G

getBitSwap 26
GetChar 11, 12
GetFileHeader 22
GetInputStateReg 11
getLevel 10
GetName 22
GetOutputStateReg 11
GetString 14
GetWorkingMode 11

H

Halt 32

I

ind_rec 13
ind_send 13
initKernelContext 16
interrupt_receive 14
interrupt_send 14
IntOff 10
IntOn 10
IntStatus 10
Invalid 25
io 26
isDir 20, 22
IsEmpty 31

J

Join 16, 32

K

kernelStackTop 15

L

Length 22
List 21
List() 31
ListDir 39
liste
- ~List() 31
- Append 31
- IsEmpty 31
- List() 31
- Mapcar 31
- Prepend 31
- Remove 31
- RemoveItem 31

- Search ... 31
- SortedInsert ... 31
- SortedRemove ... 31

listes ... 30
Lock ... 23
LockAcquire ... 33
LockCreate ... 33
LockDestroy ... 33
LockRelease ... 33

M

mémoire virtuelle ... 23
machine ... 23
machineState ... 15
Mapcar ... 31
Mark ... 31
memcmp ... 35
memcpy ... 35
Memory Management Unit ... 12, 25
memset ... 35
Mkdir ... 21, 33
MMU ... 12, 25, 26

N

newline ... 34
next_entry ... 23
NoException ... 24
NOFF ... 29
noff.h ... 29
noyau ... 15
NUM_TRACKS ... 12
NumClear ... 32
NumDirEntries ... 39
NumPhysPages ... 37
NumPortDist ... 38
NumPortLoc ... 38

O

Open ... 20, 23, 33
OpenFile ... 22
OpenFileTable ... 22
OpenFileTableEntry ... 22
Outils de synchronisation ... 17
owner ... 27

P

P ... 17, 33
pageDisk ... 25
PageFaultException ... 24
PageSize ... 38
PageTableEntry ... 25
PageTableSize ... 37
PError ... 32
physicalPage ... 24, 25
pilote ... 13
pilote du disque ... 14
pilotes de périphériques ... 13
pointeurs de fonctions ... 30
Prepend ... 31
Print ... 20, 21, 32
printf ... 14, 34
PrintFileSyst ... 39
ProcessorFrequency ... 37
ProgramToRun ... 38
PutChar ... 11, 12
PutString ... 14

R

répertoires
- bin ... 29
- filesys ... 8, 19
- handler ... 7, 13
- kernel ... 7, 9, 15, 36
- machine ... 9, 23, 24, 26
- start.s ... 32
- test ... 8, 32, 34
- utility ... 29, 37
- vm ... 8, 23, 25, 28

Read ... 14, 33
ReadAt ... 22

ReadCC 9
ReadFPRegister 9
ReadInt 34
ReadMem 23
readOnly 26
readonly 24
ReadOnlyException 24
ReadRegister 9
ReadRequest 12
ReadSector 14
readyList 15
ReadyToRun 15
RealMemManager 28
REC_INTERRUPT 11
receive_buffer 13
reception_finished 13
registres ACIA
    inputRegister 11
    inputStateRegister 11
    mode 11
    ouputStateRegister 11
    outputRegister 11
    registre de contrôle 11
    registres d'état 11
    registres de données 11
registres processeur
    BadVAddrReg 10
    NextPCReg 10
    NumFPRegs 10
    NumPage 10
    NumTotalRegs 10
    PCReg 10
    RetAddrReg 10
    StackReg 10
    Table 10
Release 17, 23
Remove 20, 21, 23, 31, 33
RemoveItem 31
RequestDone 15
RestoreUserState 16
Rmdir 21, 33
routine de traitement d'interruption .. 14
routines de déboguage 29
Run 15, 16
Run() 9

S

Sémaphore 17
SaveUserState 16
Scheduler 15
Search 31
SECTOR_SIZE 12
SECTORS_PER_TRACKS 12
SectorSize 38
SemCreate 33
SemDestroy 33
send_buffer 13
set 26
SetDir 20
SetFile 20
SetLevel 10
SetName 22
setPageDisk 26
SetWorkingMode 11
Signal 18
Sleep 16
SortedInsert 31
SortedRemove 31
StartThreadExecution 17
statistiques 9
strcat 35
strcmp 34
strcpy 35
strlen 35
structure des répertoires 7
swap 26
SWITCH 17
synchcons 14

SynchConsole 14
synchdisk.cc 14
synchdisk.h 14
syscall 18
système de gestion des fichiers 19
system 27
SysTime 32

T

Table des Pages Réelles 25
Table des pages virtuelles 25
TableType 37
TargetMachineName 38
Test 32
test d'état/attente active 11
Thread 15
thread élu 15
TimeSharing 37
TLB 12, 24
TLBSize 37
tolower 35
toupper 35
TPR 25
traduction d'adresses 23
Translate 12, 24
Translation Look-aside Buffer 12, 24
TranslationTable 25
tty_receive 13
tty_send 13
TtyReceive 34
TtySend 34
type
  IntStatus 10
  VoidFunctionPtr 30
  VoidNoArgFunctionPtr 30

U

unité de gestion de mémoire 26
Use 25
use 27
useACIA 38
userRegisters 15

V

V 17, 33
valid 24, 26
Variable de condition 17
Verrou 17
virtualPage 24, 27
void* newThread(char *name
  VoidFunctionPtr func
    PV 16
VoidFunctionPtr 30
VoidNoArgFunctionPtr 30
Voleur de pages 28

W

Wait 18
WantOldest 25
Write 14, 33
WriteAt 22
WriteBack 20, 21, 32
WriteFPRegister 9
WriteInt 34
writeln 34
WriteMem 23
WriteRegister 9
WriteRequest 12
WriteSector 15
writestr 14, 34

Y

Yield 16, 32