Noyau d'un système d'exploitation INF2610

Chapitre 2 : Processus

Département de génie informatique et génie logiciel

Hiver 2019

Chapitre 2 - Processus

- **Qu’est ce qu’un processus ?**
- **États d’un processus**
- **Hiérarchie des processus**
- **Processus UNIX-Linux**
  - **Création de processus**
  - **Remplacement d’espace d’adressage**
  - **Attente de la fin d’un processus fils**
  - **Terminaison de processus**
  - **Partage de fichiers entre processus**

Qu’est ce qu’un processus ?

- Le concept processus est le plus important dans un système d’exploitation. Tout le logiciel d’un ordinateur est organisé en un certain nombre de processus (système et utilisateur).

- Un processus est un programme en cours d’exécution. Il est caractérisé par (son contexte) :
  - un numéro d’identification unique (PID),
  - un espace d’adressage (code, données, piles d’exécution),
  - les fichiers ouverts, les espaces mémoire alloués, les périphériques,
  - les signaux à capter, à masquer, à ignorer, en attente et les actions associées,
  - le processus père, les processus fils, le groupe, les variables d’environnement, les statistiques, les limites d’utilisation des ressources,
  - **un état principal (prêt, en cours d’exécution, bloqué, etc.),**
  - **une priorité,**
  - **les valeurs des registres lors de la dernière suspension (CO, PSW, Sommet de Pile, etc.) ➔ contexte d’exécution,**
  - **etc.**

Qu’est ce qu’un processus ? (2)
Table des processus

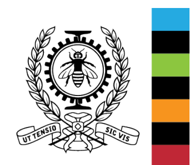
- Le système d’exploitation maintient dans une table appelée «table des processus» les informations sur tous les processus créés (une entrée par processus : Bloc de Contrôle de Processus PCB).

Un PCB est une structure de données qui représente un processus au niveau du système d’exploitation

Table des processus

PID	PCB
1	
2	
…	
n	

PCB du processus n	PCB du processus 2	PCB du processus 1
Compteur ordinal	Compteur ordinal	Compteur ordinal
Registres	Registres	Registres
État	État	État
Priorité	Priorité	Priorité
Espace d’adressage	Espace d’adressage	Espace d’adressage
Père	Père	Père
Fils	Fils	Fils
Fichiers ouverts	Fichiers ouverts	Fichiers ouverts
Actions associées aux signaux	Actions associées aux signaux	Actions associées aux signaux
….	….	….



Tannenbaum

Qu'est ce qu'un processus ? (3)
Table des processus : Changement de contexte

Processus A
Mode utilisateur

Mode kernel

Processus B
Mode utilisateur

exécution

interruption → **Sauvegarder état dans PCB A**

Charger état du PCB B → exécution

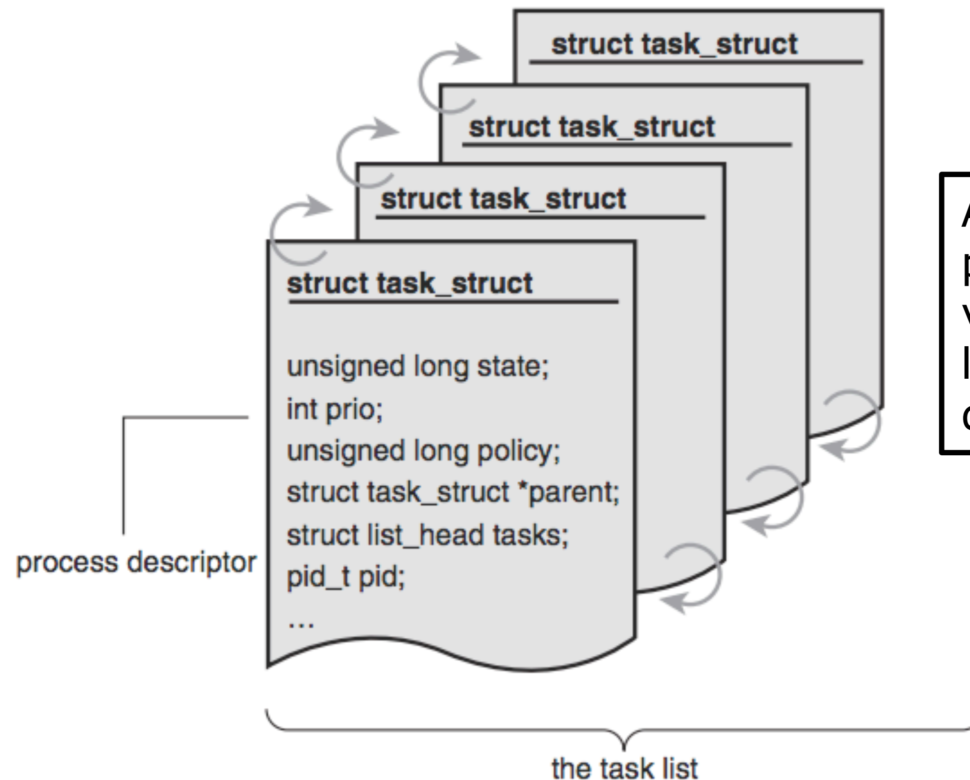
Sauvegarder état dans PCB B ← interruption

exécution ← **Charger état du PCB A**

Tannenbaum

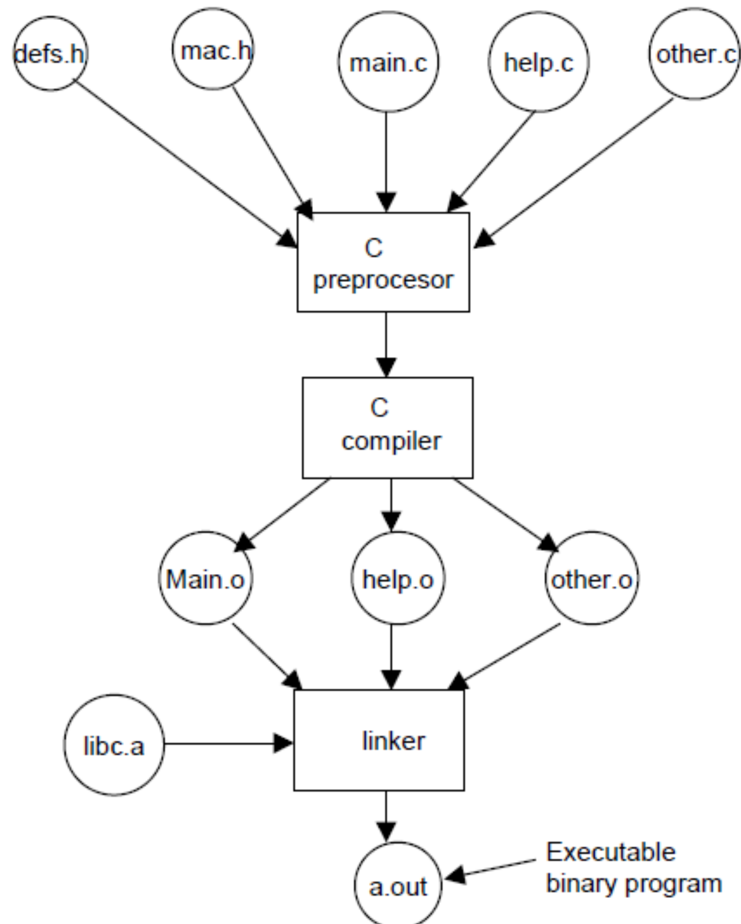
Qu’est ce qu’un processus ? (4)
Table des processus

- Dans le cas de Linux, le PCB est une structure *task_struct* et la table des processus est une liste doublement chainée.



Accès (direct/indirect) à partir d’un processeur, via un registre, au PCB de la tâche en cours d’exécution.

Qu’est ce qu’un processus ? (5)
Espace d’adressage virtuel



http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

Fichier exécutable ELF (Executable and Linkable Format)

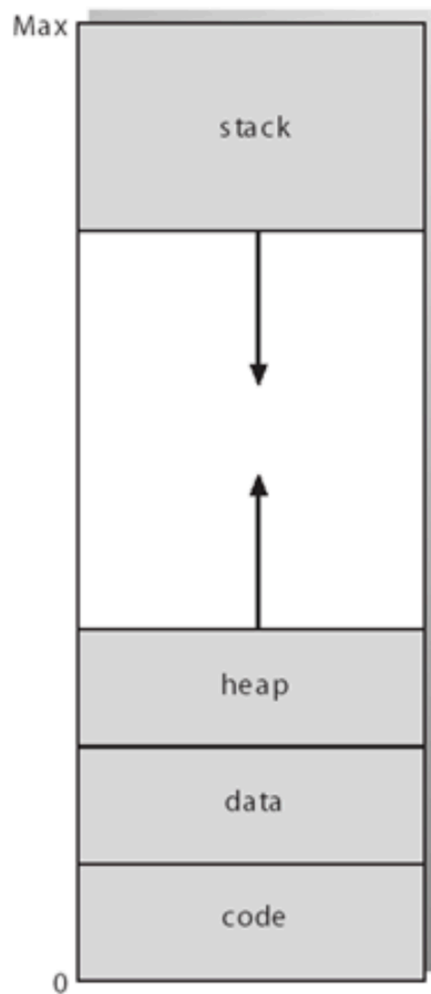
File Offset	File	Virtual Address
0	ELF header	
Program header table		
	Other information	
0x100	Text segment	0x8048100
	...	
	0x2be00 bytes	0x8073eff
0x2bf00	Data segment	0x8074f00
	...	
	0x4e00 bytes	0x8079cff
0x30d00	Other information	
	...	

Program header table

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

Le fichier exécutable permet au noyau de construire un processus (table des pages de son espace d’adressage, adresse de la première instruction à exécuter, etc.).

Qu'est ce qu'un processus ? (6)
Espace d'adressage virtuel



cat /proc/"pid"/maps
man 5 proc

Virtual address space

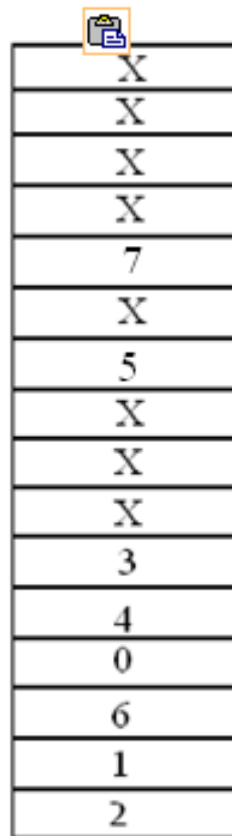
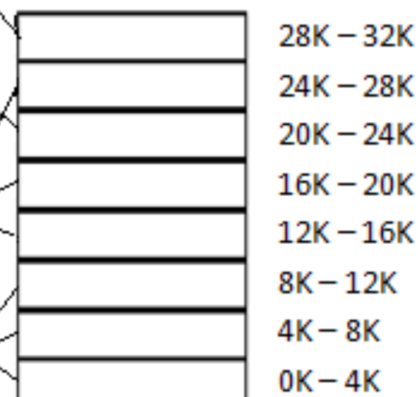


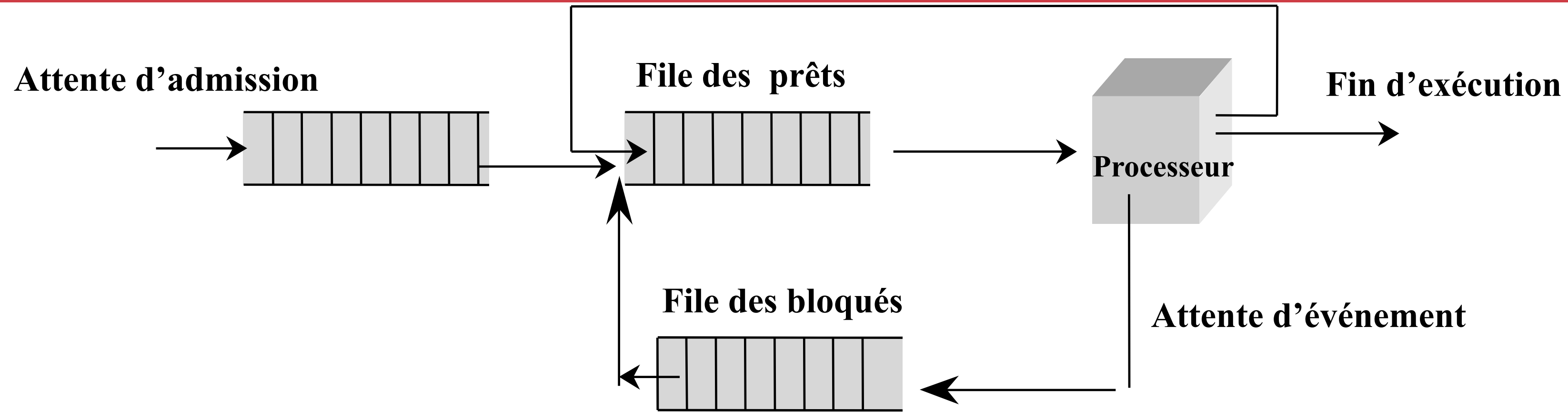
Table des pages : indique pour chaque page de l'espace d'adressage d'un processus, son emplacement en mémoire ou sur le disque, son code de protection, etc.

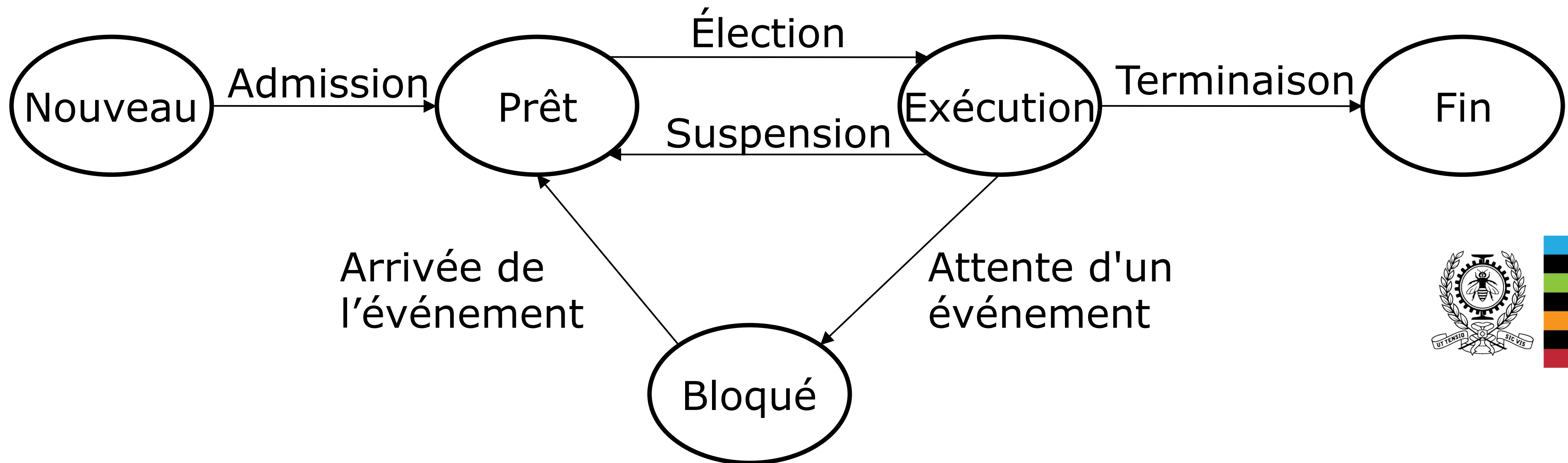
Physical memory space



Tannenbaum

États d’un processus

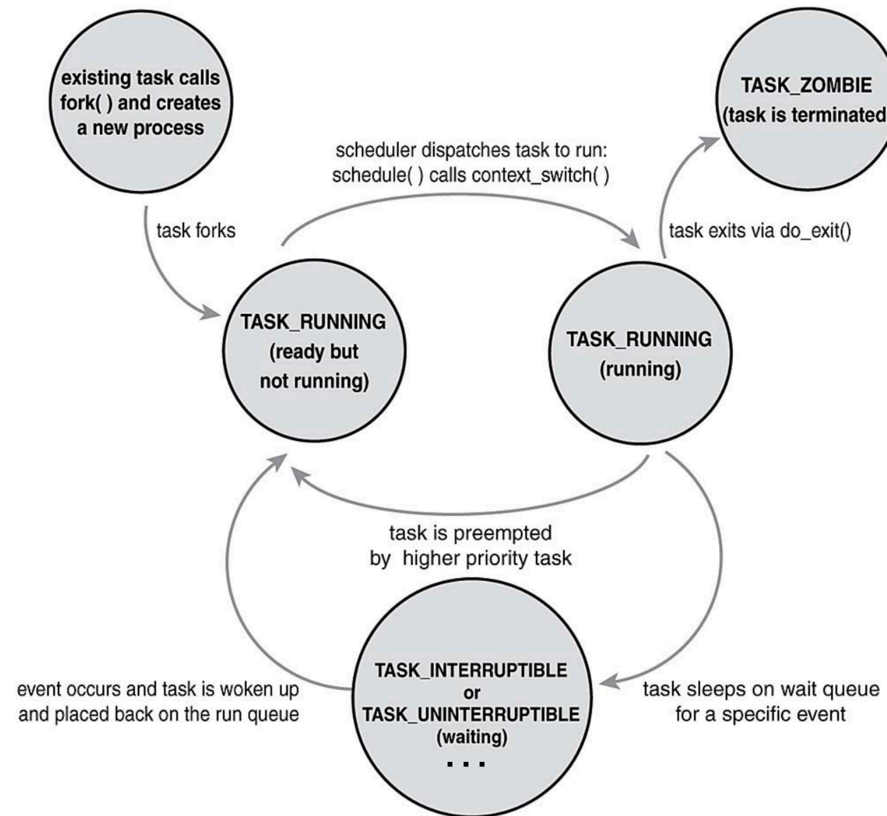




États d’un processus (2) : Unix

fork
Création
Préemption
Retour
à usager
Mémoire
suffisante
Mémoire
insuffisante
Exécution
Mode
user
suspension
Hors mémoire
Retour
Prêt en
mémoire
Prêt hors
mémoire
Élection
En mémoire
Interruption ou
Appel système
Exécution
Mode
noyau
Mise en attente
d’un evt
Arrivée de l’evt
Arrivée de l’evt
terminaison
Zombie
attente
en
mémoire
Hors mémoire
Attente
hors
mémoire

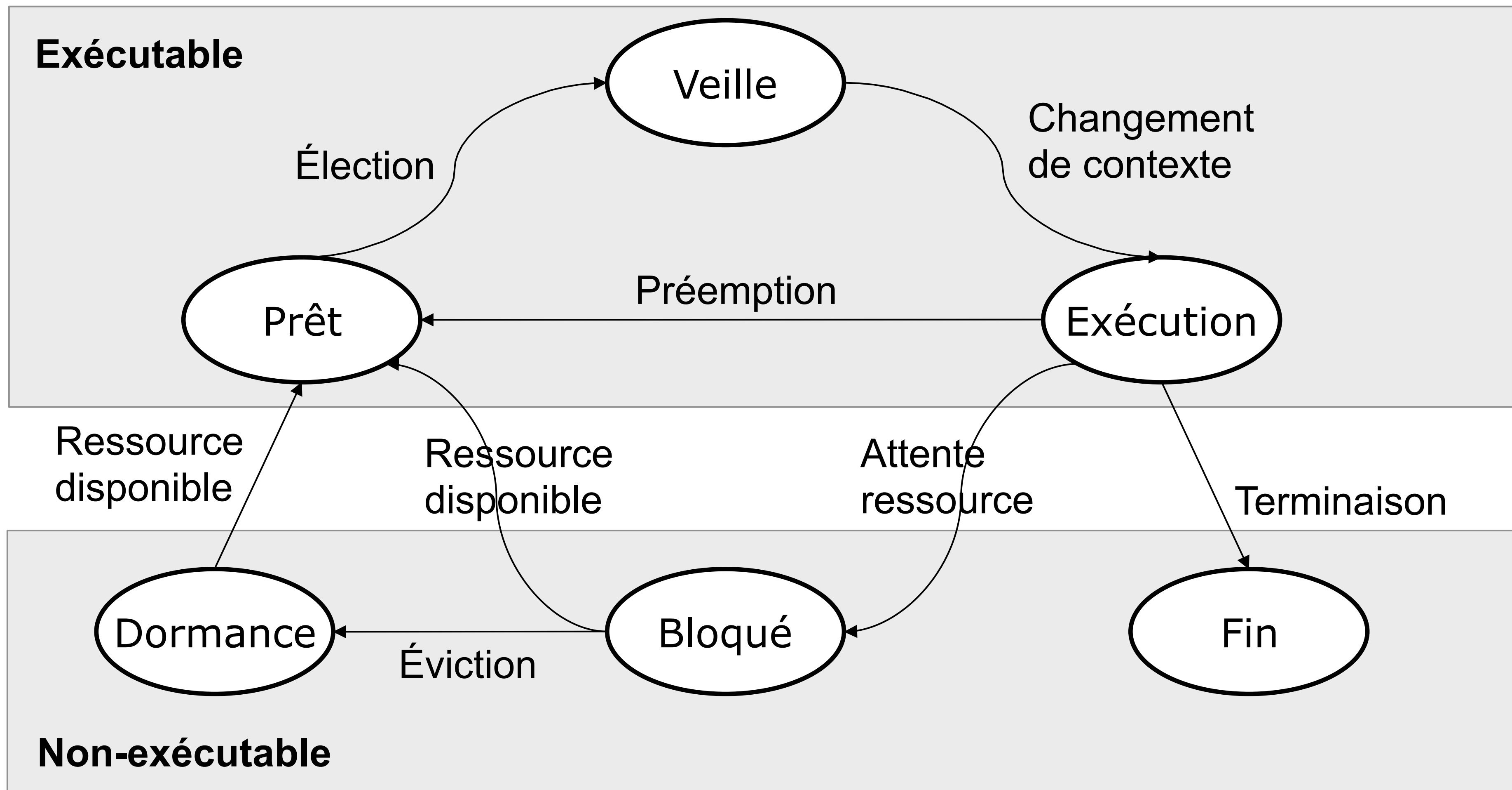
États d’un processus (3) : Linux



http://www.informit.com/articles/article.aspx?p=368650

États : R → Task_Running,
S (sleeping) → Task_Interruptible,
D → Task_Uninterruptible,
T (being traced or stopped) → Task_Stopped,
Z → Task_Zombie.

États d’un processus (4) : Windows



Hiérarchie des processus

- Le système d'exploitation fournit un ensemble d'appels système qui permettent la création, la destruction, la communication et la synchronisation des processus.

- Les processus sont créés et détruits dynamiquement.

- Un processus peut créer un ou plusieurs processus qui, à leur tour, peuvent en créer d'autres.

- Dans certains systèmes (MS-DOS), lorsqu'un processus crée un autre processus, l'exécution du processus créateur est suspendue jusqu'à la terminaison du processus créé (exécution séquentielle).

- Sous Linux/Unix/Windows, les processus créateurs et créés s'exécutent en concurrence et sont de même niveau (exécution asynchrone). Chaque processus a un parent et éventuellement un ou plusieurs fils.

Hiérarchie des processus (2) :

- Windows ne gère pas de relation hiérarchique entre processus :
  - Chaque processus a un pointeur vers le processus parent (processus créateur).
  - Si ce parent se termine, il n'y a pas d'actualisation de cette information. Un processus peut pointer vers un parent inexistant. Cela ne cause aucun problème puisque rien ne dépend de cette information de parenté.

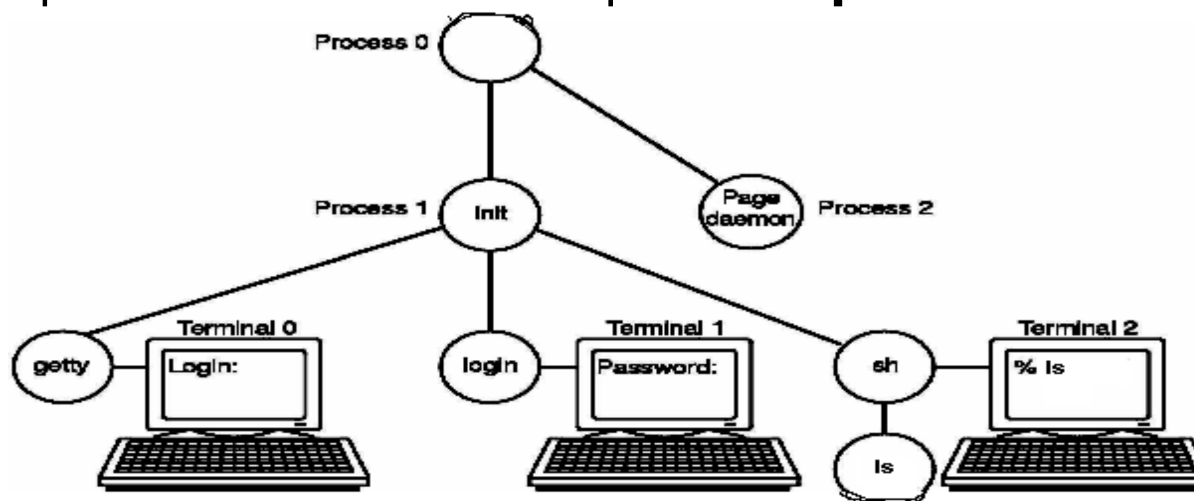
- Linux/Unix gère dynamiquement la relation hiérarchique entre processus :
  - Chaque processus a un pointeur vers le processus parent (processus créateur).
  - Si ce parent se termine, il est adopté par un autre processus vivant (**processus init de numéro 1**).

Hiérarchie des processus (3) : Linux/UNIX

- BIOS (Basic Input/Output System dans la carte mère) exécute des opérations de vérification de l'intégrité du système, charge et lance l'exécution du programme d'amorçage MBR (Master Boot Record).

- MBR charge une partie du système d'exploitation puis lance son exécution. Cette partie détermine les caractéristiques du matériel, effectue un certain nombre d'initialisations et crée le processus 0.

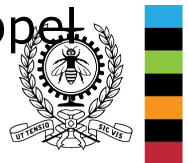
- Le processus 0 réalise d'autres initialisations (ex. le système de fichier) puis crée deux processus : **init** de PID 1 et **démon des pages** de PID 2. Ensuite, d'autres processus sont crées à partir du **processus init**.



http://www.ibm.com/developerworks/linux/library/l-linuxboot/

Processus UNIX – Linux

- Chaque processus a un numéro d'identification unique (PID). L'appel système **getpid()** permet de récupérer le PID du processus.
- Chaque processus a un père (structure arborescente). Le processus init est l'ancêtre de tous les processus utilisateur. L'appel système **getppid()** permet de récupérer le PID de son processus père.
- La création de processus est réalisée par duplication de l'espace d'adressage et de certaines tables du processus créateur (l'appel système **fork**).
- La duplication facilite la création et le partage de ressources. Le fils hérite les résultats des traitements déjà réalisés par le père.
- Un processus peut remplacer son code exécutable par un autre (appel système **exec**).



- Un processus père peut stopper/repartir/détruire (appel système **kill**) ou attendre la terminaison (appel système **wait**) de ses fils mais ne peut pas les renier.

Création de processus : fork

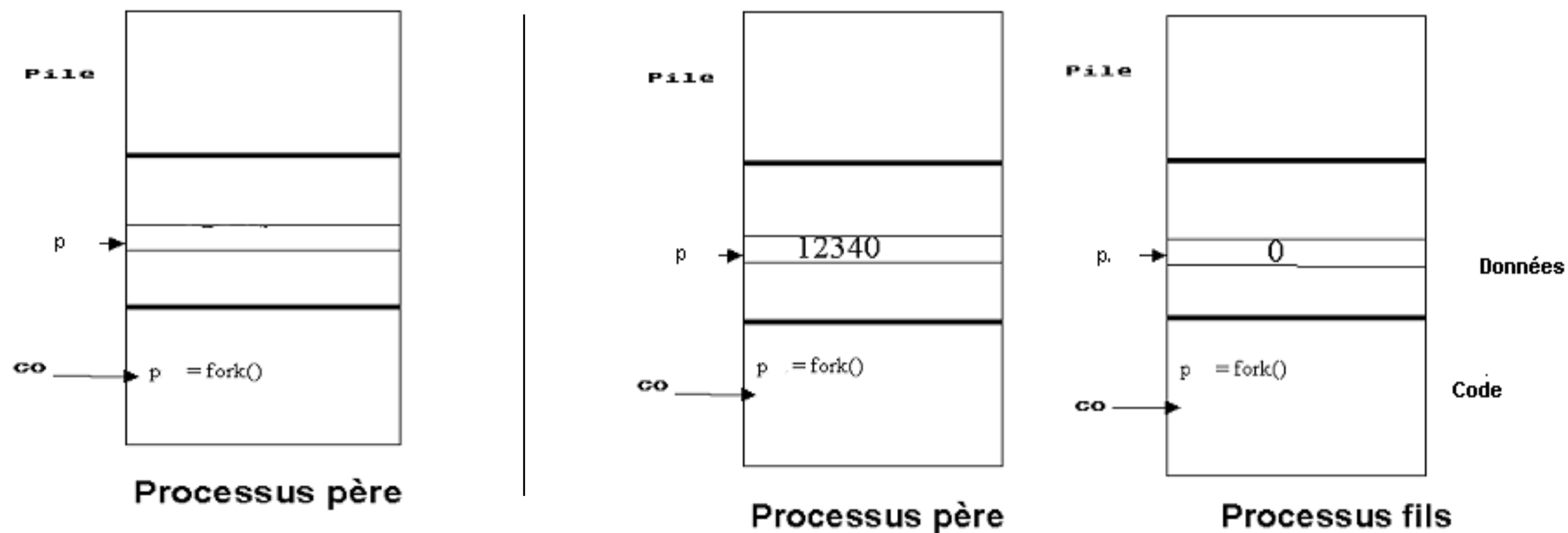
- L’appel système fork :
  - associe un numéro d’identification unique (le PID du processus);
  - ajoute puis initialise une entrée dans la table des processus (PCB). Certaines entités comme le répertoire de travail courant, la valeur d’umask, les limites des ressources sont copiées du processus parent, etc;
  - duplique l’espace d’adressage du processus effectuant l’appel à fork (code+pile+données) → Principe « Copy-on-write » partage en lecture, duplication en cas d’écriture.
  - duplique la table des descripteurs de fichiers….

- La valeur de retour est :
  - 0 pour le processus créé (fils).
  - le PID du processus fils pour le processus créateur (père).
  - négative si la création de processus a échoué (manque d’espace mémoire ou le nombre maximal de créations autorisées est atteint).

Création de processus (2) : fork

- Au retour de la fonction fork, l’exécution des processus père et fils se poursuit, en temps partagé, à partir de l’instruction qui suit l’appel à fork.

- Le père et le fils ont chacun un espace d’adressage privé.



Exemple 1 : Création d’un processus fils (chaque processus a son espace d’adressage (privé))

Création de processus (3) : Exemple 1

```
1   // programme tfork.c : appel système fork()
2   #include <sys/types.h>  /* typedef pid_t */
3   #include <unistd.h>                    /* fork() */
4   #include <stdio.h>                     /* pour perror, printf */
5   int a=20;
6   int main( ) {
7       pid_t x;
8       // création d'un fils
9       switch (x = fork()) {
10      case -1: /* le fork a échoué */
11          perror("le fork a échoué !");
12          break;
13      case 0:   /* seul le processus fils exécute  ce « case »*/
14          printf("ici processus fils, le PID %d.\n ", getpid());
15          a += 10;
16          break;
17      default:  /* seul le processus père exécute ce « case »*/
18          printf("ici processus père, le PID %d.\n", getpid());
19          a += 100;
20      }
21      // les deux processus exécutent ce qui suit :
22      printf("Fin du Process %d. avec a = %d\n", getpid(), a);
23      return 0;
24  }
```

Création de processus (4) : Exemple 1

```
jupiter% gcc -o tfork tfork.c
jupiter% ./tfork
ici processus père, le PID 12339.
ici processus fils, le PID 12340.
Fin du Process 12340 avec a = 30.
Fin du Process 12339 avec a = 120.
```

a du fils

a du père

```
jupiter% ./tfork
ici processus père, le PID 15301.
Fin du Process 15301 avec a = 120.
ici processus fils, le PID 15302.
Fin du Process 15302 avec a = 30.
jupiter%
```

a du père

a du fils

Exemple 2 : Création de plusieurs processus

- strace permet de construire l'arbre des processus créés.
- Orphelins adoptés par le processus init de pid 1.

Création de processus (5) : Exemple 2

```
1  // programme tree.c : appel système fork()
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdio.h>
5  int main( ) {
6      pid_t p;
7      int i, n=5;
8      for (i=1; i<n; i++) {
9          p = fork();
10         if (p > 0)
11             break ;
12         sleep(1);
13         printf(" Processus %d  de père %d. \n",  getpid(), getppid());
14 }
15     sleep(2);
16     return 0;
17 }
```

Création de processus (6) : Exemple 2

shell

Processus 23706

i=1
fork()
sleep(2)
return 0

Processus 23707

sleep(1)
printf(..)
i=2
fork()
sleep(2)
return 0

Processus 23708

sleep(1)
printf(..)
i=3
fork()
sleep(2)
return 0

Processus 23709

sleep(1)
printf(..)
i=4
fork()
sleep(2)
return 0

Processus 23710

sleep(1)
printf(..)
i=5
sleep(2)
return 0

Comment utiliser strace pour déterminer l'arborescence des processus créés ?

```
jupiter% gcc –o  tree  tree.c
jupiter% ./tree
Processus 23707  de pere 23706
Processus 23708 de pere 23707
Processus 23709 de pere 23708
Processus 23710 de pere 23709
```

Création de processus (7) : Exemple 2

jupiter$ strace -f -e trace=clone -o stree.txt ./tree
 Processus 26770 de père 26769.
 Processus 26771 de père 26770.
 Processus 26773 de père 26771.
 Processus 26774 de père 26773.
jupiter$ cat stree.txt
26769 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffb6cbbc790) = **26770**
26769 +++ exited with 0 +++
26770 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffb6cbbc790) = **26771**
26770 +++ exited with 0 +++
26771 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID
|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffb6cbbc790) = **26773**
26771 +++ exited with 0 +++
26773 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffb6cbbc790) = **26774**
26773 +++ exited with 0 +++
26774 +++ exited with 0 +++

Comment utiliser strace pour déterminer l'arborescence des processus ?

Création de processus (8) : Exemple 2 sans «sleep(2)»

shell

Processus 23706

i=1
fork()

terminaison

Processus 23707

sleep(1)
printf(..)
i=2
fork()

terminaison

Processus 23708

sleep(1)
printf(..)
i=3
fork()

terminaison

Processus 23709

sleep(1)
printf(..)
i=4
fork()

terminaison

Processus 23710

sleep(1)
printf(..)
i=5

terminaison

```
jupiter% gcc –o  tree  tree.c
jupiter% ./tree
jupiter%  Processus 23707  de pere 1
Processus 23708 de pere 1
Processus 23709 de pere 1
Processus 23710 de pere 1
```

Les fils qui deviennent orphelins sont adoptés par le processus **init** de PID 1

Terminaison de processus

- Un processus se termine par une demande d'arrêt volontaire (**exit**) ou par un arrêt forcé provoqué par un autre processus (appel système **kill**) ou une erreur.

```
void exit(int vstatus);
```

- Lorsqu'un processus fils se termine :
  - son état de terminaison est enregistré dans son PCB,
  - la plupart des autres ressources allouées au processus sont libérées,
  - le processus passe à l'état zombie (<defunct>).

- Son PCB et son PID sont conservés jusqu'à ce que son processus père ait récupéré cet état de terminaison. Il est alors détruit.

- Les appels système wait(&status) et waitpid(pid, &status, option) permettent au processus père de récupérer, dans status, cet état de terminaison.

Attente et terminaison de processus

- Un processus peut attendre ou vérifier la terminaison d'un de ses fils :
  - pid_t wait (int * pstatus); // attendre un fils
  - pid_t waitpid(int pid, int*pstatus, int options); //attendre le fils spécifié.

- wait(&status) et waitpid(pid, &status, options) retournent :
  - le PID du fils qui s'est terminé,
  - -1 en cas d'erreur (le processus n'a pas de fils), et
  - dans **status** des informations sur l'état de terminaison. Ces informations peuvent être récupérées au moyen de macros telles que :
    - WIFEXITED(status) : fin normale avec exit,
    - WIFSIGNALED(status) : tué par un signal,
    - WIFSTOPPED(status) : stoppé temporairement,
    - WEXITSTATUS(status) : valeur de retour du processus fils (exit(valeur)).

- waitpid(pid, &status, WNOHANG) vérifie seulement la terminaison pas d'attente. Il retourne 0 si l'exécution du processus n'est pas terminée.

Exemple 3 : État de terminaison d’un processus (rôles de wait et exit)

Exemple 3 : wait & exit

```
// programme deuxfils.c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
void fils(int i);
int main() {
    int status;
    if (fork()==0) { // premier fils
            fils(1);
    } else if (fork()==0) { // second  fils
                    fils(2);
    } else  { if (wait(&status) > 0)
                      printf("fin du fils%d\n", WEXITSTATUS(status));
                  if (wait(&status) > 0)
                      printf("fin du fils%d\n", WEXITSTATUS(status));
        }
    return 0;
}
void fils(int i) {
    sleep(2);
    _exit(i);
}
```

```
jupiter% gcc -o deuxfils deuxfils.c
jupiter% deuxfils
fin du fils1
fin du fils2
jupiter% deuxfils
fin du fils2
fin du fils1
```

Remplacement de l'espace d'adressage

- Le système UNIX/Linux offre une famille d'appels système **exec** qui permettent à un processus de remplacer son code exécutable par un autre spécifié par **path** ou **file** :

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ..., /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ..., /* (char  *) NULL */);
int execle(const char *path, const char *arg, ..., /*, (char *) NULL, char * const envp[]*/);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],  char *const envp[]);
```

- Après un exec, le processus conserve, notamment, son PID, l'espace mémoire alloué, sa table de descripteurs de fichiers et ses liens parentaux (processus fils et père).
- En cas de succès de l'appel système exec, l'exécution de l'ancien code est abandonnée au profit du nouveau.

- En cas d'échec, le processus poursuit l'exécution de son code à partir de l'instruction qui suit l'appel (il n'y a pas eu de remplacement de code).

Exemple 4 : shell simplifié
(faire exécuter une commande par son processus fils)

Exemple 4 : shell simplifié (execvp)

```
// programme test de execvp : texecvp.c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main (int argc , char * argv[]) {
    if (fork() == 0) { // il s'agit du fils
        // exécute un autre programme
        execvp(argv[1], &argv[1]) ;
        fprintf(stderr, "invalide %s\n ",  argv[1]);
        _exit(1);
    }
    int status;
   if(wait(&status) > 0)
        printf("fin du fils avec %d\n",
                    WEXITSTATUS(status));
    return 0;
}
```

```
jupiter% gcc –o texecvp
texecvp.c
jupiter% ./texecvp date
Mon Sep  3 19:17:00 EDT 2018
fin du fils avec 0


jupiter% ./texecvp ugg+kjù
invalide ugg+kjù
 fin du fils avec 1

jupiter% ./texecvp ./deuxfils
fin du fils1
fin du fils2
fin du fils avec 0
```

Exemple 5 : execv et execl (passage de paramètres)

Exemple 5 : execv & execl

```
// programme forkExec.c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // pour exit
int main( ) {
   if (fork()==0) { // premier fils
          char* args[]={"ls","-l", NULL};
         execv("/bin/ls",args);
          printf("echec de execv de ls \n");
         exit(1);
   } else if (fork()==0) { // second  fils
            execl("/bin/pwd","pwd",NULL);
            printf("echec de execl de pwd \n");
            exit(1);
      } else  { int pid, status;
         pid=wait(&status);
         printf("fin du fils %d avec %d\n", pid, WEXITSTATUS(status));
         pid=wait(&status);
         printf("fin du fils %d axec %d\n", pid, WEXITSTATUS(status));
      }
   return 0;
}
```

```
jupiter$ gcc forkExec.c –o ./forkExec
jupiter$ ./forkExec
/usagers4/p302161/codesINF2610/codesExecINF2610
fin du fils 26168 avec 0
total 36
-rwx------ 1 p302161 gigl 8496 Sep  3 15:48 deuxfils
-rwx------ 1 p302161 gigl 8568 Sep  3 17:34 forkExec
-rwx------ 1 p302161 gigl 8472 Sep  3 15:49 tree
fin du fils 26167 axec 0
```

```
jupiter$ pwd
/usagers4/p302161/codesINF2610/codesExecINF2610
jupiter$ ls -l
total 36
-rwx------ 1 p302161 gigl 8496 Sep  3 15:48 deuxfils
-rwx------ 1 p302161 gigl 8568 Sep  3 17:34 forkExec
-rwx------ 1 p302161 gigl 8472 Sep  3 15:49 tree
```

Exemple 6 : execve
(passage de paramètres)

Exemple 6 : execve

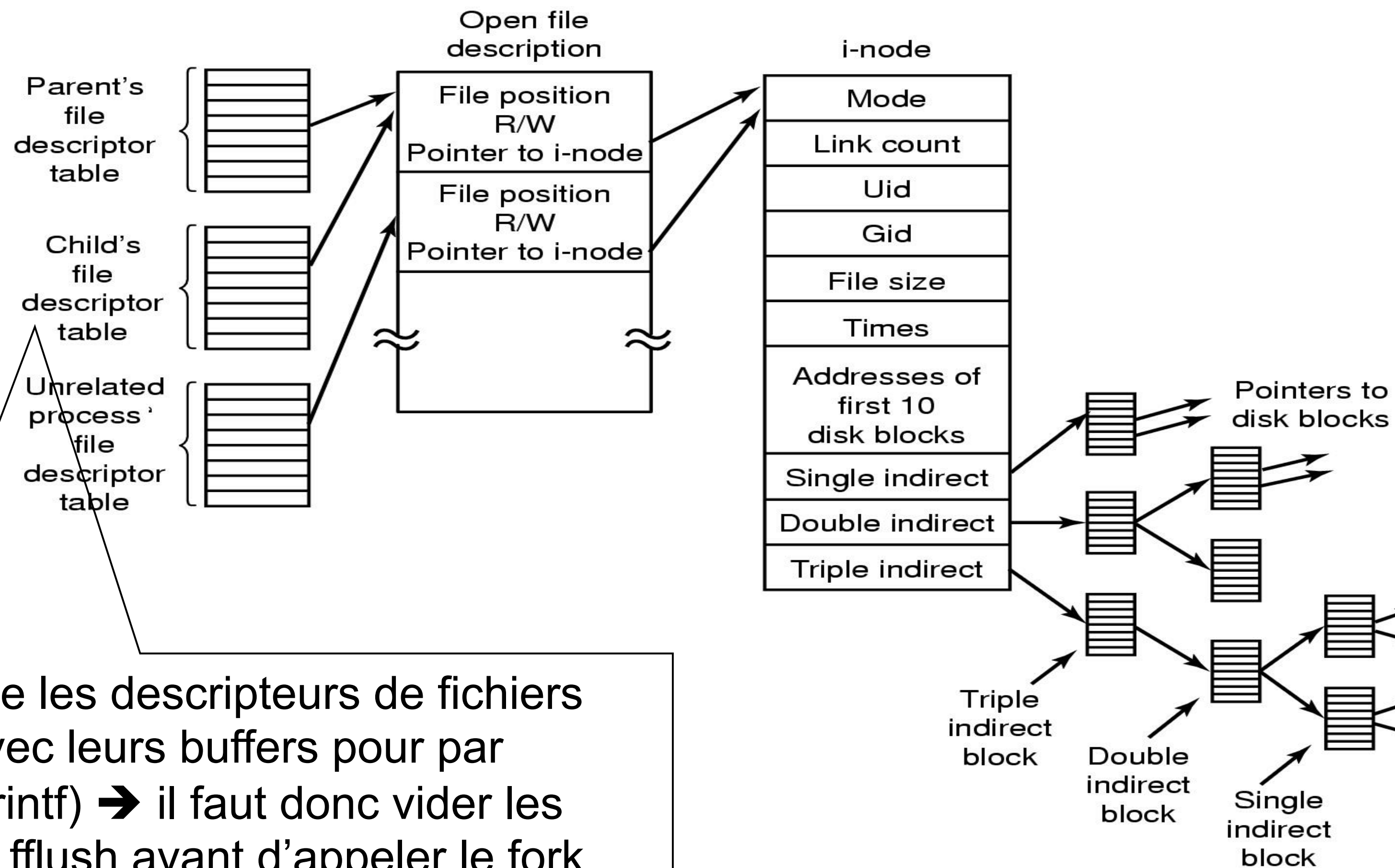
```c
// programme forkExecve.c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // pour exit
int main( ) {
    if (fork()==0) { // fils
          char* args[]={"fils", NULL};
          char* env[]={"PWD=../",NULL};
          execve("./fils", args, env);
          printf("echec de execve \n");
           exit(1);
      } // père
    int pid, status;
    pid=wait(&status);
    printf("fin du fils %d avec %d\n", pid, WEXITSTATUS(status));
    return 0;
}
```

```c
// programme fils.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // pour exit
int main(int argc, char* argv[], char*env[]) {
      printf(" argc = %d \n", argc);
      int i;
      for(i=0; i<argc;i++)
            printf(" argv[%d]=%s \n", i, argv[i]);
      for(i=0;env[i]!=NULL;i++)
             printf(" env[%d]=%s \n", i, env[i]);
        printf(" getenv(PWD)=%s\n" , getenv("PWD"));
        exit(0);
}
```

```
jupiter$ gcc forkExecve.c –o forkExecve
jupiter$ gcc fils.c  -o fils
jupiter$ ./forkExecve
 argc = 1
 argv[0]=fils
 env[0]=PWD=../
 getenv(PWD)=../
fin du fils 9276 avec 0
```

Partage de fichiers entre processus père et fils

- Le fork duplique la table des descripteurs de fichiers du processus père.



Le fils hérite les descripteurs de fichiers ouverts (avec leurs buffers pour par exemple printf) ➔ il faut donc vider les buffers par fflush avant d'appeler le fork.

http://processors.wiki.ti.com/index.php/Tips_for_using_printf#Buffered_I.2FO

Exemple 7 : printf avant fork

```
#include <unistd.h>            /* pour write et fork */
#include <stdio.h>             /* pour printf */
int main( ) {

    printf(" ici 1er printf de %d ", getpid());
    write(1," ici 1er write  ",16);
    printf(" ici 2eme printf de  %d ", getpid());
    fork();
    write(1," ici 2eme write  ", 17);
    printf("end of line printf de %d\n", getpid());
    write(1, " ici 3eme write \n",17);
    return 0;
}
```

```
jupiter%./printfwrite
 ici 1er write   ici 2eme write   ici 1er printf de 11243  ici 2eme printf de  11243 end of line printf de
11243
 ici 3eme write
ici 2eme write   ici 1er printf de 11243  ici 2eme printf de  11243 end of line printf de 11244
 ici 3eme write
jupiter%
```

Exemple 7' : printf avant fork (2)

```
#include <unistd.h>            /* pour write */
#include <stdio.h>             /* pour printf  et fflush*/
int main( ) {

    printf(" ici 1er printf de %d ", getpid());
    write(1," ici 1er write  ",16);
    printf(" ici 2eme printf de  %d ", getpid());
    fflush(stdout);
    fork();
    write(1," ici 2eme write  ", 17);
    printf("end of line printf de %d\n", getpid());
    write(1, " ici 3eme write \n",17);
    return 0;
}
```

Que fait fclose(stdout) ?

```
jupiter%./printfwrite
 ici 1er write   ici 1er printf de 11258  ici 2eme printf de  11258  ici 2eme write  end of line printf de 11258
 ici 3eme write
 ici 2eme write  end of line printf de 11259
 ici 3eme write
jupiter%
```

Exercice 1

Donnez l'arborescence de processus créés par ce programme ainsi que l'ordre de l'affichage des messages.

```
// chaine_fork_wait.c
int main()
{   int i, n=2;
        pid_t fils_pid;
        for(i=0; i<n;i++)
        {   fils_pid = fork();
              if (fils_pid > 0)
              {  wait(NULL);
                     break;
              }
        }
        printf("Processus %d de pere %d\n", getpid(), getppid());
        return 0;
}
```

Exercice 2

Donnez, sous forme d'un arbre, les différents ordres possibles d'affichage de messages (chaque chemin de l'arbre correspond à un ordre possible).

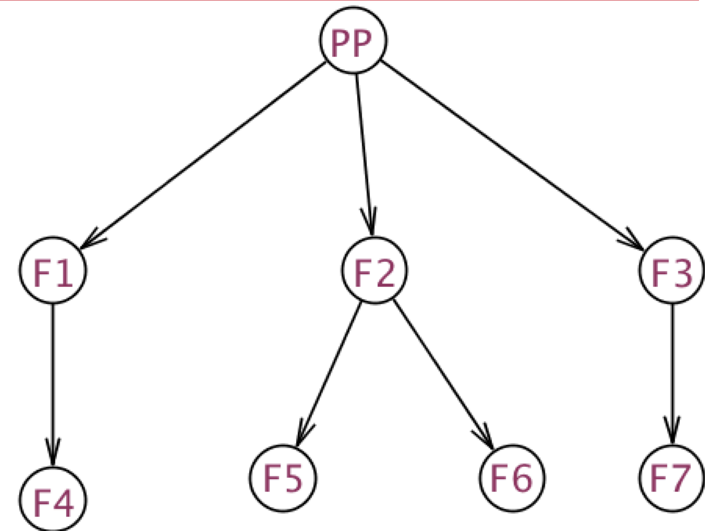
```
int main()
{   printf(“message0\n”);
      if (fork())
      {   printf(“message1\n”);
            if (fork())
                  printf(“message2\n”);
            else exit(0);
      } else printf(“message3\n”);
      return 0;
}
```

Ajouter une ligne de code pour forcer l'ordre d'affichage suivant :
message0; message3; message1;message2

Exercice 3

1- Donnez un code qui crée l'arbre de processus suivant :

- PP est le processus principal (qui exécute le code).
- Après avoir créé ses éventuels fils, chaque processus affiche à l'écran son nom et son pid puis se termine.



2 – Complétez le code pour que chaque père attende la fin de ses fils avant de se terminer.

3 – Complétez le code pour que les processus feuilles F4, F5, F6 et F7 se transforment respectivement en : date, ls, ps et cat fich.txt. La transformation de chaque processus feuille est lancée juste après l'affichage de son nom et son pid.

Passage de paramètres à un programme

Annexe

int main(int argc, const char *argv[]);

int main(int argc, const char *argv[], const char *envp[]);

Via une ligne de commande :

- argc: nombre de paramètres sur la ligne de commande (y compris le nom de l'exécutable lui-même)
- argv: tableau de chaînes de caractères contenant les paramètres de la ligne de commande.
- envp: tableau de chaînes de caractères contenant les variables d'environnement au moment de l'appel, sous la forme variable=valeur

Passage de paramètres à un programme (2)

- **int main(int argc, const char*argv[])**

Via « execv » ou « execp »

- **int execv (const char* path, const char* com[])**

    argv ← com

- **int execl (const char* path, const char* com0, const char* com1,…,)**

    argv [0] ← com0 , argv [1] ← com1, ….,

Exemple 8 : Terminaison de processus

```
// programme exec_wait.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
int main() {
    pid_t p = fork();
    if (p != 0) {
        execlp("./wait_child", "./wait_child", NULL);
        printf("execlp a échoué\n");
        exit(-1);
    } else {
        sleep(5);
        printf("Je suis le fils\n");
        exit(0);
    }
}
```

Après le fork, le père remplace son espace d'adressage par celui du programme wait_child.c

Les liens père /fils sont maintenus

```
// programme wait_child.c
#include <unistd.h>
#include <stdio.h>
#include <stlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    printf("J'attends le fils\n");
    wait(NULL);
    printf("Le fils a terminé \n");
    exit(0);
}
```

```
jupiter$ ./exec_wait
J'attends le fils…
Je suis le fils
Le fils a terminé
```

Exemple 9 : Attente de la fin d’un processus fils

```
// programme parent.c
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
int main( ) {
  int p, child, status;
  p = fork();
  if (p == -1)
    return -1;
  if(p > 0) { /* parent */
    printf ("père[%d], fils[%d]\n", getpid(), p);
    if ((child=wait(NULL)) > 0)
        printf("père[%d], Fin du fils[%d]\n", getpid(), child);
  printf("Le père[%d] se termine \n", getpid());
  } else { /* enfant */
    if ((status=execl("/home/user/a.out", "a.out", NULL)) == -1)
      printf("le programme n'existe pas : %d\n",status);
    else
      printf("cette instruction n’est jamais exécutée\n");
  }
  return 0;
}
```

Le père attend la fin du fils

```
// programme fils.c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("fils [%d]\n", getpid());
  return 0;
}
```

```
jupiter% gcc fils.c
jupiter% gcc -o parent parent.c
 jupiter% parent
  père[10524],  fils[10525]
  fils [10525]
  père[10524] Fin du fils[10525]
  Le père[10524] se termine
 jupiter%
```

Le fils change de code exécutable

Exemple 10 : Remplacement d’espace d’adressage (et errno)

```
1  // programme test_exec.c
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <errno.h>
7  int main ()
8  {        char* arg[] = {"ps", "-f", NULL};
9           printf("Bonjour\n");
10          execvp("ps", arg);
11          printf("Echec de execvp\n");
12          printf("Erreur %s\n",strerror(errno));
13          return 0;
14 }
```

Lectures suggérées

2.1 PROCESSES et 10.3 PROCESSES IN LINUX
Modern operating Systems, 4nd edition, Andrew S. Tanenbaum, publié par Pearson Education, Prentice-Hall, 2016. **Livre disponible dans le dossier Slides Aut 2018 du site moodle du cours.**