Université Ibn Tofail
Faculté des Sciences

Système d'Exploitation II

Processus

Filière
Sciences Mathématiques et Informatique (SMI)

Pr. Mohamed AMNAI
Département d'Informatique

2023/2024

1

Processus sous Unix

1.1 Introduction

1.1.1 Qu'est-ce qu'un processus ?

Un processus est un ensemble d'instructions se déroulant séquentiellement sur le processeur. En générale, un processus est l'abstraction d'un programme en cours d'exécution. Chaque exécution d'un programme donne lieu à un processus différent. A tout instant, un processeur exécute au plus un processus.

Dans un système d'exploitation multi-tâches, le processeur alterne entre l'exécution de plusieurs processus. Seuls les systèmes multi-coeur ou multi-processeur exécutent réellement plusieurs processus en même temps (un par coeur/processeur). Plus généralement, les processus partagent l'accès à différentes ressources : processeur, mémoire et périphériques.

Au cours d'exécution d'un programme (processus) A, le processeur peut interrompre l'exécution de ce processeur (A) pour exécuter un autre programme (B). Avant de basculer au processus (B) le microprocesseur marque l'instruction qu'il était en train d'exécuter afin de l'utiliser pour reprendre l'exécution du processus (A) dans une prochaine étape. Marquer l'endroit, où il s'est arrêté le processeur, représente le contexte d'exécution du processus. L'annotation qui permet de déterminer la prochaine instruction à exécuter d'un programme s'appelle le **compteur ordinal** ou **compteur d'instructions**.

En effet, un processus est caractérisé par un **programme**, des **données** et un **contexte** courant d'exécution.

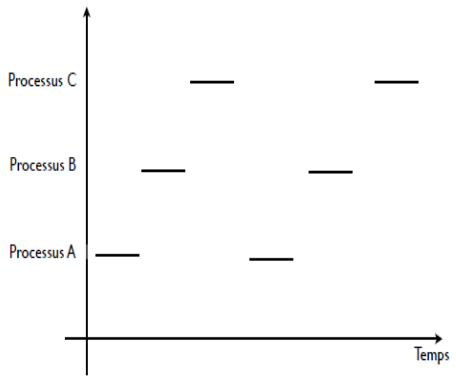

Figure 1.1: Principe de l'ordonnancement des processus

1.1.2 Partage du temps

Afin d'alterner entre plusieurs processus (A, B et C) et pour ne pas monopoliser l'utilisation du microprocesseur, à un instant donné, le A est attribué une unité de temps dite **quantum** (en milliseconde). Quand le laps de temps écoulé, le contexte de A est sauvegardé. Un autre laps de temps est attribué au processus (ex B) pour s'exécuter en sauvegardant ainsi son contexte avant de basculer à un autre processus. Supposant que A est le prochain processus, son contexte est rétabli pour continuer l'exécution du reste d'instructions (fig. **??**). Ainsi, tout instant, un seul processus utilise le processeur et un seul programme s'exécute.

Le choix du processus suivant est assuré par le système d'exploitation en utilisant la théorie d'ordonnancement (scheduling).

1.2 Hiérarchie des processus

1.2.1 Premier processus

Le processus ancêtre de tous les autres processus est le premier processus (**init**) créé au moment de démarrage de l'ordinateur. Ce premier processus fait partie des processus du noyau. À son exécution, il donne naissance à de nombreux processus exécutés en mode noyau. Ces processus assurent les services principaux et le déroulement normal du système. Ces derniers peuvent aussi créer d'autres processus système, exécutés en mode utilisateur, pour assurer d'autres services ou utiliser la machine par les utilisateurs.

1.2.2 Création d'un processus

Sous Unix, un nouveau processus est créé par duplication (par appel système **fork()**) d'un autre processus. Puis par écrasement du programme et des

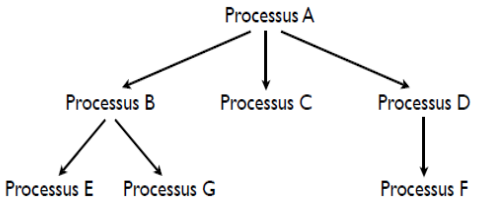

Figure 1.2: Arborescence des processus

donnés du processus cloné par le programme et les données du processus que l'on souhaite finalement créer (appel système **exec()**).

1.2.3 Processus : père et fils

Les processus des utilisateurs sont lancés par un interpréteur de commande. Ils peuvent eux-mêmes lancer ensuite d'autres processus. On appelle le processus créateur, le père, et les processus créés, les fils (voir **??**).

1.2.4 Arborescence des processus

Répéter la création, par duplication d'un processus (appelé processus fils), à partir d'un autre existant (appelé processus père) permet d'établir une arborescence de processus (fig. **??**). En principe, chaque processus a un seul père, et chaque processus peut avoir plusieurs fils.

1.2.5 Caractéristiques d'un processus

A un instant donné, un processus est caractérisé par de nombreuses informations, dont :

- Son état : exécution, suspendu, etc. ;
- Son identificateur **pid**;
- Son père **ppid** ;
- Son utilisateur **uid** ;
- Son compteur ordinal : indique la prochaine instruction à exécuter ;
- Sa pile d'exécution : mémorise l'empilement des appels de fonction ;
- Ses données en mémoire ;
- Toutes autres informations utiles à son exécution (E/S, fichiers ouverts, . .).

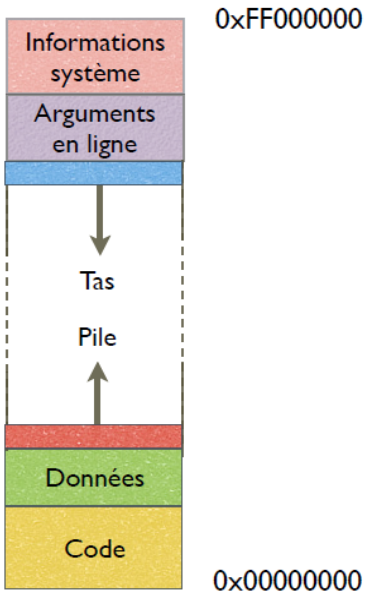

Figure 1.3: La structure de l'espace mémoire d'un processus

En pratique, sous Unix :

- la commande **ps** permet de voir la liste des processus existant sur une machine;
- **ps -xu** donne la liste des processus que vous avez lancé avec un certain nombre d'informations sur ces processus.
- **ps -axu** donne la liste de tout les processus lancés.

1.3 Structures de gestion des processus

L'exécussion d'un programme de processus nécessite différentes conditions comme l'allocation de la mémoire pour stocker le code et les données, l'allocation et l'initialisation de données utilisées par le système d'exploitation pour gérer les processus.

L'espace mémoire utilisé par un processus est divisé en plusieurs parties (fig. **??**) : le segment de code, le segment de données, la pile (stack) et le tas (heap).

1. **Segment de code** : Le segment de code est obtenu en copiant directement en mémoire virtuelle le segment de code du fichier exécutable. Il est placé dans des zones fixes de la mémoire. Le segment de code n'est généralement accessible qu'en lecture. Il est fréquent qu'il soit partagé par tous les processus exécutant le même programme.

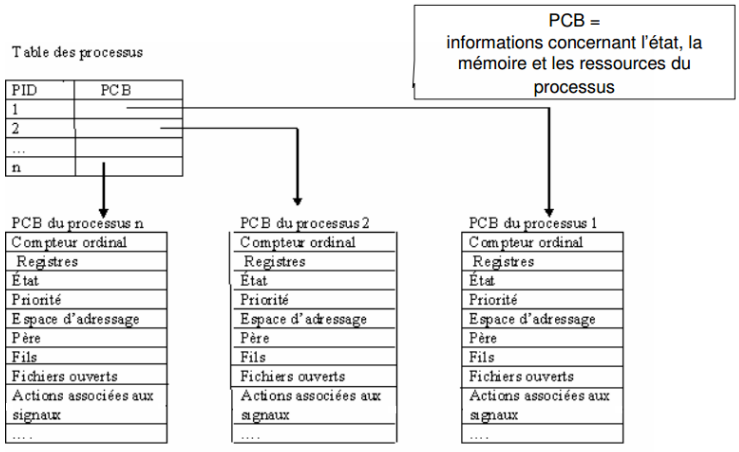

Figure 1.4: Table des processus et le Bloc de Contrôle de Processus

2. **Segment de données** : Il est composé d'un segment de données initialisées (*data*) et d'un segment de données non initialisées (**block storage segment bss**) créé dynamiquement. Le segment **bss** se résume alors aux variables locales de la fonction *main()*.

3. **Pile (stack)** : En cours de l'exécution, les données obtenues sont stockées dans un segment nommé la pile (stack). Il utilise le principe (Last in First Out) d'une pile, en empilant (ajout) et en dépilant (supprission) des données. Les données sont toujours stockées de façon contiguës. Ceci n'est pas possible en cas d'allocation dynamique car l'espace mémoire alloué par une fonction (ex : malloc()) ne pourra pas être dépilé et les données ne seront plus stockées de façon contiguë. Ceci conserve en mémoire l'emplacement des zones libres mais non contiguës.

4. **Tas (heap)** : Pour remédier le dernier problème d'allocation dynamique, le système utilise la (Tas) pour stocker des données de façon non contiguë. Il est situé à l'autre extrémité de l'espace mémoire de chaque processus.

5. **Bloc de Contrôle de Processus** : Le système d'exploitation maintient dans une table appelée «**table des processus**» les informations sur tous les processus créés. Elle contient une entrée par processus : **Bloc de Contrôle de Processus PCB** (fig. **??**). Ce bloc stock différents informations nécessaires à la gestions de processus par le système d'exploitation. Ces informations sont généralement stockées à la fin de l'espace mémoire du processus, après le tas (fig. **??**).

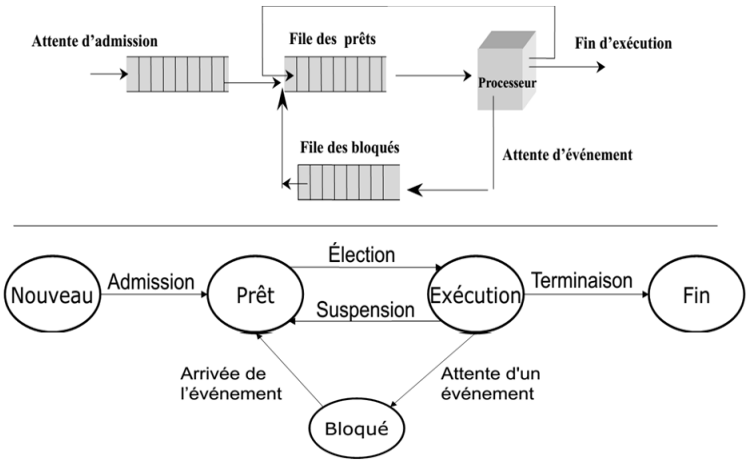

Figure 1.5: État de processus

1.4 Exécution des processus

1.4.1 État de processus

Dans un systèmes mono programmés le programme dispose de toutes les ressources de la machine, et ne quitte pas l'unité centrale avant de terminer son exécution. Par contre dans un système multiprogrammé, le temps (du processeur) est partagé par plusieurs processus. Par conséquent, un programme en exécution peut avoir plusieurs **états**.

Dans un système multitâches, plusieurs processus peuvent se trouver simultanément en cours d'exécution : ils partagent l'accès au processeur. Un processus peut prendre trois états (voir **??**, **??**):

- **Etat actif ou élu** (running): le processus utilise le processeur.
- **Etat prêt ou éligible** (ready): le processus pourrait utiliser le processeur s'il était libre (et si c'était son tour).
- **Etat en attente ou bloqué** : le processus attend une ressource (ex : fin d'une entrée-sortie).

1.4.2 Mode d'exécution

Deux modes d'exécution d'un processus peuvent avoir lieu.

- Mode **noyau** : accès sans restriction (manipulation de la mémoire, dialogue avec les contrôleurs de périphériques, . . .).
- Mode **utilisateur** : accès restreint, certaines instructions sont interdites (pas d'accès direct aux périphériques). Il peut être interrompu par d'autres processus.

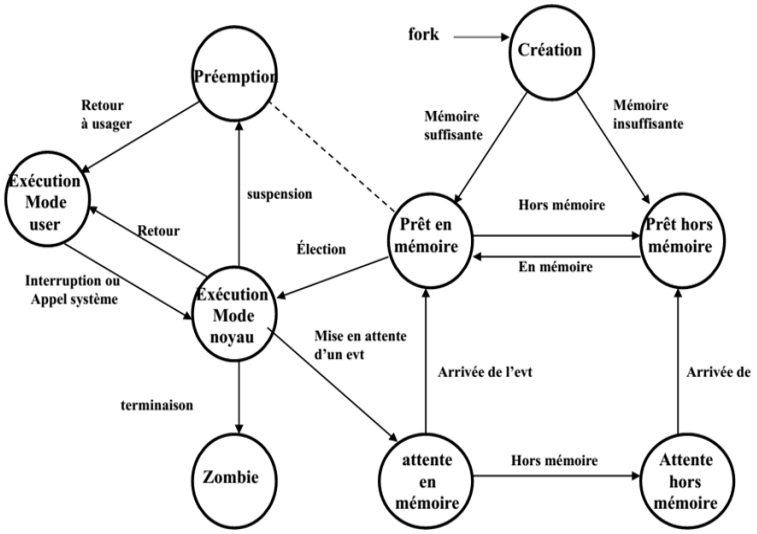

Figure 1.6: État de processus

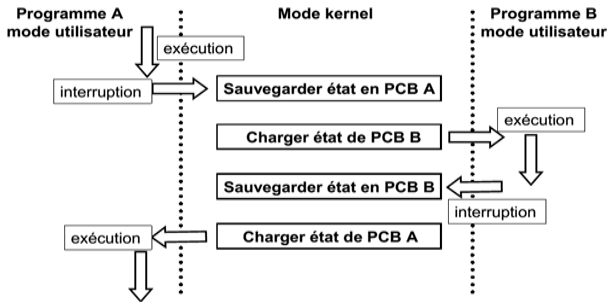

Figure 1.7: Changement de contexte

Les appels systèmes permettent à un processus en mode utilisateur d'accéder (temporairement) à des fonctions nécessitant le mode noyau.

1.4.3 Changement de contexte de processus

Le changement de contexte (**??**) entre processus est le passage du mode utilisateur au mode kernel. Il implique une gestion adéquate des PCB.

1.4.4 Notion d'image

Lors de l'exécution d'un programme, ce dernier sollicite un ensemble de composant :

- Code;
- Données (Statique, Tas, Pile);

- Contexte d'exécution
  - Pointeur d'instruction
  - Registres mémoire
  - Fichiers ouverts
  - Répertoire courant
  - Priorité

Une **image** est un ensemble d'objets qui peuvent donner lieu à une exécution (un code exécutable). Un processus exécute une image.
Un programme (code) qui ajoute +1 à des entiers (données) sauvegardées dans un fichier (contexte). L'instance en train de s'exécuter (suivi des instructions dans l'environnement) est le processus.

1.5 Programmation des processus

1.5.1 Fonctions d'identification des processus

Plusieurs identificateurs sont associés à un processus :

- Numéro de processus (pid) ;
- Numéro du processus père (ppid) ;
- Identificateur d'utilisateur réel ;
- Identificateur d'utilisateur effectif (bit setuid) ;
- Identificateur de groupe réel ;
- Identificateur de groupe effectif (bit setgid) ;
- Liste d'identificateurs de groupes.

Nous avons la possibilité de connaître ces informations dynamiquement via des appels systèmes. Ces fonctions sont les suivantes :

- **pid_t getpid(void)** : Cette fonction retourne le pid du processus.
- **pid_t getppid(void)** : Cette fonction retourne le pid du processus père. Si le père s'est terminé avant le processus fils, la valeur retournée est **1** celle du processus (*init*). Le type **pid_t** est de type *int* et à été défini par un appel à **typedef**.

L'ancêtre **pid = 1**, le processus **init**, lancé au "boot". Les orphelins (processus dont le père est mort) sont le plus souvent récupérés par **init**.

Exemple : pid du processus et de son père (**??**).

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    printf("mon pid est : %d\n", (int)getpid());
    printf("le pid de mon pere est : %d\n", (int)getppid());
    exit(EXIT_SUCCESS);
}
```

Figure 1.8: pid

1.5.2 Création de processus

La bibliothèque C propose plusieurs fonctions pour créer des processus. Cependant toute ces fonctions utilisent l'appel système **fork()** qui est la seule et unique façon de demander au système d'exploitation de créer un nouveau processus.

- **pid_t fork(void)** : Cet appel système crée un nouveau processus (voir l'Exemple 1, **??**). La valeur retournée est le **pid** du fils pour le processus père, ou **0** pour le processus fils. La valeur **-1** est retournée en cas d'erreur.
- Recopie totale (données, attributs) du processus père vers son processus fils (nouveau pid) (voir **??**, **??**).
- Le fils continue son exécution à partir de cette primitive.
- Héritage du fils :
  - La priorité ;
  - La valeur du masque ;
  - Le propriétaire ;
  - Une copie des descripteurs des fichiers ouverts ;
  - Le pointeur de fichier (offset) pour chaque fichier ouvert ;
  - Le comportement vis à vis des signaux.

Exemple 1 avec fork()

```c
#include<stdio.h> //ex1
#include <sys/types.h>
#include <unistd.h>
```

```
int main()
{
 int f;
 f = fork();

 printf ("\n Valeur retournee par la fonction fork : \%d\n",(int)f );
 printf ("Je suis le processus numero \%d\n",(int)getpid() );

return 0;
}
```

Résultat de l'exemple 1

```
[root@localhost app]# ./t1
 Valeur retournee par la fonction fork: 3952
 Je suis le processus numero 3951
 Valeur retournee par la fonction fork: 0
 Je suis le processus numero 3952
```

Figure 1.9: Exemple avec fork()

Avant fork()

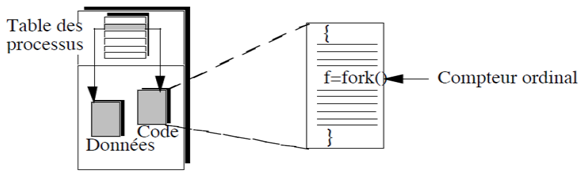

Figure 1.10: Exemple avant exécution de fork()

Après fork()

Exemple 2

```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() //ex2
{
 int f;
```

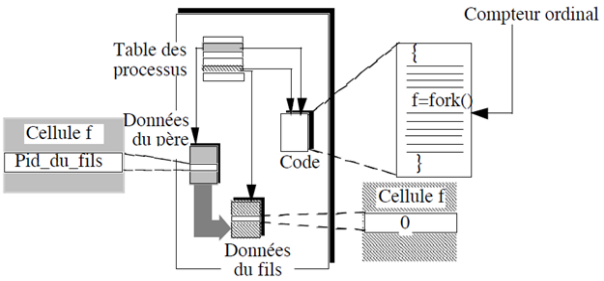

Figure 1.11: Exemple après exécution de fork()

```
 f = fork();
 if(f == -1){
  printf ("\n Erreur : le processus ne pas etre cree");
 }
 if(f == 0){
  printf ("\n Ici le processus fils \n");
  printf (" Je suis le processus numero \%d\n",(int)getpid() );
 }
 if(f != 0){
  printf ("\n Ici le processus pere \n");
  printf (" Je suis le processus numero \%d\n",(int)getpid() );
 }
return 0;
}
```

Résultats de l'exemple 2

```
[root@localhost SharedVirtuaMachines]# ./2_fork
Ici le processus pere
Je suis le processus numero 28468
[root@localhost SharedVirtuaMachines]# Ici le processus fils
Je suis le processus numero 28469
```

Figure 1.12: Exemple avec test sur le retour de fork()

Exemple 3 : Porté du code

```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()//ex3
{
```

```
 int f;
 printf ("\n Je suis seul au monde \n");
 f = fork();
 if(f == -1){
  printf ("Erreur fork()\n");
 }
 if(f == 0){
  printf ("\n Ici le processus fils \n");
 }else{
  printf ("\n Je suis le processus pere \n");
 }
 printf ("\n Et la qui suis-je  \%d\n", (int)getpid() );
return 0;
}
```

Résultats de l'exemple 3

```
[root@localhost SharedVirtuaMachines]# ./3_porte_du_code
Je suis seul au monde
Je suis le processus pere
Et la qui suis-je  28592
[root@localhost SharedVirtuaMachines]# Ici le processus fils
Et la qui suis-je  28593
```

Figure 1.13: Exemple sur la porté du code

Exemple 4 sur la porté des variables

```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()//ex3
int main(){ //ex4
 int i,j,f;
 i=5; j=2;

 f = fork();
 if(f == -1){
  printf ("Erreur fork()\n");
 }
 if(f == 0){
  //code du fils
  printf ("\nfils \%d\n",getpid());
  j--;
```

```
 }
 else{
  printf ("pere \n",getpid());
  i++;
 }
 printf ("Pid : \%d i:\%d j:\%d\n", getpid(),i,j);
return 0;
}
```

Résultats de l'exemple 4

```
[root@localhost SharedVirtuaMachines]# ./4_porte_de_varibale
pere
Pid : 28644 i:6 j:2
[root@localhost SharedVirtuaMachines]# fils 28645
Pid : 28645 i:5 j:1
```

Figure 1.14: Exemple sur la porté des variables

1.5.3 Synchronisation

Un processus père toujours prévenu de la fin d'un fils et le fils toujours prévenu fin du père. Mais il faut que le père soit en attente. Si la fin d'un fils n'est pas traitée par le père, ce processus devient un processus **zombie**. Synchronisation pour les fins d'exécution :

- Possibilités d'échanges d'informations permettant une synchronisation sur les fins d'exécutions.
- Eviter les processus zombies (fin propre).
- Nécessite que le père soit en attente.
- Information de la fin du processus fils.

1.5.4 Appel système wait() et exit()

La valeur de retour de la fonction **main()** est renvoyée grâce à un appel à la fonction **exit()**. Les paramètres de la fonction **main()** permettent au programme appelant de passer des paramètres au programme appelé. La fonction **exit()** permet au programme appelé de retourner un paramètre au programme appelant. La fonction **exit()** termine le programme et prend comme paramètre un entier signé qui pourra être lu par le programme appelant. Un **exit(0)** signifie que le programme s'est exécuté sans erreur,

- **Exit (int)**
- **-EXIT_SUCCESS** qui vaut 0
- **-EXIT_FAILURE** qui vaut 1
- Valeur du **int** est "transmise" au père : code de retour.
- Fin du processus fils après exit.
- Différentes valeurs peuvent désigner différents types d'erreurs

Il est souvent très pratique de pouvoir attendre la fin de l'exécution des processus fils avant de continuer l'exécution du processus père. La fonction **wait()** permet de suspendre l'exécution du père jusqu'à ce que l'exécution d'un des fils soit terminée.

- **pid_t wait(int *satatus)** (voir l'exemple 5, **??**)
- **Entier retourné** : **pid** du fils qui s'est **terminé** depuis l'invocation du **wait**
- **En cas d'erreur** la valeur retournée est **-1**
- L'entier pointé enregistre l'état du fils lorsqu'il est finit (valeur en paramètre dans exit)

Si le pointeur **status** est différent de **NULL**, les données retournées contiennent des informations sur la manière dont ce processus s'est terminé, comme par exemple la valeur passée à **exit()**.

Exemple 5 avec wait()

```
#include<stdlib.h> //ex5
#include<sys/wait.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){ int i,j;
printf("je suis le pid \%d avant le fork\n",  getpid());
i = fork();
if (i != 0) { //  i != 0  seulement pour le père
j = wait(NULL);
printf("le pere \%d :  wait a retourne \%d\n", getpid(), j);
}

printf("je suis \%d apres le fork, il a retourne \%d\n", getpid(), i);
```

```
exit(EXIT_SUCCESS);
}
```

Résultats de l'exemple 5

```
[amnai@localhost Documents]$ ./ex5
je suis le pid 8885 avant le fork
je suis 8886 apres le fork, il a retourne 0
le pere 8885 :  wait a retourne 8886
je suis 8885 apres le fork, il a retourne 8886
```

Figure 1.15: Exemple avec wait()

Exemple 6 avec wait et exit

```
#include<stdlib.h> //ex6
#include<sys/wait.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){ int r,s,w;

 if((r = fork()) == 0){
  printf (« Le pid du Fils \%d\n",getpid());
  //Traitement long
  exit(14);
 }else{
  //Père doit attendre la mort de son fils
  w=wait(&s);
  printf ("Le pere w : \%d s:\%d \n",w,s);
 }

return 0;
}
```

Résultats de l'exemple 6

1.5.5 Mise en sommeil d'un processus : sleep()

La fonction **sleep()** Suspend l'exécution du processus appelant pour une durée de **n** secondes : **Int sleep(int n)**

```
[amnai@localhost Documents]$ ./ex6
Fils 8938
w : 8938 s:3584
```

Figure 1.16: Exemple avec wait() et exit()

Exemple 7 avec sleep()

```
#include<stdlib.h>
#include<sys/wait.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() //ex7
{
 int PID, status;
 printf ("processus pere debut \%d\n",getpid());

 if(fork() == 0){
  printf ("processus fils \%d\n",getpid());
  exit(EXIT_SUCCESS);
 }
 PID=wait(&status);
 printf("processus pere \%d\n", getpid());
 printf ("sortie du wait \n");
 sleep(15);
 printf("PID = \%d status = \%d \n", PID, status);
 exit(EXIT_SUCCESS);
}
```

Résultats de l'exemple 7

1.5.6 Recouvrement de processus sous Unix

Un processus peut changer de code par un appel système à **exec** (voir l'exemple 8).

- Code et données remplaces
- Pointeur d'instruction réinitialisé

```
#include <unistd.h>
int execl(const char *path,
          const char *arg0, ...,const char *argn, char * /*NULL*/);
```

```
[amnai@localhost Documents]$ ./ex7
processus pere debut 9161
processus fils 9162
processus pere 9161
sortie du wait

      PID = 9162 status = 0
      [amnai@localhost Documents]$
```

Figure 1.17: Exemple avec sleep()

- **path** = nom de l'exécutable recherche dans $PATH
- **arg0** = nom exécutable affiché par **PS**
- **argn** = n - 2ieme argument de l'exécutable
- **NULL** = argument de fin de la ligne de commande

Exemple 8 avec excel()

```
int main() //ex8
{
 printf ("Je suis un programme qui va exécuter /bin/ls -l \n");

 execl("/bin/ls","ls","-l",NULL);
}
```

Résultats de l'exemple 8

```
[amnai@localhost Documents]$ ./ex8
Je suis un programme qui va exécuter /bin/ls -l
total 96
-rwxrwxr-x. 1 amnai amnai 4945 22 nov.  11:43 ex1
-rw-rw-r--. 1 amnai amnai  250 22 nov.  11:43 ex1.c
-rwxrwxr-x. 1 amnai amnai 5129 22 nov.  11:53 ex2
```

Figure 1.18: Exemple avec excel()

Exemple 9 avec excel() Lorsqu'il s'exécute **execl**, il est totalement remplacé par l'exécution du "**ls -l**", le processus avec le programme, disparaît.

```
int main() //ex9
{
 printf ("Je suis un programme qui va executer /bin/ls -l \n");

 execl("/bin/ls","ls","-l",NULL);

 printf ("Je suis le programme qui a execute /bin/ls -l \n");

 //Code qui ne sera pas exécuté !!!!!!
}
```

Résultats de l'exemple 9

```
[amnai@localhost Documents]$ ./ex9
Je suis un programme qui va exécuter /bin/ls -l
total 108
-rwxrwxr-x. 1 amnai amnai 4945 22 nov.  11:43 ex1
-rw-rw-r--. 1 amnai amnai  250 22 nov.  11:43 ex1.c
-rwxrwxr-x. 1 amnai amnai 5129 22 nov.  11:53 ex2
```

Figure 1.19: Exemple avec excel()