Chapitre 3 - Les processus

Table des matières

1 - Définitions

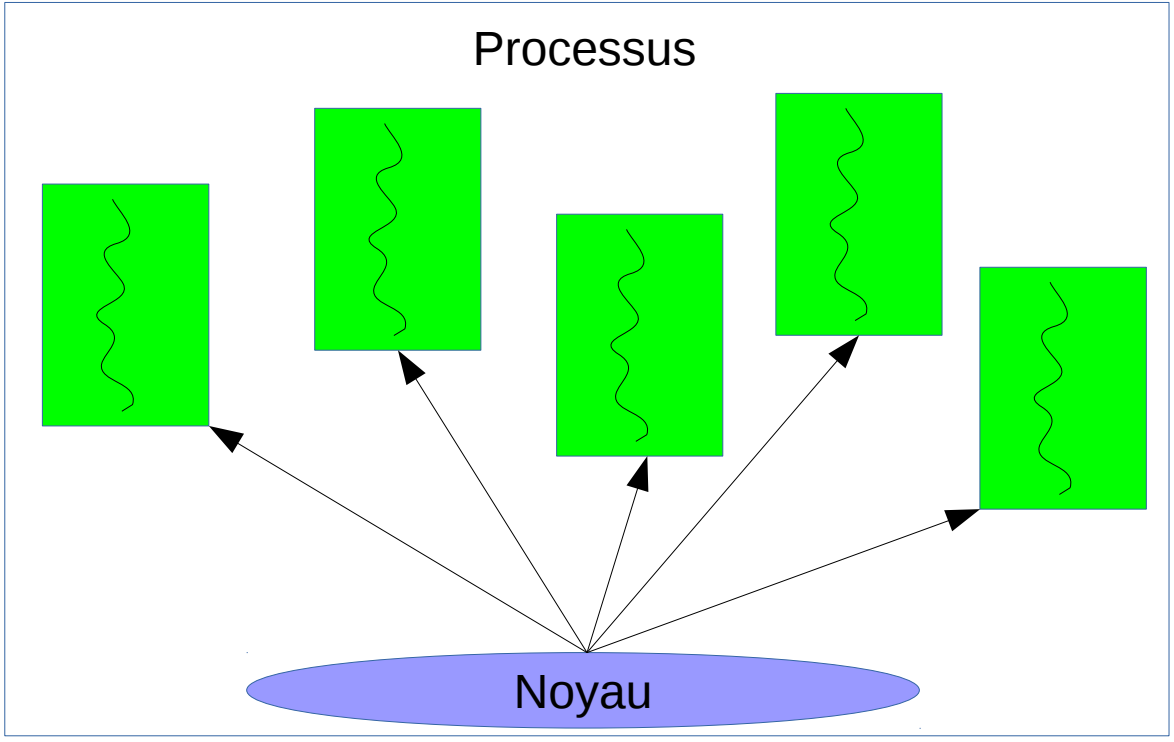
1.1 - Notion de processus

Un programme est un objet présent sur le système, une entité statique, associée à la suite des instructions qui le composent. Un processus est une entité dynamique, associée à la suite des actions réalisées par un programme, lors d'une exécution particulière.

Un processus représente donc l'exécution d'un programme par le système d'exploitation.

Dans un système d'exploitation, plusieurs processus correspondants à plusieurs programmes peuvent s'exécuter simultanément. De même, un programme doit pouvoir créer un nouveau processus en réalisant un appel système approprié.

Le système possède donc une table de processus globale permettant d'intéragir avec ces derniers.


Processus
Noyau

1.2 - Création de processus

Que ce soit par le système lui-même ou par un processus utilisateur, la création d'un nouveau processus est réalisé par un appel système spécifique permettant d'initialiser l'espace mémoire du nouveau processus créé :

- **Sous Unix** : Il s'agit de l'appel système «fork( ) », qui va simplement cloner le processus appelant, les deux processus sont donc identiques, et un autre appel système (primitives « exec...( ) ») sera nécessaire pour changer le code,
- **Sous Windows** : Il s'agit de l'appel système « CreateProcess( ) », qui prend directement en paramètre le nom du programme à charger dans le nouvel espace mémoire, les deux processus sont donc directement différents.

1.3 - Destruction d'un processus

Un processus est souvent associé à l'exécution d'un programme ponctuel, c'est à dire possédant une durée de vie bornée. Ainsi, une fois l'exécution du programme terminée, le processus se termine naturellement après sa dernière instruction.

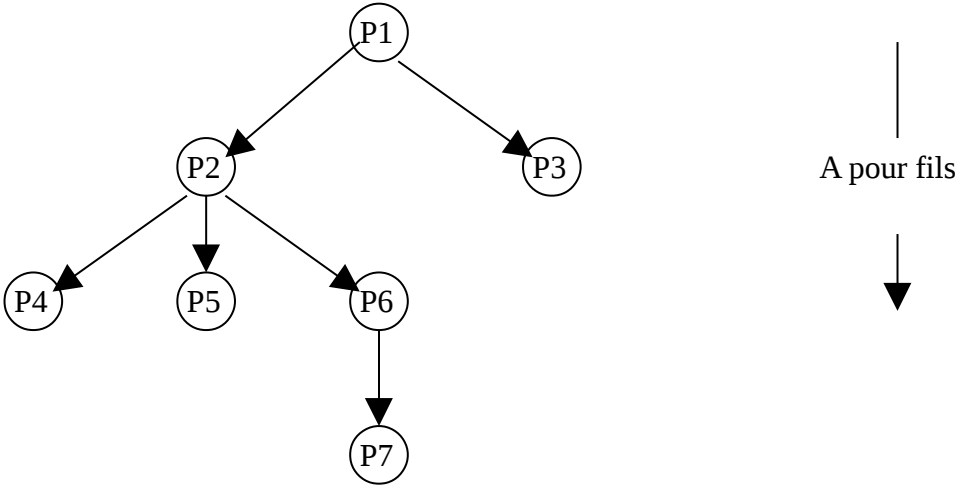
Cependant, un processus peut également se terminer pour d'autres raisons :

- Arrêt normal volontaire : Le processus exécute une instruction de sortie suite à la fin du travail alloué,
- Arrêt anormal volontaire : Le processus détecte un problème ou une erreur et décide de stopper son exécution,
- Arrêt système : Le processus est stoppé par le système, en général suite à une erreur fatale,
- Arrêt autre : Le processus peut se stopper à la demande d'un autre processus sous certaines conditions,

1.4 - Hiérarchie des processus

Un processus peut en créer un autre : le premier est appelé processus père, le second processus fils. Le fils peut à son tour créer d'autres processus dont il sera le père …

Ces suites de création conduisent à une structure arborescente :



Particularités :

- **Sous Unix** : Cette hiérarchie est très forte, tout processus possède obligatoirement un père et un seul, et un appel système spécifique permet même de connaître son père. Le père ne peut abandonner un fils volontairement, par contre s'il meurt, le fils est adopté par le processus du plus haut niveau de la hiérarchie.
- **Sous Windows** : Cette notion est relativement symbolique. En fait, lors de la création d'un processus, le créateur récupère un « handle » qui caractérise ce nouveau processus. Néanmoins, il peut transmettre cette information à d'autres processus. De plus, le processus nouvellement créé n'a pas de lien particulier avec son père.

1.5 - Etat d'un processus

Un processus possède un état qui renseigne sur ce qu'il fait :

- **Exécution** : Le processus est en cours d'exécution par le système, lorsque ce dernier aura épuisé son quota de temps, il sera placé dans l'état « Prêt »,
- **Prêt** : Le processus est prêt à être exécuté mais le processeur n'est pas disponible, il attend donc que ce dernier le devienne,
- **Bloqué** : Le processus est bloqué en attente d'une ressource, en général une entrée/sortie, il ne peut donc être exécuté pour le moment. Une fois la ressource obtenue, il sera à nouveau placé en état prêt.

Sous Unix, d'autres états existent comme Stoppé, Zombie, …

1.6 - Table de processus

Le système possède une table globale des processus existant, dans laquelle est enregistrée les informations s'y rapportant, par exemple le propriétaire du processus, le processus parent, …

Néanmoins, le processus étant interruptible par le système, ce dernier doit également y enregistrer tous les éléments qui lui permettra de reprendre son exécution :

- Compteur ordinal,
- Registres,
- Fichiers ouverts,
- ...

Chaque processus possède donc une entrée dans cette table, appelée parfois BCP pour Bloc de Contrôle du Processus.

2 - Implémentation Unix

2.1 - Accès aux données du BCP.

Chaque processus possède un bloc de contrôle (BCP) qui contient toutes ses caractéristiques.

Identité du processus

```
#include  <unistd.h>
pid_t  getpid(void);
pid_t  getppid(void);
```

Propriétaires du processus

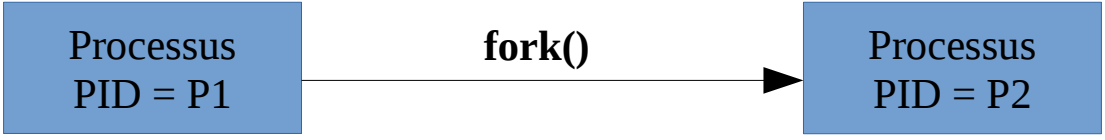
```
#include  <unistd.h>
uid_t  getuid(void);
uid_t  geteuid(void);
gid_t  getgid(void);
gid_t  getegid(void);
```

2.2 - La primitive fork

Tout processus peut décider de « cloner » par l'appel système « *fork()* ». En cas d'échec, comme tout appel système, la primitive renvoie la valeur -1 et la variable « *errno* » contient le numéro de l'erreur. En cas de succès, le processus appelant a été cloné : le processus initial est appelé ***père*** tandis que le processus cloné est appelé ***fils***. Le père garde son PID et continue sa vie tandis que le fils a un nouveau PID et commence la sienne...

Immédiatement après le « *fork()* », les deux processus exécutent le même code avec les mêmes données, les mêmes fichiers ouverts, le même environnement mais on peut les différentier par la valeur renvoyée par la primitive :

- 0 : pour le processus fils,
- PID du processus fils : pour le processus père,



Exemple d'utilisation de fork

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

int main(void) {
  pid_t  retour;

  if((retour = fork()) == -1) {
    perror("Echec de fork ...") ;
    exit(1) ;
  }
  printf("Affichage par le fils et par le père\n ");
  if(retour == 0) printf("Affichage par le fils\n ");
  else printf("Affichage par le père\n ");

  exit(0);
}
```

L'ordre dans lequel les affichages vont avoir lieu est totalement imprévisible car les deux processus père et fils sont concurrents.

Afin de pouvoir observer les deux processus avec la commande système « *ps* », la modification suivante du programme précédent serait intéressante :

```
printf("Affichage par le fils et par le père\n ");
if(retour == 0) {
  while(1) {
    printf("PID du fils : %d\n ",getpid());
    sleep(1);
  }
} else {
  while(1) {
    printf("PID du père : %d\n ",getpid());
    sleep(1);
  }
}
```

2.3 - La primitive wait

Un processus fils qui se termine envoie à son père le signal SIGCHLD et reste dans un état dit « Zombie » tant que le processus père n'aura pas consulté son code de retour.

La prolifération de processus à l'état zombie doit être formellement évitée, aussi il faut utiliser la primitive « *wait()* » qui permet d'attendre la mort d'un processus fils et :

- retourne -1 si le processus appelant n'a aucun fils,
- bloque le processus appelant si aucun fils n'est terminé,
- retourne le PID du fils terminé sinon,

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

Exemple d'attente de processus fils

```
int main(void) {
  pid_t pid,pidfils ;
  int   retour ;
  printf("Processus père de PID %d\n ",getpid()) ;
  switch(pid = fork()) {
    case (pid_t)-1 :
        perror("naufrage... ") ;
        exit(2) ;
    /* Processus Fils */
    case (pid_t)0 :
        printf("Processus fils PID=%d \n ",getpid()) ;
        sleep(30) ;
        exit(33) ;
    /* Processus père */
    default :
        pidfils = wait(&retour) ;
        printf("Mort du processus PID=%d\n ",pidfils) ;
  }
}
```

De plus, la primitive « *wait()* » renseigne sur la terminaison du processus fils :

- Si le processus fils s'est terminé seul, par exemple par un appel à « *exit(code)* ». Il est alors possible également de récupérer la valeur de « code »,
- Si le processus fils a été tué par un signal. Il est alors possible de connaitre le numéro de signal utilisé.

Le fichier <sys/wait.h> contient des fonctions de traitement de cet entier **status* qui sont macro-définies et qui permettent la réalisation d'une « autopsie » des processus fils :

```
if(WIFEXITED(retour)) {
  printf("code de retour :%d\n",WEXITSTATUS(retour));
  exit(0) ;
}
if(WIFSIGNALED(retour)) {
  printf(" tué par le signal %d\n ",WTERMSIG(retour));
  if(WCOREDUMP(retour)) {
    printf(" fichier core créé\n ") ;
  }
  exit(1) ;
}
```

Comportement :

- le code de retour du fils est égal à « 33 » sauf si il est « tué » par l'envoi d'un signal.
- le code de retour du père sera égal à 2 si la primitive « *fork()* » a échoué,
- à 1 si le processus fils a été tué,
- à 0 si le processus fils s'est auto-détruit en appelant la fonction exit.

En plus de l'autodestruction d'un processus, la fonction *exit* réalise la libération des ressources allouées au processus au cours de son exécution.

Autre façon d'éliminer les zombies

La prolifération de processus à l'état zombie doit être formellement évitée, aussi lorsque le père n'a pas à être synchronisé sur ses fils, on peut inclure dans le code du père l'instruction signal(SIGCHLD,SIG_IGN) pour éliminer les fils à l'état Z au fur et à mesure de leur apparition.

2.4 - Les primitives exec

Les primitives de la famille *exec** permettent à un processus de charger en mémoire, en vue de son exécution, un nouveau programme binaire. En cas de succès de l'opération, il y a écrasement du code, des données et par conséquent aucun retour arrière n'est possible. Le processus garde son PID et donc la plupart de ses caractéristiques : il n'y a aucune création de processus.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
  system("ps f");
  execl("/bin/ps","ps","f",NULL);
  perror("execl");
  exit(1);
}
```

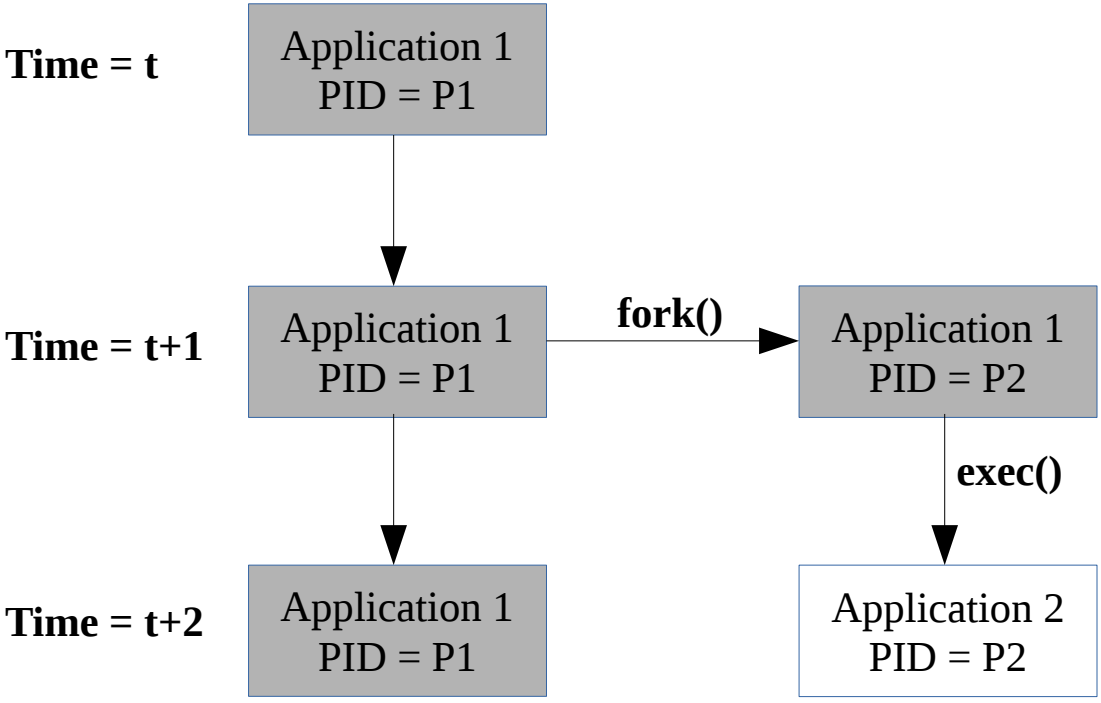
Les différentes fonctions sont les suivantes :

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., 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[]);
```

Les fonctions « execl* » prennent leurs arguments sous forme d'une liste alors que les fonctions « execv* » les prennent sous la forme d'un tableau.

2.5 - Le mécanisme du fork/exec

Principe



Fonctionnement canonique d'un shell

Un interpréteur de commandes, utilisé de façon interactive, fonctionne en boucle infinie, selon le shéma suivant :

1. Lecture ligne de commandes
2. Interprétation (substitutions...)
3. Clonage du shell avec fork()
4. Chez le père : attente du fils avec wait() puis retour en 1
   Chez le fils : changement de code avec exec() puis mort.

Exemple de Shell minimaliste

```
int main(void) {
  pid_t pid,pidfils;
  int   retour;
  char buffer[1024];

  printf("[prompt]> ");
  fgets(buffer,1024,stdin);
  if (buffer[strlen(buffer)-1]=='\n')
              buffer[strlen(buffer)-1]='\0';
  switch(pid = fork()) {
    case (pid_t)-1 :
        perror("naufrage... ");
        exit(2);
    case (pid_t)0 :
        execlp(buffer,buffer,NULL);
        fprintf(stderr,"Commande inconnue ...\n");
        exit(1);
    default :
        pidfils = wait(&retour) ;
  }
}
```