Chapitre 3

Processus

Le concept de **processus** est le plus important dans un système d'exploitation. Tout le logiciel d'un ordinateur est organisé en un certain nombre de processus séquentiels. En simplifiant, on peut considérer un processus comme un programme qui s'exécute et possède un compteur ordinal, pour indiquer à quelle instruction il est rendu, des registres, des variables et une pile d'exécution (nous verrons plus tard qu'un processus peut contenir plusieurs thread, donc plusieurs piles d'exécution). Son exécution est, en général, une alternance de calculs effectués par le processeur et de requêtes d'Entrée/Sortie effectuées par les périphériques.

3.1 Processus

3.1.1 États d'un processus

Lorsqu'un processus s'exécute, il change d'état, comme on peut le voir sur la figure **??**. Il peut se trouver alors dans l'un des trois états principaux :

- **Élu** : en cours d'exécution.
- **Prêt** : en attente du processeur.
- **Bloqué** : en attente d'un événement.

Initialement, un processus est à l'état prêt. Il passe à l'état exécution lorsque le processeur entame son exécution. Un processus passe de l'état exécution à l'état prêt ou lorsqu'il est suspendu provisoirement pour permettre l'exécution d'un autre processus. Il passe de l'état exécution à l'état bloqué s'il ne peut plus poursuivre son exécution (demande d'une E/S). Il se met alors en attente d'un événement (fin de l'E/S). Lorsque l'événement

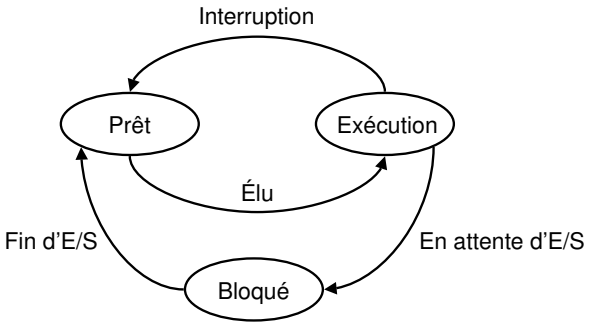

FIG. 3.1 – Les états principa-ux d'un processus.

survient, il redevient prêt.

Le modèle à trois états est complété par deux états supplémentaires : **Nouveau** et **Fin**, qui indiquent respectivement la création d'un processus dans le système et son terminaison, comme illustré à la figure **??**.

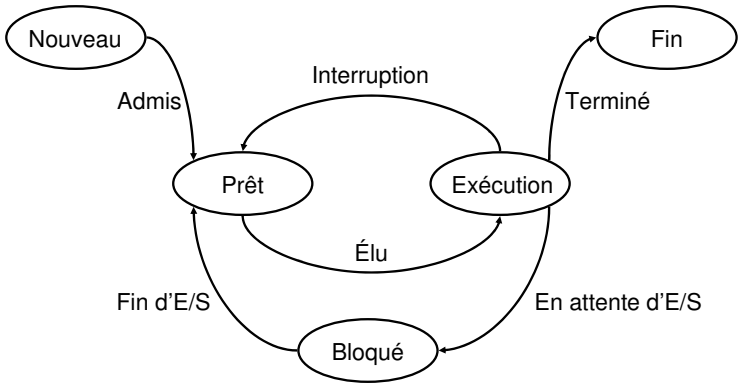

FIG. 3.2 – Modèle à cinq états d'un processus.

Consulter le Chapitre **??**, *Ordonnancement des processus* pour plus de détails sur les états de processus en Unix/Linux.

3.1.2 Implémentation de processus

Pour gérer les processus, le système d'exploitation sauvegarde plusieurs informations dans des structures de données. Il existe une table pour contenir les informations concernant tous les processus créés. Il y a une entrée par processus dans la table, appelée le **Bloc de Contrôle de Processus**

(**PCB**).

Bloc de Contrôle de Processus

Chaque entrée de la table **PCB** comporte des informations sur :
- Le `pid` du processus.
- L'état du processus.
- Son compteur ordinal (adresse de la prochaine instruction devant être exécutée par ce processus).
- Son allocation mémoire.
- Les fichiers ouverts.
- Les valeurs contenues dans les registres du processeur.
- Tout ce qui doit être sauvegardé lorsque l'exécution d'un processus est suspendue.

Sur la figure **??**, nous montrons les principales entrées de la **PCB**.

Gestion des processus	Gestion de la mémoire	Gestion des fichiers
Registres	Pointeur seg. code	Répertoire racine
Compteur ordinal (`co`)	Pointeur seg. données	Répertoire de travail
Priorité	Pointeur seg. pile	`fd`
Pointeur de pile (`sp`)		`uid`
État du processus		`gid`
Date de lancement		
Temps UCT utilisé		
Temps UCT des fils		
Signaux		
`pid`		
`ppid`		

FIG. 3.3 – Principales entrées de la **Table des processus** d'Unix. Les entrées `co`, états de processus, `pid` et `ppid` seront étudiées dans ce chapitre. Celles des descripteurs de fichiers (`fd`), signaux, priorité, temps, gestion de mémoire et répertoires seront étudiées ultfieurement.

Changement de contexte de processus

Une des principales raison pour justifier l'existence des blocs de contrôle des processus est que dans un système multiprogrammé on a souvent besoin de redonner le contrôle du CPU à un autre processus. Il faut donc mémoriser toutes les informations nécessaires pour pouvoir éventuellement

relancer le processus courant dans le même état. Par exemple, il faut absolument se rappeler à quelle instruction il est rendu. La figure **??** illustre le changement de contexte entre processus. Le processsus en cours est interrompu et un ordonnanceur est éventuellement appelé. L'ordonnanceur s'exécute en mode *kernel* pour pouvoir manipuler les **PCB**. Le changement de contexte sera expliqué plus en détail dans le Chapitre **??**, *Ordonnancement de processus*.

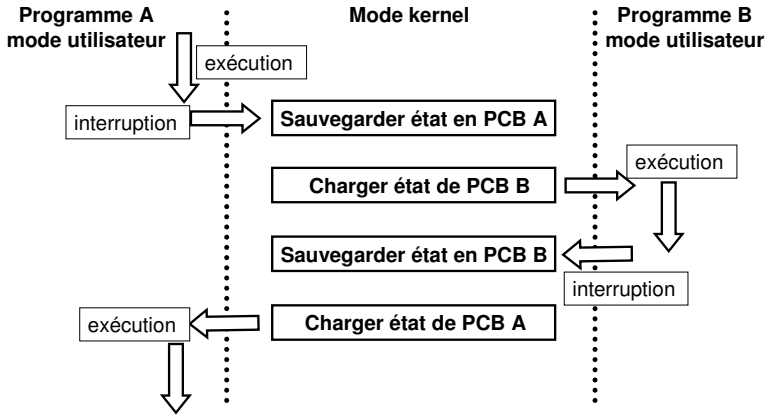

FIG. 3.4 – Le changement de contexte.

Il est important de noter que le passage au mode kernel par un appel système n'implique pas nécessairement un changement de contexte. On reste en général dans le même processus, sauf qu'on a accès à des données et des instructions qui sont interdites en mode utilisateur.

3.1.3 Les démons

Les **démons** sont des processus particuliers. Un démon s'exécute toujours en arrière-plan (*background*). Ceci implique que son père n'attend pas la fin de son exécution. Les démons ne sont associés à aucun terminal ou processus `login` d'un utilisateur. Ils sont toujours à l'écoute et attendent qu'un événement se produise. Ils réalisent des tâches de manière périodique.

Les démons démarrent normalement au début du chargement du système d'exploitation et ne meurent pas. Les démons ne travaillent pas : ils lancent plutôt d'autres processus pour effectuer les tâches. Comme exemples de démons on retrouve le **voleur de pages** de la mémoire virtuelle (voir

Chapitre **??** sur la gestion de la mémoire) ou le **démon des terminaux** qui lance `getty`, puis `login` à l'invite des utilisateurs.

3.1.4 Création et terminaison de 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 fils qui, à leur tour, peuvent créer des processus fils sous une forme de structure arborescente. Le processus créateur est appelé processus père.

Dans certains systèmes, comme MS-DOS, lorsqu'un processus crée un fils, l'exécution du processus père est suspendue jusqu'à la terminaison du processus fils. C'est ce qu'on appelle l'**exécution séquentielle**. Par contre dans les systèmes du type Unix, le père continue à s'exécuter en concurrence avec ses fils. C'est ce qu'on appelle **exécution asynchrone**. Un processus fils créé peut partager certaines ressources comme la mémoire ou les fichiers avec son processus père ou avoir ses propres ressources. Le processus père peut contrôler l'usage des ressources partagées et peut avoir une certaine autorité sur ses processus fils. Également, il peut suspendre ou détruire ses processus fils. Il peut également se mettre en attente de la fin de l'exécution de ses fils. L'espace d'adressage du processus fils est obtenu par duplication de celui du père. Il peut exécuter le même programme que son père ou charger un autre programme. Un processus se termine par une demande d'arrêt volontaire (appel système `exit()`) ou par un arrêt forcé provoqué par un autre processus (appel système `kill()`). Lorsqu'un processus se termine toutes les ressources systèmes qui lui ont été allouées sont libérées par le système d'exploitation.

3.2 Services Posix pour la gestion de processus

La norme Posix définit un nombre relativement petit d'appels système pour la gestion de processus :

- `pid_t fork()` : Création de processus fils.
- `int execl(), int execlp(), int execvp(), int execle(), int execv()` : Les services `exec()` permettent à un processus d'exécuter un programme (code) différent.
- `pid_t wait()` : Attendre la terminaison d'un processus.

- `void exit()` : Finir l'exécution d'un processus.
- `pid_t getpid()` : Retourne l'identifiant du processus.
- `pid_t getppid()` : Retourne l'identifiant du processus père.

En Linux, le type `pid_t` correspond normalement à un `long int`.

3.3 Création de processus

3.3.1 Création de processus avec `system()`

Il y a une façon de créer un sous-processus en Unix/Linux, en utilisant la commande `system()`, de la bibliothèque standard de *C* `<stdlib.h>`. Comme arguments elle reçoit le nom de la commande (et peut-être une liste d'arguments) entre guillemets.

Listing 3.1 – system.c

```
#include <stdlib.h>

int main()
{
  int return_value;
  /* retourne 127 si le shell ne peut pas s'executer
     retourne -1 en cas d'erreur
     autrement retourne le code de la commande */
  return_value = system("ls -l /");
  return return_value;
}
```

Il crée un processus fils en lançant la commande :

```
ls -l /
```

dans un shell. Il faut retenir que `system()` n'est pas un appel système, mais une fonction *C*. Ce qui rend l'utilisation de la fonction `system()` moins performante qu'un appel système de création de processus. Ceci sera discuté à la fin de la section.

3.3.2 Création de processus avec `fork()`

Dans le cas d'Unix, l'appel système `fork()` est la seule véritable façon qui permet de créer des processus fils :

```
#include <unistd.h>
int fork();
```

`fork()` est le seul moyen de créer des processus, par duplication d'un processus existant[1]. L'appel système `fork()` crée une copie exacte du processus original, comme illustré à la figure **??**. Mais maintenant il se pose un problème, car les deux processus père et fils exécutent le même code. Comment distinguer alors le processus père du processus fils ? Pour résoudre ce

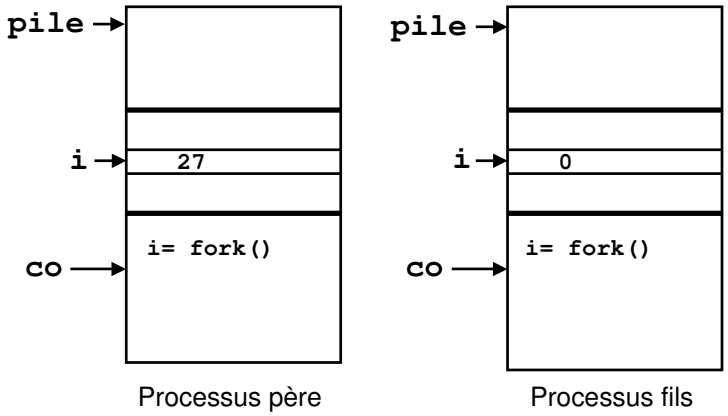

FIG. 3.5 – Processus père et son fils cloné. `i=27` après l'appel `fork()` pour le père et `i=0` pour le fils. L'image mémoire est la même pour tous les deux, notamment le compteur ordinal `co`.

problème, on regarde la valeur de retour de `fork()`, qui peut être :

- **0** pour le processus fils
- **Strictement positive** pour le processus père et qui correspond au `pid` du processus fils
- **Négative** si la création de processus a échoué, s'il n'y a pas suffisamment d'espace mémoire ou si bien le nombre maximal de créations autorisées est atteint.

Ce branchement est montré à la figure **??**. Le père et le fils ont chacun leur propre image mémoire privée. Après l'appel système `fork()`, la valeur de `pid` reçoit la valeur `0` dans le processus fils mais elle est égale à l'identifiant du processus fils dans le processus père.

Observez attentivement les exemples suivants :

---

[1]Linux a introduit un autre appel système pour la création de contexte d'un processus : `clone()`. C'est un appel système très puissant, avec un nombre d'arguments pour gérer la mémoire par le programmeur. Malheureusement `clone()` ne fait pas partie de la norme Posix, et ne sera pas étudié ici. Le lecteur intéressé peut consulter, par exemple, les articles du Linux Journal, http ://www.linuxjournal.com

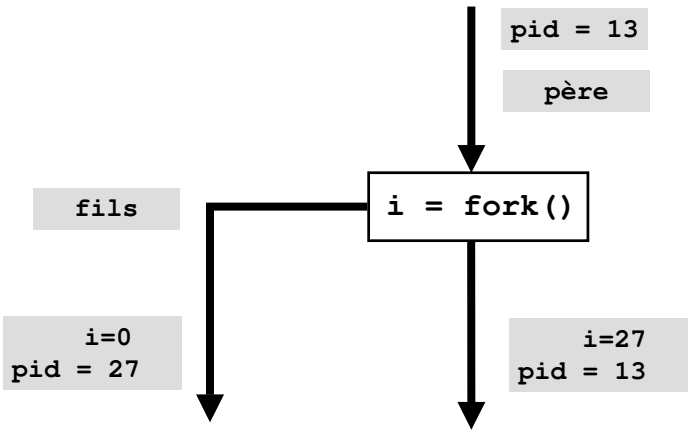

FIG. 3.6 – Services Posix : `fork()`.

► **Exemple 1.** Après l'exécution de `fils.c` le père et le fils montrent leur `pid`:

Listing 3.2 – fils.c

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

int main()
{
 pid_t fils_pid;
 fils_pid=fork();

 if (fils_pid==0)
   printf("Je suis le fils avec pid %d\n",getpid());
 else if(fils_pid > 0)
   printf("Je suis le pere avec pid %d\n", getpid());
 else
   printf("Erreur dans la creation du fils\n");
}
```

L'exécution de `fils.c` montre les `pid` du père et du fils :

```
leibnitz> gcc -o fils fils.c
leibnitz> fils
Je suis le pere avec pid 4130
leibnitz> Je suis le fils avec pid 4131
```

Remarquez que la console a réaffiché l'invite (`leibnitz>`) avant l'affichage du second message. Ceci s'explique par le fait que le processus père, qui a été lancé à partir de la console, s'est terminé avant le processus fils, et a donc redonné le contrôle à la console avant que le fils ne puisse afficher son message.

▶ **Exemple 2.** Étudiez en détail le programme suivant.

Listing 3.3 – chaine.c

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

int main(void)
{
  int     i, j, k,
          n=5;
  pid_t   fils_pid;

 for (i=1; i<n; i++)
 {
   fils_pid = fork();

   if (fils_pid > 0) // c'est le pere
      break;

   printf("Processus %d avec pere %d\n",getpid(), getppid());
 }

 // Pour faire perdre du temps au processus
 for (j=1; j<100000; j++)
   for(k=1; k<1000; k++);

}
```

Ce programme créera une chaîne de $n-1$ processus :

```
leibnitz> gcc -o chaine chaine.c
leibnitz> chaine
Processus 23762 avec pere 23761
leibnitz> Processus 23763 avec pere 23762
Processus 23764 avec pere 23763
Processus 23765 avec pere 23764
```

Si l'on omet les boucles finales du programme, on obtient un résultat inattendu :

```
leibnitz> gcc -o chaine chaine.c
leibnitz> chaine
leibnitz> Processus 23778 avec pere 1
Processus 23779 avec pere 1
Processus 23780 avec pere 1
Processus 23781 avec pere 1
```

Le problème ici est que dans tous les cas le processus père se termine avant le processus fils. Comme tout processus doit avoir un parent, le processus orphelin est donc "adopté" par le processus 1 (le processus **init**). Ainsi, on peut remarquer qu'au moment où les processus affichent le numéro de leur père, ils ont déjà éte adoptés par le processus init.

▶ **Exemple 3.** Le programme `tfork.cpp` suivant créera deux processus qui vont modifier la variable `a` :

Listing 3.4 – tfork.cpp

```
#include <sys/types.h>  // pour le type pid_t
#include <unistd.h>     // pour fork
#include <stdio.h>      // pour perror, printf


int main(void)
{
    pid_t p ;
    int   a = 20;

    // écration d'un fils
    switch (p = fork())
    {
        case -1:
            // le fork a echoue
            perror("le fork a echoue !") ;
            break;
        case 0 :
            // Il s'agit du processus fils
            printf("ici processus fils, le PID %d.\n", getpid());
            a += 10;
            break;
        default :
            // Il s'agit du processus pere
            printf("ici processus pere, le PID %d.\n", getpid());
            a += 100;
    }
    // les deux processus executent cette instruction
    printf("Fin du processus %d avec a = %d.\n", getpid(), a);
```

```
30      return 0;
}
```

Deux exécutions du programme `tfork.cpp` montrent que les processus père et fils sont concurrents :

```
leibnitz> gcc -o tfork tfork.cpp
leibnitz> tfork
ici processus pere, le pid 12339.
ici processus fils, le pid 12340.
Fin du Process 12340 avec a = 30.
Fin du Process 12339 avec a = 120.
leibnitz>
leibnitz> tfork
ici processus pere, le pid 15301.
Fin du Process 15301 avec a = 120.
ici processus fils, le pid 15302.
Fin du Process 15302 avec a = 30.
leibnitz>
```

▶ **Exemple 4.** Les exemples suivants meritent d'être etudiés en détail. Quelle est la différence entre les programmes `fork1.c` et `fork2.c` suivants ? Combien de fils engendreront-ils chacun ?

Listing 3.5 – fork1.c

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

 int main()
 {
  int i,n=5;
  int childpid;

10  for (i=1; i<n; i++)
  {
    if ((childpid=fork())<=0) break;
       printf("Processus %d avec pere %d, i=%d\n",getpid(),
              getppid(),i);
  }
  return 0;
 }
```

Listing 3.6 – fork2.c

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

int main()
{
 int i,n=5;
 pid_t pid;

  for (i=1; i<n; i++)
  {
   if ((pid=fork())==-1)
     break;
   if (pid == 0)
      printf("Processus %d avec pere %d, i=%d\n",getpid(),
             getppid(),i);
  }

  return 0;
}
```

La sortie de `fork1.c` est assez comprehensible. À chaque retour de l'appel de fork(), on sort de la boucle si on est dans un processus fils (valeur de retour égale à 0), ou si l'appel a échoué (valeur de retour négative). Si le shell, qui le père du processus crée lors de l'exécution de `fork1`, a un `pid` de 759, et le processsus lui-même un `pid` = 904 , on aura la sortie suivante :

```
leibnitz> gcc -o fork2 fork2.c
leibnitz> fork2
Processus 904 avec pere 759, i=1
Processus 904 avec pere 759, i=2
Processus 904 avec pere 759, i=3
Processus 904 avec pere 759, i=4
```

Par contre, le fonctionnement de `fork2.c` est plus complexe. Si l'appel à fork() n'échoue jamais, on sait que le premier processus créera 4 nouveaux processus fils. Le premier de ces fils continue d'exécuter la boucle à partir de la valeur courante du compteur i. Il créera donc lui-même trois processus. Le second fils en créera deux, et ainsi de suite. Et tout cela se répète récursivement avec chacun des fils. En supposant que le shell a toujours un `pid` de 759, et que le processus associé à `fork2` a un pid de 874, nous devrions obtenir l'arbre illustré à la figure **??**. Malheureusement, ce ne sera

pas le cas, puisqu'encore une fois des processus pères meurent avant que leur fils ne puissent afficher leurs messages :

```
leibnitz> gcc -o fork2 fork2.c
leibnitz>  fork2
Processus 23981 avec pere 23980, i=1
Processus 23982 avec pere 23980, i=2
Processus 23983 avec pere 23980, i=3
leibnitz> Processus 23984 avec pere 1, i=4
Processus 23985 avec pere 23981, i=2
Processus 23986 avec pere 23981, i=3
Processus 23987 avec pere 1, i=4
Processus 23988 avec pere 23982, i=3
Processus 23989 avec pere 1, i=4
Processus 23990 avec pere 1, i=4
Processus 23991 avec pere 23985, i=3
Processus 23992 avec pere 1, i=4
Processus 23993 avec pere 1, i=4
Processus 23994 avec pere 1, i=4
Processus 23995 avec pere 1, i=4
```

Nous verrons à la prochaine section comment éviter ce problème.

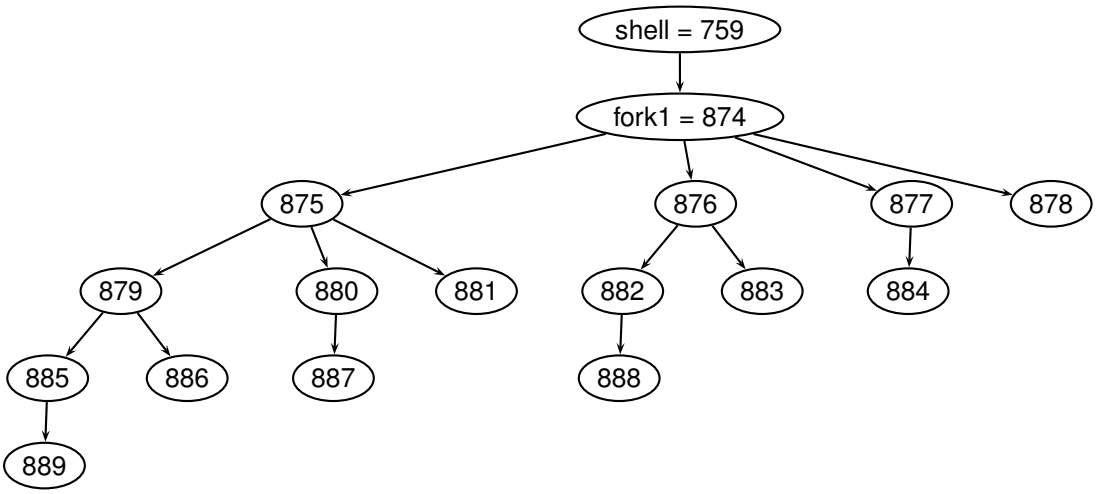

FIG. 3.7 – Arbre que nous obtiendrions avec `fork2.c` si les processus pères ne mourraient pas avant que les fils aient pu afficher leurs messages.

3.3.3 Les appels système `wait()`, `waitpid()` et `exit()`

Ces appels système permettent au processus père d'attendre la fin d'un de ses processus fils et de récupérer son status de fin. Ainsi, un processus peut synchroniser son exécution avec la fin de son processus fils en exécutant l'appel système `wait()`. La syntaxe de l'appel système est :

```
pid = wait(status);
```

où `pid` est l'identifiant du processus fils et `status` est l'adresse dans l'espace utilisateur d'un entier qui contiendra le status de `exit()` du processus fils.-

```
#include <sys/wait.h>
int wait (int *status);
int waitpid(int pid, int *status, int options);
void exit(int return_code);
```

- `wait()` : Permet à un processus père d'attendre jusqu'à ce qu'un processus fils termine. Il retourne l'identifiant du processus fils et son état de terminaison dans `&status`.
- `waitpid()` : Permet à un processus père d'attendre jusqu'à ce que le processus fils numéro `pid` termine. Il retourne l'identifiant du processus fils et son état de terminaison dans `&status`.
- `void exit()` : Permet de finir volontairement l'exécution d'un processus et donne son état de terminaison. Il faut souligner qu'un processus peut se terminer aussi par un arrêt forcé provoqué par un autre processus avec l'envoi d'un **signal**[2] du type `kill()`. Une description détaillée de `kill()` et d'autres appels système se trouve dans l'Annexe **??**.

Le processus appelant est mis en attente jusqu'à ce que l'un de ses fils termine. Quand cela se produit, il revient de la fonction. Si `status` est différent de 0, alors 16 bits d'information sont rangés dans les 16 bits de poids faible de l'entier pointé par `status`. Ces informations permettent de savoir comment s'est terminé le processus selon les conventions suivantes :

- Si le fils est stoppé, les 8 bits de poids fort contiennent le numéro du signal qui a arrêté le processus et les 8 bits de poids faible ont la valeur octale `0177`.
- Si le fils s'est terminé avec un `exit()`, les 8 bits de poids faible de `status` sont nuls et les 8 bits de poids fort contiennent les 8 bits de

[2] Les signaux et d'autres types de *Communication Interprocessus* seront étudiés au Chapitre **??**.

poids faible du paramètre utilisé par le processus fils lors de l'appel de `exit()`.
- Si le fils s'est terminé sur la réception d'un signal, les 8 bits de poids fort de `status` sont nuls et les 7 bits de poids faible contiennent le numéro du signal qui a causé la fin du processus.

Dans le fichier d'en-tête `<sys/wait.h>` sont definis différents macros qui permettent d'analyser la valeur de status afin de determiner la cause de terminaison du processus. En particulier, le processus père peut obtenir la valeur des 8 bits de poids faible du paramètre qui reçoit depuis `exit()` de la part du fils en utilisant le macro :

`WEXITSTATUS(status)`

La définition de ce macro se trouve dans `<sys/wait.h>` :

```
#define WEXITSTATUS(s) (((s)>>8)&0xFF)
```

Evidemment, si le processus appelant `wait()` n'a pas de processus fils vivants, une erreur est produite.

▶ **Exemple 5.** Rappelons-nous que le programme `fils.c` a le défaut de revenir au shell avant que le fils n'ait pu affiher son message. En utilisant l'appel `wait`, on peut facilement corriger ce problème :

Listing 3.7 – wait.c

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

int main()
{
 pid_t fils_pid;
 fils_pid=fork();

 if (fils_pid==0)
   printf("Je suis le fils avec pid %d\n",getpid());
 else if(fils_pid > 0) {
   printf("Je suis le pere avec pid %d.\n", getpid());
   printf("J'attends que mon fils se termine\n");
   wait(NULL);
 }

 else
   printf("Erreur dans la creation du fils\n");

 exit(0);
}
```

▶ **Exemple 6.** Pour éviter le problème des processus pères qui meurent avant leurs fils dans le programme `fork2.c`, il suffit d'ajouter une boucle sur l'appel `wait`. Dès qu'il ne restera plus aucun fils, l'appel retournera aussitôt une valeur négative. On saura alors que l'on peut sortir sans problème. Voici le programme modifié :

Listing 3.8 – fork2b.c

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

int main()
{
 int i,n=5;
 int pid;

  for (i=1; i<n; i++)
  {
   if ((pid=fork())==-1)
     break;
   if (pid == 0)
      printf("Processus %d avec pere %d, i=%d\n",getpid(),
             getppid(),i);
  }

  // Attendre la fin des fils
  while (wait(NULL) >= 0);

  return 0;
}
```

▶ **Exemple 7.** Création de deux fils (version 1), et utilisation des 8 bits de poids fort :

Listing 3.9 – deuxfils-1.c

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

void fils(int i);
int main()
{
```

```
    int status;
    if(fork()) // creation du premier fils
        {
        if(fork()==0) // creation du second fils
           fils(2);
        }
    else fils(1);
    if(wait(&status)>0)
        printf("fin du fils %d\n", status>>8);
    if (wait(&status)>0)
        printf("fin du fils %d\n", status>>8);
    return 0;
}

void fils(int i)
{
   sleep(2);
   exit(i);
}
```

Deux exécutions du programme `deuxfils-1.c` :

```
leibnitz> gcc -o deuxfils-1 deuxfils-1.c
leibnitz> deuxfils-1
fin du fils 2
fin du fils 1
leibnitz> deuxfils-1
fin du fils 1
fin du fils 2
leibnitz>
```

▶ **Exemple 8.** Création de deux fils (version 2), avec la simulation d'attente d'un troisième fils inexistant et l'utilisation du macro `WEXITSTATUS` :

Listing 3.10 – deuxfils-2.c

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

void fils(int i);
int main()
{
  int   status;
  pid_t pid;
```

```
  // Creation du premier fils
  pid = fork();
  if (pid == -1) perror("fork");
  else if (pid == 0) fils(1);

  // Creation du second fils
  pid = fork();
  if (pid == -1) perror("fork");
  else if (pid == 0) fils(2);

  // Attente de terminaison des deux fils

  if(wait(&status)>0)
    printf("fin du fils %d\n", WEXITSTATUS(status)) ;
  else perror("wait");

  if(wait(&status)>0)
    printf("fin du fils %d\n", WEXITSTATUS(status)) ;
  else perror("wait");

  // Ici on attend la fin d'un troisieme fils qui n'existe pas
  // Une erreur sera retournee
  if (wait(&status)>0)
    printf("fin du fils %d\n", WEXITSTATUS(status)) ;
  else perror("wait");

  return 0;
}

// Routine pour les fils
void fils(int i)
{
    sleep(2);
    exit(i);
}
```

Exécution du programme `deuxfils-2.c` :

```
leibnitz> gcc -o deuxfils-2 deuxfils-2.c
leibnitz> deuxfils-2
fin du fils 2
fin du fils 1
wait: No child processes
leibnitz>
```

3.3.4 Processus zombie

L'exécution asynchrone entre processus parent et fils a certaines conséquences. Souvent, le fils d'un processus se termine, mais son père ne l'attend pas. Le processus fils devient alors un **processus zombie**, comme illustré à la figure **??**. Le processus fils existe toujours dans la table des processus, mais il n'utilise plus les ressources du *kernel*.

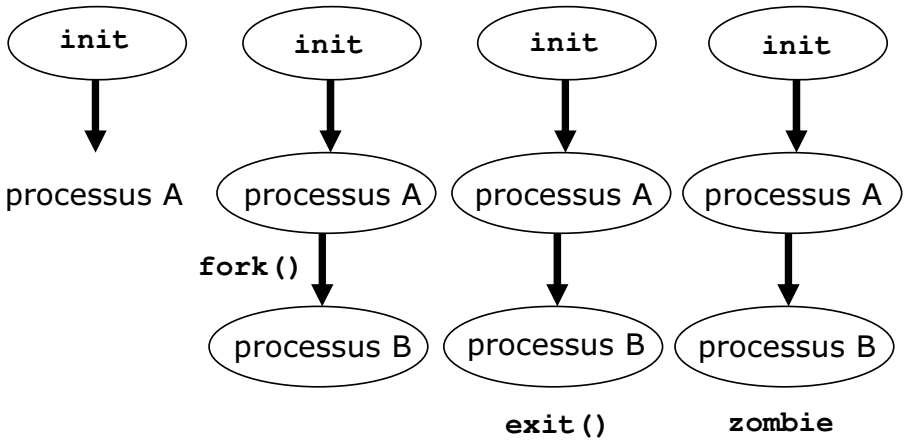

FIG. 3.8 – Processus zombie : le fils meurt et le père n'a pas fait `wait()`.

L'exécution de l'appel système `wait()` ou `waitpid()` par le processus père élimine le fils de la table des processus. Il peut y avoir aussi des terminaisons prématurées, où le processus parent se termine ses processus fils (figure **??**). Dans cette situation, ses processus fils sont adoptés par le processus `init`, dont le `pid` = 1.

▶ **Exemple 9.** Engendrer des zombies est relativement facile. Le programme `zombie.c` montre la création d'un processus fils zombie pendant 30 secondes :

Listing 3.11 – zombie.c

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

int main()
{
   pid_t pid;
```

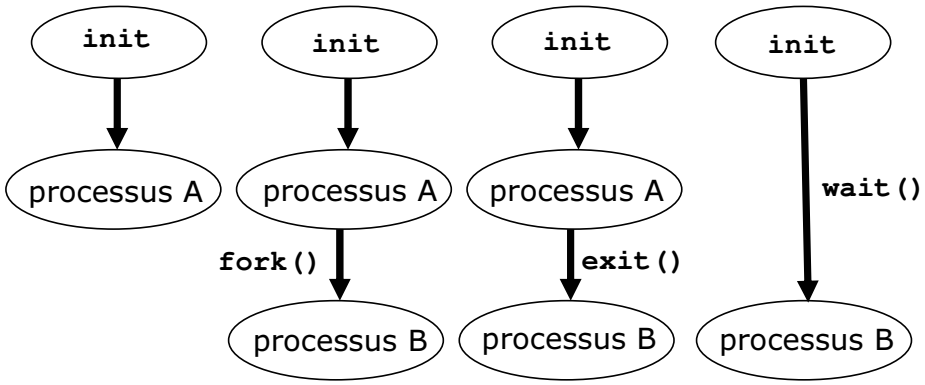

FIG. 3.9 – Processus orphelin : le père meurt avant le fils. `init` adopte l'enfant.

```
    // processus fils
    pid = fork();
    if (pid > 0 )
    {
      // Pere : dormir 30 secondes
      sleep(30);
    }
    else
    {
      // Fils : quitter immediatement
       exit (0);
    }
    return 0;
}
```

Exécution de `zombie.c` en tâche de fond : la commande `ps` permet de constater son état zombie avant et après son disparition définitive :

```
leibnitz> gcc -o zombie zombie.c
leibnitz> zombie &
[1] 5324
leibnitz> ps -u jmtorres
  PID TTY          TIME CMD
 2867 pts/0    00:00:00 tcsh
 5324 pts/0    00:00:00 zombie
 5325 pts/0    00:00:00 zombie <defunct>
 5395 pts/0    00:00:00 ps
leibnitz>
```

```
[1]    Done                          zombie
leibnitz>
leibnitz> ps -u jmtorres
  PID TTY          TIME CMD
 2867 pts/0    00:00:00 tcsh
 5396 pts/0    00:00:00 ps
leibnitz>
```

3.3.5 La famille des appels système `exec`

Un **processus fils** créé peut remplacer son code de programme par un autre programme. Le système Unix offre une famille d'appels système `exec` qui permettent de changer l'image d'un processus (figure **??**). Tous les appels système `exec` remplacent le processus courant par un nouveau processus construit à partir d'un fichier ordinaire exécutable. Les segments de texte et de données du processus sont remplacés par ceux du fichier exécutable.

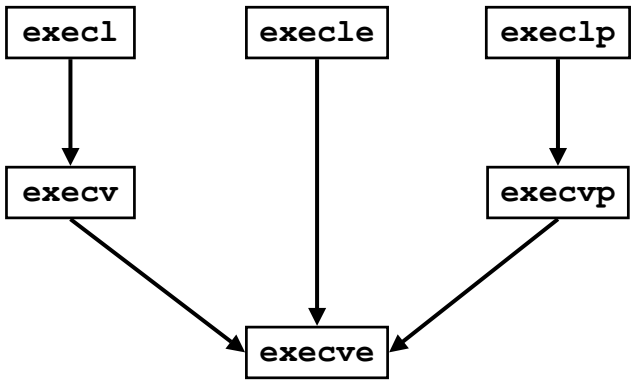

FIG. 3.10 – Services Posix : `exec`.

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

- `execl()` : permet de passer un nombre fixé de paramètres au nouveau programme.
- `execv()` : permet de passer un nombre libre de paramètres au nouveau programme.
- `execle()` : même fonctionnement qu'`execl()` avec en plus, un argument `envp` qui représente un pointeur sur un tableau de pointeurs sur des chaînes de caractères définissant l'environnement.
- `exelp()` : Interface et action identiques à celles d'`execl()`, mais la différence vient du fait que si le nom du fichier n'est pas un nom complet — par rapport à la racine — le système utilisera le chemin de recherche des commandes — les chemins indiqués par la variable `PATH` — pour trouver dans quel répertoire se trouve le programme.
- `exevp()` : Interface et action identiques à celles d'`execv()`, mais la différence vient du fait que si le nom de fichier n'est pas un nom complet, la commande utilise les répertoires spécifiés dans `PATH`.

La convention Unix veut que chaque chaîne ait la forme `nom=valeur`. Ainsi, les arguments pour `execv()` peuvent être passés, par exemple :

```
char *arguments[4]
...
 arguments[0]="/bin/ls";
 arguments[1]="-l";
 arguments[2]="/etc";
 arguments[3]="NULL";
 execv("/bin/ls", arguments);
...
```

Après l'exécution d'un appel système de la famille `exec()`, l'image mémoire d'un processus est écrasée par la nouvelle image mémoire d'un programme exécutable, mais le `pid` ne change pas. Le contenu du contexte utilisateur qui existait avant l'appel à `exec()` n'est plus accessible.

► **Exemple 10.** Le programme `fork-exec.c` montre l'utilisation des appels système `fork()` et `execvp()` :

Listing 3.12 – fork-exec.c

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

int main(int argc, char* argv[])
{
```

```
    pid_t fils_pid;

    // Liste d'arguments pour la comande "ls"
    char* args[] = {"ls", "-l", argv[1], NULL};

    // Dupliquer le processus
    fils_pid = fork ();
    if (fils_pid != 0) {
      // Il s'agit du pere
      waitpid(fils_pid,NULL,NULL);
      printf("Programme principal termine\n");
      return 0;
    }
    else
    {
      // Executer programme avec les arguments a partir du PATH
      execvp("ls", args);
      // Retourner en cas d'erreur
      perror("Erreur dans execvp");
      exit(1);
    }

    return 0;
}
```

Exécution de `fork-exec.cpp` :

```
leibnitz> gcc -o fork-exec fork-exec.cpp
leibnitz> fork-exec /
total 113
drwxr-xr-x    2 root  bin     4096 Aug 22 12:00 bin
drwxr-xr-x    2 root  root    4096 Jan 20 15:46 boot
drwxr-xr-x    2 root  root    4096 Aug 22 16:43 cdrom
drwxr-xr-x   14 root  root   40960 Jan 20 16:10 dev
drwxr-xr-x   33 root  root    4096 Jan 22 15:55 etc
drwx------    2 root  root   16384 Aug 22 11:18 lost+found
drwxr-xr-x    5 root  root    4096 Mar 16  2002 mnt
...
drwxrwxrwt    8 root  root    4096 Jan 22 16:44 tmp
drwxr-xr-x   20 root  root    4096 Jan 20 16:04 usr
drwxr-xr-x   17 root  root    4096 Aug 22 16:43 var
Programme principal termine
leibnitz>
```

3.3.6 Commentaire sur `system()`

L'exécution de la fonction de bibliothèque `system()` est essentiellement un `fork()` + `exec()` + `wait()`. Et elle est peut être encore pire que cela, car elle invoque un shell pour exécuter des commandes, ce qui est très coûteux en ressources, car le shell doit charger des scripts et des initialisations. Bien que `system()` puisse s'avérer utile pour des petits programmes qui s'exécutent très occasionnellement, on la déconseille fortement pour d'autres applications. Par exemple, le seul appel :

```
system("ls -l /usr")
```

est un gaspillage incroyable de ressources. D'abord on effectue un `fork()`, puis un `exec()` pour le shell. Puis le shell s'initialise et effectue un autre `fork()` et un `exec()` pour lancer `ls`. Il est préférable d'utiliser des appels système pour la lecture des répertoires (voir par exemple le Chapitre **??** *Système de fichiers*, Section **??** *Services Posix sur les répertoires.*) qui sont moins coûteux que la fonction `system()`.

▶ **Exemple 11.** Les programmes `parent.cpp` et `fils.cpp` montrent la création des processus avec la combinaison `fork()` et `execl()` :

Listing 3.13 – parent.cpp

```
#include <unistd.h>     // pour fork et execl
#include <sys/wait.h>   // pour wait
#include <stdio.h>      // pour printf

int main(void)
{
   int p, child, status;

   p=fork();
   if( p == -1)
      return -1;
   if( p>0) // il s'agit du pere car p > 0
   {
     printf ("Ici le pere [%d], mon fils[ %d]\n", getpid(), p);
     if((child=wait(&status))>0)
       printf("Ici pere [%d], fin du fils[%d]\n",getpid(),child);
                // Dormir pendant une seconde
     sleep(1);
     printf("Le pere [%d] se termine\n", getpid());
   }
   else
   { // il s'agit du fils
     if((status=execl(
                "/home/ada/users/jmtorres/inf3600/logiciel/a.out",
```

```
                "a.out",
                NULL))==-1)
      printf("desole le programme n'existe pas : %d\n",status);
    else
      printf("cette instruction n'est jamais executee\n");
  }
  return 0;
}
```

Listing 3.14 – fils.cpp

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

int main()
{
   printf("ici programme fils [%d] \n",getpid());
   return 0;
}
```

Exécution des programmes `parent.cpp` et `fils.cpp`. Remarque : Le fichier exécutable de `fils.cpp` est `a.out` :

```
leibnitz> gcc fils.cpp
leibnitz> gcc -o parent.cpp
leibnitz> parent
Ici le pere 19909, mon fils 19910
ici programme fils 19910
Ici pere 19909, fin du fils 19910
Le pere 19909 se termine
leibnitz>
```

▶ **Exemple 12.** Utilisation d'`execvp()` :

Listing 3.15 – texecvp.c

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

int main (int argc , char * argv[])
{
  if ( fork() == 0){
    execvp( argv[1], &argv[1]) ;
    fprintf(stderr, "on ne peut executer %s\n ", argv[1]);
```

```
10    }
      else if( wait(NULL)>0 )
        printf(" Le pere detecte la fin du fils\n");
      return 0;
}
```

Exécution du programme `texecvp.cpp` :

```
leibnitz> gcc -o texecvp texecvp.cpp
leibnitz> texecvp date
Wed Sep  4 15:42:06 EDT 2002
 Le pere detecte la fin du fils
leibnitz> texecvp ugg+kju
on ne peut pas executer ugg+kju
  Le pere detecte la fin du fils
leibnitz> gcc -o fils fils.cpp
leibnitz> fils
 ici programme fils [20498]
leibnitz> texecvp fils
 ici programme fils [20500]
 Le pere detecte la fin du fils
leibnitz>
```

3.4 Exercices

1. Dans le système Unix, est-ce que tout processus a un père ?
2. Que se passe-t-il lorsqu'un processus devient orphelin (mort de son père) ?
3. Quand est-ce qu'un processus passe à l'état **zombie** ?
4. Pour lancer en parallèle plusieurs traitements d'une même application, vous avez le choix entre les appels système `pthread_create()` et `fork()`. Laquelle des deux possibilités choisir ? Pourquoi ?
5. Qu'affiche chacun des segments de programme suivants :

(a)
```
for (i=1; i<=4; i++)
{
    pid = fork();
    if (pid != 0) printf("%d\n", pid);
}
```

(b)
```
for (i=1; i<=4; i++)
{
    pid = fork();
    if (pid == 0) break;
    else printf("%d\n", pid);
}
```

(c)
```
for (i=0; i<nb; i++)
{
    p = fork();
    if (p < 0) exit(1);
    execlp("prog", "prog",NULL);
}
wait(&status);
```

(d)
```
for (i=1; i<=nb; i++)
{
    p1 = fork();
    p2 = fork();
    if (p1 < 0) exit(1);
    if (p2 < 0) exit(1);
    execlp("prog1", "prog1",NULL);
    execlp("progc", "prog",NULL);
}
wait(&status);
```

(e) Quel est le nombre de processus créés, dans chaque cas ?

6. Écrire un programme qui lit à l'écran le nombre de fils à créer, puis les crée l'un à la suite de l'autre. Chaque fils affiche à l'écran son `pid` (`getpid()`) et celui de son père (`getppid()`). Le processus créateur doit attendre la fin de ses fils. Lorsqu'il détecte la fin d'un fils, il affiche le `pid` du fils qui vient de se terminer.
7. Écrire un programme qui lance en créant un processus le programme `prog2.cpp`, puis se met en attente de la fin d'exécution du programme.
8. Expliquez un exemple d'exécution de programme dans lequel il existe de l'incohérence entre les trois niveaux de mémoire, *i.e.* registres, mémoire principale, et mémoire secondaire.
9. Lorsqu'un nouveau processus passe en état d'exécution, tous les registres de l'UCT doivent être initialisés ou restaurés avec les valeurs au moment de l'interruption du processus. Expliquez la raison pour laquelle le registre compteur ordinal (i.e. *program counter*) est le dernier à être initialisé.
10. Quelle différence existe-il entre un programme exécutable et un processus ?
11. Est-ce qu'un processus peut exécuter plusieurs programmes, et est-ce qu'un programme peut être exécuté par plusieurs processus à la fois ? Expliquez.
12. Considérez un système ayant les caractéristiques suivantes :
    (a) Système d'exploitation multiprogrammé.
    (b) Un seul UCT.
    (c) Nombre limité de ressources.
    (d) Un processus peut quitter volontairement l'état d'exécution pour revenir à l'état prêt.
    (e) Le système peut bloquer à tout moment un processus en état d'exécution s'il a besoin des ressources utilisées par ce processus.

    Dessinez le diagramme d'état de ce système.