Chapitre 2

Les processus

1. Structure des processus

1.1. Généralités

Les processus correspondent à l'exécution de tâches : les programmes des utilisateurs, les entrées-sorties,… par le système. Un système d'exploitation doit en général traiter plusieurs tâches en même temps. Comme il n'a, la plupart du temps, qu'un processeur, il résout ce problème grâce à un pseudo-parallélisme. Il traite une tâche à la fois, s'interrompt et passe à la suivante. La commutation des tâches étant très rapide, l'ordinateur donne l'illusion d'effectuer un traitement simultané.

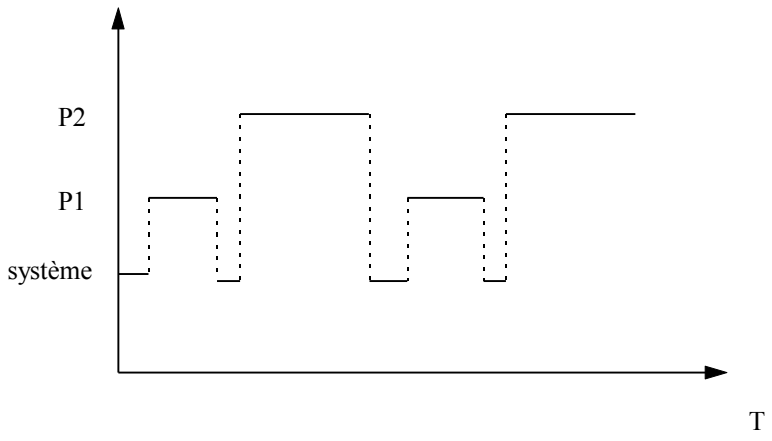

Figure 1 Le multi-tâche

Les processus des utilisateurs sont lancés par un interprète de commande. Ils peuvent eux-mêmes lancer ensuite d'autres processus. On appelle le processus créateur, le père, et les processus

créés, les fils. Les processus peuvent donc se structurer sous la forme d'une arborescence. Au lancement du système, il n'existe qu'un seul processus, qui est l'ancêtre de tout les autres.

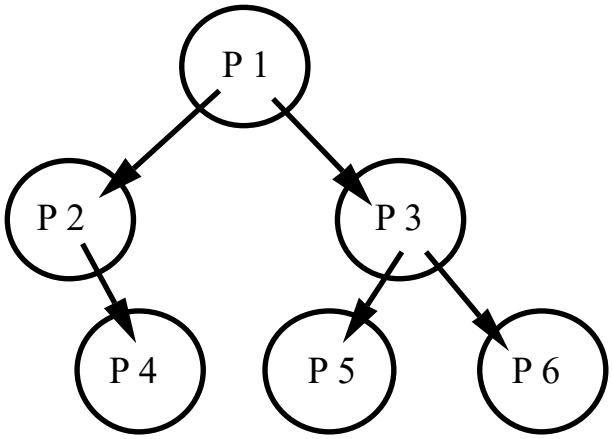

Figure 2 Le hiérarchie des processus.

Les processus sont composés d'un espace de travail en mémoire formé de 3 segments : la pile, les données et le code et d'un contexte.

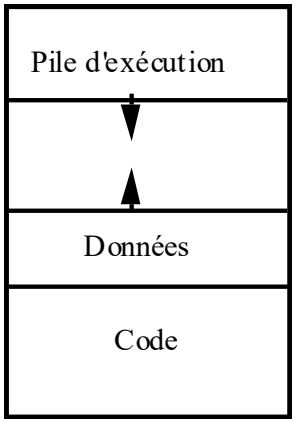

Figure 3 Les segments d'un processus.

Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter. La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire. Enfin, les appels de fonctions, avec leurs paramètres et leurs variables

locales, viennent s'empiler sur la pile. Les zones de pile et de données ont des frontières mobiles qui croissent en sens inverse lors de l'exécution du programme. Parfois, on partage la zone de données en données elles-mêmes et en tas. Le tas est alors réservé au données dynamiques.

Le contexte est formé des données nécessaires à la gestion des processus. Une table contient la liste de tous les processus et chaque entrée conserve leur contexte[1]. Les éléments de la table des processus de Minix, par exemple, ont la forme simplifiée du tableau ci-dessous.

Processus	Mémoire	Fichiers
Registres	Pointeur sur code	Masque `umask`
Compteur ordinal	Pointeur sur données	Répertoire racine
État du programme	Pointeur sur pile	Répertoire de travail
Pointeur de pile	Statut de fin d'exécution	Descripteurs de fichiers
Date de création	N° de signal du proc. tué	uid effectif
Temps CP utilisé	PID	gid effectif
Temps CP des fils	Processus père	
Date de la proch. alarme	Groupe de processus	
Pointeurs sur messages	uid réel	
Bits signaux en attente	uid effectif	
PID	gid réel	
	gid effectif	
	Bits des signaux	

Tableau 1 Les éléments du contexte de Minix.

Le nombre des emplacements dans la table des processus est limité pour chaque système et pour chaque utilisateur.

La commutation des tâches, le passage d'une tâche à une autre, est réalisée par un ordonnanceur au niveau le plus bas du système. Cet ordonnanceur est activé par des interruptions :

- d'horloge,
- de disque,
- de terminaux.

[1] Bach, op. cit., p. 158, et Tanenbaum, op. cit., p. 60.

À chaque interruption correspond un vecteur d'interruption, un emplacement mémoire, contenant une adresse. L'arrivée de l'interruption provoque le branchement à cette adresse. Une interruption entraîne, en général, les opérations suivantes :

- empilement du compteur ordinal, du registre d'état et peut être d'autres registres. La procédure logicielle d'interruption sauvegarde les éléments de cette pile et les autres registres dans du contexte du processus en cours;
- mise en place d'une nouvelle pile permettant le traitement des interruptions;
- appel de l'ordonnanceur;
- élection d'un nouveau programme;

Explication de l'interruption d'horloge, puis du disque.

Nous avons vu maintenant qu'un processus pouvait être actif en mémoire centrale (Élu) ou suspendu en attente d'exécution (Prêt). Il peut aussi être Bloqué, en attente de ressource, par exemple au cours d'une lecture de disque. Le diagramme simplifié des états d'un processus est donc :

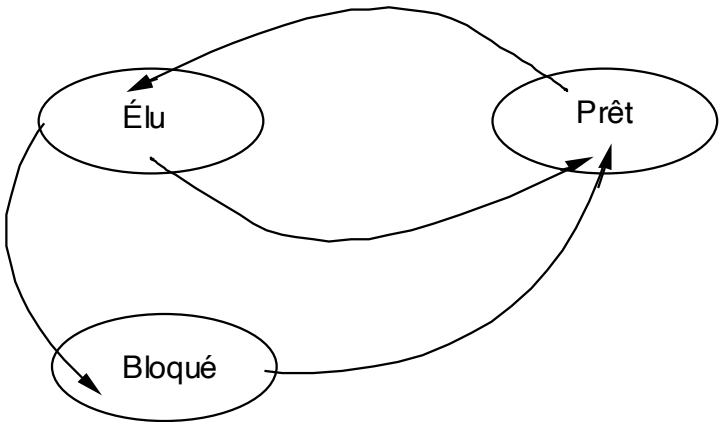

Figure 4 Les états d'un processus.

Le processus passe de l'état élu à l'état prêt et réciproquement au cours d'une intervention de l'ordonnanceur. Cet ordonnanceur se déclenchant, par exemple, sur une interruption d'horloge. Il

pourra alors suspendre le processus en cours, si il a dépassé son quantum de temps, pour élire l'un des processus prêts. La transition de l'état élu à l'état bloqué se produit, par exemple, à l'occasion d'une lecture sur disque. Ce processus passera de l'état bloqué à l'état prêt lors de la réponse du disque. Ce passage correspond d'une manière générale à la libération d'une ressource.

En fait, sous Unix, l'exécution d'un processus se fait sous deux modes, le mode utilisateur et le mode noyau. Le mode noyau correspond aux appels au code du noyau : `write`, `read`,… Le mode utilisateur correspond aux autres instructions.

Un processus en mode noyau ne peut être suspendu par l'ordonnanceur. Il passe à l'état préempté à la fin de son exécution dans ce mode – à moins d'être bloqué ou détruit –. Cet état est virtuellement le même que prêt en mémoire.

Par ailleurs, lorsque la mémoire ne peut contenir tous les processus prêts, un certain nombre d'entre eux sont déplacés sur le disque. Un processus peut alors être prêt en mémoire ou prêt sur le disque. De la même manière, on a des processus bloqués en mémoire ou bien bloqués sur disque.

Enfin, les processus terminés ne sont pas immédiatement éliminés de la tables des processus – ils ne le sont que par une instruction explicite de leur père –. Ceci correspond à l'état défunt.

Les états d'un processus Unix[2] sont alors :

[2]Bach, op.cit., p. 156.

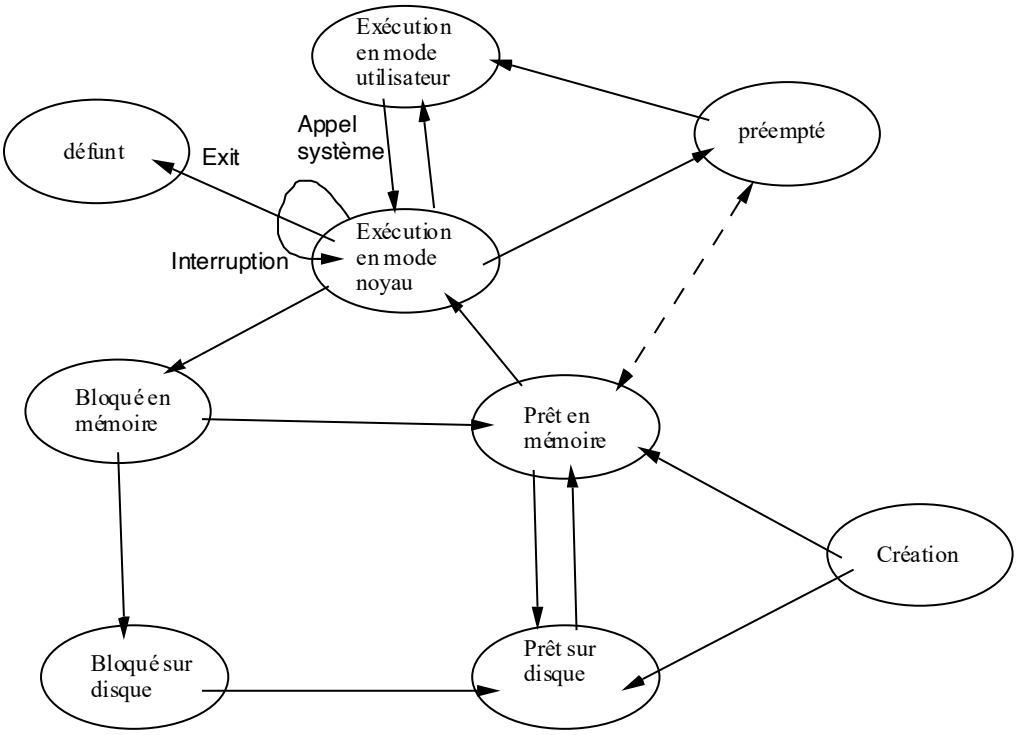

Figure 5 Les états des processus d'Unix.

1.2. Les processus sous Unix

Les fonctions fondamentales

Un processus peut se dupliquer – et donc ajouter un nouveau processus – par la fonction :

```
pid_t fork(void)
```

qui rend -1 en cas d'échec, 0 dans le processus fils, et le n° du processus fils (PID) dans le père. Cette fonction transmet une partie du contexte du père au fils : les descripteurs des fichiers standards et des autres fichiers, les redirections… Les deux programmes sont sensibles aux mêmes interruptions. À l'issue d'un `fork()` les deux processus s'exécutent simultanément.

La création d'un nouveau processus ne peut se faire que par le « recouvrement » du code d'un processus existant grâce à la fonction :

```
int execl(char *ref, char *arg0, char *argn, 0)
```

`ref` est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter. `arg0`, `arg1`, …, `argn` sont les arguments du programme. Le premier argument, `arg0`, reprend en fait le nom du programme.

Les fonctions `execle()` et `execlp()` ont des arguments et des effets semblables. `int execv(char *ref, char *argv[])` ainsi que `execve()` et `execvp()` sont analogues.

Ces fonctions rendent -1 en cas d'échec.

Par exemple :

```
void main()
{
   execl("/bin/ls", "ls", "-l", (char *) 0);
/* avec execlp, le premier argument peut n'être
que ls et non /bin/ls */
   printf("pas d'impression\n");

}
```

La création d'un nouveau processus – l'occupation d'un nouvel emplacement de la table des processus – implique donc la duplication d'un processus existant. Ce dernier sera le père. On substituera ensuite le code du fils par le code qu'on désire exécuter. Au départ, le processus initial lance tous les processus utiles au système.

Un processus se termine lorsqu'il n'a plus d'instructions ou lorsqu'il exécute la fonction :

```
void exit(int statut)
```

L'élimination d'un processus terminé de la table ne peut se faire que par son père, grâce à la fonction :

```
int wait(int * code_de_sortie)
```

Avec cette instruction, le père se bloque en attente de la fin d'un fils. Elle rendra le n° (PID) du premier fils mort trouvé. La valeur du code de sortie est reliée au paramètre d'`exit` de ce fils. On peut donc utiliser l'instruction `wait` pour connaître la valeur éventuelle de retour, fournie par

`exit()`, d'un processus. Ce mode d'utilisation est analogue à celui d'une fonction. `wait()` rend –1 en cas d'erreur.

Si le fils se termine sans que son père l'attende, le fils passe à l'état `defunct` dans la table. Si, au contraire, le père se termine avant le fils, ce dernier est rattaché au processus initial. Le processus initial passe la plupart de son temps à attendre les processus orphelins.

Grâce aux 3 instructions, `fork()`, `exec()`, et `wait()`, on peut écrire un interprète de commandes simplifié. Il prend la forme suivante :

```
while(1) {
                                    /* attend commande */
    lire_commande(commande, paramètres);
    if (fork() != 0) {
          wait(&statut);                /* proc. père */
    } else {
          execv(commande, paramètres);  /* proc. fils */
    }
}
```

Les signaux

L'instruction :

```
int kill(int pid, int signal)
```

permet à un processus d'envoyer un signal à un autre processus. `pid` est le n° du processus à détruire et `signal`, le n° du signal employé. La fonction `kill` correspond à des interruptions logicielles.

Par défaut, un signal provoque la destruction du processus récepteur, à condition bien sûr, que le processus émetteur possède ce droit de destruction.

La liste des valeurs de `signal` comprend, entre autres :

Nom du signal	N° du signal	Commentaires
`SIGUP`	1	signal émis lors d'une déconnexion

SIGINT	2	^C
SIGQUIT	3	^\
SIGKILL	9	signal d'interruption « radicale »
SIGALRM	? Les valeurs sont définies par des macro dans signal.h	signal émis par alarm(int sec) au bout de sec secondes
SIGCLD	?	signal émis par un fils qui se termine, à son père

Tableau 2 Quelques signaux d'Unix.

kill() rend 0 en cas de succès et -1 en cas d'échec.

Un processus peut modifier – « masquer » – son comportement aux signaux reçus par l'appel de la fonction :

```
void (*signal(int signal, void (* fonction)(int)))
```

avec fonction pouvant prendre les valeurs:

Nom	Action
SIG_IGN	le processus ignorera l'interruption correspondante
SIG_DFL	le processus rétablira son comportement par défaut lors de l'arrivée de l'interruption (la terminaison)
void fonction(int n)	le processus exécutera fonction, définie par l'utilisateur, à l'arrivée de l'interruption n. Il reprendra ensuite au point où il a été interrompu

Tableau 3 Les routines d'interruption d'Unix.

L'appel de la fonction signal positionne l'état des bits de signaux dans la table des processus.

Par ailleurs, la fonction signal(SIGCLD, SIG_IGN), permet à un père d'ignorer le retour de ses fils sans que ces derniers encombrent la table des processus à l'état defunct.

Quelques fonctions d'identification d'Unix

La commande Unix `ps -ef`, donne la liste des processus en activité sur le système. Chaque processus possède un identificateur et un groupe. On les obtient par les fonctions :

```
int getpid(void)
```

et

```
int getpgrp(void)
```

Les fils héritent du groupe de processus du père. On peut changer ce groupe par la fonction :

```
int setpgrp(void)
```

Chaque processus possède, d'autre part, un utilisateur réel et effectif. On les obtient respectivement par les fonctions :

```
int getuid(void)
```

et

```
int geteuid(void)
```

L'utilisateur réel est l'utilisateur ayant lancé le processus. Le numéro d'utilisateur effectif sera utilisé pour vérifier certains droits d'accès (fichiers et envoi de signaux). Il correspond normalement au numéro d'utilisateur réel. On peut cependant positionner l'utilisateur effectif d'un processus au propriétaire du fichier exécutable. Ce fichier s'exécutera alors avec les droits du propriétaire et non avec les droits de celui qui l'a lancé. La fonction :

```
int setuid(int euid)
```

permet de commuter ces numéros d'utilisateur de l'un à l'autre : réel à effectif et vice-versa. Elle rend 0 en cas de succès.

2. Ordonnancement

L'ordonnancement règle les transitions d'un état à un autre des différents processus. Cet ordonnancement a pour objectifs :

1. de maximiser l'utilisation du processeur;

2. d'être équitable entre les différents processus;

3. de présenter un temps de réponse acceptable;

4. d'avoir un bon rendement;

5. d'assurer certaines priorités;

2.1. Le tourniquet

Cet algorithme est l'un des plus utilisés et l'un des plus fiables. Chaque processus prêt dispose d'un quantum de temps pendant lequel il s'exécute. Lorsqu'il a épuisé ce temps ou qu'il se bloque, par exemple sur une entrée-sortie, le processus suivant de la file d'attente est élu et le remplace. Le processus suspendu est mis en queue du tourniquet.

Le seul paramètre important à régler, pour le tourniquet, est la durée du quantum. Il doit minimiser le temps de gestion du système et cependant être acceptable pour les utilisateurs. La part de gestion du système correspond au rapport de la durée de commutation sur la durée du quantum. Plus le quantum est long plus cette part est faible, mais plus les utilisateurs attendent longtemps leur tour. Un compromis peut se situer, suivant les machines, de 100 à 200 ms.

2.2. Les priorités

Dans l'algorithme du tourniquet, les quanta égaux rendent les différents processus égaux. Il est parfois nécessaire de privilégier certains processus par rapport à d'autres. L'algorithme de priorité choisit le processus prêt de plus haute priorité.

Ces priorités peuvent être statiques ou dynamiques. Les processus du système auront des priorités statiques (non-modifiables) fortes. Les processus des utilisateurs verront leurs priorités modifiées, au cours de leur exécution, par l'ordonnanceur. Ainsi un processus qui vient de s'exécuter verra sa priorité baisser. Pour un exemple d'exécution avec priorités, on pourra consulter Bach[3].

2.3. Le tourniquet avec priorités

[3] op. cit., p. 267-270.

On utilise généralement une combinaison des deux techniques précédentes. À chaque niveau de priorité correspond un tourniquet. L'ordonnanceur choisit le tourniquet non vide de priorité la plus forte et l'exécute.

Pour que tous les processus puissent s'exécuter, il est nécessaire d'ajuster périodiquement les différentes priorités.

2.4 L'ordonnancement des fils d'exécution

Les fils d'exécution sont soumis à un ordonnancement. Dans Windows NT, les fils d'exécution ont 32 niveaux de priorité qui sont soit fixes soit dynamiques. La priorité la plus haute est toujours celle qui s'exécute. Dans le cas d'une priorité dynamique, les valeurs de cette priorité varient entre deux bornes. Elle augmente, par exemple, lors d'une attente d'entrée-sortie. On peut changer la priorité des fils d'exécution dans Windows NT par la fonction `SetThreadPriority()`.

Avec Java, les niveaux de priorité varient entre `Thread.MIN_PRIORITY` et `Thread.MAX_PRIORITY`. La priorité normale est `Thread.NORM_PRIORITY`. Comme avec NT, la priorité la plus haute est toujours celle qui s'exécute. Certains interprètes utilisent un tourniquet qui gère les fils de même priorité sans limite de temps pour le fil qui s'exécute. D'autres interprètes limitent l'exécution d'un fils à un quantum de temps, puis passent à un autre fil d'une même priorité. Le modèle d'exécution peut être assez différent suivant les machines, par exemple entre l'IBM 580 et le Sun E3000.