Chapitre 4 : Gestion de Processus

La notion clé dans le fonctionnement des systèmes multiprogrammés est celle de processus.

Nous distinguons entre un programme et son exécution, c'est cette exécution qui est appelé **processus**. Donc : Un processus est un programme en cours d'exécution.

Un programme est une entité statique alors qu'un processus est une entité dynamique associée à la suite des actions (instructions) réalisées par un programme. Cette suite d'actions est indépendante du nombre de fois et des instants où le processus a été mis en attente du processeur.

Dire qu'un processus est parvenu, par exemple à l'action « lire(m) » signifie, soit que le processus est entrain de l'exécuter, soit que le processus est en attente du processeur et que dès qu'il reprend son exécution, il exécutera cette action.

On voit donc que pour définir un processus, la seule donnée de la suite d'actions (du programme) est insuffisante et qu'il est nécessaire de définir la notion d'état d'un processus.

1. Etat d'un processus :

L'état d'un processus permet d'exprimer :

- si le processus est réellement exécuté,
- s'il est seulement prêt à être exécuté, c'est-à-dire en attente du processeur,
- ou encore s'il est en attente de ressources autres que le processeur ou d'un évènement extérieur (terminaison d'une opération d'E/S, par exemple)

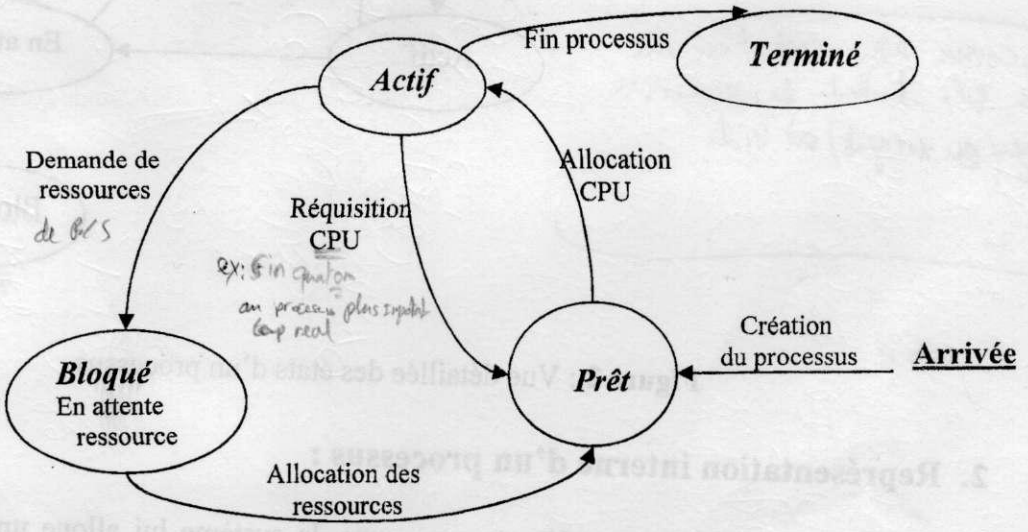


Figure 1 : Principaux Etats d'un processus dans le système

Les principaux états sont

Prêt : un processus est à l'état ***Prêt*** s'il détient toutes les ressources sauf la ressource Processeur central. Ce processus a été crée et le système lui a alloué toutes les ressources nécessaires à son exécution (MC, périphériques, …) sauf le PC.

Actif : un processus est à l'état ***Actif*** s'il détient toutes les ressources même la ressource Processeur central ; c'est-à-dire qu'il exécute ses instructions sur le processeur.

Bloqué : un processus est à l'état ***Bloqué*** s'il ne peut poursuivre son exécution parcequ'il attend une ressource ou un évènement (par exemple, la fin d'une opération E/S).

Terminé : un processus est à l'état ***Terminé*** s'il a terminé son exécution.

On trouve aussi d'autres états comme l'état **suspendu**, où le système d'exploitation met le processus en retrait temporel.

Les systèmes d'exploitation réels considèrent généralement plus d'états. Par exemple, sur le VAX/VMS, un processus peut être dans 14 états différents (attente de M.C, attente événement local,…).

C'est le SE qui modifie l'état d'un processus sous **l'effet d'évènements**. Il y a des évènements externes et des évènements internes au processus.

Par exemple, l'évènement demande de ressource (évènement interne) fait passer le processus de l'état **actif** à l'état **en attente**. L'évènement externe « attribution d'une ressource » fait passer le processeur de l'état ***en attente*** à l'état **prêt**.

La figure 2 donne une vue plus détaillée des différents états possibles d'un processus en considérant l'état suspendu.

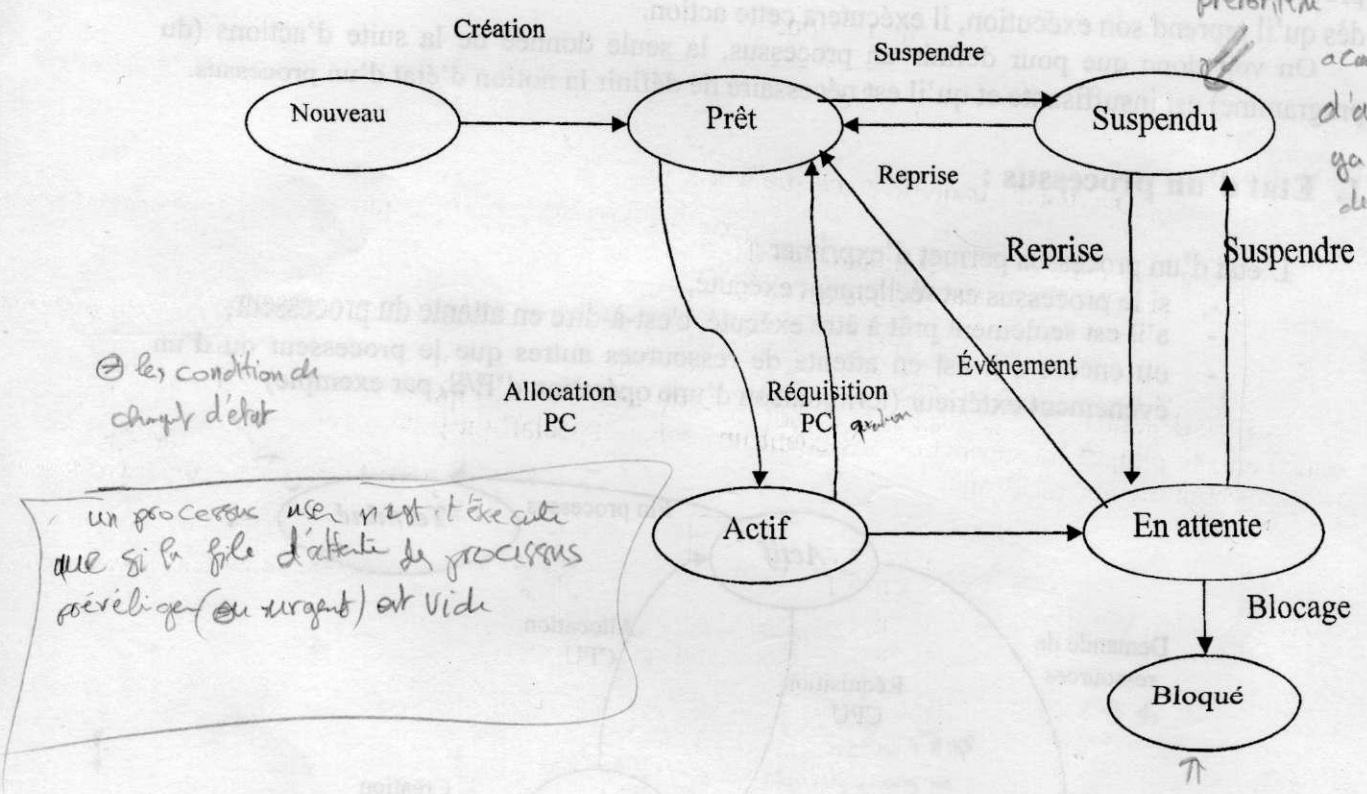


Figure 2 : Vue détaillée des états d'un processus

2. Représentation interne d'un processus :

Pour pouvoir gérer et contrôler un processus, le système lui alloue une structure où il sauvegarde les caractéristiques du processus, par exemple

- son identité (pid),
- son état
- La valeur du compteur ordinal (CO)
- Les valeurs des registres concernant l'exécution
- ses propriétaires,
- son répertoire courant,
- les ressources qui lui été attribuées, par exemple les fichiers ouverts.
- ses demandes de ressource non encore satisfaites.
- informations relatives à la mémoire allouée au processus et des pointeurs vers cette mémoire.
- informations quantitatives : temps d'U.C utilisé, temps restant autorisé.

- informations sur l'opération d'E/S du processus ; périphériques alloués, opérations en attente de terminaisons, fichiers ouverts, etc.

Un processus est donc représenté par cette structure appelée **PCB (Process Control Bloc)**

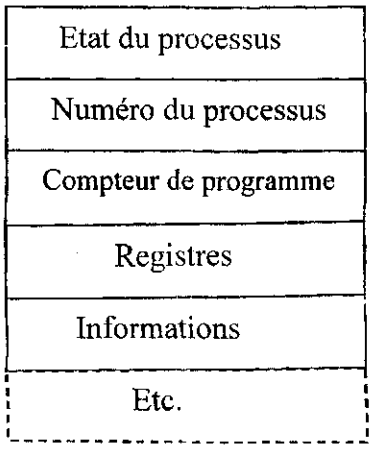


Figure 3 : Bloc de contrôle d'un processus

Ces valeurs et celle du C.O sont sauvegardées lorsque le processus quitte l'état « actif », et elles sont rechargées lorsqu'il retrouve cet état.

Les PCB sont rangés dans la zone mémoire du système. Afin de les manipuler plus rapidement, la plupart des systèmes possèdent un registre spécial qui pointe vers le PCB du processus courant en exécution.

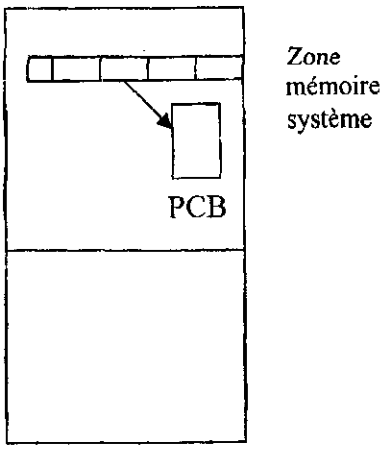


Figure 4: Table des processus du système

3. Opérations sur les processus :

Tout système d'exploitation doit être capable d'effectuer au minimum les opérations suivantes sur un processus :

- Créer un processus
- Déterminer un processus
- Mettre en attente un processus
- Réveiller un processus
- Suspendre un processus

- Reprendre un processus
- Modifier la priorité d'un processus

4. Les processus UNIX

Un processus UNIX se décompose en :

1. un espace mémoire :
   - Un processus comporte du code machine exécutable,
   - une zone mémoire (données allouées par le processus)
   - un tas
   - et une pile (pour les variables locales des fonctions et la gestion des sous-programmes)

La pile est utilisée pour l'appel et le retour de fonctions
Le tas est une zone où est réaalisée l'allocation dynamique avec les fonctions Xalloc().

2. Le bloc de contrôle du processus (PCB)

Process id
Etat
Priorité
Droits d'accès
Zones mémoire Ou table des pages
Mot d'Etat courant
Evènement attendu
Pointeur

Cette structure regroupe les informations nécessaires à l'exécution du processus comme le contenu des registres, sa pile d'exécution, son masque d'interruption, son état, son compteur ordinal, ses indicateurs, un pointeur vers sa table des pages, sa clé d'écritures en mémoire, etc.

4.1 Identifiant d'un processus :

Lors de la création d'un processus, le noyau lui attribue un numéro unique, nommé ***PID*** (processus identifier). Ce numéro permet d'identifier le processus.

Chaque processus a un utilisateur propriétaire, qui est utilisé par le système pour déterminer ses permissions d'accès aux fichiers. La commande `ps` affiche la liste des processus :

ps [-e][-l]

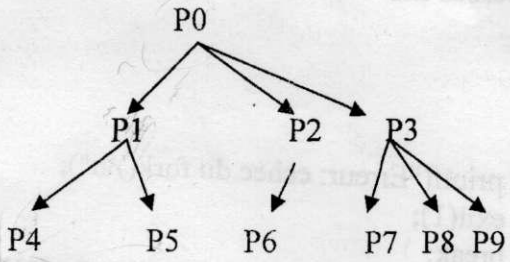
L'option -l permet d'obtenir plus d'informations.
L'option -e permet d'afficher les processus de tous les utilisateurs.

ps affiche beaucoup d'informations. Les plus importante au début sont :
- `UID` : identité du propriétaire du processus ;
- `PID` : numéro du processus ;
- `PPID` : PID du père du processus ;
- `NI` : priorité (nice) ;
- `S` : état du processus (R si actif, S si bloqué, Z si terminé).

4.2 Création d'un processus: fork()

- Lorsqu'on entre une commande dans un shell, le shell lance un processus pour l'exécuter.
- Le shell attend ensuite la fin de ce processus, puis demande la commande suivante.
- Chaque processus a ainsi un processus *père*, qui est celui qui l'a lancé.
- Le numéro du processus père est noté ***PPID*** (parent ***PID***).

Un processus n'a bien sûr qu'un seul père, mais peut lancer l'exécution de plusieurs autres processus, nommés processus *fils*. On a donc affaire à une arborescence de processus.



- Lors de l'initialisation du système UNIX, le processus de PID=0 est créé "manuellement" au démarrage de la machine, ce processus a toujours un rôle spécial pour le système, de plus pour le bon fonctionnement des programmes utilisant fork(), il faut que le PID zéro reste toujours utilisé.
- Le processus zéro crée, grâce à un appel de fork, le processus ***init*** de PID=1.
- ***init*** est l'ancêtre de tous les processus.
- Lorsqu'un processus père se termine, les processus fils non terminés sont dits ***Orphelins***.
- Le processus ***init*** accueille tous les processus orphelins de père (ceci afin de collecter les informations à la mort de chaque processus).

A bas niveau, il n'y a qu'une seule façon de donner naissance à un nouveau processus, la *duplication*. Un processus peut demander au système sa duplication en utilisant la primitive **fork()** :

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

- Tous les processus sauf le processus d'identification 0, sont créés par un appel à *fork*.
- Le processus qui appelle le fork est appelé processus *père*.
- Le nouveau processus est appelé processus *fils*.
- Tout processus a un seul processus père.
- Tout processus peut avoir zéro ou plusieurs processus fils.
- Chaque processus est identifié par un numéro unique, son **PID**.

DEUX valeurs de retour en cas de succès :

- Dans le processus père valeur de retour = le PID du fils,
- Dans le processus fils valeur de retour = zéro.

Sinon

- Dans le processus père valeur de retour = -1.

Les PID et PPID sont les seules informations différentes entre les deux processus (processus père et fils).

L'appel système ***getpid()*** retourne le PID du processus appelant.
L'appel système ***getppid()*** retourne le PID du père du processus.

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

void main(void) {
int pid; /* PID du processus fils */
int i;

pid = fork();
switch (pid) {
        case -1:  printf("Erreur: echec du fork()\n");
                  exit(1);
                  break;
        case 0:                          /* PROCESSUS FILS */
                  printf("Process fils:pid=%d\n",getpid();
                  exit(0); /* fin du processus fils */
                  break;
        default:                  /* PROCESSUS PERE */
                  printf("Ici le pere: le fils a un
                  pid=%d\n", pid );
                  wait(0); /* attente de la fin du fils */
                  printf("Fin du pere.\n");
}
}
```

4.3 Chargement/changement d'un exécutable : execlp()

La primitive ***execlp()*** permet le recouvrement d'un processus par un autre exécutable ; c'est-à-dire changer le code exécuté par un processus

int *execlp*(char *comm, char *arg, ..., NULL);

où :

comm est une chaîne de caractères qui indique la commande à exécuter.
arg spécifie les arguments de cette commande (argv).

Exemple : exécution de **ls -l /usr**

execlp("ls", "ls", "-l", "/usr", NULL);

- La fonction execlp() retourne -1 en cas d'erreur.
- Si l'opération se passe normalement, execlp() ne retourne jamais puisque qu'elle détruit (remplace) le code du programme appelant. L'appel système execlp()change l'exécutable du processus courant en chargeant un nouvel exécutable. Les régions associées au processus sont préalablement libérées

Après duplication, un fils peut "changer de programme" en utilisant la primitive execlp(). Cette primitive conserve l'identité du processus mais remplace son code exécutable (et ses données) par celui d'une nouvelle commande.

Exemple : lorsque qu'un shell exécute la commande : ***compress toto***
qui demande la compression du fichier nommé toto :

1. le shell se duplique (fork) ; on a alors deux processus shell identiques.
2. le shell père se met en attente de la fin du fils (wait).
3. le shell fils remplace son exécutable par celui de la commande `compress`;
4. la commande compress s'exécute et compacte le fichier toto ; lorsqu'elle termine, le processus fils disparaît.
5. le père est alors réactivé, et affiche le prompt suivant.

4.4 Terminaison d'un processus : `exit()`

exit() est une autre fonction qui termine le processus qui l'appelle.

```
#include <stdlib.h>
void exit(int status);
```

L'argument status est un entier qui permet d'indiquer au shell (ou au père de façon générale) qu'une erreur s'est produite. On le laisse à zéro pour indiquer une fin normale.

4.5 Attente fin d'un processus fils : wait()

```
#include <sys/types.h>
#include <sys/wait.h>
int wait( int *st );
```

L'appel wait() permet à un processus d'attendre la fin de l'un de ses fils.
Si le processus n'a pas de fils ou si une erreur se produit, wait() retourne -1.
Sinon, wait() bloque jusqu'à la fin de l'un des fils, et elle retourne son PID.
L'argument st doit être nul.

4.6 Blocage d'un processus: *sleep()*

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

L'appel sleep() est similaire à la commande shell sleep. Le processus qui appelle sleep est bloqué durant le nombre de secondes spécifié, sauf s'il reçoit entre temps un signal.

Notons que l'effet de sleep() est très différent de celui de wait() :

- wait bloque jusqu'à ce qu'une condition précise soit vérifiée (la mort d'un fils),
- alors que sleep attend pendant une durée fixée.
- sleep ne doit jamais être utilisé pour tenter de synchroniser deux processus.

4.7 Droits d'accès d'un processus :

Les droits d'un processus sont définis dans une liste de droits d'accès. En général, ils sont contenus dans le vecteur d'état. Les principaux champs liés aux droits d'un processus sont les suivants :

- Le mode d'exécution Maître/esclave
- Un indicateur d'utilisation de la mémoire virtuelle traduit la possibilité de faire un adressage absolu ou pas
- Clé d'écriture donne les droits d'accès à la mémoire