Chapitre 1 : Concepts de Processus

1. Introduction

Un programme doit être chargé en mémoire centrale pour pouvoir être exécuté. Le processeur charge l'instruction à exécuter à partir de la mémoire dans un registre appelé registre instruction RI. L'adresse de cette instruction se trouve dans un registre appelé compteur ordinal CO.

2. Notion de processus

2.1. Définition

Un processus est un programme en cours d'exécution. C'est une entité dynamique (active) créée à un instant donné, qui disparaît en général au bout d'un temps fini. Un programme est une entité passive, il peut engendrer un ou plusieurs processus.

2.2. Etat d'un processus

Un processus prend un certain nombre d'états durant son exécution, déterminant sa situation dans le système vis-à-vis de ses ressources. Les trois principaux états d'un processus sont :

- **Prêt** : le processus attend la libération du processeur pour s'exécuter.
- **Actif** : le processus est en exécution.
- **Bloqué** : le processus attend une ressource physique ou logique autre que le processeur pour s'exécuter (fin d'E/S, …etc.).

2.3. Bloc de contrôle de processus

Pour suivre son évolution, le SE maintient pour chaque processus une structure de données particulière appelée bloc de contrôle de processus (PCB : Process Control Bloc). Le contenu du PCB varie d'un système à un autre. Il peut contenir :

Numéro de processus
Etat du processus
Compteur d'instruction
Registres
…
Liste des fichiers ouverts

Figure 1 : Bloc de contrôle de processus.

-Identité du processus : chaque processus possède un identifiant (un entier) fourni par le système.

-L'état du processus : Il peut être Prêt, Actif ou Bloqué.

-Le compteur d'instructions : Le compteur indique l'adresse de la prochaine instruction à exécuter par le processus.

-Les registres du processeur : Les registres varient en nombre et en type en fonction de l'architecture de l'ordinateur. Ils englobent des accumulateurs et d'autres registres. Ces informations (**Contexte du processus**) doivent être sauvegardées avec le compteur d'instructions quand il se produit une interruption, afin de permettre au processus de poursuivre correctement son exécution après la reprise.

- Informations sur le scheduling du processeur : Ces informations comprennent la priorité du processus, les pointeurs sur les files d'attente de scheduling.

-Informations sur l'état des E/S : Les fichiers ouverts, la liste des périphériques d'E/S alloués à ce processus, etc.

-**Informations sur la gestion mémoire.**

Le système d'exploitation maintient dans une table appelée «**table des processus**» les informations sur tous les processus créés (une entrée par processus : Bloc de Contrôle de Processus PCB). Cette table permet au SE de localiser et gérer tous les processus.

Dans un système multiprogrammé, le processeur assure l'exécution de plusieurs processus en parallèle (pseudo-parallélisme). Le passage dans l'exécution d'un processus à un autre nécessite une opération de sauvegarde du contexte du processus arrêté, et le chargement de celui du nouveau processus. Ceci s'appelle la **commutation du contexte**. La commutation de contexte consiste à changer les contenus des registres du processeur central par les informations de contexte du nouveau processus à exécuter. La commutation du contexte est déclenchée suivant l'état d'un indicateur qui est consulté par le processeur à chaque point observable.

2.4. Processus sous Linux

Un processus est identifié de manière unique par un numéro (**pid: P**rocess **ID**entifier). La fonction **getpid()** indique le numéro du processus qui l'exécute.
Sous UNIX, les processus sont organisés sous forme d'une hiérarchie (chaque processus à un processus père). La fonction **getppid()** indique le numéro du processus père. Le tout premier processus créé possède le pid 1 ; c'est le processus init.
On peut faire appel a la fonction **fork()** pour créer de nouveau processus. Cette primitive crée un nouveau processus (appelé fils) qui est une copie exacte du processus appelant (processus père). La différence est faite par la valeur de retour de fork(),qui est égale à zéro chez le processus fils, et elle est égale au pid du processus fils chez le père. La primitive renvoie -1 en cas d'erreur.
Les processus peuvent transmettre des signaux (signal.h). On utilise généralement les signaux **SIGUSR1** et **SIGUSR2** qui sont à la disposition des usagers (développeurs). La fonction **kill()** permet d'émettre un signal à un processus **kill (pid_ du_ processus_destination, nom_du_signal)**.
La fonction **signal()** permet d'indiquer l'action à effectuer en réponse à un signal. Son premier paramètre est le numéro du signal, et le deuxième paramètre est soit:

- SIG_IGN, qui indique que le signal doit être ignoré;
- SIG_DFL, qui correspond à l'action par défaut pour le signal;
- Le nom de la fonction de gestion de signal (la fonction doit être de type void).

La fonction **pause()** de unistd.h permet à un processus de se mettre en attente bloquante de n'importe quel signal.

3. L'ordonnancement des processus

Il est courant que plusieurs processus soient simultanément prêts à s'exécuter (c'est-à-dire dans l'état **prêt** dans une file d'attente de demande d'allocation du processeur). Il faut donc faire un choix pour ordonnancer dans le temps leur exécution, c'est le rôle de l'Ordonnanceur (Scheduler).
L'ordonnancement des processus est un ensemble de règles définissant l'ordre d'exécution des processus en tenant compte de la disponibilité des ressources (processeurs) nécessaires, de manière à optimiser un ou plusieurs critères. On peut dire, également, que l'ordonnancement consiste à allouer une ou plusieurs tranches de temps processeur à chaque processus existant dans le système.

3.1. Algorithmes d'ordonnancent

1) L'algorithme du Premier Arrivé Premier Servi (FCFS) :
L'algorithme d'ordonnancement le plus simple est l'algorithme du *Premier Arrivé Premier Servi* (First Come First Served : FCFS). Avec cet algorithme, on alloue le processeur au premier processus qui le demande. L'implémentation de la politique FCFS est facilement gérée avec une file d'attente FIFO. Quand un processus entre dans la file d'attente des processus prêts, il est enchaîné à la queue de la file d'attente. Quand le processeur devient libre, il est alloué au processeur en tête de la file d'attente.
2) L'algorithme du Plus Court d'abord (SJF) :
Cet algorithme (Shortest Job First) affecte le processeur au processus possédant le temps d'exécution le plus court.
3) L'algorithme Shortest Remaining Time First SRTF : (plus court temps restant en premier) est la version préemptive de SJF.
4) Scheduling avec priorité :
Cet algorithme associe à chaque processus une priorité, et le processeur sera affecté au processus de plus haute priorité.
5) L'algorithme de Round Robin (Tourniquet) :
L'algorithme de scheduling du tourniquet, appelé aussi **Round Robin**, a été conçu pour des systèmes à temps partagé. Il alloue le processeur aux processus à tour de rôle, pendant une tranche de temps appelée **quantum**. Dans la pratique le quantum s'étale entre 10 et 100 ms.

Ordonnancement à files multi-niveaux

1) Définir des classes de processus. Ex : c1, c2, c3

2) Associer à chaque classe :

-Une priorité (par rapport aux autres classes) : c3 plus prioritaire, …, c1 moins prioritaire;

-Définir un algorithme d'ordonnancement c1: Tourniquet; c2: Tourniquet avec priorité; c3 : Priorité;

-Le processeur est alloué aux processus de la classe la plus prioritaire;

-On ne change de classe que si la classe la plus prioritaire est vide.

Remarque : Un processus ne peut pas changer de classe : Un processus d'une classe Ci reste dans la file de cette classe jusqu'à ce qu'il quitte le système.

Les files Multi-niveaux avec recyclage

-On dispose de **n** files de **processus à l'état prêt**: **f0, f1, f2, ..., fn-1**.

-A chaque file **fi** est associé un quantum de temps **qi** dont la valeur croit avec le rang de la file.

-Les nouveaux processus sont rangés dans la file f0.

-Lorsqu'un processus de la file **fi** a épuisé son quantum de temps sans avoir terminé son exécution, il rentre dans la file **fi+1**.

-Un processus de la file **fi** n'est servi que si toutes les files de rang inférieur à **i** sont vides.

-Les processus de la dernière file fn-1sont recyclés dans la même file.

3.2. Critères d'ordonnancement

Les principaux objectifs assignés à un ordonnanceur sont:

- Occupation maximale du processeur,
- S'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur,
- Minimiser le temps de réponse,
- Satisfaire le maximum de processus en respectant certaines contraintes telles la priorité, l'échéance (dans les systèmes temps réel), etc…

Il est impossible de créer un algorithme qui optimise tous les critères de façon simultanée. Plusieurs critères ont été proposés pour comparer et évaluer les performances des algorithmes d'ordonnancement. Les critères les plus souvent utilisés sont :

1) Utilisation du processeur central (CPU):

Le **taux** d'utilisation du processeur central est défini par le rapport temps CPU consommé/ temps de **séjour** des processus. Ce taux doit être le plus élevé possible.

2) Temps de **séjour**: C'est le temps passé par le processus dans le système. C'est la somme du temps d'exécution et du temps passé dans les différentes files d'attentes. Soient te: le temps (ou date) d'entrée processus dans le système et ts: le temps (ou date) de sortie du processus. Temps de résidence = ts – te

3) Temps **d'attente** (waiting time): C'est le temps passé dans la file des processus prêts*;* c'est-à-dire le temps total d'attente du processeur central (cpu).

Remarque : ne pas confondre avec le temps passé dans l'état bloqué (par exemple temps d'attente des entrées/sorties ou temps d'attente d'une ressource autre que le cpu,...).

4) Temps de **réponse**: C'est le temps qui sépare le moment où le processus soumet une requête et le *début de la réponse* à cette requête. Soient te: le temps (ou date) de soumission de la requête et ts: le temps (ou date) de *début de la réponse* à cette requête. Temps de réponse = ts – te

4. Ressources d'un processus

L'exécution d'un processus nécessite un certain nombre de ressources. Ces ressources peuvent être logiques ou physiques. Les ressources physiques sont la mémoire, le processeur, les périphériques etc. Les ressources logiques peuvent être une variable, un fichier, un code, etc. Certaines ressources peuvent être utilisées en même temps (ou partagées) par plusieurs processus. Dans ce cas, elles sont dites partageables ou à accès multiple. Dans le cas contraire, elles sont dites à un seul point d'accès ou à accès exclusif. Dans ce dernier cas, il est nécessaire d'ordonner l'accès à ce type de ressources pour éviter des situations incohérentes.

5. Relations entre processus : Parallélisme, coopération et compétition

La programmation parallèle consiste à exécuter simultanément plusieurs programmes, qui coopèrent pour réaliser un but commun et/ou qui sont en compétition pour la possession de ressources.
L'exécution parallèle peut se faire :

- Sur une machine ne disposant que d'un seul processeur partagé.
- Sur une machine ayant plusieurs processeurs.
- Sur plusieurs machines différentes relier par un réseau de communication.

Durant leur évolution, les processus d'un système interagissent les uns avec les autres. Selon que les processus se connaissent mutuellement ou pas, deux types interactions sont possibles :

1) Compétition : Situation dans laquelle plusieurs processus doivent utiliser simultanément une ressource à accès exclusif (ressource ne pouvant être utilisée que par un seul processus à la fois). Exemple : Processeur (cas du pseudo-parallélisme) ; Imprimante.

- Une solution possible : Faire attendre le processus demandeurs que l'occupant actuel ait fini.

2) Coopération : Situation dans laquelle plusieurs processus collaborent à une tâche commune et doivent se synchroniser pour réaliser cette tâche. Exemple : P1 produit un fichier ; P2 imprime le fichier. P1 met à jour un fichier ; P2 consulte le fichier.

- La synchronisation se ramène au cas suivant : Un processus doit attendre qu'un autre processus ait franchi un certain point de son exécution.

Dans les 2 types de relations (compétition ou coopération), on est conduit à faire attendre un processus.

6. Les processus légers (Threads)

De nombreux systèmes d'exploitation modernes offrent la possibilité d'associer à un même processus plusieurs chemins d'exécution. On parle du **multithread** qui permet d'exécuter en **parallèle** des parties d'un même processus. Les différents threads d'une application partage un même espace d'adressage en ce qui concerne leurs données (variables globales). Chaque thread a cependant, en plus des ressources communes, sa propre zone de données ou de variables locales, sa propre pile d'exécution, ses propres registres et son propre compteur ordinal.

6.1. Création d'un thread (POSIX)

La création d'un processus donne lieu à la création du **thread principal** (thread main). Un retour à la fonction main entraîne la terminaison du processus et par conséquent la terminaison de tous ses threads. Le thread principal peut créer des **threads annexes** par la fonction **pthread_create()**. Le nouveau thread crée :

-est identifié par un identificateur unique TID : Thread Identifier (pthread_self).
-exécute une fonction passée en paramètre lors de sa création.
-possède des attributs : sa stratégie d'ordonnancement, sa priorité, etc.
-peut se terminer (pthread_exit).
-peut attendre la fin d'un autre thread : appartenant au même processus (pthread_join).

6.2. Inconvénient des threads

Dans une application multi-threads, les échanges de données entre threads se font facilement par accès à l'espace virtuel commun. L'inconvénient des threads résulte ainsi de ce partage d'un même espace virtuel entre tous les threads d'un même processus. Il faut donc gérer l'accès concurrent à ces ressources pour éviter par exemple les incohérences. Ce qui nécessité de programmer des sections critiques et de synchroniser les threads.