Systèmes d'exploitation : principes, programmation et virtualisation

Chapitre 2 : processus et ordonnancement

Table des matières

Processus et ordonnancement

1 Notions de processus et threads

Nous avons vu dans la première partie de cours que le processeur pouvait exécuter une séquence d'instructions d'un programme, puis une séquence d'instructions d'un autre programme, avant de revenir au premier, et ainsi de suite. Si la durée d'exécution des séquences est suffisamment réduite, les utilisateurs ont la sensation que l'ensemble des programmes s'exécute en parallèle : on parle de *pseudo- parallélisme.* La gestion du passage d'un programme en exécution à un autre (commutation), implique le concept de *processus,* que nous allons approfondir dans ce chapitre.

1.1 Définitions

- Un processus est un programme en cours d'exécution auquel sont associés un environnement processeur et un environnement mémoire appelés contexte du processus.
- Un processus est l'instance dynamique d'un programme et incarne le fil d'exécution de celui-ci dans un *espace d'adressage protégé* (objets propres : ensemble des instructions et données accessibles)
- Un programme *réentrant* est un programme pour lequel il peut exister plusieurs processus en même temps

1.2 Etat des processus

Nous l'avons vu dans la première partie du cours, il est nécessaire de *bloquer* un processus quand il est en attente de ressources. Le processeur doit donc redevenir libre pour un autre processus, qui était auparavant bloqué. Le cycle de vie d'un processus peut donc s'illustrer par le schéma suivant :

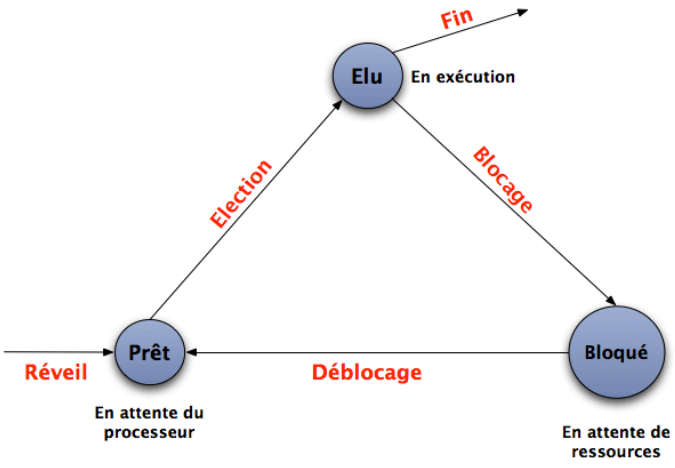

Fig. 1 : états d'un processus

Un processus est toujours créé dans l'état *prêt.* Lorsqu'il obtient le processeur, il est dans l'état *élu,* c'est l'exécution de ses instructions. Lors de cette exécution, le processus peut demander à accéder à une ressource qui n'est pas immédiatement disponible : le processus ne peut pas poursuivre son exécution tant qu'il n'a pas obtenu la ressource. Le processus quitte alors le processeur et passe dans l'état *bloqué.* L'état bloqué est l'état d'attente d'une ressource autre que le processeur. Dès que la ressource est obtenue, le processus est donc à même de reprendre son exécution. Il est dans l'attente du processeur, de nouveau l'état *prêt.* Le passage de l'état *prêt* vers l'état *élu* constitue *l'opération d'élection.* Le passage de l'état *élu* vers l'état *bloqué* est *l'opération de blocage.* Le passage de l'état *bloqué* vers l'état *prêt* est *l'opération de déblocage.*

Nous verrons que ce schéma peut être enrichi, avec des états supplémentaires liés à la gestion de la mémoire et à l'exécution d'instructions particulières.

1.3 Bloc de contrôle des processus

Le système d'exploitation stocke les informations relatives à un processus dans un bloc de contrôle, qui va permettre la sauvegarde et la restauration des contextes mémoire et processeur lors des opérations de *commutations de contexte.* Le bloc de contrôle d'un processus (PCB) contient les informations suivantes :

- un identificateur unique du processus (un entier) : le PID
- l'état courant du processus (élu, prêt, bloqué)
- le contexte processeur du processus : la valeur du compteur ordinal ("à quelle instruction il est arrivé"), la valeur des autres registres du processeur
- le contexte mémoire : ce sont des informations mémoire qui permettent de trouver le code et les données du processus en mémoire centrale
- des informations diverses de comptabilisation pour les statistiques sur les performances système
- des informations liées à l'ordonnancement du processus. Le PCB permet la sauvegarde et la restauration du contexte mémoire et du contexte processeur lors des opérations de commutations de contexte.

1.4 Threads

Le processus léger, *Thread*, constitue une extension du modèle de processus. Un processus classique est constitué d'un espace d'adressage avec un seul fil d'exécution, ce fil d'exécution étant représenté par une valeur de compteur ordinal et une pile d'exécution. Un thread est un espace d'adressage dans lequel plusieurs fils d'exécution peuvent évoluer en parallèle. Ces fils d'exécution sont chacun caractérisés par une valeur de compteur ordinal propre et une pile d'exécution privée. De cette définition, on tire plusieurs avantages :

- Les threads d'un même fil partagent code et données (ils appartiennent au même processus !) : les commutations lors d'un changement d'état d'un thread sont allégées. On ne change que les registres et le compteur ordinal, pas la mémoire centrale
- Les partages de données sont facilités : dans le cas de processus distincts, les partages de données ne sont pas inclus dans le langage, et nécessitent la mise en œuvre de services fournis par le système d'exploitation, lourds et peu souples. Par contre, les threads

partageant les mêmes données, il est généralement nécessaire de synchroniser et protéger les accès aux données, ce que nous verrons plus en avant dans le cours.

2 Ordonnancement des processus

Le système d'exploitation doit donc gérer l'ensemble des processus, dans le but de maintenir le taux d'utilisation du processeur le plus élevé possible. Pour cela, le système d'exploitation met en œuvre une *politique d'ordonnancement,* algorithme régissant la commutation des tâches.

2.1 Critères d'ordonnancement

L'algorithme d'ordonnancement doit identifier le processus qui sera élu, de manière à assurer la "meilleure" performance du système (à défaut, la moins mauvaise). Selon les algorithmes, et selon les objectifs recherchés, différents critères vont être optimisés, parmi lesquels :

- *Utilisation de l'UC* : il s'agit de prendre en compte le temps pendant lequel l'unité centrale exécute un processus.
- *Utilisation répartie* : c'est le pourcentage de temps pendant lequel est utilisé l'ensemble des ressources. On évalue l'utilisation de la mémoire, des E/S, .. plutôt que le temps processeur.
- *Débit* : c'est le nombre de processus pouvant être exécutés par le système durant une période de temps donnée. Le calcul du débit doit prendre en compte la longueur moyenne d'un processus. Sur des systèmes aux processus longs, le débit est inférieur à celui des systèmes aux processus courts.
- *Temps de rotation* : durée moyenne de l'exécution d'un processus. Il est inversement proportionnel au débit.
- *Temps d'attente* : durée moyenne d'attente des processus du processeur.
- *Temps de réponse* : le temps moyen pour répondre aux entrées de l'utilisateur (systèmes interactifs).
- *Equité* : degré auquel tous les processeurs reçoivent une chance égale de s'exécuter. On évalue entre autre le fait de ne pas permettre à un processus de souffrir de *famine*, à savoir rester bloqué indéfiniment.

2.2 Principes

Un processus est donc successivement en phases de calcul et en phase d'entrées-sorties. L'idée est donc de tenter de se faire recouvrir une phase d'E/S d'un processus 1 avec des phases de calcul d'un autre processus 2 :

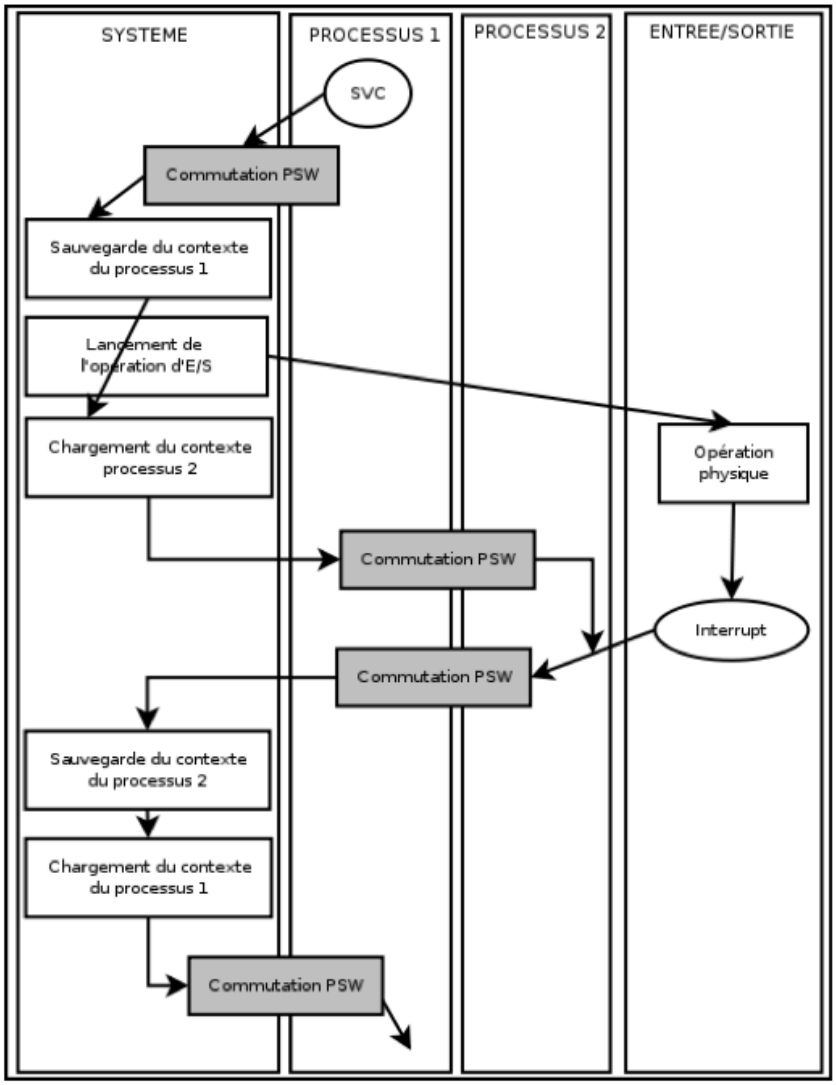

Fig. 2 : commutations

2.3 Algorithmes d'ordonnancement

Au début de l'informatique, l'ordonnancement était souvent non préemptif, ou coopératif : un processus conservait le contrôle de l'unité centrale jusqu'à ce qu'il se bloque ou qu'il se termine. Pour des travaux par lots, le système convenait, le temps de réponse n'ayant pas grande importance. Sur les systèmes interactifs actuels, c'est l'ordonnancement préemptif qui est utilisé : le système d'exploitation peut *préempter* un processus avant qu'il ne se bloque ou ne se termine, afin d'attribuer le processeur à un autre processus. On peut ainsi compléter le schéma du premier chapitre.

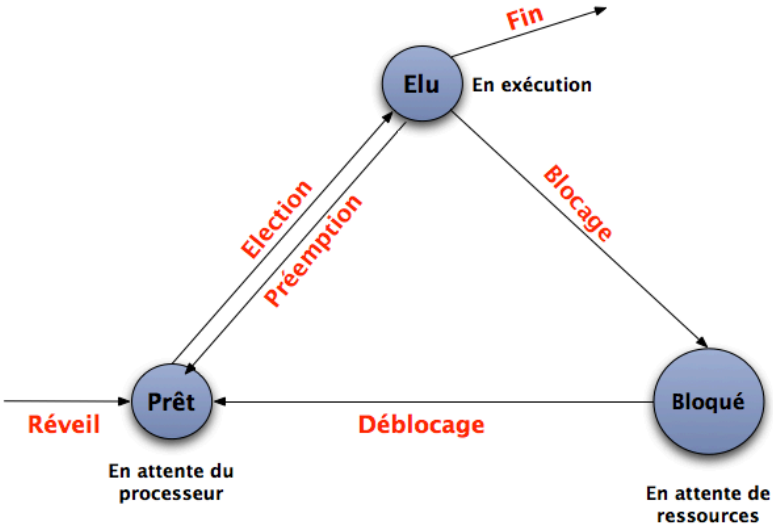

Fig. 3 : cycle de vie des processus en ordonnancement préempté

On recense six algorithmes d'ordonnancement répandus :

2.3.1 FIFO - First In First Out

Le FIFO est le plus simple des algorithmes d'ordonnancement. En effet, l'ordonnanceur (non préemptif), utilise une file dans laquelle sont stockés dans l'ordre d'arrivée les processus en attente du processeur.

Exemple d'ordonnancement FIFO :

Soit trois processus P1,P2,P3 faisant uniquement du calcul. Les temps d'exécution des trois processus sont respectivement 15, 4 et 7 millisecondes. Le chronogramme d'exécution des processus peut être représenté de la manière suivante :

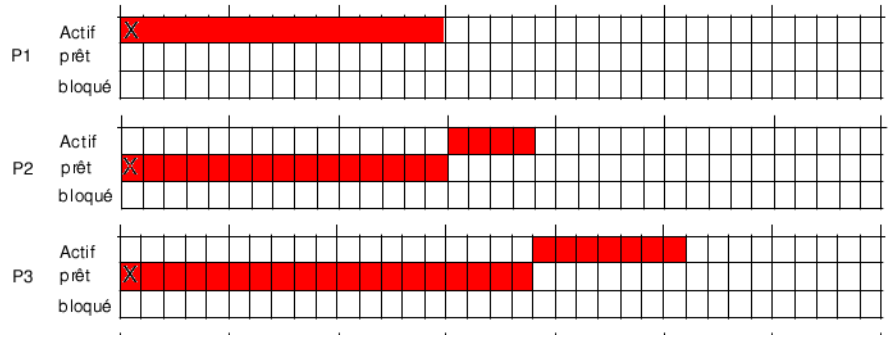

Fig. 4 : ordonnancement FIFO

L'ordonnancement FIFO a donc tendance a privilégier les processus longs ou tributaires de l'unité centrale.

2.3.2 SJF - Shortest First Job

L'ordonnancement basé sur l'algorithme du travail le plus court d'abord est également non préemptif. A partir d'une estimation du temps nécessaire à l'exécution des processus, le prochain élu est donc le plus court. Ce qui nous donne pour l'exemple précédent (en supposant qu'au départ les trois processus soient bloqués) :

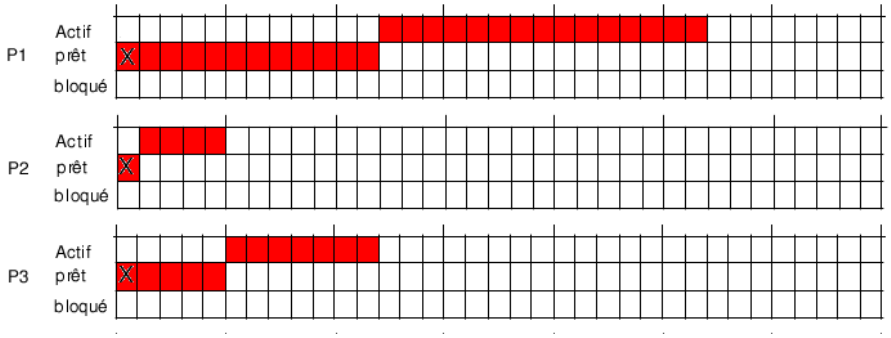

Fig. 5 : ordonnancement SJF

Cet algorithme privilégie les programmes courts.

2.3.3 SR - Shortest Remaining Time

L'ordonnancement du temps restant le plus court est la version préemptive de l'algorithme SJF : chaque fois qu'un processus passe dans l'état prêt (à sa création ou à la fin d'une E/S), on compare la valeur estimée du temps de traitement restant à celle du processus en cours. On préempte le processus en cours si son temps restant est plus long. L'algorithme privilégie les programmes courts, mais il peut y avoir un risque de famine pour les programmes longs.

2.4 RR - Round Robin

L'ordonnancement par tourniquet (*Round Robin)* est préemptif, et sélectionne le processus attendant depuis le plus longtemps. Un quantum de temps est spécifié, et à chacun de ces intervalles, on choisit une nouvelle sélection. Plus le quantum est faible, plus les temps de réponse sont courts, mais plus les commutations sont fréquentes, d'où une dégradation possible des performances.

Nous reprenons l'exemple précédent, en supposant que P1,P2 et P3 ont été soumis dans cet ordre. Le quantum est fixé à 3ms.

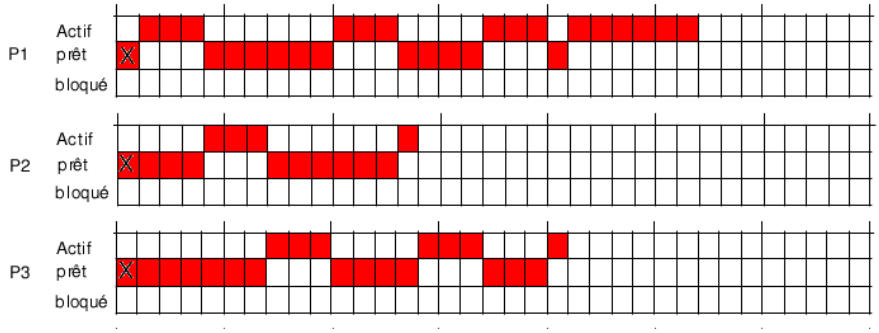

Fig. 6 : ordonnancement RR

2.5 Avec priorité

Pour l'ordonnancement préemptif à priorité, on affecte une valeur à chaque processus. Le processus élu est celui qui à la priorité la plus élevée (valeur la plus faible), et en cas d'égalité, on utilise FIFO. Les priorités peuvent être fixées en fonction des caractéristiques du processus (beaucoup d'E/S, utilisation de la mémoire,...), de l'utilisateur, ou même d'une priorité fixée par l'administrateur ou l'utilisateur (commande `nice` sous *Unix).* Pour éviter le risque de famine des processus de trop faible priorité, on peut augmenter la priorité en fonction du temps d'attente écoulé.

Exemple d'ordonnancement avec priorité :

Soit P1,P2,P3 et P4 des processus de temps d'exécution respectifs 10,1,2,3, et de priorité respectives 3,1,3,2.

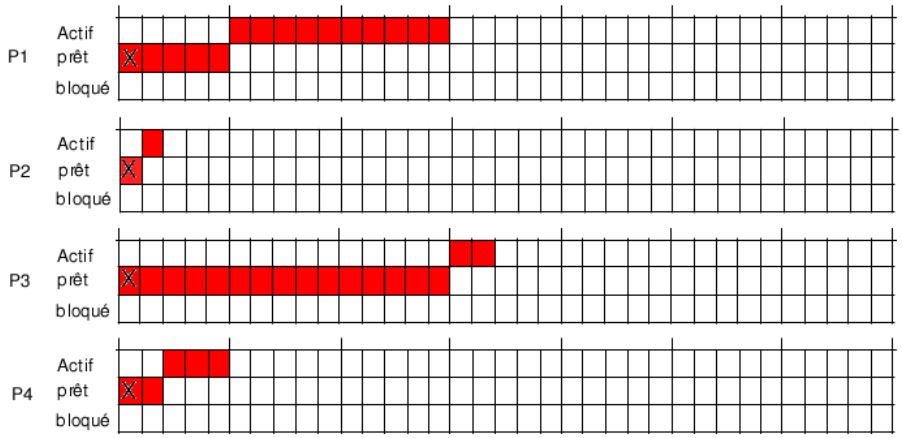

Fig. 7 : ordonnancement avec priorité

2.6 MQS - Multilevel Queue Scheduling

L'ordonnancement par files multi-niveaux est une extension de l'ordonnancement avec priorité, mais en plaçant les processus de même priorité dans des files d'attente distinctes. Par exemple, les systèmes en temps partagé supportent généralement la notion de processus en premier (*foreground*) et en arrière plan (*background*). Les premiers sont souvent des processus interactifs, alors que les seconds n'ont besoin de s'exécuter que si le processeur est libre. Donc les deux types de processus nécessitent deux ordonnancements différents.

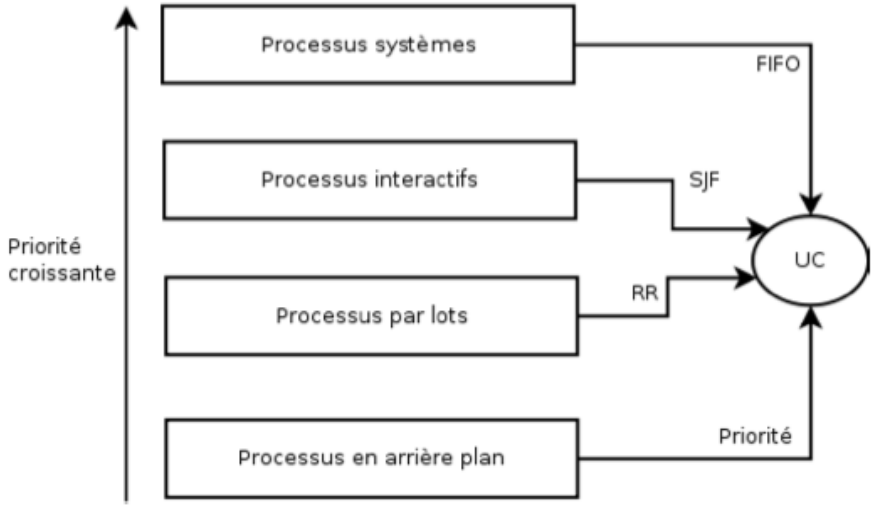

Fig. 8 : Multilevel Queue Scheduling

Dans cet exemple, chaque file de processus est ordonnancée par un algorithme différent.

Les processus des files de plus basse priorité ne peuvent s'exécuter que lorsque les processus de la file supérieure sont tous bloqués. Il y a donc risque de famine.

2.7 MFQ - Multilevel Feedback Queues

L'ordonnancement par *files multi-niveaux à retour* résout le problème des MQS en permettant aux processus de changer de file au cours du temps. On sépare ainsi les processus selon leur temps d'occupation de l'unité centrale. Si un processus consomme trop de temps processeur, il est alors déplacé vers une file de priorité moindre. Les files de plus basse priorité ont un temps processeur moindre.

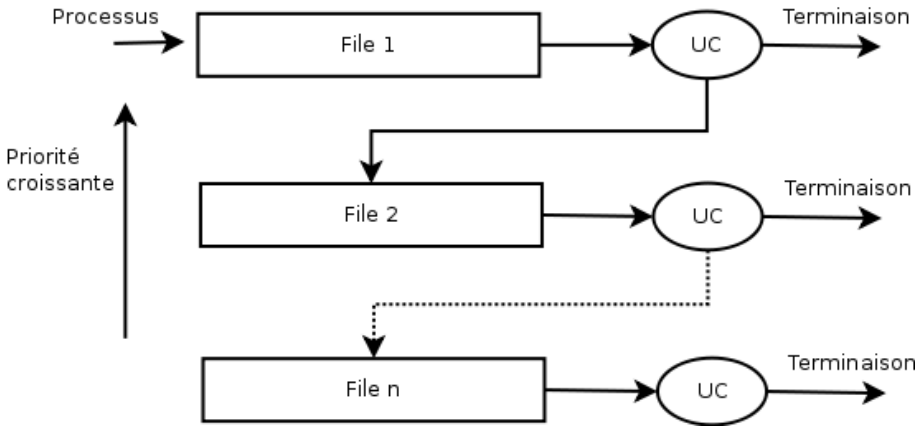

Fig. 9 : Multilevel Feedback Queues

- les implémentations des MFQ diffèrent selon :
- le nombre de files
- l'ordonnancement de chaque file
- la méthode utilisée pour déterminer la descente d'un processus en file inférieure
- la méthode utilisée pour déterminer la montée d'un processus en file supérieure

- la méthode utilisée pour choisir la file initiale d'un processus