Chapitre 8

Ordonnancement des processus

DAns un système multi-utilisateurs à temps partagé, plusieurs processus peuvent être présents en mémoire centrale en attente d'exécution. Si plusieurs processus sont prêts, le système d'exploitation doit gérer l'allocation du processeur aux différents processus à exécuter. C'est l'ordonnanceur qui s'acquitte de cette tâche.

8.1 Introduction

Tout le logiciel d'un ordinateur peut être vu comme un ensemble de processus dont l'exécution est gérée par un processus particulier : l'ordonnanceur (*scheduler* en anglais). Un ordonnanceur fait face à deux problèmes principaux :

– le choix du processus à exécuter, et
– le temps d'allocation du processeur au processus choisi.

Un système d'exploitation multitâche est *préemptif* lorsque celui-ci peut arrêter (réquisition) à tout moment n'importe quelle application pour passer la main à la suivante. Dans les systèmes d'exploitation préemptifs on peut lancer plusieurs applications à la fois et passer de l'une à l'autre, voire lancer une application pendant qu'une autre effectue un travail.

Il y a aussi des systèmes d'exploitation dits multitâches, qui sont en fait des « multi-tâches coopératifs ». Quelle est la différence ? Un *multitâche coopératif* permet à plusieurs applications de fonctionner et d'occuper des plages mémoire, laissant le soin à ces applications de gérer cette occupation, au risque de bloquer tout le système. Par contre, avec un « multi-tâche préemptif », le noyau garde toujours le contrôle (qui fait quoi, quand et comment), et se réserve le droit de fermer les applications qui monopolisent

les ressources du système. Ainsi les blocages du système sont inexistants.

8.2 Types d'ordonnanceurs

Il est possible de distinguer trois types d'ordonnanceurs : à long terme, à moyen terme et à court terme. Leurs principales fonctions sont les suivantes :

À long terme : L'ordonnanceur fait la sélection de programmes à admettre dans le système pour leur exécution. Les programmes admis deviennent des processus à l'état **prêt**. L'admission dépend de la capacité du système (degré de multiprogrammation) et du niveau de performance requis.

À moyen terme : Il fait la sélection de processus déjà admis à débarquer ou rembarquer sur la mémoire. Il effectue ses tâches de gestion en fonction du degré de multiprogrammation du système, et aussi des requêtes d'E/S des périphériques.

À court terme : L'ordonnanceur à court terme a comme tâche la gestion de la file des processus prêts. Il sélectionne — en fonction d'une certaine politique — le prochain processus à exécuter. Il effectue aussi le changement de contexte des processus. Il peut implanter un ordonnancement préemptif, non préemptif, ou coopératif. L'ordonnanceur est activé par un événement : interruption du temporisateur, interruption d'un périphérique, appel système ou signal.

8.3 Objectifs de l'ordonnanceur d'un système multi-utilisateur

Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont, entre autres :

- S'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur.
- Minimiser le temps de réponse.
- Utiliser le processeur à 100%.
- Utilisation équilibrée des ressources.
- Prendre en compte des priorités.
- Être prédictibles.

Ces objectifs sont parfois complémentaires, parfois contradictoires : augmenter la performance par rapport à l'un d'entre eux peut se faire en détriment d'un autre. Il est impossible de créer un algorithme qui optimise tous les critères de façon simultanée.

8.4 Ordonnanceurs non préemptifs

Dans un système à **ordonnancement non préemtif** ou sans réquisition, le système d'exploitation choisit le prochain processus à exécuter, en général, le **Premier Arrivé est le Premier Servi PAPS** (ou *First-Come First-Served* FCFS) ou le **plus court d'abord** (*Short Job First SJF*). Il lui alloue le processeur jusqu'à ce qu'il se termine ou qu'il se bloque (en attente d'un événement). Il n'y a pas de réquisition.

Si l'ordonnanceur fonctionne selon la stratégie **SJF**, il choisit, parmi le lot de processus à exécuter, le plus court (plus petit temps d'exécution). Cette stratégie est bien adaptée au traitement par lots de processus dont les temps maximaux d'exécution sont connus ou fixés par les utilisateurs car elle offre un meilleur temps moyen de séjour. Le temps de séjour d'un processus (temps de rotation ou de virement) est l'intervalle de temps entre la soumission du processus et son achèvement.

Considérons par exemple un lot de quatre processus dont les temps respectifs d'exécution sont a, b, c et d. Le premier processus se termine au bout du temps a ; le deuxième processus se termine au bout du temps $a+b$; le troisième processus se termine au bout du temps $a+b+c$; le quatrième processus se termine au bout du temps $a+b+c+d$. Le temps moyen de séjour $< t >$ est : $< t >= \frac{4a+3b+2c+d}{4}$. On obtient un meilleur temps de séjour pour $a \leq b \leq c \leq d$. Toutefois, l'ordonnancement du plus court d'abord est optimal que si les travaux sont disponibles simultanément.

▶ **Exemple 1.** Considérons cinq travaux A, B, C, D et E, dont les temps d'exécution et leurs arrivages respectifs sont donnés dans la table 8.1. Faire un schéma qui illustre son exécution et calculer le temps de séjour de chaque processus, le temps moyen de séjour, le temps d'attente et le temps moyen d'attente en utilisant :

1. Premier arrivé premier servi (**PAPS**)
2. Le plus court d'abord (**SJF**)

1. PAPS. Schéma d'exécution :

A	A	A	B	B	B	B	B	B	C	C	C	C	D	D	E
1				5					10					15	

Processus	Temps d'exécution	Temps d'arrivage
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

TAB. 8.1 – Données d'ordonnancement.

Au temps 0, seulement le processus A est dans le système et il s'exécute. Au temps 1 le processus B arrive mais il doit attendre qu'A termine car il a encore 2 unités de temps. Ensuite B s'exécute pendant 4 unités de temps. Au temps 4, 6, et 7 les processus C, D et E arrivent mais B a encore 2 unités de temps. Une fois que B a terminé, C, D et E entrent au système dans l'ordre[1].

Le temps de séjour pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison. Ainsi :

Processus	Temps de sejour
A	3-0 = 3
B	9-1 = 8
C	13-4 = 9
D	15-6 = 9
E	16-7 = 9

Le temps moyen de séjour est : $\frac{(3+8+9+9+9)}{5} = \frac{38}{5} = 7,6$

Le temps d'attente est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente
A	3-3 = 0
B	8-6 = 2
C	9-4 = 5
D	9-2 = 7
E	9-1 = 8

Le temps moyen d'attente est : $\frac{(0+2+5+7+8)}{5} = \frac{23}{5} = 4,4$

Il y a cinq tâches exécutées dans 16 unités de temps, alors $16/5 = 3,2$ **unités de temps par processus**.

2. Le plus court d'abord. Schéma d'exécution :

[1] Bien entendu, on suppose qu'il n'y a aucun blocage.

A	A	A	B	B	B	B	B	B	E	D	D	C	C	C	C
1				5					10					15	

Pour la stratégie **SJF** nous aurons la séquence d'exécution A,B,E,D,C, et le temps de séjour est :

Processus	Temps de séjour
A	3-0 = 3
B	9-1 = 8
E	10-7 = 3
D	12-6 = 6
C	16-4 = 12

$\frac{(3+8+3+6+12)}{5} = \frac{32}{5} = 6,4$

Processus	Temps d'attente
A	3-3 = 0
B	8-6 = 2
E	3-1 = 2
D	6-2 = 4
C	12-4 = 8

Le temps moyen d'attente est : $\frac{(0+2+2+4+8)}{5} = \frac{16}{5} = 3,2$

Il y a cinq tâches exécutées dans 16 unités de temps, alors $16/5 = 3,2$ **unités de temps par processus**.

Comment appliquer cette technique aux processus interactifs ?

Chaque processus se comporte comme suit : il attend une commande, l'exécute, attend la commande suivante, et ainsi de suite. Alors parmi les processus prêts, le processus élu est celui dont la commande à exécuter est *la plus courte en temps*. Le temps d'exécution de la prochaine commande de chaque processus est estimé en se basant sur le comportement passé du processus : Si T_0 est le temps d'exécution estimé pour la première commande et T_i le temps d'exécution mesuré pour la i^{ieme} commande, les estimations successives sont :

T_0 pour la première ;

$T_0/2 + T_1/2$ pour la seconde ;

$T_0/4 + T_1/4 + T_2/2$ pour la troisième ;

$T_0/8 + T_1/8 + T_2/4 + T_3/2$, ...

Les ordonnanceurs non préemptifs ne sont pas intéressants pour les systèmes multi-utilisateurs car les temps de réponse ne sont pas toujours acceptables.

8.5 Ordonnanceurs préemptifs

Dans un schéma d'**ordonnanceur préemptif**, ou avec réquisition, pour s'assurer qu'aucun processus ne s'exécute pendant trop de temps, les ordinateurs ont une horloge électronique qui génère périodiquement une interruption. A chaque interruption d'horloge, le système d'exploitation reprend la main et décide si le processus courant doit poursuivre son exécution ou s'il doit être suspendu pour laisser place à un autre. S'il décide de suspendre son exécution au profit d'un autre, il doit d'abord sauvegarder l'état des registres du processeur avant de charger dans les registres les données du processus à lancer. C'est qu'on appelle la *commutation de contexte* ou le *changement de contexte*. Cette sauvegarde est nécessaire pour pouvoir poursuivre ultérieurement l'exécution du processus suspendu.

Le processeur passe donc d'un processus à un autre en exécutant chaque processus pendant quelques dizaines ou centaines de millisecondes. Le temps d'allocation du processeur au processus est appelé **quantum**. Cette commutation entre processus doit être rapide, c'est-à-dire, exiger un temps nettement inférieur au **quantum**.

Le processeur, à un instant donné, n'exécute réellement qu'un seul processus, mais pendant une seconde, le processeur peut exécuter plusieurs processus et donne ainsi l'impression de parallélisme (pseudo-parallélisme).

Problèmes :

- Choix de la valeur du quantum.
- Choix du prochain processus à exécuter dans chacune des situations suivantes :
    1. Le processus en cours se bloque (passe à l'état Attente).
    2. Le processus en cours passe à l'état Prêt (fin du quantum...).
    3. Un processus passe de l'état Attente à l'état Prêt (fin d'une E/S).
    4. Le processus en cours se termine.

8.5.1 Ordonnancement du plus petit temps de séjour

L'**ordonnancement du plus petit temps de séjour** ou ***Shortest Remaining Time*** est la version préemptive de l'algorithme SJF. Un processus arrive dans la file de processus, l'ordonnanceur compare la valeur espérée pour ce processus contre la valeur du processus actuellement en exécution. Si le temps du nouveau processus est plus petit, il rentre en exécution immédiatement.

8.5.2 Ordonnancement circulaire

L'**algorithme du tourniquet**, **circulaire** ou *round robin* montré sur la figure 8.1 est un algorithme ancien, simple, fiable et très utilisé. Il mémorise dans une file du type FIFO *(First In First Out)* la liste des processus prêts, c'est-à-dire en attente d'exécution.

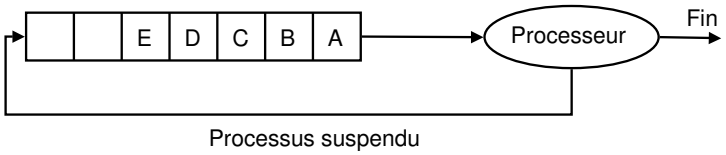

FIG. 8.1 – Ordonnancement circulaire.

Choix du processus à exécuter

Il alloue le processeur au processus en tête de file, pendant un quantum de temps. Si le processus se bloque ou se termine avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus (celui en tête de file). Si le processus ne se termine pas au bout de son quantum, son exécution est suspendue. Le processeur est alloué à un autre processus (celui en tête de file). Le processus suspendu est inséré en queue de file. Les processus qui arrivent ou qui passent de l'état bloqué à l'état prêt sont insérés en queue de file.

Choix de la valeur du quantum

Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur. Un quantum trop élevé augmente le

temps de réponse des courtes commandes en mode interactif. Un quantum entre 20 et 50 ms est souvent un compromis raisonnable[2].

▶ **Exemple 2.** Soient deux processus A et B prêts tels que A est arrivé en premier suivi de B, 2 unités de temps après. Les temps de UCT nécessaires pour l'exécution des processus A et B sont respectivement 15 et 4 unités de temps. Le temps de commutation est supposé nul. Calculer le temps de séjour de chaque processus A et B, le temps moyen de séjour, le temps d'attente, le temps moyen d'attente, et le nombre de changements de contexte pour :

- SRT
- Round robin (quantum = 10 unités de temps)
- Round robin (quantum = 3 unités de temps)

Solution SRT :

A	A	B	B	B	B	A	A	A	A	A	A	A	A	A	A	A	A	A
1				5					10					15				

Processus	Temps de séjour
A	19-0 = 19
B	6-2 = 4

Temps moyen de séjour = $\frac{19+4}{2} = 11,5$

Processus	Temps d'attente
A	19-15 = 4
B	4-4 = 0

Le temps moyen d'attente = $\frac{4+0}{2} = 2$

Il y a 3 changements de contexte.

Round robin (quantum = 10) :

A	A	A	A	A	A	A	A	A	A	B	B	B	B	A	A	A	A	A
1				5					10					15				

Processus	Temps de séjour
A	19-0 = 19
B	14-2 = 12

Temps moyen de séjour = $\frac{19+12}{2} = 15,5$

Processus	Temps d'attente
A	19-15 = 4
B	12-4 = 8

Le temps moyen d'attente = $\frac{4+8}{2} = 6$

[2] Le quantum était de 1 seconde dans les premières versions d'Unix.

Il y a 3 changements de contexte.

Round robin (quantum = 3) :

A	A	A	B	B	B	A	A	A	B	A	A	A	A	A	A	A	A	A
1				5					10					15				

Processus	Temps de séjour
A	19-0 = 19
B	10-2 = 8

Temps moyen de séjour = $\frac{19+8}{2} = 13,5$

Processus	Temps d'attente
A	19-15 = 4
B	8-4 = 4

Le temps moyen d'attente = $\frac{4+4}{2} = 4$

Il y a 5 changements de contexte.

Dans le trois solutions (SRT, RR Q_t=10 et RR Q_t=3), il y a 2 tâches exécutées dans 19 unités de temps, alors $19/2 = 9,5$ **unités de temps par processus**.

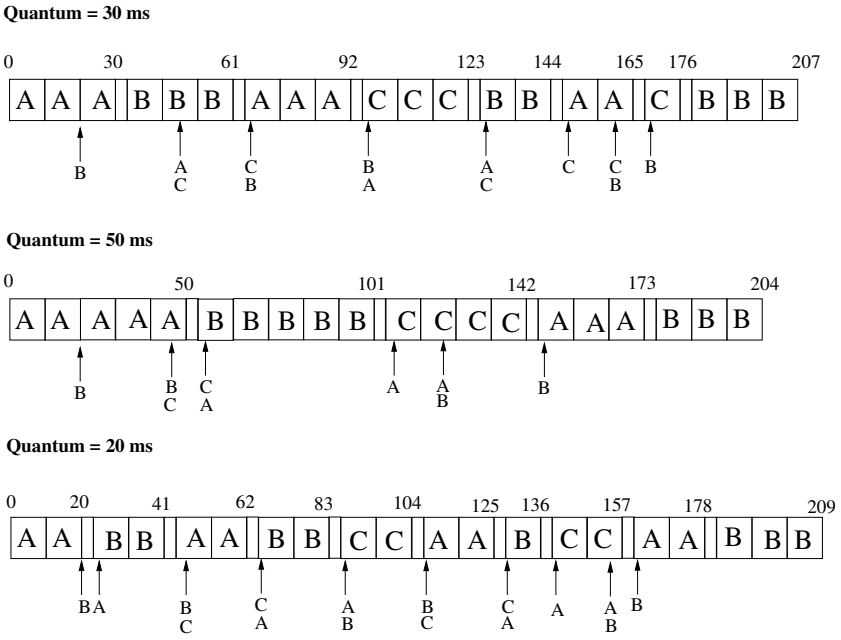

FIG. 8.2 – Diagrammes de Gantt pour différentes valeurs de quantum

▶ **Exemple 3.** Soient trois processus **A**, **B** et **C**. Le processus **A** arrive en premier suivi de **B** (20 msec après), puis **C** (46 msec après A). On suppose que l'exécution du processus **A** nécessite 80 msec de temps UCT. L'exécution du processus **B** nécessite d'abord 50 msec de temps UCT, bloquera ensuite durant 20 msec pour une entrée/sortie, puis exigera finalement 30 msec de temps UCT. L'exécution du processus **C** nécessite 40 msec de temps UCT. Le temps de changement de contexte est de 1 msec. La figure 8.2 montre les diagrammes de Gantt obtenus pour des quanta de 30 msec, 50 msec et 20 msec. La table 8.2 donne les temps de séjours moyens pour chaque situation.

Q_t	A	B	C	Ch. de contexte	Temps de séjour moyen
30	165	187	130	7	161
50	173	184	96	4	151
20	178	189	111	9	159

TAB. 8.2 – Solution des trois processus A, B, C.

8.5.3 Ordonnancement avec priorité

L'algorithme *round robin* permet une répartition équitable du processeur. Cependant il n'est pas intéressant si certains processus sont plus importants ou urgents que d'autres. L'**ordonnanceur à priorité** attribue à chaque processus une priorité. Le choix du processus à élire dépend des priorités des processus prêts.

Les processus de même priorité sont regroupés dans une file du type FIFO. Il y a autant de files qu'il y a de niveaux de priorité. L'ordonnanceur choisit le processus le plus prioritaire qui se trouve en tête de file. En général, les processus de même priorité sont ordonnancés selon l'algorithme du tourniquet.

Attribution et évolution des priorités

Pour empêcher les processus de priorité élevée de s'exécuter indéfiniment, l'ordonnanceur diminue régulièrement la priorité du processus en cours d'exécution.

La priorité du processus en cours est comparée régulièrement à celle du processus prêt le plus prioritaire (en tête de file). Lorsqu'elle devient inférieure, la commutation a lieu. Dans ce cas, le processus suspendu est

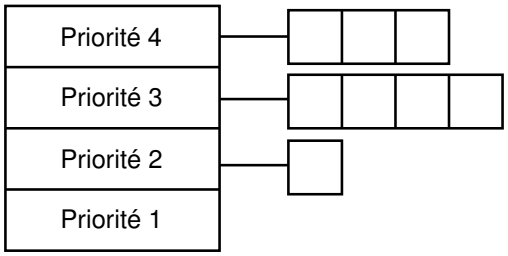

FIG. 8.3 – Ordonnancement avec priorité.

inséré en queue de file correspondant à sa nouvelle priorité. L'attribution et l'évolution des priorités dépendent des objectifs fixés et de beaucoup de paramètres.

▶ **Exemple 4.** Les processus qui font beaucoup d'E/S (qui sont souvent en attente) doivent acquérir le processeur dès qu'ils le demandent, afin de leur permettre de lancer leur requête suivante d'E/S. Lorsqu'un processus passe de l'état élu à l'état bloqué, sa priorité est recalculée. Sa nouvelle valeur est le rapport : quantum/temps réellement utilisé par le processus.

Les processus qui ont le plus grand rapport sont les plus prioritaires :

- Si le quantum = 100 ms et le temps utilisé = 2 ms, la nouvelle priorité est 50.
- Si le quantum = 100 ms et le temps utilisé = 50 ms, la nouvelle priorité est 2.

8.5.4 Files multiples (quantum variable)

Pour éviter qu'il y ait beaucoup de commutations pour les processus consommateurs de temps UCT, il est préférable d'allouer un plus grand quantum à ces processus. Lorsqu'un processus passe à l'état élu : Pour la première fois, le processeur lui est alloué pendant un quantum. Pour la seconde fois, le processeur lui est alloué pendant 2 quantum... Pour la n^{ieme} fois, le processeur lui est alloué pendant $2^{(n-1)}$ quantum.

Chaque processus a une priorité. Cette dernière dépend du nombre de quantum qui lui sera alloué lors de sa prochaine activation. Les processus dont le nombre de quantum est le plus petit sont les plus prioritaires. Les processus prêts sont répartis selon leur priorité dans des files (FIFO).

D'abord on fait l'élection du processus le plus prioritaire qui se trouve en tête de file. Lorsque l'exécution d'un processus est suspendue pour la

n-ième fois, sa priorité est recalculée (2^n) puis il est inséré à la queue de la file appropriée.

▶ **Exemple 5.** Considérons un processus qui doit effectuer des calculs pendant 100 quantum. Ce processus obtient successivement 1, 2, 4, 8, 16, 32 et 64 quantum. Mais il utilisera 37 des 64 derniers quantum. Le nombre de changements de contexte passe de 100 (cas du tourniquet) à 7. Le processeur passe moins de temps à commuter et a donc un meilleur temps de réponse. Surtout dans le cas où la commutation nécessite des transferts sur disque.

Un processus qui descend de plus en plus dans les files à priorité s'exécute de moins en moins fréquemment et favorise ainsi les processus courts en mode interactif. Pour ne pas défavoriser un processus qui s'était exécuté pendant assez longtemps avant de devenir interactif, on peut lui attribuer la plus haute priorité dès qu'un retour chariot est tapé sur le terminal associé au processus (les premières versions d'Unix utilisaient cette solution).

8.6 Ordonnancement à deux niveaux

Lors d'une commutation, si le processus élu n'est pas en mémoire, il faut le charger en mémoire. Le temps de commutation est deux à trois fois plus élevé que celui des processus qui se trouvent en mémoire centrale. Pour éviter ces va-et-vient entre le disque et la mémoire lors d'une commutation, l'**ordonnanceur à deux niveaux** déplace les processus entre le disque et la mémoire (haut niveau) et choisit le processus à exécuter parmi ceux qui sont en mémoire (bas niveau). Périodiquement, l'ordonnanceur de haut niveau retire de la mémoire les processus qui y sont restés assez longtemps et les remplace par des processus qui sont restés sur le disque pendant trop de temps. Pour élire un processus, l'ordonnanceur de bas niveau se restreint au processus en mémoire.

8.7 Cas d'étude

8.7.1 MS-DOS

MS-DOS est un système mono-utilisateur, et il n'est non plus un système multiprogrammé. Il peut y avoir, à un moment donné, plusieurs processus en mémoire mais un seul est à l'état prêt, tous les autres étant en

attente de la fin d'un fils. L'ordonnanceur des processus du système MS-DOS est très simple : il exécute un processus jusqu'à ce qu'il se termine ou crée un fils. Le processus créateur est suspendu jusqu'à ce que le processus créé se termine.

8.7.2 Unix

C'est un ordonnanceur à deux niveaux. L'ordonnanceur de bas niveau se charge d'élire un processus parmi ceux qui résident en mémoire. L'ordonnanceur de haut niveau se charge des transferts de processus entre la mémoire centrale et le disque. L'ordonnanceur de bas niveau utilise plusieurs files, une priorité est associée à chaque file (plusieurs niveaux de priorité). Les processus prêts qui sont en mémoire sont répartis dans les files selon leur priorité. Les priorités des processus s'exécutant en mode utilisateur sont positives ou nulles, alors que celles des processus s'exécutant en mode noyau sont négatives. Les priorités négatives sont les plus élevées, comme illustré à la figure 8.4.

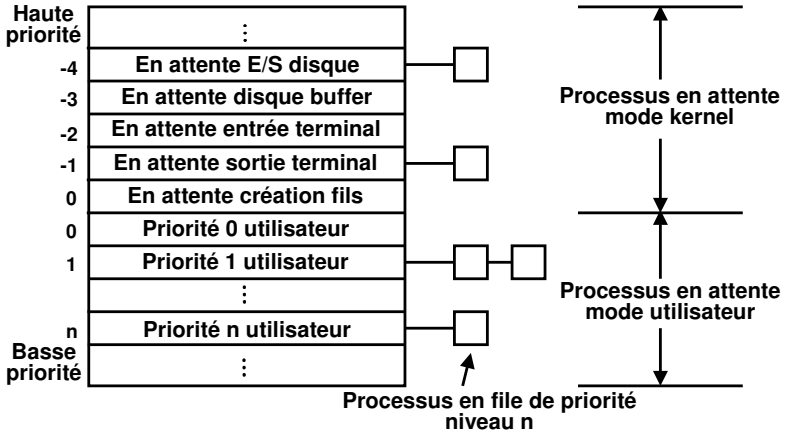

FIG. 8.4 – Ordonnancement sous Unix.

L'ordonnanceur de bas niveau choisit le processus le plus prioritaire qui se trouve en tête de file. Le processus élu est alors exécuté pendant au maximum un quantum (100 ms). S'il ne se termine pas ou ne se bloque pas au bout de ce quantum, il est suspendu. Le processus suspendu est inséré en queue de sa file. La priorité initiale d'un processus utilisateur est en général égale à 0. La commande `nice` permet d'attribuer une priorité plus basse à un processus (> 0). Les priorités des processus prêts en mémoire sont re-

calculées périodiquement toutes les secondes. La priorité d'un processus est recalculée, dans certaines versions, en ajoutant à sa valeur de base (en général 0 pour les processus utilisateurs), la moitié du temps UCT utilisé (temps en nombre de pulsations d'horloge) et la valeur de `nice`.

Lorsqu'un processus passe de l'état bloqué à l'état prêt, il se voit attribuer une forte priorité (négative). Après le calcul des priorités, les processus sont déplacés vers les files d'attente correspondant à leurs nouvelles priorités.

8.7.3 Linux

Linux offre trois politiques d'ordonnancement. Deux d'entre elles sont des politiques d'ordonnancement en temps réel "soft", c'est-à-dire qu'elles accordent aux processus une très haute priorité par rapport aux autre processus, mais ne garantissent pas une réponse dans un temps spécifié. La troisième politique d'ordonnancement utilise un algorithme de temps partagé basé sur des valeurs de priorité.

Par défaut, un processus est associé à la politique de temps partagé. Seul *root* peut associer un processus à une des classes d'ordonnancement en temps réel.

En Linux, chaque processus se voit attribuer une politique d'ordonnancement. Dans tous les cas, le processus possède aussi une valeur de priorité, variant de 1 à 40. Plus la valeur est élevée, plus la priorité est haute. Par défaut, un processus utilisateur a une valeur de priorité de 20.

Il est possible, pour un processus, de modifier sa priorité, en utilisant l'appel système `nice(valeur)`, où `valeur` est un nombre compris entre -20 et 20. Si la valeur est positive, on diminue d'autant la priorité du processus. Réciproquement, si la valeur est négative, on augmente la priorité. À noter que seul *root* peut augmenter la priorité d'un processus.

Temps réel

Une première politique d'ordonnancement, dénommée SCHED_FIFO, garantit au processus une utilisation illimitée du processeur. Il ne sera interrompu que dans une des circonstances suivantes :

- Le processus bloque sur un appel système ou se termine.
- Un autre processus de la classe SCHED_FIFO de priorité plus élevée est prêt. Dans ce cas le processus actuel est remplacé par celui-ci.

– Le processus libère lui-même le processeur, en exécutant l'appel système sched_yield().

Rien n'est plus prioritaire qu'un processus de la classe SCHED_FIFO, à l'exception d'un autre processus de la même classe qui possède une valeur de priorité supérieure.

L'autre politique d'ordonnancement en temps réel, dénommée SCHED_RR, est, contrairement à la première, préemptive. Chaque processus de cette classe se voit attribuer un quantum (temps limite d'exécution). Lorsque ce quantum sera écoulé, le contrôle sera donné à un autre processus de même priorité de la classe SCHED_RR, s'il y en a un, en appliquant l'algorithme du tourniquet. À noter que le tourniquet ne se fait qu'avec des processus de même priorité. Un processus temps réel de plus haute priorité a toujours préséance. Ainsi, si deux processus de la classe SCHED_RR avec priorité 20 s'exécutent, ils alterneront dans le processeur. Si entretemps apparaît un processus de la même classe, mais de priorité 25, c'est ce dernier qui prend le contrôle du processeur et ne le redonnera que lorsqu'il se terminera. À moins, bien sûr, que n'apparaisse un autre processus SCHED_RR de priorité supérieure ou égale, ou encore un processus SCHED_FIFO.

Le quantum attribué à un processus de la classe SCHED_RR est variable et établi selon les même principes que ceux appliqués aux processus à temps partagé, décrits à la section suivante.

Temps partagé

Nous avons vu, à la section précédente, les deux politiques d'ordonnancement em temps réel offertes par Linux. Il nous reste maintenant à voir la dernière polique d'ordonnancement, qui regroupe tous les processus de la classe OTHER. Les processus de cette classe se partagent le processeur de manière inégale, selon leur priorité et leur usage du processeur.

Premièrement, comme nous l'avons déjà dit, chaque processus possède une valeur de priorité qui lui est attribuée au moment de sa création. C'est ce que nous appellerons la *priorité statique*.

Initialement, on attribue à chaque processus un quantum dont la valeur utilise une unité de temps qui correspond normalement à 10ms. La valeur initiale du quantum est égale à la valeur de priorité. Ainsi, un processus de priorité 25 aura un quantum de 25 unités, ce qui correspond à 250 ms. Ce quantum est le temps alloué au processus. À chaque 10 ms, on diminue de 1 la valeur du quantum du processus en cours d'exécution dans le processeur.

Chaque fois que l'ordonnanceur est appelé, une note est attribuée à tous les processus. Cette note, comme nous le verrons à la section suivante, dépend à la fois de la priorité du processus et de la valeur actuelle de son quantum. C'est cette note qui permettra de déterminer quel processus prendra le contrôle du processeur.

Éventuellement, on peut arriver à une situation où tous les processus sont dans une des deux situations suivantes :

- Son quantum est 0. Il a écoulé tout le temps qui lui était alloué.
- Il est bloqué. Il n'a pas nécessairement épuisé son quantum.

Dans ce cas, tous les quanta (y compris les quanta des processus en attente qui ont encore une valeur non nulle) sont réajustés selon la formule suivante :

$$\text{Quantum} \leftarrow \text{Quantum}/2 + \text{ priorité}$$

Ceci a pour effet de favoriser les processus qui n'ont pas utilisé tout le temps qui leur est alloué. En effet, un processus qui n'a pas épuisé son quantum se retrouve avec une nouveau quantum plus élevé que l'ancien. On peut facilement vérifier que la valeur maximale du quantum est deux fois la priorité.

Comme nous le verrons dans la prochaine section, un processus qui voit son quantum augmenter peut se retrouver avec une meilleure note lorsque vient le moment de choisir un processus à exécuter.

Algorithme d'ordonnancement

Lorsque l'ordonnanceur est appelé, Linux attribue une note à chaque processus, en utilisant la méthode suivante :

```
Si le processus est de la classe SCHED_FIFO ou SCHED_RR
    Note = 1000 + priorité
Sinon
    Si Quantum > 0
        Note = Quantum + Priorité
    Sinon
        Note = 0
```

On remarquera qu'un processus membre d'une des deux classes de temps réel aura toujours priorité sur les autres. En effet, puisque le quantum ne peut dépasser le double de la priorité du processus, et que la valeur maximale de la priorité d'un processus est 40, on n'aura jamais une note supérieure à 120 pour un processus de la classe OTHER, ce qui est nettement inférieur au minimum de 1000 pour un processus em temps réel.

On remarquera aussi qu'un processus qui a écoulé tout son quantum reste en attente tant qu'il y a des processus qui peuvent s'exécuter. Comme nous l'avons déjà dit, il ne se verra attribuer un nouveau quantum que lorsque tous les autres processus auront épuisé leur quantum ou seront bloqués.

Exemple

Supposons trois processus A, B et C, tous de la classe OTHER, et dont les priorités sont les suivantes :

Processus	A	B	C
Priorité	20	18	10

Supposons qu'ils arrivent tous dans le système au même moment et qu'ils sont seuls. A et B sont des processus qui s'interrompent pour faire des appels système bloquant, alors que C ne bloque jamais.

Initialement, c'est évidemment le processus A qui a la meilleure note. C'est donc lui qui est exécuté, ayant droit à 200 ms. Supposons maintenant qu'il s'interrompt après 160 ms pour exécuter un appel système bloquant. Le système doit maintenant choisir entre B et C. B est élu et s'exécute pendant 40 ms (lui aussi bloque sur un appel système). À ce moment, le processus C prend le contrôle et utilise toutes le 100 ms qui lui sont accordées.

On se retrouve alors dans la situation suivante : A et B sont toujours bloqués, et C a un quantum nul. Le système réalisera donc un réajustement des quanta. Les processus se verront attribuer les nouvelles valeurs suivantes (rappelons qu'il reste 40 ms à A et 140 ms à B) :

Processus	A	B	C
Nouveau quantum	4/2 + 20 = 22	14/2 + 18 = 25	0/2 + 10 = 10

Comme A et B sont toujours bloqués, C s'exécute à nouveau. Supposons maintenant que A et B redeviennent prêts durant ce temps. Après 100 ms, C a épuisé son quantum. Il aura une note égale à zéro. Le système choisira alors entre A et B. Voici les notes qui sont attribuées aux processus :

Processus	A	B	C
Nouveau quantum	22 + 20 = 42	25 + 18 = 43	0

C'est donc B qui sera choisi, malgré sa priorité statique plus basse. Il est favorisé parce qu'il a utilisé une proportion plus petite du temps qui lui avait été alloué.

8.7.4 Windows

L'algorithme d'ordonnancement de Windows est semblable à celui du VAX/VMS : il est basé sur une priorité qui peut changer au cours de l'exécution des processus. Windows ordonne des threads, sans se préoccuper de savoir à quel processus ils appartiennent. On n'utilise pas un ordonnanceur autonome : quand un thread ne peut plus continuer à s'exécuter, il se met en mode superviseur pour exécuter la routine d'ordonnancement. S'il signale un objet (UP sur un sémaphore, par exemple), il vérifie s'il n'a pas ainsi réveillé un thread plus prioritaire auquel il doit céder sa place. Si le quantum est expiré et que le thread est sélectionné à nouveau, on lui attribue un nouveau quantum.

Deux autres situations exigent l'exécution de la routine d'ordonnancement :

- Une entrée/sortie vient d'être complétée
- Un timeout vient d'expirer (ce qui va en général réveiller un thread)

Un appel `SetPriorityClass` fixe la priorité de base pour tous les thread d'un processus. Les valeurs possibles sont (en ordre décroissant de priorité) : `realtime`, `high`, `above normal`, `normal`, `below normal` et `idle`. Un appel `SetThreadPriority` permet de fixer la priorité d'un thread par rapport aux autres du même processus. Les valeurs permises sont : `time critical`, `highest`, `above normal`, `normal`, `below normal`, `lowest` et `idle`. Selon la combinaison de ces deux attributs, on attribue à un thread une valeur de priorité entre 0 et 31, qu'on appelle **priorité de base**. En plus, chaque thread a une **priorié courante** qui varie dynamiquement : elle peut être plus haute que la priorité de base.

À chaque valeur de priorité une file d'attente est associée et Windows parcourt ces files de manière classique. Chaque file est traitée par l'algorithme du tourniquet. L'ordonnancement des threads se fait sans tenir compte du processus auquel il appartient. Le tableau des priorité est divisé en trois segments :

- Temps réel (16 à 31)
- Utilisateur (1 à 15)
- Thread 0 (0)

Seul le SE peut manipuler le segment temps réel. Le thread 0 n'est exécuté que lorsqu'il n'y a aucun autre thread : il initialise à zéro les pages de mémoire. S'il n'y a vraiment rien à faire, un thread idle s'exécute.

C'est la priorité courante d'un thread qui est considérée pour l'ordonnancement. La priorité courante d'un thread utilisateur, qui est initialement la priorité de base, peut être augmentée dans certaines circonstances :

- Le thread vient d'être libéré d'une attente d'entrée/sortie (ex. +1 pour disque, +6 pour clavier, +8 pour carte son)
- Le thread vient d'être libéré d'une attente sur un sémaphore (+2 pour thread d'un processus en premier plan et +1 pour les autres)

Lorsqu'un thread a expiré son quantum, sa priorité descend de 1 jusqu'à ce qu'il revienne à sa priorité de base.

Pour éviter les inversions de priorité, Windows utilise la technique suivante :

- Quand un thread n'a pas été exécuté depuis un temps qui dépasse un seuil préspécifié, il passe à la priorité 15 pour deux quanta
- Un thread non prioritaire mais qui détient un sémaphore aura donc la chance de le débloquer pour redonner le contrôle à un thread plus prioritaire en attente sur ce sémaphore

Finalement, un thread associé à une fenêtre qui vient d'être activée reçoit un plus gros quantum (typiquement 3 X quantum normal).

Suggestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Principles appliqués des systèmes d'exploitation avec Java*, Vuibert, Paris, France, 2002.

Chapître 6 : Ordonnancement d'exécution.

Chapître 20 : Le système Unix.

8.8 Exercices

1. Citez quatre événements qui provoquent l'interruption de l'exécution d'un processus en cours, dans le système Unix.
2. Quel est le rôle de l'ordonnanceur ?
3. Décrire brièvement l'ordonnanceur du système Unix. Favorise-t-il les processus interactifs ?
4. Expliquez une raison pour laquelle certains systèmes (tel Unix) assignent des priorités internes plutôt qu'externes aux processus.
5. Identifiez l'algorithme d'ordonnancement qui est le plus utilisé, et spécifiez la raison.
6. Identifiez l'algorithme d'ordonnancement sans réquisition qui fournit la meilleure performance. Pourquoi ?
7. Expliquez la différence fondamentale entre un algorithme d'ordonnancement à réquisition et sans réquisition. Lequel fournit la meilleure performance ?
8. Quel sera l'effet sur la performance du système d'un temps de quantum trop long et trop petit.
9. Expliquez la différence fondamentale existant entre un ordonnanceur de bas-niveau et un ordonnanceur de haut-niveau. Existe-il des systèmes d'exploitation qui utilisent un seul ordonnanceur ? Expliquez aussi.
10. Expliquez quel mécanisme permet à un processus d'effectuer un partage volontaire de l'UCT. Quels sont les avantages et désavantages du partage volontaire ?
11. Expliquez comment le système détermine le moment d'effectuer un changement de contexte entre deux processus. Durant un changement de contexte, quels sont les processus (utilisateur ou système) qui sont chargés et déchargés ?
12. Considérez la charge de processus montré sur la table 8.3 suivante. Pour chacun des algorithmes d'ordonnancement spécifiés ci-bas, dessinez un diagramme montrant l'état d'occupation de l'UCT, des périphériques, et des files d'attente. Aussi, calculez les temps de virement et d'attente. Supposez un temps de changement de contexte instantané (infiniment petit). Finalement, comparez les performances des divers algorithmes pour cette charge de processus.

    a) L'algorithme du premier-arrivé-premier-servi (PAPS).

Processus	Temps d'arrivée	Temps de service	Durée des E/S	Périodes des E/S
P0	0	9	2	5,9
P1	2	6	0	-
P2	5	5	1	4

TAB. 8.3 – Charge de processus.

b) L'algorithme du tourniquet avec un quantum de 5 unités de temps.

c) L'algorithme du tourniquet avec un quantum de 5 unités de temps et des niveaux de priorité externe de 3, 1, 2 respectivement pour P0, P1, P2.

d) L'algorithme du tourniquet avec un quantum de 5 unités de temps et deux UCTs fonctionnant en parallèle.

e) L'algorithme du tourniquet avec un quantum de 5 unités de temps et des niveaux de priorité interne initiaux de 20 respectivement pour P0, P1, P2. Au moment d'un changement de contexte, la priorité du processus débarquant diminue de 1 unité. Aussi, un processus qui est bloqué dans une file d'attente des E/S perd 2 unités de priorité. Si le processus n'attend pas pour effectuer son E/S alors il ne perd que 1 unité de priorité. Lorsqu'un processus dans la file d'attente de l'UCT a plus haute priorité qu'un processus en exécution, l'UCT est alors préempté avant la fin du quantum.