Système d'exploitation

III. Allocation du processeur

Kévin Perrot

Aix-Marseille Université

2014

Ce cours est issu des supports de Jean-Luc Massat en L3 informatique à Luminy.

Table des matières

1 Introduction

Le partage d'une machine entre plusieurs utilisateurs s'est très rapidement révélé nécessaire pour des raisons d'économie, de rentabilité et de convivialité. Sous cette hypothèse, le problème qui se pose alors, à chaque instant, pour chaque processeur, est de décider s'il doit poursuivre ou interrompre l'exécution du processus courant, et, dans le second cas, de déterminer le prochain processus à activer.

La règle utilisée pour effectuer ce choix est contenue dans l'algorithme d'ordonnancement, plus couramment appelé *ordonnanceur* (*scheduler*). Nous allons présenter ici quelques paramètres intervenant dans l'élaboration des divers algorithmes utilisés, en

les justifiant par l'idée directrice qui a motivé leur emploi. Il faut en fait considérer l'ordonnanceur sous trois aspects :

- Au niveau le plus élevé, le plus proche de l'utilisateur, sa fonction consiste à déterminer si un travail soumis doit être admis (tout travail admis devient un processus) ou pas. Ainsi, l'évaluation des priorités, la gestion des ressources nécessaires (règlement de conflits), le maintien de la charge du système en dessous du seuil d'écroulement et de son intégrité, en particulier en cas d'incidents imprévisibles, sont autant de tâches dont l'ordonnanceur devra s'acquitter avec inévitablement des répercussions sur les travaux soumis.
- Le rôle du niveau moyen est de déterminer l'ensemble des processus pouvant obtenir le contrôle. Autrement dit, il tient à jour les paramètres relatifs aux différents processus qui permettront de dégager ceux étant susceptibles de devenir actifs.
- Le niveau le plus bas, le plus proche du matériel, choisit parmi les processus prêts, en respectant les priorités, celui à qui le processeur va être alloué. C'est le *répartiteur* qui est toujours résident en mémoire.

Il faut bien comprendre que le fait d'allouer une ressource à un processus favorise celui-ci (au moins de façon temporaire). Ainsi, chaque rouage de l'ordonnanceur a pour effet d'appliquer une certaine politique envers les divers processus, et par conséquent, envers les utilisateurs. Or, c'est justement cette politique qui sera appelée a être éventuellement modifiée afin de satisfaire le plus grand nombre d'usagers. C'est la raison pour laquelle, ces différents points de choix devront être séparés du côté purement logique de l'ordonnanceur.

Sans donner plus de détails sur la façon d'évaluer la priorité d'un processus (ce sera l'objet du paragraphe suivant), le programme simplifié de l'ordonnanceur pourrait s'écrire :

```
pour toujours
    p := prioritaire ;
    tant que (état(p) ≠ prêt) faire
        p := suivant(p) ;
    fin-faire
    restaurer_le_contexte_de(p) ;
    donner_la_main_à(p) ;
fin-pour
```

Dans ce programme, on a supposé la liste ordonnée par priorité décroissante à partir de l'entrée « prioritaire », et de plus bouclée.

2 Stratégies d'ordonnancement de la CPU

2.1 Objectifs

Une politique d'ordonnancement doit :

1. être *équitable* : cette contrainte est satisfaite si tous les processus sont considérés de la même manière et qu'aucun n'est retardé indéfiniment ;
2. rendre le *débit maximum* : elle doit faire en sorte de satisfaire le plus grand nombre de demandes par unité de temps ;
3. pouvoir prendre en charge un *maximum d'utilisateurs interactifs* tout en assurant des temps de réponse acceptables ;

4. être *prédictible* : un même processus doit pouvoir s'exécuter dans un temps à peu près équivalent quelle que soit la charge du système ;
5. être la *moins coûteuse possible* afin de ne pas éprouver les performances générales du système en particulier dans les phases instables ;
6. avoir pour effet de *rationaliser la gestion des ressources* en :
   - recherchant une utilisation optimum ;
   - favorisant les tâches peu exigeantes en nombre et en qualité de ressources ;
   - évitant la famine (par exemple en augmentant la priorité au fur et à mesure que l'attente s'accroît).
7. mettre en œuvre des *priorités* fondées sur des critères pertinents ;
8. avoir la possibilité de *réajuster ces priorités*, soit de manière globale (nécessité de modularité), soit de manière ponctuelle au cours du temps (priorités dynamiques) ;
9. *favoriser* les processus ayant un *comportement souhaitable* ;
10. veiller à *ne pas accepter de nouveaux travaux* lorsque le système est en surcharge ; etc.

La liste des contraintes que nous venons d'énoncer est loin d'être exhaustive, mais suffit déjà à mettre en évidence les conflits (3/10, 4/8, 1/9,...). Ceci dénote donc une grande complexité dans la détermination des tâches éligibles, ce qui est justement en complète opposition avec le point 5.

Ceci explique en partie la raison pour laquelle les ordonnanceurs ne seront souvent qu'un ensemble de compromis très satisfaisants dans certains cas de figures, mais s'avérant moyennement, voire fortement critiquables dans d'autres circonstances. Ceci explique aussi pourquoi nous ne pourront pas vous présenter ensuite L'Ordonnanceur avec un grand O, mais une panoplie de réalisations dont chacune pourra s'avérer satisfaisante sur un site donné mais intolérable sur un autre.

2.2 Critères à considérer

Afin de pouvoir réaliser tout (ou plutôt partie) des objectifs présentés au paragraphe précédent, le mécanisme d'ordonnancement doit considérer :

1. *taux des entrées/sorties de chaque processus* : après que le processeur lui ait été alloué, ne l'utilise-t-il qu'un temps très court avant de réclamer un échange ?
2. *taux d'utilisation du processeur* : pour chaque processus, lorsque le processeur lui est alloué, l'utilise-t-il pendant toute la tranche de temps impartie ?
3. *fonctionnement interactif* ou *traitement par lots* : les utilisateurs interactifs émettent généralement des requêtes simples qui doivent être satisfaites très rapidement, alors que les utilisateurs « batch », n'étant pas présents peuvent subir des délais (ceux-ci devant toutefois rester dans des limites raisonnables) ;
4. *degré d'urgence* : un processus « batch » ne requiert pas de réponse immédiate alors qu'un processus « temps réel » nécessite des réponses très rapides ;
5. *priorité des processus* : les processus de forte priorité doivent bénéficier d'un meilleur traitement que ceux de priorité plus faible ;

6. *taux de réquisition* : lorsqu'un processus est de faible priorité par rapport aux autres, il en découle un fort taux de réquisition. Dans ces conditions, le système doit-il essayer de l'avantager ou au contraire attendre que les priorités redeviennent comparables afin d'éviter les temps de commutation effectués en pure perte, vu la forte probabilité de réquisition.
7. *temps cumulé d'allocation du processeur* : doit-on pénaliser un processus ayant bénéficié d'un temps d'exécution important ou au contraire le favoriser car on est en droit de penser qu'il va bientôt s'achever ? Cette question revêt une importance particulière lorsque le système est en surcharge ;
8. *temps d'exécution restant* : le temps d'attente moyen peut être réduit en exécutant de préférence les processus réclamant un temps CPU minimum pour s'achever. hélas, l'évaluation de ce temps restant est rarement possible.

2.3 Réquisition ou pas ?

On dit qu'un ordonnanceur n'opère pas de réquisition si dès lors qu'il a alloué la CPU à un processus, il lui est impossible de le lui retirer. Réciproquement, une politique d'ordonnancement autorise la réquisition si le contrôle peut être retiré à tout moment au processus actif.

Cette possibilité est absolument et trivialement nécessaire sur les sites supportant des applications « temps réel ». C'est la contrainte liée aux temps de réponse qui rend cette technique indispensable dans les systèmes interactifs. Mais la réquisition engendre un surcoût non négligeable à l'exploitation : coût en temps occasionné par les changements de contexte incessants, coût en espace engendré par la nécessité de partager la mémoire entre tous les processus.

La non-réquisition est gênante pour les travaux courts lorsqu'ils doivent attendre qu'un travail très long s'achève ; mais globalement la philosophie semble plus équitable, les temps de réponse sont mieux prévisibles car l'arrivée de travaux à forte priorité ne vient pas perturber l'ordre des travaux en attente.

La réalisation d'un ordonnanceur à réquisition est, nous l'avons déjà dit, très délicate. En particulier, le calcul des priorités ne doit pas voir l'aspect sophistiqué l'emporter sur le côté signifiant. « Rester simple est le maître-mot, mais si cela n'est pas possible, il faut au moins insister pour demeurer effectif et pertinent dans les choix ! »

2.4 Intervalle de temps et interruption d'horloge

Le système dispose d'un moyen très simple pour retirer le contrôle à un processus. Un simple décompteur d'impulsions d'horloge, dont le calibrage peut être modifié, peut déclencher une interruption prioritaire qui aura pour conséquence d'appeler un traitant d'interruption du système d'exploitation.

Un processus utilisera donc le processeur jusqu'à ce qu'il le libère volontairement, ou qu'il y ait interruption d'horloge ou tout autre type d'interruption réclamant une intervention du système. Le système reprenant le contrôle pourra alors le passer à qui bon lui semble.

L'interruption d'horloge aide à garantir des temps de réponse acceptables dans un système interactif, évite au système de rester monopolisé dans une boucle de programme et permet en outre de traiter des applications temps réels. C'est donc une technique

simple, efficace et polyvalente qui toutefois demande une attention particulière pour le calibrage du décompteur.

2.5 Calibrage de la tranche de temps

La détermination de la tranche de temps ou *quantum* est critique dans un système. Doit-elle être longue ou courte ? Doit-elle être fixe ou variable ? Doit-elle être la même pour tous les utilisateurs ou déterminée séparément pour chacun d'eux ?

Aux conditions limites, selon que l'on fait tendre le quantum vers l'infini ou vers zéro, un processus s'achèvera sans que l'ordonnanceur soit intervenu ou au contraire, tout le temps CPU sera utilisé par l'ordonnanceur lui-même et aucun processus ne pourra se dérouler.

Afin d'ajuster le quantum à une valeur optimale, il nous faut considérer la courbe du temps de réponse moyen (Figure 1). Supposons qu'on ait le moyen de faire varier le quantum grâce à un curseur. Lorsque celui-ci est à zéro, l'ordonnanceur étant la seule tâche active, le temps de réponse est infini. Dès que nous tournons légèrement le curseur, augmentant ainsi la durée du quantum, le temps de réponse commence à diminuer. Si nous continuons, nous allons arriver à une position telle que si nous tournons encore légèrement le curseur, le temps de réponse va commencer à augmenter. Nous aurons atteint la valeur optimale. Si nous tournons le curseur « à fond », le temps de réponse va décroître pour le processus actif à ce moment là, puisqu'il va se terminer sans être interrompu, mais va considérablement augmenter pour tous les autres. Le temps de réponse moyen se stabilisera à un niveau médiocre fonction du nombre moyen de tâches en attente et du temps d'exécution moyen d'une tâche. On sera arrivé à l'algorithme FIFO que nous verront dans le paragraphe suivant.

Considérons donc la valeur optimale que nous avons obtenue précédemment. Elle représente en fait une petite fraction de seconde. Mais cette valeur est-elle vraiment bien adaptée à chacun des types de tâche. Est-elle assez grande pour que la majeure partie des requêtes interactives puissent être traitées en un seul quantum ? En particulier, si nous pouvons avoir une distribution moyenne dans le temps des demandes d'échange émises par m processus interactif, il serait avantageux que la valeur du quantum soit supérieure à l'intervalle entre deux échanges car cela diminuerait d'autant le nombre de réquisitions inutiles (il vaut mieux qu'un processus n'utilise pas tout son quantum et cède le contrôle à cause d'une demande d'échange, plutôt qu'il soit interrompu, attende « son tour », récupère le contrôle pour aussitôt lancer un échange).

En fait, on s'aperçoit que la valeur de ce quantum va varier d'un système à un autre, mais aussi en fonction du taux de charge. Il peut aussi dépendre du processus.

2.6 Priorités

Les priorités peuvent être allouées automatiquement par le système ou de manière externe. Elles peuvent être méritées ou acquises. Elles peuvent être statiques ou dynamiques. Elles peuvent être allouées de façon rationnelle ou arbitraire, en particulier lorsque le système est contraint de faire une distinction entre plusieurs processus sans avoir les moyens d'être sûr de faire le bon choix.

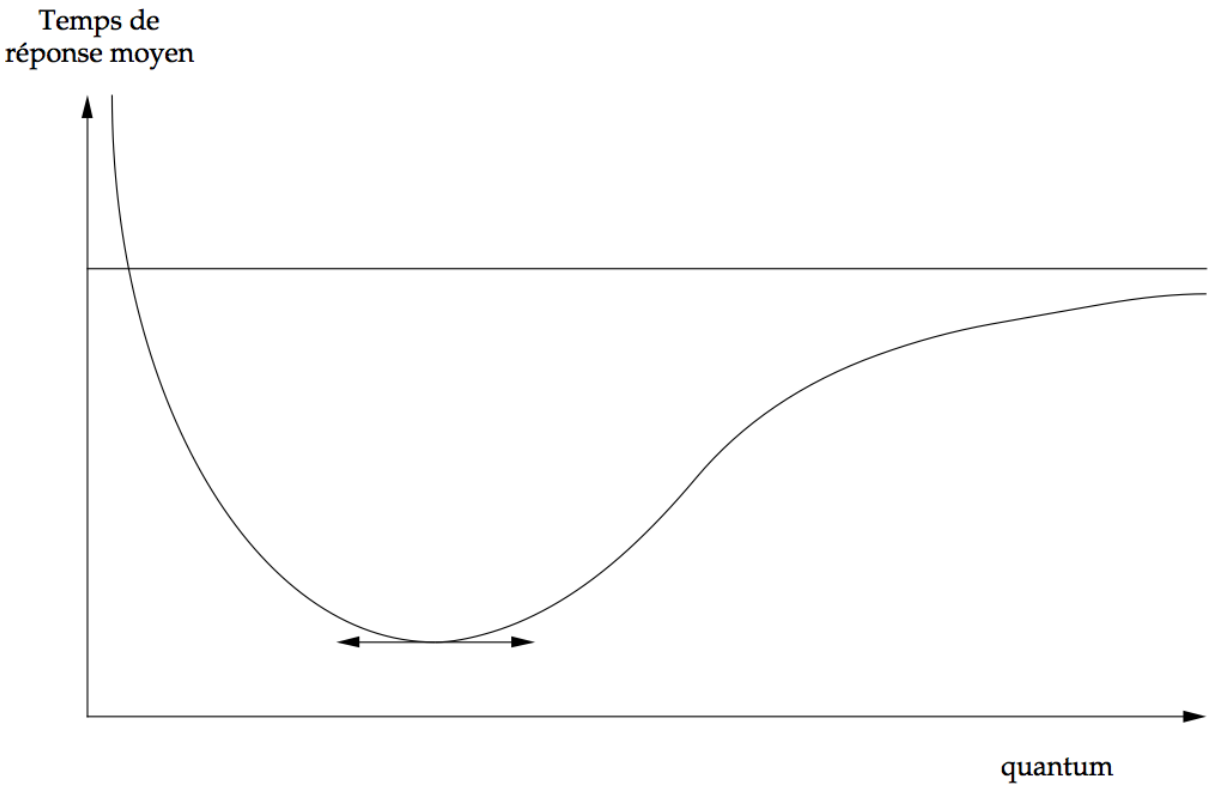

FIGURE 1 – Influence du quantum sur le temps de réponse moyen.

2.6.1 Priorités statiques et priorités dynamiques

Par définition, les *priorités statiques* ne changeant pas, elles sont d'une mise en œuvre facile et engendrent un faible coût d'exploitation. Toutefois, elles sont insensibles aux changements survenus dans l'environnement, changements qui justement peuvent nécessiter un ajustement des priorités.

A l'inverse, les *priorités dynamiques* peuvent changer en fonction de la modification de l'environnement. En particulier, nous verrons que la priorité initiale peut être réajustée très rapidement afin d'être mieux adaptée au type du processus considéré.

Il est bien évident que la gestion des priorités dynamiques est beaucoup plus complexe et engendre un coût beaucoup plus grand que celle des priorités statiques. En contre partie, leur emploi permet d'accroître considérablement le débit et la souplesse du système.

2.6.2 Priorités acquises

Un système doit offrir un service équitable et raisonnable (ou plutôt raisonnablement équitable...) à la majorité des utilisateurs d'un site. Mais il doit aussi pouvoir accepter qu'un usager bénéficie d'un traitement particulier. Celui-ci ayant, par exemple, un travail particulièrement urgent, peut désirer « payer un supplément de service » pour acquérir une priorité plus forte afin que son programme soit exécuté plus rapidement. Ce supplément se justifie sous deux aspect :

– tout d'abord, ce n'est que justice car vu sa priorité accrue, les ressources qu'il va utiliser (en particulier le processeur) seront enlevées à d'autres utilisateurs plus souvent que s'il avait conservé sa priorité normale. (Il faudrait toutefois s'assurer que le coût pour les autres usagers sera diminué en conséquence !...) ;

– l'autre aspect laisse supposer que les idées libérales se sont infiltrées jusque là car en effet, « si on ne faisait pas payer, tout le monde réclamerait un meilleur service », ce qui est bien évidemment inconcevable !...

3 Algorithmes d'ordonnancement

Nous allons, en fonction des problèmes que nous venons d'exposer et des solutions partielles qui ont été envisagées, présenter ici quelques réalisations d'ordonnanceurs en montrant pour chacune d'elles les avantages et les inconvénients envers le système et envers les utilisateurs.

3.1 Ordonnancement par échéance

Certains travaux peuvent être soumis accompagnés d'une date d'échéance. Là encore, cette option va entraîner un supplément d'autant plus important que l'échéance est proche de l'instant de lancement, à condition, bien sûr, que cette échéance soit respectée. Par contre, si ce n'est pas le cas, le service « supplémentaire » pourra être gratuit. Ce type d'ordonnancement est complexe pour plusieurs raisons :

– le système doit respecter l'échéance sans que cela implique pour autant une sévère dégradation de performances pour les autres utilisateurs ;

– le système doit planifier parfaitement l'utilisation des ressources toujours à cause de cette échéance fatidique. Or cela est particulièrement difficile car de nouveaux travaux peuvent arriver et émettre des demandes imprévisibles ;

– afin de limiter les ennuis du point précédent, l'utilisateur réclamant une échéance doit fournir au lancement la liste exhaustive des ressources qu'il utilisera, ce qui n'est pas évident, certaines étant « transparentes » pour lui (tampons d'E/S, canaux, etc) ;

– si plusieurs travaux à échéance sont lancés en même temps, l'ordonnancement va devenir tellement complexe que des méthodes d'optimisation très sophistiquées vont être nécessaires, d'où un coût d'exploitation très lourd ;

– ce surcroît de temps CPU utilisé par l'ordonnanceur ajouté aux faveurs accordées au(x) processus à échéance va inévitablement pénaliser les autres utilisateurs, ce qui risque d'engendrer des conflits de personnes. Ce facteurs doit être considéré avec une grande attention par les concepteurs des systèmes d'exploitation.

3.2 Premier arrivé, premier servi (FIFO)

La technique *FIFO* (*First In First Out*) est assurément la plus simple et de ce fait n'engendre qu'un très faible coût propre. C'est une technique sans réquisition (tout travail commencé se poursuit jusqu'à achèvement). L'ordre de priorité correspond de façon naturelle à l'ordre d'arrivée.

FIGURE 2 – L'ordonnanceur FIFO.

De ce point de vue, il est donc équitable, mais il faut toutefois regretter le fait que les longs travaux occasionnent une longue attente des travaux brefs qui suivent et, réciproquement, une multitude de petits travaux peut provoquer une longue attente de travaux importants.

FIFO présente une faible variance ; il est donc plus prédictible que la plupart des autres techniques. De toute évidence, il n'est pas utilisable dans les systèmes interactifs car il ne garantit pas un bon temps de réponse. C'est en particulier une des principales raisons qui font que cette technique est rarement utilisée aujourd'hui « à l'état brut ». Néanmoins, il faut noter qu'elle peut être associée à une technique plus sophistiquée qui accordera des priorités générales, les processus de même priorité étant considérés selon un schéma FIFO.

3.3 Tourniquet (Round Robbin)

L'ordonnancement de type *tourniquet* s'inspire de la technique FIFO, l'association d'une tranche de temps autorisant la réquisition. Les processus, au fur et à mesure qu'ils obtiennent le statut « prêt », sont rangés dans une file. A chacune de ses interventions, le distributeur alloue le contrôle au processus en tête de file. Si le temps d'exécution qui lui est ainsi imparti expire avant son achèvement, il est placé en queue de file et le contrôle est donné au processus suivant.

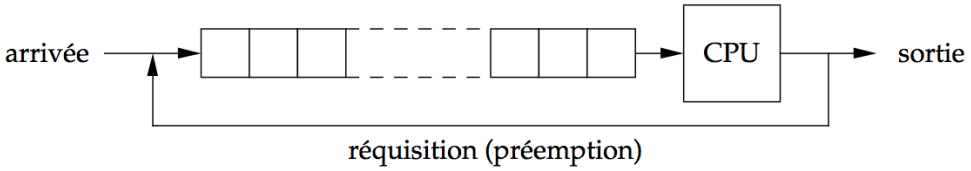

FIGURE 3 – Ordonnancement par « tourniquet ».

Cette technique est satisfaisante dans les systèmes « temps partagé » où les utilisateurs interactifs doivent bénéficier de temps de réponse corrects. Le coût de la réquisition peut être maintenu faible si les mécanismes de commutation sont efficaces et la mémoire suffisante pour contenir plusieurs processus simultanément. A noter aussi la nécessité d'un réglage judicieux du quantum pour accroître le taux d'utilisation du processeur et donc diminuer les temps de réponse (voir Figure 1).

3.4 Travail plus court d'abord (SJF)

La technique *Plus court d'abord* (SJF pour *Shortest Job First*) est encore un schéma d'ordonnancement sans réquisition (donc inutilisable en temps partagé) où le processus ayant le plus faible temps estimé d'exécution jusqu'à achèvement est prioritaire. C'est donc une technique qui a été créée pour pallier partiellement à l'inconvénient de FIFO qui autorisait l'exécution de travaux très longs avant des travaux de plus faible importance, seul leur ordre d'arrivée étant pris en compte.

SJF favorise donc les travaux brefs au détriment des plus importants. De ce fait il entraîne une variance beaucoup plus grande que FIFO, en particulier pour ce qui concerne les travaux longs. SJF fonctionne de façon à ce que la prochaine exécution puisse s'achever (et donc quitter le système) dès que possible. Cette technique tend donc à réduire le

nombre de travaux en attente, ce qui a pour conséquence de diminuer la moyenne des temps d'attente des processus.

Le principal inconvénient de SJF est qu'il requiert une connaissance précise du temps d'exécution, valeur qu'il n'est habituellement pas possible de déterminer. Le seul moyen est de se fier à une estimation donnée par les utilisateurs eux-mêmes. Cette estimation peut être bonne dans des environnements de production où les mêmes travaux sont soumis régulièrement, mais elle s'avère rarement possible dans les environnements de développement.

La connaissance de ce schéma d'ordonnancement pourrait tenter certains de sous-estimer volontairement le temps d'exécution afin de profiter d'une priorité indue. Afin d'éviter ce genre de « malhonnêteté », l'utilisateur est prévenu à l'avance que son travail sera abandonné en cas de dépassement. Cela présente deux inconvénients :

- obligation pour les usagers de majorer les estimations ;
- mauvaise rentabilité du processeur (le temps consacré aux travaux abandonnés faisant rapidement baisser le rendement).

Une seconde possibilité est donc offerte : poursuivre l'exécution du travail durant le temps estimé augmenté, si nécessaire, d'un certain pourcentage (faible en général) puis de le « mettre de côté » dans l'état où il se trouve pour reprendre son exécution plus tard. Bien entendu l'utilisateur sera pénalisé par cette attente mais aussi par un supplément de facturation.

Une troisième possibilité est de ne pas « mettre de côté » le travail, mais de poursuivre son exécution jusqu'à achèvement en facturant le temps excédent à un taux beaucoup plus élevé. Cette solution est finalement mieux acceptée car le supplément correspond effectivement à un meilleur service.

3.5 Plus court temps restant (SRT)

La stratégie *Plus court temps restant* (SRT pour *Shortest Remaining Time*) est la version avec réquisition de SJF (donc utilisable en temps partagé) où, là encore, priorité est donnée au processus dont le temps d'exécution restant est le plus faible (en considérant à chaque instant les nouveaux arrivants).

Dans SRT, un processus actif peut donc être interrompu au profit d'un nouveau processus ayant un temps d'exécution estimé plus court que le temps nécessaire à l'achèvement du premier. Là encore, et plus particulièrement du fait de la réquisition, le « designer » doit prévoir une dissuasion à l'égard des « malins » connaissant la stratégie d'ordonnancement.

Le coût de SRT est supérieur à celui de SJF : il doit tenir compte du temps déjà alloué aux processus en cours, effectuer les commutations à chaque arrivée d'un travail court qui sera exécuté immédiatement avant de reprendre le processus interrompu (à moins qu'un travail encore plus court ne survienne). Les travaux longs subissent une attente moyenne plus longue et une variance plus grande que dans SJF.

En théorie SRT devrait offrir un temps d'attente minimum, mais du fait de son coût d'exploitation propre, il se peut que dans certaines situations, SJF soit plus performant. Afin de réduire ce coût, on peut envisager plusieurs raffinements évitant la réquisition dans des cas limites :

- supposons que le processus en cours soit presque achevé et qu'un travail avec un temps d'exécution estimé très faible arrive. Doit-il y avoir réquisition ? On peut

dans ces cas de figure garantir à un processus en cours dont le temps d'exécution restant est inférieur à un seuil qu'il soit achevé quelles que soient les arrivées ;

– autre exemple : le processus actif a un temps d'exécution restant légèrement supérieur au temps estimé d'un travail arrivant. Ici encore, si SRT est appliqué « au pied de la lettre », il y a réquisition. Mais si le coût de cette réquisition est supérieur à la différence entre les deux temps estimés, cette décision devient absurde !

La conclusion de tout cela est que les « designers » de systèmes doivent évaluer avec beaucoup de précautions les coûts engendrés par des mécanismes sophistiqués car ils peuvent dans bien des cas aller à l'encontre du but recherché : le gain de temps.

3.6 Plus grand rapport ensuite (HRN)

En 1971, Brinch Hanssen propose la stratégie *Plus grand rapport ensuite* (HRN pour *Highest Response Ratio Next*). Elle corrige certains travers de SJF et en particulier le favoritisme excessif dont bénéficient les nouveaux travaux courts.

HRN peut être considéré avec ou sans réquisition. La priorité de chaque travail est fonction non seulement du temps de service, mais aussi du temps d'attente. Les priorités sont donc dynamiques et calculées par la formule :

$$\text{priorité} = \frac{\text{temps d'attente} \ + \ \text{temps de service}}{\text{temps de service}}$$

en choisissant de préférence les travaux courts si le niveau de priorité est identique. Ce système présente plusieurs avantages :

– Les travaux longs, bien qu'étant défavorisés, voient leur priorité augmenter au fur et à mesure de leur attente. Ils sont donc sûr de récupérer la CPU au bout d'un temps d'attente fini, ce qui élimine le risque de privation.

– Si on utilise HRN avec un mécanisme de réquisition de la CPU, les processus qui restent en sommeil un certain temps (après une demande d'E/S par exemple) voient leur priorité augmenter. Cette augmentation permet de leur allouer plus de temps CPU lors du réveil, ce qui est assez logique.

3.7 Tourniquet multi-niveaux

Nous avons vu les problèmes que posait dans SJF et SRT la difficulté de connaître à l'avance la quantité de temps CPU nécessaire à l'exécution d'un programme. Un travail fonctionnant essentiellement en entrée/sortie n'utilisera en fait la CPU que de courts instants. A l'opposé, un travail réclamant le contrôle en permanence monopolisera la CPU durant des heures si l'on suppose un schéma sans réquisition.

En fait, nous l'avons vu plus haut, il est préférable qu'un ordonnanceur :

– favorise les travaux courts ;

– favorise les travaux effectuant de nombreuses E/S (pour une bonne utilisation des unités externes) ;

– déterminer le plus rapidement possible la nature de chaque travail afin de le traiter en conséquence.

Le *tourniquet multi-niveaux* répond à ces attentes. Un nouveau processus est stocké en queue de la file de plus haut niveau. Il progresse dans cette file FIFO jusqu'à ce qu'il obtienne le contrôle. Si le processus s'achève ou libère la CPU pour une entrée/sortie ou une attente d'événement, il est placé en queue de la même file d'attente. Si le quantum expire avant, le processus est placé en queue de la file de niveau inférieur. Il deviendra à nouveau actif lorsqu'il parviendra en tête de cette file et à condition que celles de niveau supérieur soit vides. Ainsi, à chaque fois que le processus épuisera sa tranche de temps il passera en queue de la file de niveau inférieur à celle où il se trouvait jusqu'à ce qu'il atteigne la file de plus bas niveau.

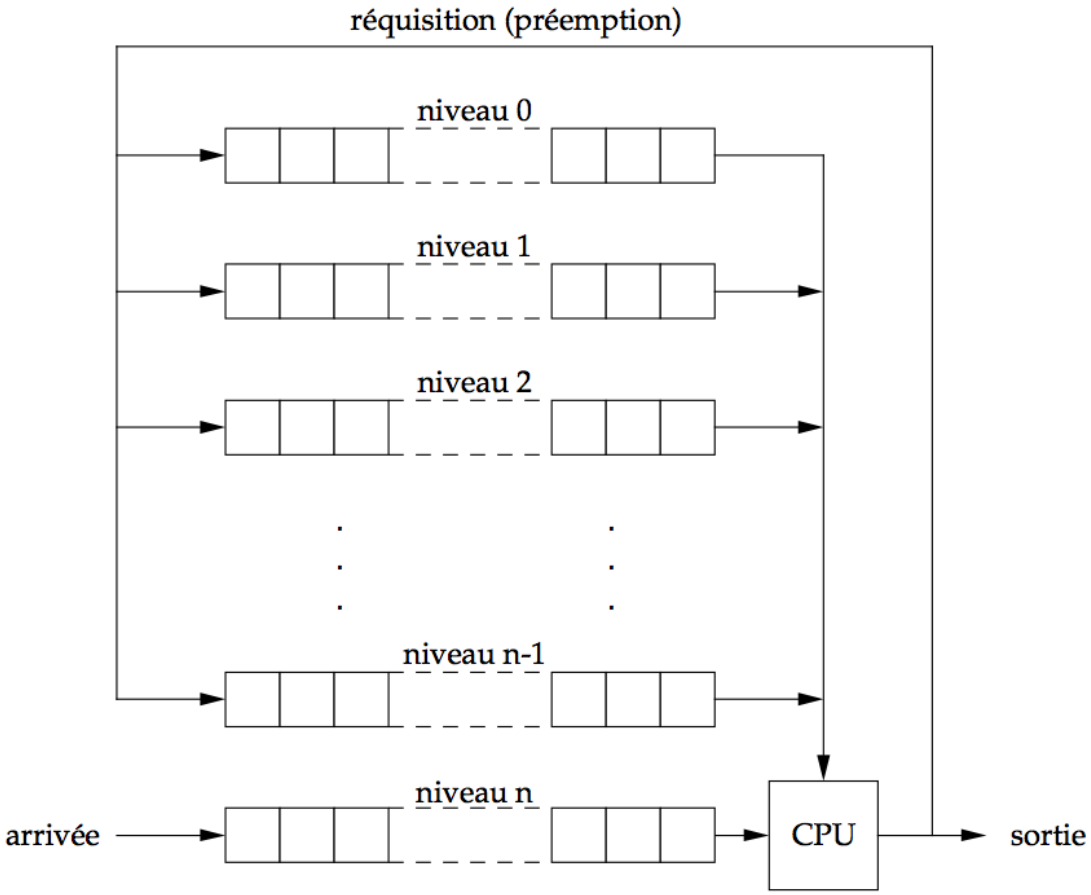

FIGURE 4 – Schéma de principe du tourniquet multi-niveaux.

Dans ce schéma d'ordonnancement, la taille du quantum s'accroît au fur et à mesure que l'on descend dans les niveaux de file. En conséquence, plus un travail est long, plus le temps CPU dont il bénéficie est grand. Mais en contre-partie, le processeur lui sera alloué plus rarement puisque les processus des files supérieures ont une plus grande priorité. Un processus en tête de quelque file que ce soit ne pourra devenir actif que si les files de niveau supérieur (si elles existent) sont vides. Il y aura réquisition dès qu'un travail arrivera dans la file de plus haut niveau. Considérons à présent la façon dont ce mécanisme s'adapte aux différents types de travaux.

Il favorisera les utilisateurs interactifs dont chaque requête sera envoyée dans la file prioritaire et satisfaite avant l'épuisement du quantum. De même, les travaux travaillant essentiellement en entrée/sortie seront avantagés si l'on suppose que le quantum de la file prioritaire est assez grand pour qu'une demande d'échange survienne avant qu'il expire. Dans ces conditions, dès que la demande d'échange se produit, le processus demandeur

est sorti de la file prioritaire et y reviendra lorsque cet échange sera effectué (bénéficiant ainsi entre chaque demande de la priorité accordée à la file de plus haut niveau).

En ce qui concerne un travail tendant à monopoliser la CPU, il débutera comme tous les autres dans la file la plus prioritaire, puis, très vite, il descendra les niveaux pour arriver dans la file la moins prioritaire, son quantum expirant à chaque étage. Là, il restera jusqu'à ce qu'il soit achevé mais... dans les hypothèses actuelles, que se passe-t-il si d'aventure un processus de ce type demande un échange ? Il est sorti de la file et lorsque l'échange aura été réalisé, il reviendra... dans la file prioritaire... à moins que le système retienne la file dont il était issu afin de l'y replacer ensuite.

Ce faisant, cette technique présuppose que le comportement passé d'un processus est une bonne indication de son comportement futur. Mais alors, un processus qui après une longue phase de calcul entre dans une phase où les échanges prédominent est désavantagé ! Ceci peut encore être résolu en associant au processus le dernier temps passé dans le réseau de files ou en convenant que tout processus montera d'un niveau dans le réseau chaque fois qu'il aura volontairement libéré la CPU avant expiration du quantum.

Le tourniquet multi-niveaux est un très bon exemple de mécanisme adaptatif. Bien sûr, le coût d'un tel ordonnanceur est supérieur à un qui n'a pas ces facultés d'adaptation, mais la meilleure adéquation de l'attitude du système vis à vis des différents types de travaux justifie amplement cette dépense. A noter une variante de ce système consistant à maintenir un processus plusieurs tours dans une même file avant qu'il passe au niveau inférieur. Habituellement ce nombre de tours s'accroît (comme la taille du quantum) en descendant dans les niveaux.