de cette matrice en appliquant les lois de la thermodynamique. Il s'agit de voir quelles sont les températures à l'issue d'un délai T. On répète le processus plusieurs fois, et on note les températures aux points échantillons, chronologiquement, à mesure que la feuille se réchauffe. L'algorithme produit alors une série de matrices étalées dans le temps.

Imaginons maintenant que notre matrice soit très volumineuse (1 million par 1 million), ce qui implique l'intervention de processus parallèles (éventuellement sur un ordinateur multiprocesseur) afin d'accélérer les calculs. Différents processus travaillent sur diverses portions de la matrice, calculant les nouveaux éléments de matrice à partir des anciens. Cependant, aucun processus ne peut démarrer sur l'itération $n + 1$ tant que l'itération n n'est pas terminée, autrement dit tant que tous les processus n'ont pas terminé leurs calculs. Pour atteindre un tel objectif, il faut programmer chaque processus de sorte qu'il exécute une opération `barrier` après en avoir fini avec l'itération en cours. Une fois que tous les processus ont terminé, la nouvelle matrice (l'entrée de la prochaine itération) est calculée, et tous les processus sont libérés simultanément pour démarrer l'itération suivante.

2.4 L'ordonnancement

Lorsqu'un ordinateur est multiprogrammé, il possède fréquemment plusieurs processus ou plusieurs threads en concurrence pour l'obtention de temps processeur. Cette situation se produit chaque fois que deux processus ou plus sont en état prêt en même temps. S'il n'y a qu'un seul processeur, un choix doit être fait quant au prochain processus à exécuter. La partie du système d'exploitation qui effectue ce choix se nomme l'**ordonnanceur** (*scheduler*) et l'algorithme qu'il emploie s'appelle **algorithme d'ordonnancement** (*scheduling algorithm*). Ces sujets seront traités au cours des sections qui vont suivre.

La plupart des problèmes applicables à l'ordonnancement des processus concernent également les threads. Mais certains sont différents. Lorsque c'est le noyau qui gère les threads, l'ordonnancement se fait généralement par thread, indépendamment du processus auquel appartient le thread. Nous commencerons par examiner ce qui concerne à la fois les processus et les threads. Puis nous aborderons les questions spécifiques aux threads. Le problème des puces multicœurs sera vu au chapitre 8.

2.4.1 Introduction à l'ordonnancement

Sur les anciens systèmes de traitement par lots (où les entrées prenaient la forme de transcriptions de cartes perforées sur une bande magnétique), l'algorithme d'ordonnancement était simple : on prenait le job suivant sur la bande. Avec les systèmes en temps partagé, l'algorithme d'ordonnancement est devenu plus complexe, car on trouve généralement plusieurs utilisateurs en attente de service. Certains mainframes combinent encore des services de traitement par lots et en temps partagé, ce qui oblige l'ordonnanceur à décider s'il va d'abord choisir un job de traitement par lots

ou un utilisateur interactif sur un terminal. Ajoutons à cela qu'un job de traitement par lots peut consister en une requête d'exécution de plusieurs programmes à la suite, mais dans cette section, nous supposerons qu'il s'agit d'une requête pour exécuter un seul programme. Le temps processus étant une ressource rare sur ces ordinateurs, un bon ordonnancement peut faire une grosse différence au niveau de la performance perçue par l'utilisateur. Par conséquent, d'énormes recherches ont été effectuées pour la mise au point d'algorithmes d'ordonnancement efficaces et intelligents.

Avec l'avènement des ordinateurs personnels, la situation a évolué de deux manières. Premièrement, il n'y a le plus souvent qu'un seul processus actif. Un utilisateur saisissant un document sur un traitement de texte ne va sans doute pas compiler un programme simultanément à l'arrière-plan. Lorsque l'utilisateur adresse une commande au traitement de texte, l'ordonnanceur n'a pas grand-chose à faire pour déterminer le processus à exécuter : le traitement de texte est le seul candidat.

Deuxièmement, les ordinateurs sont devenus si rapides avec les années que le processeur n'est plus une ressource rare. La plupart des programmes développés pour PC sont limités par la vitesse à laquelle l'utilisateur est capable de présenter des entrées (en tapant ou en cliquant), et non par celle à laquelle le processeur peut les traiter. Même les compilations – qui étaient des gouffres sans fond pour les vieux processeurs – se font aujourd'hui en l'espace de quelques secondes. Et même lorsque deux programmes, comme un traitement de texte et un tableur, s'exécutent simultanément, il importe peu de déterminer qui va avoir la priorité, car selon toute vraisemblance l'utilisateur va attendre la fin des deux processus pour continuer de travailler. L'ordonnancement n'a donc pas un rôle prépondérant à jouer sur la plupart des PC. Il existe toutefois des applications qui s'accaparent l'exclusivité du processus : le rendu d'une heure de vidéo haute résolution peut impliquer des solutions industrielles de traitement d'images pour chacune des 108 000 trames en NTSC (90 000 en PAL), mais ces applications font exception à la règle.

Si nous nous tournons vers les serveurs en réseau, la situation change. Ici, plusieurs processus se disputent du temps processeur, ce qui redonne tout son intérêt à l'ordonnancement. Par exemple, lorsque le processeur doit choisir entre exécuter un processus qui compile les statistiques du jour et un autre qui traite les requêtes utilisateur, on imagine bien le choix espéré par ces derniers.

Outre le fait de sélectionner le bon processus à exécuter, l'ordonnancement doit également se soucier de faire un usage efficace de l'UC, car les passages d'un processus à l'autre sont coûteux en termes de temps de traitement. Pour commencer, il faut procéder à un basculement du mode utilisateur vers le mode noyau. L'état du processus en cours doit être enregistré, y compris le stockage de ses registres dans la table des processus, afin qu'ils puissent être récupérés par la suite. Sur bien des systèmes, le mappage mémoire (les bits de référence de la mémoire dans la table des pages) doit également être enregistré. Après cela, la MMU doit être rechargée avec le mappage mémoire du nouveau processeur. Enfin, le nouveau processus doit démarrer. Pour couronner le tout, le basculement de processus invalide généralement la totalité du cache mémoire, obligeant son rechargement dynamique deux fois à partir de la mémoire principale (au moment d'entrer dans le noyau et au moment d'en sortir).

Vous avez compris que si les changements de processus sont trop nombreux, ils peuvent consommer un temps processeur notable, ce qui est suffisamment grave pour que l'on s'y attarde.

Comportement des processus

Pratiquement tous les processus alternent les rafales de traitement et les requêtes d'E/S (disque), comme le montre la figure 2.38. En général, le processeur s'exécute pendant un certain délai sans s'interrompre, puis un appel système est effectué pour accéder à un fichier en lecture ou en écriture. Lorsque l'appel système est achevé, le processeur recommence un traitement jusqu'à ce qu'il ait besoin de données supplémentaires ou qu'il lui faille écrire des données, et ainsi de suite. Remarquez que certaines activités d'E/S comptent comme des traitements. Par exemple, lorsque le processeur copie des bits dans une RAM vidéo pour rafraîchir l'écran, c'est du traitement, et non une opération d'E/S. En effet, le processus est sollicité. Dans cette acception, les E/S se produisent lorsqu'un processus entre en état bloqué, en attente qu'un périphérique externe ait terminé son travail.

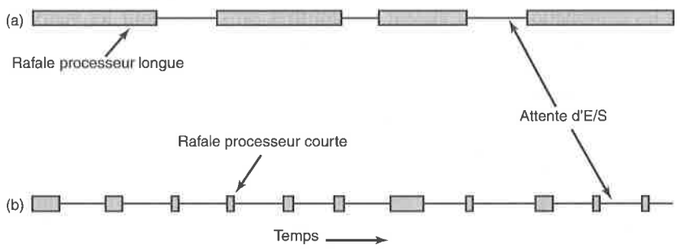

Figure 2.38 • Rafales d'utilisation de l'UC alternant avec des périodes d'attente d'E/S. (a) Un processus de traitement. (b) Un processus d'E/S.

Il est important de remarquer que certains processus, comme celui de la figure 2.38(a), passent le plus clair de leur temps à faire du traitement, tandis que d'autres, comme celui de la figure 2.38(b), sont le plus souvent en train d'attendre des E/S. Les premiers sont appelés **processus de traitement**, les seconds sont les **processus d'E/S**. Les processus de traitement présentent généralement des rafales longues d'utilisation de l'UC et de rares attentes d'E/S, tandis que les processus d'E/S présentent des rafales de traitement relativement brèves et de fréquentes attentes d'E/S. Remarquez que le facteur clé est la longueur de la rafale de traitement, et non celle de l'E/S. Les processus d'E/S sont ainsi qualifiés parce qu'ils font peu de traitement entre deux requêtes d'E/S, et non parce que leurs requêtes d'E/S sont spécialement longues à exécuter. Il faut le même temps pour émettre une demande de lecture d'un bloc de disque, que le temps de traitement ultérieur des données lues à cette occasion soit long ou pas.

À mesure que les processeurs deviennent plus rapides, les processus penchent de plus en plus vers une priorité donnée aux E/S. En effet, les processeurs progressent beaucoup plus vite que les disques en termes de performances. Ainsi, l'ordonnancement des processus d'E/S risque de devenir un sujet plus sensible à l'avenir. Pour pouvoir s'exécuter, ils doivent avoir des opportunités d'intervenir rapidement, de manière à émettre des requêtes disque et à utiliser le disque à son maximum. Comme nous l'avons vu à la figure 2.6, quand les processus font des E/S, un certain nombre est nécessaire pour que l'UC ait un taux d'occupation élevé.

Quand ordonnancer ?

Quand faut-il ordonnancer des processus ? En fait, un large éventail de situations nécessitent de l'ordonnancement. Premièrement, lorsqu'un nouveau processus est créé, il faut décider s'il faut exécuter d'abord le processus parent ou le processus enfant. Étant donné que les deux processus sont en état prêt, il s'agit là d'une décision d'ordonnancement normale et elle peut aller dans les deux sens : l'ordonnanceur peut légitimement choisir d'exécuter le fils ou le parent en premier.

Deuxièmement, il faut prendre une décision d'ordonnancement lorsqu'un processus se termine. Un autre processus doit être choisi dans le jeu de processus prêts. Si aucun processus n'est prêt, un processus d'inactivité fourni par le système s'exécute généralement.

Troisièmement, lorsqu'un processus bloque sur des E/S, un sémaphore ou autre, un autre processus doit être sélectionné pour être exécuté. Il peut arriver que la raison du blocage joue un rôle dans ce choix. Par exemple, si *A* est un processus important et qu'il attende que *B* quitte sa section critique, le fait de laisser *B* s'exécuter tout de suite après lui permet de quitter la section critique, et donc de laisser *A* s'exécuter. Mais le problème est souvent que l'ordonnanceur ne dispose pas des informations nécessaires pour prendre en compte cette dépendance.

Quatrièmement, lorsqu'une interruption d'E/S se produit, il faut également prendre une décision d'ordonnancement. Si l'interruption provient d'un périphérique d'E/S qui vient de terminer son travail, tout processus qui était bloqué en attente d'E/S peut désormais être exécuté. C'est à l'ordonnanceur de décider si le processus prêt doit être exécuté ou s'il faut donner la priorité à celui qui était en cours d'exécution au moment de l'interruption (ou encore à un autre processus quel qu'il soit).

Si l'horloge matérielle fournit des interruptions périodiques à une fréquence de 50 ou 60 Hz, par exemple, une décision d'ordonnancement peut être prise à chaque interruption d'horloge ou à chaque k^e interruption. On peut classer les algorithmes d'ordonnancement en deux catégories selon leur manière de réagir aux interruptions d'horloge. Un algorithme d'ordonnancement **non préemptif** sélectionne un processus, puis le laisse s'exécuter jusqu'à ce qu'il bloque (soit sur une E/S, soit en attente d'un autre processus) ou qu'il libère volontairement le processeur. Même s'il s'exécute pendant des heures, il ne sera pas suspendu de force. En effet, aucune décision d'ordonnancement n'intervient pendant les interruptions d'horloge.

Une fois le traitement de l'interruption terminé, le processus qui était en cours d'exécution avant l'interruption est toujours relancé.

Par opposition, un algorithme d'**ordonnancement préemptif** sélectionne un processus et le laisse s'exécuter pendant un délai déterminé. Si le processus est toujours en cours à l'issue de ce délai, il est suspendu, et l'ordonnancement sélectionne un autre processus à exécuter (s'il y en a un de disponible). L'ordonnancement préemptif nécessite une interruption à la fin du délai afin de redonner le contrôle de l'UC à l'ordonnanceur. En l'absence d'horloge, l'ordonnancement non préemptif est la seule solution.

Catégories d'algorithmes d'ordonnancement

Naturellement, des algorithmes d'ordonnancement différents interviennent dans des environnements divers. Cela est dû au fait que les différents domaines applicatifs (et les types de systèmes d'exploitation) ont des objectifs distincts. En d'autres termes, les aspects à optimiser par l'ordonnanceur ne sont pas les mêmes sur tous les systèmes. On distingue trois types d'environnements :

1. Traitement par lots.
2. Interactifs.
3. Temps réel.

Les traitements par lots sont encore très utilisés pour faire des payes, des états de stocks, de la comptabilité, du traitement de dossiers bancaires ou d'assurances, etc. Dans les systèmes de traitement par lots, il n'y a pas d'utilisateur attendant impatiemment des réponses rapides de la part de son terminal. Par conséquent, les algorithmes non préemptifs, ou les algorithmes préemptifs réservant de longs délais aux processus, sont souvent suffisants. Cette approche réduit le nombre de changements de processus, et donc améliore les performances. Les algorithmes de traitement par lots sont très généraux et s'appliquent à de nombreux autres cas que l'informatique ; de ce fait, ils font encore l'objet de nombreuses études.

Dans les environnements comptant des utilisateurs interactifs, la préemption est essentielle pour empêcher un processus donné de s'emparer de l'UC et d'en refuser l'accès à d'autres. Même si les processus ne s'exécutent pas indéfiniment de façon volontaire, un bogue du programme peut faire qu'un processus bloque tous les autres interminablement. La préemption est nécessaire pour éviter ces comportements. Les serveurs sont eux aussi dans cette catégorie puisqu'en principe ils servent de nombreux utilisateurs (distants), tous ces utilisateurs étant évidemment très pressés.

Dans les systèmes devant intégrer des contraintes de temps réel, la préemption est, curieusement, parfois inutile, car les processus savent qu'ils ne doivent pas s'exécuter pendant de longues périodes. Ils font leur travail et se bloquent rapidement. À la différence des systèmes interactifs, les systèmes en temps réel n'exécutent que des programmes dont l'objectif concourt à la réalisation de l'application. Les systèmes interactifs sont généralistes et peuvent exécuter arbitrairement des programmes peu coopératifs, voire nuisibles pour les autres.

Objectifs de l'algorithme d'ordonnancement

Pour concevoir un algorithme d'ordonnancement, il faut avoir une idée de ce qu'il est censé prendre en charge. Parmi ses objectifs, certains dépendent de l'environnement (traitement par lots, interactif ou en temps réel), mais d'autres sont souhaitables dans tous les cas. La figure 2.39 illustre certains de ces objectifs, qui seront encore détaillés par la suite. Dans tous les cas de figures, l'équité est importante. Des processus comparables doivent obtenir des services comparables. Il ne serait pas équitable d'allouer plus de temps à un processus qu'à un autre si les deux sont équivalents. Bien entendu, on peut traiter très différemment certaines catégories de processus. Par exemple, les processus de contrôle de la sécurité ou ceux d'établissement des salaires n'ont pas la même priorité pour le service informatique d'une centrale nucléaire.

L'application des politiques système relève également de l'équité. Si la politique locale édicte que les processus de contrôle de la sécurité puissent s'exécuter chaque fois qu'ils le souhaitent, même si la paye prend 30 secondes de retard, l'ordonnanceur doit faire en sorte que la politique soit respectée.

Un autre objectif consiste à exploiter toutes les parties du système chaque fois que c'est possible. Si le processeur et tous les périphériques d'E/S peuvent s'exécuter en permanence, il y aura plus de travail effectué à la seconde que si certains composants sont inactifs. Dans un système de traitement par lots, par exemple, l'ordonnancement contrôle les jobs qui doivent être placés en mémoire pour s'exécuter. Il est préférable d'avoir quelques processus liés au processeur et quelques autres liés aux E/S placés en mémoire en même temps, plutôt que de commencer par charger et exécuter tous les jobs processeur et d'attendre qu'ils aient terminé pour charger les jobs d'E/S. Si l'on emploie une telle stratégie, pendant que les processus liés au processeur s'exécutent, ils se disputent du temps processeur et le disque reste sous-utilisé. Plus tard, lorsque les jobs d'E/S arrivent, ils se disputent les accès disque, et le processeur est inactif. Il vaut mieux que l'ensemble des opérations s'effectuent simultanément au moyen d'un mélange judicieux de processus.

Figure 2.39 • Voici certains objectifs de l'algorithme d'ordonnancement, en fonction des circonstances

Tous les systèmes
Équité : attribuer à chaque processus un temps processeur équitable
Application de la politique : faire en sorte que la politique définie soit bien appliquée
Équilibre : faire en sorte que toutes les parties du système soient occupées
Systèmes de traitement par lots
Capacité de traitement : optimiser le nombre de jobs à l'heure
Délai de rotation : réduire le délai entre la soumission et l'achèvement
Utilisation de l'UC : faire en sorte que le processeur soit occupé en permanence

Figure 2.39 • Voici certains objectifs de l'algorithme d'ordonnancement, en fonction des circonstances *(suite)*.

Systèmes interactifs
Temps de réponse : répondre rapidement aux requêtes
Proportionnalité : répondre aux attentes des utilisateurs
Systèmes temps réel
Respecter les délais : éviter de perdre des données
Prévisibilité : éviter la dégradation de la qualité dans les systèmes multimédias

Les responsables de gros centres de traitement informatique, qui exécutent de nombreuses tâches de traitement par lots, se concentrent généralement sur trois unités de mesure pour savoir si leurs systèmes sont performants : capacité de traitement, délai de rotation et utilisation de l'UC. La **capacité de traitement** (*throughput*) est le nombre de jobs menés à bien par le système, à l'heure. On ne peut nier qu'il soit préférable de terminer 50 jobs à l'heure plutôt que 40. Le **délai de rotation** est le temps moyen statistique qui s'écoule entre le moment où un job est soumis et celui où il est terminé. Il permet de mesurer combien de temps l'utilisateur moyen doit attendre une sortie donnée. Ici, une seule règle s'applique : ce délai doit être le plus court possible.

Un algorithme d'ordonnancement qui optimise la capacité de traitement ne réduit pas nécessairement le délai de rotation. Par exemple, prenons un mélange de jobs courts et longs. Un ordonnanceur qui exécuterait toujours les jobs courts et jamais les jobs longs pourrait afficher une excellente capacité de traitement (au nombre de jobs à l'heure), mais au détriment d'un délai de rotation médiocre pour les jobs longs. Si les jobs courts continuent de se présenter régulièrement, les jobs longs risquent de ne jamais s'exécuter, ce qui peut donner un délai de rotation infini malgré une capacité de traitement apparemment enviable.

L'utilisation de l'UC fait souvent l'objet d'une étude particulière pour les systèmes de traitement par lots. Or, ce n'est pas là non plus une bonne unité de mesure. Ce qui importe vraiment, c'est le nombre de jobs achevés par heure (capacité de traitement) et combien de temps il faut pour qu'un job soit terminé (délai de rotation). Mesurer les performances d'un système en fonction de l'utilisation de l'UC revient un peu à classer des voitures en fonction du nombre de tours/minute du moteur. D'un autre côté, on peut quand même dire qu'il est utile de savoir que le taux d'utilisation de l'UC approche les 100 %, car cela signifie qu'il est temps d'acquérir un processeur plus puissant.

Pour les systèmes interactifs, on retiendra des objectifs différents. Le plus important est de réduire le **temps de réponse** (ou réactivité), c'est-à-dire le délai qui s'écoule entre l'émission d'une commande et la réception du résultat attendu. Sur un PC qui exécute un processus d'arrière-plan (par exemple, qui lit et stocke du courrier électronique récupéré sur le réseau), une requête de démarrage d'un programme ou d'ouverture d'un fichier doit avoir la préséance sur le processus d'arrière-plan.

Le fait de traiter prioritairement toutes les requêtes interactives sera perçu comme un bon service.

La **proportionnalité** relève un peu de la même préoccupation. Les utilisateurs se font une idée assez précise (mais souvent incorrecte) du temps que doivent prendre leurs différentes activités. S'ils perçoivent une requête comme une action complexe, ils acceptent d'attendre. Mais si une tâche leur paraît simple, ils sont irrités de devoir patienter. Par exemple, si le fait de cliquer sur une icône envoie un fax et que l'opération prend 60 secondes, l'utilisateur va trouver cela normal parce qu'il ne s'attend pas à ce que l'envoi d'un fax ne prenne que cinq secondes.

En revanche, s'il clique sur une icône pour couper la connexion et que cela prend 45 secondes, notre utilisateur va perdre patience. Ce comportement est dû à la perception courante qu'a l'utilisateur du temps que cela prend pour envoyer un fax d'une part, pour raccrocher le téléphone d'autre part. Or, dans certains cas de figure, l'ordonnanceur ne peut rien faire pour améliorer le temps de réponse. Mais, ce n'est pas toujours le cas, et il peut parfois optimiser l'opération si l'attente est due à un mauvais choix quant à l'ordre des processus.

Les systèmes temps réel ont des propriétés différentes de celles des systèmes interactifs, et donc des objectifs d'ordonnancement différents. Ils sont caractérisés par des délais butoirs qu'ils sont censés atteindre. Par exemple, si un ordinateur contrôle un périphérique qui produit des données à un débit régulier, un échec d'exécution du processus de collecte des données dans les délais impartis peut résulter en une perte de données. Par conséquent, l'impératif absolu d'un système temps réel consiste à respecter les délais (ou du moins la majorité d'entre eux).

Sur certains systèmes temps réel, et notamment ceux qui font du traitement multimédia, la prévisibilité est importante. Si le processus audio s'exécute de façon trop erratique, la qualité du son se détériorera rapidement. La vidéo est également en cause, mais l'ouïe est nettement plus sensible aux erreurs que la vue. Pour éviter ce type d'erreur, l'ordonnancement des processus doit être fortement prévisible et régulier. Dans ce chapitre, nous étudierons les algorithmes d'ordonnancement pour les systèmes de traitement par lots et les systèmes interactifs. Nous étudierons l'ordonnancement des systèmes temps réel au chapitre 7, lorsque nous en viendrons aux systèmes d'exploitation multimédias et à l'ordonnancement d'activités temps réel.

2.4.2 L'ordonnancement sur les systèmes de traitement par lots

Le moment est venu de passer des considérations générales aux algorithmes d'ordonnancement spécifiques. Dans cette section, nous verrons les algorithmes utilisés dans les systèmes de traitement par lots. Dans les sections suivantes, nous verrons ceux des systèmes interactifs et temps réel. Il faut noter que certains algorithmes sont exploités à la fois sur les systèmes de traitement par lots et interactifs. Nous les étudierons un peu plus loin dans ce chapitre.

Premier arrivé, premier servi

Le plus simple de tous les algorithmes d'ordonnancement est sans doute celui du **premier arrivé, premier servi** (*first-come first-served*). Avec cet algorithme non préemptif, les processus se voient attribuer du temps processeur selon leur ordre d'arrivée. Globalement, il n'existe qu'une seule file d'attente de processus prêts. Lorsque le premier job du jour arrive sur le système, il est démarré immédiatement et autorisé à s'exécuter aussi longtemps qu'il le souhaite. On ne l'interrompra donc pas, même s'il est très long. À mesure que d'autres jobs arrivent, ils prennent place dans la file d'attente. Lorsque le processus en cours d'exécution se bloque, le premier processus de la file est exécuté. Lorsqu'un processus bloqué redevient prêt, il est placé en queue de file d'attente, tout comme un nouveau job.

La grande force de cet algorithme est qu'il est facile à comprendre et tout aussi facile à programmer. Il assure une certaine équité, un peu comme un service de billetterie pour les matchs de football ou les places de concert pour ceux qui font l'effort de se lever à 2 heures du matin. Avec cet algorithme, une seule liste chaînée effectue le suivi de tous les processus prêts. Le fait de sélectionner un processus pour l'exécuter se résume à prélever le premier processus dans la file. L'ajout d'un nouveau job ou d'un processus non bloqué consiste simplement à le placer en queue de file. Que rêver de plus simple à comprendre et à implémenter ?

Malheureusement, le mécanisme du premier arrivé, premier servi souffre d'un inconvénient majeur. Prenons un cas de figure dans lequel un processus de traitement s'exécute à raison d'une seconde à la fois, et où de nombreux processus d'E/S utilisent peu de temps processus, mais doivent chacun effectuer 1 000 lectures disque pour se terminer. Le processus de traitement s'exécute pendant une seconde, puis il lit un bloc de disque. Tous les processus d'E/S s'exécutent alors et commencent à faire des lectures disque. Lorsque le processus de traitement récupère son bloc de disque, il s'exécute encore une fois pendant une seconde, suivi de tous les processus d'E/S en rapide succession.

Il en résulte que chaque processus d'E/S arrive à lire un bloc par seconde et prend 1 000 secondes pour se terminer. Avec un algorithme d'ordonnancement qui préempterait le processus de traitement toutes les 10 millisecondes, les processus d'E/S se termineraient en 10 secondes au lieu de 1 000, et ce sans ralentir le processus de traitement de façon palpable.

Exécution du job le plus court en premier

Voyons maintenant un autre algorithme de traitement par lots non préemptif qui suppose que les délais d'exécution soient connus par avance. Par exemple, dans une compagnie d'assurances, les responsables informatiques peuvent prédire assez précisément combien de temps il faut pour exécuter un lot de 1 000 sinistres, étant donné que des tâches analogues sont effectuées tous les jours. Lorsque plusieurs jobs d'importance égale se trouvent dans la file d'attente, l'ordonnanceur prélève le **job le plus court en premier** (*shortest job first*). Examinez la figure 2.40. Vous y voyez quatre jobs appelés *A*, *B*, *C* et *D*, avec des temps d'exécution respectifs de 8, 4, 4 et 4 minutes.

Si on les exécute dans cet ordre, le délai de rotation de *A* est de 8 minutes, celui de *B* de 12 minutes, celui de *C* de 16 minutes et celui de *D* de 20 minutes, pour un délai moyen de 14 minutes.

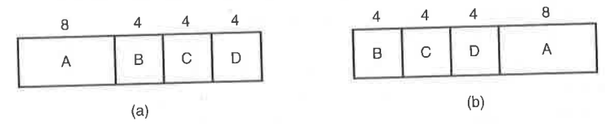

Figure 2.40 • Un exemple d'ordonnancement avec le job le plus court en premier. (a) Exécution des quatre jobs dans leur ordre original. (b) Exécution du job le plus court en premier.

Voyons ce qui se passe si l'on exécute ces quatre jobs en commençant par le plus court, comme à la figure 2.40(b). Le délai de rotation est maintenant de 4, 8, 12 et 20 minutes pour une moyenne de 11 minutes. La solution est donc optimale. Prenons l'exemple de quatre jobs, avec des délais d'exécution *a*, *b*, *c* et *d*, respectivement. Le premier job se termine à l'instant *a*, le deuxième à l'instant $a + b$, et ainsi de suite. Le délai de rotation moyen est de $(4a + 3b + 2c + d) / 4$. Il est évident que *a* contribue plus à la moyenne que les autres délais. Ce devrait donc être le job le plus court, suivi de *b*, de *c* et enfin de *d* – qui est le plus long et n'affecte que son propre délai de rotation. Ce même argument s'applique quel que soit le nombre de jobs. Il faut noter que cet algorithme n'est optimal que lorsque tous les jobs sont disponibles simultanément. Prenons un contre-exemple, où cinq jobs, de *A* à *E*, ont des temps d'exécution respectifs de 2, 4, 1, 1 et 1 minutes. Leurs instants d'arrivée sont : 0, 0, 3, 3 et 3. Au départ, seuls *A* et *B* peuvent être choisis, puisque les autres jobs ne sont pas encore arrivés. En prenant le job le plus court en premier, ceux-ci seront exécutés dans l'ordre : *A*, *B*, *C*, *D*, *E*. La moyenne sera de 4,6. Cependant, si on les exécute dans l'ordre *B*, *C*, *D*, *E*, *A*, la moyenne sera de 4,4.

Exécution du temps restant suivant le plus court

Il s'agit là d'une version préemptive de l'algorithme consistant à prélever le **job dont le temps d'exécution restant est le plus court parmi ceux qui restent à exécuter** (*shortest remaining time next*). Avec cet algorithme, l'ordonnanceur choisit toujours le processus dont le temps d'exécution restant est le plus court. Encore une fois, le temps d'exécution doit être connu par avance. Lorsqu'un nouveau job arrive, son temps total est comparé au temps restant pour le processus en cours. Si le nouveau job est plus court à exécuter que le processus en cours, ce dernier est suspendu et le nouveau job est lancé. Ce schéma favorise le service des jobs courts.

2.4.3 L'ordonnancement des systèmes interactifs

Nous allons maintenant aborder quelques algorithmes que l'on peut utiliser dans les systèmes interactifs. Ils sont très courants sur les ordinateurs personnels, les serveurs et autres systèmes de ce genre.

Ordonnancement de type tourniquet (*round robin*)

L'un des plus anciens, des plus simples et des plus équitables, est également le plus utilisé. Il s'agit de l'**algorithme de type tourniquet** (ou *round robin*). Chaque processus se voit assigner un intervalle de temps, appelé **quantum**, pendant lequel il est autorisé à s'exécuter. Si à l'issue de son quantum, le processus est toujours en train de s'exécuter, le processeur est préempté et attribué à un autre processus. Si le processus a bloqué ou s'est terminé avant la fin de son quantum, le basculement de l'UC s'effectue naturellement. L'algorithme de type tourniquet est aisé à implémenter. L'ordonnanceur maintient une liste de processus exécutables, comme le montre la figure 2.41(a). Lorsque le processus a utilisé son quantum, il est replacé en queue de liste, comme à la figure 2.41(b).

Ici, la question intéressante est celle de la durée du quantum. Le changement de processus prend un certain délai d'administration : enregistrement et chargement des registres et des mappages mémoire, actualisation des tableaux et listes, vidage et chargement du cache mémoire, etc. Supposons que ce **changement de processus**, ou **changement de contexte** comme on le nomme parfois, prenne 1 ms pour l'ensemble des opérations. Supposons également que le quantum soit défini à 4 ms. Avec ces paramètres, à l'issue de 4 ms de temps de traitement utile, le processeur doit passer (gaspiller) 1 ms à changer de processus. Cela revient à gaspiller 20 % du temps processeur avec des tâches d'administration. Bien évidemment, c'est trop.

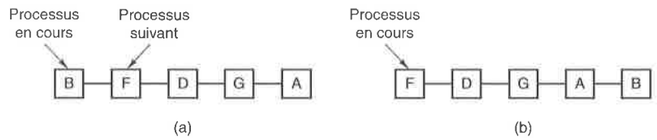

Figure 2.41 • Ordonnancement de type tourniquet. (a) La liste des processus exécutables. (b) La liste des processus exécutables après que B a utilisé son quantum.

Pour améliorer l'efficacité de l'UC, on peut définir le quantum à, disons, 100 ms. Le temps perdu n'est plus que de 1 %. Mais que se passe-t-il sur un serveur si 50 requêtes arrivent en un temps très court et qui demandent toutes des temps de calcul très différents ? 50 processus sont placés dans la liste des processus exécutables. Si le processeur est inactif, le premier va démarrer immédiatement, le deuxième ne pourra pas démarrer avant 100 ms, et ainsi de suite. Le malheureux dernier devra attendre cinq secondes avant d'avoir sa chance, à supposer que tous les autres utilisent entièrement leurs quanta respectifs. Cette situation est particulièrement désolante si les derniers processus ne demandent que très peu de temps UC. Un petit quantum aurait alors été meilleur.

Autre problème : si le quantum est plus long que la rafale processeur moyenne, la préemption sera presque impossible. En effet, la plupart des processus auront effectué une opération de blocage avant la fin du quantum, ce qui provoquera un changement de processus. L'élimination de la préemption améliore la performance dans la

mesure où les changements de processus ne surviennent que lorsqu'ils sont logiquement nécessaires, à savoir lorsqu'un processus se bloque et ne peut poursuivre.

Voici la conclusion : si le quantum est trop court, il y a trop de changements de processus, et l'efficacité de l'UC chute ; mais s'il est trop long, les temps de réponse aux requêtes interactives simples deviennent trop importants. On fixe un compromis raisonnable aux alentours de 20-50 ms.

Ordonnancement par priorités

L'ordonnancement de type tourniquet part implicitement du principe que tous les processus ont une importance égale. Or, le plus souvent, les personnes amenées à gérer et à exploiter des ordinateurs multi-utilisateurs se font une autre idée de la question. Au sein d'une université, l'ordre de préséance peut être le suivant : le recteur, les professeurs, les secrétaires, les concierges, puis enfin les étudiants. La nécessité de prendre en compte des facteurs externes conduit à faire de l'**ordonnancement par priorités**. Le principe est simple : chaque processus détient un niveau de priorité, et le processus exécutable doté de la priorité la plus élevée s'exécute en premier.

Même sur un PC mono-utilisateur, plusieurs processus peuvent intervenir simultanément, certains étant plus importants que d'autres. Par exemple, un processus démon envoyant des messages électroniques à l'arrière-plan doit bénéficier d'une priorité plus faible qu'un processus affichant une vidéo à l'écran en temps réel.

Pour éviter que les processus prioritaires ne s'exécutent indéfiniment, l'ordonnanceur peut réduire le niveau de priorité du processus en cours d'exécution à chaque top d'horloge (c'est-à-dire à chaque interruption d'horloge). Si une telle action fait chuter son niveau de priorité en dessous de celui du processus prioritaire suivant, un changement de processus s'engage. Il est également possible d'assigner un quantum à chaque processus autorisé à s'exécuter. Lorsque ce quantum est écoulé, le processus prioritaire suivant peut enfin s'exécuter.

On peut assigner des priorités aux processus de façons statique ou dynamique. Sur un ordinateur de l'armée, les processus lancés par les généraux peuvent commencer avec une priorité de 100, ceux des colonels avec une priorité de 90, et ainsi de suite pour les commandants (80), les capitaines (70) et les lieutenants (60). Autre exemple : dans un centre commercial, on évalue la priorité des jobs en termes de coûts. Les jobs de priorité supérieure coûtent, disons 100 € de l'heure, contre 75 € pour les jobs de priorité moyenne et 50 € pour les jobs de priorité inférieure. Le système UNIX inclut une commande, appelée `nice`, qui permet à un utilisateur de réduire volontairement la priorité de son processus, juste par courtoisie envers les autres utilisateurs. Personne ne l'utilise jamais !

Les priorités peuvent également être assignées dynamiquement par le système pour atteindre certains objectifs internes. Par exemple, certains processus sont fortement liés aux E/S et passent le plus clair de leur temps à attendre que celles-ci soient terminées. Lorsqu'un tel processus a besoin de l'UC, il est judicieux de le lui confier immédiatement, de façon qu'il puisse lancer sa prochaine requête d'E/S, qui pourra alors s'exécuter en parallèle avec un autre processus en cours de traitement. Faire attendre

longtemps un processus d'E/S signifie simplement que celui-ci occupe la mémoire pour un temps inutilement long. Il existe un algorithme simple permettant de bien servir les processus d'E/S. Il consiste à définir leur priorité à *1/f*, où *f* est la fraction du dernier quantum utilisé par un processus. Un processus qui n'a utilisé que 1 ms sur son quantum de 50 ms obtient une priorité de 50, tandis qu'un processus qui s'est exécuté pendant 25 ms avant de bloquer obtiendra une priorité de 2. Enfin, un processus ayant utilisé tout son quantum obtient une priorité de 1.

Il est souvent pratique de regrouper les processus dans des catégories de priorités et d'ordonnancer par catégories, tout en organisant les catégories les unes par rapport aux autres selon l'algorithme de type tourniquet. La figure 2.42 montre un système dans lequel on trouve quatre catégories de priorités. L'algorithme d'ordonnancement est le suivant : tant qu'il reste des processus exécutables dans la catégorie de priorité 4, on en exécute un pendant un quantum, selon l'algorithme de type tourniquet, et on ne s'occupe pas des catégories inférieures. Si la catégorie 4 est vide, on exécute les processus de catégorie 3 selon l'algorithme de type tourniquet. Si les catégories 4 et 3 sont vides, on exécute la catégorie 2 en tourniquet, et ainsi de suite. Si on ne se donne pas la peine de revoir les priorités périodiquement, les catégories inférieures risquent de subir une privation de ressources (famine).

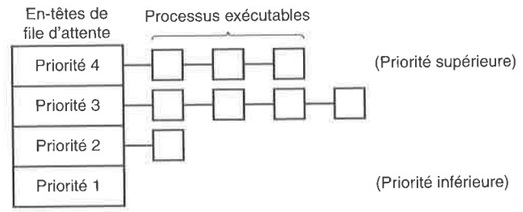

Figure 2.42 • Un algorithme d'ordonnancement avec quatre catégories de priorités.

Files d'attente multiples

L'un des premiers ordonnanceurs utilisant les priorités a été installé sur CTSS (*Compatible Time Sharing System*), un système d'exploitation de l'IBM 7094 développé au MIT. Le CTSS souffrait de délais de changement de processus très longs car la 7094 ne savait conserver en mémoire qu'un seul processus à la fois. Chaque changement impliquait l'échange du processus en cours avec le disque et la lecture d'un nouveau processus également stocké sur le disque. Les concepteurs du CTSS ont rapidement réalisé que pour réduire le nombre des échanges, il était plus efficace de donner aux processus de traitement un quantum important une fois de temps en temps, plutôt que de leur attribuer fréquemment de petits quanta. En revanche, et comme nous l'avons déjà vu, le fait de donner d'importants quanta à tous les processus ralentit les temps de réponse. La solution a donc été de créer des catégories de priorités. Les processus de priorité supérieure étaient exécutés pendant 1 quantum. Les processus de la

catégorie inférieure suivante étaient exécutés pendant 2 quanta. Les suivants l'étaient pendant 4 quanta, et ainsi de suite. Chaque fois qu'un processus avait utilisé tous les quanta qui lui avaient été attribués, il descendait d'une catégorie.

À titre d'exemple, prenons un processus qui a besoin d'un temps de traitement continu pendant 100 quanta. On commence par lui attribuer 1 quantum, puis on le stocke sur le disque. La fois suivante, on lui attribue 2 quanta avant l'échange. Lors des exécutions suivantes, notre processus récupère ainsi 4, 8, 16, 32, puis 64 quanta, même s'il n'a besoin que de 37 quanta sur les 64 derniers pour terminer son travail. Seuls 7 échanges auront été nécessaires (y compris le chargement initial) au lieu de 100 avec un algorithme de type tourniquet pur. En outre, à mesure que le processus descend de plus en plus bas dans les catégories de priorités, il s'exécute de moins en moins fréquemment, ce qui économise du temps processeur en faveur des processus interactifs courts.

La politique suivante a été adoptée pour empêcher qu'un processus ayant besoin de s'exécuter pendant longtemps au début, mais devenant interactif ultérieurement, soit pénalisé pour la suite de son exécution. Chaque fois qu'un utilisateur entrait un retour chariot sur un terminal, le processus appartenant à ce terminal remontait dans la catégorie de priorité supérieure, selon le principe qu'il devenait interactif. Un beau jour, un utilisateur chargé d'un processus à traitement lourd découvrit qu'il suffisait qu'il reste assis devant son terminal et qu'il entre des retours chariot toutes les quelques secondes (aléatoirement) pour faire des merveilles sur son temps de réponse. Il en parla à ses collègues…

Bien d'autres algorithmes ont été employés pour assigner des catégories de priorités à des processus. Par exemple, le célèbre système XDS 940, construit à Berkeley, incluait quatre catégories de priorités, appelées terminal, E/S, quantum court et quantum long. Quand un processus attendant une entrée du terminal était activé, il entrait dans la classe de priorité supérieure (terminal). Quand un processus attendant un bloc de disque était enfin prêt, il entrait dans la deuxième catégorie. Si un processus était encore en cours d'exécution quand son quantum expirait, il était tout d'abord placé dans la troisième catégorie. Cependant, les processus qui utilisaient trop souvent leurs quanta respectifs à la suite (pour des interventions du terminal ou des E/S), sans bloquer, étaient déplacés dans la dernière catégorie. Beaucoup d'autres systèmes utilisent des méthodes analogues pour favoriser les utilisateurs et les processus interactifs par rapport aux activités d'arrière-plan.

Exécuter le processus suivant le plus court

Étant donné que le fait d'exécuter le job le plus court en premier produit toujours le temps de réponse moyen le plus court sur les systèmes de traitement par lots, il serait intéressant d'appliquer la méthode aux systèmes interactifs. C'est intéressant dans une certaine mesure. Généralement, les processus interactifs suivent un schéma selon lequel ils attendent une commande, l'exécutent, en attendent une autre, l'exécutent, et ainsi de suite. Si l'on considère l'exécution de chaque commande comme un « job » distinct, on peut en effet réduire le temps de réponse global en exécutant d'abord le

soit respectée. Ainsi, si deux utilisateurs ont droit à 50 % du temps processeur, ils les obtiendront quel que soit le nombre de processus qu'ils ont lancés.

Prenons l'exemple d'un système avec deux utilisateurs, chacun ayant droit à 50 % du temps processeur. L'utilisateur 1 a quatre processus, *A*, *B*, *C* et *D*. L'utilisateur 2 n'a qu'un seul processus, appelé *E*. Avec l'algorithme de type tourniquet, la séquence d'ordonnancement suivante respecte les contraintes initiales :

A E B E C E D E A E B E C E D E...

En revanche, si l'utilisateur 1 a droit à deux fois plus de temps processeur que l'utilisateur 2, on obtiendra :

A B E C D E A B E C D E...

Naturellement, il existe de nombreuses autres possibilités, que chacun exploitera selon sa vision personnelle de l'équité.

2.4.4 L'ordonnancement des systèmes temps réel

Dans un système **temps réel**, le temps joue un rôle essentiel. Généralement, un ou plusieurs périphériques physiques extérieurs à l'ordinateur génèrent des stimuli, et l'ordinateur en question doit réagir de la manière appropriée et dans le délai qui lui est imparti. Par exemple, le calculateur d'un lecteur de disques compacts récupère les bits à mesure qu'ils arrivent et doit les convertir en musique dans un intervalle de temps très court. Si le temps de calcul est trop long, le son aura un rendu bizarre. Les systèmes de monitoring, dans les services de soins intensifs des hôpitaux, les systèmes de pilotage automatique d'un avion ou les automates des chaînes de fabrication sont autant de systèmes temps réel. Dans tous ces cas de figures, le fait d'obtenir la réponse trop tard est aussi préjudiciable que de ne pas l'avoir du tout.

On classe souvent les systèmes temps réel en deux catégories :

- Les systèmes **temps réel à tolérance zéro** (*hard real time*) qui doivent impérativement respecter leurs délais.
- Les systèmes **temps réel avec tolérance** (*soft real time*) qui ne doivent pas faillir à leurs échéances, mais pour lesquels quelques échecs restent tolérables.

Dans les deux cas, on obtient un comportement en temps réel en divisant le programme en un certain nombre de processus, dont le comportement est prévisible et connu par avance. Ces processus sont généralement courts et peuvent arriver à terme en bien moins d'une seconde. Lorsqu'un événement externe est détecté, il incombe à l'ordonnancement de planifier le processus de façon qu'il respecte son délai d'exécution.

Les systèmes temps réel ont à répondre à deux catégories d'événements : les événements **périodiques** (se produisant à intervalles réguliers) et ceux **apériodiques** (survenant de façon imprévisible). Un système peut avoir à répondre à plusieurs flots d'événements périodiques. Selon le temps de traitement découlant d'un événement, il se peut que le système ne puisse pas prendre en charge tous les événements. Supposons que nous ayons *m* événements périodiques, que l'événement *i* se produise selon

une périodicité P*i* et qu'il ait besoin de C*i* secondes de temps processeur pour gérer chaque événement. La charge ne peut être traitée que si :

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

On dit d'un système temps réel capable de remplir ce critère qu'il peut être ordonnancé. Prenons l'exemple d'un système temps réel connaissant trois événements périodiques, avec des périodes de 100, 200 et 500 ms respectivement. Si ces événements ont besoin respectivement de 50, 30 et 100 ms chacun, le système peut être ordonnancé parce que $0,5 + 0,15 + 0,2 < 1$. Si l'on ajoute un événement avec une période de 1 seconde, le système peut toujours être ordonnancé, tant que cet événement n'a pas besoin de plus de 150 ms de temps processeur par événement. Dans ce calcul, il est implicite que la surcharge engendrée par le changement de contexte est si faible qu'elle peut être ignorée.

Les algorithmes d'ordonnancement pour le temps réel peuvent être statiques ou dynamiques. Les premiers prennent leurs décisions d'ordonnancement avant que le système n'ait démarré. Les seconds prennent leurs décisions d'ordonnancement en cours d'exécution. L'ordonnancement statique ne fonctionne que lorsque des informations parfaitement justes sont disponibles par avance quant au travail à effectuer et aux délais à respecter. Les algorithmes d'ordonnancement dynamiques ne subissent pas ces contraintes. Nous étudierons ces différents algorithmes lorsque nous aborderons les systèmes multimédias temps réel, au chapitre 7.

2.4.5 La politique et le mécanisme d'ordonnancement

Jusqu'à maintenant, nous avons supposé que tous les processus du système appartenaient à des utilisateurs différents et que tous étaient en concurrence pour obtenir du temps processeur. Si cela est souvent vrai, il arrive également qu'un processus ait plusieurs enfants qui s'exécutent sous son contrôle. Par exemple, un processus de système de gestion de base de données peut avoir de nombreux enfants. Chacun va travailler sur une requête différente, ou remplir une fonction spécifique (analyse des requêtes, accès disque, etc.). Il est tout à fait possible que le processus principal ait une excellente idée de l'importance de chacun de ses enfants (ou de sa sensibilité à la rapidité d'exécution). Malheureusement, aucun des ordonnanceurs évoqués jusqu'ici n'accepte d'informations en provenance des processus utilisateur pour en tenir compte dans ses décisions. Il en résulte que les ordonnanceurs font rarement le bon choix.

La solution à cette problématique est de distinguer le **mécanisme d'ordonnancement** de la **politique d'ordonnancement**, principe établi depuis longtemps.

Cela signifie que l'algorithme d'ordonnancement peut accepter des paramètres sous une forme ou une autre, mais que ces paramètres peuvent être définis à partir des processus utilisateur. Reprenons l'exemple de la base de données. Supposons que le noyau utilise un algorithme d'ordonnancement par priorités, mais également un

appel système grâce auquel un processus peut définir (et modifier) les priorités de ses enfants. De cette façon, le parent peut contrôler dans le détail comment ses enfants sont planifiés, même si ce n'est pas lui qui est en charge de l'ordonnancement. Le mécanisme intervient au niveau du noyau, mais la politique est déterminée par le processus utilisateur.

2.4.6 L'ordonnancement des threads

Lorsque plusieurs processus ont des threads multiples, nous nous trouvons en présence de deux niveaux de parallélisme. L'ordonnancement de tels systèmes peut être très différent selon que les threads sont pris en charge au niveau utilisateur ou au niveau noyau (ou les deux).

Prenons d'abord le cas des threads de niveau utilisateur. Étant donné que le noyau n'est pas conscient de leur existence, il fonctionne comme à son habitude, sélectionnant, disons, un processus *A* et lui donnant le contrôle pour la durée de son quantum. L'ordonnanceur de threads du processus *A* décide d'exécuter, par exemple, le thread *A1*. Puisqu'il n'y a pas d'interruptions d'horloge pour les threads multiprogrammes, ce thread peut continuer de s'exécuter aussi longtemps qu'il le souhaite. S'il consomme la totalité du quantum du processus, le noyau sélectionne un autre processus à exécuter.

Lorsque le processus *A* s'exécute à nouveau, le thread *A1* reprend son exécution. Il continue de consommer tout le temps imparti à *A*, jusqu'à la fin. Cependant, ce comportement antisocial n'affecte pas les autres processus. Ceux-ci recevront une part que l'ordonnancement aura considérée comme équitable, quoi qu'il se produise au sein du processus *A*.

Imaginons maintenant que les threads de *A* aient relativement peu de travail à effectuer par flot processeur, à savoir 5 ms pour un quantum de 50 ms. Par conséquent, chacun s'exécute pendant un instant, puis restitue le processeur à l'ordonnanceur du thread. Cela peut produire la séquence *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, avant que le noyau ne bascule vers le processus *B*. La figure 2.43(a) illustre ce cas.

L'algorithme d'ordonnancement employé par le système d'exécution peut être n'importe lequel de ceux déjà décrits. Dans la pratique, les ordonnancements de type tourniquet et par priorités sont le plus couramment utilisés. La seule contrainte est l'absence d'horloge pour interrompre un thread qui se serait exécuté trop longtemps.

Voyons maintenant comment cela se passe avec les threads noyau. Le noyau sélectionne un thread à exécuter. Il n'a pas à tenir compte du processus auquel le thread appartient, mais il peut s'en informer s'il le souhaite. Le thread se voit attribuer un quantum, et il est arrêté de force au-delà du délai imparti. Avec un quantum de 50 ms et des threads qui se bloquent au-delà de 5 ms, l'ordre des threads sur une période de 30 ms peut être : *A1*, *B1*, *A2*, *B2*, *A3*, *B3*. Ce qui ne serait pas possible avec ces paramètres et des threads utilisateur. La figure 2.43(b) décrit partiellement cette situation.

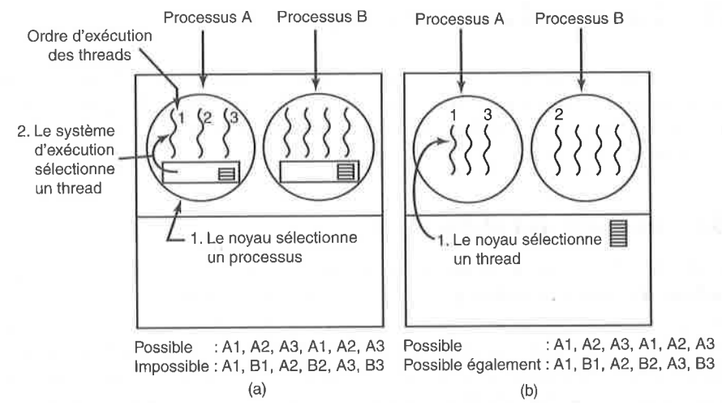

Figure 2.43 • (a) Ordonnancement possible de threads utilisateur avec un quantum de processus de 50 ms et des threads qui s'exécutent à raison de 5 ms par flot processeur. (b) Ordonnancement possible de threads noyau avec les mêmes caractéristiques que (a).

L'une des différences essentielles entre les threads utilisateur et noyau repose sur les performances. Les changements de thread au niveau utilisateur nécessitent quelques instructions machine. Avec les threads au niveau noyau, il faut procéder à un changement de contexte complet, modifier le mappage mémoire et invalider le cache, ce qui est nettement plus lent. En revanche, avec les threads au niveau noyau, le fait qu'un thread bloque sur une E/S ne suspend pas l'ensemble du processus, comme c'est le cas avec les threads au niveau utilisateur. Étant donné que le noyau sait que le fait de passer d'un thread du processus *A* à un thread du processus *B* lui prend plus de temps que d'exécuter un autre thread du processus *A*, il peut tenir compte de ces informations pour prendre des décisions. Par exemple, en présence de deux threads d'importance égale, celui qui appartient à un processus dont un thread vient juste de bloquer aura la préférence.

Autre facteur important : les threads utilisateur peuvent faire usage d'un ordonnanceur de threads spécifique à l'application. Prenons l'exemple du serveur Web de la figure 2.8. Supposons qu'un thread worker vienne juste de se bloquer alors que le thread dispatcher et deux workers sont prêts. Quel va être le suivant à s'exécuter ? Le système d'exécution, sachant ce que font les threads, peut aisément sélectionner le dispatcher, de façon que ce dernier puisse démarrer un autre worker. Cette stratégie a une incidence favorable sur le parallélisme dans un environnement où les workers bloquent fréquemment sur des E/S disque. Avec les threads noyau, le noyau ne pourrait pas savoir quelles sont les activités des différents threads (ceux-ci pourraient malgré tout être assortis de niveaux de priorité). Il reste que généralement, les ordonnanceurs de threads spécifiques à l'application permettent de perfectionner le fonctionnement d'une application au-delà des possibilités du noyau.