Prenons le cas d'un ordinateur qui possède 512 Mo de mémoire ; son système d'exploitation utilise 128 Mo et chaque programme utilisateur sollicite jusqu'à 128 Mo. Cette configuration permet à trois programmes d'être simultanément en mémoire. Avec une attente moyenne d'entrée/sortie de 80 % du temps, nous avons un taux d'utilisation UC (en ignorant le temps système du système d'exploitation) de $1 - 0{,}8^3$, soit 49 %. En ajoutant 512 Mo de mémoire, on permet au système de passer d'une multiprogrammation de degré 3 à une multiprogrammation de degré 7 : le taux d'utilisation passe ainsi à 79 %. En d'autres termes, les 512 Mo supplémentaires augmentent la capacité de traitement de 30 %.

Si l'on ajoute encore 512 Mo, le taux d'utilisation passera de 79 à 91 %, et la capacité de traitement augmentera donc de 12 %. Suivant ce schéma, on peut considérer que si le premier ajout est un bon investissement pour un ordinateur, le second ne l'est pas vraiment.

2.2 Les threads

Dans les systèmes d'exploitation traditionnels, chaque processus possède un espace d'adressage et un thread de contrôle unique. C'est en fait ainsi que l'on définit le mieux un processus. Cependant, il arrive qu'il soit souhaitable de disposer de plusieurs threads de contrôle dans le même espace d'adressage, ceux-ci s'exécutant quasiment en parallèle, comme s'il s'agissait de processus distincts (à l'exception du fait que l'espace d'adressage est commun). Dans les sections suivantes, nous aborderons ces cas de figure et leurs implications.

2.2.1 L'utilisation des threads

Pourquoi vouloir disposer d'une sorte de processus à l'intérieur même d'un processus ? Il existe plusieurs raisons à l'existence de ces miniprocessus qu'on appelle les **threads.** Examinons-en quelques-unes. La principale d'entre elles est que, dans de nombreuses applications, plusieurs activités ont lieu simultanément. Certaines de ces activités peuvent se bloquer de temps en temps. En décomposant une application en plusieurs threads séquentiels qui vont s'exécuter en quasi-parallélisme, le modèle de programmation devient plus simple.

Nous avons déjà vu cet argument. C'est précisément celui qui explique l'existence des processus. Au lieu de penser interruptions, temporisateurs et commutateurs de contexte, nous pouvons raisonner en termes de processus parallèles. Mais maintenant, avec les threads, on ajoute un élément nouveau : la capacité pour les entités parallèles à partager un espace d'adressage et toutes ses données. Cette faculté est essentielle à certaines applications, ce qui explique pourquoi cela ne fonctionnerait pas avec plusieurs processus qui, eux, ont chacun leur propre espace d'adressage.

Deuxième argument en faveur des threads : étant donné qu'aucune ressource ne leur est associée, ils sont plus faciles à créer et à détruire que les processus. Sur de nombreux systèmes, la création d'un thread est une opération 100 fois plus rapide que

celle d'un processus. Lorsque le nombre de threads change dynamiquement et rapidement, c'est une propriété intéressante.

Troisième argument : la performance. Les threads ne produisent pas de gains de performances lorsque tous sont liés au processeur, mais lorsque le traitement et les E/S sont substantiels ; le fait d'avoir créé des threads permet à ces activités de se superposer, ce qui accélère le fonctionnement de l'application.

Enfin, les threads sont utiles sur les systèmes multiprocesseurs, sur lesquels un véritable parallélisme est possible. Nous y reviendrons au chapitre 8.

Quelques exemples concrets permettront de mieux percevoir l'utilité des threads. Prenons tout d'abord l'exemple d'un traitement de texte qui affiche le document créé à l'écran, avec une mise en forme identique à celle qui sera reproduite sur la page imprimée. Entre autres, les sauts de ligne et de page sont à leur position finale, ce qui permet à l'utilisateur de les vérifier et d'apporter des modifications si nécessaire (comme éliminer les veuves et les orphelins, c'est-à-dire les lignes solitaires en haut et en bas des pages, que l'on considère comme étant inesthétiques).

Dans notre exemple, l'utilisateur écrit un livre. De son point de vue, il est plus facile de conserver la totalité du livre dans un fichier unique, dans lequel il peut rechercher les titres et sous-titres, faire des remplacements systématiques, etc. L'autre solution consiste à conserver chaque chapitre dans un fichier distinct. En revanche, créer des fichiers différents pour chaque section ou sous-section d'un livre est une très mauvaise idée qui obligerait l'utilisateur à travailler sur des centaines de fichiers pour effectuer le moindre changement se répercutant sur l'ensemble du travail. Par exemple, si la norme xxx vient d'être publiée officiellement, juste avant que le livre ne soit mis sous presse, toutes les occurrences des mots « projet de norme xxx » devront être remplacées par les mots « norme xxx » à la dernière minute. Si tout le livre se trouve dans un seul fichier, on peut faire tous les remplacements à l'aide d'une seule commande. En revanche, si le livre est réparti sur plus de 300 fichiers, il faudra lancer autant de cycles de remplacement.

Imaginons ce qui se passerait si l'utilisateur supprimait subitement une phrase de la page 1, dans un document de 800 pages. Après avoir vérifié la page ainsi modifiée, notre utilisateur veut effectuer une autre modification, à la page 600. Il saisit une commande demandant au traitement de texte d'aller à cette page (éventuellement en recherchant une suite de mots bien précise). Le logiciel est alors obligé de remettre en forme tout le livre jusqu'à la page 600, à la volée, car il ne peut définir la première ligne de la page 600 tant qu'il n'a pas traité toutes les pages précédentes. Le temps d'affichage de la page 600 risque d'être long, et l'utilisateur sera contrarié de ce délai d'attente.

C'est là que les threads peuvent intervenir. Supposons que notre traitement de texte ait été écrit sous forme de programme à deux threads. Un thread prend en charge l'interaction avec l'utilisateur et l'autre la remise en forme à l'arrière-plan. Dès que la phrase incriminée a été supprimée de la page 1, le thread interactif indique au thread de mise en forme de commencer son travail. Entre-temps, le thread interactif reste à l'écoute du clavier et de la souris et répond à des commandes simples, comme un

défilement de la page 1. Pendant ce temps, l'autre thread s'active en tâche de fond. Avec un peu de chance, la remise en forme sera terminée avant que l'utilisateur ne demande l'affichage de la page 600, et l'opération sera instantanée. Mais tant que nous y sommes, pourquoi ne pas introduire un troisième thread ? De nombreux traitements de texte incluent une fonctionnalité d'enregistrement automatique des fichiers sur le disque, à raison d'un enregistrement toutes les quelques minutes. Cela peut éviter à l'utilisateur de perdre son travail de la journée en cas d'arrêt inopiné du programme, du système ou d'un défaut d'alimentation. Le troisième thread peut prendre en charge les enregistrements sur disque sans interférer avec les deux autres. La figure 2.7 présente un exemple de logiciel avec trois threads.

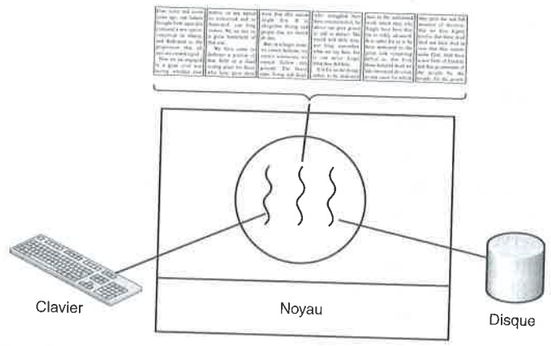

Figure 2.7 • Un traitement de texte à trois threads.

Avec un programme monothread, les commandes du clavier et de la souris sont ignorées chaque fois que l'enregistrement sur disque se déclenche, et ce jusqu'à ce qu'il soit terminé. L'utilisateur perçoit cet état de fait comme une lenteur irritante. Réciproquement, des événements clavier ou souris pourraient interrompre l'enregistrement sur disque, ce qui donnerait l'impression d'une bonne performance, mais induirait un modèle de programmation piloté par des interruptions d'une complexité difficile à gérer. Avec trois threads, le modèle de programmation est nettement plus simple. Le premier thread se contente d'interagir avec l'utilisateur. Le deuxième remet en forme le document lorsqu'on lui en donne l'instruction. Le troisième écrit périodiquement le contenu de la RAM sur le disque.

Il est clair que le fait de faire intervenir trois processus dans ce contexte ne fonctionnerait pas, car ces trois threads ont besoin d'accéder au document. Avec trois threads au lieu de trois processus, le partage de la mémoire permet à chacun d'accéder au document en cours de modification.

Des situations analogues se produisent avec bien d'autres programmes interactifs. Par exemple, un tableur est un programme qui permet à l'utilisateur d'exploiter une matrice, appelée feuille de calcul, contenant des données saisies. Certains éléments de

la feuille de calcul peuvent contenir des données hébergeant éventuellement des formules de calcul complexes. Lorsque l'utilisateur modifie un élément, plusieurs autres peuvent devoir être recalculés. Si l'on dispose d'un thread qui travaille en arrière-plan pour effectuer ces calculs, le thread interactif peut se consacrer à la gestion des autres modifications effectuées par l'utilisateur pendant le temps nécessaire aux calculs. De même, un troisième thread peut prendre en charge les enregistrements périodiques sur disque.

Un autre exemple illustre bien l'utilité des threads : un serveur de sites Web. Les requêtes de pages arrivent et les pages sollicitées sont retournées au client. Sur la plupart des sites Web, certaines pages sont plus souvent visitées que d'autres. Par exemple, la page d'accueil du site de Sony sera consultée beaucoup plus fréquemment que celle d'une fiche technique pour un caméscope spécifique. Les serveurs Web peuvent améliorer leurs performances en maintenant une collection de pages souvent demandées dans leur mémoire principale, ce qui élimine la nécessité d'un accès au disque pour les consulter. Une telle collection s'appelle un **cache** ; on utilise les caches dans de nombreux autres contextes.

La figure 2.8 illustre une manière d'organiser ce serveur Web. On y trouve un thread, appelé **dispatcher**, qui lit les requêtes entrantes à traiter. Après avoir examiné la requête, il choisit un thread **worker** inactif (c'est-à-dire bloqué) et lui transmet la requête, éventuellement en écrivant un pointeur vers le message dans un mot spécial associé à chaque thread. Le dispatcher active alors le worker, le faisant passer de l'état bloqué à l'état prêt.

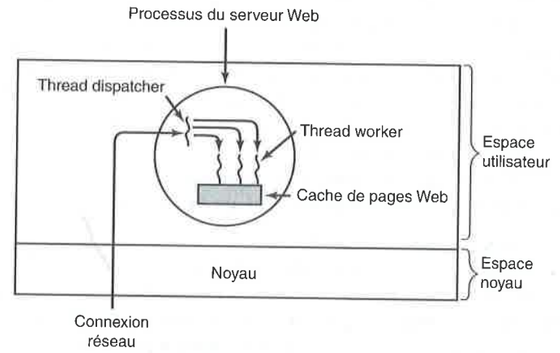

Figure 2.8 • Un serveur Web multithread.

Lorsque le worker s'active, il vérifie que la requête peut être satisfaite à partir du cache de pages Web, auquel tous les threads ont accès. Si ce n'est pas le cas, il entame une opération `read` pour récupérer la page sur le disque et se bloque jusqu'à ce que l'opération soit accomplie. Lorsque le thread se bloque pendant son interaction avec le

disque, un autre thread est choisi, éventuellement par le dispatcher, afin d'assurer une autre activité. Il se peut aussi qu'un autre worker entre en action.

Ce modèle permet au serveur de recevoir des entrées sous forme de collection de threads séquentiels. Le programme du dispatcher se compose d'une boucle sans fin qui permet d'intercepter une requête afin de la transmettre à un worker. Le code des workers inclut une boucle sans fin destinée à accepter les requêtes émanant du dispatcher et à vérifier le cache de pages Web, afin de récupérer la page sollicitée si elle s'y trouve. Si ce n'est pas le cas, le worker récupère la page sur le disque, la retourne au client et se bloque en attente d'une nouvelle requête.

La figure 2.9 vous donne un aperçu global du code. Ici, comme dans le reste du livre, `TRUE` est supposée représenter la constante `1`. Par ailleurs, `buf` et `page` sont les structures qui détiennent respectivement la requête de travail et la page Web.

Figure 2.9 • Aperçu global du code correspondant à la figure 2.8. (a) Thread dispatcher. (b) Thread worker.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Voyons comment le serveur Web peut être développé sans utiliser de threads multiples. D'abord, il peut s'exécuter dans un seul thread. La boucle du serveur Web récupère une requête, l'examine et la mène à son terme avant de passer à la suivante. Pendant qu'il attend la réponse du disque, le serveur est inoccupé ; il ne traite aucune autre requête entrante. S'il s'exécute sur un ordinateur dédié, ce qui est le cas le plus courant, le processeur est inactif pendant que le serveur attend la réponse du disque. Il en résulte clairement un nombre inférieur de requêtes traitées par seconde. Cela montre bien que les threads améliorent considérablement les performances, même s'ils sont programmés, comme habituellement, en mode séquentiel.

Jusqu'ici, nous avons vu deux conceptions possibles : un serveur Web multithread et un serveur Web monothread. Supposons qu'il ne soit pas possible de programmer des threads, mais que les concepteurs du système estiment que la perte de performances due à cette limitation soit inacceptable. Si une version non bloquante de l'appel système `read` est disponible, une autre approche devient possible. Lorsqu'une requête arrive, le seul et unique thread en place l'examine. Si celle-ci peut être satisfaite par le cache, tout va bien ; dans le cas contraire, une opération de disque non bloquante est déclenchée. Le serveur enregistre l'état de la requête en cours dans une table, puis va récupérer le prochain événement. Il peut s'agir d'une requête pour une nouvelle tâche ou d'une réponse du disque concernant l'opération précédente. S'il s'agit d'une nouvelle tâche, celle-ci démarre. S'il s'agit d'une réponse du disque, les informations concernées sont récupérées dans la table, et la réponse est traitée. Avec des E/S disque non bloquantes, une réponse devra probablement prendre la forme d'un signal ou d'une interruption.

Dans cette conception, le modèle de « processus séquentiel » des deux premiers cas est perdu. L'état du traitement doit être explicitement enregistré et restauré dans la table chaque fois que le serveur bascule d'une requête à une autre. En effet, il s'agit ici de simuler les threads et leurs piles « en dur ». Une telle conception – dans laquelle chaque intervention de traitement possède un état enregistré et où il existe un jeu donné d'événements susceptibles de se produire pour changer les états – est une **machine à nombre d'états fini**. Ce concept est largement exploité dans l'industrie de l'informatique.

Tout cela vous a permis de mieux cerner ce que les threads peuvent offrir. Ils permettent d'appliquer la notion de processus séquentiels qui effectuent des appels système bloquants (notamment pour les E/S disque) tout en autorisant le parallélisme. Les appels système bloquants facilitent la programmation et améliorent les performances. Le serveur monothread conserve la simplicité des appels système bloquants, mais perd en performances. La troisième approche autorise des performances élevées grâce au parallélisme, mais elle exploite des appels et des interruptions non bloquants ; elle est donc plus complexe à programmer. Ces modèles sont récapitulés à la figure 2.10.

Figure 2.10 • Les trois manières de construire un serveur.

Modèle	Caractéristiques
Threads	Parallélisme, appels système bloquants
Processus monothread	Pas de parallélisme, appels système bloquants
Machine à nombre d'états fini	Parallélisme, appels système non bloquants, interruptions

Les applications qui doivent prendre en charge le traitement de gros volumes de données sont également un exemple où l'intervention des threads est intéressante. L'approche normale consiste à lire dans un bloc de données, à traiter des données et à les réécrire. Mais si l'on ne peut utiliser que des appels système bloquants, le processus s'interrompt pendant que les données arrivent et repartent. Il est assurément dommage que le processeur reste inactif alors qu'il y a beaucoup de traitements à effectuer. Une telle situation est à éviter autant que possible. Les threads sont une solution. Le processus peut être structuré avec un thread d'entrée, un thread de traitement et un thread de sortie. Le thread d'entrée lit les données et les place dans un tampon mémoire d'entrée. Le thread de traitement récupère ces données dans le tampon d'entrée, les traite et place les résultats dans un tampon de sortie. Ce dernier réécrit les données traitées sur le disque. De cette façon, l'entrée, la sortie et le traitement peuvent tous avoir lieu simultanément. Naturellement, ce modèle ne fonctionne que si un appel système ne bloque que le thread appelant, et non l'ensemble du processus.

2.2.2 Le modèle de thread classique

Maintenant que nous avons vu à quoi servent les threads et comment on les utilise, essayons d'être un peu plus précis. Le modèle de processus tel que nous l'avons vu jusqu'à présent est fondé sur deux concepts indépendants : le regroupement des ressources et l'exécution. Or, il est parfois utile de les séparer et c'est là que les threads interviennent. Nous verrons d'abord le modèle classique des threads et nous examinerons ensuite le modèle de threads de Linux, qui fait le pont entre processus et threads.

On peut considérer les processus comme des moyens de regrouper des ressources liées. Un processus a un espace d'adressage contenant un programme et des données, mais aussi d'autres ressources. Parmi celles-ci, on peut trouver les fichiers ouverts, les processus enfants, les alertes en attente, les gestionnaires de signal, les informations de décompte, et bien d'autres encore. En les rassemblant sous forme de processus, on en facilite la gestion.

On peut voir les processus sous un autre angle, en considérant leur thread d'exécution (chemin d'exécution), que l'on se contente d'appeler **thread**. Le thread inclut : un compteur ordinal qui effectue le suivi des instructions à exécuter ; des registres qui détiennent ses variables de travail en cours ; et une pile qui contient l'historique de l'exécution, avec un bloc d'activation (*frame*) pour chaque procédure appelée mais n'ayant encore rien retourné. Bien qu'un thread doive s'exécuter dans un processus, ils représentent deux concepts distincts qui peuvent être traités séparément. Les processus servent à regrouper les ressources ; les threads sont les entités planifiées pour leur exécution par le processeur.

En venant s'ajouter au modèle de processus, les threads autorisent les exécutions multiples dans le même environnement de processus, avec une marge d'indépendance importante les uns par rapport aux autres. Le fait que plusieurs threads s'exécutent en parallèle au sein d'un processus est comparable au fait que plusieurs processus s'exécutent en parallèle sur un ordinateur. Dans le premier cas, les threads partagent un espace d'adressage, les fichiers ouverts et les autres ressources. Dans le second, les processus partagent la mémoire physique, les disques, les imprimantes et les autres ressources. Étant donné que les threads ont certaines propriétés des processus, ils sont parfois qualifiés de **processus légers** (*lightweight processes*). Le terme **multithreading** est également employé pour décrire la situation dans laquelle plusieurs threads sont présents dans le même processus. Comme nous l'avons vu au chapitre 1, certaines UC ont un matériel spécialisé qui gère le multithreading et permet la commutation de threads de l'ordre de la nanoseconde.

La figure 2.11(a) montre trois processus traditionnels. Chacun a son propre espace d'adressage et un thread de contrôle unique. La figure 2.11(b) représente un processus unique avec trois threads de contrôle. Bien que dans les deux cas, on se trouve en présence de trois threads, il existe une différence. Dans la première figure, chaque thread fonctionne dans un espace d'adressage différent, alors que, dans la seconde, les trois threads se partagent le même espace d'adressage.

Lorsqu'un processus multithread s'exécute sur un système monoprocesseur, les threads sont traités chacun à leur tour. À la figure 2.1, vous avez vu comment fonctionnait la multiprogrammation de processus.

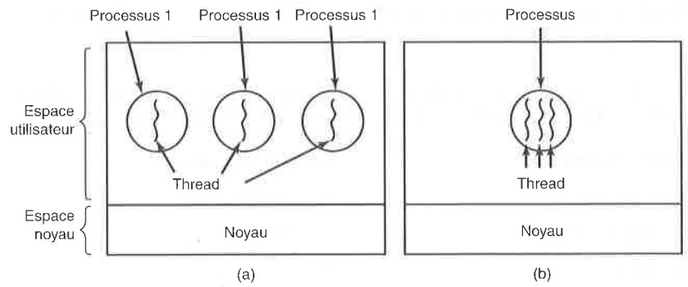

Figure 2.11 • (a) Trois processus avec un thread chacun. (b) Un processus avec trois threads.

En basculant d'un processus à l'autre, le système donne l'illusion que des processus séquentiels distincts s'exécutent en parallèle. Le multithreading fonctionne de la même manière. Le processeur bascule rapidement entre les threads, offrant l'illusion que les threads s'exécutent en parallèle. Trois threads intervenant dans un même processus semblent s'exécuter en parallèle, chacun sur un processus, et à un tiers de la vitesse réelle de l'UC.

Les différents threads d'un même processus ne sont pas tout à fait aussi indépendants que le seraient trois processus distincts. Tous les threads ont le même espace d'adressage, ce qui signifie qu'ils partagent également les mêmes variables globales. Étant donné que chaque thread peut accéder aux adresses mémoire de l'espace d'adressage du processus, il peut lire ou écrire, voire effacer totalement la pile d'un autre thread. Il n'existe pas de protection entre threads. D'une part, cela est impossible et, d'autre part, cela ne devrait pas être nécessaire. À la différence des processus, qui peuvent provenir de différents utilisateurs et qui pourraient être hostiles les uns envers les autres, un thread est toujours détenu par un utilisateur unique, qui a semble-t-il créé plusieurs threads de façon à les faire coopérer entre eux, et non à s'opposer. Outre le fait qu'ils partagent un espace d'adressage, tous les threads exploitent le même jeu de fichiers ouverts, de processus enfant, d'alertes et de signaux, etc., comme le montre la figure 2.12. Ce qui fait que l'organisation de la figure 2.11(a) devrait être utilisée lorsque les trois processus sont exempts de relations entre eux. À l'inverse, le schéma de la figure 2.11(b) est approprié lorsque les trois threads font véritablement partie de la même tâche et qu'ils coopèrent étroitement et activement les uns avec les autres.

Les éléments de la première colonne sont des propriétés de processus, et non de threads. Par exemple, si un thread ouvre un fichier, celui-ci sera visible par les autres threads du processus qui pourront effectuer des opérations de lecture et d'écriture sur le fichier en question. C'est logique dans la mesure où le processus est l'unité de gestion des ressources, ce qui n'est pas le cas du thread. Si chaque thread avait son propre espace d'adressage, ses propres fichiers ouverts, et ainsi de suite, il s'agirait d'un processus distinct. Le concept de thread a pour objet de permettre à plusieurs

threads d'exécution de partager un jeu de ressources pour travailler ensemble, afin d'accomplir une tâche donnée.

Figure 2.12 • La première colonne montre certains éléments partagés par tous les threads d'un processus. La seconde présente certains éléments privés de chaque thread.

Éléments par processus	Éléments par thread
Espace d'adressage	Compteur ordinal
Variables globales	Registres
Fichiers ouverts	Pile
Processus enfant	État
Alertes en attente	
Signaux et gestionnaires de signaux	
Informations de décompte	

À l'instar d'un processus traditionnel (à savoir un processus ne comptant qu'un seul thread), un thread peut prendre plusieurs états : en cours d'exécution, bloqué, prêt ou arrêté (terminé). Un thread en cours d'exécution utilise le processeur ; il est actif. Un thread bloqué attend qu'un événement le débloque. Par exemple, lorsqu'un thread lance un appel système pour récupérer une information depuis le clavier, il est bloqué jusqu'à ce que quelqu'un tape une entrée. Un thread peut être bloqué en attente d'un événement externe ou de l'intervention d'un autre thread pour le débloquer. Un thread prêt est planifié pour exécution et s'exécutera dès que son tour sera venu. Les transitions entre les différents états sont les mêmes qu'entre les états des processus (voir figure 2.2).

Il est important de comprendre que chaque thread possède sa propre pile, comme l'illustre la figure 2.13. Ces piles contiennent un bloc d'activation (*frame*) pour chaque procédure invoquée, qui n'a encore rien retourné. Un bloc d'activation contient les variables locales de la procédure et l'adresse de retour à employer une fois l'appel de procédure exécuté. Par exemple, si une procédure X appelle une procédure Y et que cette dernière appelle Z, pendant que Z s'exécute, les blocs d'activation de X, Y et Z se trouveront tous dans la pile. Généralement, chaque thread appelle des procédures différentes, et donc un historique d'exécution différent. C'est pour cette raison que chaque thread a besoin de sa propre pile.

Dans le cas du multithreading, les processus démarrent normalement avec un seul thread. Celui-ci est capable d'en créer de nouveaux en invoquant une procédure de bibliothèque, comme `thread_create`. Un paramètre de `thread_create` spécifie généralement le nom d'une procédure que doit exécuter le nouveau thread. Il n'est pas nécessaire (et, de toute façon, cela est impossible) de spécifier quoi que ce soit concernant l'espace d'adressage du nouveau thread, dans la mesure où ce dernier s'exécute automatiquement dans celui du thread qui l'a créé. Les threads sont parfois organisés

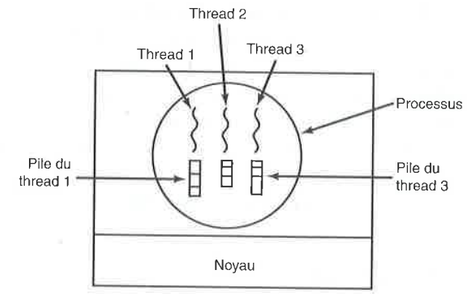

Figure 2.13 • Chaque thread a sa propre pile.

en hiérarchie, avec des relations parent-enfant, mais cette situation est rare, et dans la majorité des cas, les threads se trouvent tous sur un même pied d'égalité. Avec ou sans relation hiérarchique, le thread initial récupère généralement un identifiant de thread nommant le nouveau thread.

Lorsqu'un thread a terminé de s'exécuter, il peut s'arrêter en appelant une procédure de bibliothèque, disons `thread_exit`. Ensuite, il disparaît et ne peut plus être ordonnancé. Dans certains systèmes de threading, un thread peut attendre qu'un autre thread (spécifique) s'arrête en appelant une procédure qui pourrait s'appeler `thread_wait`. Cette procédure bloque le thread appelant jusqu'à ce qu'un thread spécifique se soit arrêté. Sous cet aspect, la création et l'arrêt des threads ressemblent fort à ceux des processus, avec approximativement les mêmes options disponibles.

On rencontre souvent un autre appel de thread, `thread_yield`, qui permet à un thread d'abandonner volontairement le processeur pour laisser un autre thread s'exécuter. Un tel appel est important, car il n'y a pas d'interruption d'horloge pour véritablement rendre obligatoire le partage du temps processeur, comme c'est le cas pour les processus. Il est donc vital que les threads sachent se montrer « polis » et qu'ils « lâchent » régulièrement du temps processeur, afin de laisser les autres threads s'exécuter. D'autres appels existent pour permettre à un thread donné d'attendre qu'un autre thread ait terminé une certaine tâche, d'annoncer qu'il a terminé son job, etc.

Si les threads sont souvent utiles, ils induisent un certain nombre de complications dans le modèle de programmation. Tout d'abord, observons les effets que peut avoir l'appel système UNIX `fork`. Si le processus parent possède plusieurs threads, le fils doit-il les posséder également ? Si ce n'est pas le cas, le processus risque de ne pas fonctionner correctement, si tous ces threads sont essentiels.

En revanche, si le processus enfant récupère autant de threads que son parent, que va-t-il se passer si le parent reste bloqué sur un appel `read` provenant par exemple du clavier ? Va-t-on trouver deux threads bloqués sur le clavier, un dans le parent et un dans le fils ? Lorsqu'une ligne sera saisie, les deux threads en récupéreront-ils une copie ? Ou seulement le parent, ou encore uniquement le fils ? Le même problème existe pour les connexions réseau ouvertes.

Un autre type de problème est lié au fait que les threads partagent de nombreuses structures de données. Que se passe-t-il si un thread ferme un fichier alors qu'un autre est encore en train de le lire ? Supposons qu'un thread, ayant remarqué que la mémoire allait manquer, en alloue davantage. Par ailleurs, un basculement de thread se produit, et le nouveau thread, ayant fait la même observation, se met lui aussi à allouer de la mémoire. La mémoire risque d'être allouée deux fois. Ces problèmes peuvent être résolus si l'on s'en donne le temps et les moyens, mais une conception soignée est indispensable au bon fonctionnement des programmes multithreads.

2.2.3 Les threads de POSIX

Pour permettre aux développeurs d'écrire des programmes contenant des threads qui soient portables, l'IEEE a défini une norme de threads (IEEE 1003.1c). Le paquetage **Pthreads** qui les définit, et qui est implémenté sur la plupart des systèmes UNIX, comprend plus de 60 appels de fonctions dont nous ne verrons que les plus importants (voir figure 2.14).

Ainsi définis, ces threads ont certaines propriétés. Chacun d'entre eux a un identificateur, un ensemble de registres (dont un compteur ordinal) et des attributs (taille de pile, paramètres d'ordonnancement...) stockés dans une structure.

Figure 2.14 • Quelques appels de fonction Pthreads.

Appel	Description
`pthread_create`	Crée un nouveau thread
`pthread_exit`	Termine le thread appelant
`pthread_join`	Attend la fin d'un thread
`pthread_yield`	Libère l'UC pour laisser un autre thread s'exécuter
`pthread_attr_init`	Crée et initialise une structure attribut de thread
`pthread_attr_destroy`	Supprime une structure attribut de thread

On crée un thread avec l'appel `pthread_create` qui renvoie une valeur identificateur de thread. Cet appel ressemble à un `fork`, l'identificateur de thread correspondant au PID (identificateur de processus), essentiellement pour identifier les threads référencés dans d'autres appels.

Lorsqu'un thread a terminé le travail qu'il devait faire, il se termine par un appel à `phtread_exit` qui libère la pile associée.

Un thread a souvent besoin d'attendre qu'un autre ait fini son travail avant de poursuivre le sien. C'est le rôle de l'appel `pthread_join` qui prend comme paramètre l'identificateur du thread à attendre.

Il peut arriver que, sans être bloqué, un thread se rende compte qu'il monopolise l'UC pendant un temps assez long et qu'il veuille par conséquent laisser un autre thread avancer un peu. Il appelle alors `pthread_yield`. C'est quelque chose de typique des threads et que les processus n'ont pas. Ces derniers, en effet, sont en compétition les uns avec les autres alors que les threads, qui ont été écrits par un même programmeur, coopèrent à l'atteinte d'un objectif commun.

Les deux appels suivants traitent des attributs. `Pthread_attr_init` crée une structure associée au thread et l'initialise avec des valeurs par défaut qui peuvent ensuite être manipulées (cas de la priorité, par exemple), et `pthread_attr_destroy` supprime la structure en libérant la mémoire qui la contenait (mais le thread en lui-même continue d'exister).

Pour mieux comprendre le fonctionnement de `Pthreads`, considérons la figure 2.15. Le programme principal boucle `NOMBRE_DE_THREADS` fois, en créant un nouveau thread à chaque itération. Si la création de thread échoue, on imprime un message d'erreur et on sort. Lorsqu'on a fini de créer les threads, le programme principal se termine.

À sa création, le thread imprime un message d'une ligne pour s'annoncer puis il se termine. L'ordre dans lequel les différents messages sont entrelacés est indéterminé et peut varier d'une exécution du programme à l'autre.

Figure 2.15 • Exemple de programme utilisant des threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NOMBRE DE_THREADS 10
void *print_hello_world(void *tid)
{
        /* Cette fonction imprime l'identificateur de thread et se termine. */
        printf("Bonjour tout le monde. Meilleures salutations du thread %d0, tid);
        pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
        /* Le programme principal crée 10 threads et se termine. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;
        for(i=0; i < NOMBRE DE_THREADS; i++) {
              printf("Main here. Creating thread %d0, i);
              status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
              if (status != 0) {
                   printf("Oops. pthread_create returned error code %d0, status);
                   exit(-1);
              }
        }
        exit(NULL);
}
```

Nous nous intéresserons à d'autres threads de ce paquetage `Pthreads` juste après avoir vu la synchronisation des processus et des threads.

2.2.4 L'implémentation de threads dans l'espace utilisateur

Il existe essentiellement deux manières d'implémenter un paquetage de threads : dans l'espace utilisateur et dans le noyau. À ce sujet, la controverse reste limitée et les implémentations hybrides sont également envisageables. Nous allons maintenant décrire ces méthodes, ainsi que leurs avantages et leurs inconvénients.

La première méthode consiste à placer le paquetage de threads dans l'espace utilisateur. Le noyau ne sait alors rien de leur existence. Pour lui, il s'agit de prendre en charge des processus ordinaires, monothreads. Le premier avantage, et aussi le plus évident, est que le paquetage de threads côté utilisateur peut être mis en œuvre dans le cadre d'un système d'exploitation ne supportant pas les threads. À une certaine époque, c'était le cas de tous les systèmes d'exploitation et certains sont encore ainsi conçus. C'est une approche avec laquelle les threads sont implémentés sous forme de bibliothèque.

Toutes ces implémentations possèdent la même structure générale, qui est illustrée à la figure 2.16(a). Le thread s'exécute « au-dessus » d'un système d'exécution, qui est une collection de procédures prenant en charge les threads. Jusqu'à présent, nous avons vu quatre d'entre eux : `thread_create`, `thread_exit`, `thread_wait` et `thread_yield`, mais on en trouve généralement d'autres.

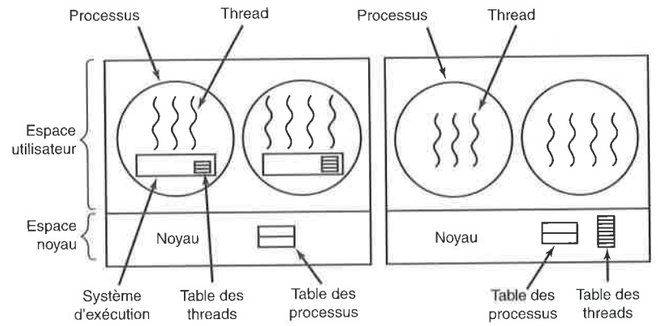

Figure 2.16 • (a) Un paquetage de threads au niveau utilisateur. (b) Un paquetage de threads pris en charge par le noyau.

Lorsque les threads sont gérés dans l'espace utilisateur, chaque processus a besoin de sa propre **table de threads** privée pour le suivi des threads du processus. Cette table est analogue à la table des processus du noyau, mais elle ne concerne que les propriétés des threads, comme le compteur ordinal, le pointeur de pile, les registres, l'état, etc. La table des threads est gérée par le système d'exécution (*run time system*).

Lorsqu'un thread passe dans l'un des états prêt ou bloqué, les informations nécessaires à son redémarrage sont stockées dans la table des threads, exactement de la même manière que le noyau stocke les informations relatives aux processus dans la table des processus.

Lorsqu'un thread accomplit une action susceptible de le bloquer localement – par exemple, il attend qu'un autre thread de son processus termine une tâche quelconque –, il invoque une procédure du système d'exécution. Cette dernière détermine si le thread doit être placé en état bloqué. Si c'est le cas, elle stocke les registres du thread dans la table des threads, sonde la table à la recherche d'un thread prêt à s'exécuter et recharge les registres machine avec les valeurs enregistrées du nouveau thread. Dès que le pointeur de pile et le compteur ordinal ont basculé, le nouveau thread recommence à s'exécuter automatiquement. Si la machine possède une instruction pour stocker tous les registres et une autre pour tous les charger, le basculement complet du thread peut être accompli en quelques instructions seulement. Ce type de basculement de thread est plus rapide que la rétention dans le noyau, ce qui apporte un argument supplémentaire en faveur des paquetages de threads dans l'espace utilisateur.

Il existe cependant une différence clé par rapport aux processus. Lorsqu'un thread a terminé de s'exécuter pour le moment, par exemple il appelle `thread_yield`, le code de `thread_yield` peut enregistrer les informations du thread dans la table des threads. En outre, il peut invoquer l'ordonnanceur de threads pour y choisir un autre thread à exécuter. Les procédures qui enregistrent ensuite l'état du thread et l'ordonnanceur sont des procédures locales, ce qui en rend l'invocation nettement plus efficace qu'un appel noyau. Par ailleurs, aucune interruption n'est nécessaire, pas plus que le changement de contexte ; le cache mémoire n'a pas besoin d'être vidé, et ainsi de suite. Tout cela fait de l'ordonnancement des threads une opération très rapide.

Les threads de l'espace utilisateur ont d'autres avantages. Ils permettent à chaque processus de disposer de son propre algorithme d'ordonnancement. Pour certaines applications, comme celles qui incluent un thread de nettoyage mémoire, le fait de ne pas avoir à se soucier de l'arrêt d'un thread à un moment inopportun constitue un plus. Les threads de l'espace utilisateur évoluent plus facilement, contrairement aux threads du noyau qui font invariablement appel à de l'espace pour les tables et la pile au sein même du noyau ; cela peut poser problème en présence d'un grand nombre de threads. Mais en dépit de leurs performances, les paquetages de threads utilisateur présentent plusieurs problèmes majeurs. Le premier est de savoir comment implémenter les appels système bloquants. Supposons qu'un thread effectue une lecture clavier avant que l'utilisateur n'ait appuyé sur une touche. Le fait de laisser le thread effectuer l'appel système est inacceptable dans la mesure où cela va arrêter tous les threads. Or, l'un des avantages des threads reste tout de même que chacun puisse utiliser des appels bloquants, mais tout en empêchant un thread bloqué d'affecter les autres. Avec les appels système bloquants, il devient difficile de cerner comment atteindre cet objectif directement.

Les appels système pourraient tous être des appels non bloquants (une lecture clavier retournerait simplement 0 octet si aucun caractère n'a été placé dans la mémoire tampon), mais demander des changements au système d'exploitation est peu intéressant.

En outre, l'un des arguments en faveur des threads utilisateur était précisément qu'ils pouvaient s'exécuter avec les systèmes d'exploitation existants. Par ailleurs, le fait d'intervenir sur la sémantique de `read` entraînerait bien des changements au niveau des programmes utilisateur.

Une autre possibilité se fait jour dans le cas où il est possible de dire à l'avance si un appel va provoquer un blocage. Dans certaines versions d'UNIX, un appel système, appelé `select`, permet à l'appelant de déterminer si un `read` à venir va provoquer un blocage. En présence d'un tel appel, la procédure de bibliothèque `read` peut être remplacée par une autre qui commence par émettre un appel `select`, puis qui émet l'appel `read` si celui-ci peut s'exécuter (c'est-à-dire qu'il ne va pas se bloquer). S'il s'avère que l'appel `read` va bloquer, il n'est pas lancé. C'est alors un autre thread qui va s'exécuter. Lors de la prochaine occasion pour le système d'exécution de reprendre le contrôle, il peut refaire une vérification pour déterminer si le `read` est devenu possible. Cette approche vous oblige à réécrire des parties de la bibliothèque des appels système ; elle est inefficace et peu judicieuse, mais vous n'avez pas tellement le choix. Le code intervenant avant et après l'appel système pour effectuer cette vérification s'appelle code **wrapper** (ou parfois **jacket**).

Le problème des défauts de pages est comparable à celui des appels système bloquants. Nous l'aborderons au chapitre 3. Pour l'instant, il suffit de dire que les ordinateurs peuvent être configurés de manière que la totalité du programme ne se trouve pas dans la mémoire principale. Si le programme appelle une instruction qui ne se trouve pas en mémoire, un défaut de page se produit, et le système d'exploitation va chercher l'instruction manquante (et celles qui l'avoisinent) sur le disque. Le processus est bloqué pendant que l'instruction sollicitée est localisée et lue. Si un thread provoque un défaut de page, le noyau, sans même connaître l'existence des threads, bloque naturellement le processus dans son ensemble jusqu'à ce que l'E/S disque soit terminée, même si d'autres threads pourraient s'exécuter pendant ce temps.

Les paquetages de threads au niveau utilisateur posent un autre problème : si un thread commence à s'exécuter, aucun autre thread de ce processus ne va s'exécuter à moins que le premier `read` ne cède volontairement la place. Dans le cadre d'un processus unique, il n'y a pas d'interruption d'horloge, ce qui rend impossible l'ordonnancement des processus par relais. À moins qu'un thread donné n'entre dans le système d'exécution de son propre gré, l'ordonnanceur n'a aucune chance de s'exécuter.

Une éventuelle solution au problème des threads qui s'exécuteraient sans fin est que le système d'exécution sollicite un signal d'horloge (une interruption) toutes les secondes, de sorte à récupérer le contrôle. Encore une fois, c'est une démarche de programmation grossière. Les interruptions périodiques à une fréquence élevée ne sont pas toujours possibles. Et même quand elles le sont, la surcharge engendrée risque d'être considérable. En outre, un thread peut aussi avoir besoin d'une interruption, qui interférerait alors avec l'utilisation de l'horloge par le système d'exécution.

Il existe un argument encore plus grave en la défaveur des threads utilisateur : les programmeurs développent généralement des threads précisément dans les applications où ceux-ci bloquent souvent, comme un serveur Web multithread. Ces threads effectuent

constamment des appels système. Une fois qu'un trap a eu lieu dans le noyau pour prendre en charge l'appel système, ce n'est plus qu'un détail pour le noyau de basculer d'un thread à l'autre en cas de blocage du plus ancien. Et si le noyau s'acquitte de ces tâches, cela élimine les appels système `select` répétés pour vérifier que les `read` sont non bloquants. Pour les applications qui sollicitent essentiellement le processeur et qui bloquent rarement, quel est donc l'intérêt d'avoir des threads ? Personne ne proposerait sérieusement de programmer le calcul des n nombres premiers ou un jeu d'échecs avec des threads, car il n'y aurait rien à gagner à le faire !

2.2.5 L'implémentation de threads dans le noyau

Supposons maintenant que le noyau ait connaissance des threads et en assure la gestion. Il n'est pas nécessaire de disposer d'un système d'exécution pour chacun, comme le montre la figure 2.16(b). En outre, il n'y a pas de table des threads dans chaque processus. C'est le noyau qui possède une table des threads chargée du suivi de tous les threads du système. Lorsqu'un nouveau thread veut créer ou détruire un thread existant, il effectue un appel noyau qui prend alors en charge la création ou la destruction en actualisant la table des threads du noyau.

La table des threads du noyau dispose, pour chaque thread, de registres, de l'état et d'autres informations. Ces informations sont les mêmes que pour les threads utilisateur, mais elles sont maintenant conservées dans le noyau et non plus dans l'espace utilisateur (le système d'exécution). Elles constituent un sous-ensemble des informations que les noyaux traditionnels maintiennent au sujet de leurs processus monothreads, à savoir l'état du processus. En outre, le noyau maintient également une table des processus traditionnelle pour suivre les processus. Tous les appels susceptibles de bloquer un thread sont implémentés sous forme d'appels système, ce qui est beaucoup plus complexe que de programmer un appel de procédure du système d'exécution. Lorsqu'un thread se bloque, le noyau peut, s'il le souhaite, exécuter un autre thread du même processus (s'il y en a un de prêt) ou un thread d'un autre processus. Avec les threads utilisateur, le système d'exécution continue d'exécuter des threads de son propre processus jusqu'à ce que le noyau cesse de lui allouer du temps processeur (ou qu'il ne reste plus de threads prêts à s'exécuter). Étant donné qu'il est relativement lourd de créer et de détruire des threads dans le noyau, certains systèmes adoptent une approche « environnementalement correcte » et recyclent leurs threads. Lorsqu'un thread est détruit, il est marqué comme étant non exécutable, mais ses structures de données noyau n'en sont pas autrement affectées. Ensuite, lorsqu'un nouveau thread doit être créé, un ancien thread est réactivé, ce qui réduit la surcharge. Le recyclage des threads est également envisageable pour les threads utilisateur, mais étant donné que la surcharge engendrée par la gestion des threads est nettement inférieure, il est moins intéressant de s'en soucier.

Les threads noyau ne nécessitent aucun appel système bloquant supplémentaire. En outre, si un thread d'un processus donné provoque un défaut de page, le noyau peut aisément déterminer si le processus possède d'autres threads exécutables ; si c'est le cas, il peut en exécuter un en attendant que la page sollicitée soit récupérée sur le disque. Le principal inconvénient des threads noyau est qu'un appel système est une

démarche lourde ; si les opérations sur les threads (création, arrêt, etc.) sont nombreuses, la surcharge sera importante.

Pour autant, les threads noyau ne résolvent pas tous les problèmes. Par exemple, que se passe-t-il lorsqu'un processus multithread fait un `fork` ? Le nouveau processus a-t-il autant de threads que l'ancien ou n'en a-t-il qu'un ? Bien souvent, le meilleur choix dépend de ce qu'il y a à faire après. Si l'on s'apprête à faire un `exec` pour démarrer un nouveau programme, il est probable qu'un seul thread est le bon choix, mais si c'est pour continuer l'exécution tous les threads sont préférables.

Autre point : la gestion des interruptions. Les interruptions sont envoyées aux processus, pas aux threads, au moins dans le modèle classique. Lorsqu'une interruption survient, quel est le thread qui va la prendre en compte ? Un thread peut manifester son intérêt pour tel ou tel type d'interruption et ainsi se porter candidat pour la traiter, mais qu'arrive-t-il si plusieurs threads veulent traiter la même interruption ? Bien d'autres problèmes sont ainsi posés par les threads.

2.2.6 Les implémentations hybrides

Plusieurs méthodes ont été testées pour tenter de combiner les avantages des threads utilisateur et noyau. L'une d'elles consiste à employer des threads noyau, puis à multiplexer les threads utilisateur sur un ou plusieurs threads noyau, comme le montre la figure 2.17. C'est le programmeur qui détermine le nombre de threads noyau ainsi que le nombre de threads utilisateur multiplexés sur chacun d'eux.

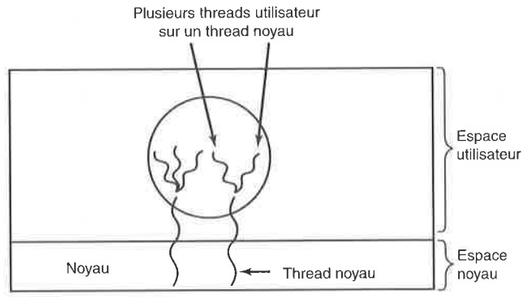

Figure 2.17 • Multiplexer les threads utilisateur et les threads noyau.

Dans cette conception, le noyau ne connaît *que* les threads noyau, et il les ordonnance. Certains de ces threads peuvent disposer de threads utilisateur multiplexés. Ceux-ci sont créés, détruits et ordonnancés exactement comme les threads utilisateur d'un processus qui s'exécute sur un système d'exploitation dépourvu de fonctionnalités de multithreading. Dans ce modèle, chaque thread noyau possède un jeu donné de threads utilisateur qui l'utilisent tour à tour.

2.2.7 Les activations de l'ordonnanceur

Les threads noyau présentent beaucoup d'avantages, mais ils sont notablement plus lents à l'exécution que les threads utilisateur. Les chercheurs ont donc tenté de combiner les avantages des threads utilisateur (leurs performances) et ceux des threads noyau (simples à mettre en œuvre). Nous allons maintenant décrire une approche nommée **activation de l'ordonnancement**.

Les objectifs de l'activation de l'ordonnanceur consistent à imiter la fonctionnalité des threads noyau, mais avec les performances supérieures et la plus grande souplesse offertes par les paquetages de threads mis en œuvre dans l'espace utilisateur. En particulier, les threads utilisateur ne devraient pas avoir à effectuer d'appels système spéciaux non bloquants, ni à déterminer par avance si certains appels sont non bloquants. Il reste que, lorsqu'un thread bloque sur un appel système ou un défaut de page, il devrait être possible d'exécuter d'autres threads dans le même processus, s'il y en a de prêts. Pour être efficace, cette méthode évite les transitions inutiles entre les espaces utilisateur et noyau. Si un thread se bloque en attendant qu'un autre fasse quelque chose, par exemple, il n'y a pas de raison de faire intervenir le noyau. On peut ainsi éviter des pertes de performances dues aux transitions entre les deux espaces. Le système d'exécution de l'espace utilisateur peut bloquer le thread de synchronisation et en planifier un autre lui-même.

Lorsque l'on emploie les activations de l'ordonnanceur, le noyau assigne un certain nombre de processeurs virtuels à chaque processus et laisse le système d'exécution (l'espace utilisateur) allouer des threads aux processeurs. Ce mécanisme peut également être exploité sur un multiprocesseur dans lequel les processeurs virtuels peuvent être de véritables UC. Le nombre de processeurs virtuels alloués à un processus est initialement de un, mais le processus peut en solliciter d'autres ; il peut aussi retourner les processeurs dont il n'a plus besoin. Le noyau peut également récupérer des processeurs virtuels déjà alloués afin de les assigner à d'autres processus prioritaires.

Ce schéma fonctionne sur le principe suivant : lorsque le noyau « voit » qu'un thread est bloqué (par exemple du fait d'un appel système bloquant ou d'un défaut de page), il avertit le système d'exécution du processus, en passant à la pile le numéro du thread en question et une description de l'événement comme paramètres. Pour que la notification ait lieu, le noyau active le système d'exécution à une adresse de départ connue, comparable à un signal sous UNIX. Ce mécanisme s'appelle un **upcall**.

Une fois activé de cette manière, le système d'exécution peut replanifier ses threads, généralement en marquant le thread en cours comme bloqué et en prélevant un autre thread dans la liste des threads prêts, en définissant ses registres, et en le réactivant. Ultérieurement, lorsque le noyau s'aperçoit que le thread initial peut s'exécuter à nouveau (par exemple le tube à partir duquel il effectuait une lecture contient désormais des données, ou la page ayant provoqué l'erreur a été récupérée sur le disque), il effectue un autre upcall en direction du système d'exécution pour l'informer de cet événement. Le système d'exécution a le choix : il peut redémarrer immédiatement le thread bloqué ou le placer dans la liste des threads prêts pour une exécution ultérieure.

Lorsqu'une interruption matérielle se produit pendant qu'un thread utilisateur s'exécute, le processeur interrompu bascule en mode noyau. Si l'interruption est causée par un événement sans intérêt pour le processus interrompu (comme c'est le cas de l'achèvement des E/S d'un autre processus), une fois que le gestionnaire d'interruption a terminé de s'exécuter, il replace le thread interrompu dans l'état où il se trouvait avant l'interruption. Cependant, si le processus est concerné par l'interruption (comme c'est le cas de l'arrivée d'une page nécessaire à l'un des threads du processus), le thread interrompu ne redémarre pas. Il est, au contraire, suspendu, et le système d'exécution démarre sur ce processeur virtuel, avec l'état du thread interrompu dans la pile. Le système d'exécution n'a plus qu'à décider du thread à planifier sur ce processeur : celui qui a été interrompu, celui qui vient de passer à l'état prêt ou un autre thread.

On peut reprocher aux activations de l'ordonnanceur qu'elles reposent entièrement sur les upcalls, un concept qui entre en contradiction avec la structure inhérente à tout système en couches. Normalement, la couche *n* offre certains services sur lesquels la couche *n* + *1* peut compter, mais la couche *n* ne peut pas appeler des procédures de la couche *n* + *1*. Les upcalls ne respectent pas ce principe fondamental.

2.2.8 Les threads spontanés

Les threads sont souvent utiles dans les systèmes distribués. Un exemple important en est la gestion des messages entrants, par exemple les requêtes de service. L'approche traditionnelle consiste à avoir un processus ou un thread qui est bloqué sur un appel système récepteur en attente d'un message entrant. Lors de l'arrivée d'un message, il accepte le message et le traite.

Cependant, une approche totalement différente est envisageable, selon laquelle l'arrivée d'un message provoque la création d'un nouveau thread par le système, qui va prendre le message en charge. C'est ce que l'on appelle un **thread spontané** (*pop-up thread*) [voir figure 2.18]. Avantage : étant donné qu'il s'agit systématiquement de nouveaux threads, ils n'ont pas d'historique (registre, pile, etc.) à restaurer. Chaque thread démarre « tout neuf » et tous sont identiques. Cela permet de les créer rapidement. Le nouveau thread reçoit un message entrant à traiter. Les threads spontanés ont pour résultat de générer une latence très courte entre l'arrivée du message et le début du traitement.

Une certaine planification préalable est nécessaire pour travailler avec les threads spontanés. Par exemple, dans quel processus s'exécutent-ils ? Si le système supporte l'exécution des threads dans le contexte du noyau, le thread peut s'y exécuter (raison pour laquelle nous n'avons pas fait apparaître le noyau à la figure 2.18). Il est généralement plus rapide et facile de faire en sorte que le thread spontané s'exécute dans l'espace du noyau plutôt que dans l'espace utilisateur. En outre, un thread spontané agissant dans l'espace du noyau peut aisément accéder à toutes les tables du noyau et aux périphériques d'E/S, ce qui peut être nécessaire pour le traitement des interruptions.

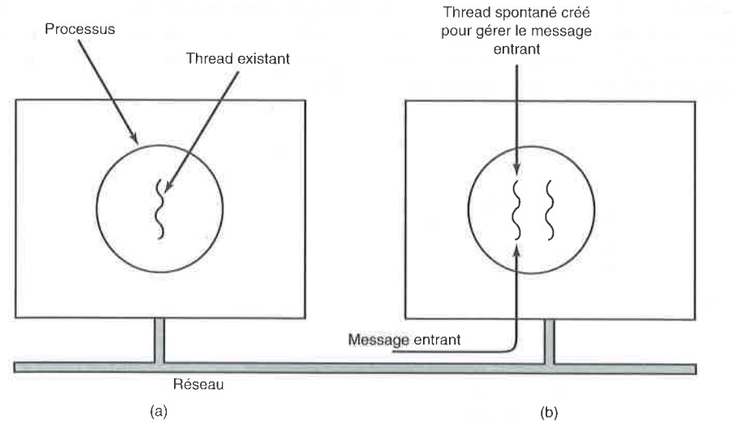

Figure 2.18 • Création d'un nouveau thread lors de l'arrivée d'un message. (a) Avant l'arrivée du message. (b) Après l'arrivée du message.

D'autre part, un thread noyau bogué peut faire plus de ravages qu'un thread utilisateur bogué. Par exemple, s'il s'exécute pendant trop longtemps et qu'il n'existe pas de moyen de le préempter, les données entrantes peuvent être perdues.

2.2.9 Du code monothread au code multithread

De nombreux programmes sont écrits pour les processus monothreads. Les convertir en programmes multithreads est bien plus délicat que l'on ne pourrait l'imaginer. Nous allons identifier quelques pièges à éviter.

Pour commencer, le code d'un thread se compose normalement de plusieurs procédures, exactement comme pour un processus. Celles-ci peuvent avoir des variables locales, des variables globales et des paramètres. Les variables locales et les paramètres ne posent aucun problème, contrairement aux variables qui sont globales pour un thread, mais pas pour l'ensemble du programme. Il s'agit de variables globales dans le sens où de nombreuses procédures du thread les utilisent (comme elles pourraient utiliser n'importe quelle variable globale), mais les autres threads ne devraient normalement pas y toucher.

Par exemple, prenons la variable *errno* maintenue par UNIX. Lorsqu'un processus (ou un thread) effectue un appel système qui échoue, le code d'erreur est placé dans cette variable. À la figure 2.19, le thread 1 exécute l'accès à l'appel système pour savoir s'il a la permission d'accéder à un certain fichier. Le système d'exploitation retourne la réponse dans la variable *errno*. Une fois que le contrôle retourne « entre les mains » du thread 1, mais avant qu'il n'ait pu lire la valeur de *errno*, l'ordonnanceur juge que le thread 1 a disposé de suffisamment de temps processeur pour le moment, et

bascule vers le thread 2. Ce dernier exécute un appel open qui échoue, ce qui provoque le remplacement de la valeur de *errno*, et le code d'accès du thread 1 est irrémédiablement perdu. Lorsque, ultérieurement, le thread 1 redémarre, il lit la mauvaise valeur et adopte ainsi un comportement erroné.

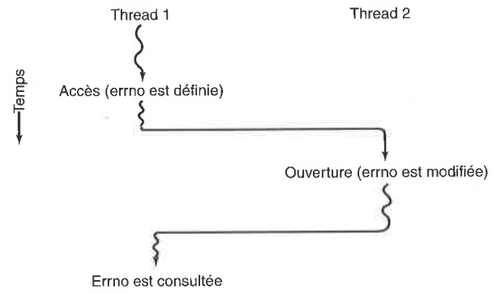

Figure 2.19 • Conflits entre les threads concernant l'utilisation d'une variable globale.

Il existe diverses manières de résoudre ce problème. L'une d'elles consiste à interdire purement et simplement les variables globales. Aussi intéressante soit-elle, cette solution entraînerait des conflits avec la plupart des logiciels existants. Une autre consiste à assigner à chaque thread ses propres variables globales, comme le montre la figure 2.20. De cette manière, chaque thread possède sa propre copie de *errno* et d'autres variables globales, ce qui permet d'éviter les conflits. En effet, cette décision crée un nouveau niveau de portée pour les variables, un niveau de variables visibles par toutes les procédures d'un thread. Ce niveau vient s'ajouter aux portées existantes de variables visibles par une seule procédure et de variables visibles depuis tous les « endroits » du programme.

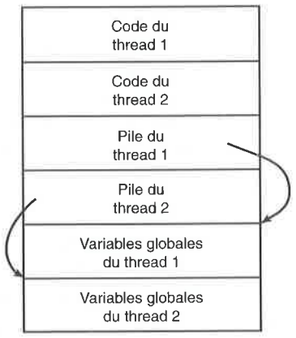

Figure 2.20 • Les threads peuvent avoir des variables globales privées.

Il n'est cependant pas si simple d'accéder à des variables globales privées. La plupart des langages de programmation ont une manière propre d'exprimer les variables locales et globales, mais ne prévoient pas de formes intermédiaires. Il est possible d'allouer une portion de mémoire aux variables globales, et de la passer à chaque procédure du thread sous forme de paramètre supplémentaire. Si la solution n'est guère élégante, elle a le mérite de fonctionner.

Une autre solution consiste à introduire de nouvelles procédures de bibliothèque qui permettent de créer, de définir et de lire ces variables globales dont la portée est limitée aux threads. Voici le premier appel :

```
create_global("bufptr");
```

Cet appel alloue de la mémoire à un pointeur, appelé `bufptr`, dans le tas (segment de mémoire) ou dans une zone de stockage spéciale réservée au thread appelant. L'emplacement de cette zone n'a pas d'importance dans la mesure où seul le thread a accès à la variable globale. Si un autre thread crée une variable globale portant le même nom, celle-ci aura un emplacement différent, ce qui évitera les conflits avec la variable existante.

Deux appels sont nécessaires pour accéder aux variables globales : l'un pour les écrire et l'autre pour les lire. Pour écrire, la ligne suivante :

```
set_global("bufptr", &buf);
```

fera l'affaire. Elle stocke la valeur d'un pointeur à l'emplacement préalablement créé par l'appel de `create_global`. Pour la lecture d'une variable globale, vous pouvez écrire :

```
bufptr = read_global("bufptr");
```

ce qui retourne l'adresse stockée dans la variable globale, de façon que l'on puisse accéder à ses données. Le second problème posé par la transformation d'un programme monothread en programme multithread est que de nombreuses procédures de bibliothèque ne sont pas réentrantes. Cela signifie qu'elles n'ont pas été conçues pour qu'un second appel soit effectué en direction d'une procédure donnée pendant qu'un appel antérieur est toujours en cours. Par exemple, pour envoyer un message en réseau, on peut très bien programmer l'assemblage du message dans un tampon fixe, dans la bibliothèque, puis préempter le noyau pour qu'il l'envoie. Que se passe-t-il si un thread ayant assemblé son message dans le tampon, une interruption force un basculement vers un second thread qui remplace immédiatement le tampon contenant son propre message ?

De la même façon, les procédures d'allocation de mémoire, comme `malloc` sous UNIX, maintiennent des tables essentielles pour l'utilisation de la mémoire, comme une liste liée de portions de mémoire disponibles. Pendant que `malloc` est occupée à la mise à jour de ces listes, celles-ci peuvent présenter temporairement un état incohérent, avec des pointeurs ne pointant nulle part. Si un basculement de thread se produit pendant cet intervalle et qu'un nouvel appel soit initié par un autre thread, un pointeur non valide pourra être utilisé, ce qui conduira à un arrêt inopiné du programme.

La résolution de tous ces problèmes revient très exactement à réécrire toute la bibliothèque, ce qui n'est pas vraiment simple.

Une autre solution consiste à ajouter du code wrapper à chaque procédure afin de définir un bit pour marquer la bibliothèque comme utilisée. Tant que l'appel antérieur n'a pas été entièrement exécuté, toute tentative émanant d'un autre thread pour utiliser une procédure de bibliothèque est bloquée. Bien qu'il soit possible de rendre cette approche opérationnelle, cela pèse lourdement sur le parallélisme potentiel des threads.

Prenons maintenant les signaux. Certains sont logiquement spécifiques aux threads, tandis que d'autres ne le sont pas. Par exemple, si un thread invoque `alarm`, il est logique que le signal qui en résulte soit transmis au thread originaire de l'appel. Or, lorsque les threads sont implémentés entièrement au sein de l'espace utilisateur, le noyau ignore leur existence ; il peut donc difficilement diriger le signal vers le bon thread. Une complication supplémentaire surgit si un processus ne peut avoir qu'une alarme en cours à un instant donné alors que plusieurs threads peuvent invoquer `alarm` indépendamment.

D'autres signaux, comme les interruptions clavier, ne sont pas spécifiques aux threads. Qui doit les intercepter ? Un thread désigné ? Tous les threads ? Un nouveau thread spontané qui vient d'être créé ? Par ailleurs, que se passe-t-il si un thread modifie les gestionnaires de signal sans en informer les autres threads ? Et quid d'un thread qui souhaite intercepter un signal particulier (par exemple, l'utilisateur a appuyé sur Ctrl+C) alors qu'un autre thread, lui, veut que ce signal arrête le processus. De telles situations peuvent se produire si un ou plusieurs threads exécutent des procédures de bibliothèque standard tandis que d'autres sont développés par le programmeur. Il est clair que cela pose des problèmes de compatibilité. En général, les signaux sont suffisamment difficiles à gérer dans un environnement monothread. Le passage en multithread n'en facilite pas la prise en charge.

Le dernier problème est celui de la gestion de la pile. Dans de nombreux systèmes, lorsqu'une pile de processus déborde, le noyau lui alloue automatiquement de l'espace supplémentaire. Lorsqu'un processus possède plusieurs threads, il doit également avoir plusieurs piles. Si le noyau n'est pas conscient de la présence de ces piles, il ne peut pas les « agrandir » automatiquement lorsque survient une erreur de pile. En fait, il risque même de ne pas réaliser que l'erreur mémoire est due à l'augmentation de l'espace nécessaire à une pile.

Ce ne sont certes pas des problèmes insurmontables, mais ils montrent que le simple fait d'introduire des threads dans un système existant sans passer par une reconception substantielle du système est une démarche qui ne peut pas fonctionner. La sémantique des appels système doit être redéfinie, et les bibliothèques doivent être réécrites, à tout le moins. Et tout cela doit être fait de façon à assurer la compatibilité descendante avec les programmes existants pour le cas limité d'un processus ne possédant qu'un seul thread.

2.3 La communication interprocessus

Il arrive souvent que les processus aient besoin de communiquer entre eux. Par exemple, dans un pipeline du shell, la sortie du premier processus doit être passée au deuxième processus, et ainsi de suite. Il existe donc un besoin de communication entre les processus, de préférence de façon structurée et en évitant les interruptions. Dans les sections suivantes, nous aborderons quelques-uns des problèmes liés à cette **communication interprocessus** (*IPC, InterProcess Communication*).

On rencontre essentiellement trois problèmes. Le premier coule de source : comment un processus fait-il pour passer des informations à un autre processus ? Le deuxième repose sur la nécessité de s'assurer que deux processus, ou plus, ne produisent pas de conflits lorsqu'ils s'engagent dans des activités critiques (deux processus de réservation aérienne tentent de récupérer pour deux clients différents le dernier siège disponible). Le troisième concerne le séquençage en présence de dépendances : si le processus *A* produit des données et que le processus *B* les imprime, *B* doit attendre que *A* ait terminé pour pouvoir remplir sa tâche. Nous verrons ces trois problématiques l'une après l'autre à partir de la section suivante.

Il est important de mentionner que deux de ces problèmes s'appliquent également aux threads. Pour ce qui est du passage de l'information, cela est aisé pour les threads dans la mesure où ceux-ci partagent un espace d'adressage commun (les threads qui sont situés dans des espaces d'adressage différents et qui ont besoin de communiquer entrent dans le cadre de la communication interprocessus). Cependant, les deux autres problèmes, à savoir le fait qu'ils s'évitent les uns les autres et que leur séquençage soit approprié, s'appliquent bel et bien aux threads. Aux mêmes problèmes, les mêmes solutions. Nous parlerons donc de ces problèmes dans le contexte des processus, mais n'oubliez pas que les points abordés sont également applicables aux threads.

2.3.1 Les conditions de concurrence

Dans certains systèmes d'exploitation, les processus qui travaillent ensemble peuvent partager un espace de stockage commun dans lequel chacun peut lire et écrire. Cet espace de stockage partagé peut se trouver dans la mémoire principale (éventuellement dans une structure de données du noyau), ou il peut s'agir d'un fichier partagé. L'emplacement de la mémoire partagée ne modifie pas la nature de la communication, ni celle des problèmes qui peuvent survenir. Pour voir comment fonctionne la communication interprocessus dans la pratique, prenons un exemple simple, mais courant : le spouleur d'impression. Lorsqu'un processus veut imprimer un fichier, il entre son nom dans un **répertoire de spoule** spécial. Un autre processus, le **démon d'impression**, regarde périodiquement s'il y a des fichiers à imprimer ; si c'est le cas, il les imprime et supprime leurs noms du répertoire. Supposons que notre répertoire de spoule possède un grand nombre d'entrées, numérotées 0, 1, 2, et ainsi de suite, chacune pouvant accueillir un nom de fichier. Disons qu'il existe deux variables partagées : `out`, qui pointe vers le prochain fichier à imprimer, et `in`, qui pointe vers la prochaine entrée libre du répertoire. Ces deux variables peuvent très bien être