2

Processus et threads

Nous allons maintenant nous lancer dans une étude détaillée de la conception et de la construction des systèmes d'exploitation. Le concept central de tout système d'exploitation est le *processus*. Un processus est une abstraction d'un programme en cours d'exécution. Tout le reste repose sur ce concept, et il est important que le concepteur du système d'exploitation (et l'étudiant) maîtrise parfaitement la notion de processus le plus tôt possible.

Les processus sont une des plus anciennes et des plus importantes abstractions des systèmes d'exploitation. Grâce au (pseudo)-parallélisme, ils permettent de transformer une unique UC en un ensemble d'UC virtuelles. Sans eux, l'informatique moderne n'existerait pas. Ce chapitre est consacré à leur présentation ainsi qu'à celle de leurs cousins germains, les threads.

2.1 Les processus

Tous les ordinateurs modernes sont capables de faire plusieurs choses simultanément. Un serveur Web, par exemple, doit être capable de savoir si la page demandée est en cache. Si elle y est, il l'envoie et si elle n'y est pas, il lance une requête disque pour la charger. Mais du point de vue de l'UC, une requête disque prend un temps énorme. Pendant qu'on attend que la page soit chargée, de nombreuses demandes de pages peuvent survenir. S'il y a plusieurs disques, il se peut qu'on doive accéder à certains d'entre eux bien avant que la première requête soit satisfaite. De toute évidence, il est nécessaire de modéliser et de contrôler ce parallélisme. C'est ce à quoi nous aident les processus et les threads.

Prenons maintenant le cas d'un PC. Lorsqu'on le met en marche, de nombreux processus se lancent, processus le plus souvent inconnus de l'utilisateur. On peut trouver ainsi un processus qui attend l'arrivée des courriels, un autre qui passe le disque à l'antivirus dès que de nouvelles souches antivirales sont disponibles, etc. Si l'on ajoute tous les processus dont l'utilisateur est conscient – impression d'un fichier,

copie d'un CD, surf sur le Web, etc. –, on comprend l'importance d'une bonne gestion des processus.

Dans tout système multiprogrammé, l'UC passe d'un processus à l'autre, pour les servir à raison de quelques dizaines ou centaines de millisecondes chacun. S'il est vrai que, *stricto sensu*, le processeur (l'UC) n'exécute qu'un seul processus à la fois, il peut intervenir pour plusieurs programmes en l'espace d'une seconde, ce qui va donner à l'utilisateur l'illusion de la simultanéité. Certains appellent cela du **pseudo-parallélisme**, pour marquer une différence avec le véritable **parallélisme** matériel des systèmes multiprocesseurs dans lesquels deux processeurs ou plus partagent une même mémoire physique. Or, il n'est pas évident de suivre plusieurs activités en parallèle. C'est pourquoi les concepteurs de système d'exploitation ont élaboré, au fil des années, un modèle conceptuel reposant sur les processus séquentiels, qui facilite la gestion de ce parallélisme. Ce modèle, ses utilisations et certaines de ses conséquences font l'objet de ce chapitre.

2.1.1 Le modèle de processus

Dans ce modèle, tous les logiciels exécutables de l'ordinateur, et parfois même le système d'exploitation, sont organisés en un certain nombre de **processus séquentiels**, que l'on appellera **processus** pour être plus brefs. Un processus est tout simplement un programme qui s'exécute, avec les valeurs du compteur ordinal des registres et des variables. Du point de vue conceptuel, chaque processus possède son propre processeur virtuel (sa propre UC virtuelle). En réalité, le processeur bascule constamment d'un processus à l'autre. Mais pour comprendre le système, il est plus aisé d'y penser comme à une collection de processus s'exécutant (pseudo) parallèlement, que d'essayer de suivre la manière dont le processeur bascule entre les programmes. Ce basculement rapide est appelé **multiprogrammation**, comme nous l'avons vu au chapitre 1.

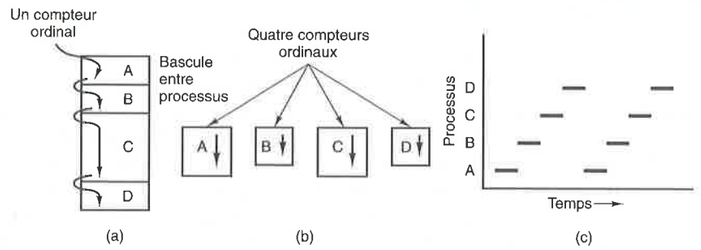

Figure 2.1 • (a) Multiprogrammation de quatre programmes. (b) Modèle conceptuel de quatre processus séquentiels indépendants. (c) Un seul programme est actif à un instant donné.

À la figure 2.1(a), vous voyez un ordinateur gérant quatre programmes en mémoire. À la figure 2.1(b), vous observez la présence de quatre processus, chacun avec son propre flot de contrôle (autrement dit, avec son propre compteur ordinal logique).

Chacun s'exécute indépendamment des autres. Naturellement, il n'existe qu'un seul compteur ordinal physique ; ainsi lorsque les processus s'exécutent, leur compteur ordinal logique est chargé dans le véritable compteur ordinal. Lorsqu'il a terminé de s'exécuter (pour le moment), le compteur ordinal physique est enregistré dans le compteur ordinal logique du processus en mémoire. À la figure 2.1(c), vous voyez que sur un intervalle assez long, tous les processus ont progressé, mais qu'à un instant donné, un seul processus s'exécute vraiment.

Dans ce chapitre, nous supposerons que nous n'avons qu'un seul processeur. C'est une hypothèse de moins en moins vraie puisqu'on peut disposer maintenant de puces multicœurs de deux ou quatre processeurs si ce n'est pas plus. Nous analyserons ces multicœurs au chapitre 8, mais pour le moment contentons-nous d'un seul processeur.

Lorsque le processeur passe d'un processus à un autre, la vitesse de traitement d'un processus donné n'est pas uniforme, et probablement non reproductible si le même processus s'exécute une nouvelle fois. C'est pourquoi il ne faut pas programmer des processus en effectuant des hypothèses de temporisation. Prenons l'exemple d'un processus d'E/S qui démarre un lecteur de bandes pour restaurer des fichiers sauvegardés, qui exécute une boucle à vide 10 000 fois pour permettre au lecteur d'arriver à sa vitesse de rotation, puis qui émet une commande de lecture du premier enregistrement. Si le processeur décide de passer à un autre processus au cours de l'exécution de la boucle à vide, le processus du lecteur risque de s'exécuter à nouveau une fois que la tête de lecture aura dépassé le premier enregistrement. Lorsqu'un processus est ainsi assorti d'exigences de temps réel critiques, autrement dit lorsqu'un événement particulier *doit* se produire dans un intervalle de temps spécifié en millisecondes, il est nécessaire de prendre des mesures spéciales afin qu'il se produise. Mais normalement, la plupart des processus ne sont pas affectés par la multiprogrammation sous-jacente de l'UC, ni par les vitesses relatives des différents processus.

La différence entre un processus et un programme est subtile, mais cruciale. Une analogie pourrait ici aider le lecteur à mieux comprendre. Prenons l'exemple d'un informaticien porté sur la gastronomie, en train de confectionner un gâteau d'anniversaire pour sa sœur. Il dispose d'une cuisine équipée, il a la recette du gâteau et tous les ingrédients nécessaires : farine, œufs, sucre, extrait de vanille, etc. Dans cette métaphore, la recette représente le programme (c'est-à-dire un algorithme exprimé dans la notation appropriée), l'informaticien représente le processeur (l'unité centrale) et les ingrédients sont les données d'entrée. Le processus est l'activité qui consiste à confectionner le gâteau : lecture de la recette, mélange des ingrédients et cuisson.

Imaginons maintenant que notre informaticien se mette subitement à hurler parce qu'il vient d'être piqué par une guêpe. Son cerveau enregistre le point de la recette où il en était arrivé (l'état dans lequel le processus en cours est enregistré). Il va chercher une brochure sur les premiers secours et commence à en suivre les instructions. C'est ici que l'on voit notre processeur basculer d'un processus (confection du gâteau) vers un autre processus de priorité supérieure (administrer un antidote). Chaque processus découle d'un programme différent (recette et brochure sur les premiers soins).

Une fois le problème de la piqûre résolu, notre informaticien retourne à son gâteau, au point où il l'avait laissé.

Cela pour vous montrer qu'un processus est une activité. Il inclut un programme, une entrée, une sortie et un état. Un processeur peut être partagé par plusieurs processus, à l'aide d'un algorithme d'ordonnancement intervenant pour déterminer à quel moment arrêter de travailler sur un processus pour en servir un autre.

Notez que si un programme s'exécute deux fois, il donne lieu à deux processus. Il est ainsi possible d'imprimer deux fichiers en même temps avec le même traitement de texte si l'on dispose de deux imprimantes. Le fait que les deux processus soient issus du même programme est sans importance : le système d'exploitation est sans doute capable de n'avoir en mémoire qu'une seule copie du code du traitement de texte mais conceptuellement on a bien deux processus distincts qui s'exécutent.

2.1.2 La création d'un processus

Les systèmes d'exploitation ont besoin de savoir créer des processus. Sur les systèmes les plus simples, ceux conçus pour exécuter une seule application (par exemple le contrôleur d'un four à micro-ondes), il peut être envisageable que tous les processus susceptibles d'intervenir soient lancés au démarrage du système. En revanche, les systèmes généralistes ont besoin de disposer d'une méthode pour créer et arrêter les processus, selon le cas, au cours de leur activité.

Il existe essentiellement quatre événements provoquant la création de processus :

1. Initialisation du système.
2. Exécution d'un appel système de création de processus par un processus en cours d'exécution.
3. Requête utilisateur sollicitant la création d'un nouveau processus.
4. Lancement d'un travail en traitement par lots.

Lors de l'amorçage du système d'exploitation, plusieurs processus sont créés. Certains sont des processus de premier plan, à savoir des processus qui interagissent avec l'utilisateur et accomplissent des tâches pour eux. D'autres sont des processus d'arrière-plan, non associés à une utilisation particulière de l'ordinateur. Ces processus ont des fonctions spécifiques. On peut trouver par exemple un processus d'arrière-plan conçu pour accepter les courriers électroniques entrants ; celui-ci sera inactif la plupart du temps, mais se réactivera lors de l'arrivée d'un message. Autre exemple : un processus d'arrière-plan peut être conçu pour accepter les requêtes entrantes de pages Web hébergées sur l'ordinateur et s'activer lors de l'arrivée d'une requête. Les processus qui restent à l'arrière-plan pour gérer des activités qui concernent les courriers électroniques, les pages Web, les news, les impressions, etc., s'appellent des **démons** (*daemons*). Sur les gros systèmes, on en trouve généralement des dizaines. Sous UNIX, le programme `ps` sert à afficher la liste des processus en cours d'exécution. Sous Windows, c'est le Gestionnaire des tâches qui intervient.

Au-delà des processus créés lors de l'amorçage, de nouveaux processus peuvent être créés après coup. Il arrive souvent qu'un processus en cours d'exécution émette des appels système pour créer un ou plusieurs nouveaux processus qui l'aideront dans son activité. Il est particulièrement utile de créer de nouveaux processus lorsque la tâche à accomplir peut être divisée en plusieurs processus, qui interagissent entre eux tout en étant indépendants. Par exemple, si l'on récupère d'importantes quantités de données sur un réseau afin de les traiter ultérieurement, il peut être pratique de créer un processus pour récupérer les données et les placer dans un tampon partagé, tandis qu'un second processus ôtera les éléments de données et les traitera. Sur un ordinateur multiprocesseur, le fait que chaque processus puisse s'exécuter sur un processeur différent peut également en accélérer l'exécution.

Sur les systèmes interactifs, les utilisateurs peuvent démarrer un programme en saisissant une commande ou en (double) cliquant sur une icône. L'une ou l'autre de ces actions lance un nouveau processus qui exécute le programme concerné. Sur les systèmes UNIX exécutant X Window, le nouveau processus prend possession de la fenêtre dans laquelle il a été démarré. Avec Microsoft Windows, lorsqu'un processus démarre, il ne possède pas de fenêtre, mais peut en créer une ou plusieurs. C'est ce qui se produit le plus souvent. Sur ces deux systèmes d'exploitation, les utilisateurs peuvent ouvrir plusieurs fenêtres simultanément, chacune exécutant un processus donné. À l'aide de la souris, l'utilisateur peut sélectionner une fenêtre et entrer en interaction avec le processus, par exemple en saisissant des données.

Le dernier cas de figure impliquant la création de processus ne s'applique qu'aux systèmes de traitement par lots que l'on trouve sur les gros mainframes. Les utilisateurs peuvent soumettre des tâches de traitement par lots au système (même à distance). Lorsque le système d'exploitation constate qu'il dispose des ressources nécessaires à l'exécution d'un job supplémentaire, il crée un nouveau processus et exécute le job suivant de la file d'attente.

Techniquement, dans tous ces cas de figures, un nouveau processus est créé du fait qu'un processus existant exécute un appel système de création de processus. Il peut s'agir d'un processus utilisateur en cours, d'un processus système invoqué à partir du clavier ou de la souris, ou d'un processus gestionnaire de traitement par lots. Cet appel demande au système d'exploitation de créer un nouveau processus et indique, directement ou indirectement, quel programme y exécuter.

Sous UNIX, le seul appel système permettant de créer un nouveau processus est l'appel `fork`, qui crée un clone du processus appelant. Après l'appel `fork`, les deux processus, parent et enfant, ont la même image mémoire et les mêmes fichiers ouverts. Généralement, le processus enfant exécute alors `execve`, ou un autre appel comparable, pour modifier son image mémoire et exécuter un nouveau programme. Par exemple, lorsqu'un utilisateur adresse une commande `sort` au shell, ce dernier engendre un processus enfant qui exécute `sort`. Ce procédé en deux étapes permet à l'enfant de manipuler ses descripteurs de fichiers après l'appel `fork`, mais avant `execve`, pour effectuer la redirection de l'entrée standard, de la sortie standard et de l'erreur standard.

À l'inverse, sous Windows, un seul appel de fonction Win32, `CreateProcess`, prend en charge à la fois la création du processus et le chargement du programme approprié dans le nouveau processus. Cet appel prend dix paramètres, et notamment le programme à exécuter, les paramètres de la ligne de commande destinés au programme, différents attributs de sécurité, des bits de contrôle permettant de vérifier l'héritage des fichiers ouverts, des informations de priorité, la déclaration de la fenêtre que le processus doit créer (s'il y en a une) et un pointeur vers une structure contenant les informations relatives au processus nouvellement créé à retourner à l'appelant. Outre l'appel `CreateProcess`, Win32 inclut environ une centaine d'autres fonctions pour gérer et synchroniser les processus, entre autres tâches.

Tant sous UNIX que sous Windows, une fois qu'un processus a été créé, le parent et le fils disposent désormais de leur propre espace d'adressage. S'il arrive que l'un des processus modifie un mot dans son espace d'adressage, le changement n'est pas visible par l'autre processus. Sous UNIX, l'espace d'adressage initial du fils est une *copie* de celui du parent, mais il existe deux espaces d'adressage distincts, et il n'y a pas de mémoire partagée en écriture (certaines implémentations d'UNIX partagent le texte du programme entre les deux processus dans la mesure où il n'est pas modifiable). En revanche, il est possible qu'un processus nouvellement créé partage certaines ressources avec son créateur, comme c'est le cas des fichiers ouverts. Sous Windows, les espaces d'adressage du parent et du fils sont différents dès le début.

2.1.3 La fin d'un processus

Une fois qu'un processus a été créé, il commence à s'exécuter, quelle que soit sa tâche. Cependant, rien n'est éternel, pas même un processus. Tôt ou tard, le nouveau processus s'arrête pour diverses raisons :

1. Arrêt normal (volontaire).
2. Arrêt pour erreur (volontaire).
3. Arrêt pour erreur fatale (involontaire).
4. Le processus est arrêté par un autre processus (involontaire).

La plupart des processus s'arrêtent parce qu'ils ont terminé la tâche qui leur incombait. Lorsqu'un compilateur a terminé de compiler le programme qu'on lui a confié, il exécute un appel système pour indiquer cet état de fait au système d'exploitation. Cet appel est un `exit` sous UNIX et un `ExitProcess` sous Windows. Les programmes que l'on peut afficher à l'écran prennent également en charge l'arrêt volontaire. Les traitements de texte, les navigateurs internet et autres programmes analogues sont équipés d'une icône ou d'un élément de menu qu'on peut utiliser pour indiquer au processus de supprimer les fichiers temporaires ouverts, puis pour arrêter celui-ci.

Le deuxième motif d'arrêt d'un processus est la survenue d'une erreur fatale. Par exemple, si l'utilisateur saisit la commande :

```
cc foo.c
```


pour compiler le programme `foo.c` et que le fichier correspondant n'existe pas, le compilateur s'arrête purement et simplement. En règle générale, les processus interactifs prenant en charge des interfaces visuelles ne s'interrompent pas si on leur fournit de mauvais paramètres. Ils sont généralement conçus pour afficher une boîte de dialogue demandant à l'utilisateur de recommencer l'opération.

Le troisième motif d'arrêt est la survenue d'une erreur provoquée par le processus, souvent due à un bogue du programme. Par exemple, les erreurs peuvent provenir de l'exécution d'une instruction illégale, d'une référence mémoire inexistante ou d'une division par zéro. Dans certains systèmes, c'est le cas d'UNIX, un processus peut indiquer au système d'exploitation qu'il souhaite prendre lui-même en charge certaines erreurs, auquel cas le processus est signalé (interrompu) au lieu d'être arrêté.

Le quatrième motif d'arrêt d'un processus peut intervenir lorsqu'il exécute un appel système indiquant au système d'exploitation d'arrêter un ou plusieurs autres processus. Sous UNIX, cet appel est `kill`. La fonction Win32 correspondante est `TerminateProcess`. Dans les deux cas, l'appel d'arrêt doit demander l'autorisation nécessaire auprès du processus à arrêter. Sur certains systèmes, lorsqu'un processus s'arrête, que ce soit volontairement ou pas, tous les processus dont il est à l'origine sont immédiatement suspendus. Cependant, ce n'est le cas ni d'UNIX, ni de Windows.

2.1.4 La hiérarchie des processus

Sur certains systèmes, lorsqu'un processus crée un autre processus, les processus parent et enfant continuent d'être associés d'une certaine manière. Le processus enfant peut lui-même créer plusieurs processus, formant une hiérarchie de processus. Remarquez qu'un processus n'a qu'un seul parent. Il peut en revanche avoir zéro, un, deux enfants, ou plus.

Sous UNIX, un processus ainsi que l'ensemble de ses enfants et de ses descendants successifs forment un groupe de processus. Lorsqu'un utilisateur envoie un signal clavier, celui-ci est délivré à tous les membres du groupe de processus associé au clavier. Il s'agit généralement de tous les processus actifs créés dans la fenêtre active. Individuellement, chaque processus peut intercepter le signal, l'ignorer, ou adopter l'action par défaut qui consiste à être arrêté (tué) par le signal.

Pour découvrir un autre exemple dans lequel la hiérarchie des processus joue un rôle, voyons comment UNIX s'initialise lors de son démarrage. Un processus spécial, appelé `init`, est présent dans l'image d'amorçage. Lorsqu'il s'exécute, il lit un fichier indiquant combien de terminaux sont présents. Ensuite, il génère un nouveau processus par terminal. Ces processus attendent une ouverture de session (`login`). Si l'une d'elles réussit, le processus de login exécute un shell pour accepter des commandes. Ces commandes peuvent lancer d'autres processus, et ainsi de suite. Ainsi, tous les processus de l'ensemble du système appartiennent à une arborescence unique, dont `init` est la racine.

À la différence d'UNIX, Windows n'intègre pas le concept de hiérarchie des processus. Tous les processus sont égaux. Le seul moment où l'on trouve une certaine hiérarchie est celui de la création du processus. À ce moment-là, le parent récupère un jeton spécial (appelé **handle**) qu'il peut utiliser pour contrôler le fils. Cependant, il peut passer ce handle à un autre processus, ce qui invalide la hiérarchie. Sous UNIX, les processus ne peuvent pas « déshériter » leurs enfants.

2.1.5 Les états des processus

Bien que chaque processus constitue une entité indépendante, avec son propre compteur ordinal et son état interne, certains processus ont besoin d'entrer en interaction avec d'autres. Un processus peut générer une sortie qu'un autre processus utilisera comme entrée. Dans la commande shell :

```
cat chapter1 chapter2 chapter3 | grep tree
```

le premier processus, qui exécute `cat`, concatène trois fichiers et envoie le résultat vers la sortie standard. Le second processus, qui exécute `grep`, sélectionne toutes les lignes contenant le mot « tree ». Selon les vitesses relatives des deux processus – qui dépendent de la complexité relative des programmes et du temps de traitement que leur consacre le processeur –, il peut arriver que `grep` soit prêt à s'exécuter, mais qu'il n'y ait pas de donnée pour l'alimenter. Il doit donc être bloqué jusqu'à ce qu'une donnée soit disponible.

Un processus se bloque parce que, logiquement, il ne peut poursuivre son exécution. En général, il attend une entrée non encore disponible. Il se peut également qu'un processus conceptuellement prêt à s'exécuter et en mesure de le faire soit interrompu parce que le système d'exploitation a décidé d'allouer le processeur à un autre processus. Il s'agit là de deux circonstances complètement différentes. Dans le premier cas, la suspension est inhérente au problème : la ligne de commande ne peut être traitée tant qu'elle n'a pas été saisie par l'utilisateur. Dans le second cas, c'est un aspect technique du fonctionnement du système qui est en cause : le temps de traitement est insuffisant pour fournir à chaque processus son propre processeur privé. À la figure 2.2, vous pouvez voir un diagramme d'état montrant les trois états que peut prendre un processus :

1. En cours d'exécution (c'est-à-dire utilisant le processeur en cet instant).
2. Prêt (exécutable, temporairement arrêté pour laisser s'exécuter un autre processus).
3. Bloqué (ne pouvant pas s'exécuter tant qu'un événement externe ne se produit pas).

Logiquement, les deux premiers états sont analogues. Dans les deux cas, le processus est prêt à s'exécuter, mais dans le deuxième, le processeur est provisoirement indisponible. Le troisième état est différent des deux autres dans le sens où le processus ne peut pas s'exécuter, même si le processeur n'a rien d'autre à faire.

Comme le montre la figure 2.2, quatre transitions sont possibles entre ces trois états. La transition 1 se produit lorsque le système d'exploitation découvre qu'un processus ne peut pas poursuivre son exécution. Sur certains systèmes, le processus doit effectuer un appel système, comme `block` ou `pause`, pour entrer en état bloqué. Sur d'autres, et c'est le cas d'UNIX, lorsqu'un processus effectue une lecture dans un canal de communication ou dans un fichier spécial (par exemple, un terminal) et qu'il ne trouve pas d'entrée disponible, il est automatiquement bloqué.

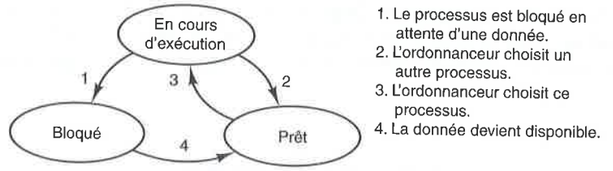

Figure 2.2 • Un processus peut être en cours d'exécution, bloqué ou prêt. Les transitions entre les états apparaissent dans cette figure.

Les transitions 2 et 3 sont provoquées par l'ordonnanceur de processus, dans le cadre du système d'exploitation, sans même que le processus n'en soit informé. La transition 2 se produit lorsque l'ordonnanceur juge que le processus en cours d'exécution a eu suffisamment de temps pour s'exécuter et qu'il est temps de laisser un autre processus bénéficier du temps processeur. La transition 3 a lieu lorsque tous les autres processus ont profité du temps processeur et qu'il est l'heure pour le premier processus de pouvoir s'exécuter. L'ordonnancement consiste à choisir le processus à exécuter et la durée de cette exécution. C'est un point important que nous aborderons plus loin dans ce chapitre. De nombreux algorithmes ont été conçus pour tenter d'équilibrer les demandes concurrentes, afin de garantir l'efficacité du système dans son ensemble, ainsi que l'équité entre les processus. Nous en découvrirons quelques-uns plus loin dans ce chapitre.

La transition 4 est provoquée lorsque l'événement externe attendu par le processus se produit. Il peut s'agir de l'arrivée d'une donnée quelconque. Si aucun autre processus n'est en cours d'exécution à cet instant, la transition 3 est déclenchée et le processus commence à s'exécuter. Dans le cas contraire, le processus doit rester pendant un certain temps en état prêt, jusqu'à ce que le processeur redevienne disponible et que son tour arrive d'être traité.

Grâce au modèle de processus, il devient plus facile de se représenter ce qui se passe à l'intérieur du système. Certains processus exécutent des programmes qui comprennent des commandes saisies par l'utilisateur. D'autres font partie du système et gèrent des tâches telles que le transport de requêtes pour les services de fichiers ou la gestion des détails de l'exécution d'un disque ou d'un lecteur de bandes. Lorsqu'une interruption de disque se produit, le système prend la décision d'arrêter l'exécution du processus en cours et exécute le processus du disque (qui était bloqué dans l'attente

de cette interruption). Ainsi, au lieu de penser en termes d'interruptions, on peut réfléchir en termes de processus utilisateur, de processus disque, de processus terminal, etc., qui se bloquent lorsqu'ils attendent un événement. Lorsque la lecture a eu lieu sur le disque ou que le caractère attendu a été saisi, le processus en attente est débloqué et devient éligible pour reprendre son exécution.

Cette approche est représentée par le modèle de la figure 2.3. On y voit le niveau inférieur du système d'exploitation, où se trouve l'ordonnanceur, avec un ensemble de processus qui reposent sur cette base. Toute la gestion des interruptions, des détails du démarrage et de l'arrêt des processus est confinée à l'intérieur de ce que nous appelons ici l'ordonnanceur, et qui implique bien peu de code. Le reste du système d'exploitation est agréablement structuré sous forme de processus. Sachez cependant que peu de systèmes existants sont aussi « proprement » structurés que celui de cette figure.

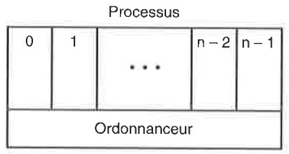

Figure 2.3 • La couche inférieure d'un système d'exploitation structuré en processus gère des interruptions et prend en charge l'ordonnancement. Viennent s'y superposer les processus séquentiels.

2.1.6 L'implémentation des processus

Pour implémenter le modèle de processus, le système d'exploitation maintient une table (un tableau de structures) appelée **table des processus** et contenant une entrée par processus. Certains auteurs appellent également ces entrées des **blocs de contrôle de processus**. Ces entrées contiennent des informations relatives aux éléments suivants : état du processus, compteur ordinal, pointeur de pile, allocation de mémoire, état des fichiers ouverts, ordonnancement, ainsi que toute information relative aux processus devant être enregistrée lorsque le processus bascule de l'état en cours d'exécution vers les états prêt ou bloqué. Ainsi, le processus pourra redémarrer ultérieurement comme s'il n'avait jamais été interrompu.

La figure 2.4 illustre les champs les plus importants que l'on retrouve dans les systèmes classiques. Les champs de la première colonne sont liés à la gestion du processus. Les deux autres concernent respectivement la gestion de la mémoire et celle des fichiers. Il faut noter ici que les champs de la table des processus sont étroitement dépendants du système. Cela dit, cette figure vous donne une idée générale des informations nécessaires.

Maintenant que nous connaissons le contenu type d'une table des processus, il devient plus aisé d'expliquer comment l'illusion de processus séquentiels multiples est maintenue sur un ordinateur ne possédant qu'un seul processeur et plusieurs

périphériques d'E/S. Un emplacement (souvent à un endroit fixe au bas de la mémoire), appelé **vecteur d'interruption**, est associé à chaque classe de périphériques d'E/S : lecteurs de disquettes, disques durs, timers, terminaux. Ce vecteur d'interruption contient l'adresse de la procédure de service d'interruption. Supposons que le processus utilisateur 3 soit en cours d'exécution et qu'une interruption de disque se produise. Le compteur du processus utilisateur 3, le mot d'état du programme et éventuellement d'autres registres sont placés sur une pile (active) par le matériel à l'origine de l'interruption. L'ordinateur passe alors à l'adresse spécifiée dans le vecteur d'interruption du disque. C'en est fini pour la partie matérielle. À partir de là, c'est à la partie logicielle de poursuivre, et plus précisément à la procédure de gestion des interruptions.

Figure 2.4 • Quelques champs d'une entrée type de la table des processus.

Gestion du processus	Gestion de la mémoire	Gestion de fichier
Registres	Pointeur vers un segment de texte	Répertoire racine
Compteur ordinal	Pointeur vers un segment de données	Répertoire de travail
Mot d'état du programme	Pointeur vers un segment de la pile	Descripteurs de fichiers
Pointeur de la pile		ID utilisateur
État du processus		ID du groupe
Priorité		
Paramètres d'ordonnancement		
ID du processus		
Processus parent		
Groupe du processus		
Signaux		
Heure de début du processus		
Temps de traitement utilisé		
Temps de traitement du fils		
Heure de la prochaine alerte		

Toutes les interruptions commencent par sauvegarder les registres, souvent dans une entrée de la table des processus du processus en cours. Ensuite, l'information placée sur la pile par l'interruption est supprimée, et le pointeur de pile est défini de manière à pointer vers une pile temporaire utilisée par le gestionnaire de processus. Des actions telles que la sauvegarde des registres et le positionnement du pointeur de pile

ne peuvent pas être exprimées dans des langages de haut niveau comme le C ; elles sont donc accomplies par une petite routine en langage assembleur. Le plus souvent, la même routine sert à toutes les interruptions dans la mesure où la sauvegarde des registres est toujours une tâche identique, quelle que soit la cause de l'interruption. Lorsque la routine a terminé de s'exécuter, elle appelle une procédure C pour prendre en charge le reste des tâches à accomplir pour ce type spécifique d'interruption. Naturellement, nous supposons ici que le système d'exploitation est écrit en C ; c'est d'ailleurs le cas de tous les systèmes d'exploitation dignes de ce nom. Lorsque l'ordonnanceur a terminé son travail – au cours duquel il pourra très bien avoir fait passer certains processus en état prêt –, il est appelé pour exécuter le prochain processus. Ensuite, le contrôle retombe entre les mains du code assembleur, qui charge les registres et les mappages mémoire pour le processus désormais actif et commence son exécution. La gestion des interruptions et l'ordonnancement sont illustrés à la figure 2.5. Notez bien que certains détails changent d'un système à un autre.

Figure 2.5 • Résumé des tâches que le système d'exploitation accomplit à son niveau le plus bas lorsqu'une interruption a lieu.

1	Le matériel place dans la pile le compteur ordinal, etc.
2	Le matériel charge un nouveau compteur ordinal à partir du vecteur d'interruption
3	La procédure en langage assembleur sauvegarde les registres
4	La procédure en langage assembleur définit une nouvelle pile
5	Le service d'interruption en C s'exécute (généralement pour lire des entrées et les placer dans le tampon)
6	L'ordonnanceur décide du prochain processus à exécuter
7	La procédure C retourne au code assembleur
8	La procédure en langage assembleur démarre le nouveau processus actif

Lorsque le processus se termine, le système d'exploitation affiche une invite (*prompt*) et attend une nouvelle commande. À réception de cette nouvelle commande, il met en mémoire un nouveau programme en écrasant l'ancien.

2.1.7 La modélisation de la multiprogrammation

La multiprogrammation permet d'améliorer l'utilisation de l'UC. Dans une première approche, on suppose que si un processus travaille en moyenne seulement 20 % de son temps lorsqu'il est en mémoire, alors l'UC devrait être occupée constamment avec cinq processus en mémoire. Toutefois, ce modèle est trop optimiste puisqu'il suppose que les cinq processus ne sont pas en attente d'entrée/sortie en même temps.

Appréhender l'utilisation de l'UC d'un point de vue probabiliste constitue une meilleure approche. Supposons qu'un processus passe une fraction p de son temps à

attendre la fin d'une entrée/sortie. Avec n processus minimaux en mémoire, la probabilité que les n processus soient en attente d'entrée/sortie est p^n (et, dans ce cas, l'UC sera inoccupée). Le taux d'utilisation de l'UC est donné par la formule :

$$\text{Taux d'utilisation de l'UC} = 1 - p^n$$

La figure 2.6 montre le taux d'utilisation de l'UC en fonction de n, le **degré de multiprogrammation**.

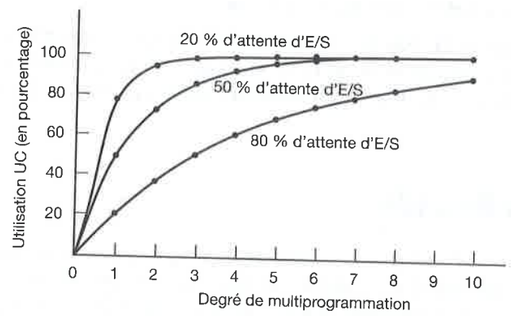

Figure 2.6 • Le taux d'utilisation de l'UC en fonction du nombre de processus en mémoire.

D'après la figure, il est clair que si les processus passent 80 % de leur temps à attendre les entrées/sorties, il faut que 10 processus au moins soient en mémoire simultanément pour que l'utilisation de l'UC soit supérieure à 90 %. Sachant qu'une attente de processus interactif (la saisie d'un utilisateur sur un terminal, par exemple) est un état d'attente d'entrée/sortie, il est évident que des taux de 80 % d'attente et plus ne sont pas inhabituels. Cela s'applique aussi aux systèmes de traitement par lots : les processus qui sollicitent fortement les accès au disque atteindront souvent ce pourcentage d'attente, voire plus.

Pour être tout à fait exact, il faut préciser que le modèle probabiliste que nous venons de voir est approximatif. Il suppose implicitement que tous les n processus sont indépendants, ce qui signifie qu'il serait tout à fait acceptable qu'un système avec cinq processus en mémoire, par exemple, en ait trois en cours d'exécution et deux en attente. Or, avec une seule UC, il est impossible que trois processus s'exécutent en même temps : si l'UC est occupée alors qu'un processus est prêt, ce dernier devra attendre. Par conséquent, les processus ne sont pas indépendants. On peut bien sûr construire un modèle plus précis en suivant la théorie des files d'attente. Toutefois, le modèle qui s'appuie sur la multiprogrammation (qui permet aux processus d'avoir recours à l'UC quand elle est inoccupée) est valide, même si les courbes réelles représentées à la figure 2.6 sont légèrement différentes.

Bien que le modèle de la figure 2.6 soit simplifié, il peut servir à faire des prédictions spécifiques – mais approximatives – sur la performance d'une UC.

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