Systèmes d'Exploitation

Chapitre 1. Généralité

1.1 Définition

Le système d'exploitation peut être défini comme programme système qui commande **l'exécution** de programmes d'application. Il est responsable du chargement et de l'exécution de programmes d'application. Il veille à la disponibilité des configurations matérielles et logicielles requises avant d'exécuter le programme d'application.

1.1 Le système d'exploitation est une machine virtuelle

Le système d'exploitation est une machine virtuelle plus simple à programmer que la machine réelle. Il offre une **interface** de programmation à **l'utilisateur** qui n'a donc pas **besoin** de connaître le **fonctionnement** réel de la **machine** : l'utilisateur demande au système d'effectuer certaines tâches et le système se charge ensuite de dialoguer avec la machine pour réaliser les tâches qui lui sont demandées. Dans cette optique, la machine réelle est en fait cachée sous le système d'exploitation et ce dernier permet donc un dialogue abstrait entre la machine et l'utilisateur. Par exemple, l'utilisateur se contente de dire « je veux écrire mes données dans un fichier sur le disque dur » sans se préoccuper de l'endroit exact où se trouvent ces données, ni de la façon dont il faut s'y prendre pour les copier sur le disque dur.

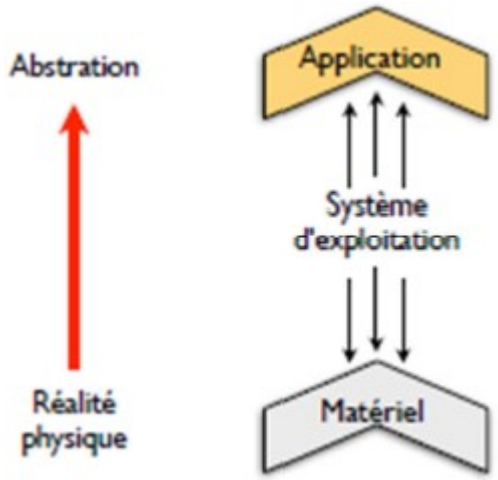

Figure 1 – Le système d'exploitation permet un dialogue abstrait entre la machine et l'utilisateur

Notons qu'à cet effet le système d'exploitation se doit de proposer une interface permettant ce dialogue abstrait et c'est pour cela que des concepts dénués de toute réalité physique sont généralement utilisés lorsqu'on parle des systèmes d'exploitation. Citons par exemple les variables qui permettent de faire référence à des valeurs stockées quelque part en mémoire, les fichiers qui représentent l'abstraction du stockage de données ou les processus qui sont l'abstraction d'un programme en cours d'exécution.

En prenant l'analogie automobile, nous dirions que le système d'exploitation en tant que machine virtuelle est ce qui relie les éléments mécaniques de la voiture au conducteur. Il propose une interface plus ou moins standard (le volant, les pédales d'accélérateur, de frein et

d'embrayage, etc.) qui permet au conducteur de dialoguer avec le moteur et d'obtenir un travail de ce dernier, sans pour autant se soucier du fonctionnement réel de ce moteur.

1.2 Rôle d'un système d'exploitation

Dans un système informatique, le système d'exploitation est le programme qui agit comme intermédiaire entre l'utilisateur et la partie matérielle. Un système informatique possède un ensemble de ressources pour la circulation, le traitement et le stockage de données.
Ainsi, du point de vue machine, un système d'exploitation peut être considéré comme gestionnaire de ressources.

1.2.1 Le système d'exploitation est un gestionnaire de ressources

Les ressources d'une machine étant tous les composants qui sont utilisés pour effectuer un travail et qu'un utilisateur de la machine pourrait s'approprier. À ce titre, tous les périphériques comme la mémoire ou les disques durs sont des ressources. Les registres du processeur ou le temps passé par le processeur à faire des calculs sont aussi des ressources.

Le système d'exploitation est un gestionnaire de ressources, c'est-à-dire qu'il contrôle l'accès à toutes les ressources de la machine, l'attribution de ces ressources aux différents utilisateurs de la machine et la libération de ces ressources quand elles ne sont plus utilisées.
Ce contrôle est capital lorsque le système permet l'exécution de plusieurs programmes en même temps ou l'utilisation de la machine par plusieurs utilisateurs à la fois. En particulier, il doit veiller à ce qu'un utilisateur ne puisse pas effacer les fichiers d'un autre utilisateur ou à ce qu'un programme en cours d'exécution ne détruise pas les données d'un autre programme stockées en mémoire.
Un autre aspect capital du contrôle des ressources est la gestion des **conflits** qui peuvent se produire quand **plusieurs** programmes souhaitent accéder en même temps aux mêmes données. Supposons par exemple qu'un utilisateur exécute deux programmes, chacun d'eux écrivant dans le même fichier. Il y a de grandes chances que, si aucune précaution n'est prise, le résultat contenu dans le fichier ne soit pas celui escompté.
Si nous nous référons à notre analogie automobile, le système d'exploitation en tant que gestionnaire de ressources est représenté par le limiteur de vitesse qui oblige le conducteur à conduire sagement ou par le système d'arrêt automatique en cas d'endormissement qui empêche le conducteur endormi d'entrer en collision avec un mur.

1.2.2 Evolution des systèmes d'exploitation

Les systèmes d'exploitation et l'architecture de l'ordinateur sont historiquement liés. La combinaison de l'architecture de l'ordinateur avec un système d'exploitation est connue sous le nom de plate-forme informatique. Ces changements architecturaux affectent la structure et la performance des systèmes d'exploitation.
Les premiers ordinateurs ne possédaient pas vraiment de système d'exploitation. Le moniteur était chargé avec l'unique programme à exécuter qui se greffait sur ce système rudimentaire tout comme une fonction ou un sous-programme se greffe aujourd'hui sur le programme principal. Les entrées-sorties étaient rudimentaires, réduites à un ruban perforé, les programmes écrits en langage, c'est à dire directement codés en suite de chiffres, exécutés pas à pas et modifiés directement en mémoire au moyen d'un jeu de clés !
Vers 1950 est apparu le moniteur d'enchaînement. Il permettait l'exécution d'une série de travaux en séquence, c'est à dire l'un après l'autre. C'est l'ancêtre du traitement par lot. Il possédait déjà des fonctions de protection: maximum de temps par travail, supervision des périphériques, protection de la zone mémoire utilisée, gestion et abandon des travaux erronés.

Ces avancées étaient fondamentales car elles introduisaient un environnement autour du programme à exécuter.
L'informatique moderne naît dans les années 1960. On peut résumer rapidement ses avancées autour de l'invention des notions suivantes :
- Apparition des processeurs d'entrées-sorties
- Multiprogrammation, c'est à dire possibilité d'exécuter plusieurs programmes simultanément.
- Compilateurs
- Temps partagé
- Mémoire paginée virtuelle. Elle permet de faire fonctionner un ensemble de programmes dont la taille est supérieure à celle de la mémoire physique.
- Les communications

1.4 Mode utilisateur versus mode noyau

Les **processeurs** fonctionnent dans deux modes différents : le mode noyau où toutes les instructions sont autorisées et le mode utilisateur où certaines instructions sont interdites. Le système d'exploitation peut utiliser cette propriété pour faciliter le contrôle qu'il exerce : il s'exécute en mode noyau alors que tous les autres programmes sont exécutés en mode utilisateur. Ces programmes utilisateur ont ainsi par essence des pouvoirs limités et certaines opérations leurs sont interdites.
Par exemple, en n'autorisant l'accès aux différentes ressources de la machine qu'aux programmes s'exécutant en mode noyau, le système d'exploitation protège ces ressources et contraint les programmes utilisateur à faire appel à lui (pas nécessairement de façon explicite) pour accéder aux ressources de la machine.

1.5 Les appels système

Sur les systèmes d'exploitation utilisant le mode noyau, tout programme utilisateur doit faire explicitement appel aux services du système d'exploitation pour accéder à certaines ressources. À ces fins, le système d'exploitation propose une interface de programmation, c'est-à-dire qu'il permet d'accéder à un certain nombre de fonctionnalités qu'il exécutera pour l'utilisateur. Les **appels** système sont **l'interface** proposée par le **système** d'exploitation pour **accéder** aux différentes **ressources** de la **machine**.
Par exemple, il est nécessaire de faire appel au système d'exploitation pour créer un fichier sur le disque dur et cela via un appel système. Si nous demandons que ce fichier soit créé dans un répertoire qui nous est interdit d'accès, par exemple un répertoire appartenant à un autre utilisateur, l'appel système va refuser de créer le fichier.

1.6 Structure d'un système d'exploitation

Le système d'exploitation d'une machine n'est en pratique pas constitué d'un seul programme. Il se compose d'au moins deux parties que l'on nomme souvent le noyau (kernel en anglais), qui est le coeur du système (core en anglais), et les programmes système. Le noyau est exécuté en mode noyau et il se peut qu'il corresponde à plusieurs processus différents.
Même si le noyau effectue l'essentiel des tâches du système d'exploitation, ce dernier ne peut se réduire à son noyau : la plupart des services qu'un utilisateur utilise sur une machine sont en fait des sur-couches du noyau. Les programmes système qui assurent ces services sont néanmoins exécutés en mode utilisateur et, donc, ils doivent faire appel au noyau comme n'importe quel utilisateur pour accéder aux ressources.

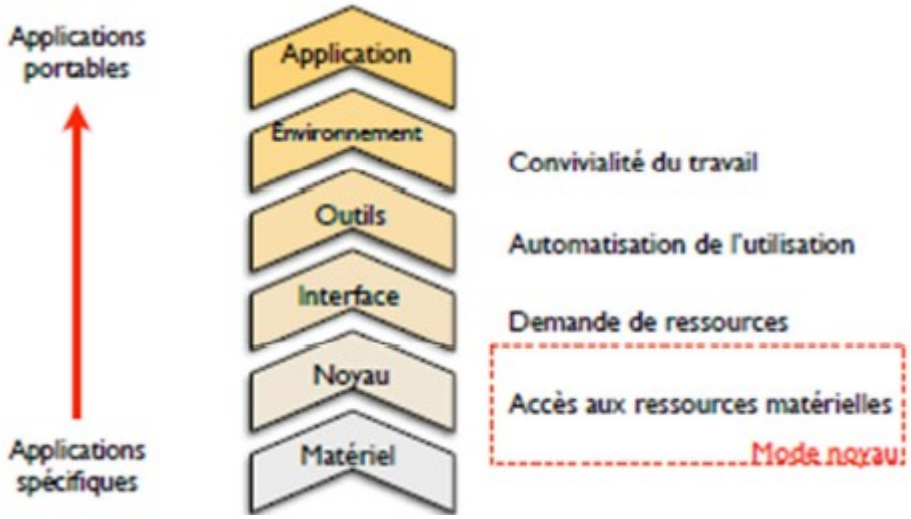

Figure 2 – Représentation en couches des systèmes d'exploitation.

1.7 Les différents types de systèmes d'exploitation
Nous présentons les différentes structures des systèmes d'exploitation.

1.7.1 Système monolithique
Le système d'exploitation est une structure hiérarchique écrit comme un ensemble de procédures dont chacune peut appeler n'importe quelle autre en cas de besoin. La structure de base suggérée par cette organisation est la suivante :
- un programme principal qui invoque la procédure de service ;
- un ensemble de procédures de services qui gèrent les appels système ;
- un ensemble de procédures utilitaires auxiliaires des procédures.

1.7.2 Les systèmes en couche
L'organisation d'un système en une série de couches est une généralisation de la structure précédente. Le système était composé par une série de couches hiérarchiques.
- La couche 0 fournissait le service de multiprogrammation CPU.
- La couche 1 était en charge de la gestion de la mémoire et des tampons
- La couche 2 se chargeait de la communication interprocessus et console
- La couche 3 était en charge des entrées sorties
- La couche 4 contenait les programmes utilisateurs
- La couche 5 était le processus opérateur.

Chaque couche masque ainsi les couches situées en dessous d'elle et oblige la couche située au-dessus à utiliser l'interface qu'elle propose. L'avantage est évident : les couches sont totalement indépendantes les unes des autres et seule l'interface entre chaque couche compte.
Cela permet de développer, de débuger et tester chaque couche en se fondant sur les couches inférieures qui sont sûres.
Même si ce principe est séduisant, il est très complexe à mettre en œuvre et les systèmes ainsi structurés sont souvent très lourds et peu performants.

1.7.3 Machines virtuelles
La plus part du temps, les utilisateurs des ordinateurs sont assis devant leur PC. Cependant, dans d'autres cas, les utilisateurs sont en face d'un terminal connecté à un mainframe ou ordinateur connecté à d'autres ordinateurs en réseau.

Dans le premier cas, le système est conçu de telle sorte qu'un seul utilisateur monopolise toutes les ressources. Ainsi l'objectif de tels systèmes est l'efficacité dans l'accomplissement des tâches soumises par l'utilisateur. Ces systèmes sont donc conçus de telle sorte qu'elles soient faciles à utiliser avec pour seule considération les performances du système, sans se soucier de l'utilisation des ressources. Dans le second cas (utilisation d'un terminal), la même ressource peut être demandée par plusieurs utilisateurs chacun assis devant son propre terminal.
Dans de tels cas, la conception de ces systèmes d'exploitation met un accent particulier sur l'accès aux ressources. Ils doivent garantir la disponibilité de ressources et un partage équitable entre les requérants.
En dernier lieu, on distingue de systèmes où l'utilisateur possède des ressources dédiées (utilisation d'un ordinateur personnel) mais comportant des ressources partagées (serveur de fichiers, serveur d'impression, etc.).
Dans ce cas, la conception de tels systèmes d'exploitation doit chercher un compromis entre la gestion des ressources et l'utilisation individuelle de la machine.

1.7.4 Les micro-noyaux

Nous avons vu que les noyaux des systèmes monolithiques ont tendance à être volumineux. Par ailleurs, si un service particulier est très rarement utilisé, il doit néanmoins être présent dans le programme du noyau et il utilisera donc de la mémoire de façon inutile. Pour éviter ces problèmes, une solution consiste à réduire le noyau à quelques procédures essentielles et à reporter tous les autres services dans des programmes système. Le noyau ainsi réduit s'appelle souvent micro-noyau.
Ce travail est néanmoins considérable car il suppose une refonte totale des systèmes d'exploitation habituels : ceux-ci ayant été développés au fil des ans, il se peut que des services très importants, mais apparus tardivement, soient implantés à la périphérie du noyau et que, donc, l'isolement de ces services demande la réécriture complète du noyau.
Un autre inconvénient des micro-noyaux réside dans la communication entre les différents services. Cette partie fondamentale permet le dialogue entre le noyau et les services (tels que le réseau, l'accès aux disques. Chaque service étant cloisonné dans son espace d'adresses, il est impératif d'établir ce mécanisme de communication entre le micro noyau et les services.
Cette couche de dialogue, qui n'existe pas dans un noyau monolithique, ralentit les opérations et conduit à une baisse de performance.

1.7.5 Le modèle client-serveur

Il consiste à implémenter la plus grande partie du système d'exploitation sous forme de processus utilisateur. La demande d'un service passe par l'envoie du service demandeur (le client) d'une requête au service pourvoyeur (serveur) qui rend le service et renvoie le résultat.

1.8 Les services des systèmes d'exploitation

Les différents travaux que doit effectuer un système d'exploitation moderne sont généralement nommés « services ». La plupart de ces travaux sont pris en charge par le noyau du système d'exploitation. Nous retrouvons :
- La gestion des processus : La gestion des processus n'a de sens que sur les machines fonctionnant en temps partagé. Elle comprend la création et la destruction dynamique de processus.
- La gestion de la mémoire : Afin de simplifier la gestion des processus, les systèmes d'exploitation modernes travaillent dans un espace mémoire virtuel, c'est-à-dire avec des adresses virtuelles qui doivent être traduites pour correspondre à des adresses physiques.

Cela permet d'allouer à chaque processus (y compris au noyau) son propre espace mémoire de telle sorte qu'il a l'illusion d'avoir la machine pour lui tout seul.
- Le système de fichiers : Le système de fichiers est un élément essentiel des systèmes d'exploitation moderne : il permet d'accéder aux différents périphériques et il propose une interface abstraite pour manipuler des données.
- La gestion des entrées / sorties : Les entrées / sorties permettent de faire transiter des données par l'ordinateur et d'utiliser ces données pour faire des calculs. Ces données peuvent provenir de périphériques, de processus présents sur la machine ou de processus présents sur d'autres machines (via un réseau).
- La gestion des communications entre machines : Il est aujourd'hui impensable de disposer d'ordinateurs à usage professionnel sans que ceux-ci soient reliés entre eux par un réseau local. Par ailleurs, l'utilisation de réseaux internationaux comme l'Internet se répand et le système d'exploitation doit donc prendre en charge la gestion des communications par réseaux.

Chapitre 2. Les processus

2.1 La gestion des processus

Un processus est l'abstraction d'un programme en cours d'exécution. Le concept de processus est capital pour la réalisation d'un système d'exploitation.

La notion de processus n'a cependant de sens que pour les systèmes d'exploitation fonctionnant en temps partagé ou en multiprogrammation. Si un système d'exploitation ne peut pas commuter d'un processus A à un processus B avant que le processus A ne se termine, il est inutile de concevoir une gestion complexe des processus : il suffit de les exécuter dans l'ordre. Un système d'exploitation digne de ce nom doit fonctionner en temps partagé !
Les systèmes qui sont de plus multi-utilisateurs doivent offrir une gestion de processus plus élaborée, qui comprend notamment la protection des processus d'un utilisateur contre les actions d'un autre utilisateur.

2.1.1 Le partage du temps

Supposons que trois programmes doivent être exécutés sur notre ordinateur. À un instant donné t, seul un des processus représentant ces programmes disposera du processeur pour exécuter les instructions de son programme. Pour faciliter l'explication, nous nommerons A, B et C les trois processus représentant l'exécution des trois programmes.
Supposons alors que nous découpions le temps en petites unités de l'ordre de la milliseconde (souvent appelées quanta) et que nous attribuions une unité de temps à chaque processus.
Pendant le laps de temps qui lui est accordé, **A** pourra exécuter un certain nombre d'instructions. Une fois ce laps de temps écoulé, A sera interrompu et il sera nécessaire de sauvegarder toutes les informations dont A a besoin pour reprendre son exécution (son contexte), en particulier le compteur d'instructions de A.
Un autre processus peut alors utiliser le processeur pour exécuter les instructions de son programme. Quel processus choisir ? A, B ou C ? Ce choix est assuré par le système d'exploitation et la façon dont les processus sont choisis s'appelle l'ordonnancement ou le scheduling. On parle aussi parfois de scheduler pour désigner la partie du système d'exploitation qui prend en charge l'ordonnancement.

Reprenons notre ordonnancement : A vient d'être interrompu. Supposons que B soit choisi. Tout d'abord, il faut rétablir le contexte de B, c'est-à-dire remettre le système dans l'état où il était au moment où B a été interrompu. L'opération consistant à sauvegarder le contexte d'un processus, puis à rétablir le contexte du processus suivant s'appelle le changement de contexte (context switching) et est essentiellement assurée par les couches matérielles de l'ordinateur.
Après rétablissement de son contexte, le processus B va disposer du processeur pour un quantum et pourra exécuter un certain nombre d'instructions de son programme.
Ensuite, B sera interrompu et un autre processus (éventuellement le même) se verra attribuer le processeur. Ce principe est résumé sur la figure 4

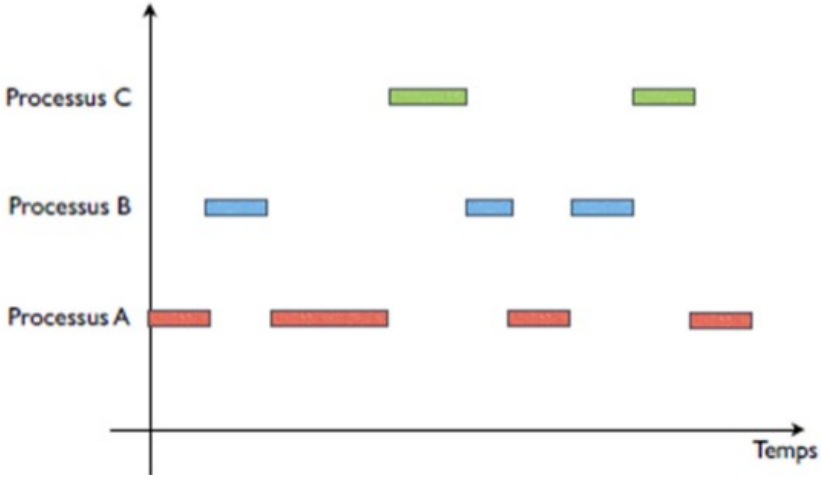

Figure 3 – Principe de l'ordonnancement des processus : chaque processus dispose du processeur pendant un certain temps, mais, à chaque instant, un seul processus peut s'exécuter.

Ainsi, à tout instant, un seul processus utilise le processeur et un seul programme s'exécute. Mais si nous observons le phénomène à l'échelle humaine, c'est-à-dire pour des durées de l'ordre de la seconde, nous avons l'illusion que les trois programmes correspondant aux trois processus A, B et C s'exécutent en même temps. Cette exécution est en fait concurrente, car les unités de temps attribuées à un processus ne servent pas aux autres.

Dans le cas de systèmes à temps partagé, tous les processus progressent dans le temps, mais un seul s'exécute à la fois. Lorsque le processeur passe d'un processus à un autre, la vitesse de traitement de processus n'est pas uniforme, ni même reproductible si le même processus s'exécutait une autre fois.

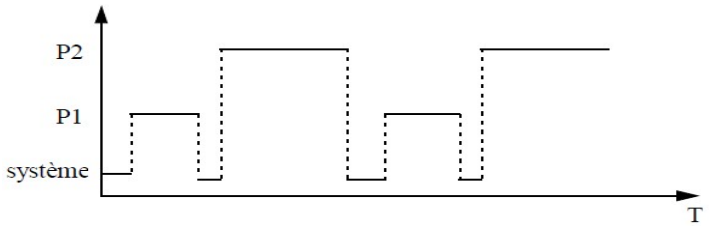

Figure 4 Le multitâche

Ainsi la commutation des processus (tâches), le passage d'un processus à une autre, est réalisée par un ordonnanceur au niveau le plus bas du système. Cet ordonnanceur est activé par des interruptions d'horloge, de disque, de terminaux.

À chaque interruption correspond un vecteur d'interruption, un emplacement mémoire, contenant une adresse. L'arrivée de l'interruption provoque le branchement à cette adresse. Une interruption entraîne, en général, les opérations suivantes :

- Empilement du compteur ordinal, du registre d'état et peut être d'autres registres. La procédure logicielle d'interruption sauvegarde les éléments de cette pile et les autres registres dans du contexte du processus en cours;
- Mise en place d'une nouvelle pile permettant le traitement desinterruptions;
- Appel de l'ordonnanceur;
- Élection d'un nouveau programme;

Les processus sont composés d'un espace de travail en mémoire formé de 3 segments : la pile, les données et le code et d'un contexte.

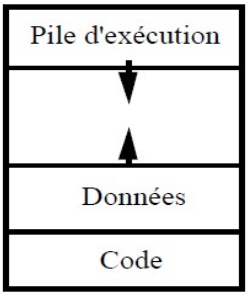

Figure 5 Les segments d'un processus.

Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter. La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire. Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile. Les zones de pile et de **données** ont des frontières mobiles qui croissent en sens **inverse** lors de l'exécution du programme. Parfois, on partage la zone de données en données elles-mêmes et en tas. Le tas est alors réservé aux données dynamiques.

2.1.2 La hiérarchie des processus

Les systèmes d'exploitation modernes permettent la création et la suppression dynamique des processus .

2.1.2.1 Une arborescence de processus

Toutes les stratégies employées sont fondées sur le même principe : tout processus est créé par un autre processus et il s'établit donc une filiation de processus en processus.
Généralement, on appelle processus père le processus qui en crée un autre et processus fils le processus ainsi créé.
Comme chaque processus fils n'a qu'un seul père, cette hiérarchie des processus peut se traduire sous forme d'arbre. En revanche, comme un processus père peut avoir plusieurs fils, le nombre de branches en chaque nœud de l'arbre n'est pas déterminé.

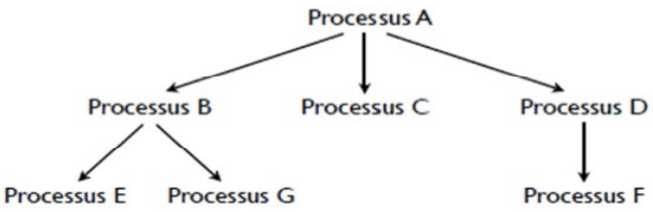

Figure 6 – Arborescence des processus.

2.1.2.2 Le premier processus

La création de processus par filiation ne peut en effet pas s'appliquer au premier processus créé : c'est l'histoire de l'œuf et de la poule ! En fait, le premier processus est créé au moment où l'ordinateur démarre et il fait généralement partie des processus représentant le noyau du système d'exploitation.
Ce premier processus est exécuté en mode noyau et effectue un certain nombre de tâches vitales. Puis il crée un certain nombre de processus, exécutés en mode noyau, qui assurent les services fondamentaux du noyau. Ces processus peuvent alors à leur tour créer des processus système, exécutés en mode utilisateur, qui assureront d'autres services du système d'exploitation ou qui permettront aux utilisateurs d'utiliser la machine.
Le nombre de processus fonctionnant en mode noyau varie d'un système d'exploitation à l'autre.

2.1.2.2 Le démarrage de la machine

La façon dont le premier processus est créé dépend beaucoup des systèmes d'exploitation et des machines. Généralement, les ordinateurs possèdent une mémoire programmable non volatile appelée PROM (Programmable Read Only Memory). Dans cette PROM est stocké un programme qui est directement chargé en mémoire, puis exécuté au moment où l'ordinateur est allumé. L'opération consistant à démarrer l'exécution de ce programme s'appelle le bootstrapping.

Les PROM ont généralement des petites capacités et il est nécessaire de faire appel au constructeur de l'ordinateur pour les reprogrammer. Le système d'exploitation ne peut donc pas être stocké dans la PROM de la machine et il se trouve généralement sur le disque dur.
Lorsque la machine a démarré, le processeur sait comment accéder au disque dur, mais il ne sait pas comment est structuré le système de fichiers et il n'est donc pas capable de lire le fichier exécutable correspondant au système d'exploitation. Sur de nombreuses machines Unix, cet exécutable s'appelle vmunix et se trouve très près du sommet de l'arborescence du système de fichiers.
Le programme contenu dans la PROM se contente donc généralement de lancer l'exécution d'un autre programme, stocké au tout début du disque dur sous une forme ne dépendant pas du système de fichiers. Ce programme, souvent appelé le programme de boot (amorce en français), peut en revanche contenir l'adresse physique absolue de l'endroit où est stocké le fichier vmunix : comme le programme de boot est aussi stocké sur disque dur, il est possible de le modifier chaque fois que le fichier vmunix est modifié (que ce soit sa place sur le disque dur ou son contenu). → update le SE

Dans la mesure où le système d'exploitation n'est pas encore exécuté, les programmes de la PROM et de boot n'utilisent pas les facilités habituellement proposées par les systèmes d'exploitation, comme la mémoire virtuelle, et ils doivent donc assurer la gestion de la mémoire. En particulier, il est nécessaire de prendre de grandes précautions lorsque le programme de boot charge en mémoire le système d'exploitation : si jamais il ne le trouve pas ou si jamais ça se passe mal, la machine reste dans un état où rien ne fonctionne !
Pour éviter ce genre de problèmes, le programme de boot et le système d'exploitation peuvent se trouver sur une clé USB ou un CDROM et ils sont d'ailleurs généralement d'abord cherchés sur ces supports avant d'être cherchés sur le disque dur.
Malheureusement, cette méthode offre souvent la possibilité de contourner les protections des ordinateurs en plaçant tout simplement un système d'exploitation très permissif sur une clé USB. Ce système sera alors lu au démarrage de la machine et rien n'empêchera alors d'accéder à toutes les données placées sur le disque dur.

Pour éviter ce type de piraterie, il est fréquent que l'ordinateur propose une protection matérielle via un mot de passe : ce dernier est directement stocké dans la PROM et ne peut donc être modifié aussi facilement. Notons cependant que si un pirate peut accéder à une machine pour lui faire lire une clé USB, rien ne l'empêche d'ouvrir la machine pour tout simplement enlever le disque dur et le lire ensuite tranquillement chez lui...
Certains systèmes ont des programmes de boot qui permettent de choisir un système d'exploitation parmi plusieurs. Par exemple, il est possible d'installer Linux sur une machine ainsi que d'autres systèmes d'exploitation comme Windows. Le programme de boot de Linux charge par défaut Linux, mais donne la possibilité à l'utilisateur de lancer l'exécution des autres systèmes d'exploitation. Il est donc tout à fait possible d'utiliser une machine avec plusieurs systèmes d'exploitation sélectionnés de façon dynamique au moment du démarrage de la machine.

2.2 Les états d'un processus

Rappelons qu'un processus est une instance d'un programme en exécution mais pointant le fait qu'un programme n'est pas à lui-même un processus, c'est une entité passive. Un processus est une entité active (dynamique) dont on peut étudier l'état au cours du temps. Un processus peut-être dans trois états différents : élu, prêt, bloqué.
- Un processus est dit élu s'il est en cours d'exécution sur le processeur. Dans le cas d'une machine multiprocesseurs plusieurs processus peuvent être élus en même temps.
Notons cependant que le nombre de processus élu doit être **égal au nombre de processeurs**.
Lorsque le quantum de temps attribué au processus élu est épuisé, l'ordonnanceur est appelé de façon asynchrone par interruption et élit un autre processus parmi les processus prêt. Le processus qui était en cours d'exécution passe de l'état élu à l'état prêt.
- Un processus est dit prêt s'il est suspendu en faveur d'un autre. Pour un processus prêt, il ne lui manque que la source processeur pour s'exécuter.
- Lorsqu'un processus en exécution attend un événement externe (décompte d'horloge, attente des données, etc.), il s'en dort et passe dans l'état bloqué . L'ordonnanceur appelé de façon explicite par ledit processus, élit un autre processus parmi les processus prêt.
Lorsque l'événement externe attendu se produit le système d'exploitation se charge de son traitement, puisque le processus se trouve dans un état bloqué. Pour ce faire, le système d'exploitation interrompt temporairement le processus en exécution pour traiter les données reçues et faire passer le processus bloqué à l'état prêt.
Le schéma ci-dessous matérialise les différentes transitions.

Nous avons vu maintenant qu'un processus pouvait être actif en mémoire centrale (Élu) ou suspendu en attente d'exécution (Prêt). Il peut aussi être Bloqué, en attente de ressource, par exemple au cours d'une lecture de disque. Le diagramme simplifié des états d'un processus est donc :

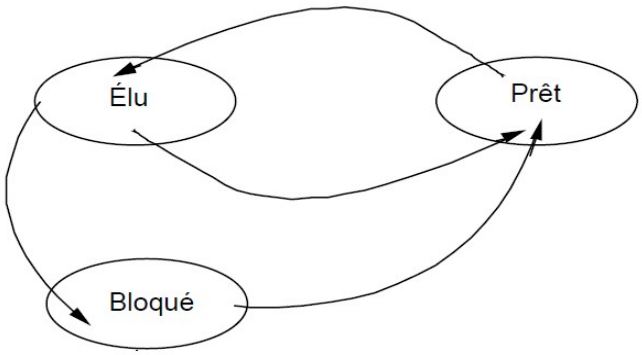

Figure 7 Les états d'un processus.

Le processus passe de l'état élu à l'état prêt et réciproquement au cours d'une intervention de l'ordonnanceur. Cet ordonnanceur se déclenchant, par exemple, sur une interruption d'horloge. Il pourra alors suspendre le processus en cours, s'il a dépassé son quantum de temps, pour élire l'un des processus prêts. La transition de l'état élu à l'état bloqué se produit, par exemple, à l'occasion d'une lecture sur disque. Ce processus passera de l'état bloqué à l'état prêt lors de la réponse du disque. Ce passage correspond d'une manière générale à la libération d'une ressource.

En fait, sous Unix, l'exécution d'un processus se fait sous deux modes, le mode utilisateur et le mode noyau. Le mode noyau correspond aux appels au code du noyau : write, read, … Le mode utilisateur correspond aux autres instructions.

Un processus en mode noyau ne peut être suspendu par l'ordonnanceur. Il passe à l'état préempté à la fin de son exécution dans ce mode – à moins d'être bloqué ou détruit –. Cet état est virtuellement le même que prêt en mémoire.

Par ailleurs, lorsque la mémoire ne peut contenir tous les processus prêts, un certain nombre d'entre eux sont déplacés sur le disque. Un processus peut alors être prêt en mémoire ou prêt sur le disque. De la même manière, on a des processus bloqués en mémoire ou bien bloqués sur disque.

Enfin, les processus terminés ne sont pas immédiatement éliminés de la tables des processus – ils ne le sont que par une instruction explicite de leur père –. Ceci correspond à l'état **défunt**.

2.2.1 Les processus sous Unix

Les fonctions fondamentales

Un processus peut se dupliquer – et donc ajouter un nouveau processus – par la fonction :

```
pid_t fork(void)
```

qui rend -1 en cas d'échec. En cas de réussite, la fonction retourne 0 dans le processus fils et le n° du processus fils – *Process Identifier* ou PID – dans le père. Cette fonction transmet une partie du contexte du père au fils : les descripteurs des fichiers standards et des autres fichiers, les redirections… Les deux programmes sont sensibles aux mêmes interruptions. À l'issue d'un fork()les deux processus s'exécutent simultanément.

La création d'un nouveau processus ne peut se faire que par le recouvrement du code d'un processus existant grâce à la fonction :

```
int execl(char *ref, char *arg0, char *argn, 0)
```

`ref` est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter. `arg0`, `arg1`, …, `argn` sont les arguments du programme. Le premier argument, `arg0`, reprend en fait le nom du programme.

Les fonctions `execle()` et `execlp()` ont des arguments et des effets semblables. Elles transmettent en plus respectivement la variable chemin (`PATH`) ou les variables d'environnement au complet.

Les fonctions `int execv(char *ref, char *argv[])` ainsi que `execve()` et `execvp()` sont des variantes analogues. Les paramètres du programme à lancer sont transmis dans le tableau de chaînes : `argv[][]`. Ce tableau est semblable aux paramètres de la fonction principale d'un programme C : `main(int argc, char **argv)`.

Ces fonctions rendent `-1` en cas d'échec.

Par exemple :

```
void main()
{
   execl("/bin/ls", "ls", "-l", (char *) 0);
/* avec execlp, le premier argument
peut n'être que ls et non /bin/ls */
   printf("pas d'impression\n");
}
```

La création d'un nouveau processus – l'occupation d'un nouvel emplacement de la table des processus – implique donc la duplication d'un processus existant. Ce dernier sera le père. On substituera ensuite le code du fils par le code qu'on désire exécuter. Au départ, le processus initial lance tous les processus utiles au système.

Un processus se termine lorsqu'il n'a plus d'instructions ou lorsqu'il exécute la fonction :

```
void exit(int statut)
```

L'élimination d'un processus terminé de la table ne peut se faire que par son père, grâce à la fonction :

```
int wait(int *code_de_sortie)
```

Avec cette instruction, le père se bloque en attente de la fin d'un fils. Elle rendra le n° PID du premier fils mort trouvé. La valeur du code de sortie est reliée au paramètre d'`exit` de ce fils. On peut donc utiliser l'instruction `wait` pour connaître la valeur éventuelle de retour, fournie par `exit()`, d'un processus. Ce mode d'utilisation est analogue à celui d'une fonction. `wait()` rend `-1` en cas d'erreur.

Si le fils se termine sans que son père l'attende, le fils passe à l'état `defunct` dans la table. Si, au contraire, le père se termine avant le fils, ce dernier est rattaché au processus initial. Le processus initial passe la plupart de son temps à attendre les processus orphelins.

Grâce aux 3 instructions, `fork()`, `exec()`, et `wait()`, on peut écrire un interprète de commandes simplifié. Il prend la forme suivante :

```
while(1) {
                                   /* attend commande */
    lire_commande(commande,
    paramètres); if (fork() !=
    0) {
          wait(&statut);                /* proc. père */
    } else {
          execv(commande, paramètres); /* proc. fils */
    }
}
```

Les fonctions Unix rendent `-1` en cas d'erreur. Dans ce cas, elles positionnent la variable entière globale `errno`. On peut imprimer cette valeur d'erreur et constater la cause qui est décrite dans le fichier `errno.h`. On peut aussi imprimer le texte en clair avec la fonction `perror(char*)`. Dans le cas d'une programmation professionnelle, il est indispensable de vérifier la valeur de retour de toutes les fonctions faisant appel au système et de se protéger des erreurs par un codage du type :

```
if ((retour = fonction()) == -1){
    perror("plantage");
    exit(0);
 }
```

2.3 Les signaux

Par défaut, un signal provoque la destruction du processus récepteur, à condition bien sûr, que le processus émetteur possède ce droit de destruction.

L'instruction :
`int kill(int pid, int signal)`

permet à un processus d'envoyer un signal à un autre processus. `pid` est le n° du processus à détruire et `signal`, le n° du signal employé. La fonction `kill` correspond à des interruptions logicielles.

La liste des valeurs de `signal` comprend, entre autres :

Nom du signal	N° du signal	Commentaires
SIGUP	1	signal émis lors d'une déconnexion
SIGINT	2	^C
SIGQUIT	3	^\
SIGKILL	9	signal d'interruption radicale
SIGALRM	14	signal émis par `alarm(int sec)` au bout de sec secondes
SIGCLD	20	signal émis par un fils qui se termine, à son père

Tableau 1 Quelques signaux d'Unix.

Les valeurs des signaux dépendent de l'implantation. La liste complète est définie par des macros dans `signal.h`
`kill()` rend `0` en cas de succès et `-1` en cas d'échec.

Un processus peut détourner les signaux reçus et modifier son comportement par l'appel de la fonction :
`void (*signal(int signal, void (*fonction)(int)))`
avec `fonction` pouvant prendre les valeurs:

Nom	Action
`SIG_IGN`	le processus ignorera l'interruption correspondante
`SIG_DFL`	le processus rétablira son comportement par défaut lors de l'arrivée de l'interruption (la terminaison)
`void fonction(int n)`	Le processus exécutera `fonction`, définie par l'utilisateur, à l'arrivée de l'interruption `n`. Il reprendra ensuite au point où il a été interrompu

Tableau 2 Les routines d'interruption d'Unix.

L'appel de la fonction `signal` positionne l'état des bits de signaux dans la table des processus.

Par ailleurs, la fonction `signal(SIGCLD, SIG_IGN)`, permet à un père d'ignorer le retour de ses fils sans que ces derniers encombrent la table des processus à l'état `defunct`.

Quelques fonctions d'identification d'Unix

La commande Unix `ps -ef`, donne la liste des processus en activité sur le système. Chaque processus possède un identificateur et un groupe. On les obtient par les fonctions :

```
int getpid(void)
```

et

```
int getpgrp(void)
```

Les fils héritent du groupe de processus du père. On peut changer ce groupe par la fonction :

```
int setpgrp(void)
```

Chaque processus possède, d'autre part, un utilisateur réel et effectif. On les obtient respectivement par les fonctions :

```
int getuid(void)
```

et

```
int geteuid(void)
```

L'utilisateur réel est l'utilisateur ayant lancé le processus. Le numéro d'utilisateur effectif sera utilisé pour vérifier certains droits d'accès (fichiers et envoi de signaux). Il correspond normalement au numéro d'utilisateur réel. On peut cependant positionner l'utilisateur effectif d'un processus au propriétaire du fichier exécutable. Ce fichier s'exécutera alors avec les droits du propriétaire et non avec les droits de celui qui l'a lancé. La fonction :

```
int setuid(int euid)
```

permet de commuter ces numéros d'utilisateur de l'un à l'autre : réel à effectif et vice-versa. Elle rend `0` en cas de succès.

2.4 Les fils d'exécution

Les interfaces graphiques, les systèmes multi-processeurs ainsi que les systèmes répartis ont donné lieu à une révision de la conception usuelle des processus. Cette révision se fonde sur la notion de fils d'exécution (*thread of control*). Un processus classique est composé d'un espace d'adressage et d'un seul fil de commande. L'idée est d'associer plusieurs fils d'exécution à un espace d'adressage et à un processus. Les fils d'exécution sont donc des processus dégradés à l'intérieur d'un processus ou d'une application. On les appelle aussi des processus poids plume ou poids léger.

2.4.1 Pourquoi des fils d'exécution

Certaines applications se dupliquent entièrement au cours du traitement. C'est notamment le cas pour des systèmes client-serveur, où le serveur exécute un `fork`, pour traiter chacun de ses clients. Cette duplication est souvent très coûteuse. Avec des fils d'exécution, on peut arriver au même résultat sans gaspillage d'espace, en ne créant qu'un fil de contrôle pour un nouveau client, et en conservant le même espace d'adressage, de code et de données. Les fils d'exécution sont, par ailleurs, très bien adaptés au parallélisme. Ils peuvent s'exécuter simultanément sur des machines multiprocesseurs

De quoi sont constitués les fils d'exécution

Éléments d'un fil de commande	Éléments d'un processus ordinaire
Compteur de programme	Espace d'adressage
Pile	Variables globales
Registres	Table des fichiers ouverts
Fils de commande fils	Processus fils
Statut	Compteurs
	Sémaphores

Tableau 3 Le contexte comparé d'un fil de commande et d'un processus.

Les éléments d'un processus sont communs à tous les fils d'exécution de ce processus.

2.4.2 Utilisation des fils d'exécution

L'avantage essentiel des fils d'exécution est de permettre le parallélisme de traitement en même temps que les appels système bloquants. Ceci impose une coordination particulière des fils d'exécution d'un processus. Dans une utilisation optimale, chaque fil de commande dispose d'un processeur. A. Tanenbaum, dans *Modern Operating Systems*, distingue trois modèles d'organisations possibles : le modèle répartiteur / ouvrier (*dispatcher / worker*), l'équipe et le pipeline.

2.4.2.1 Le modèle répartiteur/ouvrier

Dans ce modèle, les demandes de requêtes à un serveur arrivent dans une boîte aux lettres. Le fil de commande répartiteur a pour fonction de lire ces requêtes et de réveiller un fil de commande ouvrier. Le fil de commande ouvrier gère la requête et partage avec les autres fils d'exécution, la mémoire cache du serveur.

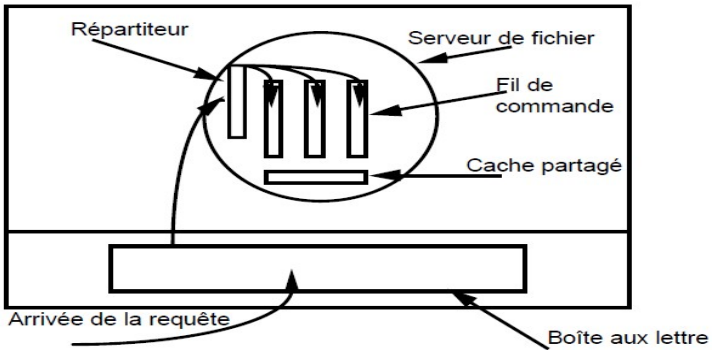

Figure 9 Le répartiteur

2.4.2.2 L'équipe

Dans le modèle de l'équipe, chaque fil de commande traite une requête. Il n'y a pas de répartiteur. À la place, on met en implante une file de travaux en attente.

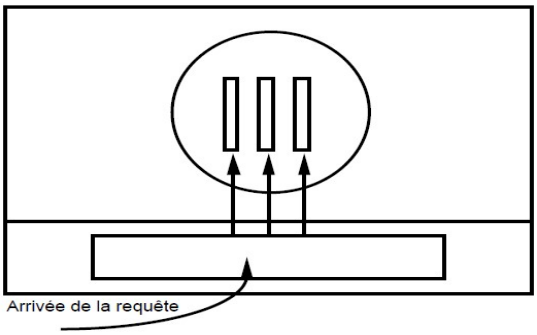

Figure 11 L'équipe

2.4.2.3 Le pipeline

Dans le modèle du pipeline, chaque fil de commande traite une partie de la requête. Plusieurs architectures matérielles implantent ce type d'organisation. Ce modèle n'est cependant pas approprié pour toutes les applications.

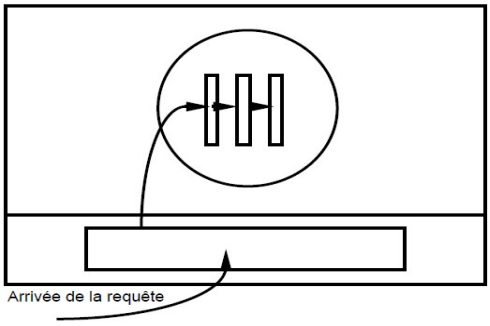

Figure 12 Le pipeline

2.4.3 L'implantation des fils d'exécution sous Windows NT

Les fils d'exécution peuvent être implantés dans l'espace utilisateur ou dans le noyau. Actuellement, peu de systèmes d'exploitation incluent des fils d'exécution dans leur noyau. Plusieurs bibliothèques externes existent, par exemple les processus poids légers de Sun. Ce n'est pas le cas de Windows NT qui a été conçu d'emblée avec des fils d'exécution.

Windows NT peut se programmer à deux niveaux : en C ou bien en C++. La programmation en C se fait par les API du *Software Development Kit*. La programmation en C++ se fait par l'intermédiaire des *Microsoft Foundation Classes* qui encapsulent les entités du système.

Un processus sous Windows est une instance d'application qui possède un espace d'adressage, de la mémoire, des fichiers et des fenêtres ouvertes. On crée un nouveau processus par la fonction CreateProcess(). Ce processus comprendra un fil de commande. On peut lui en ajouter d'autres avec la fonction :

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa, DWORD cbStack,
LPTHREAD_START_ROUTINES, lpSStartAddr, LPVOID lpThreadParm,
DWORD fdwCreate, LPDWORD lpIDThread).
```

Un fil de commande peut s'assurer l'exclusivité d'une ressource par la fonction `WaitForSingleObject()` qui correspond à un P de sémaphore d'exclusion mutuelle. On libère le sémaphore par un `ReleaseMutex()`. Les fonctions `InterLockedIncrement()` et `InterLockedDecrement()` sont applicables à des sémaphores généralisés.

2.5 Communication interprocessus

Il arrive que les processus aient besoin de communiquer entre eux par exemple dans pipeline du shell, lorsque la sortie d'un processus doit être passée à un autre processus, etc. Il est donc préférable que cette communication se fasse de manière structurée. Au cours de cette activité nous allons aborder quelques-uns des problèmes liés à la communication interprocessus.

2.5.1 Les notions de base de communication interprocessus

Les processus d'un système ne s'exécutent pas tous de manière isolée. Certains se trouvent en compétition pour une ressource donnée, d'autres ont besoin de coopérer et nécessitent par conséquent des moyens de communication et de synchronisation. Ainsi, les principaux problèmes liés à la communication interprocessus sont les suivants:

· Comment un processus fait-il pour passer des informations à un autre processus?

· Comment éviter que deux processus ou plus ne produisent pas de conflits lorsqu'ils s'engagent dans des activités critiques (tentative de récupération du dernier Mo de mémoire par deux processus?

· Comment faire le séquençage en présence de dépendances ?

Pour étayer la communication interprocessus, prenons pour exemple le spouleur d'impression. Les noms des fichiers à imprimer sont stockés dans un répertoire de spool et supprimés au fur et à mesure de leur impression. Le répertoire de spool possède plusieurs entrées numérotées de manière croissante et chaque entrée peut loger un nom de fichier. Pour gérer ce répertoire, deux variables globales partagées (in et out) sont utilisées. La variable in pointe vers la prochaine entrée libre du répertoire tandis que la variable out pointe vers le prochain fichier à imprimer. Le processus qui veut lancer l'impression lit la valeur de la variable globale in, la met dans sa variable locale suivante, puis place le nom de fichier dans l'entrée libre, du répertoire de spool, désignée par le contenu de la variable suivante puis met à jour la variable globale in en lui affectant la valeur de suivant incrémentée de 1.

Le pseudo code correspondant à cette séquence est le suivant:

```
suivant = in;
MettreNomDeFichier (suivant);
In = suivant+1;
```

Admettons qu'à un instant t, in=7 et out =4 et que deux processus A et B désirent lancer l'impression.

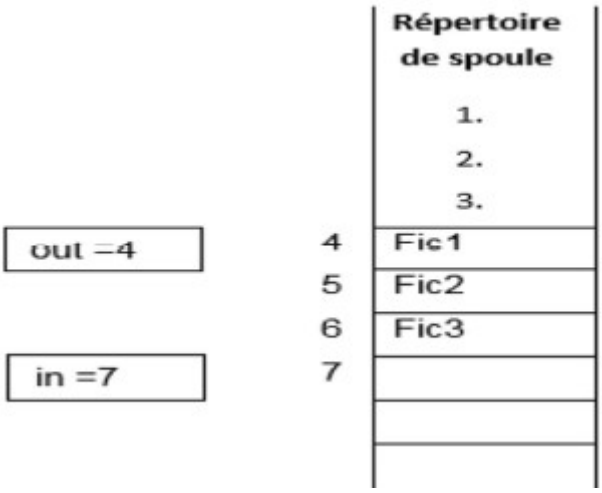

Figure 13: Intention d'accès simultanée de deux processus à la mémoire partagée

Dans ce cas, chaque processus doit exécuter la séquence d'actions d'ajout d'une tâche d'impression. Supposons que le processus A s'exécute et est interrompu par l'ordonnanceur avant de mettre à jour la variable globale in (c'est à dire interrompu quelque part entre la 1ère et la 3ème instruction). L'ordonnanceur jugeant que le processus A a eu suffisamment de temps bascule vers le processus B. Pour ce ? cas les deux processus vont utiliser le même numéro, de l'entrée libre, pour placer la tâche d'impression et l'une des tâches sera sans doute supprimée. Cependant au regard du contenu des variables in et out, le répertoire va rester cohérent et aucune anomalie ne sera détectée par le CPU.
De tels cas de figure sont appelés de situations de concurrences (race conditions en anglais).
Face à une telle situation, il est donc indispensable que le système fournisse un moyen sûr de communication interprocessus permettant la synchronisation et l'échange de données.

2.5.2 Sections critiques

Pour résoudre le cas des problèmes d'accès concurrents (voir exemple ci-dessus), il faut éviter que plus d'un processus accèdent à une donnée partagée en même temps.
Cela consiste à mettre en place un processus d'exclusion mutuelle à la ressource partagée aussi appelée section critique.
Quatre conditions sont à remplir pour résoudre ce problème:
1. deux processus ne doivent pas se trouver parallèlement dans leurs sections critiques;
2. il ne faut pas faire de suppositions sur la vitesse ou le nombre de processeurs mis en œuvre;
3. Aucun processus s'exécutant en dehors de la section critique ne doit bloquer d'autres processus;
4. aucun processus ne dit attendre indéfiniment pour entrer dans sa section critique.
5. Il existe plusieurs façons de mettre en œuvre l'exclusion mutuelle. Ci-dessous nous allons détailler quelques-unes.

2.5.3 Coordination les processus

La coordination (ou exclusion) a pour but de limiter l'accès à une ressource à un ou plusieurs processus. Ceci concerne, par exemple, un fichier de données que plusieurs processus désirent mettre à jour. L'accès à ce fichier doit être réservé à un utilisateur pendant le moment où il le modifie, autrement son contenu risque de ne plus être cohérent. Le problème est semblable pour une imprimante où un utilisateur doit se réserver son usage le temps d'une impression. On appelle ce domaine d'exclusivité, une section critique.

2.6 Les sémaphores

Le concept de sémaphore permet une solution élégante à la plupart des problèmes d'exclusion. Ce concept nécessite la mise en œuvre d'une variable, le sémaphore, et de deux opérations atomiques associées `P` et `V`. Soit `séma` la variable, elle caractérise les ressources et permet de les gérer. Lorsqu'on désire effectuer une exclusion mutuelle entre tous les processus par exemple, il n'y a virtuellement qu'une seule ressource et on donnera à `séma` la valeur initiale de 1.

Lorsqu'un processus effectue l'opération `P(séma)` :

- Si la valeur de `séma` est supérieure à 0, il y a alors des ressources disponibles, `P(séma)` décrémente `séma` et le processus poursuit son exécution,
- Sinon ce processus sera mis dans une file d'attente jusqu'à la libération d'une ressource.

Lorsqu'un processus effectue l'opération `V(séma)` :

- S'il n'y a pas de processus dans la file d'attente, `V(séma)` incrémente la valeur de `séma`,
- Sinon un processus en attente est débloqué.

`P(séma)` correspond donc à une prise de ressource et `V(séma)` à une libération de ressource. Dans la littérature, on trouve parfois d'autres terminologies, respectivement,

`wait(séma)` et `signal(séma)` ou

`get(séma)` et `release(séma)`.

2.6.1 Le problème des producteurs et des consommateurs

Ce problème est un des cas d'école favoris sur les processus. Deux processus se partagent un tampon de données de taille `N`. Un premier processus produit des données et les écrits dans le tampon. Un second processus consomme les données du tampon en les lisant et en les détruisant au fur et à mesure de leur lecture. Initialement, le tampon est vide. La synchronisation de ces deux processus peut se réaliser grâce à des sémaphores.

Les processus producteur et consommateur doivent accéder de manière exclusive au tampon, le temps d'une lecture ou d'une écriture. Un sémaphore d'exclusion mutuelle est donc nécessaire. D'autre part, on peut considérer que les ressources du processus producteur sont les emplacements vides du tampon, alors que les emplacements pleins sont les ressources du processus consommateur. Au départ, le tampon étant vide, les ressources de consommation sont nulles, alors que les ressources de production correspondent à la taille du tampon.

Les sémaphores sous Unix

Le système Unix, dans la plupart de ses versions, fournit maintenant des sémaphores. Les sémaphores ne correspondent cependant pas à une programmation classique de ce système.

On leur préfère les verrous ou bien les tubes quand c'est possible. Par ailleurs, les paramètres des fonctions sémaphores peuvent être légèrement différents suivant les versions d'Unix. En cas de doute, consultez le manuel.

À chacune des étapes de la manipulation d'un sémaphore correspond un appel système Unix :

1. Déclaration : `Type Sémaphore séma;`
2. Initialisation : `séma <- 1;`
3. Opération : `P(séma);` ou bien `V(séma);`

On doit inclure les fichiers d'en-tête : `<sys/types.h>`, `<sys/ipc.h>` et `<sys/sem.h>`. On regroupe les sémaphores en tableaux de type `ushort`. Par exemple : `ushort semarray[2]` est un tableau de deux sémaphores. La déclaration d'une variable « séma » se fait par la fonction :

```
int semget(key_t clé, int NombreSéma, int flag);
```

Cette fonction rend un entier, qui servira de référence ultérieure au sémaphore dans le programme. Elle permet l'identification universelle d'un sémaphore (à l'échelle du système). On l'utilise généralement de la façon suivante :

```
cle = ftok(UnFichier, 'UneLettre');

idSema = semget(cle, NombreSéma, IPC_CREAT | 0600);
```

L'affectation d'une variable sémaphore se fait par l'affectation du tableau. On peut ainsi initialiser plusieurs sémaphores – autant que la dimension du tableau. Par exemple :

```
semarray[0] = 1;

semarray[1] = 2;
```

puis par sa prise en compte par le système par une opération atomique :

```
int semctl(int idSema, int NombreSéma, int commande, ushort semarray);
```

La valeur de `commande` pour affecter les valeurs du tableau aux sémaphores est `SETALL`. Dans certaines versions d'Unix, la dernière variable `semarray` de type `ushort` est parfois remplacée par une structure `union semnum` qui doit contenir ce tableau.

`semctl()` permet d'autres opérations telles que la lecture de la valeur de sémaphores avec la commande `GETALL` et la destruction avec la commande `IPC_RMID`.

Les opérations P et V peuvent être groupées (on peut en effectuer plusieurs en même temps). Elles se font par la construction d'un tableau d'opérations de type `struct sembuf`. Avec la déclaration : `struct sembuf sops[2];` par exemple, il pourra y avoir jusqu'à deux opérations P ou V simultanées.

Les champs à affecter de la structure sont :

- `sem_num`, qui donne le rang du sémaphore dans le tableau précédent,
- `sem_op`, mis à -1 si c'est un P et à 1 si c'est un V.
- `sem_flg`, en général à 0.

La prise en compte des opérations se fait par :

```
int semop(int idSema, struct sembuf *sops, size_t NbOpérat)
```

On peut visualiser l'état des sémaphores par la commande `ipcs` ou `ipcs -a` et on peut détruire un sémaphore par la commande `ipcrm -s no_du_sémaphore`.

2.7 Les moniteurs

Qu'est-ce qu'un moniteur?

Un moniteur est un objet encapsulant des fonctions membre – ou des méthodes – dont une seule peut s'exécuter à un moment donné. La structure de moniteur permet d'éviter certains phénomènes d'interblocages qui peuvent apparaître avec les sémaphores. Ces interblocages sont généralement la conséquence d'erreurs de programmation vicieuses et difficiles à détecter. C'est pourquoi, on considère que l'écriture de sémaphores est délicate. La déclaration d'un moniteur est beaucoup plus facile et plus élégante par contre.

Dans l'exemple précédent, en pseudo-C, quel que soit le nombre de processus qui s'exécutent et qui font appel au moniteur, seul un exemplaire de `écrivain` ou de `lecteur` pourra s'exécuter à un moment donné.

```
monitor MonMoniteur{
    char tampon[100];
    void écrivain(char *chaîne) {}
    char *lecteur() {}
}
```

Dans l'exemple précédent, en pseudo-C, quel que soit le nombre de processus qui s'exécutent et qui font appel au moniteur, seul un exemplaire de `écrivain` ou de `lecteur` pourra s'exécuter à un moment donné.

Synchroniser les fils d'exécution

Les fils d'exécution s'exécutent dans le même espace mémoire. Ils peuvent tous accéder aux variables communes d'une classe simultanément. Comme pour les processus, ça peut conduire à des effets indésirables. Le langage Java ne dispose pas de sémaphores qu'il remplace par des moniteurs.

2.8 Ordonnancement

L'ordonnancement règle les transitions d'un état à un autre des différents processus. Cet ordonnancement a pour objectifs de :

1. Maximiser l'utilisation du processeur;
2. Être équitable entre les différents processus;
3. Présenter un temps de réponse acceptable;
4. Avoir un bon rendement;
5. Assurer certaines priorités.

2.8.1 Le tourniquet

Cet algorithme est l'un des plus utilisés et l'un des plus fiables. Chaque processus prêt dispose d'un quantum de temps pendant lequel il s'exécute. Lorsqu'il a épuisé ce temps ou qu'il se bloque, par exemple sur une entrée-sortie, le processus suivant de la file d'attente est élu et le remplace. Le processus suspendu est mis en queue du tourniquet.

Le seul paramètre important à régler, pour le tourniquet, est la durée du quantum. Il doit minimiser le temps de gestion du système et cependant être acceptable pour les utilisateurs. La part de gestion du système correspond au rapport de la durée de commutation sur la durée du quantum. Plus le quantum est long plus cette part est faible, mais plus les utilisateurs attendent longtemps leur tour. Un compromis peut se situer, suivant les machines, de 100 à 200 ms.

2.8.2 Les priorités

Dans l'algorithme du tourniquet, les quanta égaux rendent les différents processus égaux. Il est parfois nécessaire de privilégier certains processus par rapport à d'autres. L'algorithme de priorité choisit le processus prêt de plus haute priorité.

Ces priorités peuvent être statiques ou dynamiques. Les processus du système auront des priorités statiques (non-modifiables) fortes. Les processus des utilisateurs verront leurs priorités modifiées, au cours de leur exécution, par l'ordonnanceur. Ainsi un processus qui vient de s'exécuter verra sa priorité baisser. Pour un exemple d'exécution avec priorités, on pourra consulter Bach.

2.8.3 Le tourniquet avec priorités

On utilise généralement une combinaison des deux techniques précédentes. À chaque niveau de priorité correspond un tourniquet. L'ordonnanceur choisit le tourniquet non vide de priorité la plus forte et l'exécute.

Pour que tous les processus puissent s'exécuter, il est nécessaire d'ajuster périodiquement les différentes priorités.