I.S.I.T.V.

SYSTEME D'EXPLOITATION

J.L DAMOISEAUX

Tables des matières

Avertissements

Ce document est le support du cours consacré aux 6H d'enseignement à l'I.S.I.T.V., sur les systèmes d'exploitations. Ce polycopié n'est toutefois pas un véritable cours sur les systèmes d'exploitation, car seules les notions les plus importantes et les plus accessibles y sont traitées. A chaque fois que cela sera possible, un lien avec le système UNIX sera réalisé, et à ce titre, ce polycopié servira de support aux travaux pratiques sur UNIX. Enfin, les plus intéressés par cette matière pourront consulter avec profit les nombreux ouvrages existant sur le domaine, et notamment ceux cités dans la bibliographie.

Chapitre 1 / Généralités

Supposons que l'ordinateur exécute le programme suivant :

```
void main(void)
{
char c;
c = getchar();
putchar(c);
return;
}
```

et demandons-nous ce que fait celui-ci pour afficher le caractère tapé au clavier ?

Notre programme, en arrivant sur l'instruction `getchar`, fait appel au système d'exploitation pour l'informer qu'il attend un caractère au clavier. Le système d'exploitation interrompt donc notre programme et lance un programme capable de lire sur l'unité d'entrées-sorties correspondant au clavier. Lorsque l'on frappe sur une touche du clavier, le code correspondant à la lettre tapée est envoyé par le clavier à l'ordinateur. Une fois capté et stocké par le circuit électronique gérant la communication entre le clavier et l'ordinateur, le programme chargé de la lecture est informé que la donnée est disponible. A ce moment là, une copie de ce code est réalisée puis transmise système d'exploitation. Celui-ci réactive alors notre programme en lui fournissant le code du caractère. Notre programme passe à la seconde instruction qui consiste à afficher ce caractère, et un mécanisme semblable sera mis en route pour afficher le caractère à l'écran.

Chacune de ces étapes pourrait être décrite d'une manière encore plus précise. Ainsi, pour lancer le programme de lecture du clavier, le système d'exploitation détermine dans une table l'adresse sur le disque du programme en question. Puis, il assure la rotation du disque, positionne la tête de lecture à cette adresse, et transfère en mémoire la suite de 0 et de 1 composant le fichier. Enfin, après avoir chargé le contexte d'exécution du programme, le système d'exploitation lui donne la ``main'' et celui-ci peut alors s'exécuter.

Comme le laisse entrevoir cet exemple, le système d'exploitation intervient de manière cohérente à tous les niveaux du fonctionnement d'un ordinateur. Aux deux extrémités, il interagit d'une part avec le matériel (le temps d'interaction est alors de quelques milliardièmes de seconde) et d'autre part avec l'utilisateur (le temps d'interaction est de quelques secondes). Le rôle d'un système d'exploitation est donc de réduire et de dominer la complexité de la machine afin d'en faciliter l'usage à l'utilisateur (humain ou programme), celui-ci se retrouvant alors débarrasser de toutes les tâches trop proches de la machine.

1. Historique des systèmes d'exploitations

L'informatique n'existe que depuis la deuxième guerre mondiale, et il est loin le temps des machines à lampes lorsque l'on parle des ordinateurs de la cinquième génération. L'évolution des fonctionnalités et par conséquent la taille des systèmes d'exploitations est allée de pair avec les progrès technologiques. La période décrite s'étale donc de 1940 à 1990, mais au-delà des années 1970, il est difficile de dégager, dans le développement des systèmes d'exploitations, autres choses que des tendances générales.

Entre le début des années 1940 et la fin des années 1950, les ordinateurs étaient constitués de lampes à vide. Outre une taille respectable, les ordinateurs de l'époque étaient extrêmement coûteux, fragiles et sensibles à divers insectes considérés comme nuisibles. Aussi, un seul groupe de personnes concevait, construisait, et utilisait une machine. Les systèmes d'exploitation étaient alors inconnus et, il fallait réserver pendant un certain temps la salle machine, pour, au moyen cartes électriques puis de cartes perforées, travailler sur un ordinateur. L'informaticien préhistorique programmait donc un ordinateur en langage machine sur des cartes, et assurait une gestion complète du contexte d'exécution de son programme (chargement, entrées-sorties, etc.).

Entre les années 1955 et 1965, l'arrivée du transistor modifia complètement la situation en rendant les ordinateurs fiables, donc commercialisables. Le chargeur devint le premier programme résident, les premiers compilateurs (FORTRAN et Assembleur) apparurent et les entrées-sorties devinrent accessibles par des macro-instructions et gérées par l'ordinateur. L'arrivée des mémoires secondaires (disques, bandes et tambours), provoqua également le développement des systèmes de gestion des fichiers. Enfin, pour optimiser l'accès à l'ordinateur, les moniteurs d'enchaînement des travaux virent le jour ; on soumettait à l'ordinateur des fournées (batch) de travaux (bacs de cartes), chacuns précédés de cartes indiquant la nature du travail (compilation, exécution, etc.). Toutefois, un problème se posa rapidement : le débit d'entrée et de sortie des données était beaucoup plus faible que le débit de leur traitement ce qui provoquait un non travail de l'unité centrale.

Avec l'arrivée des circuits intégrés, l'acuité de ces problèmes devint très importante. On imagina donc, vers 1965, des unités d'entrées-sorties asynchrones : l'unité centrale activait l'unité d'entrées-sorties et lui précisait le nombre d'octets à transférer ainsi que leur adresse en mémoire ; tandis que l'unité centrale continuait son travail, l'unité d'entrées-sorties effectuait les transferts à son rythme, puis prévenait l'unité centrale de la fin de son travail (les concepts d'interruptions et de mémoire tampon furent introduits à ce moment).

A cette même époque, afin d'augmenter le rendement de l'UC lorsque celle-ci était bloquée dans l'attente d'informations extérieures, ou par une saturation des tampons d'entrées-sorties, le concept de multiprogrammation naquit. Lorsqu'un programme était bloqué en attente d'une entrée-sortie, le processeur lui était retiré au profit d'un autre programme qui pouvait alors s'exécuter ; le programme retiré était placé dans une file d'attente.

Cette technique de la multiprogrammation nécessita d'une part l'utilisation de circuits spécialisés dans la protection d'une tache par rapport aux autres, et d'autre part la mise en place de techniques d'ordonnancement des travaux, de partage de la mémoire et du processeur.

Malgré l'utilisation de ce cette technique, on souhaita augmenter le temps de réponse pour chaque utilisateur, et on introduisit alors dans la multiprogrammation la notion de temps partagé. Un programme en cours d'exécution pouvait perdre l'unité centrale soit pour des besoins d'entrées-sorties, soit parce qu'il était déjà resté suffisamment longtemps actif (quantum de temps ; à intervalle de temps réguliers, une horloge produit une interruption dont le traitement donne l'unité centrale au programme suivant). A noter également, que c'est à cette époque que les techniques de gestion d'une mémoire hiérarchisée virent le jour : tout ou partie d'un programme oscillera entre la mémoire primaire et la mémoire secondaire.

Au travers de cette histoire, il apparaît très clairement que l'on a toujours cherché à rentabiliser le matériel en développant un logiciel (le système d'exploitation) permettant de l'exploiter au mieux. A l'heure actuelle, le coût du matériel est devenu nettement moindre que celui du logiciel, et la conception d'un système d'exploitation demande plus de temps que celle d'une machine. Les tendances matérielles et/ou logicielles en cours sont l'intégration dans le silicium des fonctionnalités des systèmes d'exploitations, la définition d'interfaces graphiques augmentant la convivialité des machines, le développement du parallélisme et des systèmes répartis.

2. Fonctions d'un système d'exploitation

Un système d'exploitation a pour unique but de mettre à la disposition d'un ou plusieurs utilisateurs une machine virtuelle qui, d'une part libère le programmeur de la complexité du matériel, et d'autre part lui offre des fonctionnalités plus importantes que celles de la machine physique. Parmi les nombreuses fonctions assurées par un système d'exploitation, les principales sont :

- la gestion de la mémoire centrale ; comment partager la mémoire entre plusieurs utilisateurs, comment faire partager des données entre plusieurs utilisateurs, comment une machine, dont la taille mémoire est de n mots, peut elle stocker un programme et des données dont la taille est plus grande que n mots mémoire, etc.

- l'exécution des commandes d'entrées-sorties qui sont toujours lentes par rapport à la vitesse du CPU ; comment synchroniser la CPU avec l'unité d'entrées-sorties, et si plusieurs utilisateurs désirent imprimer un fichier, etc.

- la gestion du CPU ; comment exécuter plusieurs programmes à la fois, détermination de l'utilisateur ``qui à la main``, que faire s'il ne veut pas la rendre, et s'il y a une coupure de courant, etc.

- la gestion des mémoires secondaires ; comment gérer le problème de l'accès à un même fichier par plusieurs utilisateurs, comment retrouver ou stocker de l'information sur un disque, comment optimiser l'utilisation du disque, etc.

- fournir un environnement de travail à l'utilisateur ; l'interprétation d'un langage de commande et l'enchaînement des travaux demandés, etc.

3. Le système UNIX

Historiquement, le système UNIX a été conçu, dans le laboratoire de la compagnie BELL ATT, entre 1969 et 1971 par Ken Thompson, Brian Kernigham, Denis Ritchies. Entre 1971 et 1975 il est réécrit en C et continue d'être développé au sein de ce laboratoire. Mis à la disposition des universités, de l'état américain et des compagnies commerciales, les utilisateurs, disposant des sources, peuvent facilement l'étudier et le modifier.

L'année 1978 marque un tournant dans l'histoire d'UNIX puisque la première version commerciale, appelée V7, sort sur le marché. A partir de cette version, deux familles d'UNIX émergent. Tout d'abord, les laboratoires ATT et BELL réalisent des développements qui

conduiront aux versions SYSTEM V. Ensuite, les travaux réalisés à l'université de Berkley aboutissent aux versions BSD.

A l'heure actuelle, en plus de toutes les versions ``mineures``, deux grandes familles d'UNIX coexistent, celle de l'université de Berkley (BSD) et celle de la compagnie AT&T (System V). Grâce à des groupes de réflexions comme `/usr/group`, OSF, Unix International, etc., un effort de standardisation a été entrepris depuis 1984. Ainsi, l'accès au noyau se fait au travers d'interfaces comme X-OPEN ou POSIX, la gestion de l'écran graphique se fait au travers de l'interface X-WINDOW (elle-même accessible au travers d'interfaces comme MOTIF ou OPEN-LOOK).

L'histoire d'UNIX permet de tirer certaines conclusions quant à ses qualités et ses défauts :

- UNIX n'est pas un système d'exploitation jeune, il a donc fait ses preuves,
- résultat du travail de nombreuses sources d'améliorations, il est attractif, possède toutes les fonctionnalités d'un système d'exploitation "moderne", mais il manque toutefois d'homogénéité et de sûreté,
- écrit à 95% en C, UNIX est un système portable.

3.1 Présentation et Structure du système UNIX

Le système UNIX est un système multi-utilisateurs et multi-taches. Ses domaines de compétences s'étendent de la gestion au calcul scientifique, en passant par les bases de données, la bureautique, etc. Le système UNIX est une succession concentrique de couches logicielles (figure n° 1).

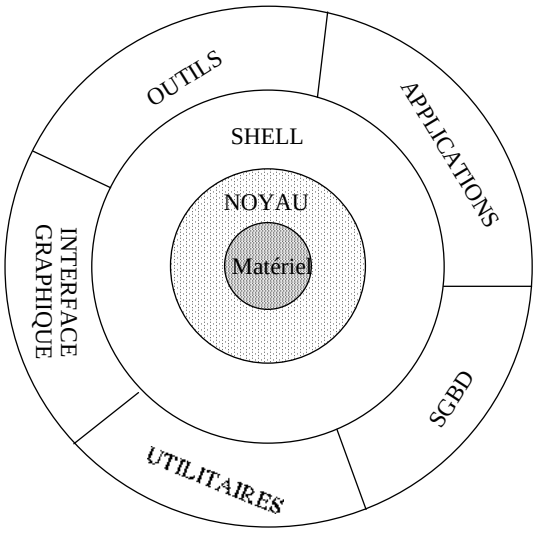

Figure n° 1 : Vue générale du système UNIX

La première de ces couches est le noyau. Composant logiciel directement en contact avec le matériel, toutes les demandes des autres couches logicielles lui parviennent via une interface de fonctions d'appel. Le noyau créé l'environnement UNIX pour une machine donnée et assure comme principales fonctions :

- l'organisation et la gestion des données du système,

- la protection du système et l'accès à l'information,
- la communication de l'information entre les différents éléments du système,
- le fonctionnement multi-utilisateurs et multi-tache en planifiant l'utilisation des ressources matériels (processeur, mémoire, etc.),
- l'établissement de statistiques quant à son activité.

La seconde de ces couches est le Shell. Le Shell est l'interpréteur des commandes introduites par l'utilisateur, et à ce titre il sert d'interface entre le noyau et l'utilisateur. Il existe plusieurs interpréteurs de commandes (les plus connus sont le Bourne Shell -le premier historiquement, le C-Shell, et le Korn Shell)[1].

Quelque soit le shell utilisé, celui-ci analyse la commande tapée au clavier par l'utilisateur, évalue le cas échéant les caractères spéciaux (*, ?, >, <, |, etc.), vérifie que la commande existe et assure son exécution par la création d'un processus adéquat. Une fois la commande exécutée, le contrôle reviendra alors au Shell qui relancera le dialogue.

La dernière couche regroupe quant à elle divers utilitaires, des applications usuelles, des logiciels de gestion de base donnée, des interfaces graphiques, etc.

3.2 Les fichiers sous UNIX

Au niveau physique, le système UNIX considère un fichier comme une suite non structurée d'octets, et lui associe un numéro unique appelé i-node. Au niveau logique, UNIX distingue trois types de fichiers :

- les fichiers ordinaires comme les documents et les programmes,
- les fichiers spéciaux comme les périphériques, les disques, etc. ; ces fichiers spéciaux travaillent soit en mode caractère, soit en mode bloc,
- les répertoires ou catalogues qui contiennent des informations sur d'autres fichiers.

3.2.1 Accès à un fichier

Avant de définir l'accès à un fichier, il convient de préciser une notion importante qui est celle de répertoire. Un répertoire est un fichier spécial qui contient le nom d'autres fichiers. Grâce à cette notion de répertoire, et grâce à l'introduction des deux fichiers que sont le `.` et le `..` (le `.` désignant le répertoire lui-même, et le `..` désignant son père -le répertoire auquel il appartient-), UNIX définit une structure arborescente (figure n° 2) de l'ensemble des fichiers (la racine de cette arborescence étant conventionnellement représentée par le caractère `/`).

[1] Nous travaillerons avec le C-Shell, dont le prompt est symbolisé par le caractère %

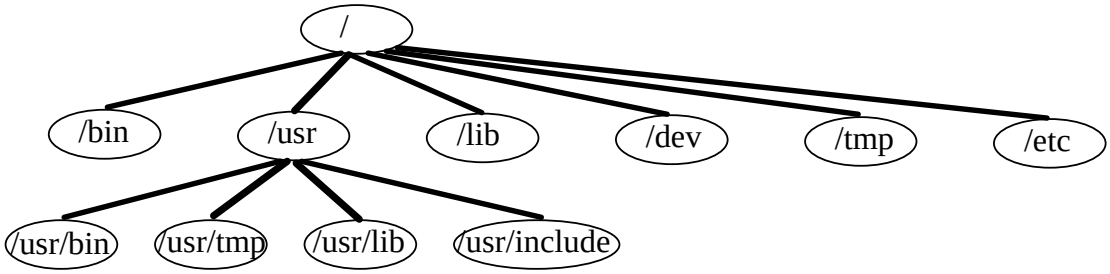

Figure n°2 : Structure arborescente du système de fichiers sous UNIX

Il existe donc, pour tout fichier, au moins un chemin partant de la racine de l'arborescence, et se terminant sur le fichier en question. Ce chemin, appelé la référence absolue, est constitué par la suite des différents répertoires traversés pour joindre le fichier (chaque élément de cette suite étant séparé par un `/`). Par exemple, le chemin suivant `/users/profs/jld/essai.c` précise que le fichier essai.c est situé dans le répertoire `jld`, lui même situé dans le répertoire `profs`, lui même situé dans le répertoire `users` appartenant au répertoire racine.

Toutefois, il est possible de construire des chemins d'accès à un fichier à partir du répertoire où l'on se trouve (chemin relatif) ; ces chemins d'accès commencent à partir des répertoires `.` et `..` . Par exemple, si le répertoire courant est `jld`, je peux désigner le fichier `more` appartenant au répertoire `bin`, en écrivant `../../../bin/more` .

3.2.2 Utilisation d'un fichier

L'utilisation d'un fichier est réglé par un principe de permissions d'accès. Il existe quatre types de permissions qui déterminent le mode d'emploi autorisé pour un fichier :

- `-` aucune permission,
- `r` `read` la lecture du fichier est possible,
- `w` `write` l'écriture du fichier est possible,
- `x` `execute` l'exécution du fichier est possible.

Pour les répertoires, ces permissions ont un sens différent :

- `r` `read` il est possible de consulter le répertoire,
- `w` `write` il est possible de créer ou de détruire des fichiers dans ce répertoire.
- `x` `execute` il est possible d'accéder aux fichiers contenus dans ce répertoire.

A ceci, UNIX ajoute une classification des types d'utilisateurs potentiels pour lesquels seront regroupés les modes d'emploi possibles des fichiers :

- `user` pour le propriétaire,
- `group` pour le groupe propriétaire,
- `other` pour le reste des utilisateurs du système.

Par type d'utilisateur sont associés les quatre modes d'emploi possibles, ce qui fait douze types de permissions par fichier.

3.2.3 Nom générique des fichiers

La création de nom générique pour les fichiers permet d'utiliser une commande sur un ou plusieurs fichiers qui répondent au modèle élaboré. Ces modèles sont construits à partir des caractères suivants :

- `?` qui remplace un caractère quelconque,
- `*` qui remplace une suite éventuellement vide de n'importe quel caractère,
- `[...]` qui remplace n'importe quel caractère repris dans la liste de caractères inclus entre crochets ; cette liste peut être formée par deux caractères séparés par un `-` qui détermine alors un ensemble de caractère.

Ainsi, le nom générique `b[0-9]*.C` désigne tous les fichiers dont le nom commence par un `b`, dont le deuxième caractère est un chiffre, et se terminant par `.C` .

3.3 La redirection des entrées-sorties

Les processus nécessitant des données en entrée ou fournissant des résultats en sortie, les lisent ou les écrivent sur leur entrée et leur sortie standard, à savoir le clavier et l'écran. Il est possible d'affecter ces fichiers d'entrée et de sortie à d'autres dispositifs que la console qui contrôle le processus.

La redirection de l'entrée standard d'un processus depuis un fichier d'entrée suit la syntaxe

```
% commande < fichier_entree
```

La redirection de la sortie standard d'un processus dans un fichier suit la syntaxe

```
% commande > fichier_sortie
```

si `fichier_sortie` existe déjà, il est préalablement effacé.

Cependant, lors d'une utilisation enchaînée de plusieurs commandes, le passage des résultats par des fichiers intermédiaires, n'est pas un mode de communication optimal. La notion de tube palie à cet inconvénient. Un tube ou pipe (symbolisé par le caractère |), permet de rediriger la sortie d'un processus sur l'entrée d'un autre. Ainsi, lors de la construction d'une fonction complexe, chaque traitement modifiera directement les résultats du traitement précédent. La syntaxe de ce mode de communication pour n processus est la suivante :

% $\texttt{processus}_1$ | $\texttt{processus}_2$ | ... | $\texttt{processus}_n$

ainsi la sortie du `processus`$_1$ est directement connectée sur l'entrée du `processus`$_2$, qui lui même voit sa sortie reliée à l'entrée du `processus`$_3$, etc.

3.4 Quelques commandes

UNIX met à la disposition d'un utilisateur plus de 200 commandes. Une commande est considérée comme une suite de mots séparés par des blancs, et peut-être décomposée comme suit :

% nom_commande option parametre

où après le nom de la commande, on trouvera éventuellement la ou les options qui influenceront son exécution, suivis du ou des paramètres sur lesquels elle agit.

Voici pour terminer quelques commandes faisant partie de la vie courante de l'utilisateur du système UNIX :

- `passwd` ; permet de changer le mot de passe de son login,
- `man sujet` ; apporte une aide en ligne sur le sujet choisi,
- `date` ; affiche la date et l'heure courante,
- `echo arguments` , affiche sur la sortie standard le ou les arguments,
- `ps` ; fournit des informations sur les processus du système,
- `who` ; affiche la liste des utilisateurs connectés à la machine,

Voici également quelques commandes et leurs principales options. Toutes aussi utiles que les précédentes, ces commandes sont toutefois plus liées au système de gestion des fichiers.

- `pwd` ; affiche la référence absolu du répertoire où le se trouve,
- `cd [chemin_relatif_ou_absolu]` ; permet de se déplacer dans l'arborescence. Utilisée sans paramètre, le répertoire privé de l'utilisateur devient le répertoire de travail.
- `ls [-lasR][nom_fichier]` ; liste le contenu d'un répertoire, et/ou les caractéristiques de ces fichiers (modes d'accés, type, etc.).
- `mkdir nom_repertoire` ; créé un répertoire vide,
- `rmdir [-i] nom_repertoire` , efface un répertoire préalablement vidé,
- `rm [-ir nom_fichier` ; efface un ou des fichiers,
- `cp src dest` ; recopie de fichiers de la source vers la destination,
- `mv src dest` déplace le fichier de la source vers la destination,

- `cat fichier` écrit les fichiers sur la sortie standard,

- `chmod arg fichier` modifie les droits d'accès à un fichier,

- `more fichier`... affiche page par page, sur la sortie standard, le contenu des fichiers,

- `find ref arg`... permet de rechercher récursivement à partir de la référence ref les fichiers satisfaisant l'expression booléenne déduite des arguments `arg`. L'argument `-name nom_fichier` permet de spécifier le nom du fichier cherché, l'argument `-print` permet d'afficher les résultats de la recherche,

- `grep expression fichier` permet de sélectionner dans un `fichier` les lignes contenant `expression` ;

- `sort [-r] fichier`... trie dans l'ordre lexicographique les différentes lignes d'un fichier ;

- `wc [-lwc] fichier...` compte pour chaque fichier (ou l'entrée standard si aucun fichier n'est précisé) ce qu'il contient.

Chapitre 2 / Les Processus

L'analyse du fonctionnement d'un système d'exploitation montre la présence d'un ensemble d'activités multiples, simultanées ou non, et présentant de nombreuses interactions mutuelles. Pour décrire ce fonctionnement, on introduit la notion de processus comme définition d'une activité élémentaire. Un processus est une entité dynamique représentant l'exécution d'un programme ; un processus naît, vit et meurt.

Au cours de son existence, un processus n'est jamais totalement isolé des autres : il partage des données ou des ressources matérielles, échange des signaux ou des informations, arrête ou tue d'autre processus, etc. Aussi, l'objet de ce chapître est de préciser la notion de processus, de montrer comment elle est mise en oeuvre et comment est programmée la coordination entre processus.

1. Définitions

1.1 Instructions, processeur, processus

Un programme est une suite ordonnée d'instructions dont la nature est fonction du langage de programmation considéré, et dont l'exécution peut être complexe. Une instruction est considérée comme indécomposable (indivisible), c'est à dire que l'on s'interdit d'observer le système pendant l'exécution d'une instruction.

Le processeur est l'entité câblée ou non capable d'exécuter une instruction.

Un processus est un programme en cours d'exécution. Un processus est défini par le code et les données du programme, son contexte d'exécution (vecteur d'état) à savoir les informations qu'il utilise explicitement (variables, procédures, ressources, etc.) et les informations utilisées par le système pour gérer l'attribution des ressources (contenus des registres -CI, RI, etc.-, la taille mémoire utilisée, ressources matérielles utilisées). Il est important de se rappeler que le vecteur d'état d'un processus ne peut être observé pendant l'exécution d'une instruction.

1.2 Ressource, état d'un processus

Une ressource est une entité pouvant servir à l'exécution d'un travail. Des exemples de ressources sont les organes de la machine (unité centrale, mémoire centrale, périphériques), un fichier, des données, etc.

Un processus est dans un état bloqué s'il lui manque la ou les ressources nécessaires à l'exécution de sa prochaine instruction ; dans le cas contraire le processus est dit actif.
Habituellement on distingue deux types de blocage possibles :

- le blocage technologique pour l'absence de ressources,
- le blocage intrinsèque pour un problème de synchronisation.

Lorsque plusieurs processus existent en même temps sur un même système, on dit qu'ils sont parallèles. Il y a deux sortes de parallélisme :

- le parallélisme vrai, où n processus s'exécutent sur m processeurs (architecture multiprocesseur),
- le parallélisme simulé, où n processus s'exécutent sur un processeur qui est successivement attribué à chacun des processus.

1.3 Accès aux ressources

Une ressource est dite locale à un processus s'il est le seul à pouvoir l'utiliser ; une ressource locale disparaît à la mort du processus.

Une ressource qui n'est locale à aucun processus est dite globale.

Une ressource est dite partageable avec n points d'accès si cette ressource peut être attribuée, au même instant, à n processus au plus (un fichier ouvert en lecture, les commandes sous UNIX).

Une ressource partageable avec un point d'accès est dite critique (processeur, imprimante, etc.).

On appelle section critique d'un processus, la phase du processus pendant laquelle la ressource critique est utilisée par ce processus.

Un ensemble de processus peut soit entrer en compétition pour l'accès à une ressource, soit fonctionner en coopération pour mener à bien une application. Que l'on soit dans une situation de parallélisme vrai ou simulé (les tranches d'activités des processus sont trop courtes pour être prises en compte dans la programmation), dès l'instant où plusieurs processus ont une ressource en commun, l'ordre dans lequel ils s'exécutent n'est pas indifférent. Il apparaît alors des problèmes de synchronisation et de communication entre les processus.

2. Exclusion mutuelle

L'exclusion mutuelle est le problème qui se pose lorsqu'une ressource critique est nécessaire à la continuation de plusieurs processus.

Pour mieux comprendre cette notion d'exclusion mutuelle et les problèmes qui lui sont associés, étudions l'exemple d'un client d'un magasin qui envoie deux commandes séparées pour deux articles différents (le montant de l'argent disponible sur le compte est m, la première facture est d'un montant m_1, la seconde d'un montant m_2). Le service comptabilité traite ces commandes simultanément grâce au programme informatique suivant :

```
/* contexte commun */
compte_utilise = FAUX;
compte_client = 1000;

/* corps du programme du processus i */
derniere_facture = mi;
while (compte_utilise == VRAI);
                  <-    /*   le   problème   se   situe   ici   */
compte_utilise = VRAI;
compte_client = compte_client - derniere_facture;
compte_utilise = FAUX;
```

Chacune des factures étant associée à un processus `i` réalisant une facturation d'un montant m_i. Pour l'instant, nous supposerons que les deux processus ont des vitesses quelconques et inconnues. Très simplement, nous avons associé au compte du client la variable booléenne `compte_utilise`, commune aux deux processus, et prenant la valeur vraie ou faux selon qu'un processus utilise ou non la ressource "compte client".

Dans le cas d'une machine multiprocesseurs et avec la variable `compte_utilise` initialisée à faux, chaque processus accède en même temps à celle-ci et donc la teste avant que l'autre ne lui ait affecté la valeur vraie. Pour chacun des processus, la valeur `m` du compte est alors lue, et la valeur finale du compte sera égale à `m -` m_1 ou `m -` m_2 (selon que le $processus_1$ ou le $processus_2$ modifie le compte en premier) au lieu d'être égale à `m -` m_1 `-` m_2.

Dans le cas d'une machine monoprocesseur et avec la variable `compte_utilise` initialisée à faux, le $processus_1$ accède à la variable `compte_utilise` et la teste, puis se trouve interrompu par le système qui donne la main au $processus_2$. Le $processus_2$ accède alors à la variable `compte_utilise` et la teste. Par conséquent les deux processus ont testé la variable `compte_utilise` avant que l'un ou l'autre des processus ne lui ait affecté la valeur vraie, ce qui conduit à la situation précédemment décrite.

L'analyse de cet exemple montre que le problème décrit vient d'une part du fait qu'un processus peut être interrompu entre deux instructions, et d'autre part que des ressources critiques, lorsqu'elles sont utilisées par un processus, doivent rester inaccessibles aux autres processus. De nombreuses solutions ont été proposées pour résoudre ce problème de l'exclusion mutuelle. Avant d'étudier les principales, définissons les propriétés essentielles que toutes devront respecter :

- à tout instant, un processus au plus peut se trouver en section critique,
- si plusieurs processus sont bloqués en attente de la ressource critique, alors aucun processus ne se trouve en section critique, et l'un d'eux doit pouvoir y rentrer au bout d'un temps fini,
- si un processus est bloqué hors d'une section critique, ce blocage ne doit pas empêcher l'entrée d'un autre processus dans sa section critique,
- la solution doit être la même pour tous les processus.

2.1 Attente active

L'attente active est une solution câblée du problème de l'exclusion mutuelle, et tous les ordinateurs ne disposent pas d'un tel mécanisme. L'emploi d'une unique variable booléenne ne convient pas car, entre le test de la variable booléenne et son affectation par un processus `i`, un autre processus `j` avait la possibilité d'accéder à cette variable. La solution la plus immédiate consiste donc à réaliser ces deux opérations simultanément :, la nouvelle opération s'appelant `TAS` (Test And Set). Cette instruction, agissant sur une variable `m`, peut se décrire ainsi :

```
instruction TAS(m)
debut
bloquer l'accès à la cellule mémoire m
lire le contenu de m
si m est faux alors
      m <- vrai
      compteur_ordinal <- compteur_ordinal + 2
sinon
      compteur_ordinal <- compteur_ordinal + 1
fin si
libérer l'accès à la cellule mémoire m
fin
```

Soit `p` une variable booléenne indiquant que la ressource critique `R` est occupée ou non ; `p` est initialisée à faux. L'entrée en section critique est réalisée par

```
E:    TAS(p)
            goto E;
```

D'après ce qui précède, le processus ne pourra sortir de cette boucle, c'est à dire exécuter l'instruction suivant le branchement, que s'il trouve `p` à faux pendant l'exécution de l'instruction `TAS`.

La sortie de la section critique est réalisée par

```
p <- faux
```

Sur l'exemple du compte client dans un grand magasin, le programme associé à un processus est le suivant :

```
/* contexte commun */
compte_utilise = FAUX;
compte_client = 1000;

/* corps du programme du processus i */
derniere_facture = m_i;
E :   TAS(compte_utilise);
goto E;
compte_client = compte_client - derniere_facture;
compte_utilise = FAUX;
```

Il est important de remarquer que pour programmer l'exclusion mutuelle d'une ressource R, on a recouru à un mécanisme câblé d'exclusion mutuelle à une autre ressource p. Cette solution est appelée attente active car le processus bloqué sur p boucle sur l'instruction de test et monopolise donc le processeur.

2.2 Les sémaphores

Les sémaphores sont une des solutions les plus élégantes au problème de l'exclusion mutuelle. Un sémaphore s est constitué d'une variable e_s et d'une file d'attente f_s. Lors de la création du sémaphore, e_s reçoit une valeur positive ou nulle et s_s est vide. On ne peut agir sur un sémaphore qu'au travers des deux primitives indivisibles suivantes :

```
Ps
début
      es <- es - 1
      si es < 0 alors
            etatr <- bloqué   r processus exécutant cette primitive
            mettre le processus r dans la file fs
      fin si
fin

Vs
début
      es <- es + 1
      si es • 0 alors
            sortir un processus q de la file fs
            etatq <- actif
      fin si
fin
```

Un sémaphore est donc un dispositif qui thésaurise un certain nombre d'autorisation de passage (e_s est ce nombre) et qui gère une file de processus en attente de passer. L'opération P_s correspond à une demande de passage, tandis que l'opération V_s est une autorisation de passage. Afin de ne pas retomber dans les travers précédents, le système devra, pour un sémaphore s, assurer l'exclusion mutuelle de e_s et f_s, ainsi que l'indivisibilité des procédures P_s et V_s.

L'exclusion mutuelle sera réalisée grâce à un sémaphore appelé mutex (mutuelle exclusion), initialisé ainsi :

```
fmutex <-    vide                    emutex <-    1
```

et utilisé comme ceci :

```
Pmutex
instructions formant la section critique
Vmutex
```

Comme on peut le constater, dès lors qu'un processus désire utiliser une ressource critique, il doit obligatoirement réaliser l'opération P_{mutex} avant. Deux cas se présentent :

- le processus décrémente d'une unité la variable e_{mutex}, et comme personne n'utilise la ressource critique, la valeur de e_{mutex} qui était égale à `1` passe à `0` ; la procédure P_{mutex} se termine.

- le processus décrémente d'une unité la variable e_{mutex}, et comme un processus utilise la ressource critique, la valeur de e_{mutex} devient ou reste négative ; le processus demandeur est alors mis dans la file d'attente f_{mutex} et la procédure P_{mutex} se termine.

La libération de la ressource critique se fait obligatoirement par la procédure V_{mutex}. Deux cas se présentent :

- le processus incrémente d'une unité la variable e_{mutex}, et comme personne d'autre n'a émis le désir d'utiliser la ressource critique, la valeur de e_{mutex} qui était égale à `0` passe à `1` ; la procédure V_{mutex} se termine.

- le processus incrémente d'une unité la variable e_{mutex}, comme d'autres processus ont émis le désir d'utiliser la ressource critique, la valeur de e_{mutex} est négative et l'on sort donc de la file d'attente l'un des processus demandeurs pour le rendre actif ; la procédure V_{mutex} se termine.

A titre d'exemple, nous reprendrons la gestion du compte client pour trois processus P_1, P_2, P_3 exécutant le programme suivant :

```
derniere_facture = n_i;
P(compte_utilise);
compte_client = compte_client - derniere_facture;
V(compte_utilise);
```

tandis que le tableau suivant récapitule le déroulement simultané de ces trois processus.

Processus et instruction	valeur de compte_utilise	file d'attente	utilisation de la ressource	valeur de la ressource
-	1	vide	non	1000 francs
1 P	0	vide	non	1000 francs
2 P	-1	2	non	1000 francs
3 P	-2	2 - 3	non	1000 francs
1 facture 100fr	-2	2 - 3	oui	900 francs
2 en attente	-2	2 - 3	non	900 francs
3 en attente	-2	2 - 3	non	900 francs

1 V	-1	2	non	900 francs
3 facture 300fr	-1	2	oui	600 francs
2 en attente	-1	2	non	600 francs
3 V	0	vide	non	600 francs
2 facture 200fr	0	vide	oui	400 francs
2 V	1	vide	non	400 francs

3. Synchronisation entre processus

Dans le cadre de la réalisation d'une tâche par plusieurs processus, il existe des relations qui fixent leur déroulement dans le temps. L'ensemble de ces relations est généralement désigné par le terme de synchronisation.

Le problème de la synchronisation consiste donc à définir un mécanisme permettant à un processus actif :

- d'en bloquer un autre ou de se bloquer lui-même en attendant un signal d'un autre processus,
- d'activer un autre processus en lui transmettant éventuellement de l'information.

Deux techniques sont envisageables pour résoudre ce problème de synchronisation :

- l'action directe qui consiste pour un processus à agir sur un autre processus en le désignant par son identité. On utilise généralement deux primitives ayant pour argument l'identité du processus à bloquer ou à activer.
- l'action indirecte qui met en jeu, non plus l'identité du processus, mais un ou plusieurs objets intermédiaires connus des processus coopérants, et manipulables par eux uniquement au travers de primitives spécifiques.

3.1 Synchronisation par sémaphores privés

La synchronisation par sémaphore privés est un mécanisme d'action indirecte. Un sémaphore `s` est un sémaphore privé d'un processus `p`, si seul ce processus peut exécuter l'opération `P(s)` ; les autres processus pouvant agir sur le sémaphore `s` uniquement par l'opération `V(s)`.

La synchronisation par sémaphore privé pose alors comme principe qu'un signal d'activation sera envoyé par la primitive `V`, et attendu par la primitive `P`. Ainsi un processus, dont l'évolution dépend de l'émission d'un signal par un autre processus, se bloque, au moyen d'une primitive `P`, derrière son sémaphore privé initialisé à zéro. Le signal de réveil de ce processus bloqué est obtenu en faisant exécuter par un autre processus une opération `V` sur le même sémaphore.

Par exemple, soit p un processus dont l'évolution dépend de l'émission d'un signal envoyé par un processus q : en introduisant le sémaphore signal initialisé à zéro, la solution du problème se programme comme suit :

```
processus p                          processus q
debut                                debut
Ip1; Ip2; … Ipn;                     Iq1; Iq2; … Iqn;
P(signal)                            V(signal)
… ;                                  … ;
fin                                  fin
```

Deux situations sont possibles :

- le processus p est déjà bloqué sur la primitive P(signal) lorsque le processus q exécute la primitive V(signal), alors le réveil devient effectif ;

- le processus p est actif (il exécute par exemple l'instruction Ip_n) lorsque le processus q exécute la primitive V(signal), alors le signal est mémorisé (le sémaphore passe à 1) et lorsque le processus p exécutera la primitive P(signal) il ne se bloquera pas.

3.2 Problème des Philosophes et des Spaghetti

Il est possible de combiner l'emploi des sémaphores d'exclusion mutuelle et des sémaphores privés, pour réaliser des modèles de synchronisation plus complexes. D'une manière générale, dès qu'un processus p, pour poursuivre ou non son évolution, a besoin de connaître la valeur de certaines variables d'état, qui peuvent être modifiées par d'autres processus (un processus q par exemple), il ne peut les consulter que dans une section critique. Comme il ne peut se bloquer à l'intérieur de celle-ci, le schéma suivant est utilisé :

```
P(mutex)
modification et test des variables d'état
si on peut continuer alors V(sempriv) fin si
V(mutex)
P(sempriv)
```

Si le test des variables d'état indique que le processus p peut continuer, alors en exécutant l'opération V(sempriv), il se donne un droit de passage, et à la sortie de la section critique il ne se bloquera pas sur l'opération P(sempriv). Dans le cas contraire, l'opération V(sempriv) est sautée, et le processus p se bloque à la sortie de sa section critique sur l'opération P(sempriv) puisque le sémaphore était initialisé à 0.

L'activation par un autre processus (le processus q par exemple) s'écrit :

```
P(mutex)
modification et test des variables d'état, suivi éventuellement
d'une opération V sur le sémaphore privé sempriv
V(mutex)
```

Etudions maintenant ces schémas au travers de l'exemple des Philosophes et des Spaghetti. Cinq philosophes se réunissent au cours d'un repas : au menu des spaghetti. Les philosophes étant des hommes aussi, ils ne peuvent penser et manger en même temps, et doivent donc alterner ces deux activités. Selon leur savoir-vivre les spaghetti se mangent avec deux fourchettes, et malheureusement le restaurateur n'a prévu qu'une fourchette par personne.

Pour résoudre ce problème, ils décident d'adopter le rituel suivant :

- chaque philosophe ne quittera pas sa place pendant le repas,
- afin de ne pas se pencher au dessus de la table, un philosophe pour manger utilisera les fourchettes situées à sa droite et à sa gauche,
- même s'il adore les spaghetti, un philosophe doit manger raisonnablement, i.e. pendant un temps limité,
- même s'il a très faim, un philosophe ne doit pas s'emparer d'une fourchette si l'autre est prise,
- au début du repas, tous les philosophes penseront.

Il s'agit donc d'élaborer un mécanisme de synchronisation qui permettent aux cinq philosophes (processus) de manger à leur faim. Les philosophes jouant le même rôle, le mécanisme de synchronisation sera identique pour tous. L'état d'un philosophe `i` est caractérisé par une variable :

```
etat[i] <-  REFLEXION   le philosophe i pense
etat[i] <-  ATTENTE     le philosophe i voudrait manger, mais
                        il lui manque des fourchettes
etat[i] <-  RIPAILLE    lorsque le philosophe i mange
```

Pour un philosophe `i`, le passage de l'état `REFLEXION` à l'état `RIPAILLE` n'est possible, d'après l'hypothèse 2, que si les philosophes situés sur sa droite et sur sa gauche sont dans un état différent de `RIPAILLE`. Si cette condition n'est pas réalisée, le philosophe `i` passe dans l'état `ATTENTE` ; cette transition sera réalisée grâce à un sémaphore privé `sempriv[i]`, initialisé à `0`. La demande de fourchettes s'écrit donc :

```
P(mutex)
si etat[gauche(i)] • RIPAILLE et etat[droite(i)] • RIPAILLE
alors
      etat[i] <- RIPAILLE
      V(sempriv[i])
sinon
      etat[i] <- ATTENTE
fin si
V(mutex)
P(sempriv[i])
```

Naturellement, le test et l'affectation des variables d'état constituent des sections critiques protégées par un sémaphore d'exclusion mutuelle, noté `mutex`.

Pour un philosophe `i`, le passage de l'état `RIPAILLE` à l'état `REFLEXION` entraîne le réveil des philosophes situés à sa droite et à sa gauche si les conditions suivantes sont remplies :

- d'une part ces deux derniers sont dans l'état `ATTENTE`,
- d'autre part on est sûr qu'ils disposeront de deux fourchettes, c'est-à-dire que le philosophe situé à la droite (gauche) du philosophe de droite (gauche) est dans état autre que `RIPAILLE`.

La restitution de fourchettes s'écrit donc :

```
P(mutex)
si etat[gauche(i)]=ATTENTE et etat[gauche(gauche(i))] • RIPAILLE
alors
      etat[gauche(i)] <- RIPAILLE
      V(sempriv[gauche(i)])
fin si
si etat[droite(i)]=ATTENTE et etat[droite(droite(i))]•RIPAILLE
alors
      etat[droite(i)] <- RIPAILLE
      V(sempriv[droite(i)])
fin si
V(mutex)
```

En définitive, l'algorithme d'un philosophe `i` est le suivant :

```
iter
      sortir si (c'est la Fin_du_Repas)
      Penser
      Demander_Fourchettes(i)
      Manger
      Restituer_Fourchettes(i)
fin iter
```

4. Les processus sous Unix

4.1 Généralités

Un processus sous UNIX correspond à un espace mémoire composé de trois zones :

- la zone système (`u-area`) contenant des informations sur le processus ; cette zone est uniquement manipulée par le système,
- la zone des données manipulées par le processus,
- la zone d'instructions ; le code du programme exécuté par le processus est stocké ici.

A chaque processus est associé un numéro d'identification, le `pid`. Ce numéro, attribué par le système, permet de garder une trace de tous les processus, et sert également à la synchronisation et la communication entre processus.

Enfin, chaque processus a un père dont le numéro d'identification est donné par le `ppid`. Cette notion de parenté entre processus apparaît dès lors qu'un processus `A` “lance” un processus `B`, le processus `A` étant le père du processus `B`, et le processus `B` étant bien évidemment le fils du processus `A`. Ainsi, sur l'exemple suivant obtenue par la commande `ps` :

```
UID PID PPID CP PRI NI VSZ   RSS  WCHAN  S  TTY   TIME       COMMA
25  966 959  1   44  0 1.91M 384K pause  S  ttyp2 0:00.45     csh
25  967 966  0   44  0 1.22M 88K  wait   I  ttyp2 0:00.01    essai
```

il apparait que le processus `essai` (`pid 967`, `ppid 966`) est le fils du processus `csh` (`pid 966`).

Il existe deux fonctions `C` qui permettent à un processus de connaître son `pid` et celui de son père. Ces deux primitives sont :

```
int getpid(void)
int getppid(void)
```

La fonction `getpid` renvoie le numéro d'identification du processus appellant, et la fonction `getppid` renvoie le numéro d'identification du père du processus appellant.

4.2 Création de processus

4.2.1 La fonction `system`

La fonction `system` est le premier des mécanismes permettant à un processus de créer un autre processus. Inclus dans le fichier entête `stdlib.h`, le prototype de cette fonction est le suivant :

```
void system(char *commande)
```

et sa réalisation a pour effet d'exécuter la commande donnée en argument, le processus appelant étant interrompu jusqu'à la fin de l'exécution de la commande.

Par exemple, après avoir compilé et exécuté en arrière plan le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
system("sleep 60");
return;
}
```

un appel à la commande `ps` nous permet de mieux comprendre les mécanismes mis en jeux :

```
UID PID PPID CP PRI  NI VSZ       RSS    WCHAN S TTY  TIME     COMMA
25  866 859  1  44    0  1.91M     384K   pause S ttyp20:00.45 csh
25  867 866  0  44    0  1.22M     88K    wait  I ttyp20:00.01 essai
25  925 867  0  44    0  1.71M     160K   wait  I ttyp20:00.01 sh –c
25  966 925  0  44    0  1.22M     96K    -     I ttyp20:00.01 sleep
```

Au cours de son exécution le processus `essai` appelle le noyau (par la fonction `system`). Cet appel entraine la création d'un processus fils de nom `sh -c` (`pid 925`). Ce processus est à son tour le père du processus `sleep` (`pid 966`) correspondant à notre commande `sleep 60` (les processus `essai` et `sh -c` seront bloqués jusqu'à la fin du processus `sleep`).

4.2.2 Les primitives `exec`

UNIX propose d'autres primitives pour créer des processus. Les primitives exec… représentent une famille de primitives dont l'objectif est la création d'un nouveau processus se substituant au processus appelant. Par exemple, la primitive `execl`, dont le protoype est le suivant :

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

substitue au processus appelant le processus dont le lancement correspond à l'exécution du programme arg_0, dont la référence absolu dans l'arborescence des fichiers est donnée par ref, et dont les éventuels arguments de travail sont arg_1, …, arg_n.

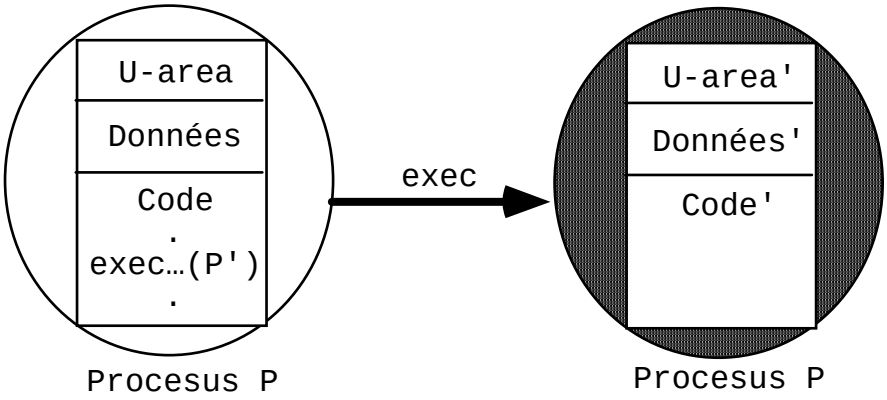

Figure n° 1 : Illustration du mécanisme de la fonction `execl`

Il est important de comprendre que le processus créé remplace le processus appelant, ce qui signifie d'une part qu'il n'y a pas de retour de la primitive `execl` (le processus appelant est purement est simplement éliminé), et d'autre part que le processus demeurant après la réussite de cette fonction possède le même numéro d'exécution, le même père, les mêmes priorités, etc. que le processus appelant (figure n°1).

Une illustration de ce mécanisme est donnée par l'exécution du programme `essai.c` suivant :

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
execl("/bin/ps","ps", "al", (char *)0);
printf("et maintenant je m'arrete\n");
return;
}
```

Au lancement du programme, le processus `essai` est créé. Ce processus de `pid 867` occupe une taille mémoire de `88Ko`.

```
UID PID PPID CP PRI NI VSZ   RSS  WCHAN S TTY   TIME         COMMA
25  866 859  1  44   0 1.91M 392K pause S ttyp2 0:00.45     csh
25  867 866  0  44   0 1.22M 88K  -     R ttyp2 0:00.01     essai
```

Après l'appel à la primitive système execl, on remarque que processus de `pid 867` correspond maintenant à l'exécution de la commande `ps al` (taille mémoire `224Ko`).

```
UID PID PPID CP PRI NI VSZ   RSS  WCHAN S TTY   TIME         COMMA
25  866 859  0  44   0 1.91M 392K pause S ttyp2 0:01.45     -csh
25  867 866  0  44   0 1.45M 224K -     R ttyp2 0:00.03     ps al
```

On notera également que le message “`et maitenant je m'arrete`” n'apparaîtra jamais à l'écran.

4.2.3 La primitive `fork`

La primitive `fork` est utilisée pour la création d'un nouveau processus obtenu par duplication du processus appelant (c'est à dire le processus père) ; le processus nouvellement créé s'exécute de manière concurente au processus qui l'a créé. Le prototype de cette fonction est le suivant :

```
int fork(void)
```

Cette primitive effectue donc une duplication du processus appelant (copie des données, du code, des priorités, etc.), ce qui signifie que le nouveau processus (appelé le fils) exécute une copie du programme mise en oeuvre par le processus appelant (nommé le père). La seule manière de distinguer le père du fils est que la valeur de retour de la fonction `fork`. Dans le processus fils elle est nulle, tandis qu'elle est égale au numéro d'identification du fils dans le processus père (figure n° 2)

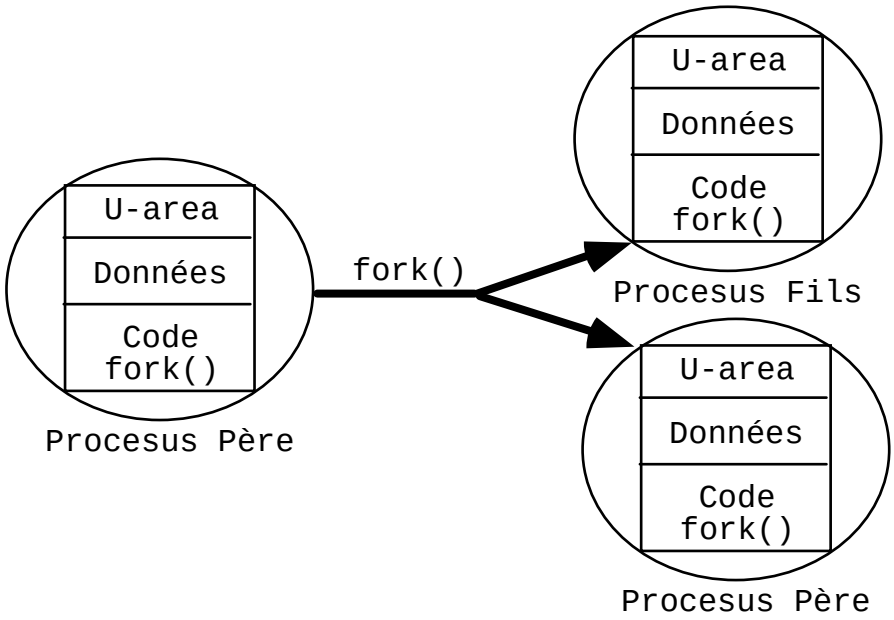

Figure n° 2 : Illustration du mécanisme de la fonction `fork`

Prenons comme exemple, l'exécution du programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
printf("Bonjour je vais me dupliquer\n");
fork();
printf("j'ai reussi\n");
return;
}
```

qui provoque à l'écran deux fois l'affichage du message "`j'ai reussi`". Une analyse plus détaillée, nous montre qu'au lancement du programme, un processus `essai` a été créé.

```
UID PID PPID CP PRI NI VSZ   RSS  WCHAN S TTY   TIME         COMMA
25  866 859  0   44  0 1.91M 400K pause S ttyp2 0:01.45     csh
25  867 866  0   44  0 1.22M 88K  -     R ttyp2 0:00.03     essai
```

Puis, au moment de l'appel à la primitive `fork`, un deuxième processus `essai` est créé. Ce processus est conforme en tous points au premier, à ceci près qu'il est son fils. Ces deux processus ayant le même code, il est donc normal que l'on voit s'afficher à l'écran deux fois le message.

```
UID PID PPID CP PRI NI VSZ   RSS  WCHAN S TTY   TIME         COMMA
25  866 859  0   44   0 1.91M 400K pause S ttyp2 0:01.45     csh
25  867 866  0   44   0 1.22M 88K  -     S ttyp2 0:00.03     essai
25  900 867  0   44   0 1.22M 88K  -     R ttyp2 0:00.03     essai
```

4.3 Synchronisation de processus

4.3.1 La primitive wait

La primitive `wait` permet de synchroniser de manière rudimentaire des processus en suspendant l'exécution du processus en cours tant que l'un de ces fils ne s'est pas terminé. Le prototype de cette fonction est :

```
int wait(int *code)
```

Lorsqu'un processus fils meurt, la fonction `wait` renvoie son numéro d'identification et place dans code l'adresse du code de terminaison de celui-ci.

4.3.2 Synchronisation par signaux

La synchronisation de processus par signaux est un mécanisme assez subtil. Un signal est une interruption logicielle informant un processus qu'un événement extérieur particulier ou anormal s'est produit dans son environnement. Par exemple, lors d'une division par zéro le signal `floating point exception` est émis à l'adresse du processus ne sachant pas calculer, lorsque vous interrompez un processus en tapant au clavier `Ctrl-C` le signal `SIGINT` est envoyé à destination de celui-ci, lorsqu'un processus viole la mémoire le signal `SIGSEGV` lui est adressé, etc.

Il existe de nombreux signaux et tous sont identifiés par une constante symbolique. La liste de ces constantes ainsi que toutes les fonctions permettant de manipuler les signaux sont définies dans le fichier standard `signal.h`. A chaque signal est associé un événement extérieur, et un comportement prédéfini du processus recevant le signal. Toutefois, un signal peut être envoyé sans que l'événement correspondant ne se soit produit -la seule information véhiculée est alors le type du signal-, et de plus le comportement standard d'un processus face à un signal peut être modifié.

Le tableau suivant présente quelques signaux, leur type, le comportement associé et le caractère modifiable ou non de ce comportement :

Nom du signal	Evénement associé	Comportement associé	Modifiable
SIGINT	frappe de CTRL-C	terminaison du processus	oui
SIGFPE	erreur arithmétique	terminaison du processus	oui
SIGKILL	signal de terminaison	terminaison du processus	oui
SIGUSR1		terminaison du processus	oui
SIGUSR2		terminaison du processus	oui
SIGCHLD	terminaison d'un fils	signal ignoré	oui

Envoi d'un signal
L'envoi d'un signal d'un processus à un autre se fait au travers de la fonction kill dont le prototype est le suivant :

```
int kill(int pid, int sig)
```

où `pid` représente le numéro du processus destinataire, et `sig` le signal émis ; cette fonction renvoie `0` si elle s'est correctement déroulée et `-1` sinon. Le processus émetteur et le processus receveur doivent en principe appartenir au même utilisateur.

Réception d'un signal
La prise en compte d'un signal par un processus se traduit par un comportement par défaut désigné symboliquement par la constante `SIG_DFL`, et définissant l'action à réaliser lors de la réception du signal. Ainsi, un processus recevant un signal pourra :

- se terminer,
- se terminer et engendrer une image mémoire (fichier `core`) qui plus tard sera analysée,
- ignorer le signal,
- s'interrompre,
- reprendre son exécution.

Toutefois, il est possible pour certains signaux (voir tableau précédent) de changer ce comportement par défaut. Ainsi, à la réception d'un signal, un processus pourra ignorer celui-ci (constante symbolique `SIG_IGN`), ou adopter un autre comportement. Dans ce cas, une fonction définie par l'utilisateur sera associée au signal, et lors de la réception du dit signal, le processus récepteur effectuera un appel à cette fonction, puis reprendra le cours de son exécution à l'endroit même où il l'avait quitté ; un signal permet donc de réaliser un appel de fonction alors qu'il n'a pas été explicitement programmer. La mise en place de ce

mécanisme dynamique de déroutement est obtenue par la fonction `signal` dont le prototype est le suivant :

```
void (*signal(int sig, void (*p_traitement)(int))) (int)
```

où `p_traitement` sera initialisé soit avec l'une des constantes symboliques `SIG_DFL` et `SIG_IGN`, soit avec l'adresse de la fonction associée au traitement du signal `sig`. Le prototype de cette fonction est le suivant :

```
void traitement(int sig)
```

Une fois le signal capté et traité, le traitement associé au signal reste toujours en place (sauf sur les versions `ATT` d'UNIX où le comportement par défaut est réactivé).

Attente d'un signal
Enfin, la suspension d'un processus dans l'attente d'un signal est réalisé par la fonction `pause` dont le prototype est le suivant :

```
int pause(void)
```

A la délivrance du signal, le processus exécutera le traitement prévu.

Exemple
A titre d'exemple, voici le changement du comportement par defaut d'un processus lors de la réception d'un signal. Ici, notre processus, au lieu de s'arrêter immédiatement à la réception du signal `SIGINT`, s'arrêtera seulement après en avoir capté 5. Voici le code exécuté par notre processus :

```
#include <signal.h>
int c = 0;

void capte(int sig) {
      printf("Aie !!!, on me reveille\n");
      c++;
      if (c > 5) {printf("j'en ai marre, bye bye\n");exit(0);}
      return;
}

void main(void) {
 signal(SIGINT,capte);
 for(;;) {
      printf("je dors!!!\n");
      sleep(5);
      }
 return;
 }
```

La fonction `capte` incrémente la variable `c`, et lorsque sa valeur sera supérieure à `5`, le programme s'arrêtera. Cette fonction est associée, grâce à la fonction `signal`, au traitement

par défaut du signal `SIGINT`, ce qui implique que chaque fois que le processus recevra ce signal, un appel à la fonction `capte` sera immédiatement réalisé.
Le programme principal de ce processus est une boucle infinie dans laquelle le processus affiche "`je dors`" avant de s'endormir réellement pendant `5` secondes (fonction `sleep`).

Chapitre 3 / Gestion des processus

Dans le chapitre précédent, nous avions traité les différents mécanismes à mettre en oeuvre, soit pour protéger des informations partagées entre plusieurs processus (exclusion mutuelle), soit pour permettre à plusieurs processus de coopérer à un même but (synchronisation). L'étude des processus étant fondamentale pour la compréhension d'un système d'exploitation, nous allons maintenant aborder les mécanismes mis en jeu pour assurer à plusieurs processus un déroulement "simultané".

1. Contexte d'un processus

De nombreuses définitions ont été données à la notion de processus. La plus courante est qu'un processus est une entité dynamique représentant l'exécution d'un programme. Il est important de bien comprendre qu'un processus étant capable de recevoir, de la part du système d'exploitation, le contrôle du processeur pour un certain temps, celui-ci n'est pas uniquement défini par le code de son programme, mais également par un ensemble d'informations nécessaires au système.

Cet ensemble d'informations est appelé généralement le contexte ou le vecteur d'état d'un processus. Il est formé de deux parties :

- les informations utilisées par le système,

- les informations utilisées par le processus lui-même.
  Lorsque le système "passera la main" d'un processus à un autre, il le fera en réalisant une opération appelée le changement de contexte.

1.1 Informations utilisées par le processus

Lorsqu'un programme s'exécute, les instructions et les données qui le composent sont situées en mémoire centrale. Ces portions de mémoire sont appelées des segments ; les instructions sont contenues dans le segment procédure, et les données dans le segment données. Les informations utilisées par un processus sont donc la ou les tables de segments, ainsi que divers indicateurs de protection. Les segments procédures peuvent être réentrants (exécutés par plusieurs processus) ce qui évite une duplication du code. Les segments données peuvent selon les situations appartenir à un processus ou plusieurs.

1.2 Informations utilisées par le système

Les informations utilisées par le système servent à gérer l'attribution des ressources pour un processus donné. Habituellement, ces informations sont :

- le contenu des registres généraux ; ces registres sont adressables et modifiables par le processus,

- le Mot d'Etat du Processeur (MEP ou en anglais PSW Program Status Word) ; le MEP contient une copie des registres spécialisés (compteur ordinal, registre d'instruction, etc. -ces registres sont en principe inaccessibles au processus-), l'état d'exécution du processeur, le pouvoir du processus (un processus s'exécute en mode maître ou en mode esclave ; le mode maître lui permet de réaliser les instructions d'entrées-sorties, celles touchant aux interruptions et celles concernant la sécurité et la protection-, la priorité du processus, et pour finir les masques d'interruptions,

- le contenu de la mémoire virtuelle.

Toutes ces informations sont situées dans le bloc de contrôle du processus (PCB Process Control Bloc), les blocs de contrôle de tous les processus étant regroupés dans la table des processus.

2. Le parallélisme

Lorsque plusieurs processus existent en même temps dans un système, on parle alors de parallélisme. Il existe deux sortes de parallélisme :

- le parallélisme vrai, où `n` processus s'exécutent sur `m` processeurs (architecture multiprocesseur),

- le parallélisme simulé, où `n` processus s'exécutent sur un processeur qui est successivement attribué, par tranche de temps, à chacun des processus.

Livrons nous à une petite étude entre le parallélisme vrai et le parallélisme simulé. Dans le parallélisme simulé, le processeur est alloué par tranches de temps, tranches de temps suffisamment courtes (entre 1/50 et 1/10 de seconde) pour donner l'illusion aux utilisateurs de disposer de la machine pour eux seuls. Essayons de voir pourquoi le pseudo-parallélisme est une bonne manière d'utiliser une machine monoprocesseur.

Supposons que nous disposions de trois processus effectuant uniquement du calcul (figure n°1). On remarque que le parallélisme simulé n'est pas un meilleur mode d'exploitation du processeur que la monoprogrammation car, que les processus se déroulent les uns à la suite des autres ou alternativement, le temps total d'exécution est le même.

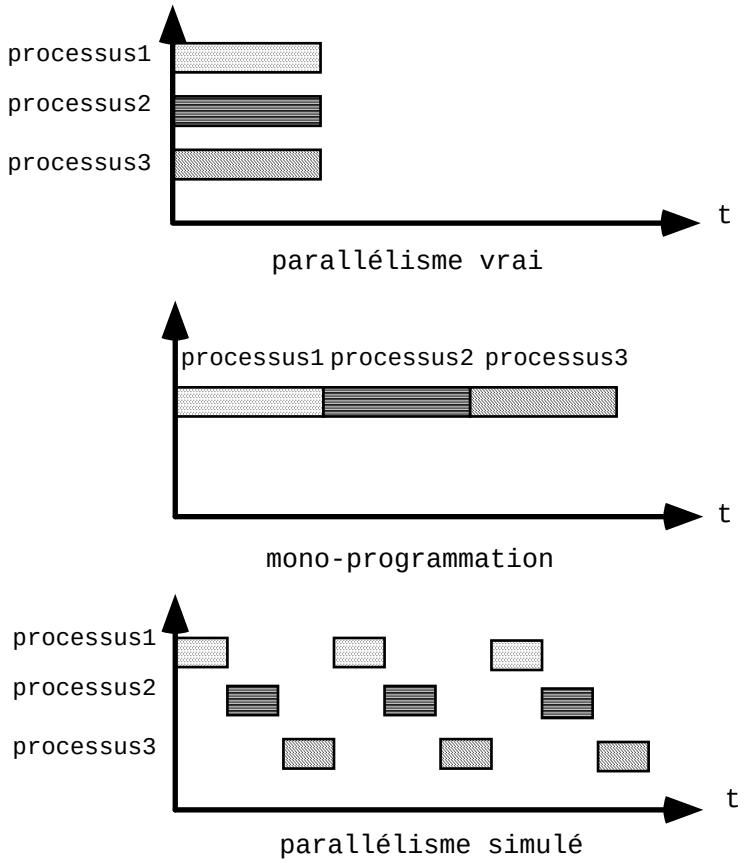

Figure n°1 : Comparaison du temps d'exécution
de trois processus n'effectuant que du calcul

Cependant, rares sont les programmes qui ne font que du calcul, et l'avantage du parallélisme simulé sur la monoprogrammation devient alors évident dès qu'un processus est par exemple bloqué dans l'attente de l'achèvement d'une entrée-sortie (échange avec l'utilisateur, lecture sur le disque, etc.).

Ainsi, à condition de disposer d'un dispositif d'entrées-sorties autonome, dès qu'un processus attend l'achèvement d'une entrée-sortie, le processeur lui est retiré au profit d'un autre processus. Si tous les programmes font suffisamment d'entrées-sorties, on arrive alors à masquer l'exécution des calculs par les temps d'attente des entrées-sorties ; chaque utilisateur a l'impression d'un temps de réponse quasiment identique à celui obtenu s'il était le seul à travailler sur la machine (figure n° 2).

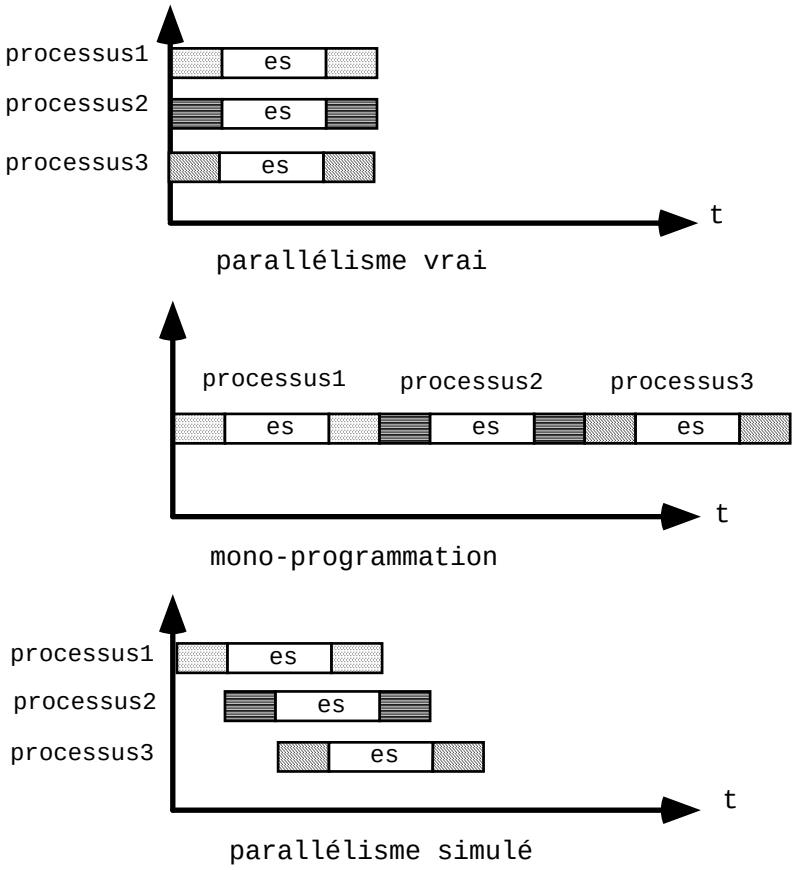

Figure n°2 : Comparaison du temps d'exécution
de trois processus effectuant du calcul et des entrées-sorties

3. Allocation du processeur réel

Nous allons donc voir maintenant comment est simulé le parallélisme sur une machine monoprocesseur, et plus exactement comment est alloué le processeur dans un système multitâches.

3.1 Définition du problème

Le problème de l'allocation du processeur peut être schématisé comme suit : au cours du temps, suivant une certaine loi d'arrivée, des processus demandent à utiliser le processeur (figure n° 3):

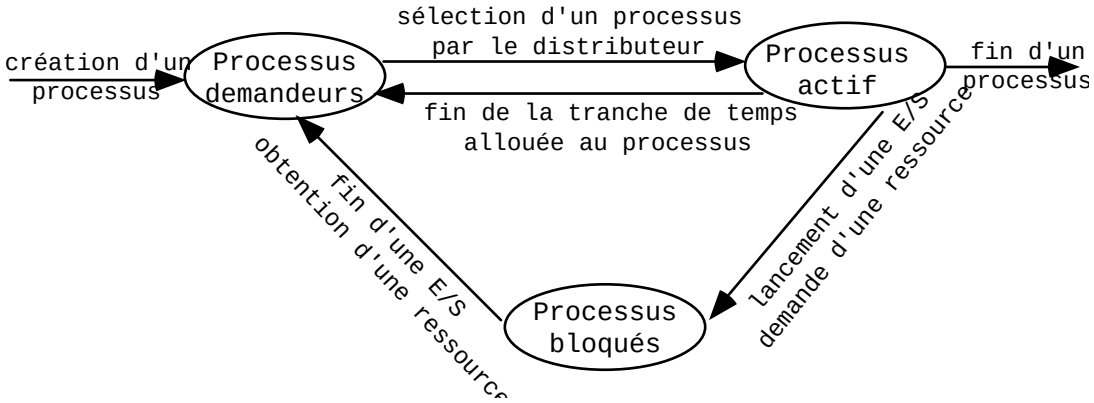

Figure n°3 : Schéma d'allocation du processeur

Dans ce schéma, on retrouve les divers états possibles d'un processus qui sont :

- actif ; seul le processus utilisant le processeur est dans cet état,
- bloqué ; cet état est attribué aux processus attendant la fin d'une entrée-sortie, la libération d'une ressource, ou en communication/synchronisation avec d'autres processus,
- attente ; cet état est attribué aux processus qui ne sont plus bloqués, et à ceux qui ont consommé leur quantum de temps ; dans les deux cas, le processus pourrait reprendre son exécution mais le processeur n'est pas disponible.

Le problème de l'allocation du processeur revient donc à définir le rôle du composant système qui aura la charge de faire transiter les processus d'un état à l'autre.

3.2 Le distributeur

Le distributeur (scheduller en anglais) est la partie du système d'exploitation chargée de veiller à la bonne attribution du processeur au cours du temps. Son rôle consiste :

- d'une part à sélectionner, en fonction d'une certaine stratégie, le processus à qui il attribuera le processeur,
- d'autre part à retirer, après un certain temps d'utilisation ou en fonction d'événements spécifiques (E/S, demande d'une ressource, etc.), le processeur au processus.

L'acquittement de cette tâche devra se faire en respectant les contraintes suivantes :

- la garantie à chaque processus d'un temps donné d'allocation,
- le respect d'un ordre de priorité entre les processus demandeurs,
- l'exécution totale d'un processus,

- la mise à l'écart d'un processus utilisant le processeur pendant un temps supérieur au maximum fixé par le programmeur ou le système.

3.3 Changement de contexte entre deux processus

L'opération, qui consiste pour le distributeur à retirer le processeur à un processus pour l'attribuer à un autre, s'appelle le changement de contexte. Dans le cas d'un changement de contexte entre le processus actif A et un processus demandeur B, cette opération peut se décomposer ainsi :

- le distributeur interrompt le processus A,
- il sauve le contexte du processus A dans la table des processus,
- il réactualise les registres de l'unité centrale à partir du contexte du processus B,
- et enfin il réactive le processus B.

Si l'on étudie ce schéma de plus près, on peut se demander "comment le distributeur s'y prend-il pour sauver le contexte de A ?". En effet, si le distributeur sauvegarde le contexte du processus A, c'est donc le processus associé au distributeur est actif et donc occupe l'unité centrale avec son propre contexte. La solution à ce problème recourt au mécanisme d'interruption. En fait, le système d'exploitation peut programmer l'horloge pour que celle-ci génère une interruption matérielle associée à une routine de changement de contexte. Le schéma précèdent peut donc être affiné ainsi :

- l'horloge génère une interruption qui est détectée par le matériel,
- le fonctionnement normal du processeur est arrêté ; le matériel sauvegarde dans une pile le contexte du processus interrompu,
- le code correspondant à l'interruption est fourni aux circuits de traitement des interruptions qui calculent l'adresse de cette dernière ; l'adresse de l'interruption est recopiée dans le compteur ordinal, le registre d'état est modifié, l'ordinateur passe alors en mode système,
- l'exécution de la routine d'interruption système entraîne la sélection d'un processus et de son contexte,
- la fin de l'interruption provoque le chargement du nouveau contexte, et ainsi rend le processus sélectionner actif.

Il est important de se rappeler que la routine d'interruption de changement de contexte, même si elle est câblée, fait partie du système d'exploitation. Les instructions qui s'y trouvent s'exécutent en mode système, le programme interrompu travaillant normalement en mode utilisateur.

3.4 Stratégie d'allocation du processeur

Les stratégies de choix d'un processus par le distributeur sont nombreuses et visent souvent à réduire le temps de réponse pour des processus nécessitant un temps d'exécution court. Toutefois, d'une manière générale, il est possible de classifier ces stratégies selon qu'elles mettent en oeuvre les deux notions suivantes :

- la préemption ou réquisition du processeur ; lorsqu'un processus important passe dans l'état "demandeur", l'algorithme doit décider ou non d'interrompre le processus en cours pour donner la main au processus important, et le cas échéant il doit déterminer dans quel délai on devra le faire,

- la nature des informations utilisées pour déclencher le changement de contexte ; à coté des stratégies entièrement fixées à priori, il en existent certaines utilisant les informations attachées au processus ou les informations recueillies au cours de l'activité du système.

3.4.1 Stratégie sans recyclage des travaux

Ces stratégies sont généralement utilisées dans le traitement par lots. Elles utilisent la valeur du temps d'exécution du processus, valeur qui en général est inconnue au moment de la demande du processus.

a) File d'attente simple (`FIFO`) : le premier arrivé est le premier servi ; on ne tient pas compte du temps d'exécution ; les travaux courts ont un temps de réponse élevé si ils passent après des travaux longs (figure n° 4).

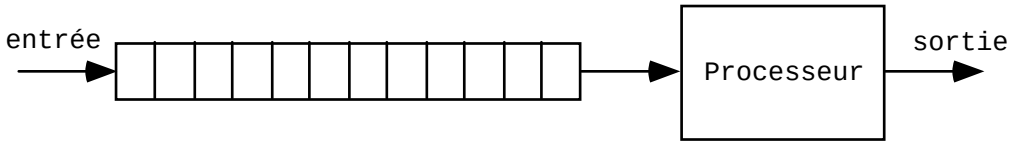

Figure n° 4 : Allocation du processeur par file d'attente simple

b) File d'attente ordonnée de manière croissante suivant le temps estimé d'exécution (figure n° 5): quand un processus arrive il est inséré à l'endroit correspondant à son temps d'exécution ; les travaux courts sont avantagés et les longs toujours retardés .

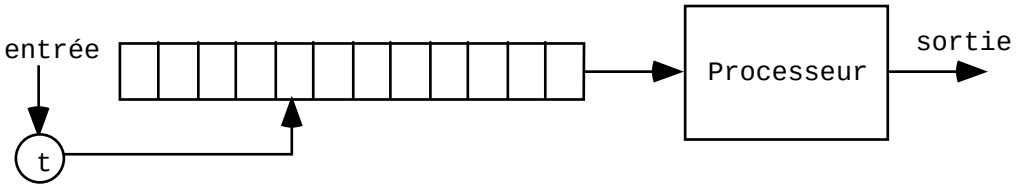

Figure n° 5 : Allocation du processeur par file d'attente ordonnée

Cette stratégie peut être combinée avec un système de priorités croissantes avec le temps d'attente et un temps limite d'utilisation du processeur. On peut également introduire le

concept de préemption : quand un processus arrive, son temps estimé d'exécution est comparé au temps estimé restant pour le travail en cours ; s'il est plus faible, le travail en cours est interrompu et retourne à sa place dans la file d'attente, tandis que le nouvel arrivant le remplace.

3.4.2 Stratégie avec recyclage des travaux

Les méthodes précédentes, en se basant sur une estimation du temps d'exécution (temps éventuellement faux ou falsifié) et en allouant le processeur à un processus durant tout ce temps, sont peu adaptées aux conditions des systèmes conversationnels où les demandes sont nombreuses et fréquentes. Les stratégies avec recyclage des travaux évitent ces inconvénients en interrompant au bout de quelques millisecondes le processus pour allouer le processeur à un autre processus ; les processus interrompus sont placés dans des files d'attentes.

a) Balayage cyclique ou tourniquet (figure n° 6): le processeur est alloué successivement au processus et ce pour un temps déterminé (quantum) ; si le processus ne s'est pas terminé au bout du temps imparti, il est interrompu et remis en queue de la file d'attente.

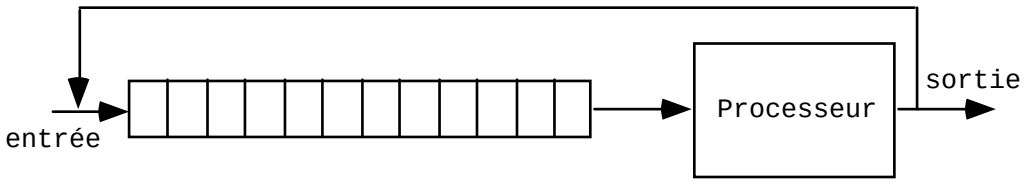

Figure n° 6 : Allocation du processeur par tourniquet

Facile à mettre en place, cette méthode garantie que tout processus est servi au bout d'un temps fini, et limite le délai de prise en compte d'un processus.

b) Recyclage à plusieurs files d'attentes (figure n° 7) : les processus demandeurs sont rangés dans n files Q_1, Q_2, … Q_n ayant chacune un quantum q_i.

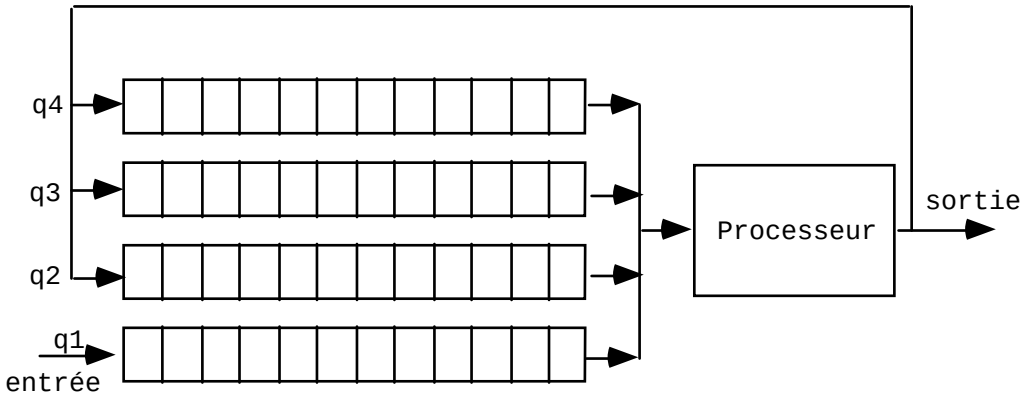

Figure n° 7 : Allocation du processeur par recyclage sur plusieurs files d'attente

Un processus situé en tête d'une file d'attente Q_i ne sera pris en compte que si toutes les files précédentes sont vides ; un processus appartenant à la file Q_i et n'ayant pas terminé, sera

placé en queue de la file Q_{i+1}. Les processus sortant de Q_n y retournent. Enfin, tout processus arrivant dans Q_1 est pris en compte dès que le processeur est libre. Cette stratégie présente les mêmes avantages que la stratégie du tourniquet, avec en plus des ajustements plus souples

3.4.3 Stratégie fondée sur la notion de priorité

La priorité est un nombre attribué à chaque processus et définissant le degré d'urgence de son déroulement. La priorité peut être définie par le programmeur ou modulé par le distributeur. Toutes les stratégies décrites précédemment peuvent intégrer cette notion de priorité. Ainsi une file peut être constitué en ordonnant les processus selon leur priorité, plusieurs files de processus de même priorité peuvent être réalisées. Dans les stratégies avec un recyclage, on peut imaginer que la priorité d'un processus diminue après chaque quantum alloué à ce travail.

Chapitre 4 / Allocation de la mémoire

L'exécution d'un processus demandant que le code du programme et les données utilisées soient présents en mémoire, cette dernière est une ressource essentielle du système d'exploitation. Sa gestion en est confiée à un allocateur qui l'attribuera tout ou partie au(x) processus demandeur(s). L'objet de ce chapitre est donc l'étude de l'allocation de la ressource mémoire au sein d'un système d'exploitation.

1. Définitions

1.1 Espace mémoire réel / espace mémoire virtuel

Lorsqu'un processus s'exécute, il référence un ensemble de ressources situées dans un espace mémoire de travail, et ceci sans se préoccuper de savoir à quelle adresse physique de la mémoire principale se trouvent ces ressources. Cet espace mémoire de travail, par opposition à la l'espace mémoire réel, est appelé l'espace mémoire virtuel ; dans l'espace mémoire virtuel les adresses mémoires sont dites virtuelles.

1.2 Mémoire virtuelle

La mémoire virtuelle est l'ensemble des mécanismes qui permettent d'établir la correspondance entre l'espace mémoire virtuel et l'espace mémoire réel (figure n°1).

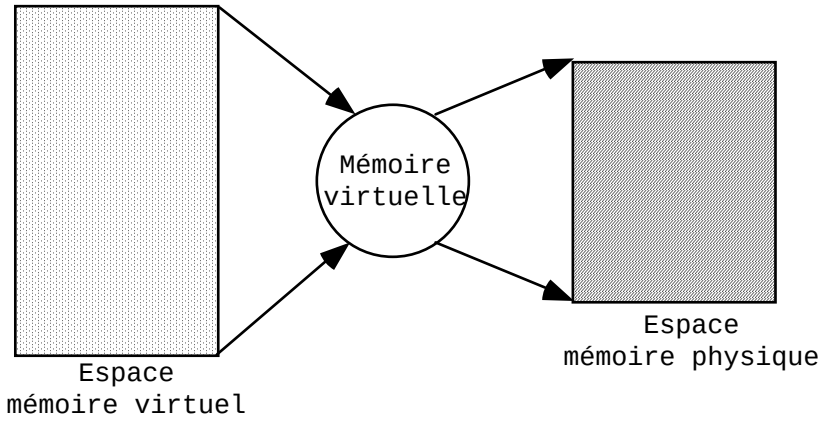

Figure n°1 : Principe de la mémoire virtuelle

La mémoire virtuelle assure comme fonctions :

- la mise en correspondance entre les adresses virtuelles et les adresses physiques,
- la gestion de l'espace mémoire réel (allocation des emplacements, transfert de l'information),
- le partage de l'information entre processus,

- la protection de l'information.

De nombreux mécanismes de mémoire virtuelle ont été élaborés, et on peut les séparer en trois grandes familles :

- ceux où la correspondance entre l'espace mémoire virtuel et l'espace mémoire réel est réalisée lors de l'écriture, de la compilation ou de l'édition de liens du programme, le résultat est un module non translatable qui doit toujours être chargé à la même adresse réelle.

- ceux où la correspondance entre l'espace mémoire virtuel et l'espace mémoire réel est réalisée au moment du chargement du programme ; le résultat de la compilation ou de l'édition de liens est un module translatable dont les adresses seront définitivement fixées lors de son chargement en mémoire réelle. Le chargeur, par réimplantation statique, peut placer le programme n'importe où dans l'espace mémoire réel.

- ceux où la correspondance entre l'espace mémoire virtuel et l'espace mémoire réel est réalisée lors de l'exécution du programme : toute adresse virtuelle est transformée en une adresse réelle au moment où elle est utilisée. Cette traduction dynamique des adresses autorise une réimplantation dynamique d'un programme au cours de son exécution.

1.3 Mémoire uniforme / Mémoire hiérarchisée

Lorsque l'espace mémoire réel est le seul support de l'information adressable par le processeur, celui-ci est dit uniforme. Dans une telle situation, la taille de l'espace mémoire virtuel est inférieure ou égale à la taille de l'espace mémoire réel. En outre, la gestion de l'espace mémoire virtuel se ramène au choix d'un algorithme de placement dans l'espace mémoire réel : une fois placées dans l'espace mémoire réel, les informations y demeureront jusqu'au moment où elles cesseront d'être utilisées.

Lorsqu'on souhaite partager, sur le principe de la réquisition, l'espace mémoire réel entre plusieurs processus, un espace mémoire réel secondaire devient nécessaire pour conserver les informations provisoirement chassées de l'espace mémoire réel ; l'espace mémoire réel est alors dit hiérarchisé. Dans une telle situation, la taille de l'espace mémoire virtuel peut être supérieure ou égale à la taille de l'espace mémoire réel, et la gestion de l'espace mémoire virtuel se ramène au choix d'un algorithme de placement et de remplacement dans l'espace mémoire réel. Cette hiérarchisation comporte au moins deux niveaux dont les caractéristiques l'un par rapport à l'autre sont un coût de stockage moindre, une capacité plus grande et un temps d'accès plus important ; en général, le premier niveau est constitué par la mémoire vive et le second par le disque dur. Dans une telle organisation, il s'agira de répartir l'information entre les différents niveaux de la hiérarchie pour obtenir un temps moyen d'accès à l'information proche de celui de l'unité rapide, avec un coût proche de celui de l'unité lente.

2. Un cas de gestion de mémoire uniforme : le PC

Dans le cas où un seul processus à la fois utilise les ressources du système, la plus simple des gestions de l'espace mémoire réel est celle pratiquée dans le monde de la micro-

informatique[2]. L'espace mémoire réel est ainsi partagé entre le système d'exploitation et l'unique processus utilisateur. Quand un processus demande à s'exécuter, le système d'exploitation le charge dans l'espace mémoire réel, l'exécute et à sa terminaison reprend la main.

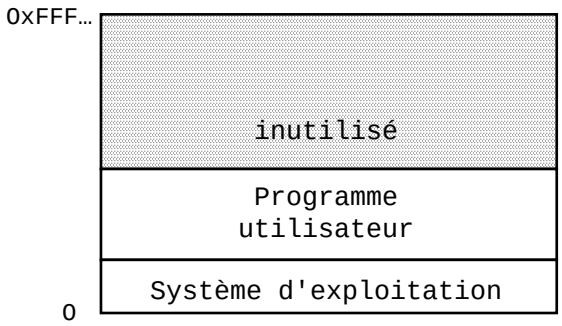

Figure n° 2 : Organisation de la mémoire sur un PC

Une telle organisation peut conduire à de graves problèmes dès lors qu'un processus accède, intentionnellement (virus) ou non (utilisation de structures de données dynamiques, récursivité infinie), à l'espace mémoire réel réservé au système d'exploitation. Ce problème a conduit à développer des solutions logicielles (fonctions permettant de tester si une opération est possible) et matérielles (registres limites) assurant une plus ou moins grande protection.

3. Gestion de la mémoire hiérarchisée

Une mémoire uniforme n'est pas adaptée au multiplexage de l'unité centrale entre un nombre important de processus, et ce pour trois raisons :

- les processus inactifs occupent inutilement une partie de l'espace mémoire réel,
- un processus ne peut pas disposer d'un espace mémoire virtuel de taille supérieure à l'espace mémoire réel,
- plus les processus sont nombreux, moins ils disposent de place.

L'utilisation d'une mémoire hiérarchisée résout ces problèmes en multiplexant l'espace mémoire réel.

3.1 Le Va et Vient (Swapping)

La méthode du Va et Vient consiste à vider sur une unité secondaire le contenu de la zone de l'espace mémoire réel attribuée à un processus, pour ensuite utiliser cette zone pour un autre processus ; lorsque l'on décide de reprendre l'exécution du premier processus, il faut recharger les informations nécessaires en mémoire centrale.

[2]Bien que Window 95 change considérablement les choses.

Que cette technique soit utilisée dans le cadre d'un partitionnement fixe ou variable de l'espace mémoire réel, deux problèmes se posent :

- à moins de disposer d'un mécanisme de réimplantation dynamique, la réactivation d'un processus interrompu implique qu'on le replace au même endroit, ce qui peut entraîner le vidage de plusieurs autres processus,

- il est nécessaire de mémoriser en permanence l'occupation de l'espace mémoire réel.

3.1.1 Gestion de l'espace mémoire réel par table de bits

Une première solution pour la gestion de l'occupation de l'espace mémoire réel consiste à utiliser une table de bits (figure n° 3) :

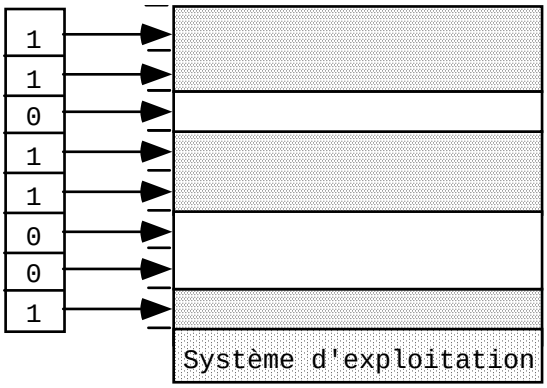

Figure n° 3 : Mémorisation de l'état d'occupation
de la mémoire par une table de bits

l'espace mémoire réel est divisé en unités d'allocation de taille fixe (quelques mots à plusieurs Koctets selon les systèmes), et à chaque unité correspond, dans une table de bits, un bit dont la valeur est `0` si l'unité est libre, et `1` sinon. D'une mise en place rapide, cette technique relativement peu gourmande en place, est lente à l'exécution. En effet, lors de l'introduction d'un processus dont la taille est `k` unités d'allocation, il est nécessaire de trouver dans la table `k` bits consécutifs à zéro, ce qui est très difficile.

3.1.2 Gestion de l'espace mémoire réel par liste chaînée

Une deuxième solution pour la gestion de l'occupation de l'espace mémoire réel consiste à utiliser une liste chaînée des zones libres et/ou occupées (figure n° 4). Les informations composant la liste peuvent être placées soit dans un espace mémoire réservé (chaque maillon contient par exemple un pointeur vers la zone libre, une indication de taille et le(s) lien(s) de chaînage), soit dans la zone libre elle-même (un en-tête placé au début de chaque espace libre établit le lien avec la zone libre suivante).

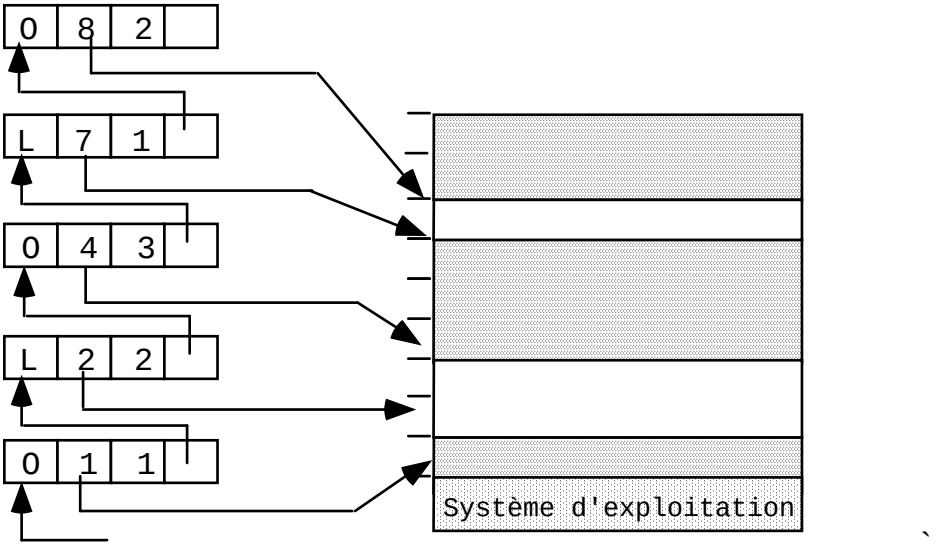

Figure n° 4 : Mémorisation de l'état d'occupation de la mémoire par une liste chaînée

De l'organisation de cette liste dépend l'efficacité des algorithmes. Le chaînage peut être construit dans l'ordre chronologique des libérations, mais le plus souvent on utilise les classements suivants :

- classement par adresses croissantes ou décroissantes,
- classement par tailles croissantes ou décroissantes, le choix dépendant de l'algorithme utilisé pour satisfaire une demande.

Lorsqu'une demande est émise, plusieurs algorithmes de sélection d'une zone sont possibles :

- on prend la première zone possible (`first fit`) qui est découpée en deux parties, l'une attribuée au processus demandeur, l'autre (appelée résidu) inutilisée ; cet algorithme est rapide puisque pratiquement sans recherche,
- on prend la plus petite zone possible (`best fit`) et on évite ainsi de fractionner une grande zone dont on pourrait avoir besoin ultérieurement,
- on peut également prendre la plus grande zone possible (worst fit) et ainsi obtenir le plus grand résidu possible.

D'une manière générale, le premier algorithme est le plus rapide de tous. Toutefois, selon l'organisation de la liste, les performances de ces algorithmes peuvent varier : dans le cas de l'algorithme de la plus petite zone possible, le classement par tailles évite de parcourir toute la liste ; si le classement est fait par adresses ou s'il n'y a pas de classement, une fois la zone allouée, le résidu reste à sa place alors qu'avec un classement par tailles il doit être déplacé. Enfin, ces techniques entraînent un phénomène d'accumulation des résidus en tête de liste, ce qui ralentit les recherches. On utilise souvent une liste circulaire qui permet de commencer l'exploration à partir de n'importe quel point.

3.1.3 Gestion de l'espace mémoire réel par subdivision ("buddy system")

Une troisième solution pour la gestion de l'occupation de l'espace mémoire réel consiste à découper celui-ci en listes de blocs dont la taille est une puissance de deux variant de `1` octet jusqu'à la taille maximale de la mémoire (figure n° 5).

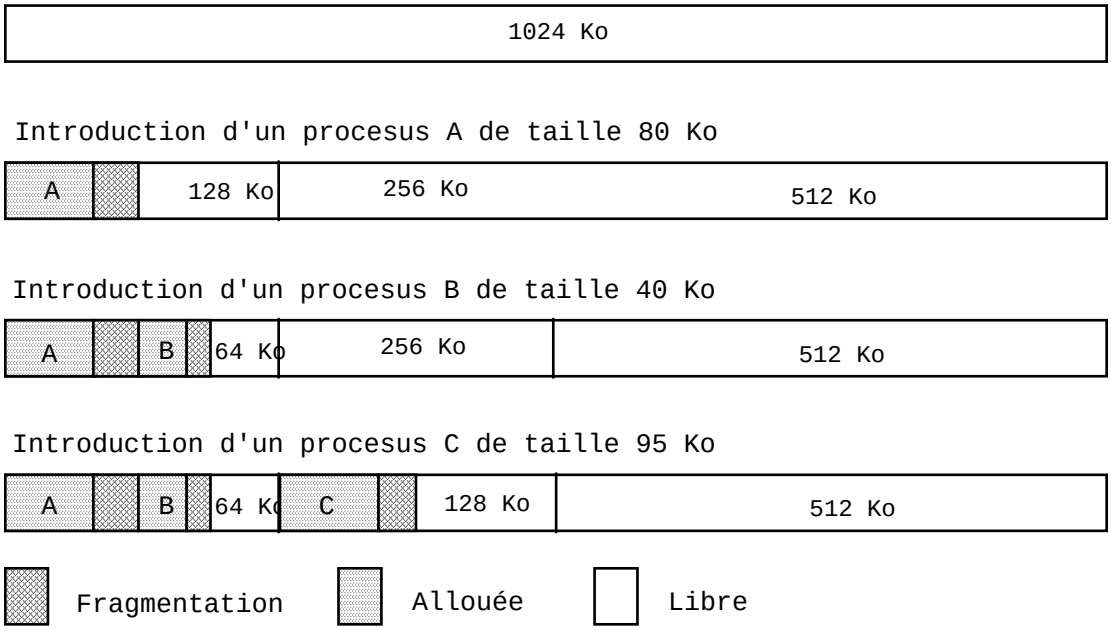

Figure n° 5 : Gestion de la mémoire selon le principe de subdivision

Au départ, il n'existe qu'une seule liste contenant un bloc de la taille de l'espace mémoire réel. Le principe de la méthode est le suivant : si `t` est la taille de l'espace mémoire réel demandé par un processus, on recherche alors un bloc dont la taille est égale à la plus petite puissance de deux qui est supérieure à `t` ; si un tel bloc n'existe pas, alors on le crée par divisions successives de blocs de taille supérieure (la division en deux d'un bloc créé deux blocs appelés les compagnons ou "buddy").

Cette solution, bien que très intéressante lors de la libération d'une zone allouée, conduit à une sous-utilisation de la mémoire. En effet, les tailles des zones étant arrondies à une puissance de deux, il se produit un phénomène important de fragmentation interne.

3.1.4 Libération d'une zone

Lors de la libération d'une zone allouée, trois situations différentes peuvent apparaître selon que la zone libérée est entourée de deux zones libres, d'une zone allouée et d'une zone libre, ou de deux zones allouées (figure n° 6).

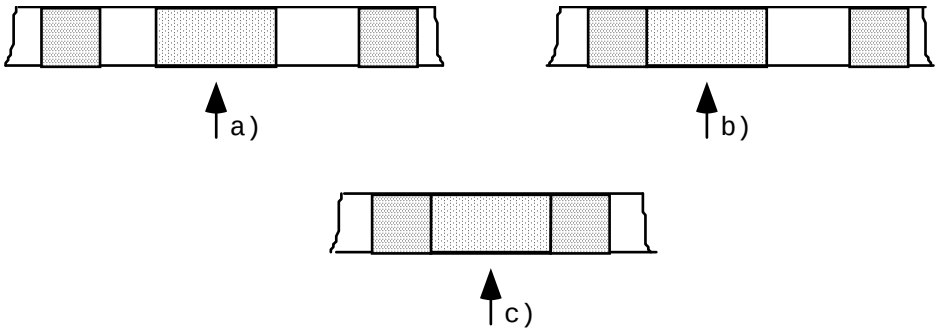

Figure n° 6 : Libération d'une zone mémoire

Chaque fois que cela est possible (cas a et b), il est utile de regrouper, dès la libération, la zone libérée avec les zones libres voisines, pour créer ainsi une zone libre de taille plus grande ; on évite ainsi le fractionnement de la mémoire. La rapidité de ce regroupement dépend de la manière dont l'espace mémoire réel est alloué : pour les listes chaînées, un ordonnancement efficace pour l'allocation ne le sera pas pour la libération ; pour l'allocation par subdivisions, il suffit de fusionner les compagnons.

3.1.5 Fragmentation et compactage

Au bout d'un certain temps d'utilisation, dans des systèmes d'allocation du type demande de zone, une fragmentation de l'espace mémoire réel se produit. Cette situation empêchant l'allocateur de trouver une zone de taille suffisante pour satisfaire toute demande nouvelle, il est alors nécessaire de compacter la mémoire. Ce compactage se traduit par un déplacement de toutes les zones allouées vers une extrémité de la mémoire, ce qui fait apparaître une zone libre dont la taille est la somme des tailles des zones libres primitives (figure n°7).

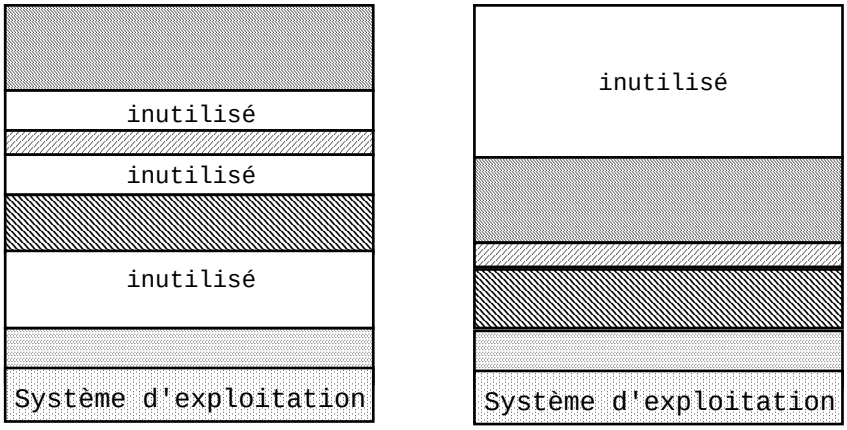

Figure n° 7 : Compactage de l'espace mémoire réel

Cette méthode très coûteuse en temps n'est pas toujours applicable, notamment lors de l'absence de mécanisme de réimplantation dynamique. Par ailleurs, des simulations ont montré que lorsque l'algorithme d'allocation ne peut satisfaire une demande, alors le taux de remplissage de l'espace mémoire réel est tel que, même après un compactage on retombera très vite dans la situation précédente ; le système passe alors son temps à compacter la mémoire.

3.2 La pagination

La pagination est un mécanisme de mise en correspondance d'un espace mémoire virtuel linéaire avec un espace mémoire réel discontinu. Ce mode d'allocation est certainement le plus courant sur bon nombre de machine. Le principe de la pagination consiste d'une part à découper l'espace mémoire virtuel en zones de taille fixe appelées pages virtuelles, et l'espace mémoire réel en pages physiques de même taille, et d'autre part à utiliser un mécanisme (la table de pages) assurant la correspondance entre les pages virtuelles et les pages physiques. Une telle organisation permet à une page virtuelle, qui passe le plus clair de son temps en mémoire secondaire, d'être implantée dans n'importe quelle page physique.

3.2.1 La pagination à un niveau

Dans le mécanisme de pagination à un niveau (figure n° 8), une adresse virtuelle est composée de deux parties : un numéro de page `p` et un déplacement `d` à l'intérieur de la page ; les `P` bits de gauche d'une adresse de `N` bits fournissent un numéro de page, et les `N-P` bits de droite le déplacement dans la page. La taille de la page est 2^{N-P}. La traduction d'une adresse virtuelle en une adresse réelle est réalisée par la table des pages située dans l'espace mémoire réel, et dans laquelle les entrées successives correspondent aux pages virtuelles consécutives. La $p^{ième}$ entrée de cette table contient le numéro `r` de la page où est implantée la page virtuelle p. L'adresse réelle `(r,d)` d'un mot d'adresse virtuelle `(p,d)` est donc obtenue en remplaçant le numéro de page `p` par le numéro de case `r` trouvée dans la table.

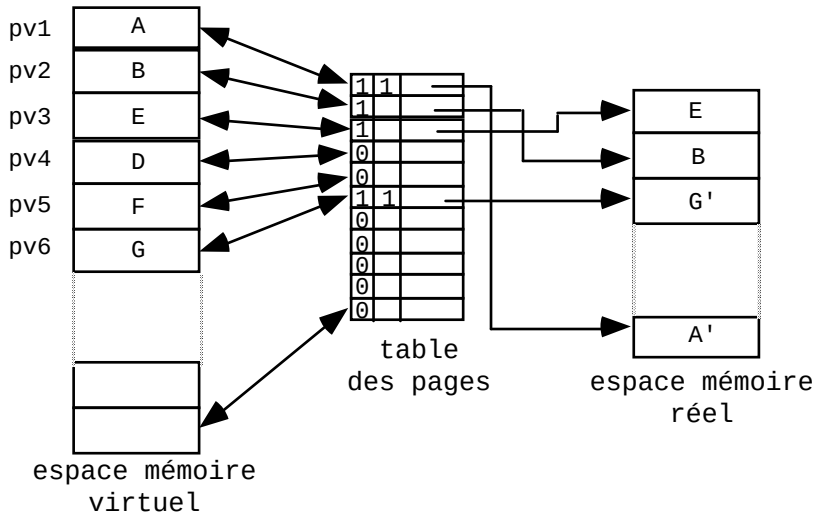

Figure n° 8 : Principe de la pagination à un niveau

Lorsque le mécanisme de traduction des adresses utilise des tables de pages situées dans l'espace mémoire réel, toute traduction ralentit considérablement le processeur. Pour éviter cela, il existe plusieurs mécanismes accélérateurs comme les mémoires caches, les mémoires associatives, etc. dans lesquels on mémorise généralement les numéros des pages les plus fréquemment utilisées. La traduction d'une adresse consiste alors à consulter d'abord ces mémoires, la consultation de la table n'ayant lieu que si le numéro de page virtuelle ne s'y trouve pas.

Les tailles des pages choisies par les constructeurs sont en général du même ordre de grandeur (de 512 à 4096 octets). Ce choix est motivé par les considérations suivantes :

- le phénomène de fragmentation interne des pages est courant dans la mesure où l'on charge, pour des raisons d'efficacité ou de protection, un programme ou des données tête des pages. De plus, statistiquement, la dernière page virtuelle n'est occupée moitié. Donc plus la taille des pages est réduite, plus la fragmentation l'est.

- la taille de la table des pages est proportionnel au nombre de pages de l'espace mémoire virtuel. Il est donc préférable d'avoir des pages de grande taille pour réduire la taille de la table, ce qui permet de la placer dans des registres ou dans une mémoire cache, d'où un gain de temps.

- le temps de lecture d'une page sur la mémoire secondaire est pratiquement proportionnel temps de positionnement de la tête de lecture. Pour réduire ce temps, il est préférable de travailler avec des pages de grande taille.

Afin d'améliorer le rendement du mécanisme de pagination, il est souvent associé à chaque entrée de la table des pages les informations suivantes :

- un bit d'invalidité qui lorsqu'il est positionné à 1 indique que a page virtuelle n'est pas déjà présente dans l'espace réel,

- des bits d'utilisation qui permettent de connaître l'usage qui est fait de la page (lecture, écriture),

- des bits de protection qui indiquent si la page peut être lue, écrite, exécutée.

3.2.2 Le remplacement de pages

Lorsqu'un processus référence une page virtuelle absente de l'espace mémoire réel, on dit q'on a un défaut de page. Le traitement d'un défaut de page est réalisé par le système qui avant de charger la nouvelle page virtuelle, devra commencer par libérer de la place dans l'espace mémoire réel en renvoyant en mémoire secondaire une page réelle. L'algorithme qui consiste à choisir la page à renvoyer s'appelle l'algorithme de remplacement. De nombreuses études ont été réalisées, et elles se distinguent essentiellement par les informations prises en compte et relatives à l'utilisation passée des pages.

Voici les principaux algorithmes de remplacement de page :

- le tirage aléatoire `RAND` ; la page éjectée est choisie au hasard,

- ordre chronologique de chargement `FIFO` ; la page éjectée est la page la plus anciennement chargée,

- ordre chronologique d'utilisation `LRU` (Least Recently Used) ; la page éjectée est la page la moins récemment utilisée,

- degré d'utilisation LFU (Least Frequently Used) ; la page éjectée est la page la moins fréquemment utilisée,

- l'algorithme optimal OPT ; s'il existe des pages qui ne seront plus référencées alors remplacer l'une de ces pages, sinon remplacer la page qui sera le plus tardivement référencée.

Le dernier algorithme est totalement irréalisable car il est impossible de connaître à l'avance le moment où les différents pages qui seront référencées. Cependant, cet algorithme sert de référence pour la comparaison des performances des autres algorithmes (figure n° 9). Toutefois, au delà de l'ordre des performances, il est important de noter que d'une part les performances des algorithmes sont toutes proches de celles de l'algorithme RAND, et d'autre part l'influence de la taille mémoire est très largement supérieure à celle de l'algorithme de remplacement.

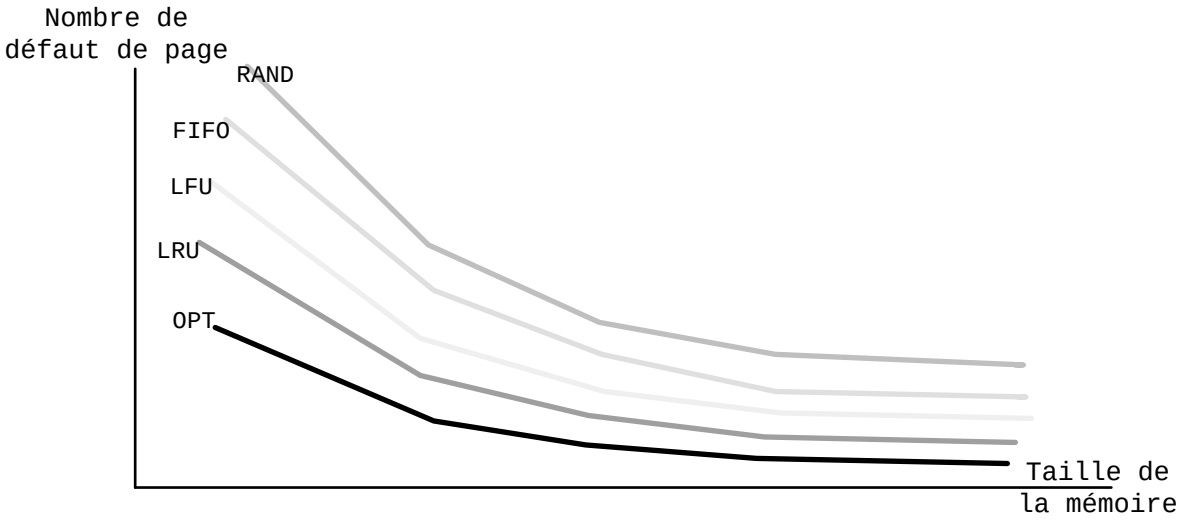

Figure n° 9 : Performances comparées des algorithmes de remplacement de page

3.3 La segmentation

La pagination présente un espace mémoire virtuel monodimensionnel puisque les adresses virtuelles sont continues entre les valeurs 0 et un nombre maximum. Dans beaucoup de cas, il est préférable de disposer de plusieurs espaces mémoires virtuels appelés des segments, qui seront logiquement associés à des entités d'un seul type (on parlera de segment des tables de symbole, de segments procédure, de segments des constantes, etc.) (figure n° 10).

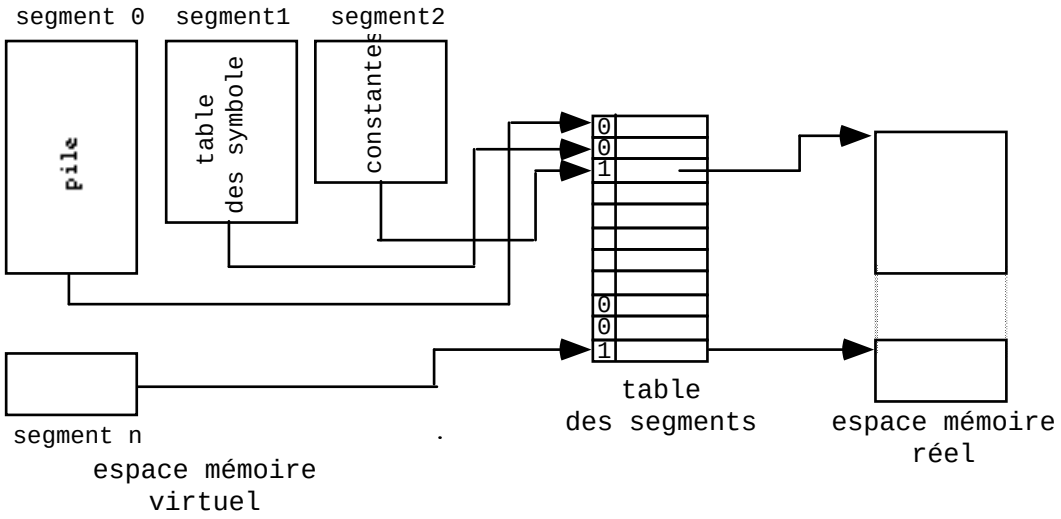

Figure n° 10 : Principe de la segmentation

Un segment est constitué d'une suite linéaire d'adresses virtuelles numérotées de zéro à une valeur maximum, la taille d'un segment variant dans cet intervalle. Chaque segment a donc son propre espace d'adressage dans lequel les adresses virtuelles commencent à zéro et n'ont aucune relation d'ordre d'un segment à l'autre. Pour spécifier une adresse virtuelle dans une mémoire segmentée, il est nécessaire de disposer d'un numéro de segment n et d'un déplacement d dans le segment. Lors de la traduction de l'adresse virtuelle, si le segment référencé n'est pas présent en mémoire, il sera alors chargé depuis la mémoire secondaire.

Contrairement à la pagination, la segmentation de l'espace mémoire réel est une technique qui doit être connue du programmeur et/ou du compilateur. De ce fait, la segmentation assure une grande protection de l'information, permet le partage de données et de procédures (réentrance), simplifie la manipulation des données dont la taille varie dynamiquement, élimine les problèmes d'éditions de liens et de compilation séparée.

Chapitre 5 / La mémoire secondaire

Dans un système d'exploitation à mémoire hiérarchisée, la mémoire secondaire est un organe de stockage de grande capacité et de faible coût, ceci relativement à la mémoire centrale appelée aussi mémoire primaire. Actuellement, la mémoire secondaire est constituée essentiellement de supports magnétiques tels que les disques, les tambours étant devenus obsolètes et les moyens futuristes (disques optiques, disques laser réinscriptibles, etc.) n'étant pas encore mûrs. L'exploitation de cette mémoire secondaire est une des fonctions essentielles d'un système d'exploitation. D'une part le système assure le contrôle du disque et fournit une interface simple à son utilisation, d'autre part il s'occupe de son organisation notamment pour la gestion des fichiers.

1. Etude du disque magnétique

1.1 Structure physique d'un disque

Un disque est composé de pistes concentriques, elles-mêmes divisées en secteurs (figure n° 1). Bien que les secteurs situés à la périphérie du disque soient plus grands que ceux situés au centre, le même nombre d'octets est utilisé pour le stockage de l'information sur tous les secteurs. Chaque secteur est repéré par des marques, le formatage d'un disque consistant à poser ces marques.

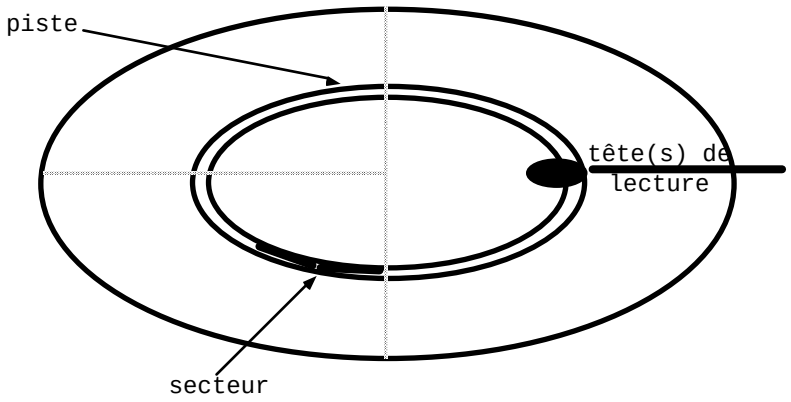

Figure n° 1 : Structure d'un disque

On introduit également les notions de bloc et de cylindre :

- un bloc est la quantité minimale lue ou écrite lors d'un accès au disque ; la taille d'un bloc est généralement de `512` ou `1024` octets,
- un cylindre correspond au nombre de pistes accessibles simultanément lors d'un accès au disque ; un cylindre contient donc autant de pistes que de têtes de lecture/écriture.

Enfin, chaque disque possède un contrôleur de disque assurant le déplacement du bras, la rotation du disque et la commande des têtes, le regroupement des séries de bits lues, etc. Ce contrôleur est un véritable ordinateur avec sa mémoire et ces registres. En outre, de nombreux contrôleur permettent un accès direct à la mémoire, déchargeant ainsi l'unité centrale du transfert de l'information. Le lien entre l'utilisateur et le contrôleur est assuré par le système d'exploitation au moyen d'un logiciel appelé le pilote (driver). Celui-ci traduira les requêtes systèmes en un programme exécuté par le contrôleur.

1.2 Le pilote du disque

L'utilisation d'un disque se fait donc au travers d'un pilote de disque. Celui-ci doit d'une part optimiser les temps d'accès au disque en fonction des demandes, et d'autre part gérer les erreurs logiques ou physiques rencontrées.

1.2.1 Les algorithmes d'ordonnancement du bras

Les temps de lecture ou d'écriture d'un bloc dépendent essentiellement des trois facteurs suivants :

- le temps de recherche ; temps nécessaire pour positionner la tête de lecture sur le bon cylindre,
- le temps de rotation ; temps nécessaire pour positionner la tête de lecture sur le bon secteur,
- le temps de transfert ; temps nécessaire pour transférer le bloc du disque dans la mémoire du périphérique.

En fait, le temps de recherche étant le plus important, sa réduction entraînera une amélioration sensible des performances du système. Aussi, il existe plusieurs algorithmes d'ordonnancement du bras visant à réduire ce temps de recherche (figure n° 2). Les principaux sont :

- FCFS (first come first served) premier arrivé premier servi ; le pilote satisfait les requêtes dans leur ordre d'arrivée. Très simple à mettre en oeuvre, cet algorithme est aussi le moins performant,
- SATF (shortest acces time first) le déplacement le plus court ; le pilote satisfait les requêtes concernant le cylindre le plus près de la position actuelle du bras, puis se déplace, dans un sens ou dans l'autre, vers le cylindre le plus proche comportant des requêtes à satisfaire. Simple à mettre en oeuvre, cet algorithme est généralement deux fois plus performant que le précédent. Toutefois, comme une demande concernant un cylindre éloigné du cylindre courant peut être indéfiniment retardée, si des demandes concernant des cylindres proches arrivent avec une fréquence suffisamment grande, cette stratégie n'est guère utilisée dans la pratique. En effet, de part ce qui précède, on en déduit que le temps moyen d'exécution d'une demande est diminué, alors que le temps maximum est

augmenté. Il n'y a pas équité entre tous les cylindres, ce qui peut aller à l'encontre de certaines règles de priorités établies entre plusieurs processus.

- stratégie de l'ascenseur ; dans cette méthode le bras du disque se déplace dans un sens donné, s'arrête au dessus de chaque cylindre pour lequel des demandes existent et les traite. Lorsque le dernier cylindre comportant une file non vide a été atteint et traité, on change le sens de déplacement et on recommence. Les performances de cet algorithme sont en général moins bonnes que celles de l'algorithme SATF, mais la contrainte d'équité entre les cylindres est maintenant respectée.

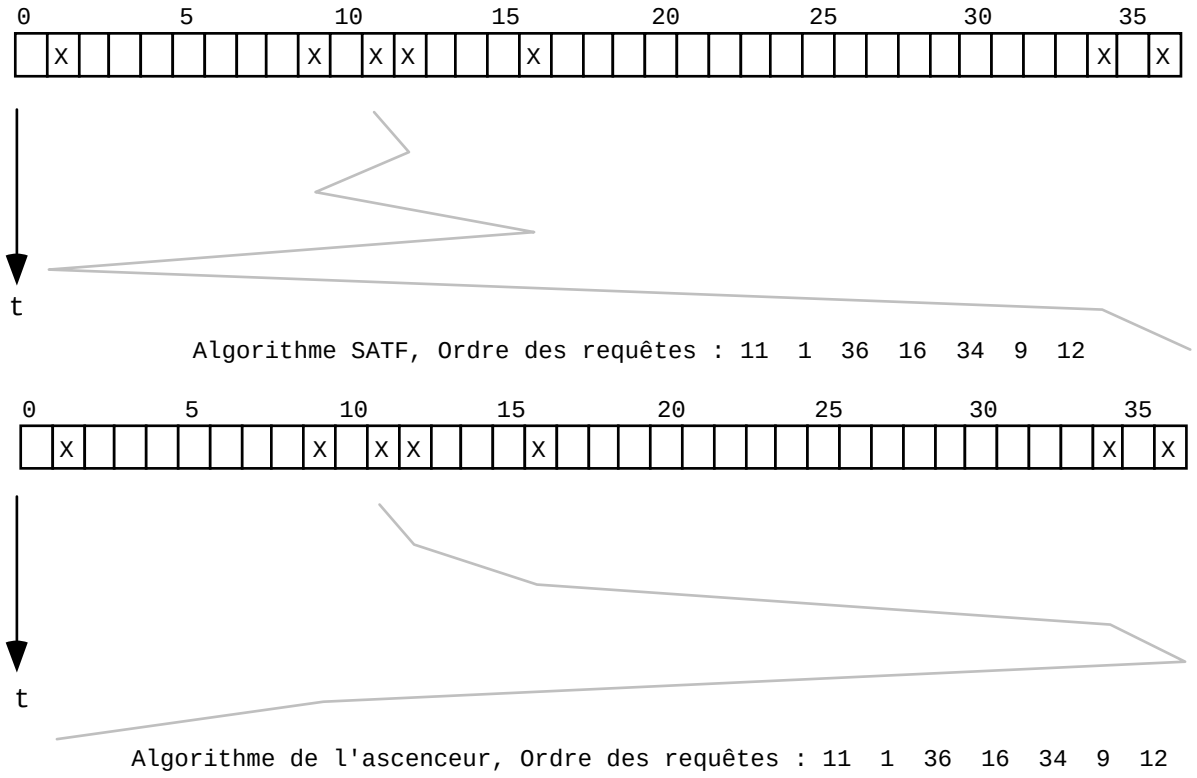

Figure n° 2 : Déplacement comparé du bras du disque
pour différents algorithmes d'ordonnancement

Il est à noter que tous ces algorithmes supposent que le pilote accepte de nouvelles requêtes pendant le traitement d'une première requête. Ceci est rendu possible par l'utilisation d'une table indexée par les numéros de cylindres, où chaque entrée pointe sur le premier élément d'une liste chaînée des différentes requêtes concernant un cylindre. En outre, quelques contrôleurs permettent au pilote de connaître le numéro de secteur situé sous la tête, autorisant alors une autre optimisation : le pilote peut, pour deux requêtes concernant le même cylindre, commencer par celle dont le secteur passera en premier sous la tête.

1.2.2 Le traitement des erreurs

Lors de l'exploitation d'un disque, le pilote doit traiter des erreurs logiques ou physiques. Les plus courantes sont :

- les erreurs de programmation (par exemple la demande d'un bloc qui n'existe pas, le positionnement sur un secteur ou un cylindre inexistant, l'utilisation d'une tête inconnue, etc.) ; en règle générale, le contrôleur vérifie les paramètres fournis par le pilote et ne les accepte pas s'ils sont incorrects. En théorie inexistantes, ces erreurs lorsqu'elles se produisent sont traitées lors de la mise à jour du système.

- les erreurs de total de contrôle ; le contrôleur vérifie par le calcul d'un checksum la validité des informations transférées. Lors de la présence de poussières ou de blocs endommagés, ce calcul est faussé, ce qui entraîne un traitement adéquat. Dans le cas d'une poussière, on répétera l'opération et on indiquera le cas échéant que le bloc est endommagé. Dans le cas d'un bloc endommagé, un traitement spécial et "maison" permet d'éviter ce bloc (liste chaînée des blocs endommagés, pistes supplémentaires accessible au seul système, contrôleurs utilisant des tables de substitution, etc.).

- les erreurs de recherche (positionnement du bras sur le mauvais cylindre) dues aux problèmes mécaniques ; un décalage, entre la position de bras mémorisée par le contrôleur et sa position réelle, peut se produire. Certains contrôleurs traitent l'erreur, d'autres se contentent de la signaler, obligeant ainsi le pilote à effectuer une opération de recalibrage du disque (déplacement du bras à sa position extrême, puis remise à zéro de la position mémorisée du bras).

2. Le système de gestion des fichiers

Le système de fichiers est l'une des parties les plus importantes d'un système d'exploitation (sur un certain système… c'est même pratiquement la seule fonction). Un premier aspect du système de gestion des fichiers est basé sur la notion de fichier et les fonctions qui lui sont associées ; cet aspect est appelé "l'interface utilisateur". Le deuxième aspect du système de gestion des fichiers est l'ensemble des mécanismes qui permettent au système d'établir la correspondance entre ce que manipule l'utilisateur et ce qu'il y a réellement sur le support de mémoire externe ; cet aspect est appelé "l'interface système".

2.1 La notion de fichier

Un fichier est un ensemble de données que l'on stocke et auxquelles on accède de manières diverses au hasard des traitements. Les caractéristiques essentielles d'un fichier sont :

- le stockage d'un volume quelconque d'informations,

- la permanence des données dont la durée de vie n'est pas liée à celle d'un programme,

- l'ordre d'accès aux informations imposé par les nécessités du traitement.

Les opérations possibles sur un fichier sont de nature :

- globale lorsque le fichier est considéré comme un tout ; parmi les opérations les plus courantes sur un fichier on trouve sa création, sa destruction, sa copie, etc.,

- partielle si l'on s'intéresse individuellement à ses éléments ; parmi les opérations les plus courantes sur un élément on trouve sa recherche, sa destruction, sa lecture ou son écriture, etc.

2.1.1 La structure et les modes d'accès d'un fichier

Pour les couches basses du système de gestion des fichiers (celles qui traitent avec le matériel), la structure d'un fichier est un ensemble d'octets ; de par son niveau, cette structure est dite physique. Par opposition, une structure logique d'un fichier a été introduit pour préciser la notion de fichier, vue du coté de l'utilisateur et/ou les couches hautes du système de gestion de fichiers. La structure logique d'un fichier peut être :

- une suite d'octets,

- une suite d'enregistrements de taille fixe, chaque enregistrement étant constitué de champs de nature diverse ; on peut lire ou écrire n'importe quel enregistrement, mais on ne peut pas insérer ou détruire un enregistrement au milieu de la liste,

- une arborescence de blocs du disque, chaque bloc contenant n enregistrements indexés ; l'accès aux enregistrements se fait par une clé et on peut insérer un bloc à n'importe quel niveau de l'arbre.

Les modes d'accès à un fichier dépendent du type du fichier. Pour les fichiers ordinaires, il est possible de réaliser plusieurs types d'accès au fichier :

- l'accès séquentiel ; on accède à une composante à la suite de l'accès à la composante précédente,

- l'accès relatif ou directe ; on accède à une composante quelconque identifiée par son rang dans le fichier,

- l'accès indexé ; on accède à une composante quelconque par la valeur d'un de ces champs.

2.1.2 Les types de fichiers

La plupart des systèmes d'exploitations possède plusieurs types de fichiers. Les principaux types de fichiers sont :

- les fichiers ordinaires ; ils sont constitués par les données et les programmes des utilisateurs,

- les répertoires ; ils constituent le moyen de hiérarchiser logiquement l'ensemble des fichiers. Le contenu d'un répertoire est interprété par un certain nombre de

fonctions du système. Ils permettent d'une part de structurer l'ensemble des fichiers en arborescence et d'autre part définissent un mécanisme de désignation extérieure des fichiers indépendants de leur localisation dans les tables du système et sur le disque.

A noter qu'il existe également d'autres types de fichiers correspondant soit aux périphériques, soit à des moyens de communication entre processus.

2.2 L'organisation du disque et la sauvegarde des fichiers

Cet aspect de l'exploitation d'un disque concernent les couches basses du système de gestion des fichiers. En effet, ici on ne se préoccupe plus de savoir comment un fichier se nomme, mais bien plutôt comment on va organiser le disque et stocker les fichiers afin d'obtenir un fonctionnement fiable et efficace du système de gestion des fichiers.

2.2.1 L'organisation du disque

Il existe deux stratégies pour stocker un fichier de `n` octets :

- on alloue n octets consécutifs sur le disque ; le principal problème de cette méthode est dû aux augmentations de taille du fichier, ce qui oblige à déplacer souvent le fichier de place,

- on divise le fichier en plusieurs blocs pas nécessairement contigus. Le premier problème à se poser est celui du choix de la taille d'un bloc. Ce choix est issu d'un compromis entre la vitesse de transfert et le taux de remplissage souhaité pour le disque ; l'expérience tend à prendre `512` octets, `1K` ou `2K` comme taille d'un bloc. Le deuxième problème qui se pose est celui de la mémorisation des blocs libres. Généralement on utilise une liste chaînée des blocs libres, ou une table de bits (le $i^{ème}$ bit de la table indiquant si le $i^{ème}$ bloc du disque est libre ou non), le choix de l'une ou l'autre des techniques étant principalement lié à la place disponible en mémoire centrale.

2.2.2 Le stockage des fichiers

Un fichier étant constitué d'un certain nombre de blocs, le système doit mémoriser la place des différents blocs le constituant. L'allocation de blocs consécutifs, bien que très simple à mettre en oeuvre, est totalement inadaptée au problème du fait de l'augmentation certaine de la taille des fichiers. Aussi, on utilise plutôt une liste chaînée des blocs constitutifs du fichier, en association ou non avec une table de pointeurs chargée en mémoire centrale.

3. Le système de gestion de fichiers d'UNIX

La notion de fichier sous UNIX ne se limite pas à la notion usuelle du fichier disque. Un fichier est un objet typé sur lequel est défini un ensemble d'opérations. Ainsi, un fichier peut être une suite d'octets sur le disque ou une ressource (physique ou logique) du système comme par exemple un terminal, une imprimante, la mémoire, etc. D'un point de vue interne, à chaque fichier correspond une entrée dans une table contenant ces attributs (par exemple

son type, ses propriétaires, ses droits d'accès, etc.). Une telle entrée est appelée un noeud d'information ou `i-node`.

3.1 La structure arborescente

Grâce à la notion de répertoire, le système de gestion des fichiers d'UNIX est organisé sous la forme d'une arborescence. Cette arborescence repose essentiellement sur l'association faite dans les répertoires entre les noms des fichiers et leur numéro d'index ; la désignation extérieure d'un fichier est réalisée en utilisant un lien associé à ce fichier dans un répertoire, ce répertoire étant lui-même désigné par le même mécanisme dans un autre répertoire. Pour être opérationnelle, cette organisation arborescente suppose une origine symbolique appelée la racine absolue, notée `/` et située à un adresse fixe du disque.

Cette structure arborescente a été normalisée sous UNIX (figure n° 3), et se décompose en l'ensemble des répertoires suivants :

- `/bin` contient la plupart des utilitaires UNIX,
- `/dev` contient les fichiers spéciaux,
- `/etc` contient les programmes et les tables d'administration du système,
- `/lib` contient les bibliothèques utilisées par des langages de programmation,
- `/tmp` est utilisé pour les fichiers temporaires,
- `/usr` contient des répertoires utilisés par le système, plus les répertoires des utilisateurs,
- `/usr/bin` contient d'autres utilitaires UNIX,
- `/usr/lib` contient d'autres bibliothèques utilisées par des langages de programmation,
- `/usr/include` contient les fichiers d'en-tête des langages du système UNIX,
- `/usr/spool` contient les fichiers nécessaire à la gestion des imprimantes.

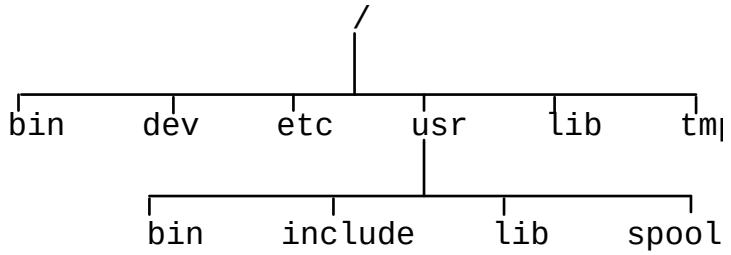

Figure n° 3 : Structure conventionnelle du système de gestion de fichiers d'UNIX

3.2 Les types de fichiers

L'une des caractéristiques du système de gestion de fichiers d'UNIX est la grande variété des types de fichiers. On y retrouve principalement :

- les fichiers ordinaires ; le contenu de ces fichiers est non structuré ce qui fait d'eux des suites d'octets caractérisés uniquement par leur longueur. Aucune interprétation (binaire ou texte) et aucune organisation (séquentielle, directe, indexée) ne sont associées à ces fichiers ; les applications sont donc libres de les traiter comme elles le souhaitent,

- les répertoires,

- les fichiers spéciaux ; ils sont associés à des dispositifs physiques comme les imprimantes, la mémoire, les disques, etc. Ces fichiers sont traités comme les fichiers ordinaires, mais les opérations de lecture ou d'écriture active des dispositifs physiques particuliers. A un niveau plus fin, on distingue les fichiers spéciaux en mode bloc (les entrées-sorties sont réalisées par blocs et transitent par des caches du système), et les fichiers spéciaux caractères (les entrées-sorties sont réalisées caractères par caractères et ne transitent pas par les caches du système).

A ces différents types de fichiers s'ajoutent également les tubes nommés, les sockets et les liens symboliques.

3.3 Organisation du disque

L'exploitation du ou des disques physiques par le système se fait grâce à une organisation logique de l'espace de données de ce(s) disque(s). Un disque physique est ainsi découpé en plusieurs disques logiques répartis dans les deux catégories suivantes :

- les disques de swap ; ils servent à la sauvegarde des contextes de processus ou des pages sorties momentanément de la mémoire.

- les systèmes de gestion de fichiers ; caractérisés par leur type qui est soit `System V`, soit `ffs/ufs` (`BSD`). L'organisation System V est la suivante : un bloc de démarrage utilisé au chargement du système, le `super bloc` contenant des informations générales sur le système (nombre de noeuds alloués et libres, liste de blocs libres, etc.), la table des`i-nodes`, et le bloc des données qui est découpé en blocs logiques allouables de taille `512`, `1024`, `2048` ou `4096` octets (figure n° 4).

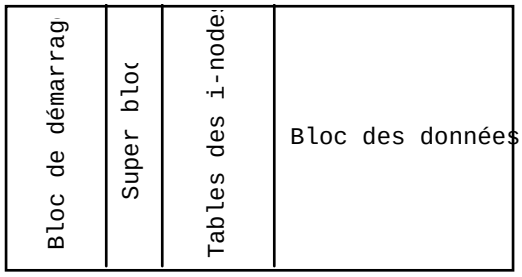

Figure n° 4 : Structure physique du disque sous UNIX système V

3.3.1 Structure d'un noeud d'information

Chaque `i-node` (figure n° 5) est un bloc de `64` octets contenant des informations comme le l'identité du propriétaire, le type et les droits d'accès au noeud pour les différents utilisateurs, le nombre de liens physiques, etc., plus 13 adresses de blocs logiques (10 adresses directes, 1 adresse indirecte simple, 1 adresse indirecte double et 1 adresse indirecte triple).

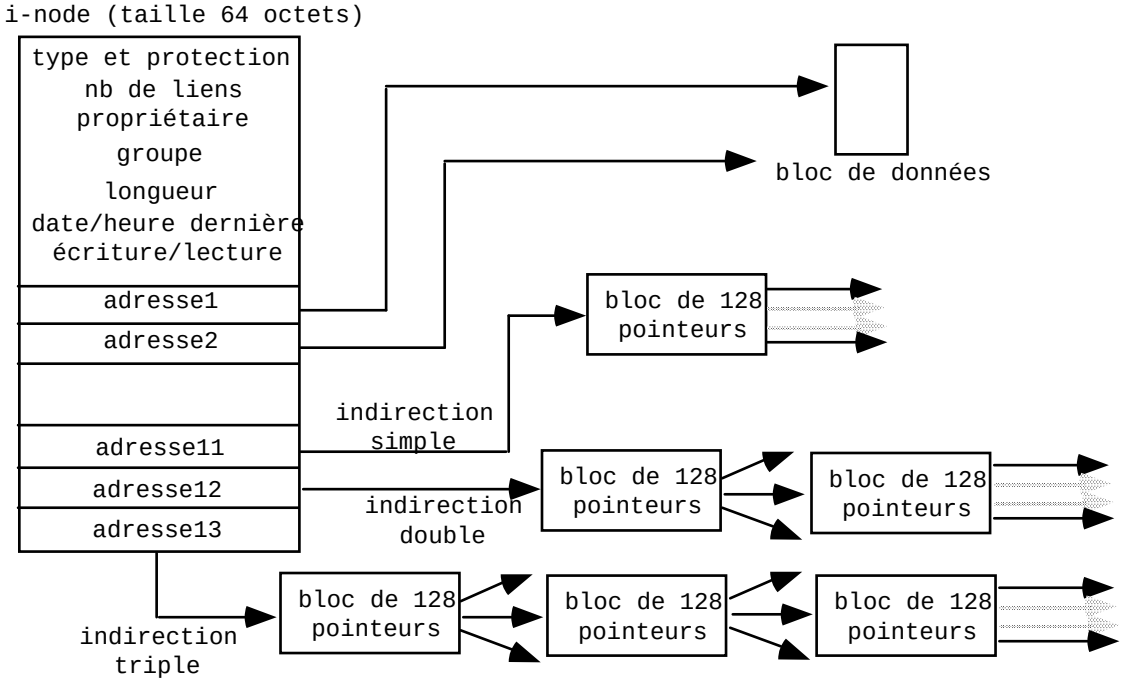

Figure n° 5 : Structure d'un `i-node`

Avec une telle structure, les adresses des fichiers d'au plus dix blocs sont toutes contenues dans le noeud d'information. Lorsque la taille d'un fichier dépasse les 10 blocs, il suffit d'attribuer un nouveau bloc du disque référencé par un pointeur à simple indirection ; pour des tailles de fichier importantes, on utilisera les pointeurs à double ou triple indirection. L'avantage d'un tel mécanisme est que quelque soit la taille du fichier, la taille de son `i-node` est constante. En outre, il ne faut au plus que trois accès disques pour trouver l'adresse de n'importe quel octet d'un fichier.

3.3.2 Structure d'un répertoire

La structure d'un répertoire UNIX est simple. Un répertoire est une table où chaque entrée contient un nom de fichier et le numéro de son noeud d'information. Un répertoire UNIX peut contenir un nombre quelconque de ces entrées codées sur 16 octets. Pour trouver un fichier, il suffit pour le système d'enchaîner la lecture du répertoire et de la table des noeuds.

3.3.3 Structure du système de gestion de fichiers

La gestion des fichiers sous UNIX est principalement réalisée par l'intermédiaire de trois tables (figure n° 6) :

- les tables de descripteurs ; chaque processus en possède une table de descripteur, et la manipulation d'un fichier se fera par un indice (un descripteur) dans cette table. A un descripteur correspond dans la table les attributs dynamiques du fichier (fermeture en cas de recouvrement, etc.), plus un pointeur sur la table des fichiers ouverts. Il est important de se rappeler que les trois premiers indices de cette table sont réservés pour l'entrée standard, la sortie standard et la sortie d'erreur standard. Enfin, un processus acquiert un descripteur soit par héritage (une recopie de la table des descripteurs de son père est réalisée à sa naissance), soit par l'utilisation des primitives `open`, `pipe` et `dup`,

- la table des fichiers ouverts ; pour chaque fichier ouvert, on y trouve en autres choses le nombre de descripteurs associés, le mode d'ouverture, la position courante et un pointeur sur le `i-node` correspondant. Une entrée dans cette table est obtenue lors de l'utilisation des primitives `open` et `pipe`.

- la table des `i-noeuds` présents en mémoire. Pour chaque `i-node` chargé en mémoire, en plus des informations présentes sur le disque, on trouve également dans cette table des informations comme le nombre d'ouvertures sur le fichier, le disque logique auquel appartient le `i-node`, etc.

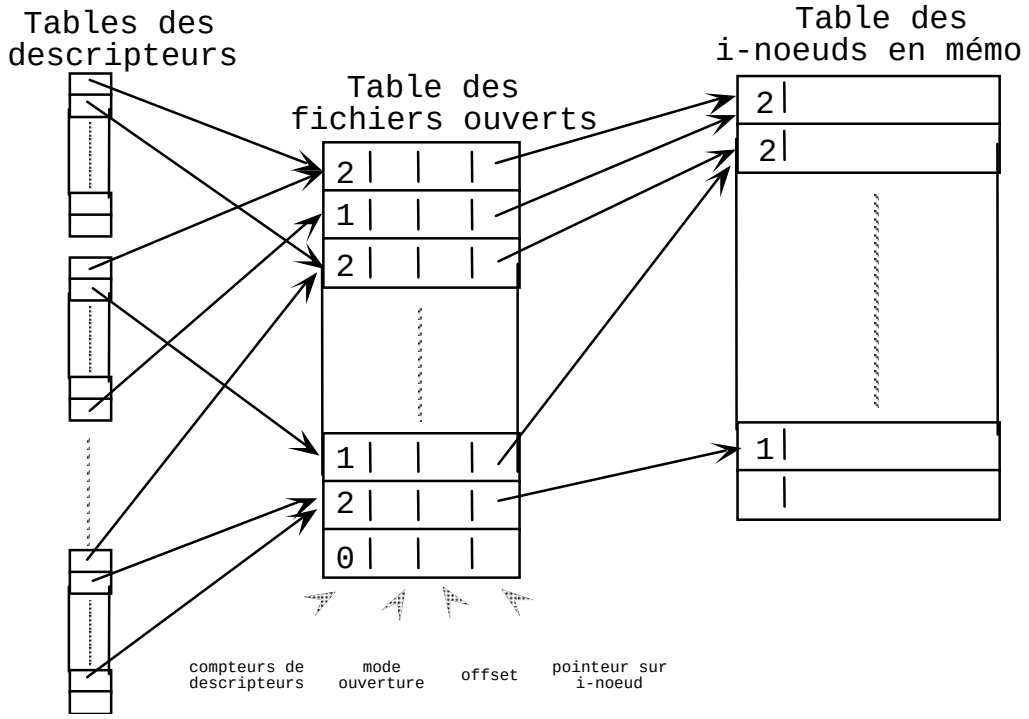

Figure n° 6 : Organisation du système de gestion de fichiers

Bibliographie

1. Systèmes d'exploitation

- Andrew Tanenbaum
  Les Systèmes d'Exploitations
  InterEditions, 1991

- Architecture des Systèmes d'Exploitations
  Michael Griffiths, Michel Vayssade
  Hermes, 1988

2. UNIX

- La programmation sous UNIX
  Jean-Marie Rifflet
  Ediscience, 1993

- Programmer UNIX
  Richard Stoeckel
  Armand Colin, 1992

- UNIX, Initiation et Utilisation
  Jean-Paul Armspach, Pierre Colin
  InterEditions, 1994

- Le système UNIX
  Steve Bourne
  InterEditions, 1985