SYSTEME D'EXPLOITATION

Objectifs :

L'objectif général de ce cours est la description du fonctionnement d'un système d'exploitation. Il introduit les concepts de base d'un système d'exploitation moderne notamment la gestion des processus, la gestion de la mémoire, le système de fichier et leur mise en oeuvre. Tous ces concepts sont illustrés sous le système d'exploitation Linux.

A la fin de ce cours, l'étudiant sera capable de :

- ✓ Comprendre et expliquer comment les tâches s'exécutent sur une machine de façon concurrente sous forme de processus.
- ✓ Décrire et implémenter les différents mécanismes de communication entre processus (signaux, tubes, etc.)
- ✓ Décrire et implémenter les mécanismes de synchronisation entre les processus.
- ✓ Décrire les différents mécanismes de partage du temps processeur aux différents processus d'un système.
- ✓ Décrire les différentes politiques de partage de la mémoire entre processus.
- ✓ Comprendre les différents systèmes de fichiers et leur mise en œuvre.

Références bibliographiques

a. Systèmes d'exploitation. Andrew TANENBAUM, 2è Edition – Pearson Education
b. Operating-System Concepts. Abraham SILBERSCHATZ, 7th Edition

1. Généralités sur les systèmes d'exploitation

1.1. Qu'est qu'un système d'exploitation

La réponse à cette question n'est pas simple. Le système d'exploitation est le logiciel le plus important de la machine, car il doit fournir :

- une gestion des ressources de celle-ci : processeurs, mémoires, disques, horloge, périphériques, communication interprocessus, et inter-machines ;
- une base pour le développement et l'exécution des programmes utilisateurs (applications).

1.2. Problématique

Pour que les programmes s'exécutent efficacement et de façon portable, il doit pouvoir gérer simultanément :

- La multiplicité des ressources matérielles,
- La complexité des composants de chacune d'elles, qui requiert la prise en compte de nombreux détails embêtants, sources de bogues.

1.2.1. Illustration 1 : utilisation d'une imprimante

Une machine multiutilisateur fournit un service d'impression, qui peut être utilisé par n'importe quel programme s'exécutant sur la machine. Pour ce faire, il faut :

- pouvoir verrouiller l'accès à l'imprimante, afin que les flots de caractères produits par les programmes désirant imprimer ne s'entrelacent pas sur le papier ;
- gérer le tampon d'impression, afin que les programmes puissent reprendre leur travail sans avoir à attendre la fin de l'impression ;

  Il s'agit dans ces deux cas, de gérer l'accès à une ressource coûteuse (argent, temps). A tout moment, il faut :
- connaitre l'utilisateur d'une ressource donnée (pour une éventuelle facturation) ;
- gérer l'accès concurrent(iel) à cette ressource ;
- pouvoir accorder l'usage (exclusif) à cette ressource ;
- éviter les conflits entre les programmes ou entre les utilisateurs.

1.2.2. Illustration 2 : le contrôleur de lecteur de disquettes NEC PD765

Ce contrôleur de lecteur de disquettes possède 16 commandes, lancées par l'écriture de 1 à 9 octets dans les registres du contrôleur, et qui permettent d'effectuer des opérations telles que :

- la lecture ou l'écriture d'un secteur ;
- le déplacement du bras de lecture ;
- le formatage de la piste ;
- l'initialisation du contrôleur, la calibration des têtes de lecture, des tests internes …

Ainsi, les commandes de lecture et d'écriture prennent 13 paramètres codés sur 9 octets, codant entre autres :

- le nombre de secteurs par piste ;
- la distance entre deux secteurs ;
- le numéro du secteur sur la piste ;
- le mode d'enregistrement ;
- la méthode de traitement des marques d'effacement …

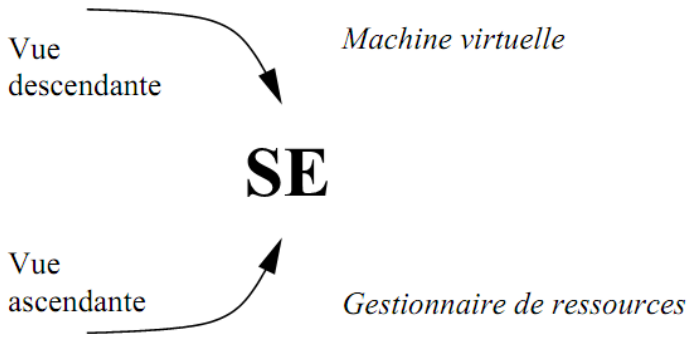
Le contrôleur retourne 23 champs d'état et d'erreur codés sur 7 octets. Il faut gérer soi-même le démarrage et la mise en veille du moteur, en choisissant le meilleur compromis entre le surcoût en temps du démarrage et l'usure des disquettes. Personne ne souhaiterait directement gérer tout cela dans ses programmes applicatifs.

1.3. Les fonctionnalités d'un système d'exploitation

Un système d'exploitation a pour but :

- de décharger le programmeur d'une tâche de programmation énorme et fastidieuse ; et lui permettre de se concentrer sur l'écriture de son application ;
- de protéger le système et ses utilisateurs de fausses manipulations ;
- d'offrir une vue simple, uniforme et cohérente de la machine et de ses ressources.

On peut considérer un système d'exploitation selon deux points de vue, illustré par le schéma ci-dessous :



La machine virtuelle fournit à l'utilisateur :

- une vue uniforme des entrées/sorties ;
- une mémoire virtuelle et partageable ;
- la gestion des fichiers et répertoires (dossiers) ;
- la gestion des droits d'accès, sécurité, et du traitement des erreurs ;
- la gestion des processus ;
- la gestion des communications interprocessus ;

Selon cette vue, le système d'exploitation masque les éléments fastidieux liés au matériel.

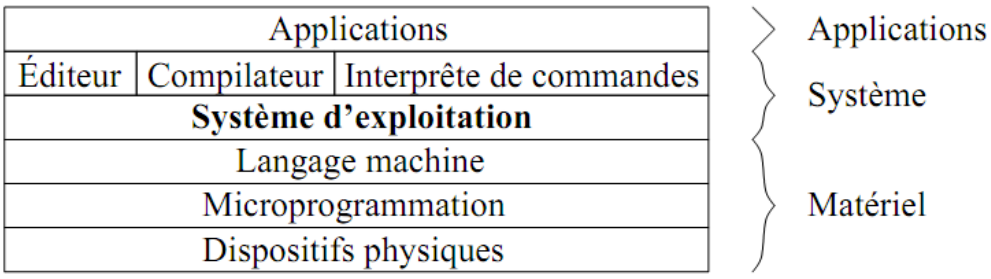
En tant que gestionnaire de ressources, le système d'exploitation doit permettre :

- d'assurer le bon fonctionnement des ressources et le respect des délais ;
- l'identification de l'utilisateur d'une ressource donnée ;
- le contrôle de l'accès aux ressources ;
- l'interruption d'une utilisation de ressource ;
- la gestion des erreurs ;
- l'évitement des conflits.

Ce rôle de gestionnaire de ressources est d'autant plus crucial pour les systèmes d'exploitation manipulant plusieurs tâches en même temps (**multitâches**).

1.4. Place du système d'exploitation dans le système informatique

Le schéma ci-dessous illustre l'interaction entre le système d'exploitation et les autres éléments du système informatique. Il sert d'interface entre les programmes utilisateur et le matériel. Cette interface masque les détails d'implémentations et les opérations réalisées sur le matériel.



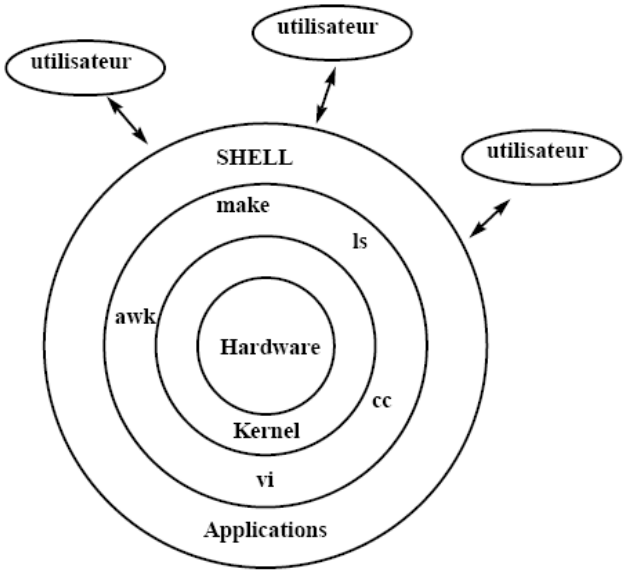
<u>Définition :</u> Il est plus facile de définir un système d'exploitation par ce qu'il fait que par ce qu'il est. – *J.L. Peterson.*

Ne sont pas système d'exploitation :

- L'interpréteur de commandes ;

- Le système de fenêtrage ;
- Les utilitaires (cp, ls, chmod, …)
- Le compilateur,
- Les éditeurs …

<u>Définition :</u> Un système d'exploitation est un ensemble de procédures manuelles et automatiques qui permet à un groupe d'utilisateurs de partager efficacement un ordinateur. – Brinch Hansen.



Parmi les outils (logiciels) présenté par le schéma :

- Le **noyau** (Kernel – anglais) représente les fonctions fondamentales tels que : la gestion de la mémoire, des processus, des fichiers, des entrées/sorties, des fonctionnalités de communication, etc. Très souvent le système d'exploitation est identifié à son noyau, qui s'exécute en **mode privilégié** (encore appelé mode noyau).
- L'interpréteur de commande (Shell – en anglais) permet la communication avec le système d'exploitation grâce à un langage de commande, ceci permet à l'utilisateur de piloter les périphériques en ignorant tout des détails/caractéristiques du matériel qu'il utilise. Celui-ci (ainsi que les outils listés précédemment) s'exécutent en mode non privilégié, car ils n'ont pas besoin d'un accès protégé au matériel.

1.5. Les appels systèmes

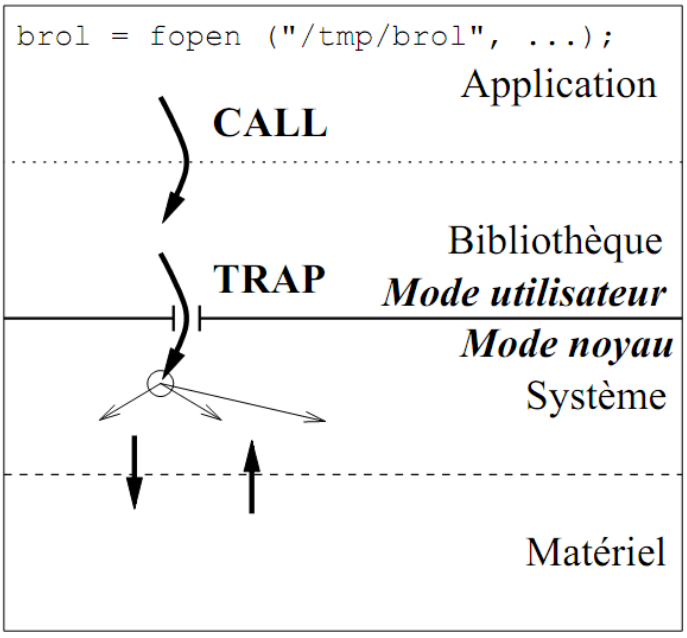
Ils constituent l'interface entre le système d'exploitation et les programmes utilisateurs (ou leurs bibliothèques) qui s'exécutent en mode non privilégié (aussi

appelé pour cette raison mode utilisateur). Les appels systèmes sont réalisés aux moyens d'instructions spécifiques (interruptions logicielles ou *traps*) qui permettent le passage en mode noyau.
L'exécution en mode noyau se différencie de celle du mode utilisateur en effectuant des fonctions supplémentaires :

- Le code et les données utilisées par le système d'exploitation ne sont accessibles qu'en mode noyau. Ainsi les portions de la mémoire les correspondant ne accessible que lors de ce passage en mode noyau ;
- Les opérations de modification de la table des segments mémoire ne sont permises qu'en mode noyau. Ceci empêche tout programme utilisateur de modifier ses droits d'accès à la mémoire.
- Les instructions de lecture et d'écriture sur les ports d'entrée/sortie du matériel ne sont permises qu'en mode noyau. Un programme utilisateur ne peut donc pas directement accéder au matériel sans passer par le système d'exploitation.

NB : Le mode noyau (privilégié) ne doit pas se confondre avec la notion de super-utilisateur qui existe dans certains systèmes d'exploitation, et qui est géré dans le code du système d'exploitation. En réalité même le super-utilisateur passe le plus clair de son temps processeur en mode non-privilégié (utilisateur).

La notion d'interruption ou **trap** permet à tout programme de dérouter son exécution [normale] pour exécuter le code du système d'exploitation.



1.6. Classification des systèmes d'exploitation

Il n'existe pas de système d'exploitation efficace dans tous les contextes d'utilisation. Plusieurs critères permettent de les classer. Ainsi, on peut les organiser suivant :

- le(s) service(s) rendu(s) aux utilisateurs : i.e. le nombre de services et/ou nombre d'utilisateurs pouvant utiliser ces services en même temps.
- l'architecteur de la machine cible : la machine possède un processeur ou plusieurs processeurs.
- L'architecture du système même : les ressources sont-elles toutes sur la machine locale ou bien, peuvent-elles être aussi localisé sur une machine distante ?

Système mono-utilisateur : le système n'accepte qu'un seul utilisateur à un moment donné.
Dans **les systèmes multiutilisateurs** plusieurs utilisateurs peuvent partager en même temps la même ressource matérielle.
Système mono-tâche : Un tel système ne peut réaliser qu'une seule tâche à la fois, c'est lorsque cette dernière est complètement réalisé qu'un autre tâche peut démarrer.
Système multitâches : c'est la possibilité pour le système d'exécuter plusieurs tâches simultanément. Ceci est possible grâce à la notion de processus (threads) qui se partage le temps processeur.

Les systèmes généralistes que nous avons (Windows, Linux, MacOs, etc.) sont des exemples de système multitâches et multiutilisateurs.

Système monoprocesseur : Un seul processeur est utilisé pour exécuter les instructions des programmes. Dans ce genre d'architecture le multitâche est réalisé par pseudo-parallélisme. Le processeur bascule d'une tâche à l'autre très rapidement de façon à donner l'illusion à l'utilisateur de les traiter toutes au même moment (en parallèle).On parle aussi de **multiprogrammation** et de **temps partagé**.

Système multiprocesseur : Le système prend en charge plusieurs processeurs sur la même machine et partage la mémoire entre eux. Dans ce cas la notion de parallélisme est réelle, mais ce type d'architecture soulève les problèmes de gestion de la mémoire.

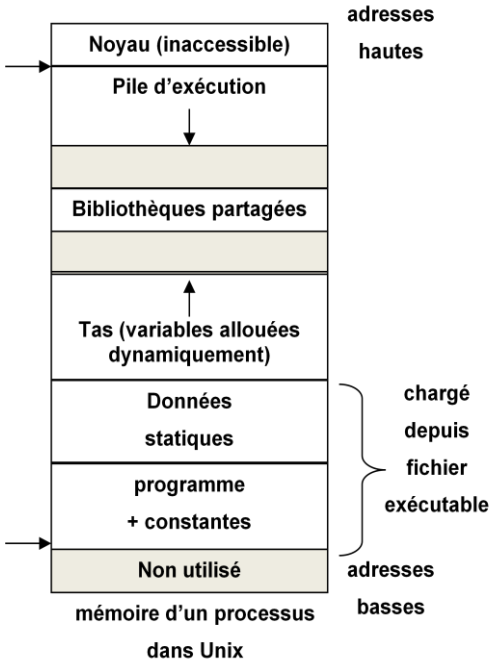
Système centralisé : toutes les ressources gérées sont sur la machine locale.

Système réparties/distribuées : Les ressources sont réparties sur un ensemble de machines et on doit disposer d'une infrastructure réseau pour les accéder. Le système dans ce cas, présente à l'utilisateur l'illusion d'une machine monoprocesseur avec des ressources centralisées. Ce type de système est fortement tolérant aux pannes.

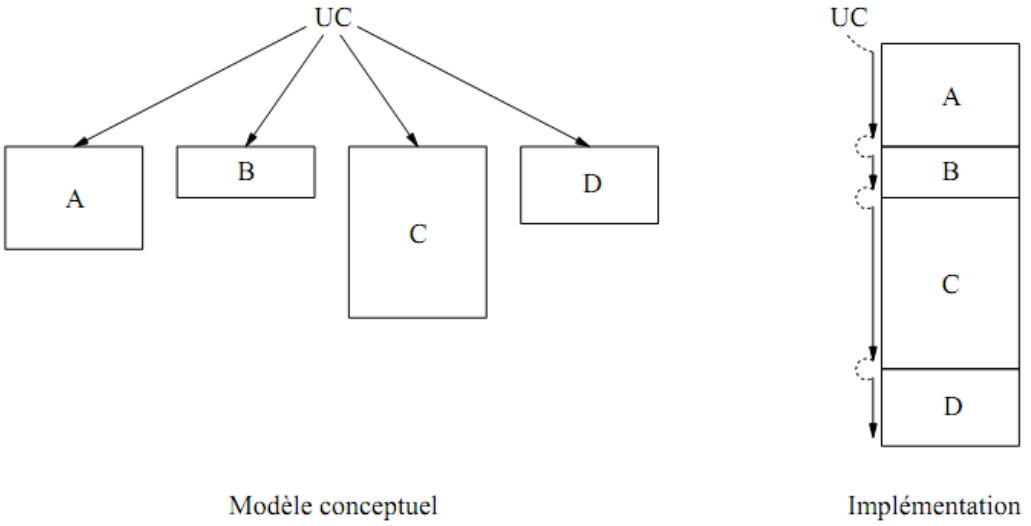
2. Les Processus

2.1. Modèle de processus

La notion de processus est centre de tout système d'exploitation, elle constitue un modèle simple pour représenter l'exécution concurrente de tâches au sein d'un système d'exploitation multitâche. Concrètement, un processus est un programme qui s'exécute, ainsi que ses données, sa pile, son compteur ordinal, son pointeur de pile et les autres contenus de registres nécessaires à son exécution.

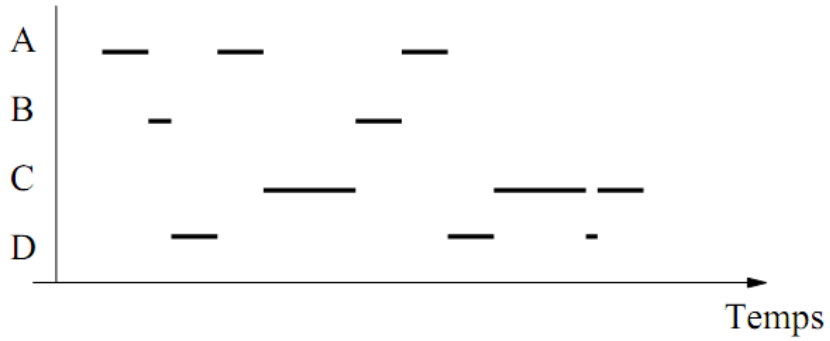


Un processus fournit l'image de l'état d'avancement de l'exécution d'un programme. Par contre, un programme est une suite d'instruction exécutable sur un ordinateur. Un processus est donc un objet dynamique tandis qu'un programme est statique. Le processeur physique commute entre les processus, sous la direction d'un **ordonnanceur**.



Modèle conceptuel Implémentation

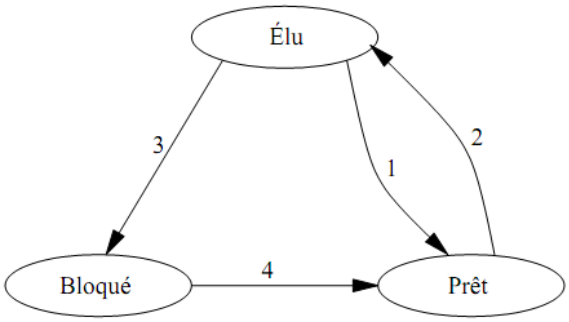
Dans un système à temps partagé, tous les processus progressent dans le temps, mais un seul s'exécute à un moment donné.



2.2. Les états d'un processus

Pendant son cycle de vie, un processus peut à à moment donné être dans l'un des trois états suivants :

- ***élu*** s'il est en cours d'exécution par le processeur. Un processus élu peut être arrêté, même s'il n'a pas achevé son exécution (le système décide d'allouer le processeur à un autre processus). Dans une machine multiprocesseur, plusieurs processus peuvent être élus en même temps, mais il y a toujours autant de processus élus que de processeurs (on peut le voir en tapant la commande Unix ***ps*** et observer le nombre de processus indiqué comme ***running*** à ce moment).
- ***bloqué*** s'il est en attente d'un évènement externe. L'évènement externe attendu est généralement une ressource (frappe clavier, bloc de donnée disque, etc.), lorsque la ressource attendu est disponible, le processus passe à l'état ***prêt.***
- ***prêt*** il a été suspendu provisoirement en faveur d'un autre. Un processus est dans cet état, s'il ne lui manque que la ressource processeur pour s'exécuter.



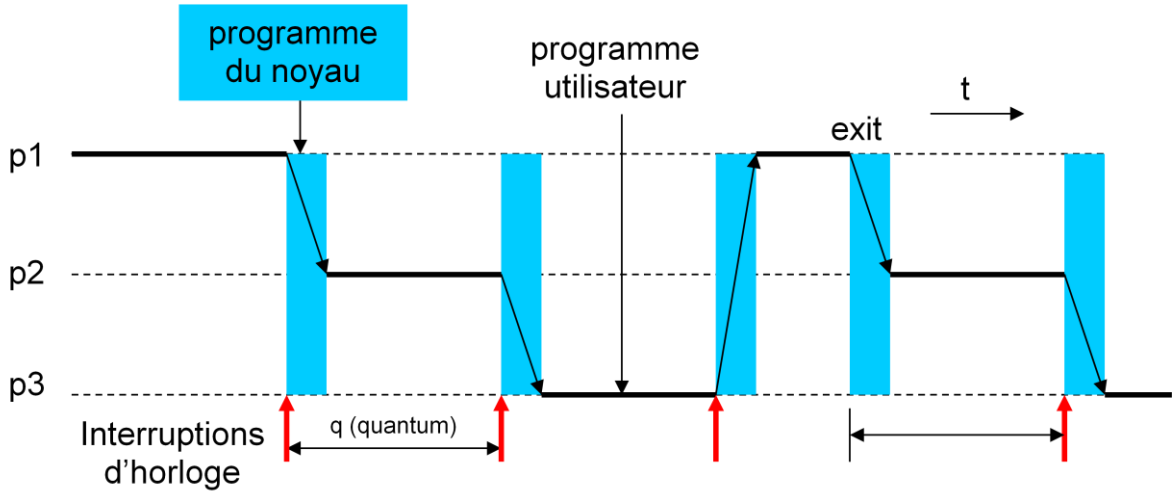
Les transitions entre états sont définies comme suit :

1. le processus a épuisé le ***quantum*** de temps qui lui a été attribué. L'ordonnanceur élit un autre processus parmi les processus prêt pour s'exécuter.

2. L'ordonnanceur a élit ce processus pour s'exécuter.
3. Le processus s'endort en attente d'un évènement.
4. L'évènement attendu par le processus se produit. C'est le système d'exploitation qui gère son traitement (et pas le processus – puisqu'il est bloqué), en interrompant temporairement le processus élu, pour traiter les données reçues et mettre le processus bloqué à l'état prêt.

2.3. Commutation de processus

Le processeur est alloué aux processus prêt pendant une certaine durée appelé ***quantum*** (de l'ordre de quelque millisecondes Ex. 10ms sur certains systèmes)**.** Une fois que ce temps est épuisé une ***interruption horloge*** fait appel au système (***ordonnanceur***) pour sauvegarder le ***contexte*** (contenu des registres du processeur en vue de reprendre plus tard, l'exécution du processus exactement là où il a été interrompu) du processus en cours, puis restaurer le contexte d'un autre processus et lui allouer le processeur. Toutes ces opérations prennent un temps non nul (de l'ordre de 0.5ms)



Pour mettre en œuvre ce modèle, le système dispose d'une table appelée : ***table de processus*** qui n'est jamais ***swappée***. Chaque entrée de cette table correspond à un processus particulier et contient les informations telles que :

- Les valeurs du compteur ordinal, du pointeur de pile, et des autres registres du processeur ;
- Son PID, son état, sa priorité, son vecteur d'interruptions, et les autres informations nécessaires à la gestion des processus par le système ;
- Son occupation mémoire,
- La liste des fichiers ouverts par le processus,

- …, tout ce qui doit être sauvegardé par le système lorsque le processus passe de l'état élu à l'état prêt.

Pour éviter que cette table ne soit trop énorme, toutes les autres informations *non critiques* sont conservées dans une seconde zone, contenue l'espace de chaque processus et est donc *swappable*.

2.4. L'ordonnancement des processus

L'ordonnanceur (*scheduler*) définit l'ordre dans lequel les processus prêt utilise le processeur et la durée de cette utilisation, tout ceci sur la base d'un algorithme appelé *algorithme d'ordonnancement.* Pour les systèmes de *traitement par lots*, l'algorithme d'ordonnancement est relativement simple, puisqu'il consiste à exécuter le programme suivant de la file d'attente (FIFO). Pour les systèmes multiutilisateurs, multitâches, multiprocesseurs, l'algorithme d'ordonnancement peut devenir très complexe.

Le choix d'un algorithme d'ordonnancement dépend de l'utilisation que l'on souhaite faire de la machine, et doit posséder les critères suivants :

- équité : chaque processus doit pouvoir disposer de la ressource processeur ;
- efficacité : l'utilisation du processeur doit être maximale (il doit travailler à 100% de son temps) ;
- temps de réponse : il faut minimiser le temps de réponse pour les utilisateurs interactifs ;
- temps d'exécution : il faut minimiser le temps d'exécution de chaque travail effectué en traitement par lots ;
- rendement : le nombre de travaux réalisés par unité de temps doit être maximal ;

Certains de ces critères sont contradictoires, et il a été montré [Kleinock 1975] que tout algorithme d'ordonnancement qui favorise une catégorie de travaux le fait au détriment des autres.

2.4.1. Ordonnancement circulaire ou tourniquet (round robin)

Il s'agit de l'un des mécanismes d'ordonnancement les plus simples et des plus robuste. Il consiste à attribuer à chaque processus un quantum de temps pendant lequel il a le droit de s'exécuter. Si le processus s'exécute jusqu'à épuisement de son quantum, le processeur est réquisitionné par l'ordonnanceur et attribué à un autre processus. Si en revanche, le processus se bloque (***attente passive***) ou se termine avant la fin de son quantum, le processeur est immédiatement attribué à un autre processus.

La principale question de ce type d'ordonnancement réside au choix de la durée du quantum par rapport à celle du changement de contexte. S'il est trop petit, ceci réduit l'efficacité du processeur, s'il est trop grand cela réduit considérablement l'interactivité du système lorsque de nombreux processus sont en attente d'exécution.

2.4.2. Ordonnancement avec priorité

Avec l'ordonnancement précédent, on suppose que tous les processus sont d'égales importances, ce qui n'est pas vrai dans la pratique. L'ordonnancement avec priorité va essayer de favoriser certaines classes de processus par rapport à d'autres en associant à chaque processus un numéro de priorité. Ainsi l'ordonnanceur lance le processus prêt de priorité la plus élevée. Pour éviter que le processus de plus grande priorité ne s'accapare le processus, l'ordonnanceur abaisse à chaque interruption d'horloge la priorité du processus actif. Si sa priorité devient inférieure à celle du deuxième processus de priorité élevé, la commutation a lieu.

On peut allouer dynamiquement la priorité aux processus, en allouant une priorité élevée aux processus effectuant beaucoup d'E/S, l'empêchant ainsi d'occuper la mémoire trop longtemps. Une méthode pour atteindre cet objectif consistera à donner comme priorité à un processus, l'inverse de la fraction de *quantum* précédemment utilisé. Ainsi, un processus ayant utilisé ***2ms*** d'un quantum de ***100ms***, aura une priorité de *50* ; contre 2 pour un autre processus ayant utilisé 25ms de son *quantum*.

Les processus de même priorité sont classés dans une file d'attente, il en résulte plusieurs files d'attente selon les priorités. Chaque file d'attente est gérée suivant l'algorithme du *tourniquet*. S'il n'y avait pas une évolution dynamique des priorités, les processus de faible priorité ne seraient jamais élus, conduisant à une situation de *famine* (privation de ressources).

2.4.3. Ordonnancement du plus court d'abord

Ce type d'ordonnancement s'applique lorsqu'on dispose d'un ensemble de tâches dont on peut connaitre la durée à l'avance, comme par exemple dans le cas d'un traitement par lots de transactions journalières bancaires. Si la file d'attente contient plusieurs tâches de même priorité, on minimise le temps de réponse en effectuant toujours celle ayant le temps le plus court.
En effet, si l'on dispose de n tâches de durée $t_1 \leq t_2 \leq \cdots \leq t_n$, que l'on ordonne dans un certain ordre $i, j, \cdots, k$, le temps de réponse moyen est :

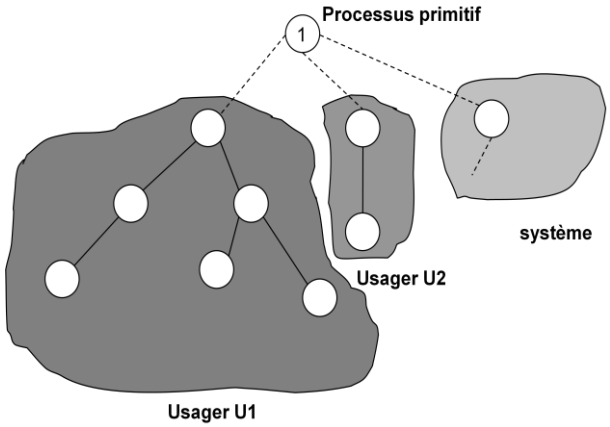
$$T = \frac{t_i + (t_i + t_j) + \cdots + (t_i + t_j + \cdots + t_k)}{n} = \frac{nt_i + (n-1)t_j + \cdots + t_k}{n}$$

La contribution de plus grand poids étant celle de t_i, on minimisera T en prenant $t_i = t_1$, puis $t_j = t_2$, et ainsi de suite …

2.5. Relation entre processus

Trois opérations de base peuvent être réalisées sur un processus : sa création, la mise en pause et sa terminaison.

La création du processus est réalisé par un appel système : la commande `pid_t pid = fork()` (qui retourne un identifiant de processus). Cette création est obligatoirement faite par un autre processus qui de ce fait est appelé processus parent. Ainsi un processus n'a qu'un seul parent, et peut avoir plusieurs ou 0 enfant. Ceci définit une arborescence ou hiérarchie de processus. Dans les systèmes de type Unix, tous les processus dérive d'un ancêtre commun : le processus ***init*** qui lui est crée au démarrage du système.



1 programme,2 processus

```
if (fork() != 0) {
printf("je suis le père, mon PID est %d\n", getpid());
} else {
printf("je suis le fils, mon PID est %d\n", getpid());
/* en général exec (exécution d'un nouveau programme) */
}
...
}
```

NB : le processus père se distingue du processus fils par le résultat retourné par l'appel système `fork()` :

- Pour le père : il est renvoyé l'ID du fils ou bien –1 si l'opération se déroule mal.
- Chez le fils : la commande retourne toujours 0.

La terminaison d'un processus peut se faire de deux façons :

- Autodestruction : le processus se termine lui-même (commande `exit(statut)`). La variable statut vaut 0 si tous se passe bien, sinon un code correspond à l'erreur est retournée.
- Destruction par un autre processus (commande `kill(pid_t)`).

NB : Certains processus ne se termine jamais, on parle alors de ***démons.***

Un processus peut être mis en attente par l'appel système `sleep(int d)`, où d est la durée de l'attente en seconde.

Un processus peut attendre la fin de l'exécution de son fils par les commandes :

- `pid_t wait(int *ptrStatut)` : la variable `ptrStatut` recueille le statut de terminaison du fils, `wait` retourne le `PID` du fils.
- `pid_t waitpid (pid_t pid, int *ptrStatut)`: est utilisé pour attendre la fin de l'exécution d'un fils particulier.

2.6. Processus et gestion des interruptions

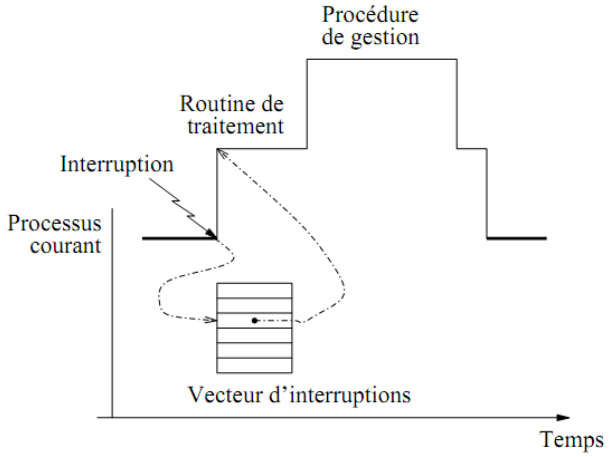
Le traitement d'une interruption par le système s'effectue en appelant une ***routine*** de traitement associée à cette interruption, dont l'adresse est stockée dans les cases d'un tableau indexé par le type d'interruption : ***le vecteur d'interruption.***

Numéro d'interruption	Gestionnaire
0	Horloge
1	Disque
2	(Pseudo-)terminaux (`tty`)
3	Autres périphériques
4	Logiciel (trap)
5	Autres

Des numéros d'interruptions différents sont couramment affectés aux différents périphériques du système, et permettront d'accéder aux routines de traitement à travers le vecteur d'interruption.

Le traitement d'une interruption s'effectue par le système de la façon suivante :

- le processeur sauvegarde la valeur de son compteur ordinal, détermine le type d'interruption, passe en mode noyau et charge la nouvelle valeur du compteur ordinal à partir du vecteur d'interruption ;
- la routine de traitement de l'interruption sauvegarde les autres registres du processeur puis appelle la procédure principale de gestion de l'interruption.
- au retour de la procédure de gestion, la routine de traitement restaure les registres processeur, puis recharge la valeur du compteur ordinal.



3. Communication interprocessus.

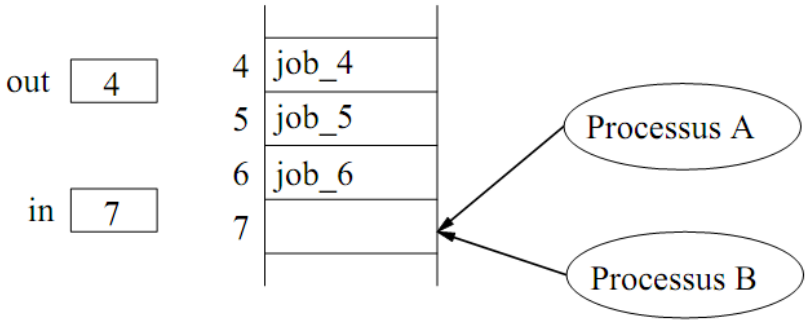
Les processus d'un système ne s'exécutent pas toujours de façon isolée. Certains processus ont besoin de ***coopérer***, et pour cela ils nécessitent des moyens de communications et de synchronisation. Parfois les processus peuvent se trouver en ***compétition*** pour l'utilisation d'une ressource du système. Le partage de la ressource peut nécessiter un accès exclusif, soit à cause de la nature physique de la ressource (*non partageable*), soit parce que les opérations sur cette ressource peuvent provoquer des incohérences ou des interblocages.

On peut rencontrer essentiellement trois problèmes, lorsque les processus sont appelé à compétir ou à coopérer :

- Comment un processus fait-il pour passer des informations à un autre processus ?
- Comment peut-on assurer que deux processus ou plus, ne produisent pas de conflits lorsqu'ils s'engagent dans des activités critiques (partage d'une ressource exclusive) ?
- Comment ordonner des activités d'un ensemble de processus en cas de dépendance ?

3.1. Conditions de la concurrence : illustration.

Pour illustrer le problème de concurrence, considérons le problème de sérialisation des travaux d'impression : ***le spoule d'impression***. Lorsqu'un processus veut imprimer un fichier, il entre son nom dans un répertoire spécial de spoule. Un autre processus, le ***démon d'impression***, regarde périodiquement s'il y a des fichiers à imprimer, si c'est le cas il les imprime et supprime leurs noms du répertoire. Supposons que le répertoire de spoule est un tableau et possède un très grand nombre d'entrées, numérotées 0, 1, 2, … chacune pouvant accueillir un nom de fichier. On suppose qu'il existe deux variables partagées par tous les processus : *in* et *out* qui pointe respectivement sur la prochaine entrée libre et sur le prochain numéro de fichier à imprimer. A un instant donné, les entrées 0 à 3 sont vides (fichiers déjà imprimés) et les entrées 4 à 6 sont pleines (fichiers à imprimer). On a donc : *in = 7* et *out = 4*.



Si presqu'au même moment deux processus A et B souhaite lancer l'impression d'un fichier, ils doivent chacun exécuter une séquence semblable à :

```
next_free_slot = in;
placer_fichier(next_free_slot);
in = next_free_slot + 1;
```

Le processus A lit la valeur de *in* et stocke la valeur 7 dans sa variable locale `next_free_slot.` A ce moment là, une interruption se produit, le système jugeant que ce processus (A) a bénéficié de suffisamment de temps processeur passe la main à B. Le processus lit également la valeur de *in* qui n'a pas pu être modifié par A. B continue son exécution, il place le nom du fichier qu'il veut imprimer dans la prochaine entrée disponible qui est 7 selon lui, et actualise la variable *in* qui passe à 8.

Finalement, quand le processus A s'exécute à nouveau, il prend les choses là où il les avait laissées. Il examine `next_free_slot` et place donc son nom de fichier dans l'emplacement 7, écrasant ainsi ce que B venait d'y placer.

- En interne le répertoire du spoule d'impression reste cohérent, le démon d'impression ne remarquera donc rien.
- Le processus B par contre ne verra jamais son fichier imprimé.

On appelle ***conditions de concurrence*** une situation où deux ou plusieurs processus lisent et écrivent des données partagées, et où le résultat final dépend de l'ordre dans lequel les processus sont ordonnancés.

3.2. Sections critiques

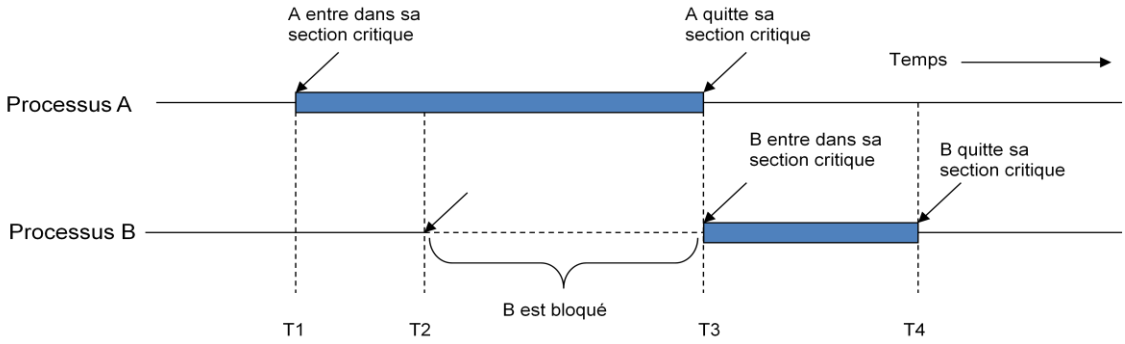
La solution au problème identifié précédemment consiste à interdire la modification de données partagées à plus d'un processus à la fois. , c'est-à-dire définir un mécanisme d'***exclusion mutuelle*** sur des portions spécifiques du code appelées ***sections critiques.*** L'ensemble des instructions d'une section critique devraient être exécuté de façon indivisible (***atomique***).

On peut formuler quatre conditions pour garantir un *bon comportement* d'exclusion mutuelle :

1. Il faut toujours s'assurer que deux processus n'entre ***jamais en même temps*** en section critique sur une même ressource.
2. Aucun hypothèse n'est faite sur la vitesse relative d'exécution des processus, ni sur le nombre de processeurs.
3. aucun processus suspendu en dehors d'une section critique ne peut bloquer les autres.

4. aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

La première condition est suffisante à elle seule pour éviter les conflits d'accès. Cependant, elle ne suffit pas à garantir le bon fonctionnement du système, en particulier en ce qui concerne l'égalité d'accès aux sections critiques.



Il existe de nombreux mécanisme pour mettre en œuvre l'exclusion mutuelle, chacun présentant des avantages et des inconvénients.

3.2.1. Le masquage des interruptions

Il s'agit du moyen le plus simple pour éviter l'accès concurrent à une ressource. Il consiste à désactiver toutes les interruptions avant d'entrer dans une section critique, et à les restaurer à la sortie de celle-ci. Ainsi, le processus en section critique ne peut être interrompu au profit d'un autre processus, puisque le masquage des interruptions empêchera l'ordonnanceur de s'exécuter.

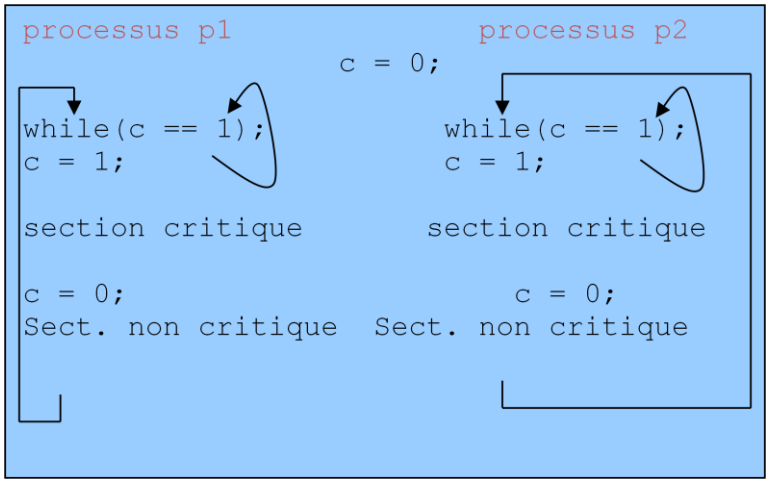
Cette solution est assez dangereuse car elle autorise un processus utilisateur à masquer les interruptions, ce qui fait qu'un processus peut s'accaparer le processeur pour une longue durée sans en être inquiété. De plus, le système peut être bloqué, si le processus *oublie* de restaurer les interruptions.

Cependant, cette solution est très souvent utilisée par le système d'exploitation lui-même pour manipuler ses structures internes, quoique inapproprié pour les processus utilisateur.

3.2.2. Attente active avec variable de verrouillage

Avec la solution du masquage des interruptions, lorsqu'un processus entre en section critique, tous les autres processus sont bloqués, même si la section critique ne concerne qu'un seul autre processus. Il faut donc pouvoir définir autant de sections critiques indépendantes que nécessaire. Pour cela, on déclare une variable *verrou* par section critique. La variable est mise à 1 par le processus entrant dans la section critique considérée. La variable est remise à 0 par le processus lorsqu'il quitte la

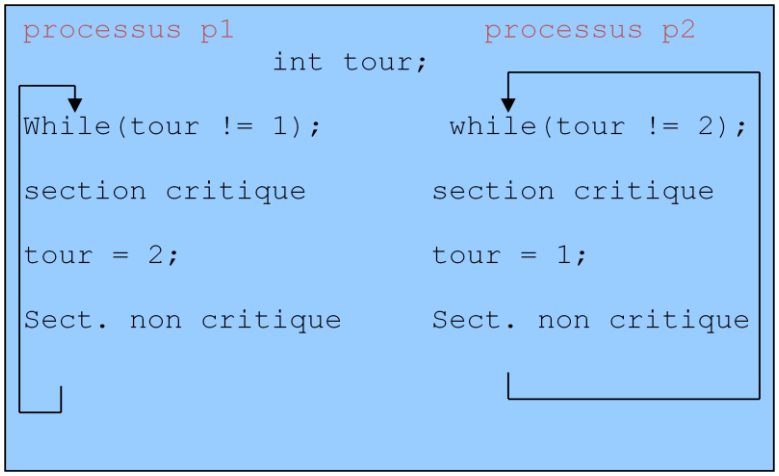
section critique. Si la variable est déjà positionnée à 1 lorsqu'un autre processus veut entrer dans cette section critique, celui-ci fait une attente active par bouclage.



Cette solution pose le même problème que celui identifié à la section 3.1, les deux processus peuvent entrer en section critique en même temps. De plus, lorsque les choses se déroulent bien (pas d'interruption du processus juste à la sortie du *while*), une consommation inutile du temps processeur en résulte, car le processus en attente d'entrée en section critique boucle sans rien faire.

3.2.3. Attente active avec variable d'alternance

Dans la solution précédent, les deux processus modifie la variable *verrou* en même temps ce qui conduit à une incohérence. Dans cette solution, on assure l'exclusion mutuelle dans ce sens que deux processus ne peuvent pas se trouver en section critique en même temps. La variable *tour* définit quel processus a le droit d'entrer en section critique à un moment donné.



Elle ne respecte toutefois pas les deuxième et troisième règles des sections critiques. En effet, lorsqu'un processus quitte la section critique, il s'interdit d'y revenir avant

que l'autre processus n'y soit entré à son tour. Si les vitesses des deux processus sont très différentes, le plus rapide attendra toujours le plus lent.

3.2.4. La solution de Peterson

Cette solution améliore la précédente, en permettant au processus dont ce n'est pas le tour de rentrer quand même en section critique, lorsque l'autre processus n'est pas encore prêt à le faire. Cette solution est basée sur deux primitives : ***entrer_region*** et ***quitter_region*** qui entoure la section critique.

```
int tour ;                /* variable de tour */
int intéresse [2]         /*Drapeaux */

void entrer_region (int p){
     intéresse [p] = TRUE ;   /* p veut entrer ...*/
     tour = p ;               /* est passé en dernier */
     while ((tour == p)       /* Attendre son tour */
                    && (intéresse [1-p])) ;
}

void quitter_region (int p){
     intéresse [p] = FALSE ;
}
```

Si l'un des deux processus désire entrer et que l'autre n'est pas intéressé à entrer, le premier processus entre immédiatement en section critique. Tant que l'un des processus est en section critique, l'autre ne peut y entrer. Si les deux processus sont en compétition pour entrer en section critique, le dernier à vouloir entrer écrase la valeur de la variable *tour* positionnée par le premier, et n'entre dons pas en section critique.

3.2.5. L'instruction TSL

Il s'agit d'une solution au problème d'exclusion mutuelle basée sur un support matériel, car c'est une instruction machine. L'instruction TSL (*Test and Set Lock*) charge le contenu d'un mot mémoire dans un registre, puis met une valeur non nulle à cet emplacement. Les deux opérations se réalisent de façon indivisible. Pour garantir cette atomicité, le processeur qui l'exécute verrouille le bus de données (*lock*), pour empêcher les autres processeurs d'accéder à la mémoire pendant la durée de l'opération.

En se basant sur cette instruction, on peut mettre en œuvre une exclusion mutuelle, en se servant d'une variable *verrou* initialement positionnée à 0, comme illustré par le langage assembleur ci-dessous :

```
entrer_region :                          quitter_region :
     TSL REG, $verrou                         MOVE  $verrou, #0
     CMP REG, #0                              RET
     JNE entrer_region
     RET
```

Lorsque le *verrou* est à 0, n'importe quel processus peut le positionner à 1. Ensuite, l'ancienne valeur du *verrou* est comparée à 0. Si elle est différente de 0, cela signifie que le *verrou* était déjà positionné ; le programme retourne simplement au début et refait le test. *Verrou* finira bien par être à 0, lorsque le processus actuellement en section critique en aura terminé. Pour supprimer le *verrou* le processus stocke un 0 dans celui-ci.
Cette solution est valable et efficace, même dans le cas des systèmes multiprocesseurs.

3.2.6. La solution avec sommeil et activation

La solution de Peterson et celle du TSL sont bonnes, mais elle souffre du problème d'attente active, et donc consommatrice du temps processeur. D'autre solution plus efficace, bloque le processus en attente d'entrer en section critique au lieu de consommer le temps processeur.
Les primitives *sleep* et *wakeup* sont deux appels systèmes. La première endort le processus qui l'évoque en rendant la main à l'ordonnanceur, alors que la deuxième permet de réveiller le processus endormi dont l'identifiant est passé en paramètre.

3.2.6.1. Modèle producteur-consommateur

C'est un modèle de communication entre deux processus où ces derniers se partagent une zone de mémoire commune de taille fixe. Le producteur doit pouvoir ranger dans la zone commune des données qu'il produit en attendant que le consommateur soit prêt à les récupère. Le consommateur ne peut consommer des données inexistantes.

Hypothèses de base :

1. Les données sont de taille constante.
2. Les vitesses respectives des deux processus (producteur et consommateur) sont quelconques.

Règle 1 : Le producteur ne peut plus ranger des données si le tampon es ***plein***.
Règle 2 : Le consommateur ne peut pas prendre de données si le tampon est ***vide***.
Règle 3 : Exclusion mutuelle au niveau de l'objet, le consommateur ne peut prendre un objet que le producteur est entrain de produire.

Règle 4 : si le producteur (resp. le consommateur) est en attente parce que le tampon est plein (reps. vide), il doit être averti dès que cette condition cesse d'être vraie.

```
#define  N 100           /* taille de la mémoire */
int compteur = 0 ;       /* nombre d'objet courant */

void producteur (void){  /* processus producteur */
  int item ;
  while(TRUE){                /* répéter en continu */
    item = produire_objet() ; /* générer un élément */
    if (compteur == N)        /* si le tampon est plein */
     sleep() ;                /* s'endormir ... */
    mettre_objet(item) ;      /* placer l'élément ... */
    compteur ++ ;             /* un objet de plus */
    if (compteur == 1)        /* le tampon était vide ? */
     wakeup(consommateur) ;   /* réveiller le consommateur */
  }
}

void consommateur (void){     /* processus consommateur */
  int item ;
  while (TRUE){               /* répéter en continu */
    if (compteur == 0)        /* si le tampon est vide ... */
     sleep() ;                /* s'endormir ... */
    item = retirer_objet() ;  /* retirer un élément ...*/
    compteur -- ;             /* un objet de moins */
    if (compteur = N-1)       /* le tampon était-il plein ? */
     wakeup(producteur) ;     /* réveiller le producteur */
    consommer_objet(item) ;   /* consommer l'objet */
  }
}
```

3.2.6.2. Discussion

Cette solution est élégante, mais elle conduit aux mêmes conflits d'accès que ceux décrit à la section 3.1, du fait de l'accès concurrent à la variable *compteur*. Pour l'illustrer considérons la situation suivante : le tampon est vide, et le consommateur vient de lire la valeur 0 dans la variable compteur ; l'ordonnanceur l'interrompt et donne la main au producteur, qui ajoute un nouvel objet dans le tampon. Il incrémente le compteur, et appelle *wakeup* pour réveiller le consommateur. Comme ce dernier n'est pas encore endormi, le *wakeup* est sans effet. L'ordonnanceur donne ensuite la main au consommateur, celui continu où il s'était arrêté … il s'endort.

Comme le producteur n'appellera plus la routine *wakeup*, et que le consommateur est endormi, le producteur finira par remplir le tampon et s'endormira à son tour ; et donc les deux processus dormiront pour toujours.

Ici encore, le problème réside dans le fait de la non-atomicité du test et de l'action qui lui est associée. Une solution simple consiste à conserver un ***bit d'attente d'éveil***. Lorsqu'un *wakeup* est envoyé à un processus non encore endormi, ce bit est positionné. Plus tard quand ce processus quand ce processus essayera de se mettre en sommeil, il testera d'abord la valeur ce bit. Il ne s'endormira que si ce bit est à 0, s'il est à 1, il le mettra à 0 et ne s'endormira plus.

3.2.7. Les sémaphores

Le problème de synchronisation de plusieurs processus a considérablement évolué avec l'invention des ***sémaphores*** par Dijkstra en 1965. Il s'agit d'un outil puissant, facile à implémenter et à [utiliser ?]. Un *sémaphore* est une structure de données à deux champs :

- Une variable entière encore appelée valeur du sémaphore. Le *sémaphore* sera dit ***binaire*** si sa valeur ne peut être que 0 et 1, il est dit ***général*** sinon.
- Une file d'attente de processus [endormi] sur ce sémaphore.

Le *sémaphore* est manipulé par deux primitives :

a. **Down**

Cette opération teste la valeur du sémaphore, si elle est supérieure à 0, sa valeur est décrémentée, et l'activité du processus peut continuer. Si la valeur du sémaphore est nulle, le processus qui a fait appel à la primitive *down* est mis en sommeil sans que le down ne se termine pour l'instant. Les activités de vérification, de modification et de mise en sommeil éventuel sont toutes effectuées de façon ***atomique*****.** On garanti que, une fois que le sémaphore a démarré, aucun autre processus ne peut y accéder tant que l'opération n'est pas terminée ou bloquée.

b. **Up**

Cette primitive incrémente la valeur du sémaphore si aucun processus n'est en attente sur celui-ci. Si un ou plusieurs processus sont en attente sur ce sémaphore, l'un d'entre eux (choisi au hasard) par le système est réveillé pour terminé son *down*. Ainsi, une fois un *up* accompli sur un sémaphore contenant des processus en sommeil, le sémaphore sera toujours à 0, mais il contiendra un processus en sommeil en moins. L'opération d'incrémentation du sémaphore et du réveil du processus est également indivisible.

NB : Un *sémaphore binaire* est utilisé communément pour réaliser l'*exclusion mutuelle* entre plusieurs processus. Il suffit pour cela d'effectuer un *down* juste avant d'entrer en *section critique* et un *up* juste à sortir de cette section.

3.2.7.1. Application des sémaphores : producteur-consommateur

```
#define  N 100                  /* taille de la mémoire */
semaphore mutex ;               /* sémaphore d'exclusion mutuelle */
semaphore vide ;                /* sémaphore nombre de place vide */
semaphore plein ;               /* sémaphore nombre de place occupée*/

sem_init(mutex, 1) ;            /* un seul processus en sect. crit. */
sem_init(vide, N) ;             /* le tampon est complètement vide */
sem_init(plein, 0) ;            /* aucun emplacement occupé ... */

void producteur (void){         /* processus producteur */
  while(TRUE){                  /* répéter en continu */
    item = produire_objet() ;   /* produire un élément */
    down(vide) ;                /* y a-t-il de la place ? */
    down(mutex) ;               /* on verrouille le tampon */
    mettre_objet(item) ;        /* placer l'élément ... */
    up(mutex) ;                 /* déverrouiller le tampon */
    up(plein) ;                 /* un objet est à prendre */
  }
}

void consommateur (void){       /* processus consommateur */
  while (TRUE){                 /* répéter en continu */
    down(plein) ;               /* attente d'un objet ... */
    down(mutex) ;               /* on verrouille le tampon */
    item = retirer_objet() ;    /* retirer un élément ...*/
    up(mutex) ;                 /* déverrouiller le tampon */
    up(vide) ;                  /* une place libérée */
    consommer_objet(item) ;     /* consommer l'objet */
  }
}
```

L'utilisation des sémaphores, bien qu'émanant une simplification conceptuelle de la gestion des exclusions mutuelles, peut s'avérer fastidieuse et propice aux erreurs. Ainsi, si l'on inverse par mégarde les deux *down* dans le code du consommateur, et si le tampon est vide, le consommateur se bloque sur le sémaphore *plein* après avoir positionné le sémaphore *mutex,* et le producteur se bloque sur son appel *mutex* : on arrive à ***une situation d'interblocage.***

3.2.8. Les moniteurs

Hoare (1974) et Brinch Hansen (1975) ont introduit l'idée de ***moniteur*** pour proposer des mécanismes de la synchronisation au niveau du langage (de haut niveau). Un moniteur est constitué d'un ensemble de procédure et de données regroupées dans un module ayant la propriété que ***seul un processus*** peut être actif dans le moniteur à un instant donné.

C'est le compilateur qui garantit l'exclusion mutuelle au sein du moniteur (grâce à la mise en œuvre d'un sémaphore binaire par exemple). Il suffit alors au programmeur de convertir les *sections critiques* en procédure du moniteur, un fois ceci fait, deux processus ne pourront jamais exécuter leurs *sections critiques* en même temps.

Même si le moniteur garantit l'exclusion mutuelle de ses processus, un autre mécanisme est nécessaire pour bloquer un processus entré dans le moniteur lorsqu'il n'est pas à mesurer de poursuivre son exécution (car en attente d'un ressource par exemple). Ceci permettra à un autre processus du moniteur de s'exécuter, et lui apporter la ressource dont il a besoin pour poursuivre son exécution.

Cette fonctionnalité est assurée par des variables dites ***de condition,*** qui sont en quelque sorte des sémaphores auxquelles sont associées les primitives ***wait*** et ***signal***. La primitive *wait* bloque le processus appelant sur la variable de condition passée en paramètre, alors que la primitive *signal* permet à l'un des processus bloqué sur la variable de condition de se réveiller.

```
moniteur Producteur_Consommateur
     condition full, empty ;
     integer count ;
     procedure mettre(item :integer) ;
          begin
               if count = N then wait(full) ;
               mettre_objet(item);
               count:= count +1;
               if count = 1 then signal(empty)
          end
     fonction retirer : integer ;
          begin
               if count = 0 then wait(empty) ;
               retirer := retirer_objet;
               count := count - 1;
               if count = N - 1 then signal(full)
          end
     count := 0 ;
end moniteur
```

```
procedure producteur ;
  begin
     while true do
       begin
          item := produire_objet ;
          Producteur_Consommateur.mettre(item) ;
       end
  end

procedure consommateur;
  begin
     while true do
       begin
          item := Producteur_Consommateur.retirer(item) ;
          consommer_objet(item) ;
       end
  end
```

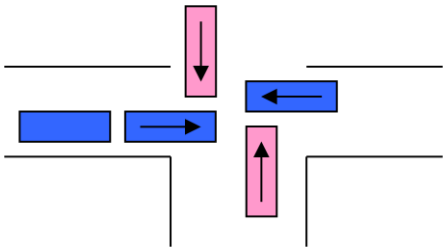
Les primitives *wait* et *signal* ressemblent beaucoup aux primitives *sleep* et *wakeup*. Cependant, elles ne présentent pas les mêmes défauts, puisque leur utilisation exclusive à l'intérieur du moniteur garantit l'exclusion mutuelle, et annule tout risque d'exécution concurrente que l'on a rencontré avec *sleep* et *wakeup*.

3.3. Les interblocages

3.3.1. Caractérisation

On appelle *interblocage* (*deadlock*) une situation dans laquelle plusieurs processus (au moins 2) sont bloqués, car chacun d'eux atteint un évènement que seul un autre processus de l'ensemble peut provoquer. Comme tous les processus attendent, aucun d'eux ne pourra jamais produire l'évènement attendu par l'(les) autre(s) et tous les processus attendront indéfiniment. On ne peut résoudre une situation d'interblocage sans une intervention extérieure.

Exemple d'interblocage : embouteillage dans un carrefour routier.



Pour qu'un interblocage se produise les quatre conditions doivent être vérifiées :

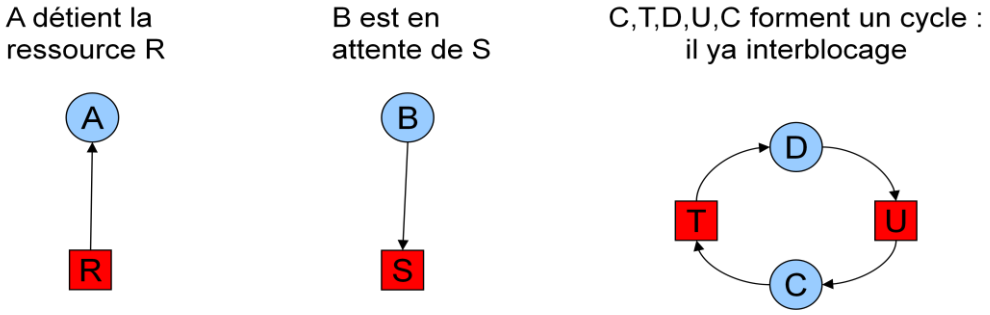
1. *exclusion mutuelle* : chaque ressource est soit disponible, soit attribuée à un seul processus.
2. *détention et attente :* un processus ayant obtenu des ressources peut en demander de nouvelles.
3. *pas de réquisition :* les ressources déjà obtenues par un processus ne peuvent être retirées de force.
4. *attente circulaire :* il doit y avoir un cycle mettant en cause au moins deux processus, chacun attendant une ressource détenue par l'autre.

3.3.2. Modélisation

Les interblocages peuvent être modélisés par un graphe orienté possédant deux types de nœuds :

- les carrés : représentent des ressources
- les cercles : représentant les processus.

Un arc allant d'une ressource (carré) vers un processus (cercle) signifie que le processus a demandé et obtenu cette ressource. Par contre, un arc allant d'un processus (cercle) vers une ressource (carré) signifie que le processus est bloqué en attente de cette ressource.



3.3.3. Réactions façon aux interblocages

On peut réagir en général de quatre façons face aux interblocages :

1. Ignorer le problème : c'est encore appelé la politique de l'Autriche.
2. Détecter les interblocages y remédier : graphe de ressource et recherche de cycle dans le graphe. En cas d'interblocage on peut revenir en arrière (*rollback*) ou supprimer certains processus.
3. Les éviter de manière dynamique : faire des réservations globales (en bloc) ou avec ***précaution*** de ressource. Inconvénient : limitation du parallélisme des applications.
4. Les prévenir en empêchant l'apparition d'une des quatre conditions de leur existence.

3.4. Applications : Quelques problèmes classiques

3.4.1. Le diner des philosophes

Il s'agit d'un problème classique de *synchronisation* posé et résolu par Dijkstra [1965] qui modélise des processus qui entre en concurrence pour un accès exclusif à un nombre limité de ressources.

Cinq philosophes sont assis autour d'une table ronde. Chacun a devant lui une assiette de spaghettis. Mais ceux-ci sont si glissants qu'il faut deux fourchettes pour les manger. Entre deux assiettes se trouve une fourchette. L'activité d'un philosophe est partagée entre manger et penser. Quand un philosophe a faim, il tente de prendre les deux fourchettes encadrant son assiette. S'il y parvient, il mange, puis il se remet à penser. Comment écrire un algorithme qui permette à chaque philosophe d'exercer ses activités sans que les philosophes se retrouvent en situation d'interblocage ?

Solution naïve:

```
#define N 5                     /* nombre de philosophes */
void philosophe(int i)          /* le philosophe n°i */
{
   while(TRUE){
     penser();                  /* le philosophe pense */
     prendre_fourchette(i);     /* il prend sa fourchette gauche */
     prendre_fourchette((i+1)%N);/* il prend sa fourchette droite */
     manger();                  /* il mange des spaghettis */
     poser_fourchette(i);       /* il pose sa fourchette gauche */
     poser_fourchette((i+1)%N); /* il pose sa fourchette droite */
   }
}
```

Cette solution, d'apparence évidente ne fonctionne pas. Car les philosophes peuvent prendre tous leur fourchette gauche en même temps. Aucun ne pourra prendre sa fourchette de droite, il y aura un interblocage.

Algorithme déterministe et correct

```
#define N 5                     /* nombre de philosophes */
#define GAUCHE (i+N-1)%N        /* n° du voisin à gauche de i */
#define GAUCHE (i+1)%N          /* n° du voisin à droite de i */
#define PENSE 0                 /* le philosophe pense */
#define FAIM 1 /* le philosophe essai de prendre les fourchettes */
#define MANGE 2                 /* le philosophe mange */

int etat[N]                     /* suivre l'état des philosophes */
semaphore mutex ;               /* exclusion mutuelle */
semaphore s[N] ;                /* un sémaphore par philosophe */
```

```
void philosophe(int i)        /* le philosophe n°i */
{
   while(TRUE){
     penser();                /* le philosophe pense */
     prendre_fourchettes(i);/* prendre 2 fourchettes ou bloque */
     manger();                /* il mange des spaghettis */
     poser_fourchettes(i);    /* déposer ses 2 fourchettes */
   }
}

void prendre_fourchettes(int i)
{
   down(mutex) ;              /* entrer en section critique */
   etat[i] = FAIM ;      /* le philosophe a faim */
   test(i) ;                  /* tente de prendre 2 fourchettes */
   up(mutex) ;                /* quitter la section critique */
   down(s[i]) ;               /* bloque s'il n'a pas pu prendre
                              les 2 fourchettes */
}

void poser_fourchettes(int i)
{
   down(mutex) ;              /* entrer en section critique */
   etat[i] = PENSE ;          /* le philosophe a fini de manger */
   test(GAUCHE) ;        /* proposer au philosophe de gauche */
   test(DROITE) ;        /* proposer au philosophe de droite */
   up(mutex) ;                /* quitter la section critique */
}

void test(int i)
{
   if ((etat[i] == FAIM)  &&    /* si i demande les fourchettes */
       (etat[GAUCHE] != MANGE) && /* et qu'elles sont libres */
       (etat[DROITE] != MANGE)){ /* toutes les deux */
     etat[i] = MANGE ;  /* récupère implicitement les fourchettes */
     up(s[i]) ;         /* réveille le philosophe i s'il dormait */
   }
}
```

3.4.2. Le problème des lecteurs et des rédacteurs

Il s'agit d'un autre problème classique qui modélise l'accès à d'une base de données, où plusieurs processus peuvent lire simultanément la base, mais un seul processus peut y écrire à la fois. Une solution correcte, déterministe et efficace qui donne la priorité aux lecteurs est la suivante :

```
semaphore mutex ;         /* exclusion mutuelle à reader */
semaphore bd ;            /* contrôle l'accès à la base */
int reader = 0 ;          /* nombre de lecteurs potentiels */

void lecteur(void)
{
  down(mutex) ;           /* section critique */
  reader ++ ;             /* un lecteur de plus ... */
  if (reader == 1)        /* si c'est le premier lecteur ... */
     down(bd) ;           /* ... il verrouille la base */
  up(mutex) ;             /* sortie de la section critique */
  lire_base() ;
  down(mutex) ;           /* récupère l'accès exclusif à reader */
  reader -- ;             /* et un lecteur de moins */
  if (reader == 0)        /* si c'est le dernier lecteur ... */
     up(bd) ;             /* ... il déverrouille la base */
  up(mutex) ;
  utiliser_donnees_lues() ;
}

void redacteur(void)
{
  down(bd) ;              /* accès exclusif à la base */
  ecriture_base() ;
  up(bd) ;                /* libérer l'accès exclusif */
}
```

3.5. Exercices/Recherches personnelles

- Communication interprocessus par des signaux.
- Communication interprocessus par des tubes non nommés
- Communication interprocessus par des tubes nommés

4. La gestion de la mémoire

La mémoire constitue une ressource critique pour le système d'exploitation, dont le mauvais usage peut avoir des effets dramatiques sur les performances globales du système. Dans un système multiprogrammé, plusieurs processus sont en mémoire en même temps. La façon la plus simple d'allouer de la mémoire à un processus, consiste à lui réserver des cases contiguës. Si le nombre de tâches devient élevé (de façon à ne pas tenir dans la mémoire principale), pour satisfaire au **principe d'équité** et **minimiser le temps de réponse** des processus, il va falloir ***simuler*** la présence en mémoire centrale de tous les processus. Pour ce faire on fera usage d'un *va et vient* ou ***recouvrement (swapping)*** des processus qui consiste à en stocker temporairement certains dans l'espace disque.

Les fonctionnalités d'un *bon gestionnaire* de mémoire sont :

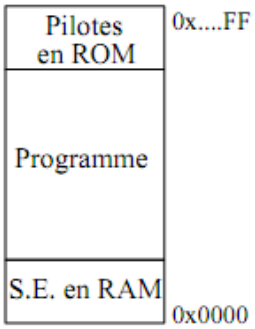
- connaître quelles sont les parties libres de la mémoire physique ;
- allouer de la mémoire aux processus, en évitant autant que possible le gaspillage ;
- récupérer la mémoire libérée par un processus (à sa terminaison) ;
- offrir aux processus l'illusion d'un mémoire plus grande que la mémoire physique, en gérant les recouvrements et la pagination avec le disque (*mémoire virtuelle*).

4.1. Gestion de la mémoire sans recouvrement ni pagination.

C'est le mécanisme de gestion le plus simple, car l'on ne considère que la mémoire physique effectivement disponible. On accède à celle-ci directement par les adresses physiques.

4.1.1. Monoprogrammation

En *monoprogrammation*, il n'y a qu'un seul processus (le système d'exploitation lui-même) en mémoire à un moment donné, qui peut utiliser l'ensemble de la mémoire physique disponible. Dans la pratique, dans ce type de système, une partie du système d'adressage est dédiée au système d'exploitation, qui se compose des fonctions de démarrage et de gestion de bas niveau (BIOS : initialement en ROM) et l'interpréteur de commandes (qui fait partie du système d'exploitation).



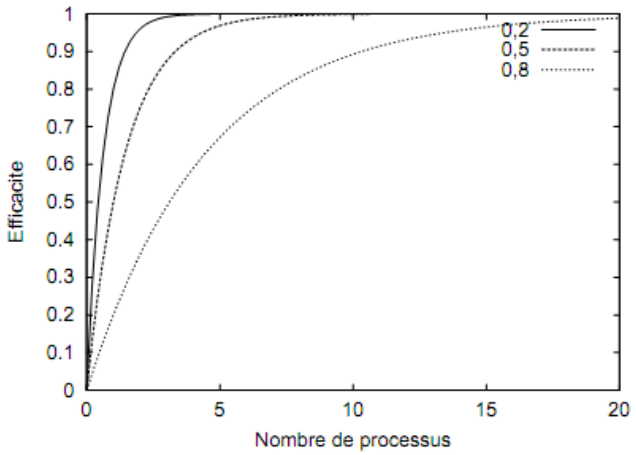
A la fin de chaque programme, l'interpréteur de commande demande le prochain programme à lancer. Cette technique qui a tendance à disparaitre peut être illustrée par les PC sous l'ancien système MS-DOS.

4.1.2. Multiprogrammation

Aujourd'hui, la *multiprogrammation* est de plus en plus utilisée dans les ordinateurs. Cette technique permet de subdiviser un gros programme en processus indépendants, et à plusieurs utilisateurs de travailler en temps partagé sur la même machine.

En général, un processus passe une bonne partie de son temps à faire des entrées/sortie (lecture/écriture dans le disque, attente de la saisie d'une donnée du clavier etc.). Si on suppose par exemple que les processus occupe le processeur 20% de son temps (calcul utile) alors il faudra lancer 5 processus au minimum, pour avoir une change d'utiliser pleinement le processeur (dans ce cas les 5 processus sont supposés ne jamais être en phase d'entrées/sortie simultanément). On peut alors modéliser ce comportement de façon probabiliste :

Soit p la fraction de temps passé par un processus en attente d'E/S. Si n processus sont chargés en mémoire, la probabilité qu'il soit simultanément en attente est p^n ,ce qui fait un taux d'utilisation du processeur de $1 - p^n$.

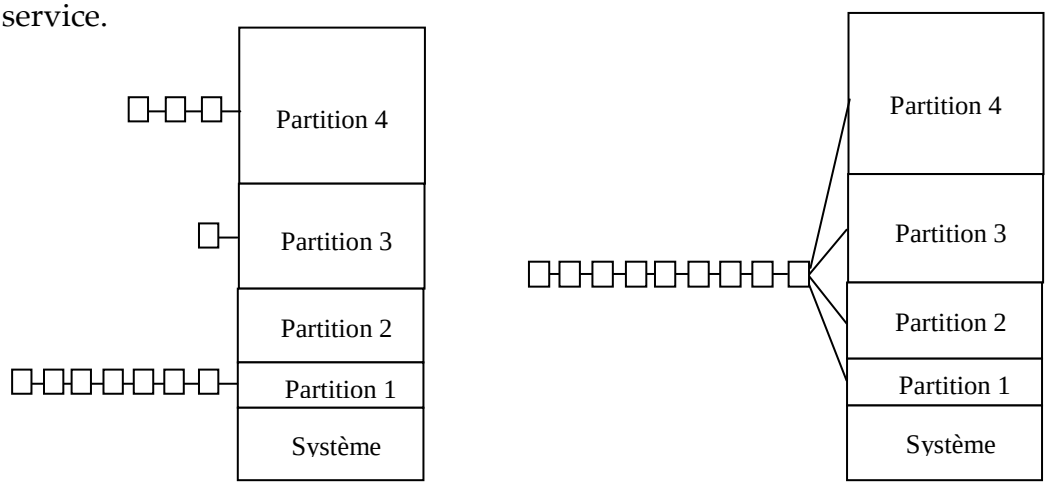


La courbe modélisant ce comporte est présentée ci-dessus. Si ***n*** est le nombre de processus indépendant en mémoire centrale, *plus le nombre* ***n*** *est élevé, plus le taux d'utilisation (efficacité) du processeur l'est aussi.* On a donc intérêt à augmenter la taille de la mémoire centrale.

A titre d'exemple, avec 1Mo de mémoire, un système d'exploitation de 200Ko, et des processus de 200Ko passant 80% de leur temps en attente, on peut alors loger 4 processus en mémoire. : on obtient ainsi un taux d'utilisation de 60%. En doublant la mémoire (2Mo), on peut loger désormais 9 processus et l'efficacité du processeur passe alors à 87%. Avec un troisième Mo supplémentaire on passe à 96%.

4.2. Multiprogrammation avec partition fixes

La principale question lorsqu'on fait usage de la multiprogrammation est celle de savoir comment organiser la mémoire de la façon la plus efficace possible. Un moyen consiste à diviser la mémoire en *partitions* généralement de taille inégales (MTF : *Multiprogramming with a Fixed number of Tasks*). Dès qu'une nouvelle tâche est lancée, elle est placée dans la file d'attente de la file d'attente de la plus petite partition pouvant la contenir, l'espace restant est alors perdu. En procédant ainsi, certaines partitions peuvent rester inutilisées, car leur file d'attente est vide et les tâches en attente pour une partition donnée doivent attendre (même des partitions plus grandes sont vides). Une alternative consiste à utiliser une file d'attente unique : dès qu'une partition se libère, on y place la plus grande tâche de la file d'attente qui peut y tenir. Mais cette stratégie a l'inconvénient de défavoriser les tâches de petite taille (généralement interactives) – bien qu'il serait souhaitable de leur offrir le meilleur service.



Pour éviter ce phénomène de famine, une autre solution consiste à dire qu'un tâche ne peut pas être ignorée plus de *k* fois. Chaque fois qu'il est laissé de côté son score augmente d'un point. Quand il est passé à *k*, la tâche ne peut plus être ignorée.

4.3. Code translatable et mécanisme de protection

En multiprogrammation, un processus peut être chargé n'importe où en mémoire. Les adresses dans le code du processus sont générées au moment de la compilation par l'éditeur des liens, mais du au fait qu'on ne connaît pas l'emplacement exacte du processus lorsqu'il sera exécuté, elles sont fausses et le programme ne peut directement être exécuté. Comment résoudre ce problème de correspondant entre l'adresse dans le code du processus et l'adresse physique réelle ? Une solution a ce problème consiste à générer les adresses relatives pour le processus pendant l'édition des liens, et utiliser un registre particulier (***registre de base***) qui conservera chaque

fois qu'un processus sera chargé en mémoire, l'*adresse de début de la partition* (ou bloc mémoire). L'adresse physique sera alors déterminée par la formule suivante :
adresse_physique = adresse_base (contenu du registre) + adresse_relative (dans le processus)

Cependant cette solution n'empêche pas les tentatives d'accès (lecture/écriture) de cette zone mémoire par les autres processus. Comment donc protéger la zone mémoire appartenant à un processus donné ?

Solution :

Une première solution a ce problème, fut proposé sur les IBM360. Elle consiste à diviser la mémoire en blocs de 2Ko et d'associer à chacun des blocs mémoire une clé à 4bits. Pendant l'exécution, le mot d'état du processeur (*PSW : Program Status Word*) contient un champ de 4bits, différent à pour chacun processus exécuté. Le processeur ne peut alors accéder qu'aux blocs mémoire dont la clé est la même que celle du *PSW*. Avec ce mécanisme on ne peut avoir qu'au plus 16 processus en mémoire simultanément. Seul le système d'exploitation peut modifier les codes des blocs et celui du *PSW* et ce en mode privilégié.

Une autre solution, plus générique à ces deux problèmes consiste à doter le processeur de deux registres spéciaux accessibles uniquement en mode noyau : le *registre de base et le registre de limite.* La valeur du registre de base est implicitement ajoutée pendant l'exécution à toutes les adresses générée par le processeur et elle est comparée avec la valeur du registre de limite : les valeurs d'adresse inférieures ou supérieure à limite déclenche une exception. Avec cette solution, un programme peut être déplacé en mémoire de façon transparente, seuls les registres de base et de limite seront modifiés pour matérialiser ce déplacement. Le principal inconvénient de cette solution est qu'il faut effectuer deux opérations à chaque accès de la mémoire (une addition et une comparaison).

4.4. Le va-et-vient

Dès que le nombre de processus devient supérieur au nombre de partitions en mémoire, il faut pouvoir simuler la présence en mémoire de tous les processus pour pouvoir satisfaire au *principe d'équité* tout en minimisant le temps de réponse des processus. La technique de va-et-vient (ou recouvrement – ***swapping*** en anglais) permet de conserver temporairement sur le disque des images de processus afin de libérer la mémoire centrale pour d'autres processus.

4.4.1. La multiprogrammation avec partitions variables

Utiliser le va-et-vient avec une gestion de la mémoire par partitions de taille fixe conduit à un gaspillage de celle-ci, puisque que la taille des programmes peut être

bien plus petite que celle des partitions qui les abritent. Ceci rend nécessaire la possibilité de gérer la mémoire avec des partitions de taille variable. Ainsi, aux cours du temps, le nombre, la position et la taille des partitions varient, en fonction des processus en mémoire. On améliore ainsi grandement l'utilisation de la mémoire, en rendant cependant les politiques et mécanismes d'allocation et de libération plus complexes.

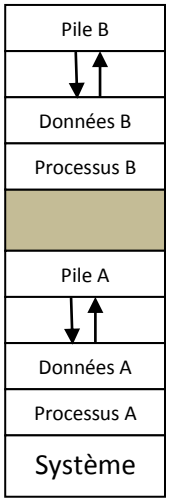
4.4.2. Opérations sur la mémoire

Lorsque la mémoire devient trop fragmentée, il est possible de la **compacter** en déplaçant tous les processus vers le bas de la mémoire et regroupant ainsi tous les blocs libres. Cette opération de *compactage* est très couteuse en temps CPU et doit être réalisée le moins souvent. Il faut par exemple 2.7s pour déplacer tous les blocs dans une mémoire de 256Mo copiant 4 octets en 40ns.

Avec cette solution de gestion par partitions variable, lorsqu'un processus fait une requête d'*allocation dynamique de mémoire,* on lui alloue de la place dans le tas (***heap***) si le système le peut, sinon de la mémoire supplémentaire contiguë à la partition du processus (agrandissement de celle-ci). Quand il n'y a pas de place, il est nécessaire de déplacer un ou plusieurs processus :

- dans la mémoire centrale, pour obtenir une partition plus grande
- vers le disque, par technique de va-et-vient, en libérant de la mémoire.

NB: Dans la plupart des systèmes, on alloue à chaque processus un espace mémoire légèrement plus grand que sa taille courante, au cours de son chargement ou lors de sa migration à partir du disque. L'espace libre dans la zone mémoire du processus est alors placé entre les segments susceptibles de grandir (la pile et le tas).

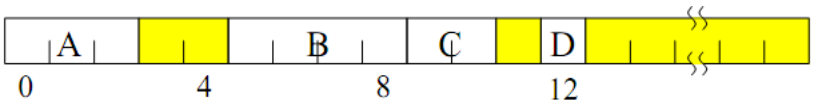


Il existe trois façons de mémoriser l'occupation de la mémoire, minimiser l'espace perdu, et réduire autant que possible la *fragmentation* : les tables de bits *(bits maps), l*es listes chaînées et les subdivisions (*buddy).*

4.4.3. Stratégies de gestion de la mémoire

4.4.3.1. Gestion par tables de bits (bits maps)

Dans ce mécanisme, la mémoire est répartie en ***unités d'allocations*** de quelques octets à quelques Ko. A chaque unité, correspond un bit de la table des bits : à 0 si l'unité est libre, à 1 sinon. Cette table est gérée par le système et conservé en mémoire centrale.



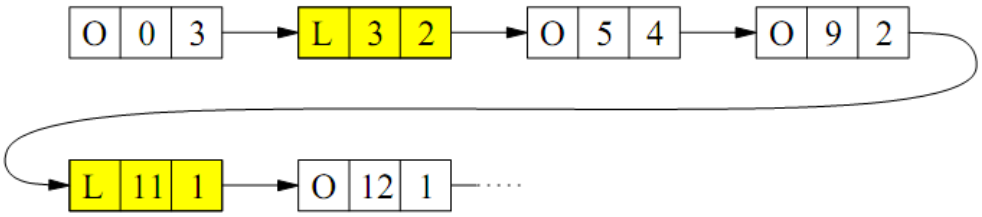
1	1	1	0	0	1	1	1
1	1	1	0	1	0	0	…
…							

Cette approche présente deux inconvénients :

- Choix de la taille de l'unité d'allocation : une taille petite impose une table plus grande (plus de temps pour les opérations de recherche), une taille trop grande entrainera un gaspillage pour les derniers blocs de mémoire alloué à un processus (et seront partiellement remplies).
- Recherche de *k* bits libres dans la table, qui nécessite des manipulations coûteuses sur les masques de bits. Car il s'agira à chaque allocation mémoire (retour d'un swap par exemple), de rechercher suffisamment de 0 consécutifs dans la table pour que la taille des ces unités permette de loger le processus.

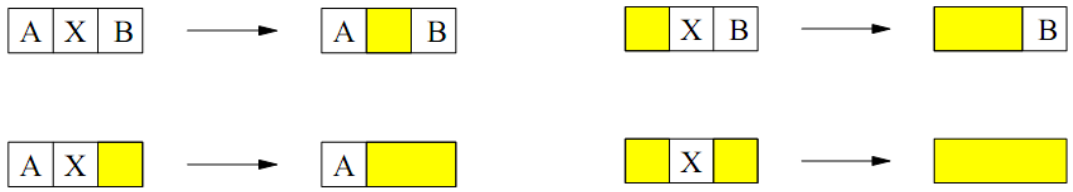
4.4.3.2. Gestion par listes chaînées

Cette technique consiste à gérer une liste chaînée des segments libres et occupés, dans laquelle chaque segment est représenté par un bit d'état, son adresse de début, et sa longueur. Cette liste est triée par adresse de début croissante.



Afin de conserver à la liste la plus courte taille possible, et limiter la fragmentation, un compactage est effectué à chaque libération de mémoire : il est fusionné avec son prédécesseur et/ou son successeur si ceux-ci sont également libres. On ne peut donc avoir plus de deux éléments consécutifs décrivant un segment libre. Pour des raisons

d'efficacité la liste est doublement chainée. Un processus qui se termine a deux voisins (saut s'il est le premier ou le dernier de la liste), qui peuvent être des emplacements occupés ou libres. Quatre situations peuvent alors se présenter.



Plusieurs algorithmes peuvent être utilisés pour allouer de la mémoire à un processus nouvellement crée (ou chargé à partir du disque) :

- *First fit (première zone libre) :* La liste est parcourue jusqu'à ce qu'on trouve un segment libre de taille suffisante pour contenir l'espace mémoire demandé. S'il reste de l'espace libre dans le segment, la cellule correspondante est scindée pour conserver la trace de cet espace. C'est l'algorithme le plus rapide car il limite sa recherche autant que possible.

- *Next fit (zone libre suivante) :* Cet algorithme fonctionne de la même façon que le précédent, à la seule différence qu'il mémorise en plus la position de l'espace libre alloué. Quand il est de nouveau sollicité pour trouver un emplacement libre, la recherche dans la liste est faite à partir de la position mémorisée (au lieu de recommencer au début de la liste). Des simulations montrent que les performances de cet algorithme sont légèrement meilleures que celle du *first fit.*

- *Best fit (meilleur ajustement) :* La liste est entièrement parcourue, et on choisit finalement le plus petit segment dont la taille est supérieure à celle de la mémoire demandée. L'idée est celle d'éviter de fractionner une grande zone mémoire dont on pourrait avoir besoin ultérieurement. Mais dans la pratique, on constate que cet algorithme fait perdre plus de place que les deux algorithmes précédents, car il a tendance à remplir la mémoire aves des petites zones libres inutilisables plus tard. L'algorithme *first fit* donne en moyenne des zones plus grandes.

- *Worst fit (plus grand résidu):* cet algorithme est similaire au *best fit* à la seule différence qu'on cherche la plus grande zone possible, ie celle qui donnera le résidu le plus utilisable possible. Les expériences montrent que cet algorithme ne donne pas non plus les meilleurs résultats car il détruit en premier les segments de grande taille, qui ne seront plus disponible par la suite.

- *Quick fit (placement rapide) :* Cet algorithme essaie d'accélérer les recherches en gérant séparément la liste des segments libres et celle des segments occupés. De

plus, la liste des segments libres est organisées en plusieurs listes séparées, chacune pour gérer les tailles courantes (une liste pour les blocs libres de 4Ko, une pour les blocs de 8Ko, etc.). Ceci simplifie grandement la recherche, rend plus difficile la fusion des blocs libérés.

4.4.3.3. Gestion par subdivision

La gestion par subdivision (*buddy system)* encore appelée ***allocation des frères siamois*** s'appuie sur la représentation binaire des adresses pour simplifier les processus d'allocation et de libération de la mémoire. Le gestionnaire de la mémoire ne manipule une liste constitué uniquement des blocs libres dont la taille est une puissance de 2 (1, 2, 4, 8octets, …, Taille max de la mémoire). Cette liste est chaînée au moyen d'un vecteur de listes de telle sorte que la liste d'indice *k* contienne les blocs de taille 2^k.

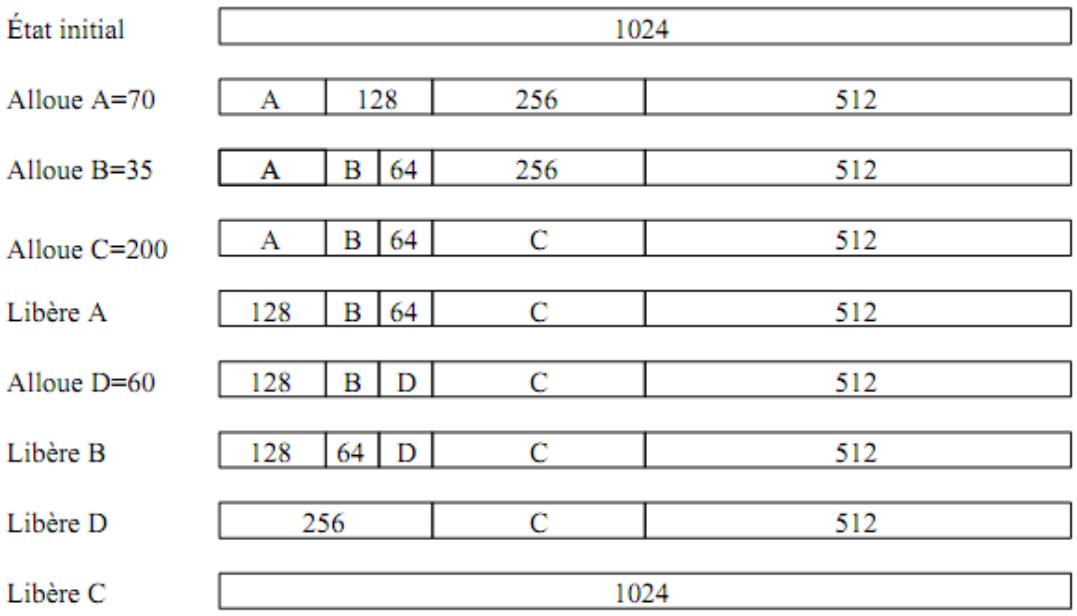
Initialement toutes les listes sont vides, excepté celle dont la taille de blocs est égale à la taille de la mémoire. Celle-ci contient une seule entrée qui pointe sur l'intégralité de la mémoire.

Lorsqu'on doit allouer de la mémoire, on recherche un bloc libre dans la liste correspondant à la plus petite puissance de deux, supérieure ou égale à la taille demandée. Si la liste est vide, on sélectionne un bloc dans la liste de puissance immédiatement supérieure, et l'on le divise en deux moitiés (*frères siamois*) : l'un sera utilisé pour allouer l'espace mémoire recherché, l'autre est chaîné dans la liste des blocs libres de la taille correspondante. Si la liste de puissance de deux immédiatement supérieure est elle-même vide, on remonte le tableau de listes jusqu'à trouver un bloc libre, que l'on divise récursivement autant de fois que nécessaire pour obtenir un bloc de la taille voulue.
Lorsqu'on libère un bloc et que son frère siamois est lui-même déjà libre, les deux blocs sont fusionnés pour redonner un unique bloc de taille supérieure. Ce processus se poursuit récursivement jusqu'à ce qu'aucune fusion ne soit plus possible, et le bloc résultat est placé dans la liste des blocs de la taille correspondante.

Illustration : Une mémoire de 1Mo est gérée suivant la stratégie de subdivision. On alloue successivement des blocs de A=70Ko, B=35Ko, C=200Ko, puis A libère son bloc, on alloue D=60Ko. B libère son bloc, ensuite D et enfin C. La figure ci-dessous présente l'ensemble des mécanismes mis en œuvre par le gestionnaire de la mémoire.

Remarque : Avec cette stratégie, l'allocation et la libération des blocs est très simple, mais elle est très souvent source de beaucoup de gaspillage de mémoire : car un processus qui a besoin de $2^n + 1$ octets va se voir allouer un espace de 2^{n+1} octets.

On parle alors de ***fragmentation interne*** car l'espace perdu fait partie des segments alloués, par opposition à la ***fragmentation externe***, qui se produit avec la gestion de la mémoire par liste chainée.



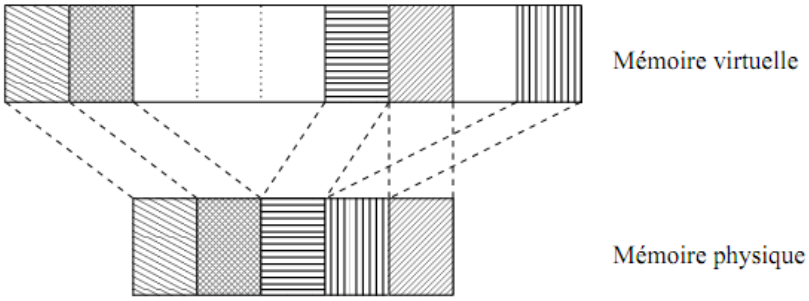
4.5. La mémoire virtuelle

Un programme doit pouvoir être plus grand que la mémoire disponible, c'est d'ailleurs la situation qui prévalait au début de l'informatique. La solution qui était alors adopté à cette période était celle de subdiviser le programme en parties ou blocs indépendant appelés ***overlay*** ou *segment de recouvrement*. Un zone permanente du programme est alors chargée en mémoire, mais les fonctions externes implémentées dans les *overlays* ne sont alors chargées que lorsque nécessaire. Cependant cette solution est trop lourde car elle exige la migration vers la mémoire de la l'intégralité de l'*overlay* lorsqu'une fonction externe est évoquée, de plus le programmeur doit penser à découper convenablement son programme en *overlay*.

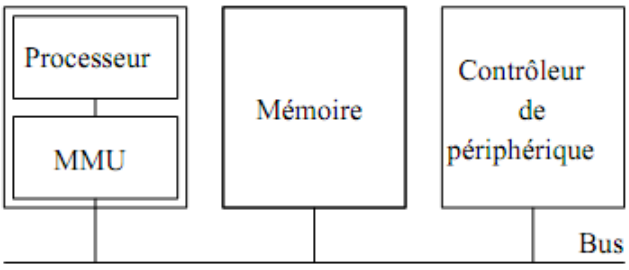
Une autre solution consiste à utiliser le mécanisme de mémoire virtuelle. La taille du programme et de ses données peut alors dépasser la taille de la mémoire physique disponible, c'est le système (*et non le programmeur*) qui se charge de découper le programme en bloc appelés ***segments*** ou ***pages***, et de ne charger que les *segments* ou *pages* nécessaire à l'avancement immédiat du processus. Avec cette solution, un processus posséder plusieurs *pages* ou *segments* en mémoire, ceux-ci ne sont pas nécessairement contigus.

4.5.1. La pagination

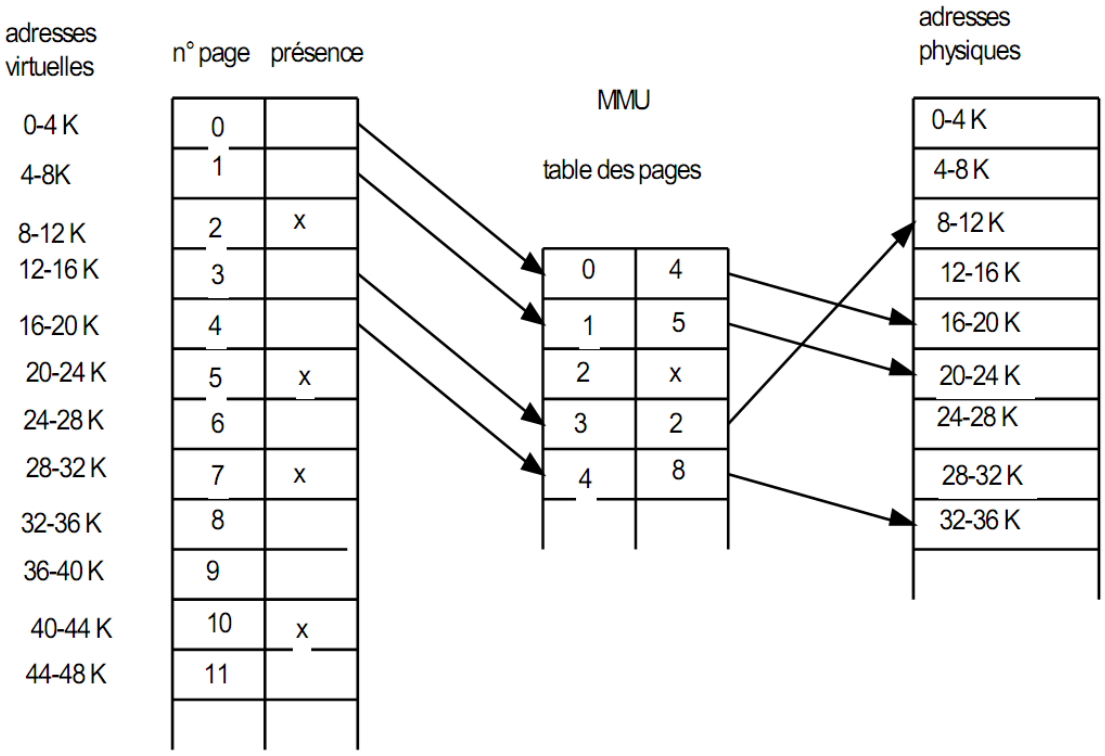
Il s'agit de la technique de mémoire virtuelle la plus utilisée. L'espace d'adressage virtuel est alors découpé en unités de taille fixe appelés *pages*. Cette taille est une puissance de deux (habituellement comprise entre 512 et 4096 octets). La mémoire centrale est également découpée en unités physiques de même taille appelées *cadres de pages.* Les échanges entre la mémoire et le disque portent uniquement sur des pages entières.



De ce fait, un processus donné a potentiellement un espace mémoire virtuel illimité, devant être logé dans une mémoire physique limité. On parle alors ***d'adressage virtuel.*** Pour le processus c'est le système qui s'occupera de charger les pages utilisées. Les adresses virtuelles utilisées par le processeur pour accéder à l'ensemble de l'espace d'adressage virtuel sont traduites en adresses physiques par un circuit spécial placé entre le processeur et le bus mémoire : le *MMU (Memory Management Unit)*

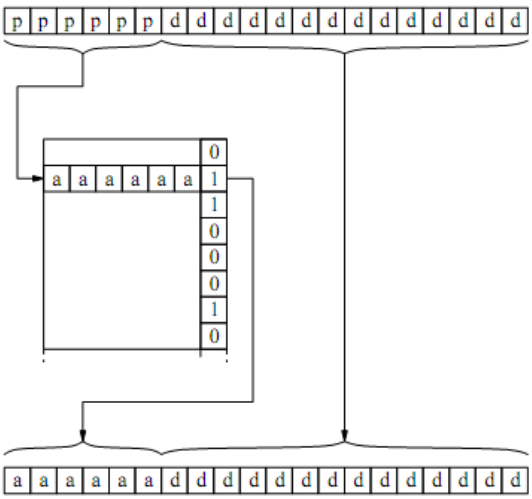


La demande de pages à charger peut être plus élevée que le nombre de cadres disponibles. Une gestion de l'allocation des cadres libres est nécessaire. La MMU possède une table d'indirection des pages pour convertir les adresses virtuelles en adresses physiques (*table des pages*), qui donne pour chaque page virtuelle le numéro de page physique (ou cadre de page) sur laquelle elle est placée. Les pages virtuelles non actuellement mappées en mémoire centrale sont identifiés par un bit de présence. Une page est dite *mappée* ou *chargée* si elle est physiquement présente en mémoire.



L'indexation de la table des pages se fait en décomposant l'adresse virtuelle en un numéro de page : ***page index*** et un déplacement : ***offset*** dans la page. L'index sert à adresser la table des pages, qui fournit le numéro de page physique remplaçant l'index de page dans l'adresse physique.

Dans la figure précédente nous avons des pages de 4Ko. L'adresse virtuelle 12292 sera vue comme $12 * 1024 + 4 = 12Ko + 4$ Soit un déplacement de 4 octets dans la page virtuelle 3. Cette page virtuelle se trouve dans le cadre de page 2. L'adresse physique correspondante est donc $8Ko + 4 = 8196$.



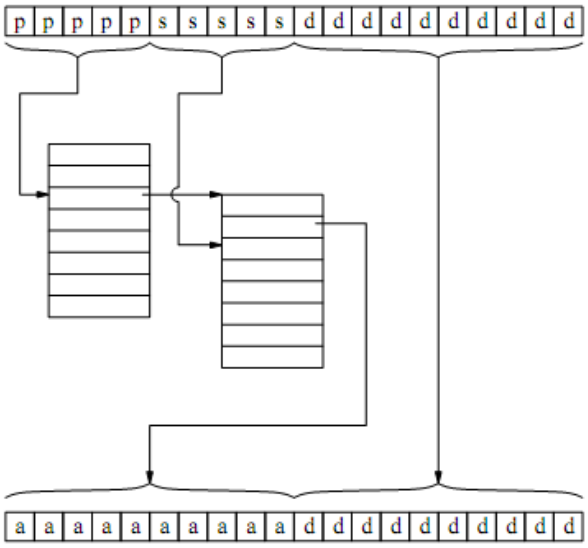
Sur la figure, la page virtuelle 2 n'est pas mappée. Une adresse virtuelle comprise entre 8192 et 12287 donnera lieu à un ***défaut de page :*** il s'agit d'une tentative d'accès

à une adresse virtuelle correspondant à une page non mappée. En cas de défaut de page une *interruption* (de type *trap*) se produit et le processeur est rendu au système. Celui-ci doit alors effectuer les opérations suivantes :

- déterminer la page à charger ;
- déterminer la page à décharger sur le disque pour libérer un cadre de page ;
- lire sur le disque la page à charger ;
- modifier les bits de présence et la table des pages.

Il peut arriver que la table des pages consomme beaucoup de mémoire. Par exemple, avec un espace d'adressage de 2^{32} octets, et des pages de 2^{12} octets, on doit avoir 2^{20}, soit une table des pages de 1Mo. Pour réduire la table des pages et éviter d'avoir des espaces vides, on lui applique son *propre principe.* On crée une table à deux niveaux d'indirections : la partie haute de l'adresse permet de déterminer la table des pages devant être utilisée, qui elle-même est accédée au moyen de la partie médiane de l'adresse afin de déterminer le numéro de la page physique utilisée.

Ainsi, pour cet l'exemple, avec un premier niveau indexé sur 10 bits, et un deuxième niveau également sur 10 bits, on réduit considérablement la mémoire physique occupée par la table des pages, en évitant de conserver l'intégralité de celle-ci en mémoire



4.5.2. Algorithmes de remplacement des pages

Lorsqu'***un défaut de page*** se produit, une interruption est déclenché et le système doit choisir une page à enlever de la mémoire afin de faire de la place pour la page qui doit être chargée (s'il n'y a pas de cadre de page libre en mémoire). Si la page qui doit être supprimée a été modifiée depuis son dernier chargement à partir du disque, elle doit être réécrite sur le disque.

4.5.2.1. Algorithme du peu fréquemment utilisé (*NRU : Not Recently Used*)

Cet algorithme associe à chaque page deux bits d'information :

- R (le plus à gauche) ou encore *bit de référence*, il est positionné à 1 chaque fois qu'une page est accédée (lue)
- M ou *bit de modification* est positionné chaque fois que la page est modifiée par le processus.

De façon régulière (à chaque interruption horloge), le système remet à 0 tous les bits R des pages présentes en mémoire. Les bits R-M forment le code d'un index de valeur 0 à 3 en base 10.

0 : $\bar{R}\bar{M}$ non accédée et non modifiée
1 : $\bar{R}M$ non accédée mais modifiée. Ce produit lorsque le bit R d'une page déjà modifiée est remis à zéro par la routine d'interruption.
2 : $R\bar{M}$ accédée mais non modifiée.
3 : RM accédée et modifiée.

Lorsqu'une page doit être remplacée, le système parcourt la table des pages et choisit une des pages de plus petite valeur binaire. Cet algorithme suppose qu'il vaut mieux retirer une page modifiée mais non accédée récemment, qu'une page non modifiée mais plus fréquemment utilisée. Cet algorithme est facile à implémenter et fournit des performances suffisantes dans la plupart des usages.

4.5.2.2. Algorithme premier entré, premier servi *(FIFO : First In – First Out)*

L'idée de cet algorithme est de supprimer la page qui ayant le plus résidé longtemps en mémoire. Cette solution peut être mise en œuvre en indexant chaque page par sa date de chargement et en constituant une liste chaînée, la première page de la liste étant la plus ancienne et la dernière page, la plus récemment chargée. Le système remplacera en cas de nécessité la page en tête de la liste et chargera une nouvelle page en queue de liste.
Deux critiques sont généralement faite à cet algorithme :

- Ce n'est pas parce qu'une page est ancienne en mémoire, qu'on s'en sert le moins souvent.
- L'algorithme n'est pas stable : quand le nombre de cadres augmente, le nombre de défaut de page ne diminue pas nécessairement (on parle alors d'*anomalie de BELADY :* qui a proposé en 1969 un exemple à 4 cadres qui provoque plus de défaut de pages qu'avec 3 cadres.)

Cet algorithme peut être amélioré pour prendre en compte les pages le plus souvent utilisées.

Amélioration 1 : Le système examine les bits R et M de l'algorithme précédent, et remplace la page la plus ancienne et de plus petite valeur binaire.
Amélioration 2 : Une autre amélioration de l'algorithme FIFO est connue sous le non de l'*algorithme de la seconde chance*. On teste le bit R de la page la plus ancienne, s'il est à 0 la page est remplacée, sinon il est remis à 0 et la page est placé en queue de file. Si toutes les pages ont été référencées, alors l'algorithme de la seconde chance est identique à l'algorithme FIFO classique.

Anomalie de BELADY
Si l'on a cinq pages virtuelles que l'on accède selon la séquence 012301401234, on aura neuf défauts de page avec trois pages mémoires contre dix avec quatre pages mémoires.

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
			0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P			P	P	

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
				0	0	0	1	2	3	4	0	1
	P	P	P	P			P	P	P	P	P	P

4.5.2.3. Algorithme du moins fréquemment utilisé *(LRU : Least Recently Used)*
L'idée est de retirer la page la moins récemment référencée. Pour cela, le système indexe chaque page par le temps écoulé depuis sa dernière référence et il constitue une liste chaînée des pages par ordre décroissant de temps depuis la dernière référence. Cet algorithme est stable, mais il nécessite une gestion matérielle couteuse de la liste qui est modifié à chaque accès à une page.

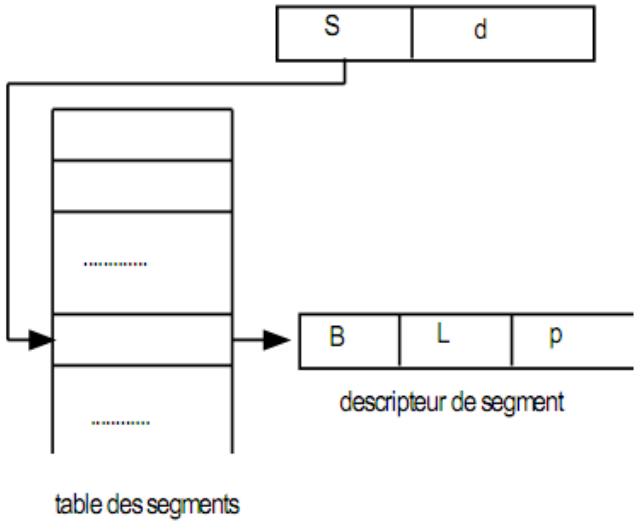
Il existe des solutions qui implémentent par logiciel une version dégradée de l'algorithme LRU :

- **NFU (*Not Frequently Used)* ou LFU (*Least Frequently Used*)** : le système gère un compteur du nombre de référence pour chaque page. Il cumule dans ce compteur le nombre de bit R juste avant la RAZ de R au top horloge. Lors d'un remplacement, on choisit la page ayant la plus petite valeur de compteur.
  Inconvénient : Une page qui a été fortement référencée dans le passé, mais qui n'a pas été utilisée ne sera pas remplacée avant un bon moment.
- **NFU modifié ou algorithme du vieillissement** (*aging*) : Le cumul des bits R est alors réalisé en décalant vers la droite (division entière par 2) la valeur du compteur et en recopiant le bit R sur le poids fort. Cet algorithme de vieillissement garantit que les pages les plus récemment utilisées auront les plus grandes valeurs de compteur.

4.5.3. La segmentation

Dans cette solution, l'espace d'adressage d'un processus est divisé en ***segments***, générés à la compilation. Chaque segment est repéré par son numéro S et sa longueur variable L. Un segment est un ensemble d'adresses virtuelles contiguës.

Contrairement à la pagination, la segmentation est « **connue** » du processus : une adresse n'est plus donnée de façon absolue par rapport au début de l'adressage virtuel; l'adresse est donnée par un couple (S , d) où S est le n° du segment et d le déplacement dans le segment, $d \in [0, L[$. Pour calculer l'adresse physique, on utilise une ***table des segments*** :



B : adresse de base (adresse physique de début du segment)
L : longueur du segment ou limite
p : protection du segment

L'adresse physique correspondant à l'adresse virtuelle (S, d) sera donc B + d, si $d \leq L$.
La segmentation est approprié pour la gestion des objets communs (rangés chacun dans un segment), surtout lorsque leur taille évolue dynamiquement.

5. Le système de fichiers

La plupart des applications informatiques ont besoin de la conservation permanente des informations. L'un des critères déterminant de l'efficacité d'un système d'exploitation est le stockage persistent, rapide et fiable de grandes quantités de données. Ces informations sont stockées sous formes de ***fichiers***, vus comme une suite d'enregistrements logiques d'un type donné qui ne peuvent être manipulés qu'au travers d'opérations spécifiques. Il s'agit d'un objet nommé, résidant hors de l'espace d'adressage du processus, mais offrant une interface permettant la lecture et l'écriture de données dans ce dernier.

L'utilisateur n'est pas concerné par l'implantation des enregistrements, ni par la façon dont sont réalisées les opérations. Ils sont gérés par le système d'exploitation. Leur structure, nommage, accès, utilisation, protection et implantation constituent des éléments majeurs de la conception du système d'exploitation.

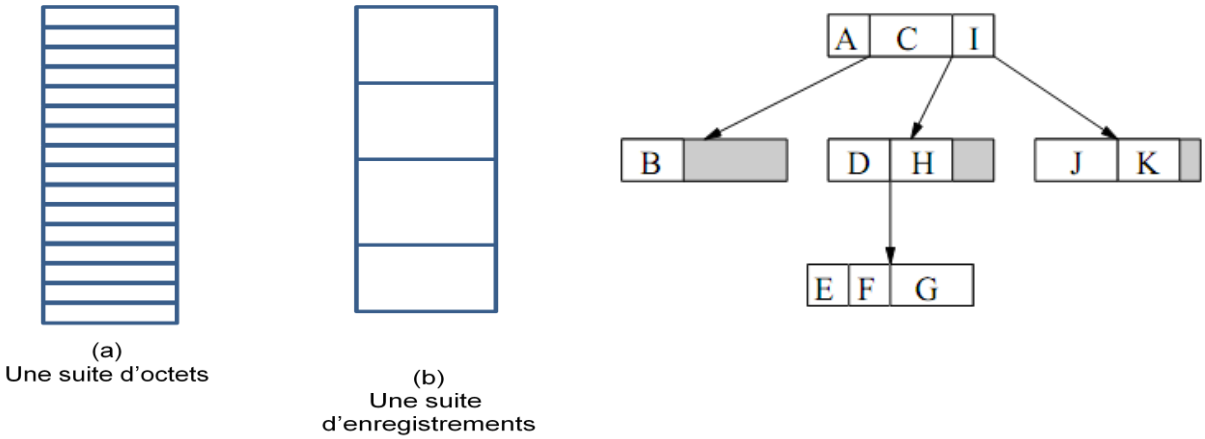
La partie du système d'exploitation qui est en charge de la gestion des fichiers est appelée: ***système [de gestion] de fichiers*** (***File System***). Du point de vue de l'utilisateur, les services attendus d'un SGF comprennent :

- L'indépendance vis-à-vis du type de périphérique supportant le système de fichier ;
- La création des fichiers ;
- La suppression de fichiers ;
- La gestion de protections et de droits d'accès ;
- L'ouverture de fichiers avec gestion des accès concurrents ;
- La fermeture de fichiers avec libération automatique des ressources utilisées ;
- La lecture et l'écriture de données ;
- La structuration de données en enregistrements.

5.1. Structure d'un fichier

En fonction du système, différentes organisations sont proposées pour structurer les données dans un fichier :

a. Une suite d'octets : organisation la plus simple. Il s'agit d'une suite d'octets sans structure visible ; se sont les différents programmes qui leur donnent une signification. **Exple** : Unix, DOS, Windows, …
b. Une suite d'enregistrement de taille fixe ;
c. Un arbre d'enregistrements de taille variable indexés chacun par une clé et groupés par blocs de façon hiérarchique.



(a)
Une suite d'octets

(b)
Une suite
d'enregistrements

5.1.1. Types de fichiers et attributs de fichiers

Les types de fichiers les plus couramment définis sont les suivants :

- Les fichiers ordinaires : contenant les données utilisateurs ;
- Les répertoires : permettant d'indexer d'autres fichiers de façon hiérarchique ;
- Fichiers spéciaux de type caractère : modélisent les périphériques d'entrée/sortie travaillant caractère par caractère (clavier, écran, imprimante, sockets) ;
- Les fichiers spéciaux de type bloc : modélisent des périphériques d'E/S travaillant sur des blocs (disque)

Les attributs d'un fichier sont des informations qui caractérisent celui-ci, ils sont inscrits par le système et utilisé par ce dernier pour décrire/gérer le fichier. On distingue entre autre :

- Heure et date de création du fichier ;
- Date du dernier référencement ;
- Date de la dernière modification ;
- Taille du fichier ;
- Propriétaire du fichier ;
- Droits d'accès au fichier ;
- …

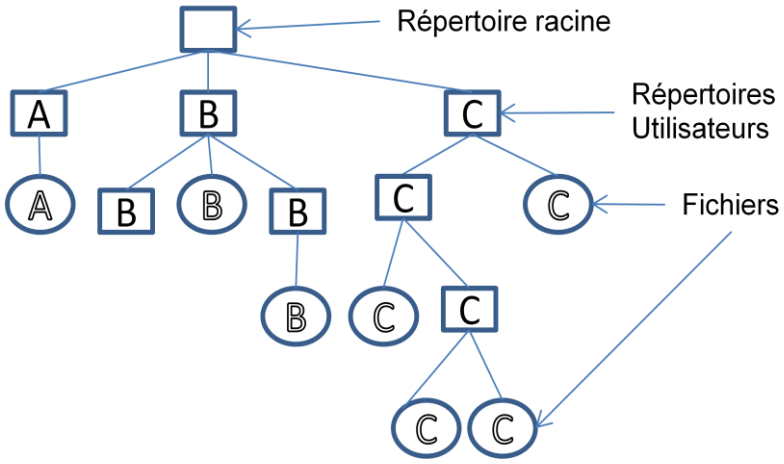
5.1.2. Les fichiers ordinaires

Ils sont eux-mêmes subdivisés en plusieurs types, et reconnus en tant que tel en fonction de leur nature. Le typage d'un fichier ordinaire peut être :

- Un typage fort : une extension est alors ajoutée (par le système ou l'utilisateur) au fichier pour donner une indication sur son contenu. **Exple** : fichier exécutable *.com, .bin* ou *.exe* sous DOS
- Un typage faible : le système doit réaliser une inspection du contenu pour déterminer la nature réelle du fichier.

5.1.3. Les répertoires

Encore appelés *catalogues* ou *dossiers*, ils sont utilisés pour garder la trace des fichiers ou les organiser de façon hiérarchique. Le répertoire est organisé comme une liste d'entrées dont chacune fait référence à un fichier unique. Chaque entré est détrite par les attributs d'un fichier, le type de fichier (répertoire, lien symbolique, fichier spécial en mode caractère ou bloc, etc.), l'organisation du fichier sur le disque, etc.



Les opérations qu'on peut réaliser sur un répertoire sont :

- La création d'une nouvelle entrée (ajout de fichier d'un type donné) ;
- La suppression d'une entrée existante, ceci va conduire à la suppression physique du fichier si cette entrée était la dernière référence vers ce dernier (lien symbolique).
- Le renommage d'une entrée, gérée comme la création d'une nouvelle référence vers ce fichier, suivie de la suppression de l'ancienne référence (réalisé par la commande ***mv*** sous Unix).

5.2. Organisation physique du système de fichiers

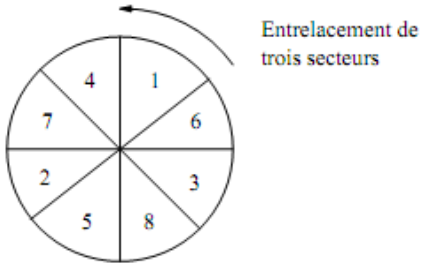
5.2.1. Stockage du fichier sur le disque

Une grande part de la performance du SGF est justifiée par la rapidité d'accès aux données du fichier, lui-même dépend de leur structuration physique au niveau du disque.

5.2.1.1. Allocation contiguë

Elle consiste à allouer les blocs logiques constituant un fichier donné de façon contiguë sur le disque. Cette approche à l'avantage d'offrir des accès rapides, mais rend difficile l'allocation de nouveaux fichiers du fait de la *fragmentation externe* qui en résulte. Il est alors nécessaire d'utiliser un outil de *défragmentation* pour regrouper les espaces libres du disque. De plus si la taille d'un fichier augmente, il peut être

nécessaire de déplacer physiquement celui-ci sur le disque pour trouver le nombre de blocs suffisants à son enregistrement. Souvent la contiguïté logique des blocs n'implique pas nécessairement la contiguïté physique de ces blocs sur le disque. Car pour optimiser le *débit de transfert* et minimiser la *latence* entre deux accès à des blocs logiques qui se suivent, certains systèmes *entrelacent* (*interleaving* – en anglais) ceux-ci, de sorte que le délai entre deux demandes d'accès à des blocs consécutifs corresponde au temps de passage de la tête de lecture d'un bloc logique au bloc suivant par rotation du disque.



Entrelacement de trois secteurs

Un autre avantage de ce type d'allocation est qu'il est facile de conserver la trace de l'emplacement des blocs : il suffit alors de mémoriser l'adresse disque du premier bloc et le nombre de bloc du fichier.

5.2.1.2. Allocation par liste chaînée

Avec cette méthode, seul le numéro logique du premier bloc est conservé dans le répertoire. Une zone spécifique de chaque bloc contient le numéro logique du prochain bloc du chaînage. Lorsque la taille du fichier change, la gestion des blocs occupés est plus simple, il n'y a plus de limitation de taille, si ce n'est la taille du disque lui-même. Deux inconvénients sont toutefois à noter avec ce type d'allocation :

- La perte d'un pointeur de bloc entraîne la perte de toute la fin du fichier, sinon il faut faire un double chaînage (pour augmenter la résilience du système).
- Le mode d'accès est séquentiel (ce qui rend la lecture de fichier plus lente).

Afin d'accélérer les accès non séquentiels, les chaînages peuvent être séparés des blocs eux-mêmes. On a alors une table possédant une entré pour chaque bloc, indiquant le du bloc suivant dans le chaînage du fichier. Ce système a été introduite par MS-DOS qui conserve pour des raisons de fiabilité plusieurs copie de cette table encore appelé *FAT (File Allocation Table*).

Les avantages de cette implémentation sont l'accélération des accès aux listes chaînées, et la séparation physique entre les données et les pointeurs de bloc. Ceci facilite le traitement et la sauvegarde des informations de chainage. Le principal

inconvénient est qu'elle implique de parfois maintenir des tables de grande taille, même si peut de blocs sont alloués.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
—	—	EOF	13	2	9	8	—	4	12	3	—	EOF	EOF	—

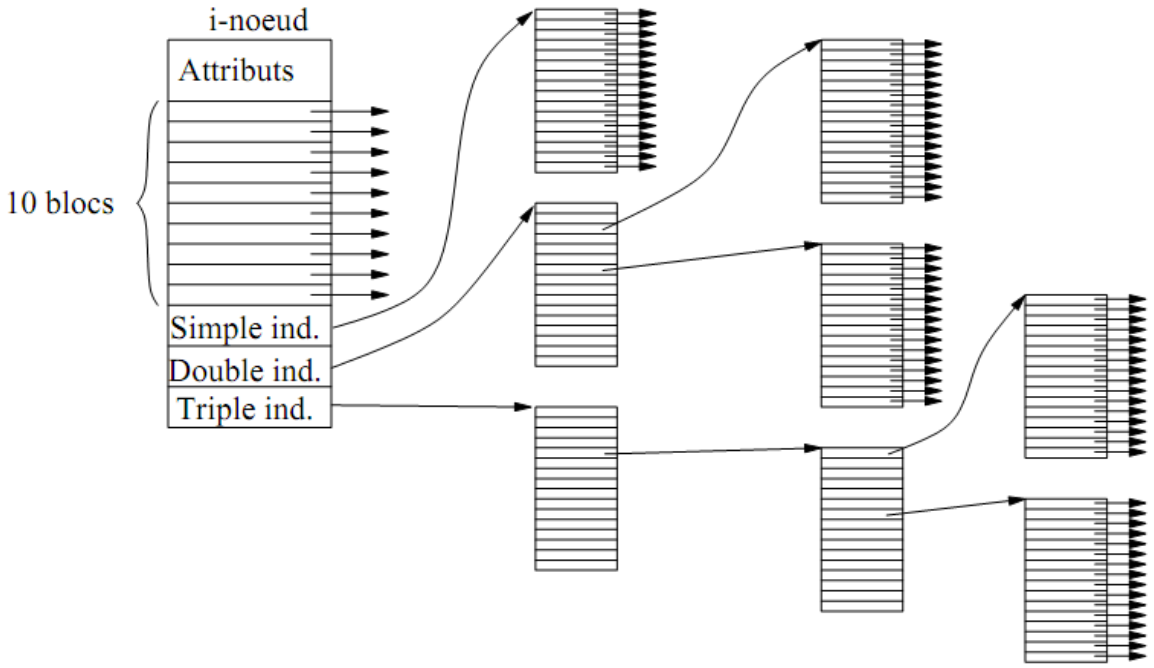
A	6
C	10
B	5

A : 6 → 8 → 4 → 2
B : 5 → 9 → 12
C : 10 → 3 → 13

Pour l'illustrer, un disque de 4Go découpé en blocs de 4Ko nécessite environ 1 million d'entrées (dont l'adresse est codée sur 32 bits) dont 20 sont utilisés.

5.2.1.3. Organisation par i-nœuds

Ici l'idée est d'associer à chaque fichier la liste de ses blocs, sous forme d'une table à capacité variable. Ceci, afin de ne pas causer de surcoût aux petits fichiers, tout en offrant de la flexibilité pour les gros. La table possède plusieurs niveaux d'indirection (similaire à la table des pages à plusieurs niveau gérée par la MMU).



Les blocs d'index contiennent 128 ou 256 pointeurs sur des blocs de données (simple indirection) ou sur d'autres blocs d'index (double et triple indirection). Le nombre de blocs de chainage alloué est fonction de la taille du fichier, exprimée en nombre de blocs :

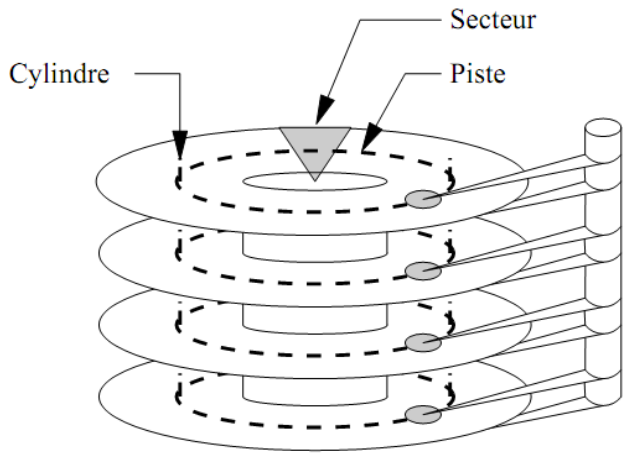
- Pour moins de 10 blocs, le fichier est stocké dans l'i-nœud lui-même.
- Pour moins de 266 blocs, on utilise un unique bloc d'indirection sur le disque (10 + 256) ;
- Pour moins de 65802 blocs, on utilise deux niveaux d'indirection sur le disque ($10 + 256 + 256^2$)
- Jusqu'à 16 millions de blocs, on utilise deux niveaux d'indirection sur le disque ($10 + 256 + 256^2 + 256^3$)

Cette structure est extrêmement puissante, car il ne nécessite qu'au plus trois accès disque pour avoir l'indice de bloc situé à une position quelconque dans le fichier. Elle permet également de manipuler des fichiers de très grande taille, dans lesquels seuls les blocs écrits sont effectivement alloués.

5.2.2. Organisation de l'espace disque

5.2.2.1. Structure physique

Les disques durs sont construits comme un empilement de disques magnétiques rigides dont la surface est balayée par un ou plusieurs jeux de bras portant des têtes de lecture.



- La piste : zone couverte par la tête de lecture en un tour disque lorsque le bras reste dans une position donnée ;
- Le cylindre : zone couverte sur tous les disques par l'ensemble des têtes de lecture en un tour de disque lorsque le bras reste dans la même position ;
- Le secteur : portion de disque représentant un fraction de la surface angulaire totale.

5.2.2.2. Taille des blocs

La taille des blocs ou *unité d'allocation disque* est un paramètre critique pour le système. Une taille plus grande que la moyenne des fichiers génèrera un gaspillage

d'espace (*fragmentation interne*), alors qu'une taille trop petite ralentira les accès du fait du grand nombre de blocs à accéder pour obtenir la même quantité d'information.

5.2.2.3. Gestion des blocs libres.

Une fois la taille des blocs choisie (de l'ordre de quelques Ko – 4Ko par exemple), on doit pouvoir mémoriser les blocs libres. Les deux techniques les plus répandues sont :

Liste chainée des numéros de blocs :

On maintient une liste de blocs libres. Une possible implémentation consisterait à indiquer à chaque début de bloc libre d'indiquer l'adresse du bloc suivant. Le principal inconvénient est qu'il faudra alors parcourir l'ensemble de la surface disque pour allouer les blocs les uns après les autres. L'implémentation classique retenue par la plupart des systèmes à disposer d'un chainage de blocs libres dont chacun contient le plus possible d'adresses de blocs libres.

Pour l'illustrer, un disque de 2Go organisé en blocs de 2Ko adressés sur 24bits (seuls 20bits sont effectivement utilisé), on peut stocker 680 adresses de blocs libres par bloc en plus du chainage et du nombre d'emplacements utilisés. Pour le stockage de l'ensemble des blocs libres il faudra au plus 1543 blocs.

La table des bits :

On maintient comme se fut le cas avec la mémoire centrale, une table de bit pour mémoriser l'occupation des blocs du disque. Ainsi pour l'exemple illustratif l'ensemble de la table de bit occuperait 64 blocs de 2Ko.

5.2.2.4. Cohérence du système de fichier

L'utilisation de des caches disques et le système pose des problème de synchronisation et de cohérence des données en cas d'arrêt brutal du système. Tous les systèmes de fichiers disposent donc de programmes utilitaires (*scandisk* – Windows, *fsck* – Unix), dont le but est de vérifier et de restaurer la cohérence du système de fichiers, en préservant si possible l'intégrité des données. Cette vérification du SGF s'effectue à deux niveaux : les blocs et les fichiers.

Niveau bloc :

- On construit une table comportant deux compteurs par bloc, dont l'un mémorise le nombre de fois que le bloc est supposé faire partie d'un fichier, et l'autre le nombre de fois qu'il apparait dans la liste des blocs libres ;

- On parcourt ensuite les i-nœuds, en inspectant pour chacun d'eux la liste de ses blocs déclarés ;
- On parcourt ensuite la liste des blocs libres ;

L'ensemble du système est dans un état cohérent si tous les blocs ont l'un de leurs compteurs à 1 et l'autre à 0. Sinon, il y a une erreur, qui peut être de plusieurs types :

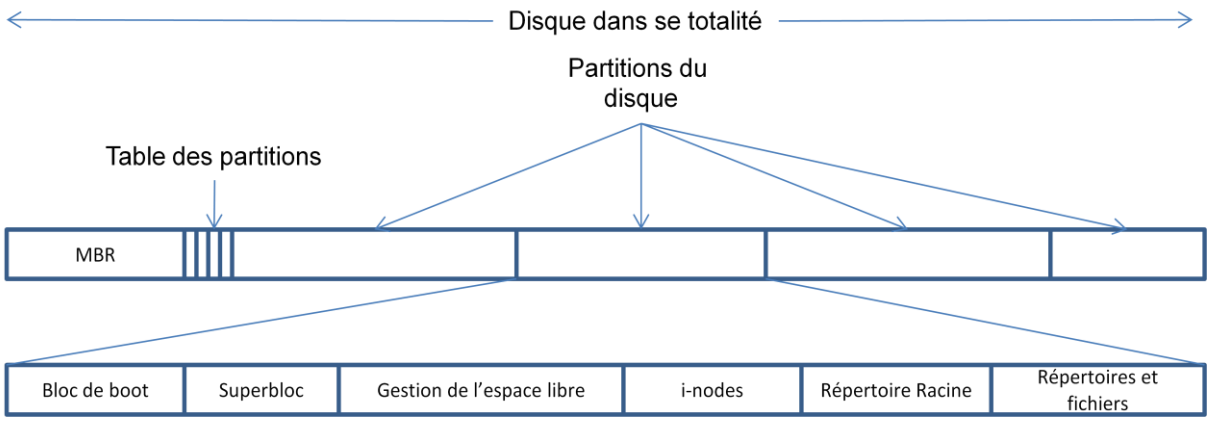
- Les deux compteurs sont à zéro : le bloc perdu est réaffecté à la liste des blocs libres ;
- Le compteur de fichier est égal à zéro et le compteur de bloc libre est supérieur à 1 : les doublons du bloc multiplement libre sont effacés de la liste des blocs libres.
- Le compteur de fichier est égal à 1 et le compteur de bloc libre est supérieur à zéro : les occurrences du bloc faussement libres sont enlevées de la table des blocs libres ;
- Le compteur de fichiers est supérieur à 1 : on effectue autant de copies du bloc multiplement utilisé qu'il y a de doublons, et on remplace le numéro du bloc par les numéros des blocs copiés dans chacun des fichiers incriminés. Les données des fichiers sont surement corrompues, mais la structure du système de fichiers est restaurée.

Niveau fichier :

En parcourant tous les répertoires, on note pour chaque i-nœud le nombre d'entrées (n) qui pointent sur ce nœud.

- Si n = 0, le fichier est mis dans un répertoire spécial (LOST+FOUND)
- Si n = nombre L de liens déclarés dans l'i-nœud, c'est correct

5.3. Structuration logique du disque



- ***Le superbloc :*** contient tous les paramètres clés du système de fichiers, il est chargé en mémoire quand l'ordinateur démarre. Il s'agit entre autre: du nombre magique identifiant le type de systèmes de fichiers, le nombre de blocs du SF, taille du système de fichier, nombre de blocs libres, pointeur sur le premier bloc libre dans la liste des blocs libres, taille de la liste des i-nodes, nombre et liste des i-nodes libres etc.
- Blocs libres du SGF, table de bits ou listes chaînées.
- Table des i-nodes : un tableau de structures de données, une par fichier, donnant toutes les informations d'un fichier.
- Répertoire racine: contient le haut de l'arborescence du SF.
- Le reste du disque contient les répertoires et fichiers.