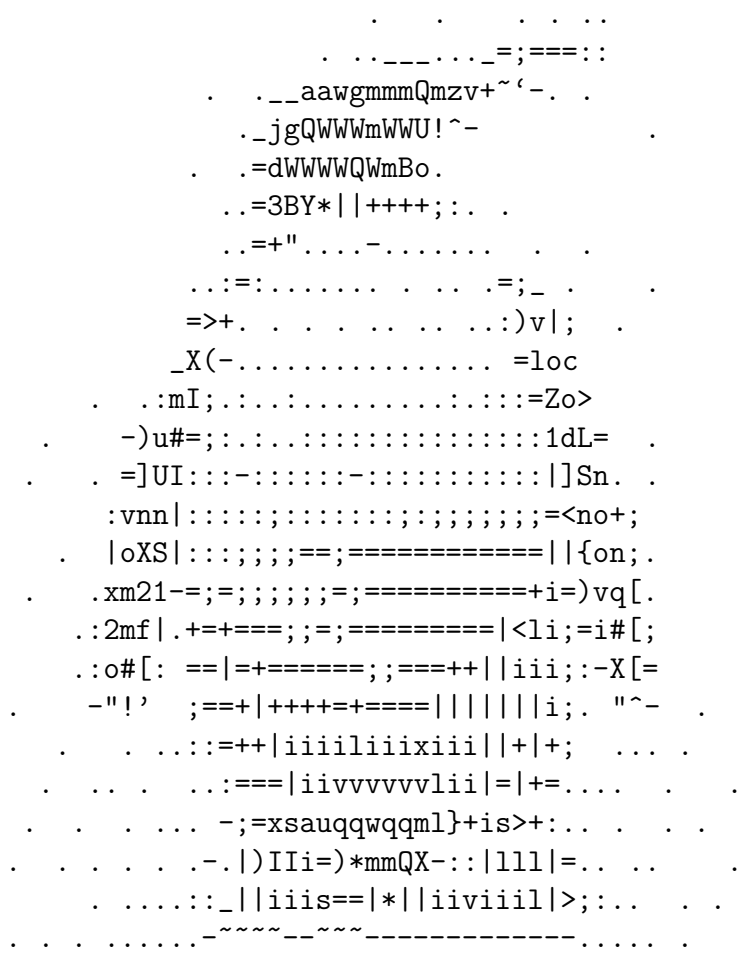


Systèmes d'exploitation

Clot Denis
clot@univ-lyon1.fr
http://bat710.univ-lyon1.fr/~dclot

Avant-propos

Ce cours d'initiation aux systèmes d'exploitation a été réalisé pour les besoins d'un enseignement s'insérant dans le cadre de la formation MIAG de l'ufr d'informatique de l'université Claude Bernard Lyon 1. Ce support est accessible aux étudiants de l'ufr par le lien `ftp://~dclot/pub/OSs/cours/SysEx.pdf`.

Sa conception repose entièrement sur les outils classiques associés à LaTeX, et sur le logiciel de dessin Xfig.

Remerciements

N'étant pas spécialiste dans le domaine, j'ai sollicité beaucoup d'aide auprès des autres enseignants de l'ufr afin de construire un plan de cours et d'obtenir des références pertinentes. Une aide précieuse m'a été apportée par Jacques Bonneville et Yves Pagnotte à travers leurs supports respectifs. Je remercie également Anne-Lyse Papini pour son support concernant l'administration NT. Certaines parties du document ont été entièrement concues par Michel Lamure. Cette aide fut un véritable soulagement dans cette course...

Enfin, je souhaite remercier les étudiants de la MIAG, qui ont suivi ce cours, pour la bonne ambiance qu'ils ont créée tout au long du semestre, pour la partage d'expérience qu'ils ont établi et leurs critiques qui m'aideront prochainement à améliorer ce support ainsi que mon approche du sujet. En particulier, merci à Jean-Noël Mégoz pour le signalement de nombreuses coquilles et imprécisions.

Villeurbanne
25 octobre 2004
Denis Clot

Table des matières

Chapitre 1

Vue d'ensemble

Limites du cours : Les systèmes d'exploitations interviennent dans divers contextes :

- Gestion d'un ordinateur individuel (système d'exploitation simple). Les services à assurer sont la gestion de fichiers et l'exécution de programme. Les points critiques sont la fiabilité, l'efficacité, la simplicité d'utilisation et la facilité à étendre la machine par de nouveaux programmes et de nouveaux périphériques.
- Gestion de procédés industriels (systèmes en temps réel). Les services à assurer sont la gestion des capteurs, la prise en compte du temps physique, des capacités de réactions aux évènements (...en particulier aux situations d'urgence, aux pannes), et la tenue d'un journal des évènements. Le principal point critique est la fiabilité. Ces systèmes apparaissent pour la commande de robots, pour le guidage de satellites...
- Gestion de transactions (systèmes transactionnels, le plus souvent en réseau). Les services à assurer sont la gestion de base de données, la gestion des transactions et des accès concurrents. Les points critiques sont la disponibilité, la fiabilité et la tolérance aux pannes. Ces systèmes sont utilisés pour les systèmes de reservation de billets, la gestion des comptes bancaires...
- Systèmes en temps partagé. Les services fournis sont les mêmes que ceux d'un OS simple en prenant en compte le fait que les ressources sont partagées par une communauté d'utilisateurs. Il faut donc ajouter des possibilités de partager l'information et notamment des possibilités de communiquer entre usagers. Les points critiques sont la disponibilité, la fiabilité et la sécurité. Unix/Linux par exemple ...

Ce partitionnement est pratique pour les besoins de la présentation, mais les contextes dans lesquels les services réclamés appartiennent à plus d'un des contextes présentés ci-dessus sont fréquents. Nous nous limiterons à l'étude des OSs dans le cadre des systèmes en temps partagé.

Un système d'exploitation (SE ou OS pour Operating System) a pour fonction de servir d'interface entre les programmes des utilisateurs et les ressources matérielles en essayant d'atteindre deux objectifs :

- fournir une interface rendant l'utilisation des ressources aussi simple que possible. Côté utilisateur, le système d'exploitation fournit une abstraction du matériel simple à comprendre et à utiliser via un jeu d'appels systèmes. La représentation de l'ordinateur que l'utilisateur peut se faire est relativement isolée des particularités matérielles. Côté matériel, les mauvaises manipulations de l'utilisateur sont bloquées par l'OS. Finalement l'OS protège chaque partie de l'autre...
- gérer et contrôler les ressources mises à disposition, e.g. l'allocation du ou des processeurs, de la mémoire, l'accès aux périphériques..., de la façon la meilleure possible selon les contraintes du contexte. Il parait assez naturel aujourd'hui que cela se fasse en plus dans le cadre d'exécutions concurrentes d'applications. L'OS joue aussi le rôle d'un multiplexeur de ressources.

Après un rapide survol historique permettant d'introduire des concepts cruciaux dans la logique de leur avènement, nous présenterons le cadre général dans lequel se place un système d'exploitation. Le chapitre terminera sur les structures rencontrées dans les constructions d'OSs.

1.1 Très courte (et laconique et lacunaire) histoire des systèmes

Thomas WATSON, président d'IBM, 1943 : *Je pense qu'il y a un marché mondial pour environ 5 ordinateurs.*

Les faits qui ont été relevés ci-après l'ont été pour introduire des concepts chers à ce cours sur les OSs. Ils peuvent être complétés par la lecture :

- de nombreux documents présents dans `~dclot/pub/OSs/`
- de pages web telles que
  - `www.grappa.univ-lille3.fr/polys/intro-info`
  - `www.histoire-informatique.org`
  - `www.softlord.com/comp`
- de livres : [KRA87], [TW97], [TAN99]

Ces pointeurs ne constituent pas des références incontournables, mais sont pour la plupart de simples résultats de recherche googéliennes.

1.1.1 Première génération - 1945-1955

Pas d'OS. Pour exécuter un programme, il faut réserver l'ordinateur sur une plage horaire et pendant cette plage, tout faire soit même : les programmes sont mis au point par interaction directe (exécution pas à pas, modification directe de la mémoire).

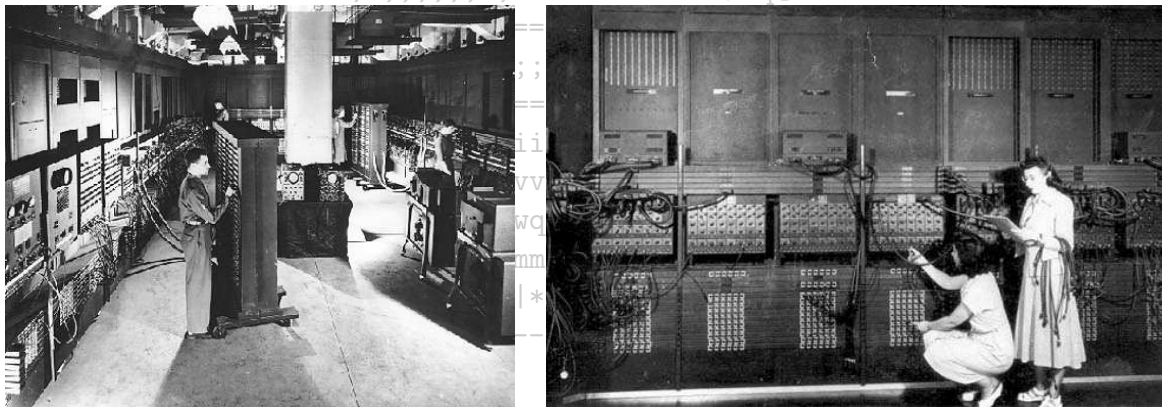


Fig. 1.1 – Seuls face au matériel !

1.1.2 Seconde génération - 1955-1965

– Séparation des constructeurs, programmeurs, opérateurs, service de maintenance.

FIG. 1.2 – Opérateurs en salle des machines

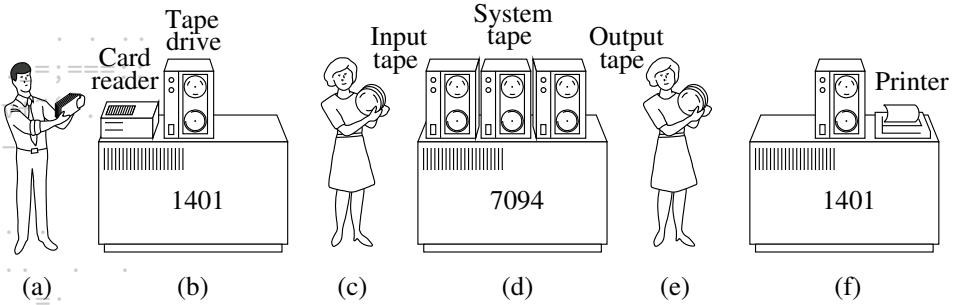


FIG. 1.3 – Traitement par lots [1] : (a) Le programmeur porte sa ou ses cartes au 1401 (b) Le 1401 transfère sur une bande le contenu des cartes (c) Une opératrice porte la bande au 7094 (d) Le 7094 exécute les programme (e) L'opératrice transporte la bande vers le 1401 (f) Le 1401 imprime les résultats

– Apparition du batch : les étapes pour l'exécution d'un programme sont : l'écriture du programme sur papier, le transfert sur carte perforée, la remise à un opérateur, l'exécution du programme, la collecte des résultats par l'opérateur, et la remise des résultats au programmeur. Ce processus long a rapidement conduit à l'apparition du traitement des travaux par lots (batch) pour limiter les attentes. Le principe est de regrouper la collecte des programmes à exécuter ainsi que la distribution des résultats. Un "moniteur d'enchaînement", l'ancêtre des OSs modernes, permettait de lire, de placer en mémoire et d'exécuter séquentiellement l'ensemble des travaux d'un lot. La gestion des ressources (processeur, mémoire, I/O) lui incombait. Il mettait en place des mécanismes de protection des travaux contre les erreurs du programme en cours (limitation du temps de calcul, supervision des I/O, protection de la zone mémoire réservée au moniteur).
  Ces premières fonctions ont nécessité des ajouts aux capacités des machines tels que la gestion d'une horloge, la restriction de certaines instructions et la protection de la mémoire.

[1] Figure tirée de [TW97]

1.1.3 Troisième génération - 1965-1980

- Introduction des circuits intégrés
- Apparition de la multiprogrammation (multiprogramming), des E/S tamponnées (spooling) et du partage du temps (time sharing) :
  - Multiprogramming : lorsqu'un processus procède à des I/O sur un disque, le CPU est innocupé jusqu'à la fin des I/O. Un autre processus pourrait profiter du CPU...C'est ce que permet le multiprogramming. La mémoire est partitionnée et l'OS alloue chaque partition à un processus différent qui peut utiliser le CPU lorsque celui-ci est sous-exploité. C'est l'OS qui décide, via un ordonnanceur, du travail à poursuivre. L'OS/360 d'IBM est le premier à proposer cette fonctionnalité.
  - Spooling (abrév. de Simultaneous Peripheral Operation On-Line) : action de récupérer les données envoyées vers un périphérique, de les mettre dans une file d'attente (servant aussi de tampon), et de ne les envoyer que lorsque que le périphérique est prêt. Utilisé pour le traîtement des tâches puis surtout pour les imprimantes.
  - Time sharing : malgré le gain de temps apporté par le multiprogramming et le spooling dans les «systèmes batch», les temps d'attente étaient toujours longs. Le temps partagé, évolution logique du multiprogramming, a été conçu dans le cadre du partage du temps de calcul d'une machine centrale. Il permet à plusieurs utilisateurs de ce connecter via des terminaux sur la machine centrale. En changeant très souvent de travail courant et en tenant compte des besoins très variables des utilisateurs (compilation très gourmande en temps CPU par opposition à l'édition de fichiers), il donne l'illusion d'une exécution concurrente et simultanée des différents travaux. Le premier système à temps partagé fut CTSS (Compatible Time Sharing System) au MIT.
- Naissance de MULTICS, puis d'Unix et de POSIX : après le succès de l'OS CTSS, le MIT, Bell labs et General Electric décident de se lancer dans le projet d'un nouveau système baptisé MULTICS (MULTiplexed[2] Information and Computing System), permettant de subvenir aux besoins de calculs de milliers d'utilisateurs. Ce fut un fiasco causé d'une part par la chute des prix du matériel (rendant les mini-ordinateurs abordables et le projet obsolète) et d'autre part par la difficulté de l'entreprise, mais de nombreux concepts furent retenus (système de fichiers hiérarchique, processus, mémoire virtuelle). L'un des concepteurs ré-écrivit MULTICS sur un PDP-7 pendant ses heures de loisirs. Ainsi naquit UNICS (UNiplexed Information and Computing System), le grand frère d'Unix. UNICS fut repris comme projet à part entière chez Bell labs pour devenir Unix, avec l'intention de prôner une nouvelle approche pour la conception de logiciel : utiliser plusieurs outils basiques et les faire communiquer grâce aux pipes (alors nouveau concept de communication entre processus), plutôt que construire de grandes applications d'un seul tenant. A l'origine, son code source était disponible. De nombreuses versions d'Unix ont vu le jour, générant beaucoup d'entropie et de confusion. Pour restaurer un peu d'ordre dans la famille des Unix, IEEE développa la norme POSIX (Portable Open System Interface eXchange). POSIX définit un ensemble minimal d'appels système qu'un système Unix doit fournir.

[2] Le multiplexage est un terme qui existe dans le jargon informatique français. On trouve des définitions assez proches :
- Action de faire passer plusieurs communications à travers un seul et même canal. On peut rencontrer du multiplexage temporel ou de fréquence.
- Consiste à partager un même support physique pour obtenir plusieurs liaisons logiques.

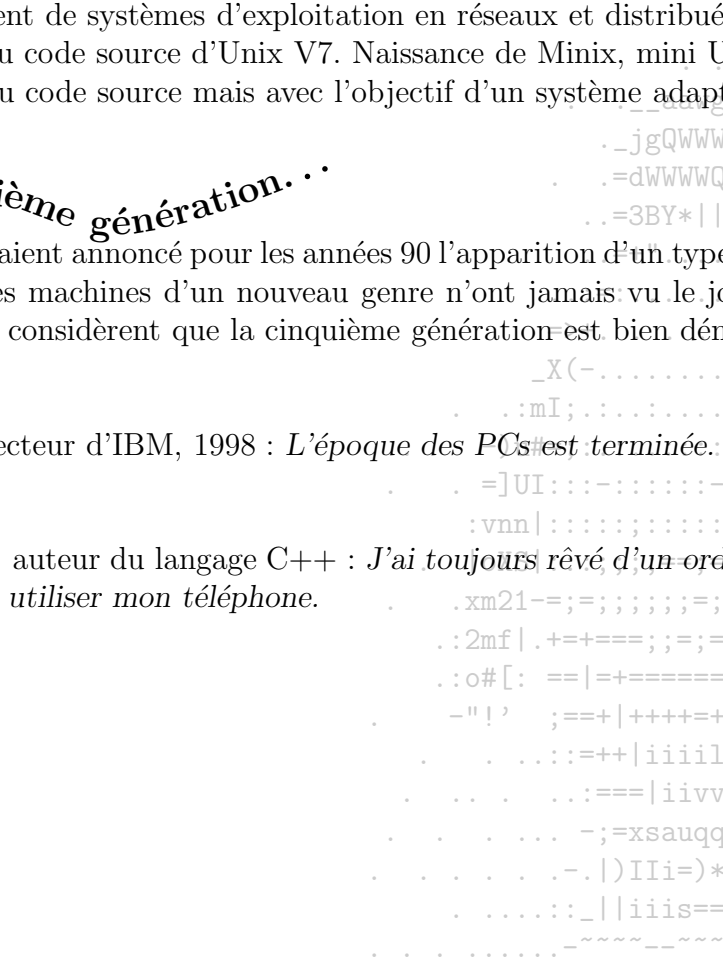
– Prolifération des mini-ordinateurs (principalement la famille PDP(-1 à -11) de DEC).

1.1.4 Quatrième génération - 1980-. . .

- Apparition des microcircuits intégrés
- Mise sur le marché des ordinateurs personnels (PC). Industrie du software “user-friendly”
- Développement de systèmes d’exploitation en réseaux et distribués.
- Protection du code source d’Unix V7. Naissance de Minix, mini Unix à vocation éducative, puis de Linux qui prolonge Minix dans la même logique de partage du code source mais avec l’objectif d’un système adapté à un environnement de production.

1.1.5 Cinquième génération. . .

Les japonais avaient annoncé pour les années 90 l’apparition d’un type d’ordinateurs dit de «cinquième génération» dédié à des applications d’intelligence artificielle, mais ces machines d’un nouveau genre n’ont jamais vu le jour, et les évolutions majeures récentes sont plutôt à chercher du côté d’Internet. Toutefois, certains considèrent que la cinquième génération est bien démarrée avec les ordinateurs dédiés au traitement parallèle.

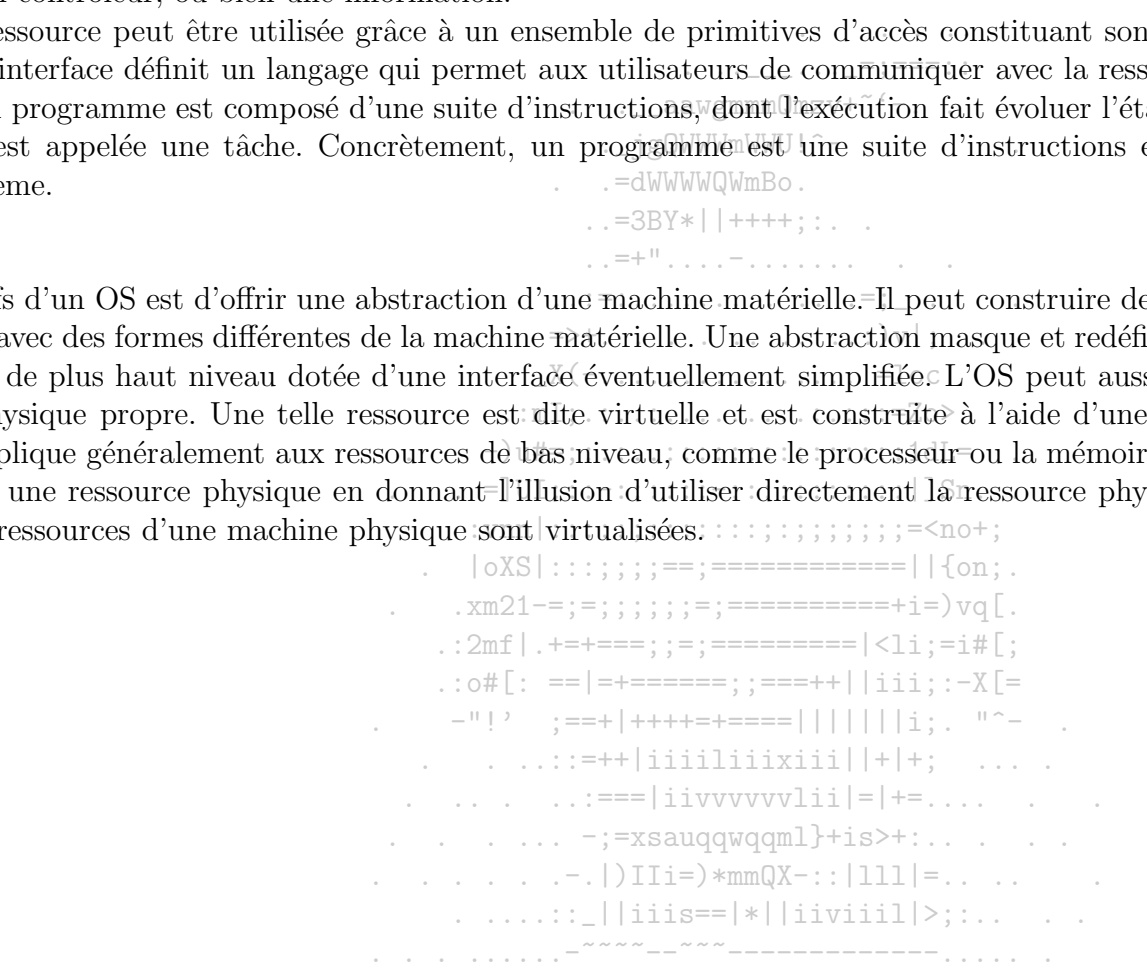


Lou Gerstner, Directeur d’IBM, 1998 : *L’époque des PCs est terminée.*

Bjarne Stroustrup, auteur du langage C++ : *J’ai toujours rêvé d’un ordinateur qui soit aussi facile à utiliser qu’un téléphone. Mon rêve s’est réalisé. Je ne sais plus comment utiliser mon téléphone.*

1.2 Généralités sur les ressources

Ressource : on désigne par le terme ressource, tout objet logiciel ou matériel pouvant être employé par un programme. Une ressource peut être un périphérique ou un contrôleur, ou bien une information.
Interface : une ressource peut être utilisée grâce à un ensemble de primitives d'accès constituant son interface et un ensemble de règles constituant son mode d'emploi. L'interface définit un langage qui permet aux utilisateurs de communiquer avec la ressource.
Programme : Un programme est composé d'une suite d'instructions, dont l'exécution fait évoluer l'état de la machine. L'activité résultant de l'exécution d'un programme est appelée une tâche. Concrètement, un programme est une suite d'instructions effectuant des appels aux interfaces des ressources composant le système.



Un des objectifs d'un OS est d'offrir une abstraction d'une machine matérielle. Il peut construire des abstractions nouvelles qui permettent d'offrir une machine logicielle avec des formes différentes de la machine matérielle. Une abstraction masque et redéfinit l'organisation d'une ressource en définissant une nouvelle ressource de plus haut niveau dotée d'une interface éventuellement simplifiée. L'OS peut aussi simuler le comportement d'une ressource n'ayant pas d'existence physique propre. Une telle ressource est dite virtuelle et est construite à l'aide d'une ou de plusieurs ressources de plus bas niveau. La virtualisation s'applique généralement aux ressources de bas niveau, comme le processeur ou la mémoire. Généralement, l'objectif d'une ressource virtuelle est de multiplexer une ressource physique en donnant l'illusion d'utiliser directement la ressource physique. Par extension, on parle de machine virtuelle lorsque toutes les ressources d'une machine physique sont virtualisées.

1.2.1 Ressources matérielles

Un ordinateur se compose d'un ou de plusieurs processeurs, d'une mémoire principale, d'un ou plusieurs disques constituant la mémoire secondaire, de cartes réseaux, de contrôleurs matériels et de périphériques d'entrées-sorties. Ces différents composants sont reliés par différents bus d'interconnexion (Cf Fig 1.4).

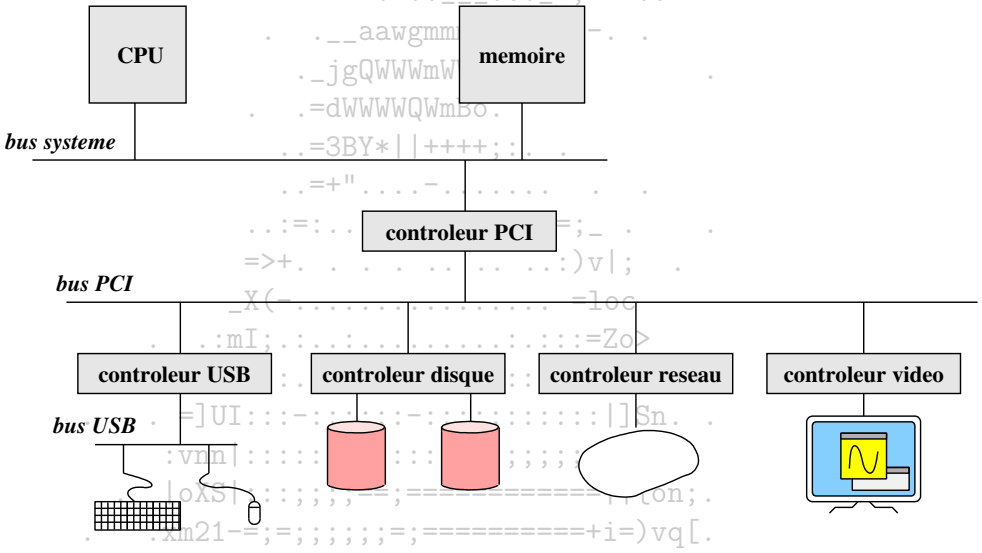


FIG. 1.4 – Architecture d'un ordinateur moderne[3]

Processeur

Un processeur (central processing unit ou CPU) est le composant destiné à interpréter et à exécuter de façon séquentielle les instructions en langage machine d'un programme. Il comporte au moins les entités suivantes :

- *une unité arithmétique et logique (UAL)* dont le rôle est d'effectuer les opérations arithmétiques et logiques comme les additions ou les comparaisons ;
- *une unité de contrôle (UC)* qui extrait les instructions stockées en mémoire, les décode et les exécute en utilisant si besoin l'UAL ;
- *les bus d'adresses et bus de données* reliant les différentes entités ;
- *l'horloge* rythmant le processeur. À chaque cycle d'horloge, le processeur peut effectuer une ou plusieurs opérations (en pipeline ou en superscalaire).

[3]Figure tirée de [FAS01]

Le processeur fournit un modèle de programmation sur lequel repose la conception du système d'exploitation. Ce modèle de programmation peut être caractérisé par les quatre points suivants :

- *un jeu d'instructions* constituant l'interface du processeur ;
- *le contexte du processeur* défini par les registres du processeur. Ce contexte est constitué des registres généraux et des registres spéciaux décrivant l'état du processeur ;
- *exception & événement* : lorsqu'une exception (e.g. un déroutement causé par une division par zéro, un accès mémoire invalide, une instruction invalide,...) ou un événement (e.g. une interruption matérielle déclenchée par un périphérique) surviennent, une commutation de contexte est effectuée par le processeur. Cette commutation interrompt l'exécution séquentielle du programme de façon à exécuter un traitant d'exception (Cf Fig 1.5). Ce traitant d'exception doit être mis en place par le système d'exploitation pour chaque type d'exception supporté par le processeur. Pour pouvoir reprendre l'exécution de la séquence d'instructions interrompue, le système d'exploitation doit, au minimum, sauver le contexte du processeur que le traitant va modifier durant sa propre exécution.

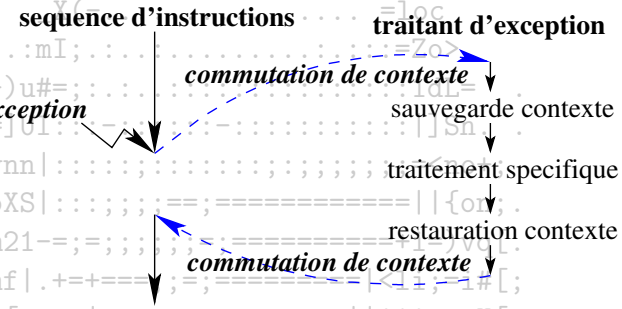


FIG. 1.5 – Commutation de contexte engendré par une exception[4]

- *mode superviseur & mode utilisateur* : Pour des raisons de protection, un processeur fournit au minimum deux modes de fonctionnement ; le mode utilisateur et le mode superviseur. Cela permet de réserver des instructions privilégiées qui sont exécutables uniquement en mode superviseur. Un programme s'exécutant en mode utilisateur peut appeler un service nécessitant l'exécution en mode superviseur en effectuant un appel système. Ce dernier se comporte comme une exception et comme le traitant est mis en place par le système d'exploitation, ce dernier peut ainsi garantir la cohérence globale du système.

[4]Figure tirée de [FAS01]

Mémoire principale

La mémoire principale est une mémoire physique (Random Access Memory ou RAM) interne à la machine. Cette mémoire stocke de manière temporaire des données et les instructions lors de l'exécution des programmes. Le processeur ainsi que les contrôleurs de périphériques peuvent y accéder directement. La mémoire physique est généralement composée des éléments suivants :

- *bus système* assurant la connexion du processeur à la mémoire principale. Sa vitesse détermine le temps d'accès à la mémoire. Comme les processeurs deviennent de plus en plus rapides, il constitue l'un des principaux obstacles à l'amélioration de la vitesse des ordinateurs modernes. Sa lenteur d'accès à la mémoire peut entraîner la consommation de nombreux cycles d'horloge du processeur. La figure 1.6 montre le cheminement des données et de la traduction des adresses entre les différents composants lors de l'accès à la mémoire.

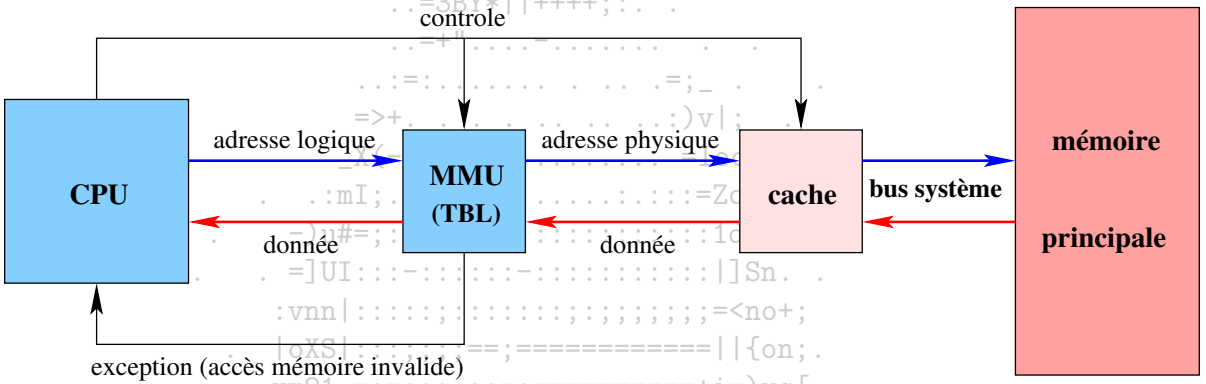


FIG. 1.6 – Flux d'exécution lors de l'accès à la mémoire principale[5]

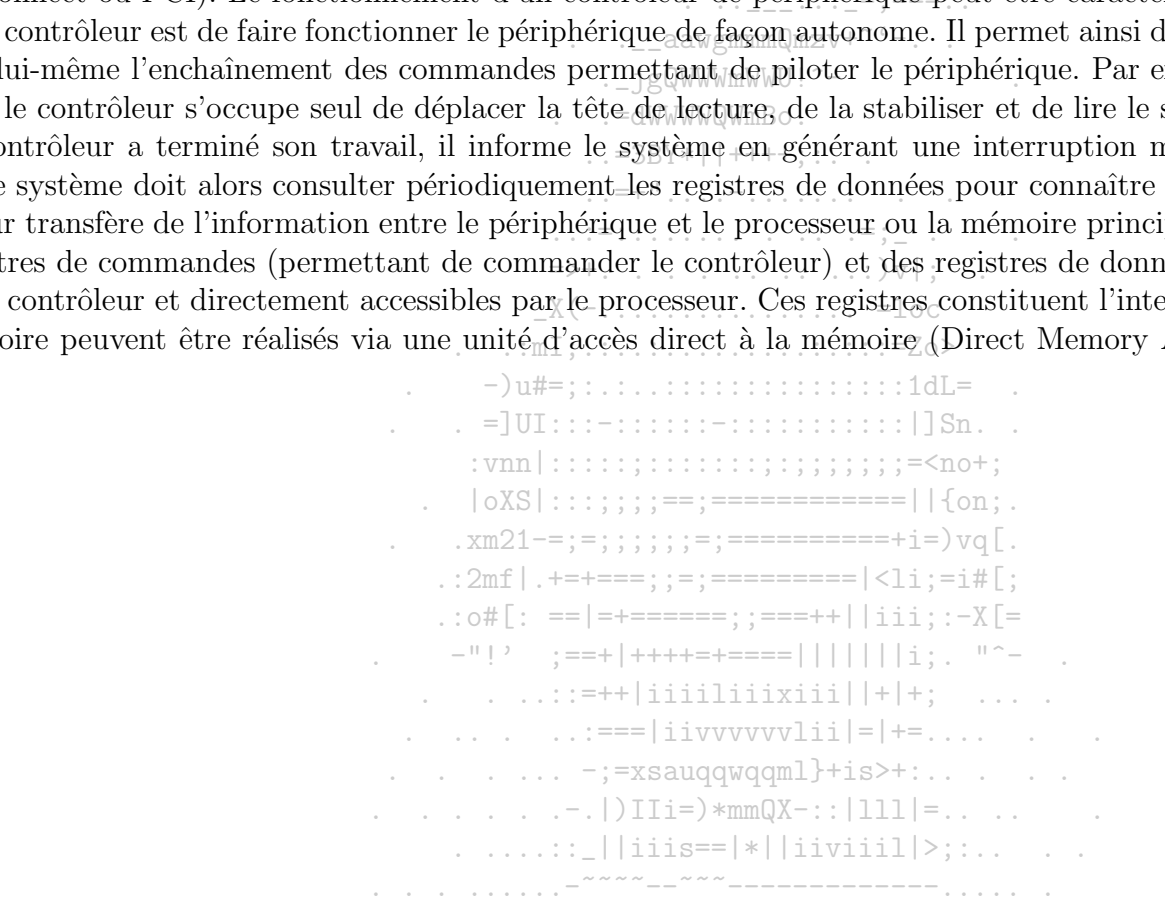
- *mémoire cache*. L'utilisation d'un composant mémoire cache très rapide diminue les temps d'accès à la mémoire en conservant les dernières données accédées. Un tel mécanisme engendre des problèmes de cohérence et de politique de remplacement des données. La mémoire cache peut être interne (cache L1) ou externe (L2) au processeur. Il peut aussi exister des caches non unifiés : un cache pour les instructions et un cache pour les données.
- *unité de gestion de la mémoire (Memory Management Unit ou MMU)*. C'est un composant matériel permettant de traduire les requêtes de lecture ou d'écriture à des adresses logiques (utilisées par le processeur) en des requêtes à des adresses physiques (utilisées par la mémoire principale). Cette translation est réalisée grâce à une table appelée Translation Look-aside Buffer (TLB). Ce composant est utilisé pour construire une mémoire virtuelle et traiter les exceptions lors de l'accès à une adresse logique n'ayant pas de correspondance en mémoire principale. La mémoire virtuelle est divisée en petites unités appelées des pages. Une table de pages permet au système de décrire la mémoire virtuelle. La TLB est en fait un cache de cette table de pages. Généralement, ce composant est intégré au processeur.

[5]Figure tirée de [FAS01]

Périphériques d'entrées-sorties

Les périphériques d'entrées-sorties (I/O device) permettent à la machine de communiquer avec l'extérieur et en particulier avec les utilisateurs. Ces périphériques sont multiples et très variés : interfaces réseaux, disques, écrans, claviers, souris, imprimantes, carte son, etc. Chaque type de périphérique est contrôlé par un dispositif de commande appelé contrôleur. Tous les contrôleurs sont reliés au bus système via un bus, par exemple un bus local (Peripheral Component Interconnect ou PCI). Le fonctionnement d'un contrôleur de périphérique peut être caractérisé de la façon suivante :

- Le rôle de ce contrôleur est de faire fonctionner le périphérique de façon autonome. Il permet ainsi de décharger le processeur durant les entrées-sorties en réalisant lui-même l'enchaînement des commandes permettant de piloter le périphérique. Par exemple, lors d'une requête de lecture d'un secteur d'un disque, le contrôleur s'occupe seul de déplacer la tête de lecture, de la stabiliser et de lire le secteur voulu.
- Lorsqu'un contrôleur a terminé son travail, il informe le système en générant une interruption matérielle. Certains contrôleurs n'offrent pas cette possibilité, le système doit alors consulter périodiquement les registres de données pour connaître l'état du contrôleur.
- Un contrôleur transfère de l'information entre le périphérique et le processeur ou la mémoire principale. Les transferts avec le processeur sont réalisés via des registres de commandes (permettant de commander le contrôleur) et des registres de données (permettant de consulter l'état du contrôleur) offerts par le contrôleur et directement accessibles par le processeur. Ces registres constituent l'interface du contrôleur de périphérique. Les transferts avec la mémoire peuvent être réalisés via une unité d'accès direct à la mémoire (Direct Memory Access ou DMA) qui doit être programmée par le système.



1.2.2 Ressources systèmes

Les ressources systèmes sont des abstractions de bases classiquement offertes par les systèmes d'exploitation. Elles sont soit directement construites au-dessus des ressources physiques, soit construites à partir d'autres ressources abstraites.

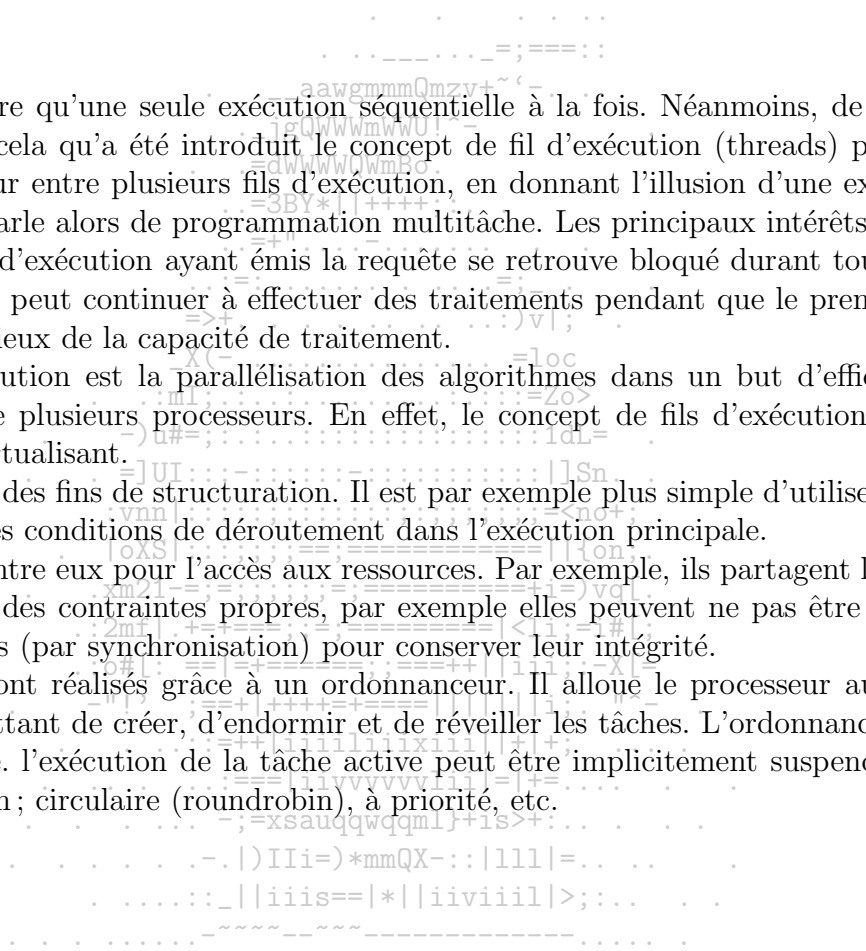
Fil d'exécution (threads)

Un processeur classique ne peut permettre qu'une seule exécution séquentielle à la fois. Néanmoins, de nombreux programmes nécessitent plusieurs chemins d'exécution simultanés. C'est pour cela qu'a été introduit le concept de fil d'exécution (threads) permettant de virtualiser le processeur. Cette virtualisation permet le partage du processeur entre plusieurs fils d'exécution, en donnant l'illusion d'une exécution parallèle. Ainsi, un programme peut être conçu en utilisant plusieurs tâches, on parle alors de programmation multitâche. Les principaux intérêts de cette programmation sont les suivants :

- Lors d'une lecture sur un disque, le fil d'exécution ayant émis la requête se retrouve bloqué durant tout le temps de l'entrée-sortie. Or, en mettant en oeuvre un second fil d'exécution, on peut continuer à effectuer des traitements pendant que le premier est bloqué. Cela permet une plus grande efficacité du système en profitant au mieux de la capacité de traitement.
- La deuxième utilisation des fils d'exécution est la parallélisation des algorithmes dans un but d'efficacité. Néanmoins, cet argument est valable uniquement si la machine est dotée de plusieurs processeurs. En effet, le concept de fils d'exécution n'accélère pas le processeur mais il permet uniquement de mieux l'utiliser en le virtualisant.
- Les fils d'exécution sont aussi utilisés à des fins de structuration. Il est par exemple plus simple d'utiliser un ou plusieurs fils d'exécution pour traiter des événements plutôt que d'ajouter des conditions de déroutement dans l'exécution principale.

Les fils d'exécution sont en concurrence entre eux pour l'accès aux ressources. Par exemple, ils partagent la même mémoire d'exécution du programme. D'autre part, certaines ressources possèdent des contraintes propres, par exemple elles peuvent ne pas être accessibles simultanément. Il peut être alors nécessaire de sérialiser l'accès à ces ressources (par synchronisation) pour conserver leur intégrité.

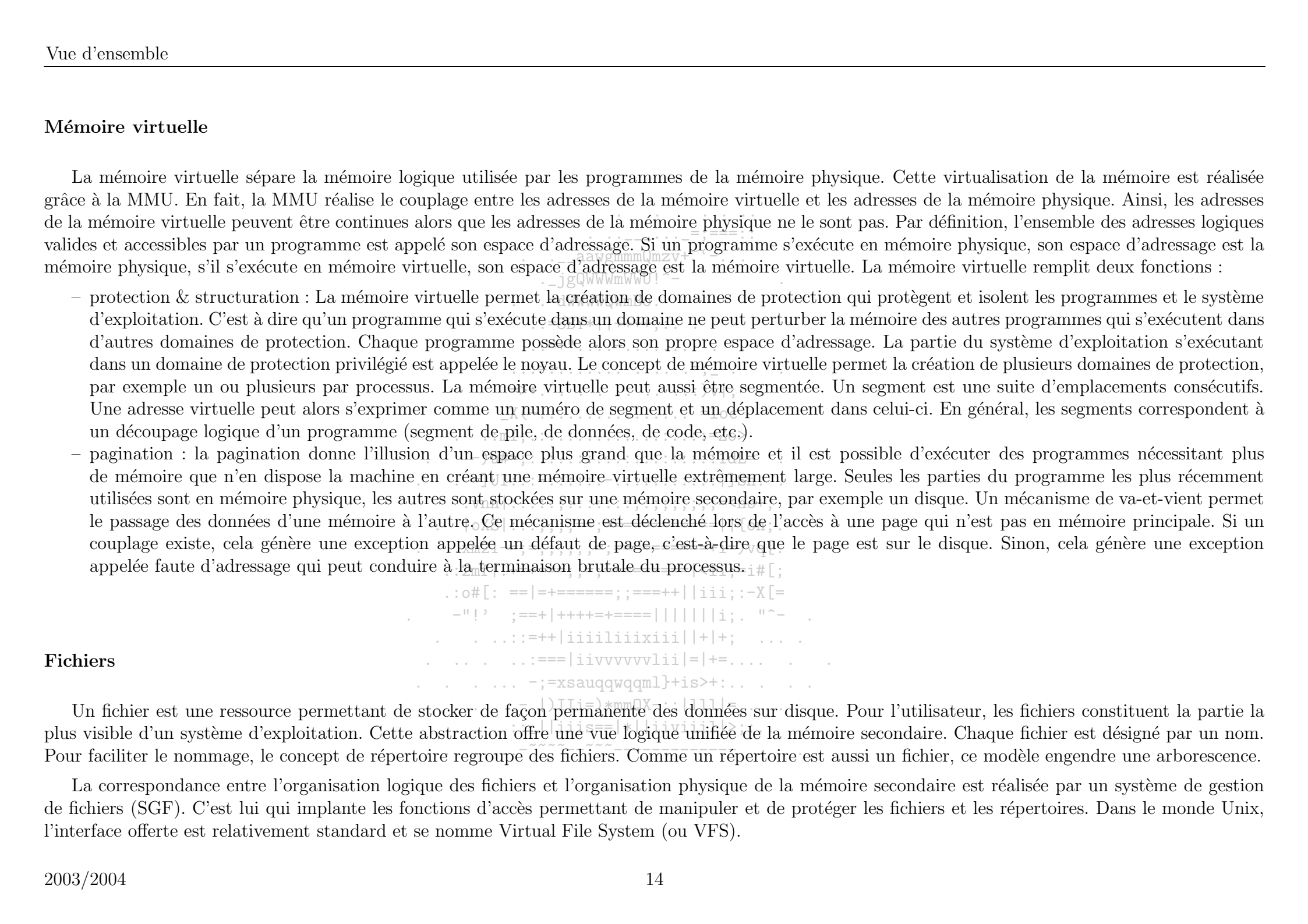
La gestion et le partage du processeur sont réalisés grâce à un ordonnanceur. Il alloue le processeur aux tâches. L'interface d'un ordonnanceur est généralement composée des méthodes permettant de créer, d'endormir et de réveiller les tâches. L'ordonnancement peut être coopératif (i.e. le processeur est ré-alloué explicitement) ou préemptif (i.e. l'exécution de la tâche active peut être implicitement suspendue). Le choix de la tâche à qui est alloué le processeur dépend de la politique d'allocation ; circulaire (roundrobin), à priorité, etc.



Mémoire virtuelle

La mémoire virtuelle sépare la mémoire logique utilisée par les programmes de la mémoire physique. Cette virtualisation de la mémoire est réalisée grâce à la MMU. En fait, la MMU réalise le couplage entre les adresses de la mémoire virtuelle et les adresses de la mémoire physique. Ainsi, les adresses de la mémoire virtuelle peuvent être continues alors que les adresses de la mémoire physique ne le sont pas. Par définition, l'ensemble des adresses logiques valides et accessibles par un programme est appelé son espace d'adressage. Si un programme s'exécute en mémoire physique, son espace d'adressage est la mémoire physique, s'il s'exécute en mémoire virtuelle, son espace d'adressage est la mémoire virtuelle. La mémoire virtuelle remplit deux fonctions :

- protection & structuration : La mémoire virtuelle permet la création de domaines de protection qui protègent et isolent les programmes et le système d'exploitation. C'est à dire qu'un programme qui s'exécute dans un domaine ne peut perturber la mémoire des autres programmes qui s'exécutent dans d'autres domaines de protection. Chaque programme possède alors son propre espace d'adressage. La partie du système d'exploitation s'exécutant dans un domaine de protection privilégié est appelée le noyau. Le concept de mémoire virtuelle permet la création de plusieurs domaines de protection, par exemple un ou plusieurs par processus. La mémoire virtuelle peut aussi être segmentée. Un segment est une suite d'emplacements consécutifs. Une adresse virtuelle peut alors s'exprimer comme un numéro de segment et un déplacement dans celui-ci. En général, les segments correspondent à un découpage logique d'un programme (segment de pile, de données, de code, etc.).
- pagination : la pagination donne l'illusion d'un espace plus grand que la mémoire et il est possible d'exécuter des programmes nécessitant plus de mémoire que n'en dispose la machine en créant une mémoire virtuelle extrêmement large. Seules les parties du programme les plus récemment utilisées sont en mémoire physique, les autres sont stockées sur une mémoire secondaire, par exemple un disque. Un mécanisme de va-et-vient permet le passage des données d'une mémoire à l'autre. Ce mécanisme est déclenché lors de l'accès à une page qui n'est pas en mémoire principale. Si un couplage existe, cela génère une exception appelée un défaut de page, c'est-à-dire que le page est sur le disque. Sinon, cela génère une exception appelée faute d'adressage qui peut conduire à la terminaison brutale du processus.



Fichiers

Un fichier est une ressource permettant de stocker de façon permanente des données sur disque. Pour l'utilisateur, les fichiers constituent la partie la plus visible d'un système d'exploitation. Cette abstraction offre une vue logique unifiée de la mémoire secondaire. Chaque fichier est désigné par un nom. Pour faciliter le nommage, le concept de répertoire regroupe des fichiers. Comme un répertoire est aussi un fichier, ce modèle engendre une arborescence.

La correspondance entre l'organisation logique des fichiers et l'organisation physique de la mémoire secondaire est réalisée par un système de gestion de fichiers (SGF). C'est lui qui implante les fonctions d'accès permettant de manipuler et de protéger les fichiers et les répertoires. Dans le monde Unix, l'interface offerte est relativement standard et se nomme Virtual File System (ou VFS).

Sockets

La notion de socket est une abstraction qui représente la connexion entre deux ordinateurs. Le rôle des sockets est d'offrir une interface permettant d'accéder aux différents protocoles réseaux. Les protocoles réseaux sont des abstractions de communications offrant des services d'adressage, de fiabilité, de contrôle de flux, etc.

Pilotes de périphériques

La partie logicielle permettant de commander un contrôleur de périphérique se nomme un pilote de périphérique (driver). Le pilote utilise directement l'interface fournie par le contrôleur et gère éventuellement la programmation des DMA. Le pilote traîte les interruptions émises par le matériel et détecte et corrige les cas d'erreur. Bien évidemment, il existe un pilote différent pour chaque type de contrôleur. Le pilote offre une interface de plus haut niveau permettant d'effectuer simplement des requêtes sur les périphériques. Par exemple, un pilote de contrôleur d'interface réseau fournit une interface constituée des primitives send et receive.

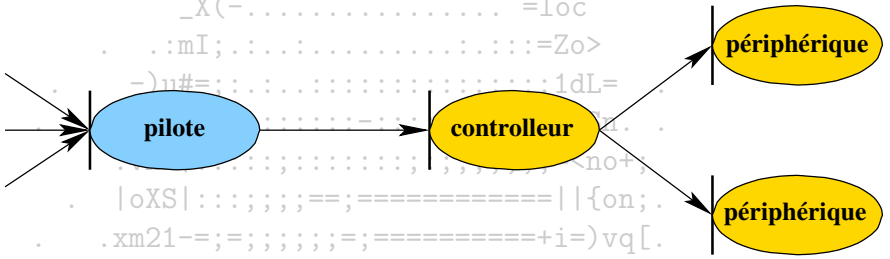


Fig. 1.7 – Abstraction d'un pilote de périphériques[6]

[6]Figure tirée de [FAS01]

1.3 Les processus

Un processus est l'abstraction d'un programme qui s'exécute, c'est l'unité de travail dans un système. Cette abstraction se retrouve dans tous les systèmes d'exploitation. Pour accomplir sa tâche, un processus requiert des ressources : du temps processeur, de la mémoire, des flux de données, etc. Ces ressources lui sont fournies par le système d'exploitation qui joue donc le rôle de multiplexeur de ressources entre les différents processus. Les techniques de multiplexage sont abordées plus loin.

Protection du système d'exploitation

Dans les systèmes d'exploitation multiprogrammés, les processus sont construits en utilisant un ou plusieurs fils d'exécution et un ou plusieurs domaines de protection. Pour des raisons de cohérence de fonctionnement et de protection des processus, l'accès aux ressources et aux informations du système doit être possible uniquement via les interfaces des abstractions fournies par le système d'exploitation. Il est donc nécessaire de protéger le système et de contrôler les appels à ses interfaces. Mais comment garantir cela sachant que le système ne peut avoir qu'une confiance limitée dans les processus ?

Bien que la protection puisse être obtenue par des techniques logicielles, par exemple en se basant sur des langages de programmation sûrs, la manière la plus sûre et la plus efficace consiste à utiliser les mécanismes de protection matériels fournis par le processeur. À cet effet, le processeur fournit trois mécanismes que nous avons déjà abordés : le mode superviseur, l'unité de gestion de la mémoire et l'appel système. Ces mécanismes sont suffisants pour assurer cette protection. Premièrement, un domaine de protection réservé au système d'exploitation garanti que le système ne sera pas corrompu ni espionné de façon volontaire ou involontaire par les processus s'exécutant dans leur propre domaine de protection. Deuxièmement, les exécutions du système en mode superviseur et des processus en mode utilisateur garantissent que les processus ne peuvent accéder directement aux ressources matérielles et en particulier aux informations privilégiées du processeur. Troisièmement, le seul point d'entrée dans le système et donc le seul moyen de passer en mode superviseur étant l'appel système, le système n'a plus qu'à vérifier à ce moment, la validité des appels à ses interfaces pour garantir la protection du système.

Organisation de la mémoire d'un processus

L'espace d'adressage d'un processus est habituellement découpé en deux parties, une pour le domaine de protection du système d'exploitation et une pour le domaine de protection du processus. Chaque partie est appelée un segment et il est possible de définir des droits d'accès sur chaque segment. Le segment du système d'exploitation, placé en début ou en fin de l'espace, est commun à chaque processus et n'est pas accessible par le processus. L'intérêt de ce découpage est de rendre le partage d'information entre le système d'exploitation et le processus aussi efficace que possible. En revanche, un processus organise à sa guise son domaine de protection : son domaine peut être découpé en plusieurs segments, par exemple, un pour le code, un pour la pile, un pour les données statiques et un pour les données dynamiques.

1.4 Généralités sur l'utilisation des ressources

Le rôle d'un système d'exploitation est d'abstraire les ressources en vue de simplifier leur utilisation par les processus. Mais il ne suffit pas de connaître l'existence d'une ressource pour pouvoir l'utiliser. La question est de savoir comment un processus peut et doit utiliser les ressources proposées par le système d'exploitation.

Un processus obtient les ressources en les demandant au système qui les lui alloue dans la mesure du possible. Cette allocation permet au système de partager l'ensemble des ressources entre les différents processus et elle renvoie généralement au processus un nom lui permettant d'accéder à la ressource. Les noms sont utilisés par le système et les processus pour désigner les ressources qui les composent. Comme une ressource peut être utilisée simultanément par plusieurs processus, il est nécessaire de synchroniser les accès à celle-ci. L'opération de synchronisation coordonne les accès à une ressource en vue de garantir son intégrité.

1.4.1 Désignation et liaison

Un système utilise des noms pour désigner les ressources qui le composent. Un nom permet :
- la désignation de la ressource, c'est-à-dire de la distinguer des autres ressources du système ;
- la création d'une liaison pour pouvoir accéder la ressource dans le système.

Noms

A une ressource peuvent être associés différents noms selon le niveau d'abstraction auquel on se place (e.g. un fichier peut être désigné d'au moins quatre manières différentes : par un nom symbolique, par un numéro attribué lors de son ouverture, par l'adresse d'un descripteur en mémoire et par l'adresse d'un descripteur sur disque).

En pratique, la fonction de désignation associe un nom symbolique (par exemple une chaîne de caractères) et un nom interne interprétable par les couches basses du système ou directement par le matériel. Ce nom interne est souvent une adresse, ou encore un identificateur unique à partir duquel le système peut retrouver l'adresse par l'association statique ou dynamique à un nom de niveau inférieur. La suite de relations passant d'un nom symbolique à une ressource qu'il désigne s'appelle la résolution de nom.

Contexte de désignation

Les noms n'existent pas de manière absolue, ils font généralement référence à un contexte de désignation lui-même soumis à une autorité de gestion qui est responsable de la forme et de la création des noms. Cette notion s'applique aussi bien aux noms internes qu'aux noms symboliques. Les contextes de désignation peuvent être hiérarchisés. C'est le cas pour les répertoires dans un système de fichiers. Un contexte de désignation peut aussi restreindre

l'ensemble des noms utilisables à un instant donné par un processus, notamment pour des raisons de protection ou d'efficacité (si un processus ne peut désigner une ressource, il ne pourra y accéder et on évite ainsi des vérifications coûteuses à l'exécution).

Liaison

Une liaison est un canal de communication permettant à un programme d'interagir avec une ressource. Une liaison peut éventuellement être composite et être composée de plusieurs sous liaisons. Selon le moment de liaison, une liaison peut être statique ou dynamique. Dans le premier cas, la liaison est fixée une fois pour toutes, soit lors de l'écriture du programme, soit dans une phase de chargement et d'édition de liens, préalable à l'exécution. Dans ce cas, le nom fait aussi office de liaison, comme par exemple un nom langage. Dans le second cas, la liaison est réalisée au moment de l'exécution, soit lors du premier accès, soit à chaque accès.

La liaison dynamique est nécessaire quand les informations nécessaires à la liaison sont connues uniquement à l'exécution. Par exemple, quand les ressources sont créées dynamiquement. Une méthode générale pour réaliser la liaison dynamique repose sur l'indirection à travers un descripteur dont le nom est connu statiquement et dont le contenu est modifiable.

1.4.2 Allocation

L'allocation de ressource à un processus est implantée par un composant logiciel appelé allocateur. Absolument toutes les ressources sont contrôlées par un allocateur, sachant que toutes les ressources matérielles sont allouées au système lors de son initialisation. L'utilisation d'une ressource peut être découpée en trois phases. Au préalable, une phase d'acquisition permet d'allouer la ressource au processus demandeur. Deuxièmement, la phase d'utilisation proprement dite. Finalement, une phase de libération permettant de rendre la ressource. En fait, un processus peut utiliser une ressource uniquement si elle lui a été allouée par le système d'exploitation. C'est-à-dire si le système a donné au processus des moyens pour accéder à la ressource et qu'il lui a donné un nom ou directement une liaison.

L'allocation peut être temporelle, c'est-à-dire que le processus utilisant la même ressource va changer au court du temps, c'est par exemple le cas pour le processeur. L'allocation peut aussi être un découpage de la ressource, c'est-à-dire que la ressource va être découpée en plusieurs ressources plus petites qui pourront être utilisées par différents processus, le découpage s'effectue selon l'unité d'allocation de la ressource. L'objectif de l'allocation est donc d'utiliser au mieux la ressource selon certaines politiques, par exemple le respect de certaines contraintes de qualité de service.

Acquisition

L'acquisition d'une ressource par un processus peut être explicite ou implicite. Si l'acquisition est implicite, c'est le système qui alloue automatiquement la ressource au processus. Par exemple, lors du démarrage d'un processus, le système lui alloue implicitement de la mémoire, pour ses données et ses instructions, et du temps processeur permettant de commencer son exécution. Si l'acquisition est explicite, l'acquisition doit être explicitement formulée sous la forme d'une requête à l'allocateur de la ressource.

La réaction de l'allocateur d'une ressource à une requête peut prendre différentes formes. Soit la requête peut être satisfaite et la ressource est allouée. Soit la requête ne peut pas être satisfaite, par exemple, si la ressource ou tous ses fragments sont déjà alloués. L'allocateur peut alors refuser l'allocation ou mettre en attente le processus jusqu'à ce qu'une ressource équivalente soit libérée par un autre processus.

Libération

À la fin de l'utilisation de la ressource, le processus doit la libérer. Cette libération peut être explicite ou implicite. La libération explicite est le dual de l'allocation explicite. Par exemple les ressources d'un processus sont libérées automatiquement sans code spécifique, par le système, à la terminaison du processus. La libération implicite peut aussi être une réquisition ou une préemption, c'est à dire un retrait forcé de la ressource par l'allocateur. C'est par exemple ce que fait un ordonnanceur lorsqu'il retire le processeur à un fil d'exécution pour l'allouer à un autre. Cette réquisition est aussi utilisée dans les algorithmes de mémoire virtuelle.

1.4.3 Synchronisation

Nous avons vu qu'un système consiste en un ensemble de processus, ces processus étant eux-mêmes composés d'un ou plusieurs fils d'exécution. Ces fils d'exécution peuvent partager des ressources et accéder de façon concurrente à celles-ci. Lors d'accès concurrents à des ressources il se pose des problèmes d'incohérence. Il faut donc mettre en oeuvre des mécanismes de synchronisation qui permet d'ordonnancer de façon séquentielle les fils d'exécution. Il existe plusieurs mécanismes de synchronisation que nous allons voir ci-dessous. Il faut noter que la synchronisation est inutile s'il n'y a qu'un seul fil d'exécution. La synchronisation est aussi utilisée en interne dans le système d'exploitation pour sérialiser ses accès concurrents aux ressources.

Exclusion mutuelle

L'exclusion mutuelle consiste à étendre à des séquences d'actions la propriété d'indivisibilité des actions du niveau de base. La propriété d'indivisibilité est à la base des mécanismes de synchronisation. La séquence de code exécutée en exclusion mutuelle est appelée une section critique. L'exclusion mutuelle est utilisée pour garantir qu'une ressource n'est pas accédée par plus d'un fil d'exécution à la fois.

De nombreux algorithmes permettent d'assurer l'exclusion mutuelle. Le premier algorithme à attente active pour deux processeurs est celui de Dekker reposant uniquement sur l'indivision des accès en écriture ou en lecture. Une version simplifiée a été proposée par Peterson. Des algorithmes pour n processeurs existent.

Sémaphores

Le sémaphore est un autre mécanisme de synchronisation qui supprime le problème de l'attente active. Le sémaphore est l'association d'un compteur et d'une file d'attente. La file sert à placer les processus bloqués, en attente du signal d'une condition de réveil. L'interface offerte pour les sémaphores se

résume à deux opérations : P (attente) et V (signal). Le compteur est décrémenté lors d'une attente et incrémenté lors d'un signal. L'attente est effective uniquement si le compteur devient négatif. Ainsi, si un signal est exécuté par le processeur avant l'attente, celle-ci ne sera pas bloquante.

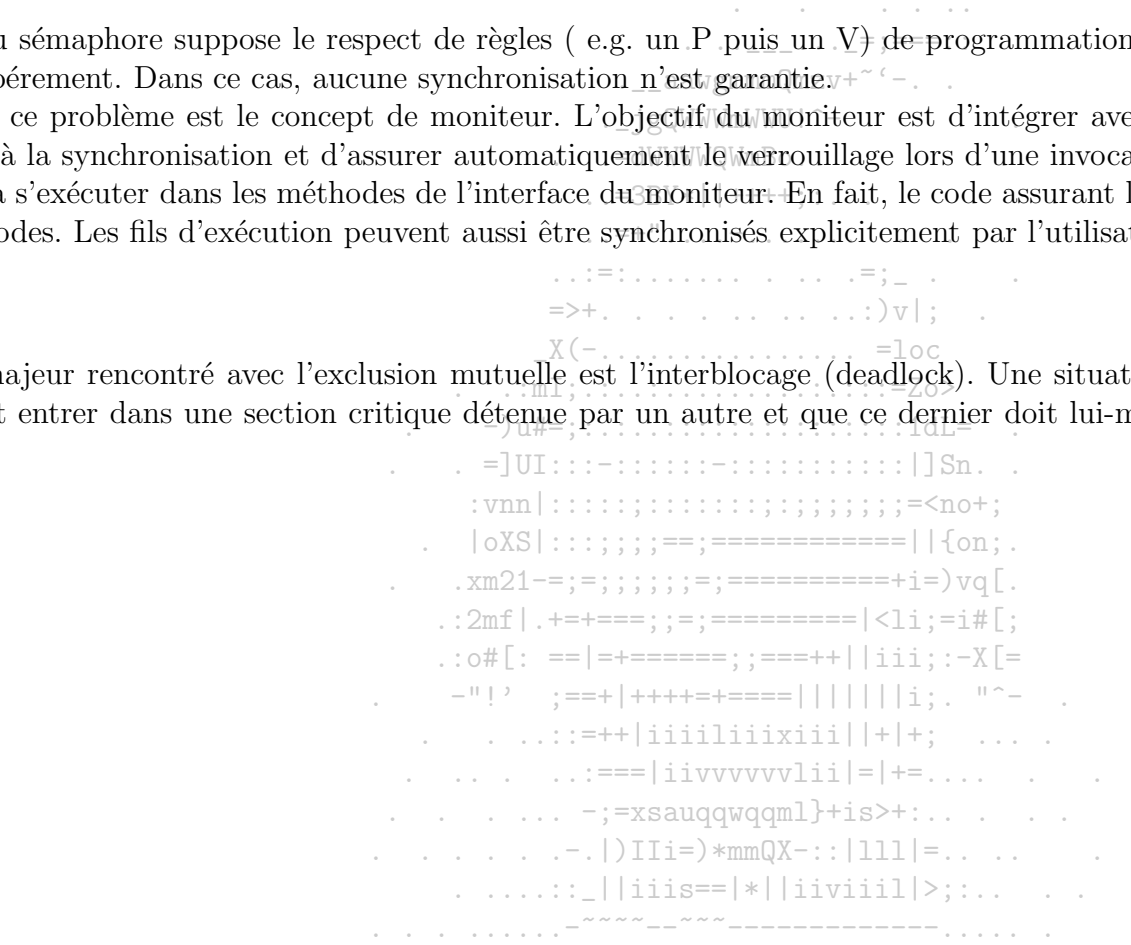
Moniteurs

L'utilisation du sémaphore suppose le respect de règles (e.g. un P puis un V) de programmation auxquelles les programmeurs peuvent ne pas obéir, par erreur ou délibérement. Dans ce cas, aucune synchronisation n'est garantie.

Une solution à ce problème est le concept de moniteur. L'objectif du moniteur est d'intégrer avec les interfaces de la ressource et de façon implicite le code nécessaire à la synchronisation et d'assurer automatiquement le verrouillage lors d'une invocation de méthode. Le moniteur assure que seul un fil d'exécution pourra s'exécuter dans les méthodes de l'interface du moniteur. En fait, le code assurant la synchronisation est ajouté par le compilateur dans l'en-tête des méthodes. Les fils d'exécution peuvent aussi être synchronisés explicitement par l'utilisation de conditions et des méthodes wait et signal.

Interblocage

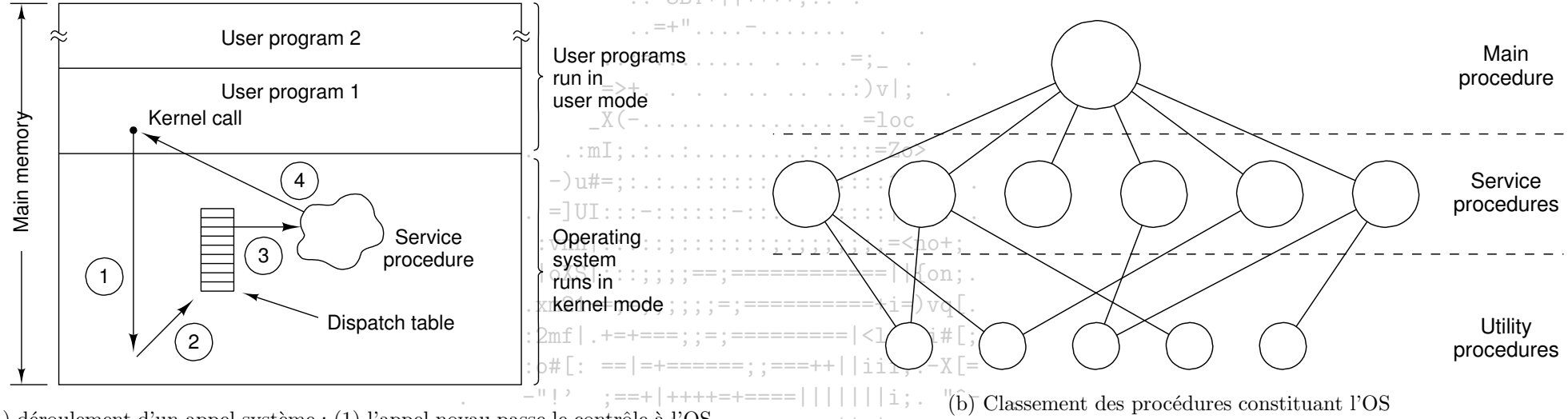
Le problème majeur rencontré avec l'exclusion mutuelle est l'interblocage (deadlock). Une situation d'interblocage se produit par exemple lorsqu'un fil d'exécution doit entrer dans une section critique détenue par un autre et que ce dernier doit lui-même entrer dans une section critique détenue par le premier.



1.5 Structures de OSs

1.5.1 Systèmes monolithiques

Sans véritable structure, l'OS est un ensemble de procédures pouvant s'appeler les unes les autres. Les appels systèmes sont effectués en plaçant leur paramètres dans des registres et en faisant un appel noyau. Cet appel a pour effet de faire basculer du mode esclave au mode maître, ce qui donne le contrôle à l'OS. Celui-ci récupère les paramètres présents dans le registre, identifie la procédure correspondant aux paramètres de l'appel système et l'execute si les paramètres sont corrects. Finalement, l'OS rend le contrôle au programme appelant. Ce fonctionnement permet de classer les différentes procédures en trois



(a) déroulement d'un appel système : (1) l'appel noyau passe le contrôle à l'OS (2) l'OS identifie l'appel système (3) l'OS lance la procédure correspondant à l'appel (4) le contrôle est rendu au programme initial

(b) Classement des procédures constituant l'OS

FIG. 1.8 – Systèmes monolithiques

couches : d'abord un programme principal qui gère les appels systèmes, les procédures correspondant aux actions spécifiques des appels systèmes et enfin des procédures utilitaires implémentant les opérations de bases sur lesquelles les précédentes se fondent. Le point faible de cette absence de structure est l'absence de protection entre les programmes entre eux d'une part, et les programmes et l'OS d'autre part. Cela rend l'exécution simultanée de programmes hasardeuse...

1.5.2 Systèmes en couches

Un de objectifs des systèmes en couche est de renforcer la protection. THE (Technische Hogeschool Eindhoven) est le premier OS a utiliser le concept de couche, mais la structuration en couche reste logicielle. Il n'y avait pas encore les protections matérielles nécessaires. Multics est le premier OS à réellement

5	Utilisateur
4	Processus utilisateur
3	gestion des I/O
2	communication entre processus et terminaux
1	gestion de la mémoire
0	allocation du processeur

implanter un OS multi-couches. Le matériel proposait alors plusieurs mécanismes de protection, notamment au niveau de la mémoire. Le gain en sécurité et en fiabilité a permis de construire les premiers systèmes à temps partagé.

La structuration en couche consiste à structurer l'OS en couches numérotées de 0 à n. Cette conception vise à décomposer le problème en une suite de sous-problèmes supposés plus simples à résoudre. Idéalement, chaque couche s'appuie sur la couche qui lui est immédiatement inférieure, i.e. les appels sont, en théorie, possibles uniquement entre deux couches successives.

Cette architecture a évolué vers un modèle à deux couches, ou plutôt anneaux : l'anneau 0 réservé au noyau de l'OS et l'anneau 1. Le noyau est un ensemble minimum de fonctions de l'OS sur lesquelles les autres fonctions de l'OS reposent. Pour sa protection, le noyau s'execute toujours dans l'anneau 0 et en mode surperviseur. Il a ainsi accès à toutes les ressources avec tous les droits. La couche noyau s'interpose entre le matériel et les processus correspondant à des applications ou des fonctions systèmes. L'appel du noyau par un processus est un appel système. Le flux inverse prend la forme d'un

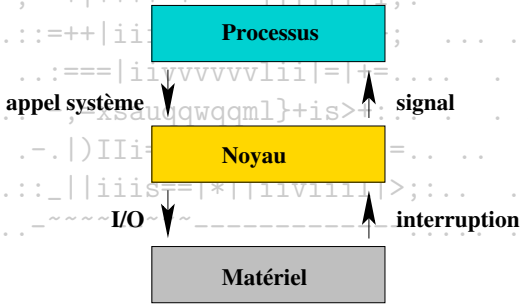


FIG. 1.9 – Communication dans les OS à noyau

signal. L'appel du matériel par le noyau est une entrée/sortie. Le matériel génère des interruptions pour communiquer avec le noyau.

1.5.3 Machines virtuelles

Une des premiers OS à gérer le concept de machine virtuelle a été l'adaptation temps partagé de l'OS/360 d'IBM, proposé vers 1968 sous le nom de CP/CMS, puis sous le nom de VM/370 en 1979.

Le cœur de l'OS, appelé moniteur de machine virtuelle ou VM/370, s'exécute à même le matériel et fournit à la couche supérieure plusieurs machines virtuelles. Ces machines virtuelles sont des copies conformes de la machine réelle avec ses interruptions, ses modes noyau/utilisateur, etc...

Chaque machine virtuelle peut exécuter son propre OS. Lorsqu'une machine virtuelle exécute en mode interactif un appel système, l'appel est analysé par le moniteur temps partagé de cette machine, CMS. Toute instruction d'I/O, toute instruction d'accès mémoire est convertie par VM/370 qui les exécute dans sa simulation du matériel. La séparation complète de la multiprogrammation et de la machine étendue rend les éléments du OS plus simples et plus souples. VM/370 a gagné en simplicité en déplaçant une grande partie du code d'un OS dans le moniteur CMS.

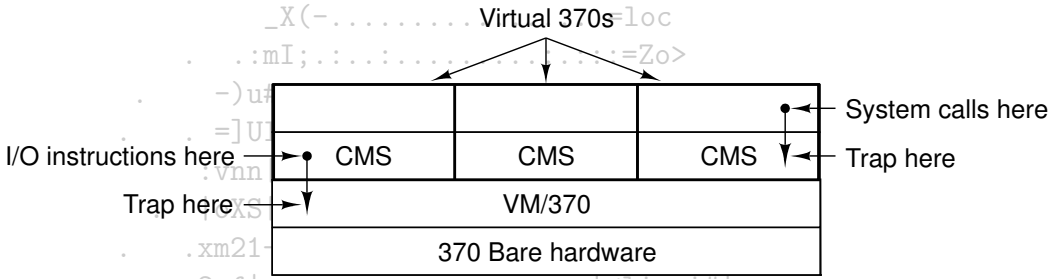


FIG. 1.10 – Systèmes en couches

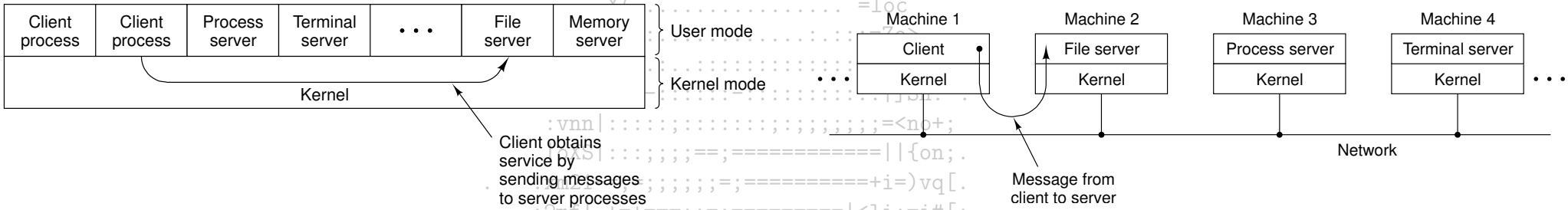
1.5.4 Architecture client-serveur

Cette tendance s'est accentuée dans les OS contemporains en tentant de réduire le OS à un noyau minimal. Une des formes les plus accentuées de cette évolution est l'architecture client/serveur.

La plupart des fonctionnalités d'un OS sont reportées dans des processus utilisateurs. Pour demander un service comme la lecture d'un bloc de fichier, le processus utilisateur ou processus client envoie une requête à un processus serveur qui effectue le travail et envoie une réponse. Le noyau ne gère que la communication entre les clients et les serveurs. Cependant, le noyau est souvent obligé de gérer certains processus serveurs critiques comme les pilotes de périphériques qui adressent directement le matériel.

La décomposition du OS en modules très spécialisés le rend facile à modifier. Les serveurs s'exécutent comme des processus en mode utilisateur et non pas en mode noyau. Comme ils n'accèdent donc pas directement au matériel, une erreur n'affecte que le serveur et pas l'ensemble de la machine.

En outre, ce modèle est bien adapté aux systèmes distribués. Un client n'a pas besoin de savoir si le OS fait exécuter sa requête par un serveur de sa propre machine ou celui d'une machine distante.



(a) déroulement d'un appel système : (1) l'appel noyau passe le contrôle à l'OS (2) l'OS identifie l'appel système (3) l'OS lance la procédure correspondant à l'appel (4) le contrôle est rendu au programme initial

(b) Classement des procédures constituant l'OS

FIG. 1.11 – Systèmes monolithiques

Première partie

Processus

Chapitre 2

Processus : définition, représentation et parallélisation

Les premiers ordinateurs ne permettaient que l'exécution d'un programme. Le programme avait alors le contrôle complet du système. De nos jours, les ordinateurs permettent à plusieurs programmes de s'exécuter de façon concurrente. Cette évolution nécessite un meilleur contrôle et une isolation de certains programmes. Ces besoins ont conduit à l'émergence du concept de processus. Les processus du système d'exploitation exécutent des codes systèmes et les processus utilisateurs exécutent des codes utilisateurs.

Divers termes se rencontrent dans la littérature ou l'usage courant : processus, job, tâche...Tous pointent vers le concept de processus que nous définissons précisément ci-après.

2.1 Définition

Un raccourci pour simplifier le concept de processus consiste à le présenter comme une instance de programme en exécution. De façon plus précise, un processus est l'ensemble composé des éléments suivants :

- le code du programme exécuté
- un compteur ordinal indiquant l'adresse de la prochaine instruction à exécuter
- le contenu des registres du CPU associés à son exécution
- la pile contenant des données temporaires telles que des variables locales
- les données de variables globales nécessaires à son exécution

Un processus fournit l'image de l'état d'avancement de l'exécution d'un programme.

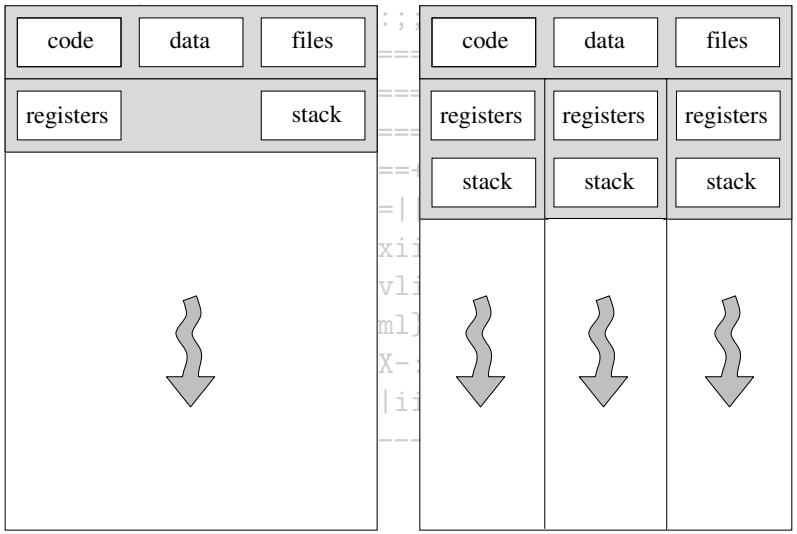
Un processus est à un instant précis dans un état sujet à variation. Les différents états varient d'un système à l'autre, mais on retrouve de façon universelle les états suivants :

– élu - le processus est en cours d'exécution sur le CPU
– bloqué - le processus est en attente d'un signal extérieur, d'une ressource autre que le CPU...lorsque l'attente prend fin, le processus passe dans l'état prêt
– prêt - le processus suspendu est en attente du CPU

2.2 Digression sur les threads

La définition d'un processus ci-dessus sous-tend la modélisation des processus pour leur gestion par les OSs. Ainsi, la plupart des OSs implémentent l'ensemble des informations liées à un processus par des blocs de contrôle de processus (PCB ou process control bloc en anglais), ces blocs étant eux même regroupés sous forme de table, etc. Ces détails seront repris plus tard (cf chap. 6).

Cette présentation convient parfaitement aux processus classiques comportant un seul fil d'exécution (mono-thread). En revanche, pour la gestion des threads au sein d'un OS, certains problèmes se posent, nécessitant des modélisations et des politiques de gestions particulières...Ces aspects seront éventuellement considérés plus loin. Le lecteur peut se pencher sur [SGG03].



2.3 Représentation des processus

Un processus est un programme en exécution. Il correspond à une suite d'instructions à exécuter. Mais ces instructions sont-elles des instructions d'un langage de programmation, du langage machine ou quelque chose d'intermédiaire ? Afin de rester à un niveau conceptuel général, nous ferons reposer cette présentation sur la notion de tâche : une tâche est une unité élémentaire de traitement. Le niveau auquel on fait référence est laissé libre. Un processus P peut être associé à un ensemble d'exécutions de tâches T_1, T_2,...T_n.

Les exécutions des tâches peuvent être séquentielles ou parallèles. Dans le cas d'un processus séquentiel, la notation (abusive) $P = T_1 T_2 \dots T_n$ permet de décrire le processus P comme étant la suite des exécutions des tâches T_1, T_2,...T_n. La description de processus pour lesquels les exécutions de certaines tâches sont parallélisables fait intervenir davantage de structures :

- A chaque tâche T_i sont associés deux événements :
  - d_i, le début de l'exécution de la tâche T_i qui correspond à la lecture des paramètres d'entrée, l'acquisition des ressources nécessaires à l'exécution, l'initialisation de ces ressources et éventuellement un chargement d'informations.
  - f_i, la fin de l'exécution de la tâche T_i, i.e. l'écriture des résultats, la libération des ressources acquises et éventuellement la sauvegarde d'informations.
- L'ensemble E des tâches du processus muni d'une relation $<$ de précédence permet de décrire certaines contraites sur les exécutions des tâches. Cette relation vérifie :
  - pour tout élément $T \in E$, il n'y a pas $T < T$.
  - pour tout couple (T, T'), il n'y a pas simultanément $T < T'$ et $T' < T$.
  - pour tous T, T' et T'' vérifiant $T < T'$ et $T' < T''$, on a $T < T''$.

$T < T'$ signifie que le début de l'exécution de T' ne peut avoir lieu avant la fin de l'exécution de T. T est dit prédécesseur de T' et T' est dit successeur de T. Si ni $T < T'$, ni $T' < T$ ne sont vérifiées alors T et T' sont parallélisables. L'information donnée par le couple $(E, <)$ est souvent représenté par son graphe de précédence (afin de supprimer toute redondance) :

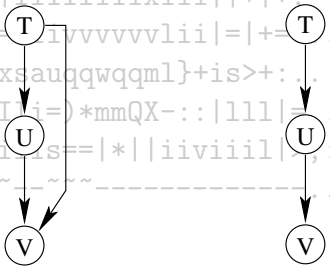
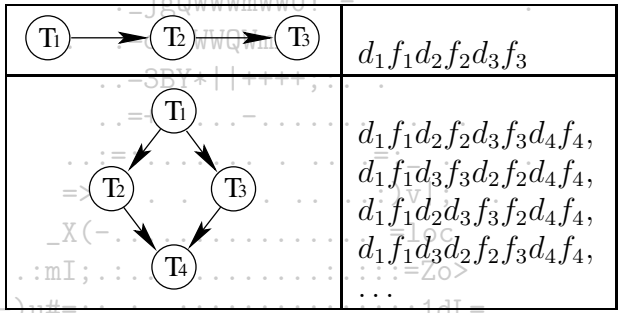


FIG. 2.1 – Le graphe d'une relation de précédence et son graphe de précédence

– Etant donné le système de tâches $(E, <)$, considérons le langage définit par les mots construits à partir de l'alphabet $A = \{d_1, f_1, d_2, \dots d_n, f_n\}$ et vérifiant
  - pour tout $T \in E$, tout mot contient exactement une occurence de d et une occurence de f au sein de tout mot.
  - pour tout $T \in E$, l'occurence de d est située avant celle de f au sein de tout mot.
  - $T < T'$ si et seulement si l'occurence de f est située avant celle de d'.

Ces mots correspondent aux comportements possibles du système de tâches $(E, <)$.



Tab. 2.1 – Exemples de systèmes et comportements

Le langage du système décrit tous les comportements possibles du système.
Etant donnés deux systèmes de tâches, il est naturel de pouvoir les exécuter de façon séquentielle ou parallèle pour créer un nouveau système de tâches plus complexe. Ceci nous conduit aux opérations de composition parallèle et de produit de deux graphes de précédence. La composition parallèle de deux graphes G_1 et G_2 est l'union de G_1 et G_2. Le produit du graphe G_1 par le graphe G_2 est le graphe qui reprend les contraintes de G_1 et G_2 auxquelles est ajoutée la contrainte qu'aucune tâche de G_2 ne peut débuter avant la fin de toutes les tâches de G_1.

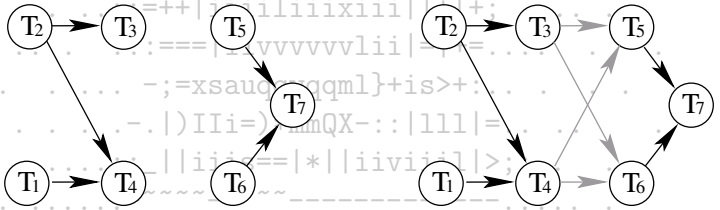
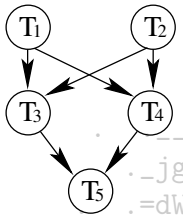


Fig. 2.2 – Opérations sur les graphes de précédence

Pb : Comment sont définis les langages des systèmes obtenus ?

Un système de tâches comportant des tâches dont les exécutions peuvent être parallèles est bien décrit par son graphe de précédence. Pour le décrire dans un langage de programmation évolué, les balises **parbegin** et **parend** sont utilisées.



$\Leftrightarrow$ parbegin T_1, T_2 parend
parbegin T_3, T_4, parend
T_5

Supposons que les tâches ci-dessus soient définies comme suit :

- $T_1 : X := 1$
- $T_2 : Z := 10$
- $T_3 : X := X + Z$
- $T_4 : Y := X + Z$
- T_5 : Afficher Y

Pour le comportement $d_1 f_1 d_2 f_2 d_3 f_3 d_4 f_4 d_5 f_5$, le résultat final est 21. En revanche, pour $d_1 f_1 d_2 f_2 d_4 f_4 d_3 f_3 d_5 f_5$, le résultat final est 11 ! Les comportements du système considéré ne conduisent pas tous au même résultat final. Le système n'est donc pas déterministe. Il est dit non déterminé ou indeterminé.

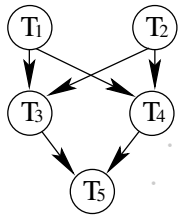
Ainsi, notre façon de décrire un processus inclus la description de processus non déterministes, alors que nous nous intéressons de façon implicite aux processus déterministes. Avant de passer à l'étude du caractère déterminé d'un système, signalons qu'il existe d'autres façons de modéliser des processus : les automates finis et les réseaux de Pétri. Ces modèles sont bien adaptés à certains types de processus, mais présentent des particularités qui nous éloignent de la généralité que nous cherchons. Nous renvoyons les curieux à [BB93] pour un aperçu et d'autres références.

2.4 Déterminisme des processus

Le problème du déterminisme d'un processus est inhérent à l'exécution en parallèle de plusieurs tâches. En effet, un processus pour lequel les tâches sont exécutées séquentiellement est déterminé (aux interférences avec d'autres processus avec lesquels il est en concurrence près). On ne veut cependant pas renoncer à la possibilité de paralléliser des tâches car une grande parallélisation des tâches permet une augmentation du degré de multiprogrammation et donc un meilleur rendement du CPU. Il est donc important de savoir maîtriser le caractère déterministe d'un système de tâches et la parallélisabilité de ses tâches.

2.4.1 Etats d'un système de tâches

Reprenons notre dernier exemple de système de tâches :



avec parbegin $T_1 : X := 1 ; T_2 : Z := 10$ parend
parbegin $T_3 : X := X + Z ; T_4 : Y := X + Z$ parend
T_5 : Afficher Y

Le problème vient du fait que l'information manipulée par les tâches T_4 et T_3 est partagée sans contrôle. On dit qu'elles sont interférentes.

Supposons que la mémoire centrale soit constituée de m cellules. Nous posons $M = (C_1, \ldots, C_m)$. Considérons le comportement $w = a_1 a_2 \ldots a_l$. Nous définissons l'état s_k dans lequel se trouve le système après l'événement a_k par le m-uplet des valeurs contenues dans chaque cellule de la mémoire : $s_k = [C_1(k), \ldots, C_m(k)]$. Nous notons $s_0 = [C_1(0), \ldots, C_m(0)]$ l'état initial des cellules avant exécution.

Ecrivons les états du système ci-dessus pour le comportement $d_1 f_1 d_2 f_2 d_3 f_3 d_4 f_4 d_5 f_5$:

	d_1	f_1	d_2	f_2	d_3	f_3	d_4	f_4	d_5	f_5
$C_1 \equiv X$	0	0	1	1	1	1	11	11	11	11	11
$C_2 \equiv Y$	0	0	0	0	0	0	0	0	21	21	21
$C_3 \equiv Z$	0	0	0	0	10	10	10	10	10	10	10

A chaque tâche T est associé son domaine de lecture $L_T = \{C_{i_1}, \ldots, C_{i_\alpha}\}$ i.e. l'ensemble des cellules qui sont lues par la tâche, et son domaine d'écriture $E_T = \{C_{j_1}, \ldots, C_{j_\beta}\}$ i.e. l'ensemble de cellules modifiées par la tâche. Ainsi, chaque tâche T réalise une fonction F_T de son domaine de lecture vers son domaine d'écriture qui à un p-uplet de valeurs d'entrée fait correspondre un q-uplet de valeurs de sortie.

Dans notre illustration, T_4 a pour domaine de lecture $\{X,\ Z\}$ et pour domaine d'écriture $\{Y\}$, et elle réalise la fonction telle que $C_2 := C_1 + C_3$.

Remarquons qu'un système de tâches est entièrement défini par la donnée de son graphe de précédence, les domaines de lecture et d'écriture de chaque tâche et les fonctions associées. A partir de ces éléments, nous pouvons calculer tous les états liés à un comportement w donné. En général, au lieu de considérer la suite entière que prend une cellule C_j, on retire les valeurs correspondant à des intervalles pendant lesquels aucune tâche n'a écrit dans C_j. On la note $V(C_j, w)$. Ici, par exemple, nous avons $V(C_1, w) = \{0, 1, 11\}$.

2.4.2 Système déterminé et conditions de Bernstein

Un système de tâches déterminé est un système $S = (E, <)$ tel que, pour tous comportements w et w' et pour toute cellule C de la mémoire, les suites $V(C, w)$ et $V(C, w')$ sont identiques.

Rem : Remarquons qu'un processus séquentiel est bien déterminé selon cette définition puisqu'il n'admet qu'un seul comportement.

Notre exemple illustre un cas de processus non déterminé.

Soit un système de tâches $S = (E, <)$. Deux tâches T et T' de E sont dites non-interférentes vis-à-vis de S si elles remplissent l'une des conditions de Bernstein :
- soit T est un prédécesseur ou un successeur de T'
- soit $L_T \cap E_{T'} = E_T \cap L_{T'} = E_T \cap E_{T'} = \emptyset$

Dans notre exemple, les tâches T_1 et T_2 sont non-interférentes car elles remplissent la première condition de Bernstein. De même pour les tâches T_1 et T_3 qui se succèdent. En revanche, T_3 et T_4 interfèrent car $X \equiv C_1 \in E_{T_3} \cap L_{T_4}$.

Soit $S = (E, <)$ un système de tâches.

i) Si le système S est constitué de tâches deux à deux non-interférentes, alors il est déterminé pour toute interprétation.

ii) Si le système S est déterminé et si pour chaque tâche T de E, le domaine d'écriture E_T est non vide, alors les tâches de S sont deux à deux non-interférentes.

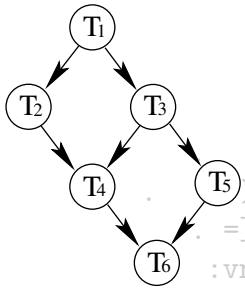
Rem : La condition de non-interférence est plus forte que le caractère déterminé d'un système. Ceci est dû au fait que, si une tâche n'écrit pas dans au moins une cellule, elle peut interférer avec d'autres tâches sans qu'il soit possible de le détecter en observant les valeurs écrites.

2.5 Parallélisme maximal

Est-il possible d'améliorer la parallélisation d'un processus sans en changer la nature ? Un processus utilise-t-il de façon optimale la multiprogrammation ? A partir des concepts précédents, nous allons préciser le sens du mot nature et étudier la parallélisabilité des tâches d'un processus.

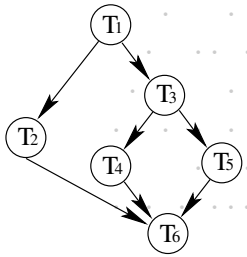
Deux systèmes de tâches $S = (E, <)$ et $S' = (E, <')$ sur le même ensemble de tâches sont dit équivalents s'ils sont déterminés et si, pour tous comportements w de S et w' de S' et pour toute cellule C de la mémoire, les suites $V(C, w)$ et $V(C, w')$ sont identiques.

Un système de tâches de parallélisme maximal est un système déterminé dont le graphe de précédence G vérifie la propriété suivante :
– la suppression de tout arc (T, T') du graphe G entraine l'interférence des tâches T et T'.



Cellule	dans le domaine de lecture de	dans le domaine d'écriture de
C_1	T_1, T_6	T_2
C_2	T_6	T_4
C_3	T_2, T_3	
C_4	T_2, T_3, T_4	T_1, T_6
C_5	T_5	T_3, T_5

1. Vérifier que le système est bien déterminé.
2. Le système est-il de parallélisation maximale ? Considérer la suppression de l'arc (T_2, T_4).



Cellule	dans le domaine de lecture de	dans le domaine d'écriture de
C_1	T_1, T_6	T_2
C_2	T_6	T_4
C_3	T_2, T_3	
C_4	T_2, T_3, T_4	T_1, T_6
C_5	T_5	T_3, T_5

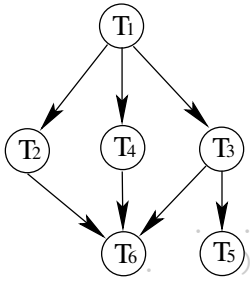
3. Vérifier que le système obtenu est bien équivalent au premier.

Nous disposons du résultat suivant pour la parallélisation maximale d'un processus :

Soit $S = (E, <)$ un système déterminé. Il existe un unique système S' de parallélisme maximal équivalent à S. Ce système est défini comme le système $S' = (E, <')$ où $<'$ est la fermeture transitive de la relation R suivante :

$$R = \{(T, T') | T < T' \text{ et } (L_T \cap E_{T'} \neq \emptyset \text{ ou } E_T \cap L_{T'} \neq \emptyset \text{ ou } E_T \cap E_{T'} \neq \emptyset) \text{ et } E_T \neq \emptyset \text{ et } E_{T'} \neq \emptyset\}$$

Vérifier que le système suivant est un système équivalent au système précédemment considéré et de parallélisme maximal :



Cellule	dans le domaine de lecture de	dans le domaine d'écriture de
C_1	T_1, T_6	T_2
C_2	T_6	T_4
C_3	T_2, T_3	
C_4	T_2, T_3, T_4	T_1, T_6
C_5	T_5	T_3, T_5

Chapitre 3

Ordonnancement des processus

3.1 Ordonnancement ?

L'ordonnancement des processus est un sous problème de l'allocation de ressources. Les ressources peuvent être le CPU, l'espace mémoire, un canal d'I/O...Certaines d'entres elles peuvent être retirées au processus de façon provisoire. Les ressources préemptibles sont le CPU, les canaux d'I/O. D'autres ne peuvent être reprises jusqu'à ce que le processus les rende. C'est le cas de la mémoire, des fichiers, ...

L'OS a deux activités principales concernant les ressources :

- l'allocation : pour répondre au problème "Qui a quoi ?"
- l'ordonnancement : pour résoudre le problème associé "Combien de temps la ressource est-elle utilisée par ce processus ?"

Il y a un mécanisme matériel qui permet à l'OS d'opérer l'ordonnancement de la ressource CPU. Chaque processus étant unique et imprévisible du point de vue de son temps d'exécution, il présente le risque de monopoliser le CPU trop longtemps. Pour éviter les situations de bloquage du CPU par un processus, la plupart des ordinateurs ont une horloge qui génère des interruptions périodiquement. A chacune des interruptions, l'OS reprend le contrôle du CPU et reconsidère l'allocation du CPU.

L'ordonnancement de processus peut être considéré à différents niveaux :

- Court terme : si l'OS est préemptif, à chaque interruption d'horloge, l'allocation du CPU est reconsidérée. Quel processus choisir ? Le choix est construit par un algorithme.
- Moyen terme : concerne les problèmes de gestion de mémoire, car il peut être meilleur de placer un processus hors de la mémoire principale pour faciliter l'exécution d'autres processus...

– Long terme : faut-il admettre l'exécution d'un processus dans les cas critiques où trop de processus sont en cours, ou dans le cas où un utilisateur a déjà lancé un nombre important de processus ? ...

Nous allons nous focaliser ici sur l'ordonnancement de processus à court terme en étudiant les algorithmes classiques d'ordonnancement permettant à l'ordonnanceur (scheduler) de définir l'ordre dans lequel les processus prêts utilisent l'UC et la durée d'utilisation. Le but dans ce cas est une bonne utilisation de la ressource CPU (i.e. un fort taux d'occupation ou un faible taux d'inactivité) grâce à la multiprogrammation. En recouvrant les phases d'I/O d'un processus par les phases de calculs des autres processus, le taux d'utilisation du CPU est augmenté et le débit des processus est accru.

Rem : dans le cas d'une architecture multiprocesseur, c'est le répartiteur (dispatcher) qui alloue un certain CPU au prochain (selon l'ordonnanceur) processus.

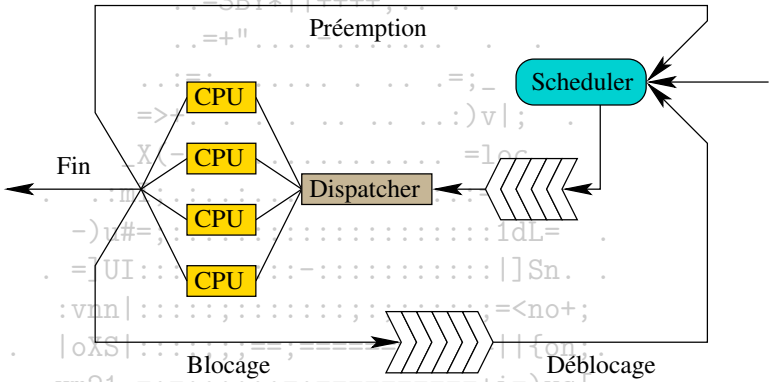


FIG. 3.1 – L'ordonnanceur ordonne, le répartiteur alloue

Il y a deux types de stratégies d'allocation :

– l'ordonnancement préemptif (ou avec réquisition) qui vise à pouvoir suspendre temporairement un processus qui pourrait poursuivre son cours.
– l'ordonnancement non-préemptif (ou sans réquisition) qui consiste à élir un nouveau processus lorsque celui en cours prend fin ou se retrouve dans l'état bloqué.

Le premier type est celui vers lequel nous tendons car il est bien adapté à la multiprogrammation et au contexte multi-utilisateurs. Mais il engendre de nombreuses complications : les accès concurrents et leurs solutions de gestion (Cf. chapitre suivant). A l'opposé, les stratégies non-préemptives sont simples et faciles à implémenter, mais peuvent laisser un processus bloquer tout le système très longtemps. En fait, le choix d'une stratégie dépend du domaine d'application et chaque stratégie peut être adéquate pour un certain domaine.

Nous verrons que les algorithmes d'ordonnancement sont la recherche d'un équilibre entre certains des critères suivants selon le contexte d'application :
- équitabilité : chaque processus reçoit sa part du temps CPU
- efficacité : le processeur est occupé 100% du temps
- temps de réponse : faible pour les systèmes interactif
- temps d'exécution : faible pour minimiser l'attente des travaux en traitement par lots
- rendement : maximiser le nombre de processus exécutés dans le temps
- faible temps de traitement moyen
- temps de réponse maximum : attente bornée entre la soumission et le traitement dans les systèmes temps réel

Pour évaluer les performances des algorithmes, nous pouvons utiliser le modèle de système de tâches. Sa souplesse nous permet d'assimiler un processus à une tâche ou une chaîne de tâches. A chaque tâche T_i sont associés deux réels :
- t_i représentant sa date d'arrivée dans la file d'attente de l'UC.

Le réel τ_i peut avoir plusieurs interprétations :
- τ_i peut représenter la durée exacte qu'il faut à un processeur pour exécuter la tâche. En général, cette valeur est impossible à calculer à l'avance. Cette interprétation est utilisée pour faire des comparaisons de performances d'algorithmes à postériori.
- τ_i, durée (estimée) d'utilisation du CPU avant le prochain blocage (due à des opérations d'I/O). Ainsi, la notation devient inadaptée et devrait être remplacée par τ_{in}. Nous ferrons l'économie de cette notation dans la suite, mais signalons que ci-dessous, là où τ_i est associé à la première interprétation, il pourrait également faire référence à ce deuxième sens.
- τ_i peut être remplacée par une durée estimée e_i de l'exécution de T_i. La méthode utilisée classiquement pour construire les e_i est la relation de récurrence :

$$e_i = \alpha e_{i-1} + (1 - \alpha)\tau_i$$

avec un e_0 choisi arbitrairement et un α figé (en général $\alpha = \frac{1}{2}$).
- τ_i peut aussi représenter une borne supérieure à la durée d'exécution de la tâche T_i. Cet usage s'interprète pour modéliser les situations dans les pires des cas dans le contexte des systèmes en temps réel.

Ces réels permettent de determiner des indicateurs tels que le temps moyen de traîtement, l'attente moyenne, le temps de réponse moyen, etc.

L'algorithme d'ordonnancement construit une assignation des tâches du système qui doit respecter les contraintes fixées par la relation de précédence ! Le plus souvent, un système de tâches sera utilisé pour un ensemble de processus indépendants sans contraintes de précédence, mais l'hypothèse d'indépendance d'un ensemble de processus n'est pas toujours de mise. Par ailleurs, nous pourrons être amenés à utiliser le modèle du système de tâches pour étudier l'exécution des tâches d'un processus où les contraintes de précédence existeront bien.

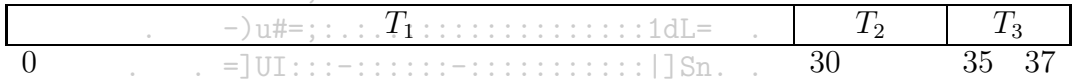
3.2 Algorithmes sans réquisition

Les algorithmes sans réquisitions laissent la ressource CPU au processus qui l'utilise jusqu'à ce qu'il la rende. Nous supposons ici que les tâches sont indépendantes.

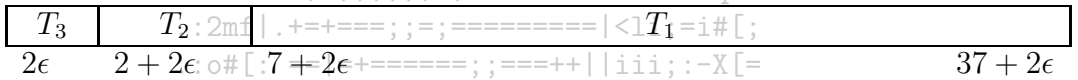
3.2.1 Ordonnancement FIFO

Cet ordonnancement consiste à servir les processus dans leur ordre d'arrivée dans la file d'attente (i.e. fonctionnement FIFO). Cette organisation est facile à mettre en place mais du point de vue du temps de traitement moyen, ses performances sont souvent médiocres.

	τ_i	t_i
T_1	30	0
T_2	5	ϵ
T_3	2	2ϵ



Le temps de traitement moyen est $t_{moy} = \frac{30+(35-\epsilon)+(37-2\epsilon)}{3} \approx 34$, alors que l'assignation

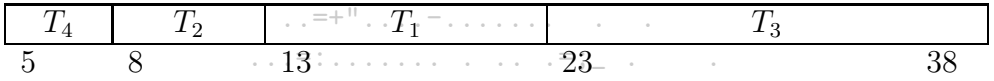


a un temps de traitement moyen de $t_{moy} = \frac{(2+2\epsilon-2\epsilon)+(7+2\epsilon-\epsilon)+(37+2\epsilon)}{3} \approx 15,3$

3.2.2 Ordonnancement PCTE

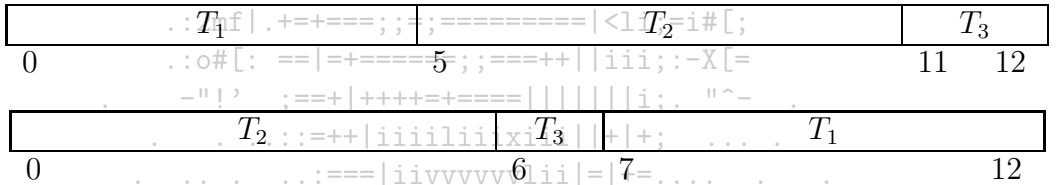
L'ordonnancement par ordre inverse du temps d'exécution (PCTE≡Plus Court Temps d'Exécution) consiste à élir la tâche de temps d'exécution le plus court sans tenir compte de l'ordre d'arrivée dans la file.

	τ_i	t_i
T_1	10	0
T_2	5	2
T_3	15	3
T_4	3	4



Le temps de traitement moyen est $t_{moy} = 18,25$. Observons que cette valeur est minimale. L'ordonnancement PCTE permet de minimiser le temps moyen de traitement par rapport à toute autre stratégie si toutes les tâches sont présentes dans la file d'attente au moment où débute l'assignation.

	τ_i	t_i
T_1	5	0
T_2	6	0
T_3	1	1



La condition d'application de PCTE est primordiale pour sa propriété d'optimalité : le temps moyen de traitement pour PCTE (première assignation) est de 9 alors qu'il est possible d'atteindre le temps de 8 par la seconde assignation. Utiliser PCTE au fil des arrivées des processus ne conduit pas à un ordonnancement optimal du point de vue du temps moyen de traitement.

Nous savons qu'il est impossible de connaitre à l'avance les temps d'exécution des tâches, et donc l'algorithme ne peut être utilisé tel quel. Cependant, il reste interessant car il permet de comparer les performances d'un algorithme réellement implantable aux valeurs minimales qu'il est possible d'obtenir. De plus, en utilisant des estimations pour les temps d'exécution, cet algorithme peut donner de bonnes performances. C'est notamment le cas pour l'ordonnancement de jobs pour lesquels les utilisateurs ont estimé une durée.

3.3 Algorithmes avec réquisition (préemptifs)

3.3.1 Ordonnancement circulaire ou tourniquet (round robin)

Il s'agit d'un algorithme ancien, simple et fiable. L'OS gère une liste circulaire de processus. Chaque processus de la file des processus prêts se voit attribuer un quantum de temps CPU selon son ordre d'arrivée dans la file. Si le processus se bloque avant la fin de son quantum ou s'il prend fin, le processus suivant utilise son quantum de temps. Sinon, le processus est interrompu à la fin du quantum et est placé à la fin de la file des processus prêts.

Le seul problème de cette méthode est de choisir la taille du quantum. La commutation de processus necessite quelques μsecondes et si le quantum est du même ordre de grandeur, le temps passé à commuter devient trop important. D'un autre côté, si le quantum est très grand, le temps de réponse peut devenir inacceptable dans un système interactif.

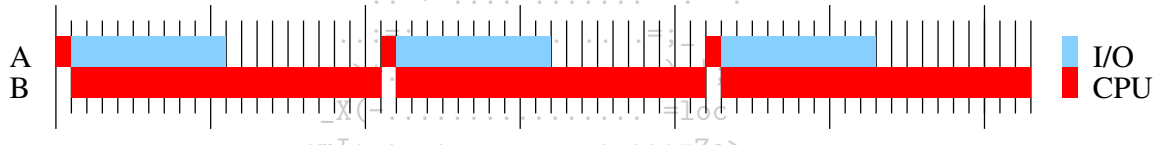


FIG. 3.2 – quantum = 10ms

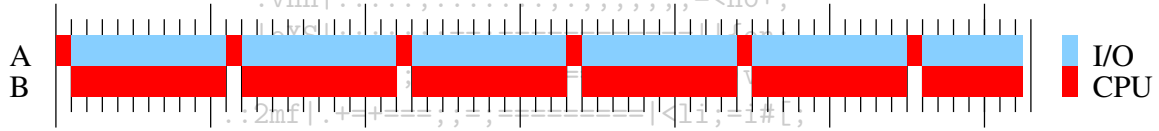


FIG. 3.3 – quantum = 1ms

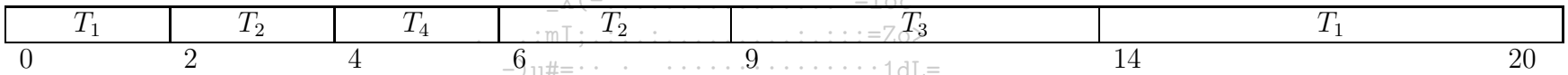
FIG. 3.4 – Comparaison des ordonnancements round robin et FIFO

Pb : que se passe-t-il si dix processus nécessitent 1000 quanta chacun ? L'ordonnancement circulaire est équitable mais peu efficace du point de vue de temps de traitement moyen !

3.3.2 Ordonnancement PCTER

PCTER est l'acronyme signifiant Plus Court Temps d'Exécution Restant. Cet algorithme est l'ordonnancement PCTE auquel est ajouté le mécanisme de préemption du CPU et l'exécution par quantum. Il gère une table contenant, pour chaque tâche T_i, sa durée d'exécution restante τ'_i. Au départ, τ'_i vaut τ_i. L'algorithme choisi une tâche pour laquelle la valeur de τ'_i est minimale et le répartisseur lance son exécution. A la fin du quantum, l'ordonnanceur retire à la valeur τ'_i de la tâche qui utilisait le CPU, la valeur du quantum. Pour le quantum suivant, il choisit à nouveau la tâche ayant la valeur τ'_i minimum. Ci-dessous, un exemple avec un quantum de 2 unités de temps. Le temps moyen d'exécution est de 10 unités de temps.

	τ_i	t_i
T_1	8	0
T_2	5	2
T_3	5	3
T_4	2	4



La plupart du temps, PCTER est aménagé de sorte qu'il devienne "plus court temps d'utilisation du CPU". Ainsi, on favorise les processus qui opèrent de nombreuses entrées sorties et qui utilisent le CPU sur des durées très courtes.

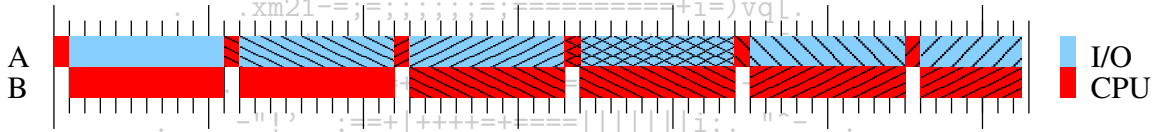


FIG. 3.5 – Exemple où le bloquage est assimilé à une fin d'execution

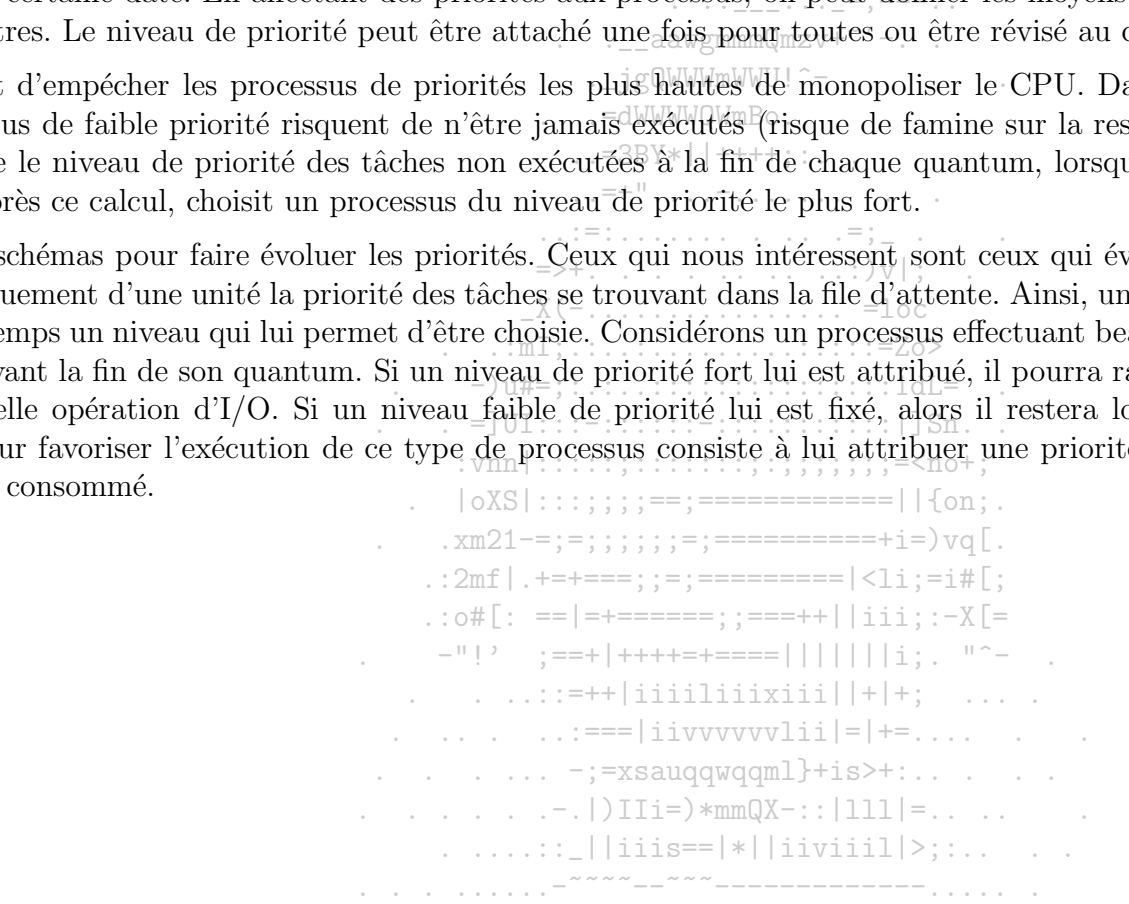
L'ordonnancement PCTER permet de minimiser le temps moyen de traitement par rapport à toute autre stratégie avec réquisition.

3.3.3 Ordonnancement avec priorités

Dans un système temps partagé multi-utilisateurs, tous les utilisateurs sont considérés, à priori, égaux et le CPU est réparti équitablement entre toutes les tâches. Cependant, certaines contraintes d'éfficacité remettent en cause cette idéalisation : l'amélioration de performances globales d'un système nécessite que certaines tâches systèmes ne soient pas devancées par des tâches utilisateurs. Dans certains cas, il est crucial que des tâches fournissent leurs résultats avant une certaine date. En affectant des priorités aux processus, on peut donner les moyens à l'OS de faire exécuter des tâches plus prioritaires avant toutes les autres. Le niveau de priorité peut être attaché une fois pour toutes ou être révisé au cours du temps.

Le problème est d'empécher les processus de priorités les plus hautes de monopoliser le CPU. Dans le cas d'une attribution définitive du niveau de priorité, les processus de faible priorité risquent de n'être jamais exécutés (risque de famine sur la ressource CPU). Dans le cas d'évolution des priorités, le système recalcule le niveau de priorité des tâches non exécutées à la fin de chaque quantum, lorsque l'interruption horloge redonne le contrôle à l'OS. L'ordonnanceur, après ce calcul, choisit un processus du niveau de priorité le plus fort.

Il existe divers schémas pour faire évoluer les priorités. Ceux qui nous intéressent sont ceux qui évitent les risques de famine. Par exemple, l'OS peut augmenter périodiquement d'une unité la priorité des tâches se trouvant dans la file d'attente. Ainsi, une tâche qui a au départ une faible priorité atteint au bout d'un certain temps un niveau qui lui permet d'être choisie. Considérons un processus effectuant beaucoup d'I/O. Les chances sont fortes pour qu'un tel processus bloque avant la fin de son quantum. Si un niveau de priorité fort lui est attribué, il pourra rapidement reprendre son exécution et probablement effectuer une nouvelle opération d'I/O. Si un niveau faible de priorité lui est fixé, alors il restera longtemps en mémoire, l'occupant inutilement. Une méthode simple pour favoriser l'exécution de ce type de processus consiste à lui attribuer une priorité inversément proportionnelle à la dernière fraction de quantum qu'il a consommé.



Comment tricher sous VAX/VMS et OS/2 ?

événement	valeur
expiration d'un quantum	0
terminaison d'uno opération I/O	2
libération d'une ressource demandée par ailleurs	3
terminaison d'une sortie vers un terminal	4
terminaison d'une entrée depuis un terminal	6
création d'un processus	6

Le système VAX/VMS utilise un ordonnancement par priorités qui traite différemment les processus normaux (temps partagé et processus de fond) des processus temps-réels. Les niveaux 16 à 31 sont réservés aux processus temps réels. Ils sont ordonnancés uniquement sur la base de leur priorité qui n'est modifiable que par l'opérateur. Dès qu'un processus de priorité plus forte arrive, le CPU est requisitioné pour ce nouveau processus.

Les priorités des processus normaux, comprises entre 15 et 0, évoluent avec le temps. Chaque processus est initialisé avec un niveau de priorité de base qui peut être augmenté par certains événements. Si, au moment où l'événement se produit, un processus normal est choisi sur la base de sa priorité courante celle-ci est augmentée de la valeur associée à l'événement. Par la suite, chaque fois que le processus est élu, sa priorité décroit, mais ne retombe jamais sous son niveau initial. Les processus qui ont même niveau courant sont ordonnancés par un tourniquet.

Le système OS/2 utilise aussi un ordonnacement de même type sur des priorités. Il distingue trois types de processus : ceux à délai critique (les plus prioritaires), les normaux et les tâches de fond, Les processus à délai critique et les tâches de fond ont des priorités immuables. Par contre, les priorités des autres évoluent selon la durée des phases de calcul. Un processus qui a des phases de calcul courtes voit sa priorité légérement augmenter. C'est le contraire pour les processus dont les phases de calcul sont très longues. Toutefois, les niveaux de priorités des processus normaux ne peuvent atteindre ceux des processus à délai critique ou des tâches de fond.

Les programmeurs astucieux ajoutent à leur processus l'affichage périodique de caractères pour accroitre la priorité de leur processus et donc réduire le temps d'exécution `(ô_ô)`.

3.3.4 Ordonnancement avec files multiples

Première organisation : les processus sont organisés dans différentes files d'attente. Ces files matérialisent des classes de processus. À chaque classe de processus est associé un niveau de priorité. Une répartition du temps CPU proportionnelle au niveau de priorité est effectuée.

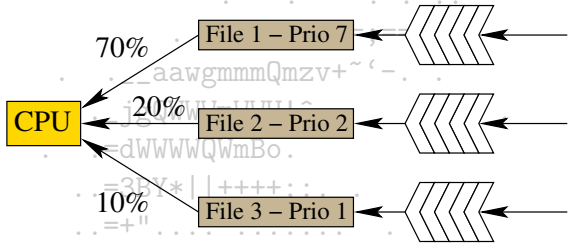


FIG. 3.6 – Répartition du CPU

Chaque file est ordonnancée par un algorithme adapté au type de tâche qui constitue la file.

Deuxième organisation : les files sont organisées de façon hiérarchique. C'est la priorité de la tâche qui détermine la file dans laquelle elle entre. Une tâche d'une file ne peut être exécutée que si toutes les files de niveau supérieur sont vides. Les priorités des tâches évoluent dynamiquement et c'est le système qui doit tenir à jour les files en faisant remonter vers les files supérieures les tâches dont la priorité a été augmentée.

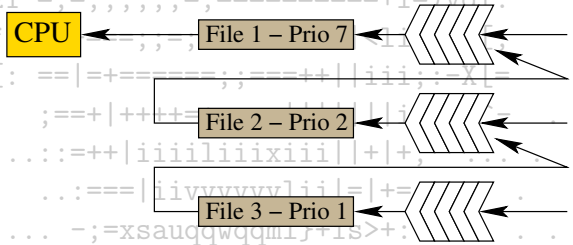


FIG. 3.7 – Organisation en files hiérarchiques et remontée

Tout le problème est de définir les règles d'évolution de priorités. Indiquons les techniques de vieillissement visant à faire augmenter la priorité d'un processus de façon régulière jusqu'à ce qu'il soit servi. Les situations de famine sont ainsi évitées.

Autre organisation : les organisations proposées ci-dessus ne mettent pas en évidence tous les degrés de liberté qui peuvent être utilisés pour organiser des files multiples. Ci-dessous, nous illustrons trois files hiérarchiques, une file ne pouvant soumettre ses travaux que lorsque les files supérieures sont vides. La première files reçoit tout nouveau processus et les traite avec une gestion FIFO pour un quantum de 8mus. Si le processus n'est pas achevé à la fin de ce quantum, il est alors placé en queue de la seconde file qui est gérée en FIFO sur un quantum de 16mus. Si le processus n'est pas achevé à la fin de ce quantum, il est alors placé en queue de la dernière file qui est gérée par PCTER sur un quantum de longueur donné.

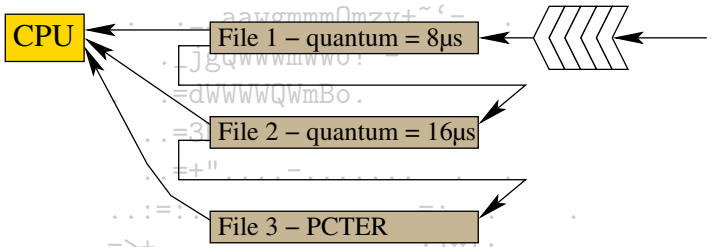


FIG. 3.8 – Autre organisation en files

Si la souplesse qu'il est possible d'introduire dans l'ordonnancement des processus est importante dans ce type d'ordonnancement, il est hélas également complexe à implanter !

Dans le système Unix, chaque processus dispose d'un numéro de priorité, qui fixe le niveau de la file d'attente où il est placé. Le système différencie les processus systèmes et les processus utilisateurs. Les premiers sont toujours prioritaires sur les suivants. A l'intérieur de chaque classe, il y a plusieurs niveaux de priorité, chacun étant associé à une file d'attente. Les numéros de priorité des processus évoluent dynamiquement

Tout d'abord, chaque fois qu'un processus est mis en attente, sa priorité est recalculée en fonction de la raison qui a causé son attente. L'idée sous-jacente à ce calcul est de tenter de diminuer les risques de blocage en favorisant les processus qui, au réveil, pourront se terminer le plus rapidement et ainsi libérer leurs ressources. Ainsi un processus mis en attente de la terminaison d'une opération d'I/O avec le disque reçoit une priorité plus importante qu'un processus mis en attente à la suite d'une demande d'I/O avec le disque. En effet, le second a besoin d'obtenir un tampon, qui pourrait être libéré par le premier, s'il se termine avant. Ensuite, à chaque interruption d'horloge, la priorité d'un processus prêt est augmentée en fonction du temps passé dans la file d'attente. Cette modification de la priorité peut avoir pour conséquence un transfert du processus vers une file d'attente de niveau supérieur.

3.3.5 Ordonnancement par une politique (d'équitabilité)

L'ordonnancement par politique a pour objectif un partage équitable du processeur entre les processus. Chaque processus parmi n devrait profiter du $\frac{1}{n}$ de la puissance de calcul.

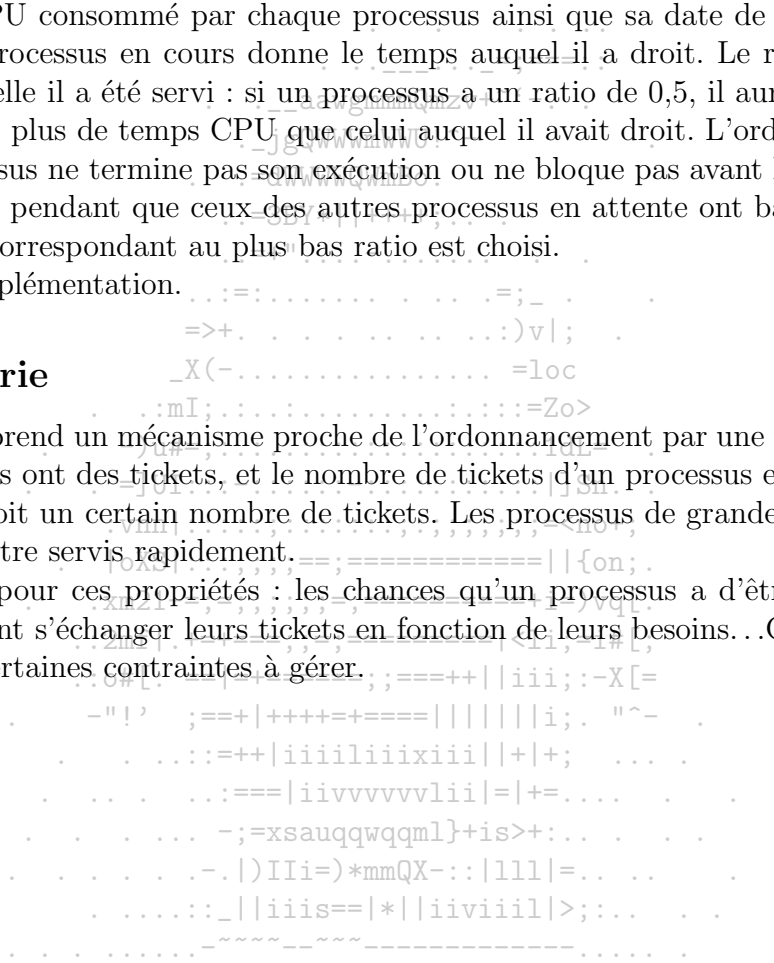
Le système garde une trace du temps CPU consommé par chaque processus ainsi que sa date de création. Pour un processus donné, le temps écoulé depuis sa création rapporté au nombre de processus en cours donne le temps auquel il a droit. Le ratio du temps CPU qu'il a consommé et du temps auquel il a droit indique la qualité avec laquelle il a été servi : si un processus a un ratio de 0,5, il aurait dû avoir deux fois plus temps CPU qu'il a reçu. Si son ratio est de 2, il a consommé deux fois plus de temps CPU que celui auquel il avait droit. L'ordonnanceur choisi le processus qui possède le ratio le plus bas pour lui allouer le CPU. Si le processus ne termine pas son exécution ou ne bloque pas avant la fin de son quantum, alors à la fin de son quantum de temps, le ratio de ce processus s'est accru pendant que ceux des autres processus en attente ont baissé. Si le ratio est toujours le plus bas, le CPU lui est à nouveau attribué, sinon, un processus correspondant au plus bas ratio est choisi.

La difficulté de cette approche est son implémentation.

3.3.6 Ordonnancement par loterie

L'idée de l'ordonnancement par loterie reprend un mécanisme proche de l'ordonnancement par une politique d'équitabilité. Le lot pour le ticket gagnant est un quantum de temps CPU. Les processus ont des tickets, et le nombre de tickets d'un processus est directement lié aux chances qu'il a de gagner à la loterie. A son initialisation, un processus reçoit un certain nombre de tickets. Les processus de grande importance peuvent recevoir un nombre important de tickets, leur donnant de fortes chances d'être servis rapidement.

Ce mécanisme de loterie est intéressant pour ces propriétés : les chances qu'un processus a d'être servi sont proportionnelles au nombre de tickets qu'il possède. Des processus coopérant peuvent s'échanger leurs tickets en fonction de leurs besoins...Cette méthode d'ordonnancement est mise en œuvre lorsque les autres méthodes achoppent sur certaines contraintes à gérer.



3.3.7 Ordonnancement à deux niveaux

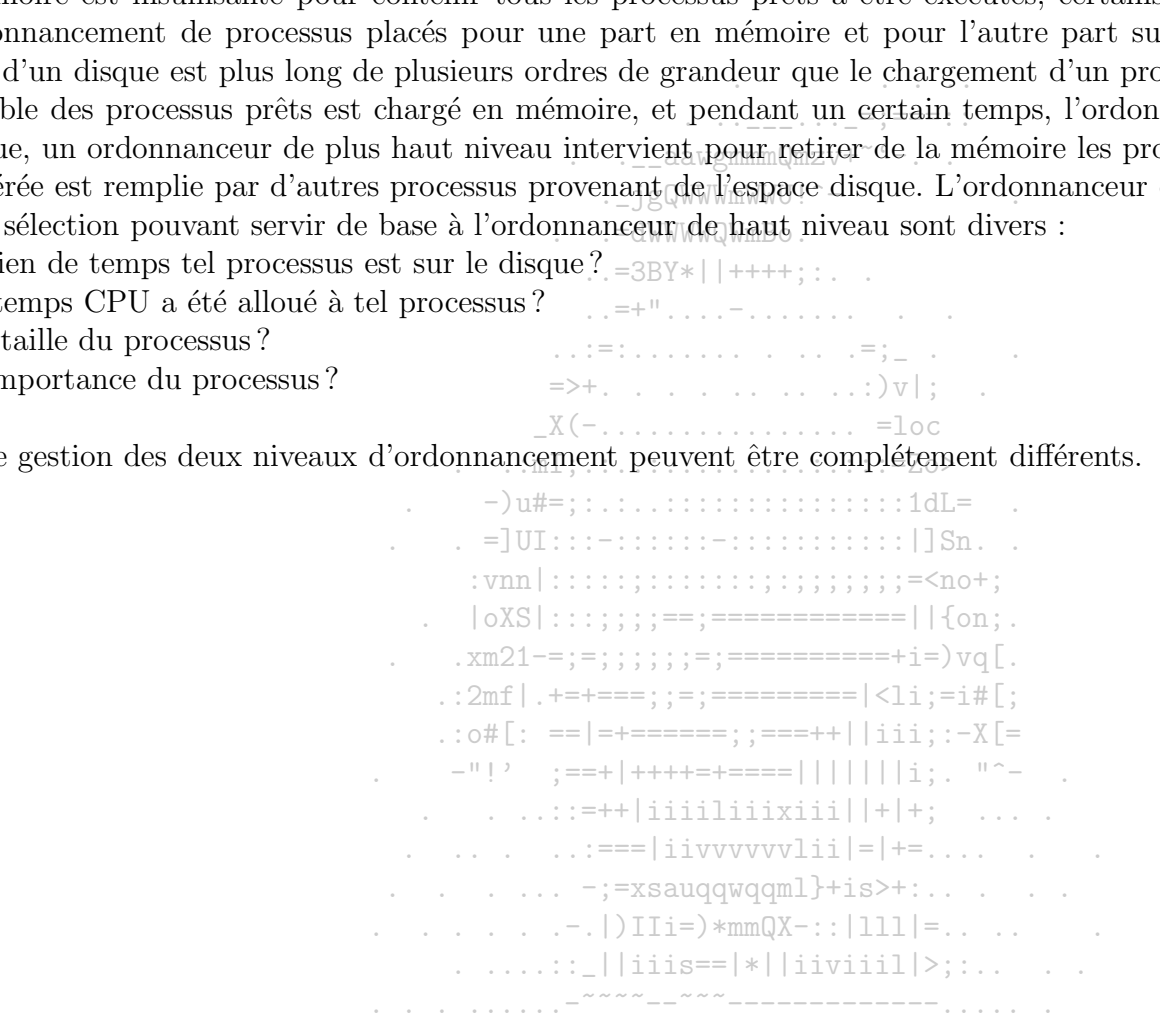
Lorsque la mémoire est insuffisante pour contenir tous les processus prêts à être exécutés, certains d'entre eux doivent être placés sur le disque. La difficulté de l'ordonnancement de processus placés pour une part en mémoire et pour l'autre part sur disque provient du fait que le chargement d'un processus à partir d'un disque est plus long de plusieurs ordres de grandeur que le chargement d'un processus présent en mémoire.

Un sous-ensemble des processus prêts est chargé en mémoire, et pendant un certain temps, l'ordonnanceur se restreint à ne servir que ces processus. De façon périodique, un ordonnanceur de plus haut niveau intervient pour retirer de la mémoire les processus qui y ont été suffisamment longtemps et la place mémoire libérée est remplie par d'autres processus provenant de l'espace disque. L'ordonnanceur de bas niveau gère alors les nouveaux processus.

Les critères de sélection pouvant servir de base à l'ordonnanceur de haut niveau sont divers :

- depuis combien de temps tel processus est sur le disque ?
- combien de temps CPU a été alloué à tel processus ?
- quelle est la taille du processus ?
- quelle est l'importance du processus ?
- ...

Les mécanismes de gestion des deux niveaux d'ordonnancement peuvent être complétement différents.



3.3.8 Ordonnancement pour le temps réel

Dans le contexte d'une application temps réel, le rôle d'un OS est souvent de réagir à différents événements. Les réponses de l'OS à ces événements prennent la forme de processus. Évidemment, on considère qu'une réaction à un événement a été faite lorsque le processus correspondant a pris fin. La difficulté du temps réel est que ces réactions doivent avoir lieu dans un court laps de temps : après la survenue d'un événement, il y a une échéance pour la réaction que l'OS doit produire. Ces échéances (ou dead line) se classent en deux catégories :

- les échéances absolues : la contrainte d'exécution avant une date doit être remplie. La situation où un processus ne finit pas à temps est aussi grave que celle où un processus ne finit pas.
- les échéances "au mieux" : le dépassement du delai pour l'exécution est acceptable.

Dans le contexte du temps réel, les programmes sont souvent divisés en processus différents dont les performances et les comportements sont connus à l'avance.

Les événements peuvent être périodiques ou non (d'occurence imprévisible). S'il y a m événements périodiques, avec le i-ième événement intervenant toutes les p_i secondes et nécessitant c_i secondes de temps CPU, l'OS sera en mesure de faire face à ces contraintes si

$$\sum_{i=1}^{m} \frac{c_i}{p_i} \leq 1$$

Un système en temps réel vérifiant cette condition est dit ordonnançable (on suppose que la durée de commutation est négligeable).

L'ordonnancement peut être dynamique (au fil des événements) ou statique (fait à l'avance). Ci-dessous, trois algorithmes dynamiques :

- algorithme à taux monotone : chaque processus a une priorité fixée proportionnellement à sa fréquence. L'ordonnancement consiste à élir le processus de plus haute priorité.
- algorithme de la plus proche échéance : lorsqu'un événement survient, le processus concerné est ajouté à la liste des processus prêts. La liste est triée par échéance. L'ordonnancement consiste à élir le processus dont l'échéance est la plus proche.
- algorithme de la plus petite marge : pour chaque processus, une marge égale au temps dont il dispose pour terminer son exécution est calculée. Le processus choisi correspond à la marge la plus petite.

3.3.9 Ordonnancement de chaînes de tâches

Les méthodes d'ordonnancement présentées jusqu'ici sont basées sur l'hypothèse d'indépendance des processus (ou des tâches) à gérer. Cette hypothèse n'est, en pratique, que rarement vérifiée. Quand est-il du cas le plus courant, celui où les processus communiquent et où les tâches d'un processus sont fortement liées ? Une optimisation de tel ou tel critère est, dans ce cas, un vrai pari.

Un des rares cas où l'on dispose de résultats est celui de la gestion parallèle de chaînes de tâches séquentielles dont toutes les tâches sont présentes dans la file d'attente à l'instant initial. Notons C_1, C_2,...C_r les chaînes de tâches et posons :

$$C_i = T_{i_1} T_{i_2} \dots T_{i_{k_i}} \text{ et } \sum_{i=1}^{r} k_i = n$$

τ_{i_j} est une durée indicative sur le temps d'exécution de T_{i_j}. De plus, nous appelons sous-chaîne initiale de C_i toute chaîne $C'_{i.p} = T_{i_1} T_{i_2} \dots T_{i_p}$ avec $1 \leq p \leq k_i$. La durée d'exécution moyenne de $C'_{i.p}$ est

$$\epsilon_{i,p} = \frac{1}{p} \sum_{h=1}^{p} \tau_{i_h}$$

La sous chaîne minimale $C'_{i.p}$ est celle minimisant son temps d'exécution moyen parmi les sous-chaînes initiales de C_i.

L'algorithme suivant produit une assignation minimisant le temps d'exécution moyen :

```
début
        pour chaque Ci, calculer la durée d'exécution moyenne ε_{j,p} de la sous-chaîne minimale C'_{i.p} ;
        tant qu'il existe des tâches non-assignées,
        faire
                déterminer un indice j tel que ε_{j,p} = min{ε_l | 1 ≤ l ≤ n}
                ajouter à l'assignation les tâches de la sous-chaîne C'_{j.p_j} et les supprimer de C_j ;
                recalculer ε_{j,p} pour la nouvelle chaîne C_j.
        fin faire
fin
```

Rem : si chaque chaîne se réduit à une seule tâche, cet algorithme n'est autre que PCTE !

3.3.10 Application

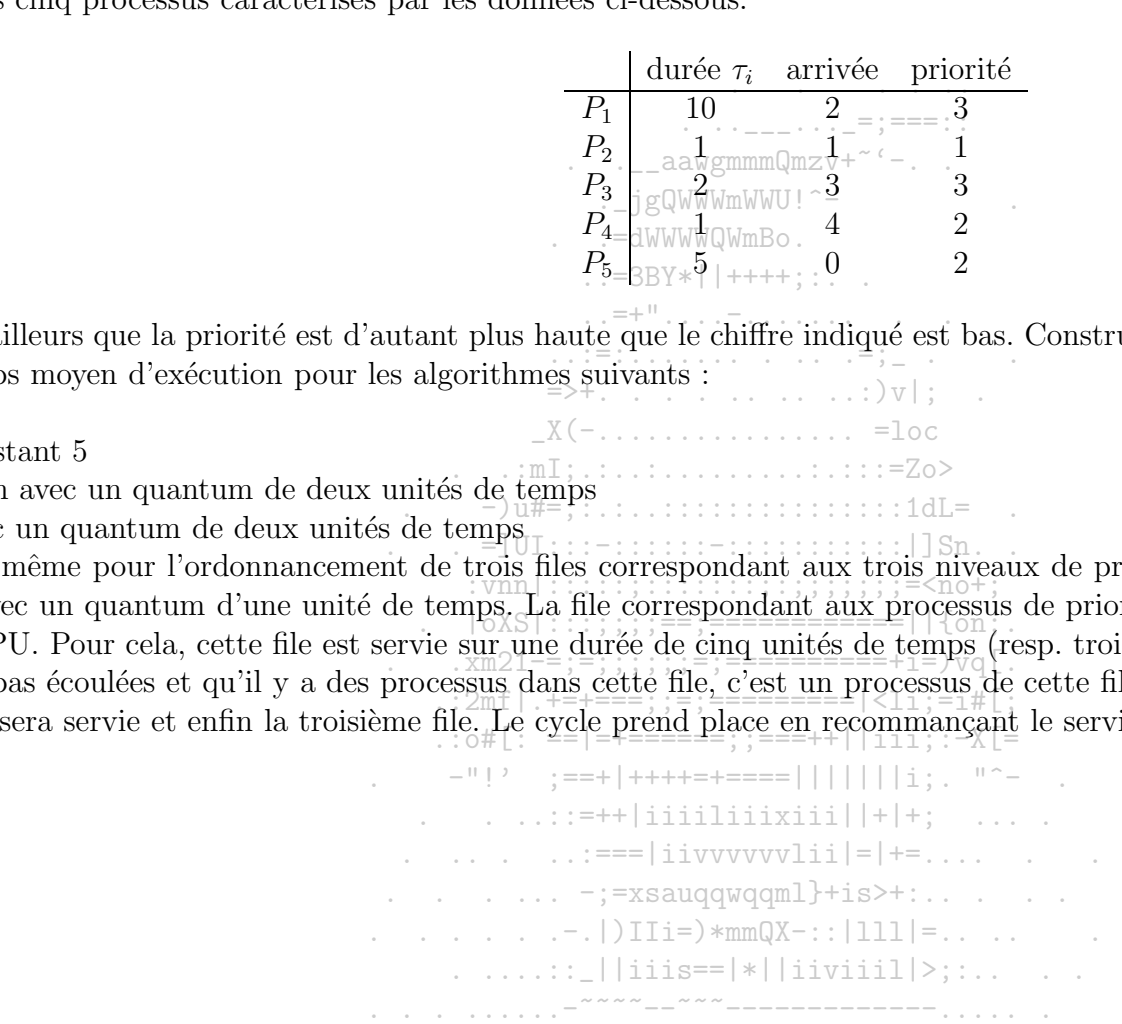
Considérons les cinq processus caractérisés par les données ci-dessous.

	durée τ_i	arrivée	priorité
P_1	10	2	3
P_2	1	1	1
P_3	2	3	3
P_4	1	4	2
P_5	5	0	2

Précisons par ailleurs que la priorité est d'autant plus haute que le chiffre indiqué est bas. Construire le diagramme (de Gantt) d'allocation du CPU et déterminer le temps moyen d'exécution pour les algorithmes suivants :

- FIFO
- PCTE à l'instant 5
- Round Robin avec un quantum de deux unités de temps
- PCTER avec un quantum de deux unités de temps

Enfin, faire de même pour l'ordonnancement de trois files correspondant aux trois niveaux de priorité et décrit ci-après : chaque file est ordonnancée par un PCTER avec un quantum d'une unité de temps. La file correspondant aux processus de priorité 1 (resp. 2 et 3) se voit allouer 50% (resp. 30% et 20%) du temps CPU. Pour cela, cette file est servie sur une durée de cinq unités de temps (resp. trois et deux). Plus précisément, tant que les cinq unités de temps ne sont pas écoulées et qu'il y a des processus dans cette file, c'est un processus de cette file qui sera servi. Puis la file correspondant au second niveau de priorité sera servie et enfin la troisième file. Le cycle prend place en recommançant le service de la première file s'il y a lieu, etc.



Chapitre 4

Processus et ressources

Des processus qui s'exécutent en parallèle dans un système partagent les ressources que l'OS met à leur disposition. Ces processus peuvent s'ignorer mutuellement et concourir pour l'acquisition de ressources (dans le cas le plus probable où elles sont en nombre insuffisant pour satisfaire les besoins de tous) ou ils peuvent coopérer pour réaliser un partage profitable globalement en communiquant.

Dans un cas comme dans l'autre, les actions d'un groupe de processus peuvent conduire ses éléments à une situation de blocage : chaque processus a en sa possession une ressource et a besoin pour poursuivre d'une ressource détenue par un autre processus. La figure ci-contre illustre un tel cas où les ressources sont en exclusion mutuelle (une ressource ne peut être occupée que par un processus au maximum) et ne sont pas réquisitionnables (une ressource occupée ne peut être libérée que par le processus qui la détient). Les processus sont en attente circulaire.

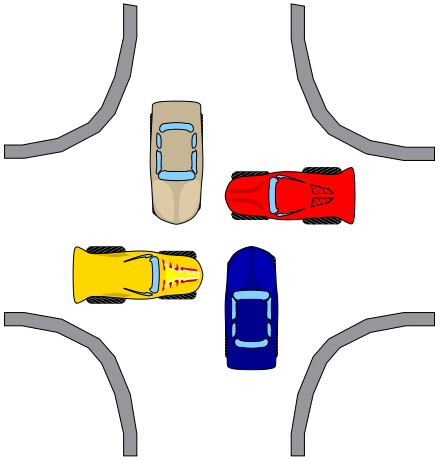


FIG. 4.1 – Situation de blocage

Comment faire face à de telles situations ? L'OS peut mettre en œuvre divers moyens pour parer à cette éventualité. Tout d'abord laisser les situations de blocage se déclarer, les détecter et intervenir en reprenant les ressources de quelques processus...L'OS peut également prévenir les situations de blocage (Cf caractérisation des blocages) et dans ce cas nous parlerons de prévention de blocage, ou encore contrôler qu'à chaque étape il reste un moyen d'éviter le blocage en utilisant des techniques d'évitement. Ces dernières sont plus souples mais aussi plus coûteuses. Il est important d'avoir présent à l'esprit que la plupart des systèmes font l'impasse sur cette problématique (y compris Unix) pour des raisons de coût, mais également en raison de la faible fréquence de blocages. Le redémarrage est alors la solution la plus simple...

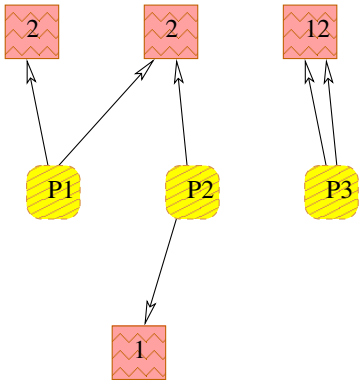
Après une formalisation des entités en jeu, nous donnerons une caractérisation des blocages. Suivront les présentations des différentes possibilités de gestion des «deadlocks».

4.1 Formalisation des ressources

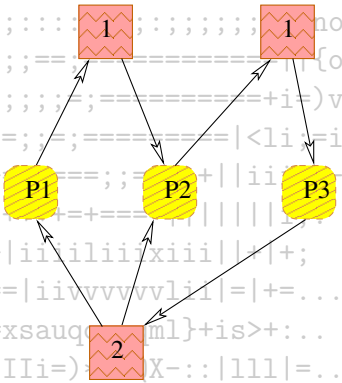
Il est possible d'utiliser le formalisme des graphes pour représenter les ressources d'un système et les besoins d'un ensemble de processus. Etant donnés m types de ressources notés R_1, ..., R_m, R_j étant disponible en v_j exemplaires, et n processus, nous considérons le graphe défini comme suit :

- A tout processus correspond un noeud «processus». De même, à toute ressource correspond un noeud «ressource», ce nœud étant valué par le nombre d'exemplaires associés à la ressource. L'ensemble des sommets du graphe considéré est constitué par l'union de ces deux ensembles de nœuds.
- Les arcs considérés sont de deux types :
  - ceux symbolisant la détention d'un exemplaire d'une ressource R_j par un processus P_i ; un tel arc est orienté (du nœud) de la ressource R_j vers le (nœud du) processus P_i.
  - ceux symbolisant la requête d'un exemplaire d'une ressource R_j par un processus P_i ; un tel arc est orienté du processus P_i vers la ressource R_j.

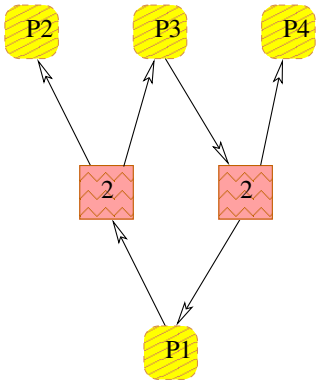
La logique sous-jacente à ce graphe implique que le nombre d'arcs sortant d'un nœud ressource n'excède pas sa valuation. Par ailleurs, plusieurs arcs peuvent exister entre deux nœuds. Un tel graphe, dit graphe d'allocation des ressources systèmes, permet de représenter l'état d'allocation des ressources à un instant donné.



(a) Exemple de graphe d'allocation



(b) Exemple de graphe présentant un cycle avec blocage



(c) Exemple de graphe présentant un cycle sans blocage

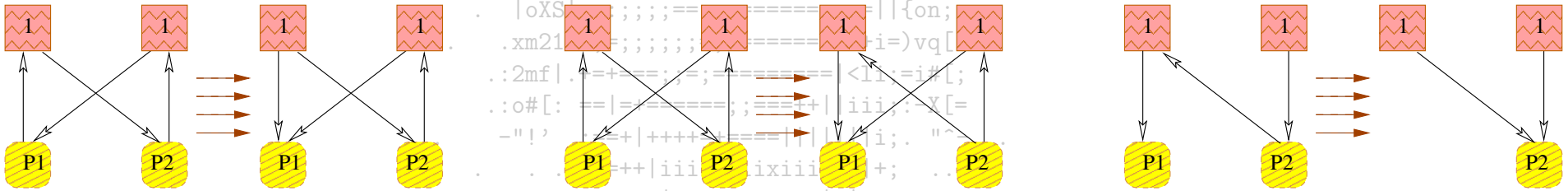
FIG. 4.2 – Exemple de graphe d'allocation

4.2 Définition et caractérisation des inter blocages

Un interblocage est un état du système dans lequel plusieurs processus sont bloqués, chaque processus étant privé d'une partie des ressources qu'il nécessite pour pouvoir poursuivre son cours par un autre processus bloqué.

Différentes conditions nécessaires caractérisent un inter blocage :

1. L'occurrence d'un inter blocage nécessite l'accès à des ressources en exclusion mutuelle. En effet, si une ressource n'est pas accédée en exclusion mutuelle, alors tout processus y a accès à tout moment sans restriction. Une telle ressource ne peut être source de blocage. . .Par ailleurs, l'exclusion mutuelle garantit une cohérence dans l'usage qui est fait des ressources dont le but à priori n'est pas d'être partagées.
2. La réalisation d'un inter blocage met en jeu des ressources exclusivement non-préemptibles. Seul le processus détenant les ressources peut les liberer. Dans le cas où les ressources seraient préemptibles, il serait possible de requisitionner les ressources de certains processus pour temporairement satisfaire les besoins d'un processus particulier.
3. L'occurrence d'un inter blocage implique une attente circulaire : un premier processus P_1 attend une ressource détenue par un processus P_2, P_2 attend une ressource détenue par P_3, . . .et P_m attend une ressource détenue par P_1.
4. L'occurrence d'un inter blocage implique que tout processus impliqué dans le blocage détient une ressource et nécessite une ressource détenue par un autre. Cette condition est un affaiblissement de la précédente. S'il y a attente circulaire, cette condition est nécessairement réalisée.



(a) non respect de l'exclusion mutuelle (b) non respect de la non-préemptibilité (c) non respect de l'attente circulaire

FIG. 4.3 – Mises en défaut des conditions ci-dessus

Ainsi, chacune des quatre conditions ci-dessus est nécessaire mais non suffisante pour la réalisation d'un inter blocage. Les graphiques ci-dessus tendent à illustrer cette insuffisance. E.g. la figure 4.2(c) comporte un cycle, mais l'éventuelle libération d'une instance de la seconde ressource par le processus P_4 pourrait mettre fin à l'attente de P_3 et casser le cycle. De même, la figure 4.3(b) comporte un cycle, mais la retrait à P_2 de la ressource manquante à P_1 permet à P_1 de poursuivre.

Les propriétés suivantes relient la situation d'inter blocage et un graphe d'allocation de ressources :
- S'il n'existe pas de cycle dans le graphe d'allocation, alors il n'y a pas d'inter blocage.
- S'il existe un cycle, un inter blocage est possible. Dans le cas où chaque ressource est en un unique exemplaire, alors la présence de cycle implique un blocage. Si toutes les ressources prises dans un cycle ont une unique instance, alors tous les processus pris dans le cycle sont bloqués.

4.3 Gestion des blocages

4.3.1 Méthodes de prévention

Pour prévenir d'une situation de blocage, il suffit d'empêcher la réalisation de l'une des quatre conditions nécessaires vues plus haut. Considérons consécutivement ces conditions.

Exclusion mutuelle (cond. 1)

Casser l'accès en utilisation exclusive revient à changer la nature même de l'emploi d'une ressource. Nous avons vu plus haut la nécessité du mécanisme d'exclusion mutuelle pour l'accès aux ressources. Cette condition n'est donc pas à infirmer de façon directe...Toutefois, dans certains cas, il est possible de mettre en place un mécanisme d'accès aux ressources permettant d'éviter les blocages. L'utilisation d'un mécanisme de spooling peut permettre d'éviter certains blocages. Le principe consiste à confier l'accès à la ressource à un processus «démon», ce dernier permettant aux demandeurs de placer leur requête dans une file d'attente. Ce mécanisme est particulièrement adapté à certains périphériques d'I/O.

Préemptibilité des ressources (cond. 2)

Le caractère de non-préemptibilité de certaines ressources entraine la rétention de ressource par les processus. Une façon d'empêcher la rétention des ressources par un processus qui est bloqué en attente d'autres ressources consiste à obliger le processus à rendre l'ensemble de ses ressources si sa demande ne peut être satisfaite immédiatement. Le processus pourra reprendre son cours lorsque l'ensemble des ressources qu'il nécessite (celles qu'il a éventuellement libérées et celles qu'il à dernièrement réclamé) pourra lui être alloué.

Un affaiblissement de cette démarche est la démarche suivante : si un processus se retrouve bloqué en attente d'autres ressources, ces ressources sont recherchées parmi les ressources des autres processus en attente. Si toutes les ressources de la demande peuvent être rassemblées, elles sont retirées aux processus qui les utilisent et ainsi le processus peut poursuivre. Sinon, le processus passe en attente. Ce protocole de gestion est applicable pour la gestion de la mémoire par exemple.

Attente circulaire (cond. 3)

Une façon simple d'empecher l'attente circulaire consiste à numéroter les ressources et à imposer aux processus d'opérer leurs demandes de ressource par ordre croissant. Ainsi, une requête sur la ressource numérotée i n'est permise que si les ressources détenues par le processus ont toutes un numéro inférieur strictement à i. Si une requête sur la $i^{\text{ème}}$ ressource doit toutefois être formulée par le processus alors qu'il détient des ressources de numéro supérieur ou égal à i, il doit libérer ces dernières et reformuler des requêtes par ordre croissant de numérotation de ressource.

Détention de ressources (cond. 4)

Casser cette condition revient à n'autoriser des requêtes que par les processus n'ayant aucune ressource. Ainsi, un processus doit formuler ses requêtes avant son exécution ou relacher toutes ses ressources avant de reformuler un ensemble de requêtes visant à satisfaire ses besoins antérieurs et ses nouveaux besoins.

Cette solution présente deux inconvénients majeurs :

- il y a une sous-utilisation des ressources ; c'est d'ailleurs un inconvénient qui se retrouve pour la prévention des conditions 3 et 4.
- il y a un risque de famine.

4.3.2 Méthodes d'évitement

Les méthodes de prévention présentent l'inconvénient d'engendrer une sous-utilisation des ressources et une diminution du débit des traitements. Une idée alternative à celles déjà proposées consiste à utiliser des informations sur les besoins des processus pour gerer les requêtes portant sur les ressources. Le modèle le plus simple, et aussi le plus utile, est basé les besoins au pire en ressources de chaque processus. . .A partir de ces informations et des informations caractérisant l'état d'allocation des ressources, la recherche de risque d'interblocage est effectuée.

Notion d'état sur

Un état est dit sur si le système peut trouver un ordre pour les allocations à chaque processus allant jusqu'au maximum de ses besoins en évitant l'interblocage.

Une séquence de processus $< P_1, \cdots, P_n >$ est dite séquence sure pour l'état d'allocation actuel si pour chaque P_i, les besoins qu'il peut formuler sont satisfaisables, compte tenu des ressources libres et des ressources détenues par les P_j pour tout $j \leq i$. Ainsi, si les ressources attendues par P_i ne sont pas immédiatement disponibles, il peut attendre que les P_j aient tous fini avec la certitude qu'alors il pourra obtenir ces ressources. Si aucune séquence sure n'existe, le système est dit en état «incertain».

Un état sur ne peut comporter un interblocage.

Un interblocage correspond à un état incertain.

Un état incertain ne comporte pas nécessairement d'inter blocage. Un état incertain peut conduire à un blocage. Tant que le système est dans un état sur, il peut éviter les états incertains (et les blocages).

Exemple : Considérons un système comportant douze instances d'une ressource. Le tableau suivant décrit les besoins de trois processus relativement à cette ressource.

	besoin max	besoin courant
P_0	10	5
P_1	4	2
P_2	9	2

$< P_1, P_0, P_2 >$ est une séquence sure. En effet, il reste trois instances libres. P_1 peut en utiliser deux puis finir. Il y aurait alors cinq instances libres. P_0 peut les utiliser puis finir. Il y aurait alors sept instances libres. P_2 peut les utiliser puis finir.

Un système peut passer d'un état sur à un état incertain. Supposons que le système laisse P_2 réclamer une instance supplémentaire. Le tableau récapitulatif est le suivant :

	besoin max	besoin courant
P_0	10	5
P_1	4	2
P_2	9	3

Il reste deux instances libres. Seul P_1 peut les utiliser puis finir. Il y aurait alors quatre instances libres. C'est insufisant pour P_0 et P_2. Le système est à présent dans un état incertain !

La notion d'état sur va nous servir à la construction d'algorithme d'évitement de blocage : une requête sera examinée pour savoir si elle conduit vers un état sur. Si c'est le cas, elle pourra être acceptée.

Algorithme pour le graphe d'allocation de ressources

Au graphe défini précédemment, nous ajoutons des arcs symbolisant les éventuels besoins futurs, représentés en pointillés. Un tel arc peut se transformer en arc plein de demande d'allocation lorsque le besoin devient réel et que le processus exprime cette requête. De la même façon, la libération d'une instance a pour effet la transfomation d'un arc plein de détention en un arc en pointillés du processus vers la ressource.

Dans le cas où chaque ressource est disponible en un exemplaire, une allocation qui engendre un cycle par transformation d'arc $P_i \rightarrow R_j$ en arc $R_j \rightarrow P_i$ engendre un état incertain. Tout le problème est de savoir si à un moment donné un besoin peut être satisfait (en transformant l'arc $P_i \rightarrow R_j$ en arc $R_j \rightarrow P_i$) sans entrainer la formation d'un cycle.

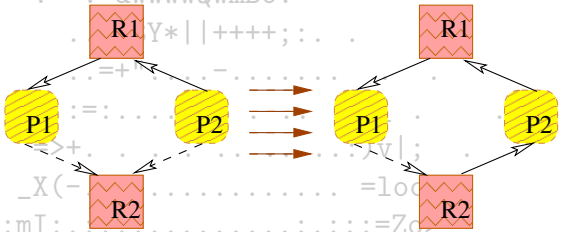


FIG. 4.4 – Passage d'un état sur à un état incertain

Sur la figure 4.4, nous observons dans la partie gauche du graphique un système dans un état certain. Le processus P_2 exprime alors le besoin d'une instance de R_2. Nous observons que l'allocation de l'instance réclamée à P_2 fait basculer le système d'un état sur à un état incertain, puisqu'il y a apparition d'un cycle. Si P_1 réclame une instance de R_2 il y aura inter blocage.

Algorithme du banquier

Cet algorithme permet de traiter le cas où les ressources sont en plusieurs exemplaires. Pour chaque processus P_i, les besoins maximum sont connus pour chacune des ressources. Ces besoins ne peuvent évidemment pas excéder les ressources totales du système.

L'état du système à un instant donné est décrit par l'ensemble des tableaux suivants :

- $A[j]$ est le nombre d'instances de R_j disponibles ;
- $Max[i, j]$ est le nombre maximum d'instance de R_j que le processus P_i peut réclamer ;
- $D[i, j]$ est le nombre d'instances de R_j détenues par P_i ;
- $N[i, j]$ est le nombre d'instances de R_j que le processus P_i peut réclamer en plus de celles qu'il detient déjà.

L'algorithme suivant permet de qualifier l'état (sur ou incertain) du système :

1. `W:=A`
   `F[i]:=false pour tout i`
2. `trouver i tel que F[i]=false`
   `et N[i]<=W`
   `si aucun i trouvé, aller en 4`
3. `W:=W+D[i,.]`
   `F[i]:=true`
   `aller en 2`
4. `si F[i]=true pour tout i`
   `alors l'etat est sur`
   `sinon l'etat est incertain`

L'algorithme suivant permet de déterminer le statut d'une requête :

La requête examinée est représentée par le vecteur $R_i[j]$ dont l'élément de base est le nombre d'instances de R_j demandées.

1. `si Ri[j]<=N[i,j] pour tout j`
   `alors aller en 2`
   `sinon rejeter la requête`
2. `si Ri[j]<=A[j] pour tout j`
   `alors aller en 3`
   `sinon Pi doit attendre`
3. `considerer l'etat defini par`
   `A:=A-Ri`
   `Di:=Di+Ri`
   `Ni:=Ni-Ri`
   `et tester s'il est sur`
   `si oui, le requete est acceptable`
   `sinon Pi attend`

Exemple : Considérons les données suivantes :

	R_1	R_2	R_3	R_4
P_1	3	0	1	1
P_2	0	1	0	0
P_3	1	1	1	0
P_4	1	1	0	1
P_5	0	0	0	0
alloués	5	3	2	2
libres	1	0	2	0

	R_1	R_2	R_3	R_4
P_1	1	1	0	0
P_2	0	1	1	2
P_3	3	1	0	0
P_4	0	0	1	0
P_5	2	1	1	0

Le premier tableau indique les ressources allouées. Le tableau suivant spécifie les besoins à satisfaire. A partir de ces données, déterminons si le système est dans un état sur.

Le vecteur A est donné par la dernière ligne du premier tableau. La matrice D est donnée par les cinq premières ligne du même tableau et la matrice N est donnée par le second tableau. A partir de cette identification, nous pouvons appliquer le premier algorithme ci-dessus pour déterminer l'état du système.

1. `W=[1 0 2 0]` - Les éléments de la quatrième ligne de `N` sont inférieurs à ceux de `W`, pris colonnes à colonnes. On peut donc choisir 4 comme valeur de `i`. Une fois P_4 arrivé à son terme, les éléments disponibles seront `W+D[4,.]`. On a donc à présent `W=[2 1 2 1]`.
2. Les éléments de la première ligne sont inférieurs à ceux de `W`, pris colonnes à colonnes. On choisit donc 1 comme valeur de `i`. On a alors `W=[5 1 3 1]`.
3. 3 convient comme valeur de `i`. On a alors `W=[6 2 4 1]`.

4. 2 convient comme valeur de `i`. On a alors `W=[6 3 4 1]`.
5. 3 convient comme valeur de `i`.

Ainsi, tous les processus ont été servis. Le système est donc dans un état sur. Supposons à présent que le processus P_5 exprime la requête décrite par `[1 0 1 0]`. Pour déterminer si cette requête peut être immédiatement satisfaite, appliquons le second algoritme présentée ci-dessus :

- `[1 0 1 0]<=[2 1 1 0]`. On passe donc à l'examen des ressources disponibles...
- `[1 0 1 0]<=[1 0 2 0]`. On peut donc simuler l'allocation et tester l'état du système obtenu...
- Dans le nouveau système, la cinquième ligne du second tableau devient `[1 1 0 0]`, la cinquième ligne du premier tableau devient `[1 0 1 0]` et `A` devient `[0 0 1 0]`.

1. 4 convient comme valeur de `i`. On a alors `W=[1 1 1 1]`.
2. 1 convient comme valeur de `i`. On a alors `W=[4 1 2 2]`.
3. 2 convient comme valeur de `i`. On a alors `W=[4 2 2 2]`.
4. 3 convient comme valeur de `i`. On a alors `W=[5 3 3 2]`.
5. 5 convient comme valeur de `i`. Tous les processus ont été servis. L'état est donc sur et l'allocation peut donc d'opperer sans risque.

Remarquons que nous aurions pu servir P_5 dès le second tour et ainsi trancher la question dès ce stade, puisque nous serions retombés dans la configuration de la question précédente !

Application : Considérons les données suivantes :

	R_1	R_2	R_3	R_4
P_1	3	0	1	1
P_2	0	1	1	0
P_3	1	1	1	0
P_4	1	1	0	1
P_5	0	0	0	0
alloués	5	3	3	2
libres	1	0	1	0

	R_1	R_2	R_3	R_4
P_1	1	1	0	0
P_2	0	1	0	2
P_3	3	1	0	0
P_4	0	0	1	0
P_5	2	1	1	0

Déterminer l'état du système ainsi décrit. Le processus P_5 exprime la requête décrite par `[0 0 1 0]`. Est-ce que satisfaire à cette demande comporte des risques ?

4.3.3 Méthodes de détection et de restauration

Les méthodes de détection et restauration sont les dernières méthodes actives de gestion des deadlocks. A la différence des précédentes, elles présentent le risque de pertes. En revanche, elles sont potentiellement bien moins couteuses que les précédentes. Nous préciserons cela plus loin.

Cas où chaque ressource est en unique exemplaire

A partir du graphe d'allocation de ressources, nous construisons un graphe d'attente obtenu en faisant disparaitre les nœuds R_j mais en conservant les relations induites entre les nœuds des P_i :

$P_i \rightarrow P_k$ existe si et seulement s'il existe une ressource R_j telle que les arcs $P_i \rightarrow R_j$ et $R_j \rightarrow P_k$ existent.

La figure 4.5 illustre cette transformation.

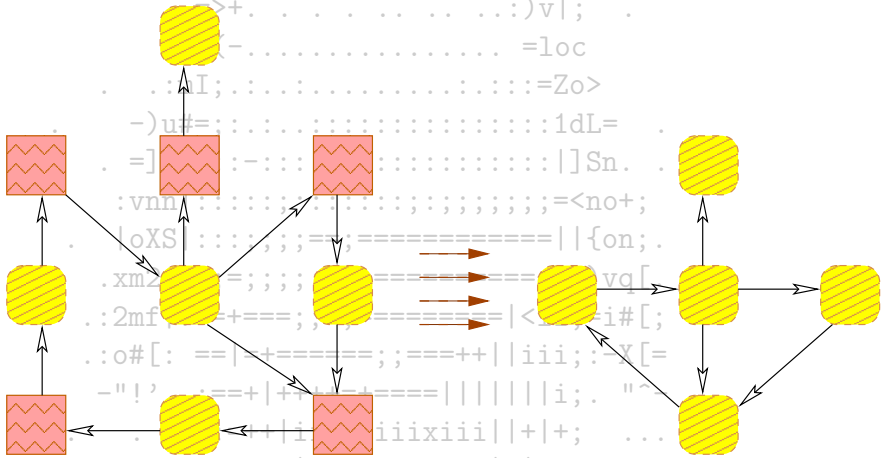


FIG. 4.5 – Construction d'un graphe d'attente

Dans un graphe d'attente, un cycle existe si et seulement si un interblocage est présent dans le système. Pratiquement, il est possible d'avoir recours à des algorithmes de recherche de circuit pour révéler la présence de circuit.

Cas où chaque ressource peut être en multiple exemplaire

Les données caractérisant l'état du système sont les suivantes :

– $A[j]$ est le nombre d'instances de R_j disponibles ;
– $D[i,j]$ est le nombre d'instances de R_j détenues par P_i ;
– $R[i,j]$ est le nombre d'instances de R_j que le processus P_i réclame.

L'algorithme suivant permet de détecter la présence d'un interblocage dans le système :

1.
```
W:=A
pour tout i, faire
      si D[i,.]<>0
       alors F[i]:=false
       sinon F[i]:=true
```

2.
```
trouver i tel que :
   - F[i]=false
   - R[i,.]<=W
Si aucun i ne convient alors aller en 4.
```

3.
```
W:=W+D[i,.]
F[i]:=true
aller en 2
```

4.
```
si F[i]=false pour certain i, il y a interblocage.
De plus, si F[i]=false, alors Pi est bloque
```

Application : Considérons les données suivantes :

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2
alloués	7	2	6
libres	0	0	0

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	0
P_5	0	0	2

Examinons la présence d'interblocage (nous pourrons également déterminer l'état du système). `A` est donnée par la dernière ligne du premier tableau. `D` est donnée par les cinq premières lignes du même tableau tandis que `R` est donnée par le second tableau.

1. `W=[0 0 0]` et `F=[F F F F F]`

2. 1 convient comme valeur de `i`. Par suite `W=[0 1 0]` et F=[T F F F F].
3. 3 convient comme valeur de `i`. Par suite `W=[3 1 3]` et F=[T F T F F].
4. 2 convient comme valeur de `i`. Par suite `W=[5 1 3]` et F=[T T T F F].
5. 4 convient comme valeur de `i`. Par suite `W=[7 2 4]` et F=[T T T T F].
6. 5 convient comme valeur de `i`. Par suite `W=[7 2 6]` et F=[T T T T T].

Ainsi, il n'y a pas d'interblocage dans le système.

Exercice : Examiner à nouveau la question si on tient compte d'une demande d'une instance supplémentaire de R_3 par P_3.

Usage de l'algorithme de détection de deadlock

Nous disposons d'un moyen de détection des inter blocages. Il reste à savoir quand l'utiliser. La réponse est probablement fonction de deux paramètres :
– la fréquence à laquelle interviennent les inter blocages
– le nombre de processus susceptibles d'être bloqués lors d'un interblocage

Idéalement, il est intéressant de lancer l'algorithme de détection dès qu'une requête ne peut être satisfaite immédiatement. Cela permet en plus d'identifier un des processus en cause dans l'éventuel blocage. Toutefois cela génère un coût calculatoire important.

Une solution plus économique consiste à lancer l'algorithme à des intervalles réguliers (e.g. une fois par heure) ou lorsque l'utilisation du CPU tombe sous un seuil donné (e.g. 40%).

4.3.4 Traitement des interblocages

En prévision de la détection d'un blocage, il est possible d'envisager un traitement manuel du problème ou une gestion par le système. Dans un cas comme dans l'autre, nous pouvons tuer un ou plusieurs processus ou choisir de retirer des ressources à des processus bloqués...

Terminaison de processus

La solution la plus radicale consiste à mettre fin à tous les processus. Alternativement, nous pouvons procéder à la terminaison des processus un par un, jusqu'à la fin de la situation de blocage. Divers raffinement peuvent être portés à cette deuxième solution. Notemment concernant l'ordre de choix des processus à terminer. L'idéal recherché est de mettre fin en priorité à des processus induisant le moins de perte...Ce critère ne conduisant à aucun critère pratique effectif, divers indicateurs peuvent servir à guider le choix : la priorité du processus, le temps d'exécution consommé, les types de ressources employés...

Préemption de ressources

Cette solution a pour objectif de reprendre certaines ressources à des processus bloqués pour les attribuer à d'autres processus bloqués jusqu'à ce que la situation de blocage prenne fin. Elle soulève différentes questions :

- Quelle ressource est visée par la préemption ? Quel processus choisir ?
- Que faire du processus amputé d'une ressource ? Le redémarrage est souvent la solution réalisable à moindre frais...
- Existe-t-il un risque de famine pour certains processus ? Comment s'assurer que les processus choisis ne seront pas toujours les mêmes (ou plus exactement, que ce ne seront pas toujours les instances de certains programmes qui souffriront de la préemption).

Chapitre 5

Communication des processus

Les processus ont besoin de communiquer. La communication entre processus peut se baser sur :

- le partage d'objets et de variables. Malgré le mot "partage", des situations de concurrence peuvent se produire entre les processus au cours desquelles les variables sont mal utilisées. Le partage cohérent de données entre plusieurs processus est une chose difficile à atteindre.
- l'échange de messages

Considérons le problème de la gestion des impressions d'un système. Lorsqu'un processus souhaite imprimer un fichier, il place le nom du fichier dans le repertoire servant de tampon (spouleur). Périodiquement, le démon d'impression vérifie si des noms de fichiers ont été déposés dans ce répertoire, et si c'est le cas, il en choisi un pour lancer son impression et le retire du spouleur.

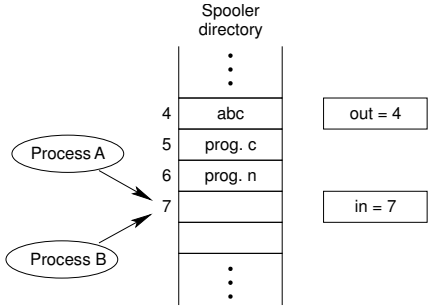


FIG. 5.1 – Pb de partage d'information dans le contexte de l'impression

Supposons que le spouleur dispose d'un nombre illimité d'emplacements numérotés pour stocker les noms de fichiers. Deux variables facilitent la gestion du spouleur : *in* contient la valeur du prochain slot dans lequel sera placé le nom du fichier à imprimer et *out* contient le numéro du prochain slot qui servira à l'impression. Dans notre exemple, nous supposons que le prochain slot libre est $in = 7$. A peu près simultanément, deux processus A et B décident d'effectuer des impressions. A lit et enregistre le contenu de in. Une commutation en faveur de B permet à B de lire et d'enregistrer in, de placer le nom de fichier à imprimer dans le slot et de placer la valeur 8 dans la variable in. Après un temps, le processus A reprend la main : il écrase le contenu du 7^{e}slot par le nom du fichier à imprimer et place la valeur 8 dans in.

Dans ce cas, la gestion des impressions ne devient pas chaotique, mais une impression est perdue. C'est l'accès concurrentiel sur la variable in qui est à la source de ce problème. Si l'opération de lecture de la variable in et sa mise à jour pouvaient se faire à l'abri des interruptions, tout se passerait mieux...

5.1 Exclusion mutuelle et sections critiques

Comment éviter de telles situations ? En empéchant plus d'un processus de lire ou écrire sur des ressources partagées, i.e. en mettant en place un mécanisme d'exclusion mutuelle permettant de s'assurer que lorsqu'un processus utilise une ressource partagée, les autres ne pourront en faire de même. La plupart du temps, les processus sont occupés à faire des calculs internes sans accéder à des ressources partagées, mais lorsqu'ils y accèdent, ils peuvent se trouver en situation de concurrence pour leur utilisation.

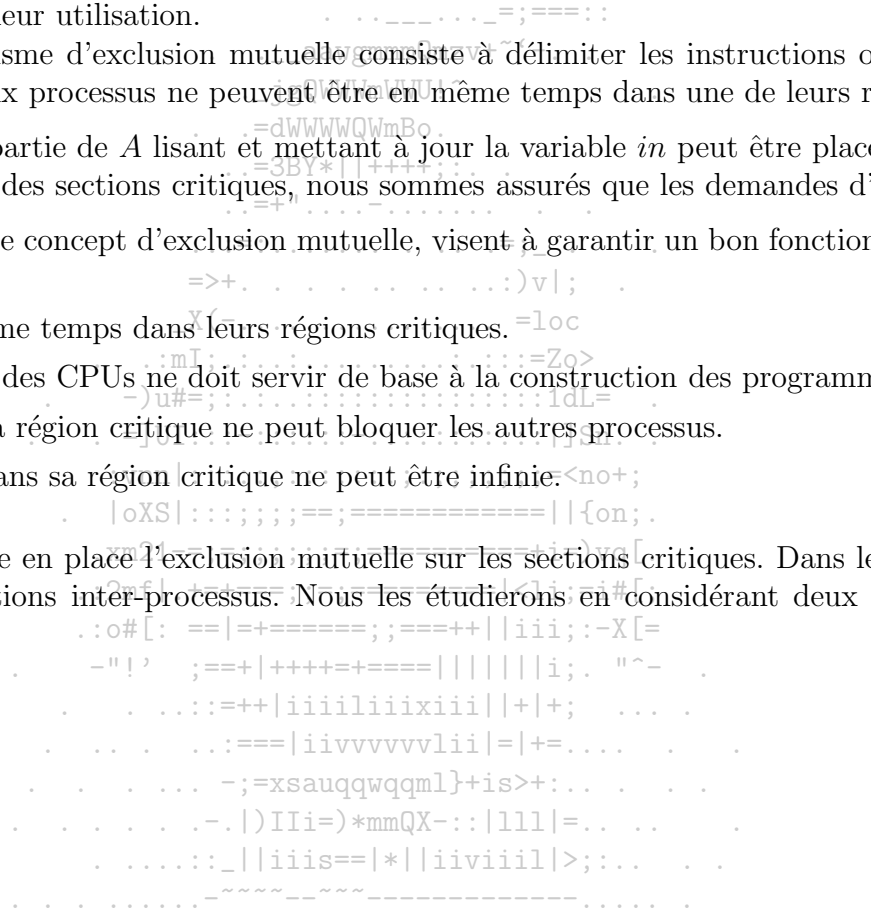
Une façon de mettre en place un mécanisme d'exclusion mutuelle consiste à délimiter les instructions opérant sur les ressources partagées dans des sections dites critiques et à s'assurer que deux processus ne peuvent être en même temps dans une de leurs régions critiques.

Ainsi, dans notre exemple précédent, la partie de A lisant et mettant à jour la variable in peut être placé dans une section critique, et de même pour B. Avec un mécanisme d'exclusion mutuelle des sections critiques, nous sommes assurés que les demandes d'impression ne pourront plus être perdues.

Les conditions suivantes, qui complètent le concept d'exclusion mutuelle, visent à garantir un bon fonctionnement de processus parallèles et partageant des ressources :

1. Deux processus ne peuvent être en même temps dans leurs régions critiques.
2. Aucune hypothèse sur la vitesse du ou des CPUs ne doit servir de base à la construction des programmes correspondant aux processus.
3. Aucun processus s'exécutant hors de sa région critique ne peut bloquer les autres processus.
4. L'attente d'un processus pour entrer dans sa région critique ne peut être infinie.

Il y a différentes solutions visant à mettre en place l'exclusion mutuelle sur les sections critiques. Dans les sections suivantes, nous nous y interessons ainsi qu'à d'autres solutions de communications inter-processus. Nous les étudierons en considérant deux processus pouvant s'exécuter en parallèle et comportant chacun une section critique.



5.2 Solutions à l'exclusion mutuelle avec attente active

5.2.1 Solutions logicielles

Méthode des co-routines ou mécanisme d'alternance stricte

```
int tour ; /* variable de commutation de droit a la */
           /* section critique de valeur 1 pour P1, 2 pour P2 */
main () {
 tour = 1; /* P1 peut utiliser sa section critique */
 parbegin
       p1();
       p2();
 parend
}

p1() {
 for ( ; ; ) {
        while (tour == 2); /* c'est au tour de P2 ; P1 attend */
        crit1;
        tour = 2; /* on redonne l'autorisation a P2 */
        reste1;
        }
}

p2() {
 for ( ; ; ) {
        while (tour == 1); /* c'est au tour de P1 ; P2 attend */
        crit2;
        tour = 1; /* on redonne l'autorisation a P1 */
        reste2 ;
        }
}
```

Critiques :

- cette solution permet l'exclusion mutuelle des sections critiques, mais le blocage de `p1` ou `p2` entraine le blocage de l'autre...Cette solution ne vérifie donc pas la troisième condition posée page 65 et n'est pas acceptable.
- L'ordre de démarrage des routines est fixé.
- Les routines sont contraintes de fonctionner à la même fréquence.
- Par ailleurs, la solution est basée sur des tests en boucles consommant du temps CPU pour pas grand chose ! Ce mode d'attente d'un processus est dite attente active.

Utilisation de variables de vérouillage

```
int c1, c2 ;
/* cles de P1 et P2 - valeur 0: le processus est en section critique */
/*                  - valeur 1: il n'est pas en section critique */
main () {
        c1=c2 = 1; /* initialement, aucun processus en section critique */
        parbegin
                p1();
                p2();
        parend
}

p1 () {
        for ( ; ; ) {
                while (c2 == 0); /* P2 en section critique, P1 attend */
                c1 = 0; /* P1 entre en section critique */
                crit1 ;
                c1 = 1; /* P1 n'est plus en section critique */
                reste1 ;
        }
}

p2 () {
        for ( ; ; ) {
                while (c1 == 0); /* P1 en section critique, P2 attend */
                c2 = 0; /* P2 entre en section critique */
                crit2 ;
                c2 = 1 ; /* P2 n'est plus en section critique */
                reste2;
        }
}
```

Critiques :

- L'exclusion mutuelle n'est pas assurée, car les deux processus peuvent entrer ensembles dans leur section critique. Cette proposition n'est pas acceptable.
- `p1` et `p2` sont découplés et fonctionnent de façon indépendante. Les situations d'interblocage ne sont plus possibles.
- La construction est basée sur l'attente active de chacun des processus...

Solution de Peterson

```
int turn; /* whose turn is it? */
int interested[2]; /* all values initially 0 (FALSE) */
void enter_region(int process); /* process is 0 or 1 */{
        interested[process] = TRUE; /* show that process is interested */
        turn = 1-process; /* set flag */
        while (turn != process && interested[1-process] == TRUE) /* null statement */ ;
}
void leave_region(int process) {
        interested[process] = FALSE; /* indicate departure from critical region */
}
p1 () {
        enter_region(0)
        crit1 ;
        leave_region(0)
        reste1 ;
    }
p2 () {
        enter_region(1)
        crit1 ;
        leave_region(1)
        reste1 ;
    }
main () {
        turn= 1; /* initialement, aucun processus en section critique */
        parbegin
                p1();
                p2();
        parend
}
```

La solution proposée par Peterson repose sur l'emploi d'une variable partagée `turn` et de variables de demande.
Critique :

- Supposons que `p1` et `p2` sont tous les deux dans leur région critique. Alors nous avons (`interested[0]=False` ou `turn=1`) et (`interested[1]=False` ou `turn=0`) et enfin (`interested[0]=TRUE` et `interested[1]=TRUE`). Ceci est impossible donc l'exclusion mutuelle est bien assurée.
- On vérifie que `p1` et `p2` ne peuvent être en même temps en attente active.
- On vérifie que l'un ne peut bloquer l'autre.
- On vérifie que chaque processus qui a fait sa demande d'entrée dans sa zone critique attend au plus un passage de l'autre dans sa section critique. La quatrième condition est aussi satisfaite.

5.2.2 Solutions matérielles

A côté de ces mises au point de solutions logicielles pour la mise en place de l'exclusion mutuelle existent de possibilités de solutions offertes par le matériel.

Masquage des interruptions[1]

Si l'OS le permet, le processus peut masquer les interruptions arrivant au CPU lors de l'entrée dans sa région critique et les réactiver juste après. Cette solution présente deux inconvénients :

- Si pour une certaine raison (plantage ou simple omission) le processus ne rétablit pas la gestion des interruptions, le système entier est bloqué.
- Sur une machine multi-processeur, ce mécanisme n'empêche pas des processus s'exécutant sur d'autre CPUs d'entrer en région critique. Cette proposition n'est donc pas valable en général.

Toutefois, ce mécanisme, s'il est risqué au niveau de processus utilisateurs peut s'avérer intéressant pour des processus du noyau.

Instruction TSL

Certaines machines (notamment multi-processeur) disposent d'une instruction élémentaire TSL (Test and Set Lock) qui fonctionne comme suit : le contenu d'un mot de la mémoire passé en paramètre est lu et placé dans un registre passé en paramètre, puis ce mot mémoire est affecté d'une valeur non nulle, et ce de façon indivisible, i.e. aucun autre CPU ne peut accéder le mot mémoire lu jusqu'à la fin de l'instruction. Cette possibilité qui s'offre à nous permet de reconsidérer l'emploi des variables de vérouillage.

```
int lock;
p1 () {
    int test;
    TSL(test,lock);
        while (test=1) do TSL(test,lock);
    crit1 ;
    lock=0;
    reste1;
}
p2 () {
    int test;
    TSL(test,lock);
    while (test=1) do TSL(test,lock);
    crit2 ;
    lock=0;
    reste2 ;
}
main () {
    lock=0;
    parbegin
        p1();
        p2();
    parend
}
```

[1] Il n'y a pas vraiment d'attente active pour l'entrée en section critique dans cette solution. Le masquage des interruptions apparait ici pour simplifier notre présentation.

5.3 Solutions à l'exclusion mutuelle avec attente passive

Les solutions précédentes ont la caractéristique de gaspiller la ressource CPU. Mais leur principal défaut est assez inattendu comme le montre l'exemple suivant.

Considérons deux processus, H un processus de haute priorité et B un processus de priorité moins forte. L'ordonnancement est tel que H est élu dès qu'il est dans l'état prêt. A un certain instant, B entame sa section critique. Un instant après H est prêt et donc il est élu. Supposons que H atteigne une région critique. Il entre en attente active car B est toujours dans sa section critique. H attend activement, sans jamais se bloquer. Il monopolise donc le CPU et empêche B de compléter sa section critique.

Ce phénomène est référencé sous le nom de problème d'inversion de priorité.

Nous allons voir à présent des solutions faisant passer un processus de l'état d'exécution à l'état bloqué pour éviter toute attente active.

5.3.1 Sémaphores

Les sémaphores sont une invention de Dijkstra permettant notamment d'apporter une solution au problème de l'exclusion mutuelle des sections critiques.

Un sémaphore est une structure à deux champs :
- une variable entière, ou valeur du sémaphore. Un sémaphore est dit binaire si sa valeur ne peut être que 0 ou 1, général sinon.
- une file d'attente de processus ou de tâches.

Dans la plupart des cas, la valeur du sémaphore représente à un moment donné le nombre d'accès possibles à une ressource.
Seules deux fonctions permettent de manipuler un sémaphore noté s :
- P(s) ou down(s)
- V(s) ou up(s)

La fonction P(s) permet de décrémenter le sémaphore d'une unité à condition que le sémaphore ne devienne pas négatif. Si le sémaphore a déjà la valeur 0, alors le processus appelant P(s) est placé dans la file d'attente du sémaphore et reste dans l'état bloqué jusqu'à ce qu'un appel de V(s) le réveille.
La fonction V(s) permet de réactiver un processus de la file d'attente du sémaphore et dans le cas où la file d'attente est vide, le compteur du sémaphore est incrémenté d'une unité.

Implémentation possible de la fonction P(s) :

```
debut
        a=1; /* a est un registre */
        TSL (&a, &verrou)
        tant que a == 1
            TSL (&a, &verrou)
        fin tant que
        si  (valeur du semaphore > 0)
        alors   decrementer cette valeur
        sinon   - suspendre l'execution du processus en cours qui a appele P(s),
                - placer le processus dans la file d'attente du semaphore,
                - le processus passe de l'etat ACTIF a l'etat  ENDORMI.
        finsi
        verrou=0
fin
```

Remarques :
- La fonction P(s) est ininterruptible grâce à l'instruction TSL intervenant dans sa construction.
- La variable `verrou` est partagée entre P(s) et V(s).
- Il y a attente active sur la variable `a`

Implémentation possible de la fonction V(s) :

```
debut
    a=1;
    TSL (&a, &verrou)
    tant que a = 1
            TSL (&a, &verrou)
    fin tant que
    si  (file d'attente non vide)
    alors
            - choisir un processus dans la file d'attente du semaphore,
            - reveiller ce processus. Il passe de l'etat ENDORMI a l'etat ACTIVABLE.
    sinon   incrementer la valeur du semaphore si c'est possible
    finsi
    verrou=0;
fin
```

Remarques :
- La fonction V(s) est aussi ininterruptible.
- La variable `verrou` partagée entre P(s) et V(s) permet à ces fonctions de s'exclurent mutuellement.
- Il y a attente active sur la variable `a`

Appliqués au problème de l'exclusion mutuelle des sections critiques de deux processus, les sémaphores permettent de construire la solution suivante :

```
SEMAPHORE s;   /* declaration tres symbolique */
main () {
    SEMAB (s,1);    /* declaration tres symbolique ; initialise le semaphore binaire s a 1 */
    parbegin
            p1 () ;
            p2 () ;
    parend
}

p1 () {
    for ( ; ; ) {
        P(s);
         .................  /* section critique de p1 */
        V(s);
        .................  /* section non critique de p1 */
    }
}

p2 () {
    for ( ; ; ) {
            P(s);
            .................  /* section critique de p2 */
            V(s);
            .................  /* section non critique de p2 */
    }
}
```

La démonstration formelle que l'algorithme résout l'exclusion mutuelle et ne crée pas d'interblocage est donnée dans [BA86] page 63. Si l'on étend cet algorithme à n processus ($n > 2$) , il peut y avoir privation (par exemple, P1 et P2 peuvent se réveiller mutuellement et suspendre indéfiniment P3 ou d'autres processus).

J.M. Morris a proposé en 1979 un algorithme de résolution de l'exclusion mutuelle sans privation pour n processus dans Information Processing Letters, volume 8, pages 76 à 80.

5.3.2 Moniteurs

Les sémaphores mal employés peuvent conduire à des situation d'interblocage (deadlock). Les erreurs issues d'un mauvais emploi de sémaphores sont délicates à repérer car leurs occurences sont imprévisibles et non reproductibles.

Les moniteurs sont des primitives d'un niveau conceptuel supérieur qui permettent de gérer ces problèmes de synchronisation. Un moniteur est un ensemble de routines, de variables et de structures de données placé au sein d'un "module" spécial. Lors de l'exécution d'un programme utilisant un moniteur, le corps du moniteur est exécuté dès le début pour initialiser l'ensemble des variables du moniteur. Les processus peuvent faire appel aux procédures du moniteur, mais ne peuvent acceder directement aux données du moniteur par des procédures extérieures.

On peut prévoir plusieurs moniteurs pour différentes tâches qui vont s'exécuter en parallèle. Chaque moniteur est chargé d'une tâche bien précise et chacun a ses données et ses instructions réservées. Si un moniteur M1 est le seul moniteur à avoir accès à la variable u1, on est sûr que u1 est en exclusion mutuelle. De plus, comme les seules opérations faites sur u1 sont celles programmées dans M1, il ne peut y avoir ni affectation, ni test accidentels.

La propriété importante des moniteurs pour la réalisation de l'exclusion mutuelle est que seul un processus peut être actif dans un moniteur à un instant donné. Cela confère des avantages indéniables aux moniteurs :

- au lieu d'être dispersées dans plusieurs processus, les sections critiques sont transformées en procédures d'un moniteur.
- la gestion des sections critiques n'est plus à la charge de l'utilisateur, mais elle est réalisée par le moniteur, puisqu'en fait le moniteur tout entier est implanté comme une section critique.

Fonctionnement

L'entrée au moniteur est gérée par une file d'attente. Pour améliorer l'accès au moniteur, des mécanismes de synchronisation existent. La synchronisation d'un moniteur repose sur la manipulation de variables de type condition, rappelant les sémaphores. Une telle variable désigne une file de processus en attente. Deux opérations permettent de manipuler une variable X de type condition : wait et signal. Un processus exécutant un $wait(X)$ est placé dans la file d'attente de X et libère l'accès au moniteur. L'appel de $signal(X)$ par un autre processus déclenche le réveil d'un processus de la file de X.

L'opération *signal* pose deux problèmes :

- Quel processus réveiller dans la file d'attente ?
- Quel processus doit reprendre son exécution ? Celui réveillé ou bien ce dernier doit-il être mis en attente le temps pour le processus appelant de compléter son exécution dans le moniteur ?

Il y a différentes écoles...

Limitation

La principale limitation du concept de moniteur est que peu de langages disposent d'un type moniteur...(Concurrent Pascal). À cela s'ajoute leur invalidité sur des architectures distribuées où chaque CPU dispose de sa propre mémoire.

5.4 Echange de messages

Certains ont estimé que les sémaphores sont de trop bas niveau et les moniteurs descriptibles dans un nombre trop restreint de langages. Ils ont proposé un mode de communication inter-processus qui repose sur deux primitives qui sont des appels système (à la différence des moniteurs) :

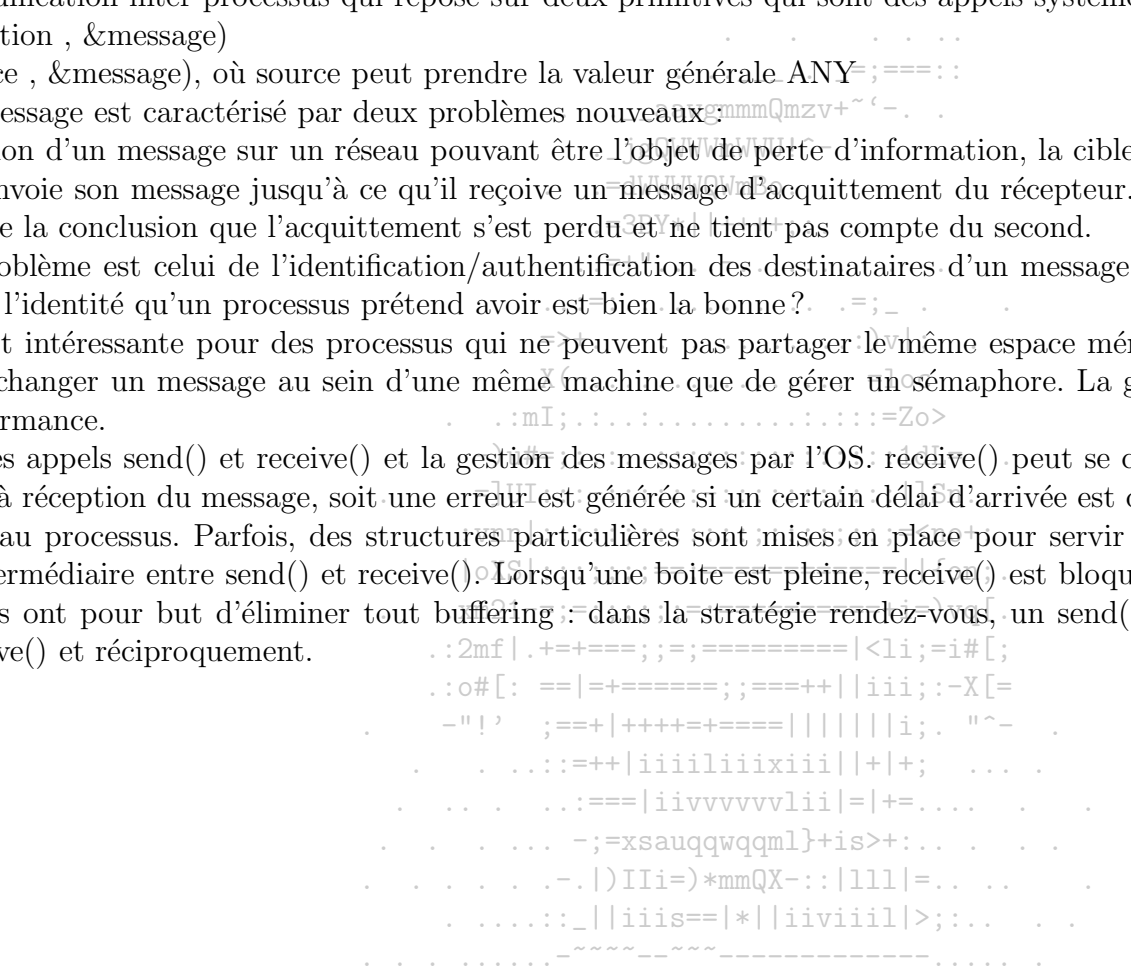
- send (destination , &message)
- receive (source , &message), où source peut prendre la valeur générale ANY

L'échange de message est caractérisé par deux problèmes nouveaux :

- La transmission d'un message sur un réseau pouvant être l'objet de perte d'information, la cible d'un message doit accuser réception d'un message. L'émetteur envoie son message jusqu'à ce qu'il reçoive un message d'acquittement du récepteur. De son côté, le récepteur qui reçoit deux messages identiques tire la conclusion que l'acquittement s'est perdu et ne tient pas compte du second.
- Le second problème est celui de l'identification/authentification des destinataires d'un message. Comment les processus sont nommés et comment s'assurer que l'identité qu'un processus prétend avoir est bien la bonne ?

Si cette solution est intéressante pour des processus qui ne peuvent pas partager le même espace mémoire, elle révèle sa faiblesse dans le cas contraire : il est plus long d'échanger un message au sein d'une même machine que de gérer un sémaphore. La généralisation de cette solution peut donc poser des problèmes de performance.

Revenons sur les appels send() et receive() et la gestion des messages par l'OS. receive() peut se comporter de différentes manières : soit le processus reste bloqué jusqu'à réception du message, soit une erreur est générée si un certain délai d'arrivée est dépassé. send() peut envoyer directement à l'adresse qui a été assignée au processus. Parfois, des structures particulières sont mises en place pour servir de tampons aux messages envoyés : des mailboxes peuvent jouer l'intermédiaire entre send() et receive(). Lorsqu'une boite est pleine, receive() est bloqué jusqu'à libération d'un peu de place dans celle-ci. Certaines stratégies ont pour but d'éliminer tout buffering : dans la stratégie rendez-vous, un send() intervenant avant un receive() est bloqué jusqu'à l'arrivée d'un receive() et réciproquement.



5.5 Applications au problème du producteur-consommateur

Un modèle de communication entre processus avec partage de zone commune (tampon) est le modèle producteur-consommateur. Le processus producteur doit pouvoir ranger en zone commune des données qu'il produit en attendant que le processus consommateur soit prêt à les consommer. La gestion des impressions illustre bien ce mode de communication. Nous supposerons que le tampon est borné, mais le cas non borné se déduit facilement du cas borné : seule l'attente du producteur disparait. La généralisation du problème du producteur et du consommateur au cas où il y a m producteurs et n consommateurs ne sera pas considérée ici. Hormis la traditionnelle gestion des travaux FIFO qui sera mise de côté pour la clarté de l'exposé, les contraintes naturelles de ce problème sont les suivantes :

Contrainte 1 Le producteur ne peut pas ranger un objet si le tampon est plein.

Contrainte 2 Le consommateur ne peut pas prendre un objet si le tampon est vide.

Contrainte 3 Il y a exclusion mutuelle au niveau de l'objet : les accès aux objets et aux variables liées aux objets doivent être placés dans des sections critiques.

```
#define N 100                                   /* number of slots in buffer */
int count = 0;                                  /* number of items in buffer */
void producer(void){
    while (TRUE) {
            produce_item();
            if (count == N) sleep();    /* if buffer full, go to sleep */
            enter_item();               /* put item in buffer */
            count = count+1;
            if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void){
    while (TRUE) {
            if (count == 0) sleep();    /* if buffer empty, got to sleep */
            remove_item();              /* take item out of buffer */
            count = count-1;
            if (count == N-1) wakeup(producer); /* was buffer full? */
            consume_item();
    }
}
```

Une solution pour le respect des deux premières contraintes consiste à endormir le producteur (resp. consommateur) lorsque le tampon est plein (resp.vide) et à le réveiller lorsque le tampon n'est plus rempli (resp. vide). Les routines `sleep()` et `wakeup()` permettent de réaliser ces fonctions de réveil et d'endormissement.

La variable `count` permet de connaître l'état du tampon. Ici, les sections critiques ne sont pas définies et des accès concurrents sur la variable `count` peuvent conduire à une situation où les deux processus sont endormis...

5.5.1 Solution avec les sémaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void){
    int item;
    while (TRUE) {
            produce_item(&item); /* generate something to put in buffer */
            down(&empty);        /* decrement empty count */
            down(&mutex);        /* enter critical region */
            enter_item(item);    /* put new item in buffer */
            up(&mutex);          /* leave critical region */
            up(&full);           /* increment count of full slots */
    }
}

void consumer(void){
    int item;
    while (TRUE) {
            down(&full);         /* decrement full count */
            down(&mutex);        /* enter critical region */
            remove_item(&item);  /* take item from buffer */
            up(&mutex);          /* leave critical region */
            up(&empty);          /* increment count of empty slots */
            consume_item(item);  /* do something with the item */
    }
}
```

Dans le code ci-contre, les routines de manipulation de sémaphores P() et V() sont renommées up() et down(). Ces notations anglaises sont bien répandues, redonnant un sens aux mnémoniques P et V, abscons pour la plupart des personnes ne parlant pas hollandais.

Si `empty` est nul, alors il n'y a plus de slot de libre et `down(&empty)` provoque le sommeil du producteur, sinon il est décrémenté. Si `mutex` est nul, alors une des deux routines est dans sa région critique et donc l'autre est endormie, sinon l'entrée dans la région critique est possible et `mutex` passe à zero, bloquant les autres sections critiques. Si `full` est nul, alors il n'y a pas un seul slot rempli et `down(&full)` provoque le sommeil du consommateur.

5.5.2 Solution avec les moniteurs

Le moniteur ci-dessous permet de regrouper les sections critiques des processus producteur et consommateur. Deux variables de type condition sont utilisées pour synchroniser les processus :

- `full` est utilisé pour le sommeil et le réveil du producteur. Le producteur s'endort avec `wait(full)` lorsque le tampon est plein et `signal(full)` est exécuté par le consommateur lorsqu'il a libéré un slot du tampon.
- `empty` est utilisé pour les état du consommateur. Le consommateur s'endort avec `wait(empty)` lorsque le tampon est vide et se réveille par le `signal(empty)` généré par le producteur lorsqu'il place un objet dans le tampon vide.

Ici, les appels à `signal()` ont lieu à la fin des procédures du moniteur, i.e. juste avant la sortie du moniteur. La proposition de construction de Brinch Hansen visant à régler le comportement après l'appel à `signal()` est ici adoptée. Les variables de type condition ne sont pas des compteurs. Les signaux peuvent être perdus. Ainsi, les appels `wait()` doivent intervenir avant les appels `signal()` associés aux mêmes variables.

```
monitor ProducerConsumer
    condition full, empty ;
    integer count;
    procedure enter;
    begin
            if count=N then wait(full);
            enter_item;
            count:=count+1;
            if count=1 then signal(empty);
    end;

    procedure remove;
    begin
            if count=0 then wait(empty) ;
            remove_item; count:=count-1;
            if count=N-1 then signal(full)
    end;

    count := 0;
end monitor;
```

```
procedure producer;
begin
        while true
        do
                begin
                produce_item;
                ProducerConsumer.enter
                end
        end;

procedure consumer;
begin
        while true
        do
                begin
                ProducerConsumer.remove;
                consume_item
                end
        end;
```

5.5.3 Solution par échange de messages

L'échange de message permet de construire une solution au problème du producteur et du consommateur qui permet de délaisser l'usage de variable placée en mémoire partagée. Pour cette solution, nous posons les conditions suivantes :

- Les messages sont tous de la même taille.
- Les messages qui ne sont pas reçus sur le champ sont bufferrisés par l'OS.

```
#define N 100                                   /* number of slots in the buffer */
void producer(void){
    int item;
    message m;                          /* message buffer */
    while (TRUE) {
            produce_item(&item);        /* generate something to put in buffer */
            receive(consumer, &m);      /* wait for an empty to arrive */
            build_message(&m, item); /* construct a message to send */
            send(consumer, &m);         /* send item to consumer */
    }
}
void consumer(void){
    int item, i;
    message m;
    for (i = 0; i < N; i++)
            send(producer, &m);         /* send N empties */
    while (TRUE) {
            receive(producer, &m);      /* get message containing item */
            extract_item(&m, &item); /* extract item from message */
            send(producer, &m);         /* send back empty reply */
            consume_item(item);         /* do something with the item */
            }
    }
```

Le consommateur commence par envoyer N messages vides. Ces messages jouent le rôle des N slots en mémoire partagée. Ainsi, le nombre de messages est constant au cours du temps.

Si le producteur produit plus vite que le consommateur consomme, alors il va se retrouver bloqué, en attente d'un message vide lorsque tous les messages seront déjà remplis. Réciproquement, le consommateur est bloqué lorsque aucun message ne lui parvient.

5.6 Problèmes classiques

5.6.1 Le problème du diner des philosophes

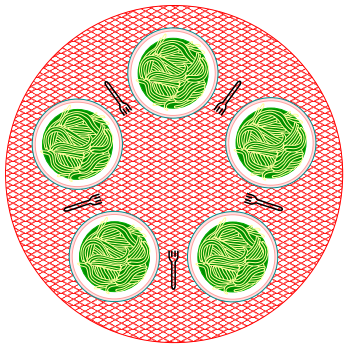


FIG. 5.2 – La table des philosophes

Cinq philosophes sont attablés autour de pâtes au basilic. La plupart du temps, chacun d'eux songe et ponctuellement, décide de se nourrir lorsque la faim se manifeste. Pour manger quelques spaghettis, un philosophe a besoin des deux fourchettes placées à sa gauche et à sa droite. Si ces dernières sont libres, il mange un peu et repose ses fourchettes pour reprendre le fil de sa réflexion. Sinon, il doit attendre que les deux fourchettes soient libres pour pouvoir manger.

Le problème est d'écrire un programme pour le comportement du philosophe tel que chaque philosophe arrive à se nourrir (i.e. évitant les situations de famine pour la ressource "spaghetti"). Ci-dessous, nous proposons une solution naïve ne tenant pas compte de probables mauvais enchaînements. Supposons que dans une première passe, chaque processus philosophe prenne la fourchette située à sa gauche. Chacun d'eux la conserve jusqu'à ce que la fourchette droite soit libre, ils restent ainsi bloqués sans jamais se nourrir. La solution proposée n'est donc pas viable. Supposons que cette proposition soit aménagée pour qu'après la prise de la fourchette gauche, si la fourchette droite n'est pas libre, la fourchette gauche est libérée provisoirement. Le même enchaînement peut se produire entrainant le même phénomène de blocage. Une possibilité pour éviter le blocage est de placer les instructions après `think()` en zone critique délimité par un sémaphore. Ceci pose toutefois un autre problème : seul un philosophe peut manger alors que deux philosophes pourrait manger. Ce défaut est accentué dans le cas d'un nombre de philosophes plus important.

Dijkstra a posé ce problème et lui a apporté une solution. Depuis, il est traditionnel de tester de nouvelles routines de synchronisation et de montrer qu'elles permettent de construire une solution à ce problème. Ce problème est intéressant pour modéliser des processus qui sont en concurrence pour un accès exclusif à des ressources en nombre limité (périphériques d'I/O).

```
#define N 5 /* number of philosophers */
void philosopher(int i) /* i: philosopher number, from 0 to 4 */{
    while (TRUE) {
            think(); /* philosopher is thinking */
            take_fork(i); /* take left fork */
            take_fork((i+1) % N); /* take right fork; % is modulo operator */
            eat(); /* yum-yum, spaghetti */
            put_fork(i); /* put left fork back on the table */
            put_fork((i+1) % N); /* put right fork back on the table */
    }
}
```

Un sémaphore encadrant les cinq dernières instructions permet une solution limitant le nombre de philosophes pouvant manger ensemble à un.

```
#define N 5             /* number of philosophers */
#define LEFT (i-1)%N    /* number of i's left neighbor */
#define RIGHT (i+1)%N   /* number of i's right neighbor */
#define THINKING 0      /* philosopher is thinking */
#define HUNGRY 1        /* philosopher is trying to get forks */
#define EATING 2        /* philosopher is eating */

typedef int semaphore; /* semaphores are a special kind of int */
int state[N];          /* array to keep track of everyone's state */
semaphore mutex = 1;   /* mutual exclusion for critical regions */
semaphore s[N];        /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */{
    while (TRUE) {
            think();
            take_forks(i);
            eat();
            put_forks(i);
    }
}

void take_forks(int i) {
    down(&mutex);       /* enter critical region */
    state[i] = HUNGRY; /* record that philosopher i is hungry */
    test(i);            /* try to acquire 2 forks */
    up(&mutex);         /* exit critical region */
    down(&s[i]);        /* block if forks were not acquired */
}
```

```
void put_forks(i) {
    down(&mutex);        /* enter critical region */
    state[i] = THINKING; /* philosopher finished eating */
    test(LEFT);          /* see if left neighbor can now eat */
    test(RIGHT);         /* see if right neighbor can now eat */
    up(&mutex);          /* exit critical region */
}

void test(i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
    }
}
```

A l'issue de la routine `take_forks`, le i^{e}philosophe est prêt à manger si ses voisins ne mangent pas. Sinon, il s'endort car les deux fourchettes ne sont pas disponibles en même temps. Un philosophe qui a fini de manger change son état et applique `test` à son voisin de gauche : s'il peut manger, il est réveillé. Puis il fait de même avec son voisin de droite.

Cette proposition est une bonne solution : elle assure un parallélisme maximal au niveau de la restauration des philosophes.

5.6.2 Le problème des rédacteurs et des lecteurs

Le problème des rédacteurs et des lecteurs modélise bien les accès à une base de données. Une base d'information est ouverte en lecture à des processus "lecteurs" et ouverte en écriture à des processus "rédacteurs". Plusieurs processus peuvent lire en même temps la base, mais si un processus procède à une écriture, il est le seul à pouvoir accéder à la base. Même les opérations de lecture sont interdites pendant une écriture.

Comment programmer les processus "rédacteurs" et les processus "lecteurs" ?

```
typedef int semaphore;
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the data base */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void){
    while (TRUE) {
            down(&mutex);           /* get exclusive access to 'rc' */
            rc=rc+1;                /* one reader more now */
            if (rc == 1) down(&db);/* if this is the first reader ... */
            up(&mutex);             /* release exclusive access to 'rc' */
            read_data_base();       /* access the data */
            down(&mutex);           /* get exclusive access to 'rc' */
            rc=rc-1;                /* one reader fewer now */
            if (rc == 0) up(&db);   /* if this is the last reader ... */
            up(&mutex);             /* release exclusive access to 'rc' */
            use_data_read();        /* noncritical region */
    }
}

void writer(void){
    while (TRUE) {
            think_up_data();        /* noncritical region */
            down(&db);              /* get exclusive access */
            write_data_base();      /* update the data */
            up(&db);                /* release exclusive access */
    }
}
```

Si aucun lecteur ne lisait la base, alors celle-ci est peut être utilisée par un rédacteur. Il y a donc test de l'exclusion mutuelle sur la base. Une fois l'accès permis sur la base, la lecture a lieu et lorsque le lecteur a fini, s'il est le dernier, il s'assure de lever l'exclusion mutuelle sur la base.

Cette proposition est-elle satisfaisante ? Utiliser un sémaphore supplémentaire...

```
typedef int semaphore;
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the data base */
semaphore w = 1;                        /* controls access to writing */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void){
    while (TRUE) {
            down(&w);
            up(&w);
            down(&mutex);           /* get exclusive access to 'rc' */
            rc=rc+1;                /* one reader more now */
            if (rc == 1) down(&db);/* if this is the first reader ... */
            up(&mutex);             /* release exclusive access to 'rc' */
            read_data_base();       /* access the data */
            down(&mutex);           /* get exclusive access to 'rc' */
            rc=rc-1;                /* one reader fewer now */
            if (rc == 0) up(&db);   /* if this is the last reader ... */
            up(&mutex);             /* release exclusive access to 'rc' */
            use_data_read();        /* noncritical region */
    }
}

void writer(void){
    while (TRUE) {
            down(&w);
            think_up_data();        /* noncritical region */
            down(&db);              /* get exclusive access */
            write_data_base();      /* update the data */
            up(&db);                /* release exclusive access */
            up(&w);
    }
}
```

Soluton simple, sans privilège pour l'une des deux routines...

```
typedef int semaphore;
semaphore mutex1 = 1;
semaphore mutex2 = 1;
semaphore mutex3 = 1;
semaphore db = 1;
semaphore r = 1;
int rc = 0;
int wc = 0;

void reader(void){
    while (TRUE) {
            down(mutex3);
                    down(r);
                            down(mutex1);
                                    rc++;
                                    if (rc == 1) then down(db);
                            up(mutex1)
                    up(r);
            up(mutex3);
            read_data_base();
            down(mutex1);
                    rc--;
                    if (rc == 0) then up(db);
            up(mutex1);
    }
}
```

```
void writer(void){
        while (TRUE) {
            down(mutex2);
                    wc++;
                    if (wc == 1) then down(r);
            up(mutex2);
            down(db);
                    write_data_base();
            up(db);
            down(mutex2);
                    wc--;
                    if (wc == 0) then up(r);
            up(mutex2);
        }
```

Solution inspirée de celle de Courtois, Heymans et Parnas. Les processus rédacteurs ont une priorité absolue sur les processus lecteurs.

5.6.3 Le problème du barbier

```
#define CHAIRS 5                    /* # chairs for waiting customers */
typedef int semaphore;
semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void){
    while (TRUE) {
            down(customers);   /* go to sleep if # of customers is 0 */
            down(mutex);       /* acquire access to 'waiting' */
            waiting=waiting-1; /* decrement count of waiting customers */
            up(barbers);       /* one barber is now ready to cut hair */
            up(mutex);         /* release 'waiting' */
            cut_hair();        /* cut hair (outside critical region) */
    }
}

void customer(void){
    down(mutex);               /* enter critical region */
    if (waiting < CHAIRS) {    /* if there are no free chairs, leave */
            waiting=waiting+1; /* increment count of waiting customers */
            up(customers);     /* wake up barber if necessary */
            up(mutex);         /* release access to 'waiting' */
            down(barbers);     /* go to sleep if # of free barbers is 0 */
            get_haircut();     /* be seated and be serviced */
    }
    else {
            up(mutex);         /* shop is full; do not wait */
    }
}
```

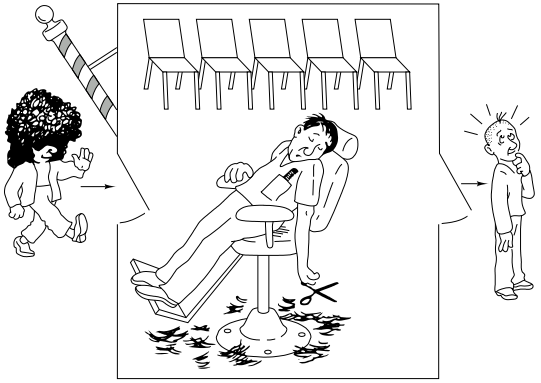


FIG. 5.3 – Le barbier

Dans l'échoppe du barbier, il y a un barbier, une chaise pour le client servi et n chaises pour permettre à quelques clients de patienter. S'il n'y a pas de clients, le barbier fait la sieste. Lorsqu'un client arrive, si le barbier dort, il le réveille. Sinon, s'il reste de la place dans la salle d'attente, il s'installe sur un chaise. Si la salle d'attente est pleine, il repart. Comment programmer le barbier et les clients ?

Que penser de la proposition ci-contre ?...

Chapitre 6

Les processus sous *x

Nous allons nous pencher sur l'implémentation du concept de processus par Unix d'une part et Linux d'autre part. Sans surprise, les similarités sont nombreuses : ces deux OSs visent à permettre la multiprogrammation des processus dans un environnement multi-utilisateurs. Précisons que derrière ces désignations, Unix et Linux, se cachent plus des familles d'OSs et une philosophie que des codes figés et immuables. Les informations concernant Linux ont d'ailleurs une valeur très relative dans le temps : Linux est toujours en pleine effervescence. Les informations fournies ici sont à vérifier (et à mettre à jour) par rapport à une distribution donnée et surtout à la version de son noyau.

Après une présentation des contextes des processus dans chacun des OSs, nous décrirons les mécanismes de création de processus et d'exécution de programme, les mécanismes d'ordonnancement, puis nous finirons avec les moyens de communication entre processus.

6.1 Implémentations des processus

Le contexte d'un processus à un instant donné est l'ensemble d'informations qui décrit le processus pour que le système puisse, avec les structures sur lesquelles il s'appuie et les opérations dont il dispose, arrêter le processus à cet instant et le redémarrer plus tard sans qu'il soit perturbé par cet arrêt.

6.1.1 Description des processus sous Unix[1]

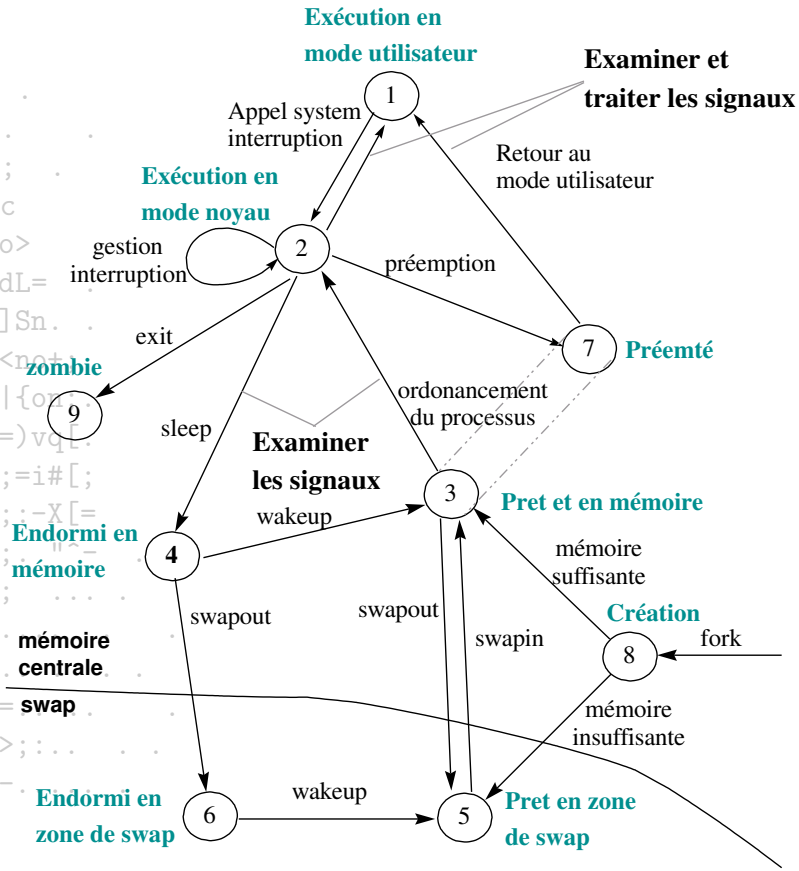
[1] sources : [REV98]

Le contexte d'un processus est composé par :
- l'état du processus
- son mot d'état informant notamment sur : le compteur ordinal, le pointeur de pile, les valeurs des registres actifs...
- les valeurs des variables globales statiques ou dynamiques
- son entrée dans la table des processus
- sa zone U
- les piles user et system
- les zones de code et de données

Les états d'un processus sous Unix

Revenons sur les différents états d'un processus sous Unix :

1. le processus s'exécute en mode utilisateur
2. le processus s'exécute en mode noyau
3. le processus ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. le processus est endormi en mémoire centrale
5. le processus est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible. (ce mode est différent dans un système à pagination).
6. le processus est endormi en zone de swap (sur disque par exemple).
7. le processus passe du mode noyau au mode utilisateur mais est préempté et il est effectué un changement de contexte pour élire un autre processus.
8. le processus est créé. Il n'est pas encore prêt et n'est pas endormi. C'est l'état initial de tout processus sauf le swappeur.
9. le processus est dans l'état zombie. Il vient de réaliser un exit, il apparaît uniquement dans la table des processus où il est conservé le temps pour son processus père de récupèrer le code de retour (exit status) et d'autres informations de gestion (coût de l'exécution sous forme de temps, et d'utilisation des ressources). L'état zombie est l'état final des processus.



Unix regroupe les informations permettant l'arrêt et le redémarrage rapide de ses processus dans une structure particulière : la table des processus.

La table des processus

La table des processus est dans la mémoire du noyau. C'est un tableau de structure `proc` (<`sys/proc.h`>). Cette structure contient les informations ci-dessous qui doivent toujours être accessibles par le noyau :

- état : ce champ permet au noyau de prendre des décisions sur les changements d'état à effectuer sur le processus.
- adresse de la zone u
- adresses : taille et localisation en mémoire (centrale, secondaire). Ces informations permettent de transférer un processus en ou hors mémoire centrale.
- UID propriétaire du processus : permet de savoir si le processus est autorisé à envoyer des signaux et à qui il peut les envoyer.
- PID,PPID identificateurs du processus et de son père : ces deux valeurs sont initialisées dans l'état 8, création pendant l'appel système fork.
- évènement : un descripteur de l'évènement attendu quand le processus est dans un mode endormi.
- priorités : plusieurs paramètres sont utilisés par l'ordonnanceur pour sélectionner l'élu parmi les processus prêts.
- vecteur d'interruption du processus : ensemble des signaux reçus par le processus mais pas encore traités.
- divers : des compteurs utilisés pour la comptabilité (pour faire payer le temps CPU utilisé) et que l'on peut manipuler par la commande alarm, des données utilisées par l'implémentation effective du système, etc.

La zone U

La zone u de type `struct user` définie dans <`sys/user.h`> est la zone utilisée quand un processus s'exécute que ce soit en mode noyau ou mode utilisateur. Une unique zone u est accessible à la fois : celle de l'unique processus en cours d'exécution (dans un des états 1 ou 2). Le contenu de la zone u est composé par :

- un pointeur sur la structure de processus de la table des processus.
- les uid réel et effectif de l'utilisateur qui déterminent les divers privilèges donnés au processus, tels que les droits d'accès à un fichier, les changements de priorité, etc.
- des compteurs des temps (users et system) consommés par le processus
- un masque de signaux : sur système V sous BSD dans la structure proc
- un terminal de contrôle du processus si celui-ci existe.
- la dernière erreur rencontrée pendant un appel système.
- la valeur de retour du dernier appel système.
- les structures associées aux entrées-sorties, les paramètres utilisés par la bibliothèque standard, adresses des buffers, tailles et adresses de zones à copier, etc.
- ".", et "/" : le répertoire courant et la racine courante (c.f. chroot())
- la table des descripteurs : position variable d'un implémentation à l'autre.
- les limites de la taille des fichiers, de la mémoire utilisable, etc (c.f. ulimit en Bourne shell et limit en Csh).
- le masque de création de fichiers umask.

Les identifiants d'un processus sous Unix

Les fichiers sous Unix ont des propriétés d'appartenance et d'accès. Ses propriétés sont définies pour trois classes d'utilisateurs : le propriétaire, les membres du groupe auquel appartient le propriétaire et tous les autres utilisateurs. L'identifiant UID est l'identifiant du propriétaire, GID est celui du groupe. Les droits de base sont la lecture, l'écriture et l'exécution. Ces divers droits sont hérités par les processus d'un utilisateur et utilisés pour les accès par le processus au système de fichier. Les identifiants utilisés par un processus sont :

- UID, GID : UID et GID de l'utilisateur auquel appartient le processus.
- effective UID et GID : certains processus permettent de changer les permissions associées aux UID et GID de l'utilisateur afin de permettre au processus d'endosser les privilèges d'un certain utilisateur ou groupe, nécessaires à ses opérations. Les UID et GID effectifs sont les nouveaux identifiants à partir desquels le noyau détermine les permissions du processus.

6.1.2 Description des processus sous Linux[2]

Chaque processus est représenté par une structure de données de type `task_struct` (Cf. `/usr/include/linux/sched.h`). Cette structure comporte directement ou indirectement (via des pointeurs) l'information relative à un processus :

- l'état du processus
- des identifiants : PID, UID, GID, etc
- des liens de parenté
- la date de création et le temps CPU consommé
- des pointeurs vers les fichiers ouverts en lecture, en écriture, un pointeur vers le répertoire contenant le programme du processus et un pointeur vers le pwd.
- l'utilisation de la mémoire virtuelle
- le contexte du microprocesseur, i.e. l'état de ses registres, de ses piles, etc

Chaque processus est associé à un objet de type `task_struct` qui lui est propre.

Les états d'un processus sous Linux

Les états possibles d'un processus sous Linux sont les suivants :

1. le processus s'exécute ou est prêt (TASK_RUNNING).
2. le processus est en attente d'un événement ou d'une ressource et interruptible (par un signal) (TASK_INTERRUPTIBLE).
3. le processus est en attente d'un événement ou d'une ressource et non interruptible (TASK_UNINTERRUPTIBLE).
4. le processus est bloqué (TASK_STOPPED)
5. Bien qu'ayant pris fin, le processus est toujours dans le vecteur de tâche (TASK_ZOMBIE).

Le processus peut être placé dans le swap, mais cet état est rarement utilisé...

[2] sources : [RUS99]

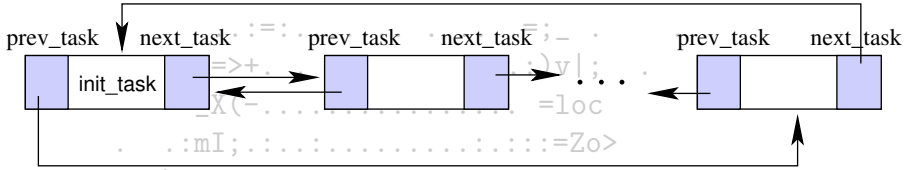
Le rôle de la table des processus sous Unix est rempli sous Linux par un vecteur de pointeurs faisant les liens vers chaque structure représentant un processus.

Le vecteur `task` et les listes de processus

Le vecteur `task` est un vecteur dont chaque élément pointe vers une structure contenant la donnée de type `task_struct` caractérisant un processus et une pile du mode noyau. L'équivalent de la plupart des informations stockées dans la table des processus se trouve dans l'objet de type `task_struct` pointé par le vecteur.

En plus de ce vecteur `task`, Linux gére différentes listes de processus pour diverses raisons :

– une liste de tous les processus.



– une liste des processus dans l'état TASK_RUNNING (run queue). L'OS peut ainsi manipuler les processus exécutables plus rapidement.

– une liste des taches libres utilisée pour faciliter la création et la destruction d'un processus.

– les processus attendant un événement peuvent être dans les états TASK_INTERRUPTIBLE et TASK_UNINTERRUPTIBLE. Ils sont répartis sur différentes classes de listes correspondant notamment à des événements particuliers (gestion d'interruptions, de timing, de synchronisation par le noyau). Ces listes sont les wait queues.

– une liste renseignant sur les relations de filiation...

– les processus dans l'état TASK_STOPPED ou TASK_ZOMBIE n'apparaisent pas dans des listes particulières.

À ces listes s'ajoutent des tables de hachage sur les PIDs permettant de retrouver rapidement l'information concernant un processus.

Les identifiants d'un processus sous Linux

En plus du PID, PPID, UID et GID réels et effectifs, utilisés pour la définition des privilèges d'un processus, Linux complète la liste par deux autres paires d'identifiants. Reprenons ces identifiants :

– file system UID et GID : ces identifiants sont normalement les mêmes que les UID et GID effectifs. Ils sont utilisés dans des mécanismes d'accès à des systèmes de fichiers montés par NFS.

– saved UID et GID : ces identifiants sont définis par la norme POSIX. Ils ont pour but de conserver les vrais UID et GID pour les processus qui changent d'UID et/ou de GID par des appels systèmes.

6.2 Création des processus

La création de processus intervient dans différents contextes :
- lors de l'initialisation du système
- exécution d'un processus système
- exécution d'un processus solicitée par l'utilisateur via son environnement de travail
- exécution d'un processus en batch

6.2.1 Initialisation du système

A l'initialisation du système, le processeur est utilisé en mode noyau. Les structures de données destinées à gérer le système par le noyau sont créées. Le processus de PID 0 est créé à partir des structures qui sont décrites dans l'image du noyau : il est créé statiquement. Tous les autres processus seront créés dynamiquement. Ce processus appelé swappeur ou gestionnaire de page a un rôle particulier. Il crée le processus de PID 1 puis il entre en attente sur tous les processeurs.

Le processus de PID 1, appelé init, jouera le rôle d'ancêtre de tous les autres processus et accueillera tous les processus orphelins.

6.2.2 création d'un processus sous Unix

Tous les processus, sauf le swappeur, sont créés par l'appel fork() qui crée une copie du processus appelant. Dans le processus fraîchement créé, le code exécutable du père est remplacé par le code du fils, et les variables, piles et autres objets sont mis à jour par rapport au nouveau processus.

6.2.3 création d'un processus sous Linux

Le processus de réplication implémenté sous Unix pour la création d'un processus est ralenti par la copie de l'espace d'adressage du père. Linux dispose de trois mécanismes différents pour créer un processus :
- copy-on-crite : les processus père et fils partagent des pages qui sont lues. Lorsque l'un d'eux écrit sur une page, Linux met à sa disposition sa propre copie de la page modifiée.
- __clone() : les processus père et fils partagent des pages et les descripteurs de fichiers. Les processus ainsi crées sont appelés processus "legers" (lightweigth process).
- vfork() : les processus père et fils partagent des pages, mais le père est bloqué jusqu'à ce que le fils se termine ou exécute un nouveau programme. Ce mécanisme est proche de celui mis en place par fork().

Les deux premiers mécanismes permettent de réduire les coûts engendrés par la copie intégrale de l'espace d'adressage du père.

6.3 Exécution de programmes

Qu'est-ce qu'un fichier exécutable et comment l'utiliser ?

6.3.1 Fichiers exécutables

Deux familles de fichiers exécutables se distinguent :

- les scripts. Il s'agit de fichiers contenant le texte de leurs instructions en clair (lisible par l'homme), et nécessitant l'interpréteur adéquat pour exécuter leurs instructions. Lorsqu'un script est lancé, il doit être reconnu comme script et l'interpréteur associé est exécuté. Perl, bash, ksh et java définissent des langages interprétés.
- les fichiers binaires. Le programme d'instruction a été compilé dans un certain format binaire. Le code exécutable résultant ainsi que des données constitue le contenu du fichier.

Par convention, la première ligne d'un script donne le nom de son interpréteur. Par exemple, pour démarrer un script reposant sur ksh (korn shell), la première ligne devrait ressembler à :

```
#!/bin/ksh
```

Unix et Linux reconnaisent le format ELF (Executable and Linkable Format), qui est le format le plus répandu sur ces plate-formes. Les formats binaires reconnus par Linux doivent être définis dans le noyau lors de la compilation du noyau, ou doivent être décrits dans des modules accessibles au noyau. Le noyau conserve une liste des formats de fichiers reconnus et lorsqu'une tentative d'exécution a lieu, chaque format de fichier est testé jusqu'à ce que l'un d'eux se révèle le bon.

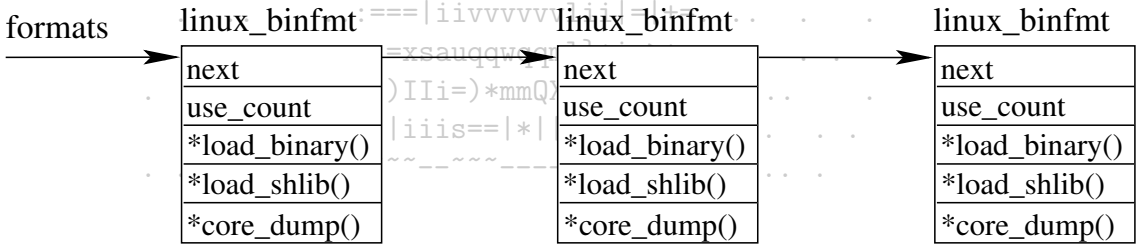


Fig. 6.1 – formats binaires reconnus par Linux

6.3.2 Lancement d'exécution

Sous Unix et Linux, l'exécution de programme peut être lancée :
– de façon interactive, via le shell ou l'environnement de travail (windowmanager)
– de façon automatique, en programmant des tâches en utilisant at, batch, crontab ou sleep...

Le shell reste la principale interface entre les utilisateurs et le système. Lorsqu'un utilisateur se connecte à un système Unix ou Linux via un terminal, il est placé sous le contrôle d'un shell de connexion. Un shell est à la fois un programme en exécution et un interpréteur de commandes qui lit chaque commande entrée au clavier et la transmet au noyau. Les shells sont nombreux, mais ils peuvent être regroupés en deux familles en raison de leurs filiations :
– sh, bash, ksh...
– csh, tcsh...

Chaque shell est basé sur une grammaire et une sémantique qui lui est propre. En plus de l'interprétation qu'ils font des commandes, ils se distinguent par des fonctionnalités qu'ils proposent : mécanisme d'historique, d'édition de la ligne de commande, d'alias, de contrôle des processus...

Nous allons approfondir l'utilisation du shell en nous intéressant plus particulièrement au korn shell.

syntaxe du Korn shell

Lors de l'analyse lexicale, l'entrée est découpée en «tokens»(léxèmes). La description suivante de la syntaxe du korn shell, issue de [BK95][3] et très succinte, définit ses jetons.

1. Caractères spéciaux

   ksh traite les caractères suivants de façon particulière : **|** **&** **;** **<** **>** **(** **)** **$** **`** **** **"** **'** **Space** **Tab** **Newline**. Pour les représenter eux-mêmes il faut les "quoter".

   ksh traite les caractères ***** **?** **[** **]** de façon particulière lorsqu'il traite les patrons.

   ksh traite les caractères **#** et **~** de façon particulière lorsqu'ils sont placés en début de mot.

   ksh traite les caractères **=** **[** **]** de façon particulière lorsqu'il traite les assignations de variables.

2. **Newline**

   Un retour chariot (**Newline**) est un jeton pouvant servir à mettre fin à une commande simple ou à séparer les parties d'une commande composée.

3. Commentaires

   Les caractères placés entre le caractère # non quoté et le caractère de fin de ligne sont traités comme un commentaire.

4. Opérateurs

   Les opérateurs sont des jetons. Ils sont reconnus partout où ils ne sont pas quotés. Il y a des opérateurs de redirection : **>** **>>** **>&** **>|** **<** **<<** **<<-** **<&** **<>**. Ces derniers peuvent être précédés d'un chiffre compris entre 1 et 9 et correspondant à un descripteur de fichier.

[3] La description de la syntaxe proposée dans [BK95] correspond à la version de ksh datée du 28/12/93. Les versions antérieures n'implémentent pas la totalité de ce qui est repris ici. Précisons par ailleurs que des programmes proches de ksh, tels que pdksh (public domain Korn shell) et bash (Bourne again shell), ne sont pas pas conformes au ksh de 1993 !

Et il y a des opérateurs de contrôle : **|** **&** **;** **(** **)** **||** **&&** **;;** **((** **))** **|&** **;&**.

5. Mots

   Un mot est un jeton constitué de nombre arbitraire de caractères et séparé à droite et à gauche par un ou plusieurs caractères non quotés d'espace, de tabulation et de retour chariot, ou l'un des caractère de contrôle ci-dessus.

6. Mots réservés

   Les mots réservés de ksh sont les suivants : **{**, **}**, **case**, **select**, **until**, **while**, **done**, **do**, **elif**, **else**, **esac**, **function**, **for**, **!**, **if**, **then**, [[,]], **in**, **time** et **fi**. ksh reconnaît ces mots réservés lorsqu'ils apparaissent :
   - comme premier mot d'une ligne,
   - après un opérateur **;** **|** **||** **&** **&&** **(** **)**,
   - comme premier mot après un mot réservé exceptés **case**, **for**, **in**, **select** et [[,
   - après **case**, **for** ou **select**, **in** est le seul mot réservé reconnu.

   De plus ksh ne reconnait pas les mots réservés dans les cas suivants :
   - si le mot est utilisé comme patron dans un **case**,
   - si le mot est utilisé comme patron à l'intérieur de **(** **)**,
   - après le signe = d'une affectation de variable,
   - à l'intérieur d'un here-document,
   - entre [[et]],
   - si un ou plusieurs caractères du mot sont quotés.

7. Alias

   Un alias est une chaîne d'un ou plusieurs caractères différents des caractères spéciaux.

8. Identifiants

   Un identifiant est un nom de variable ou de fonction. C'est une chaîne faite de caractères alphanumériques et du caractère **_** et dont le premier caractère n'est pas un chiffre.

9. Noms de variables

   Le nom d'une variable simple est un identifiant. Le nom d'une variable composée est un identifiant précédé d'un **.** ou de plusieurs identifiants séparés par un **.** et optionellement précédé d'un "**.**".

10. Affectation de variables simples

    Les affectations sont de la forme :
    - *nomvar*=*valeur*
    - *nomvar*[*index*]=*valeur*

11. Patrons

    Les patrons sont des chaînes composées de caractères spéciaux et d'autres caractères se représentant eux-mêmes. Ces chaînes fournissent des descriptions de chaînes de caractères utilisées lors du développement des commandes pour la recherche de fichiers, dans les entrées de construction **case**, dans les tests [[**..**]] et le développement des sous-chaînes. Les caractères spéciaux ***** et **?** et les paires [**..**] peuvent apparaître n'importe où dans la chaîne décrite.

    [**..**] La paire de crochets représente un caractère apparaissant dans la description de caractères spécifiée entre les crochets. Les descriptions possibles sont composées des motifs suivants :

    x-y décrit la plage des caractères ASCII compris entre le caractère **x** et le caractère **y**. Plusieurs plages peuvent être juxtaposées.

    - Le signe - placé juste après le [se représente dans la description de caractères.

    ! Le signe **!** placé juste après le [inverse le sens de la description. La paire de crochets représente n'importe quel caractère différent de ceux décrits.

    ] Le signe] placé juste après le [se représente. Il faut le placer juste après le **!** dans le cas d'une inversion de sens.

    \ Le signe \ placé devant **-**,], **!** et \ retire le sens spécial de ces caractères.

    [**:***class***:**] où *class* est l'une des classes de caractères suivantes : [**:al-**

num:], [**:alpha:**], [**:digit:**], [**:lower:**], [**:upper:**],...

? représente n'importe quel caractère.

***** représente n'importe quelle chaîne de caractères.

A partir de ces patrons élémentaires, il est possible de construire des listes de patrons avec les opérateurs **&** et **|** (ET et OU...) s'intercalant entre patrons décrits ci-dessus. A partir des listes de patrons, il est possible de construire des sous-patrons, permettant un niveau de description encore plus fin :

?(*pattern-list*) décrit zero ou une occurence de la *pattern-list*.

*****(*pattern-list*) décrit zero ou plusieurs occurences de la *pattern-list*.

+(*pattern-list*) décrit une ou plusieurs occurences de la *pattern-list*.

@(*pattern-list*) décrit une occurence de la *pattern-list*.

!(*pattern-list*) décrit toutes les chaînes exceptées celles décrites par la *pattern-list*.

Nous renvoyons les curieux à [BK95] pour un exposé plus développé.

12. Expressions arithmétiques

Le ksh permet d'utiliser le résultat qu'il évalue à partir d'expressions arithmétiques dans diverses situations : comme indice d'une variable tableau, pour préciser n'importe quel argument de **let**, à l'intérieur de **((..))** ou de **$((..))**, comme argument de **shift**...[4] Le ksh permet d'évaluer les expressions en effectuant les calculs avec des flottants double précision (mais aucune vérification d'overflow n'est effectuée ! sic...). Un certain nombre de fonctions sont utilisables telles que **abs**, **acos**, **asin**, **atan**, **cos**, **cosh** ...Aux opérateurs binaires classiques (+, −, ∗ et /) s'ajoutent les opérateurs unaires d'incrémentations (++, −−)et les opérateurs bit-à-bit (**~**, **|**, **^**). ksh permet d'inclure des opérateurs logiques dans les expressions. Le résultat d'un opérateur logique est converti en entier selon la convention suivante : VRAI correspond à la valeur un alors zero correspond à FAUX.

13. Expressions conditionnelles

ksh permet de construire des expressions conditionnelles dont l'évaluation renvoie VRAI ou FAUX. Ces expressions sont utilisées avec les commandes **test**, **[..]** ou **[[..]]**. Notons que les opérateurs doivent toujours être séparés de leur(s) opérande(s) par un ou plusieurs caractères d'espace ou de tabulation.

Les tests peuvent porter sur les informations et l'état d'un fichier :

-e *file* teste si le fichier *file* existe.

-r *file* teste si le fichier *file* existe et lisible.

-f *file* teste si le fichier *file* existe et est un fichier normal.

-d *file* teste si le fichier *file* existe et est un répertoire.

-p *file* teste si le fichier *file* existe et est un pipeline.

-L *file* teste si le fichier *file* existe et est un lien symbolique.

Les tests peuvent porter sur les chaînes. Un chaîne seule sans opérateur est évaluée à VRAI si elle est non nulle. Deux chaînes peuvent être comparées, ou une chaîne et un patron :

string1 **!=** *string2* teste si les chaînes diffèrent.

string1 **<***string2* teste si *string1* est inférieure à *string2*.

string **==** *pattern* teste si la chaîne *string* correspond au patron *pattern*.

Les tests peuvent porter sur une paire de fichiers. Par exemple :

file1 **-nt** *file2* verifie sur le fichier *file1* est plus récent que *file2* ou que *file2* n'existe pas.

Enfin, les tests permettent de comparer des expressions arithmétiques :

expr1 **-gt** *expr2* verifie que *expr1* est plus grande que *expr2*.

La liste est à compléter à partir de [BK95] ou plus simplement par un **man ksh**...

14. Quotes

Le "quoting" est le mécanisme permettant de bloquer l'interprétation d'une chaîne. Il est ainsi possible d'inhiber le sens des caractères spéciaux, des mots réservés du ksh, des alias. Cela permet également

[4] Nous renvoyons les curieux à [BK95] pour un exposé complet.

de retarder la substitution des paramètres à l'intérieur d'un here document. Les possibilités de quoting sont variées et subtiles :

\ permet de bloquer l'interprétation dans certaines situations :
- non quoté, le caractère \ est retiré et le caractère suivant garde son sens littéral (i.e. n'est pas interprété)
- à l'intérieur de **'..'**, \ garde son sens litéral
- à l'intérieur de chaînes C ANSI, il est interprété selon la définition des séquences d'échappement.
- à l'intérieur de **".."**, \ garde son sens litéral sauf lorsqu'il est suivi de **$**, #, \ ou ".
- à l'intérieur de **`..`**(backquotes), \ garde son sens litéral sauf lorsqu'il est suivi de **$**, # ou \.

Newline permet de joindre deux lignes sauf s'il est placé dans **'..'** ou après un #.

'..' permet de bloquer l'interprétation de tous les caractères ainsi délimités. Il faut utiliser \' ou "'" pour qu'une simple quote retrouve son sens littéral.

$'..' permet de décrire une chaîne C ANSI. Seules les séquences d'échappement sont interprétées au sein de la chaîne.

".." bloque l'interprétation des caractères sauf **$**, **`**, " et \ qui sont interprétés comme suit :

$((..)) est substitué par le résultat de l'évaluation de l'expression arithmétique.

$(..) substitution de la commande précisée entre **(..)** par son résultat. Les jetons placés entre **(..)** forment la commande

`..` ancien mécanisme de substitution de commande du Bourne shell. A éviter en raison de sa complexité...

$.. ou **${..}** développement du paramètre.

\ suivi de **$**, **`**, " et \, bloque l'interprétation de ces caractères. Sinon, \ est interprété de façon littérale.

15. Redirections

Dans les opérations de redirections, un ou plusieurs espacements sont autorisés entre le nom de fichier et l'opérateur, mais aucun entre le chiffre du descripteur de fichier et l'opérateur.

< *word*, *n***<** *word* Permettent d'ouvrir le fichier déterminé par l'expansion de *word* pour l'entrée standard ou le fichier associé à *n*.

<< *word*, *n*<< *word* Crée un here document. L'entrée standard de la commande est branchée sur l'entrée standard actuelle et sera fermée lorsque la chaîne *word* sera lue.

<<- *word*, *n*<<- *word* Comme précédemment, mais ksh retire les caractères de tabulation au début des lignes de l'entrée redéfinie.

<&*word*, *n*<&*word* Permettent de dupliquer, déplacer ou fermer une entrée. Si l'expansion de *word* est un chiffre, ksh duplique l'entrée associée au descripteur de fichier donné par *word* sur l'entrée standard (ou *n*). Si *word* est un chiffre suivi de -, déplace l'entrée associée au descripteur de fichier donné par *word* vers l'entrée standard (ou *n*). Si *word* est -, l'entrée standard (ou *n*) est fermée. Si *word* est p, ksh connecte la sortie du co-processus sur l'entrée standard...

<> *word*, *n*<> *word* Ouvre le fichier donné par *word* pour la lecture et l'écriture de l'entrée standard (ou *n*).

> *word*, *n*> *word* Ouvre le fichier donné par *word* pour servir de sortie standard. Si le flag d'option noclobber est positionné, ksh produit un message d'erreur et rien n'est écrit vers le fichier. Sinon, le fichier est créé si les droits le permettent ou écrasé s'il existe déjà.

>| *word*, *n*>| *word* Ouvre le fichier donné par *word* pour servir de sortie standard. Ne tiens pas compte de l'option noclobber.

>> *word*, *n*>> *word* Utilise le ficher donné par *word* pour concaténer la sortie standard.

>& *word*, *n*>& *word* Permettent de dupliquer, déplacer ou fermer une sortie. Fonctionne de manière analogue à la commande manipulant l'entrée décrite ci-dessus.

Mécanisme d'interprétation

Lorsqu'une commande est envoyée au korn shell, celui-ci lui applique un traitement en deux phases :

1. lecture de la commande, décomposition en "tokens". Dans cette première phase, ksh détermine si la commande est simple ou composée pour savoir quoi lire.
2. ksh développe et exécute une commande à chaque fois qu'elle est utilisée. Une commande composée telle que **while-do-done** est développée et exécutée à chaque passage dans la boucle.

Voyons précisément comment ces phases se déroulent pour les commandes simples. Pour le traitement des commandes composées, consulter [BK95]

Lecture des commandes

1. Découpage des commandes (splitting input into commands)

   Il est important de préciser que les alias (et les unalias) n'affectent pas les commandes placées sur la même ligne. Par ailleurs, si plusieurs commandes composées sont placées sur la même ligne, elles sont toutes lues avant que la première ne soit exécutée. Notons qu'une définition d'une fonction est une commande composée et qu'un alias placé dans une définition peut n'avoir l'effet souhaité...

   Si l'entrée de ksh est un dot script (fichier lancé par la commande "**.**"), l'ensemble des lignes est lu avant que l'exécution ne commence.

   Si l'entrée du ksh est le terminal, il lit le plus petit nombre de lignes permettant de constituer une commande complète.

   Si l'entrée du terminal ne correpond pas à une commande complète, le ksh affiche le prompt secondaire **PS2** pour indiquer que la commande doit être complétée.

2. Découpage de l'entrée en jetons (splitting input into tokens)

   L'entrée est découpée en jetons. Un jeton peut être :
   - un opérateur de redirection ou de contrôle
   - un caractère **Newline**
   - un mot réservé
   - une affectation
   - un mot
   - un here-document

   Lorsqu'un jeton incomplet est entré, le prompt secondaire est affiché.

   Pendant ce découpage a lieu la substitution des alias définis.

3. Détermination du type de commande

   Si le premier jeton est l'un des mots réservés suivants , alors ksh attend une commande composée : **{ case for function if until select time while [[!**

   Si le premier jeton est :
   (, alors ksh lit une commande fermée jusqu'à la **)** associée.
   ((, alors ksh lit une expression arithmétique jusqu'à la **))** associée.
   Si le premier jeton est l'un des mots réservés suivants , alors ksh produit un message d'erreur ou prend fin s'il n'est pas ouvert en mode interactif : **do done elif else fi in then }]] | || & && ; ;; |& ;&**

   Tous les autres jetons donnent lieu à la lecture d'une commande simple.

4. Affectation des variables (variable assignments)

   Les affectations peuvent concerner des variables simples ou composées. Les affectations des variables composées peuvent avoir lieu partout où une affectation simple peut intervenir. La règle est qu'aucun caractère d'espacement ne peut intervenir avant le caractère =.

5. Lecture des commandes simples
   Si ksh lit une commande simple, il lit tous les jetons jusqu'à l'un des caractères suivants : **;** **|** **&** **||** **|&** **Newline**. ksh classe les jetons en trois catégories :
   - Les redirections : les jetons de redirections sont lus de gauche à droite.
   - Les affectations de variables
   - Les mots de commande : les mots restants sont les mots de commande. ksh vérifie si le premier mot est un alias à substituer.
6. Substitution des alias
   ksh vérifie les alias sur le premier mot de la commande. ksh implémente un mécanisme de substitution d'alias visant à le prémunir de récursion infinies. La commande **alias** affiche les alias définis (et **unalias** permet de supprimer un alias donné). Si la valeur d'un alias se termine par un espace ou une tabulation, ksh applique le mécanisme de substitution au mot suivant. Notons que les scripts n'héritent pas des définitions d'alias !
7. Alias prédifinis
   Il existe un certain nombre d'alias prédéfinis par le ksh.
8. Substitution des messages...

Développement d'une commande simple Avant l'exécution, ksh traite chaque jeton de mots d'une commande simple comme suit.

	type de jeton		
	Affectation de variable	Mot de commande	opérateur de redirection
Lecture de la commande			
Substitution d'alias	Non	Note 1	Non
Substitution des messages	Oui	Oui	Oui
Exécution des commandes			
Développement des ~	Note 2	Oui	Oui
Substitution des commandes	Oui	Oui	Oui, Note 3
Développements arithmétiques	Oui	Oui	Oui, Note 3
Développements des paramètres	Oui	Oui	Oui, Note 3
Découpage des champs	Non	Note 4	Non
Développement des noms de fichiers	Non	Oui, Note 5	Note 6
Traitement des quotes	Oui	Oui	Oui

Note 1 : appliqué au premier mot de la commande.
Note 2 : fait après = et chaque **:**.
Note 3 : excepté après les opérateurs << et <<-
Note 4 : appliqué seulement aux portions résultant de susbstitution de commande ou de développement de paramètre.

Note 5 : sauf si **set -f** est positionné.
Note 6 : effectué avec les shells interactifs si le développement conduit à un seul chemin.

1. Développement des ~ (tilde expansion)
   ksh cherche les mots commençant par ~ et agit comme suit :
   ~ tout seul est remplacé par la valeur de la variable HOME.
   ~+ est remplacé par la valeur de la variable PWD.
   ~- est remplacé par la valeur de la variable OLDPWD.
   ~ suivi d'un nom d'utilisateur est remplacé par le chemin complet de son home.
   ~ suivi de n'importe quoi d'autre est laissé inchangé.
   De plus, ksh contrôle la présence dans les affectations de variables de ~après les signes = et **:** pour opérer à d'éventuelles substitutions comme ci-dessus.
2. Substitution des commandes (command substitution)
   ksh vérifie chaque mot pour chercher des commandes insérées dans **$(..)** ou dans **`..`**(paire de backquotes). Les **$(..)** et **`..`** sont remplacés par la sortie de la commande dans laquelle les éventuels caractères Newline finaux sont otés. Cette sortie ne se voit appliquer ni l'expansion de paramètres, ni l'expantion arithmétique, ni la substitution de commande. De plus, si cette substitution a lieu à l'intérieur d'une paire de double quotes (**"..**", grouping quotes), le découpage en champs n'a pas lieu. La commande précisée est exécutée dans un sous shell. Il n'y a donc aucun effet de bord affectant le shell courant : par exemple, les variables, les alias définis dans le sous shell ne sont pas transmis au shell parent.
3. Développements arithmétiques (arithmetic expansion)
   Chaque **$((..))** est remplacé par la valeur de l'expression arithmétique qu'il contient.
4. Développements des paramètres (parameter expansion)
   ksh cherche dans chaque mot les caractères **$** pour voir s'ils correspondent à une possible substitution de variable. Si une variable n'est pas définie et si l'option **nounset** est positionée (**set -u**), ksh génère un message d'erreur sur stderr. Si la variable n'est pas définie et **nounset** n'est pas positionnée, alors la variable est remplacée par la chaîne vide. Sinon, la variable est remplacée par son contenu.
   Si la substitution d'une variable a lieu à l'intérieur de **"..**", le découpage en champs et l'expansion des noms de fichiers ne sont pas appliqués au contenu de la variable.
5. Découpage des champs (field splitting)
   La variable **IFS** contient les caractères servant à délimiter les champs. ksh scanne les résultats des substitution de commandes, des expansions arithmétiques et des développements de variables et recherche les caractéres de l'IFS pour casser ces résultats en champs. Si la valeur de l'IFS est nulle, aucun champ n'est formé.
6. Développement des noms de fichiers (pathname expansion)
   Aprés l'étape de formation des champs, ksh parcours chaque champ pour chercher les caractères non quotés *, ?, [et (. Si un ou plusieurs de ces caractères apparaissent dans un champ, ksh le traite comme un patron. L'ensemble des fichiers correspondant au patron est substitué à ce dernier, et les fichiers sont listés par ordre alphabétique selon les règles aditionnelles suivantes :
   * et ? ne peuvent être substitués par \. Les noms de fichiers doivent absolument respecter chaque \ présent dans le patron.
   Un . placé au début de patron correspond à un nom de fichier...
   Si aucun fichier ne correspond au patron, ksh laisse le champ inchangé.
7. Traitement des quotes (Quote removal)
   Les caractères spéciaux \, " et ' sont retirés par ksh à moins qu'ils ne soient quotés. Les arguments de valeur nulle qui sont explicités (i.e. placés entre quotes) sont conservés. Les arguments de valeur nulle qui sont implicites (placés tels quels) sont retirés.

Exécution d'une commande simple Une commande simple peut être une affectation de variable, une opération de redirection, une commande spéciale propre à ksh, une fonction, une commande régulière propre à ksh ou, enfin, un programme.

1. Pas de nom de commande ou d'arguments

   Les opérations de redirections sont effectuées dans un sous-shell. Pour cette raison, elles n'ont d'effet que si elles concernent la création d'un fichier.

   Les affectations de variables sont effectuées de gauche à droite.

   La valeur retour d'une opération d'affectation est vrai sauf dans les cas suivants :
   - l'affectation a fait intervenir une commande qui a échouée.
   - une redirection a échouée à l'intérieur d'un script ou d'une fonction.
   - une affectation sur une variable en lecture seule a été tentée.
   - une affectation d'un type invalide a été tentée sur une variable de type entier ou flottant.

2. Commandes spéciales intégrées au ksh (special built-in command)

   Le shell exécute les commandes intégrées au shell dans le shell courant, modifiant directement son environnement. Ces commandes sont les suivantes : **. : alias break continue eval exec exit export newgrp readonly return set shift trap typeset unalias unset**

   Sauf pour **exec**, les opérations de redirection ne s'appliquent qu'à la commande intégrée sans affecter l'environnement courant. Les redirections appliquées à **exec** sans autre argument affectent les fichiers ouverts dans l'environnement courant.

   Les commandes spéciales intégrées sont traitées comme suit par ksh : les affectations de variables sont effectuées avant les redirections. Ces affectations restent valides jusqu'à ce que la commande prenne fin. Une erreur dans ces instructions provoque la sortie du script qui les contient.

3. Fonctions

   Si la commande n'est pas une commande spéciale intégrée, ksh vérifie parmi les définitions de fonctions. Si une fonction non définie a été déclarée avec **typeset -fu**, sa définition sera cherchée avec l'information portée par la variable **FPATH** puis avec **PATH**.

   Les redirections spécifiées dans l'appel de cette fonction ne s'appliquent qu'à cette fonction sans affecter l'environnement courant. En revanche, des redirections spécifiées avec **exec** dans la définition de la fonction modifie l'environnement courant.

   D'autres éléments sont partagés par la fonction et le shell l'appelant :
   - les variables
   - le repertoire de travail
   - les alias, les définitions de fonctions et les attributs
   - la variable spéciale **$**
   - les fichiers ouverts

   Si la fonction est déclarée avec la syntaxe **function** *nom*, ksh l'exécute dans un environnement particulier. Les éléments suivants ne sont pas partagés entre la fonction et le shell l'appelant :
   - les paramètres positionnels
   - la variable spéciale #
   - les variables spécifiées dans une liste d'affectations de variables placée dans l'appel de la fonction
   - les variables déclarées avec **typeset** dans la fonction
   - les options
   - les trappes de signaux

   Si une fonction est déclarée avec la syntaxe *name()*, ksh l'exécute comme un dot script, i.e. dans l'environnement courant. Seuls les éléments suivants ne sont pas partagés entre la fonction et le shell :
   - les paramètres positionnels
   - la variable spéciale #

   La valeur de retour d'un fonction est celle de la dernière commande exécutée par la fonction.

4. Commandes intégrées standards du ksh (regular built-in command)
   Les commandes standards du shell sont les suivantes : **bg builtin cd command disown echo false fg getconf getopts hist jobs kill print printf read test true umask wait whence**
   builtin permet d'ajouter ou de retirer des commandes standards intégrées. La plupart de ces commandes ont des effets de bord sur le shell courant.
5. Recherche de commande via PATH (PATH search)
   Si la commande spécifiée contient un caractère \, ksh exécute la commande précisée avec son chemin. Sinon, ksh cherche la commande dans les répertoires indiqués dans la variable **PATH**. L'ordre des répertoires dans cette variable est respecté dans la recherche. Si la commande n'est pas trouvée, ksh génére un message d'erreur et retourne comme valeur 127.
   Si la commande est trouvée dans un répertoire, ksh vérifie si ce répertoire est spécifié dans la variable **FPATH**. Si c'est le cas, ksh attend du fichier de la commande une définition de fonction. Sinon, ksh détermine s'il s'agit d'une commande intégrée et l'exécute dans l'environnement courant. Sinon, ksh exécute la commande donnée dans un environnement séparée. Ainsi, les commandes ne peuvent avoir d'effet sur le shell courant.
   Si le programme se termine en renvoyant la valeur Faux, et si une trappe sur **ERR** est définie, ksh exécute l'action définie associée. Si l'option **errexit** (**set -e**) est active, ksh prend fin avec l'erreur retournée par la commande. Sinon ksh exécute la commande suivante.

Cas des commandes composées Une commande composée peut être un pipeline de commande, une liste ou une commande commençant par un mot réservé ou l'opérateur de contrôle **(**.

Une redirection après une commande composée s'applique à la commande complète, sauf dans le cas de pipeline ou de liste de commande et dans le cas de la commande **time**. Dans ces dernier cas, l'opérateur de redirection ne s'applique qu'à la dernière commande. Pour appliquer un opérateur de redirection à l'ensemble des commandes dans ces derniers cas, il faut utiliser l'opérateur de regroupement commençant par **{**.

Il n'est pas possible d'effectuer d'affectation de variable dans une commande composée.

1. Pipeline (ou tube) de commandes
   C'est une séquence de commandes qui sont séparées les unes des autres par un **|**. Le format exact est le suivant : *commande* [**|** [*Newline*] *commande*] ....
   La sortie standard de chaque commande (sauf la dernière) est branchée sur l'entrée standard de la commande suivante.
   Chaque commande est exécutée comme un processus séparé, sauf eventuellement la dernière. La valeur de retour est celle de la dernière commande.
2. Commande **time**
   La syntaxe est **time** [*pipeline*]. Si *pipeline* est spécifiée, ksh l'exécute et affiche sur stderr les temps écoulés total, système et utilisateur. Sinon, ksh affiche le temps cummulé du shell et de ses enfants.
3. Commande de négation
   La syntaxe est **!** *pipeline*. ksh exécute le pipeline et la valeur retour est inversée, i.e est Vrai (zéro) si la valeur du pipeline est Faux (différent de zéro), et Faux si la valeur renvoyée par le pipeline est Vrai.
4. Liste de commandes
   Une liste de commande peut être un pipeline ou une combinaison quelconque des formats présentés ci-dessous. Les opérateurs de liste ont une précédence inférieure à celle de l'opérateur de pipeline, et sont appliqués avec une précédence plus forte pour **&&** et **||** que pour **;**, **&** et **|&**.

Les formats, dans lesquels *list* peut être remplacée par n'importe quelle liste ou pipeline, permettant de construire une liste sont :

list [**&&** [*Newline*...] *pipeline*]...Si la valeur de retour de *list* est Vrai, ksh exécute le pipeline suivant. Sinon l'exécution prend fin. La valeur retournée est celle du dernier pipeline exécuté.

list [|| [*Newline*] *pipeline*]...Si la valeur de retour de *list* est Vrai, l'exécution prend fin. Sinon, ksh exécute la suite d'instruction. La valeur retour est celle du dernier pipeline exécuté.

list [**;** *pipeline*]...ksh exécute en séquence les commandes. La valeur de retour est celle de la dernière commande exécutée.

list **&** [*pipeline* **&**]...ksh lance l'exécution de *list* sans attendre sa fin. La valeur retournée est Vrai.

list **|&** . ksh lance l'exécution de *list* comme un processus séparé mais avec ses stdin et stdout connectés à ceux du shell. La valeur retournée est Vrai.

[*Newline*] *list* [*Newline*] est la syntaxe de ce qui est désigné plus bas par listes composée.

5. Commandes conditionnelles

Construit selon le format [[*expression-test* [*newline*...]]]. *expression-test* doit être une expression conditionnelle (décrite plus haut) ou une combinaison d'expressions conditionnelles basée sur les constructions suivantes :

(*expression-test*)

!*expression-test*

expression-test **&&** *expression-test* . La valeur est Vrai si les deux expressions sont évaluées à Vrai.

expression-test **||***expression-test* . La valeur est Vrai si l'une des expressions est Vrai. Si la première est Vrai, ksh n'évalue pas la seconde

Les constructions sont listées par précédence décroissante.

Deux autres constructions permettent des exécutions conditionnelles :

if *liste-composée*

then *liste-composée*

[**elif** *liste-composée*

then *liste-composée*]

[**else** *liste-composée*]

fi

ksh ex·écute d'abord **if** *liste-composée*, et si la valeur renvoyée par *liste-composée* est Vrai, alors ksh exécute **then** *liste-composée*. Sinon, ksh tente les éventuels **elif** *liste-composée* jusqu'à ce que l'un d'eux renvoie Vrai et dans ce cas exécute le **then** *liste-composée* associé. Sinon, ksh exécute l'éventuel **else** *liste-composée*.

ksh renvoie la valeur du dernier **then** ou **else** exécuté, ou bien Vrai.

La seconde construction est :

case *word* **in**

[[(] *patron* [| *patron*...) *liste-composée* **;;**]

[[(] *patron* [| *patron*...) *liste-composée* **;&**]

...

esac

Consulter [BK95] ou les pages man pour cette construction.

6. Commandes itératives

for *var* [**in** *word*...]

do *liste-composée*

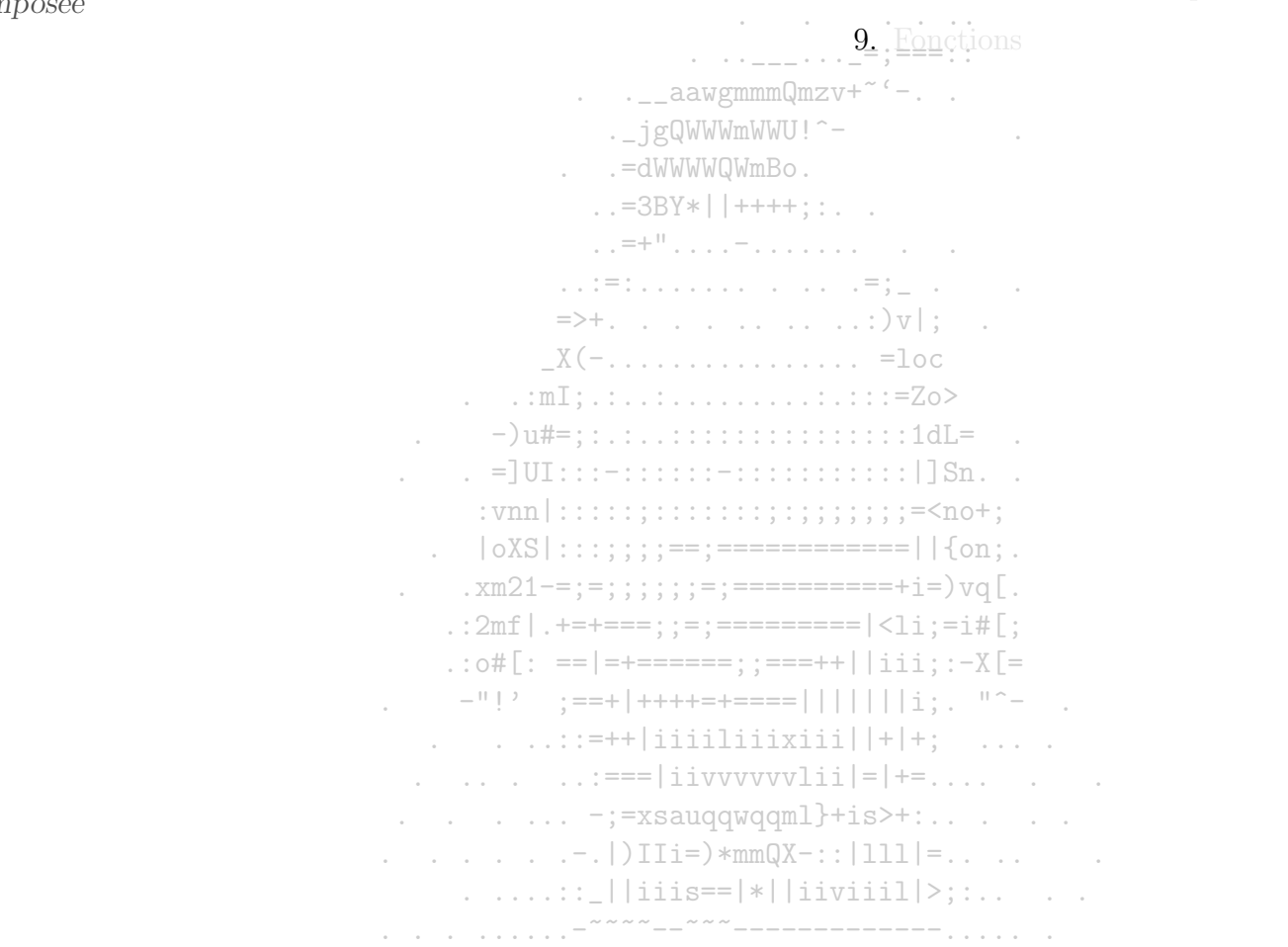
done

ksh applique la substitution de commande, de variable, le développement arithmétique, le découpage en champ, le développement de noms de fichiers et le traitement des quotes à chaque mot placé entre **in** et **do** pour générer une liste de valeur. Si **in** *word* n'apparait pas dans cette construction, ksh utilise les arguments positionnels comme s'il avait placé un **in "$@"**.

Pour chaque item généré, ksh affecte à *var* cette valeur et exécute *liste-composée*.

La valeur renvoyée est celle de la dernière *liste-composée* exécutée ou Vrai si aucune commande n'a été exécutée.

for (([*expr_init*] ; [*boucle_cond*] ; [*boucle_expr*]))
do *liste-composée*
done



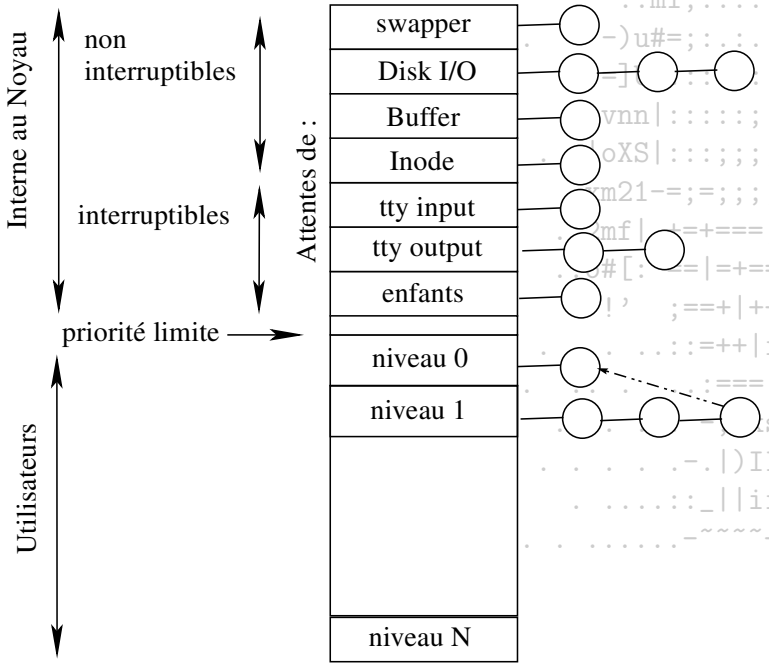
6.4 Ordonnancement des processus

6.4.1 Ordonnancement sous U

L'ordonnancement sous Unix est préemptif. Son objectif est la maximisation des quantités suivantes :

- le débit : c'est le nombre moyen de processus exécutés en un temps donné.
- le taux utile : c'est la proportion de temps réellement utilisé pour exécuter des processus utilisateurs.

Le système d'ordonnancement des processus sous UNIX (BSD 4.3 et system V4) utilise plusieurs files d'attente qui vont matérialiser des niveaux de priorité différents et à l'intérieur de ces différents niveaux de priorité, un système de tourniquet gère l'élection d'un des processus prêts à l'exécution.



Les niveaux de priorité

Le scheduler parcourt les listes une par une de haut en bas jusqu'à trouver une liste contenant un processus éligible. Ainsi tant qu'il y a des processus de catégorie supérieure à exécuter les autres processus sont en attente de l'unité centrale.

Dans les listes internes au noyau, de simples files d'attente sont utilisées avec la possibilité de doubler les processus endormis de la même liste (en effet seul le processus réveillé par la fin de son entrée/sortie est éligible). Pour les processus utilisateurs, la même règle est utilisée mais avec une préemption basée sur une politique du tourniquet.

Une priorité de base est calculée pour placer le processus dans la bonne file d'attente puis cette priorité évolue. Un processus qui utilise l'unité centrale voit sa priorité augmenter. Un processus qui libère l'unité centrale pour demander une entrée/sortie ne voit pas sa priorité changer. Un processus qui utilise tout son quantum de temps est arrêté et placé dans une nouvelle file d'attente.

Attention : plus la priorité est grande moins le processus est prioritaire.

Evolution de la priorité

Regardons la priorité et l'évolution de la priorité d'un processus utilisateur au cours du temps. Les fonctions suivantes sont utilisées dans une implémentation BSD.

Pour calculer la priorité d'un processus utilisateur, le scheduler utilise l'équation suivante qui est calculée tous les 4 tics d'horloge (valeur pratique empirique) :

$$P_{usrpri} = PUSER + \frac{P_{cpu}}{4} + 2 \times P_{nice}$$

PUSER est une constante, P_{cpu} est une variable basée sur le temps CPU consommé et P_{nice} est une valeur précisée par le propriétaire du processus. La valeur de P_{usrpri} est tronquée à l'intervalle PUSER..127. En fonction de cette valeur, le processus est placé dans une des listes correspondant à son niveau courant de priorité. Ceci nous donne une priorité calculée qui augmente linéairement en fonction de l'utilisation de l'unité centrale (il advient donc un moment où le processus actif devient le processus le moins prioritaire !).

P_{nice} est une valeur spécifiée par le programmeur grâce à l'appel système nice. Elle varie entre -20 et +20 et seul le super utilisateur peut spécifier une valeur négative.

P_{cpu} donne une estimation du temps passé par un processus sur l'unité centrale. A chaque tic d'horloge, la variable P_{cpu} du processus actif est incrémentée, ce qui permet de matérialiser la consommation d'unité centrale du processus. Afin que cette valeur ne devienne pas trop pénalisante sur le long terme (comme pour un shell) elle est atténuée toutes les secondes par la formule suivante :

$$P_{cpu} = \frac{2 \times load}{2 \times load + 1} \times P_{cpu} + P_{nice}$$

La valeur de *load* (la charge) est calculée sur une moyenne du nombre de processus actifs pendant une minute.

Pour ne pas utiliser trop de ressources, les processus qui sont en sommeil (sleep) voient leur P_{cpu} recalculé uniquement à la fin de leur période de sommeil grâce à la formule :

$$P_{cpu} = \left(\frac{2 \times load}{2 \times load + 1} \right)^{sleep\_time} \times P_{nice}$$

La variable *sleep_time* étant initialisée à zéro puis incrémentée une fois par seconde.

Les classes de priorité

La priorité des processus en mode système dépend de l'action à réaliser.

PSWAP	0	processus en cours de swap
PINOD	10	processus en attente d'une lecture d'information sur le système de fichiers
PRIBIO	20	processus en attente d'une lecture/écriture sur disque
PZERO	25	processus limite
PWAIT	30	processus en attente de base
PLOCK	35	processus en attente sur un verrou
PSLEP	40	processus en attente d'un évènement
PUSER	50	priorité de base pour les processus en mode utilisateur

Le choix de l'ordre de ces priorités est très important, car un mauvais choix peut entraîner une diminution importante des performances du système.

Il vaut mieux que les processus en attente d'un disque soient plus prioritaires que les processus en attente d'un buffer, car les premiers risquent fort de libérer un buffer après leurs accès au disque (de plus il est possible que ce soit exactement le buffer attendu par les autres processus). Si la priorité était inversée, il deviendrait possible d'avoir un interblocage ou une attente très longue si le système est bloqué par ailleurs.

De la même façon, le swappeur doit être le plus prioritaire et non interruptible : si un processus est plus prioritaire que le swappeur et qu'il doit être swappé en mémoire ...En Demand-Paging le swappeur est aussi le processus qui réalise les chargements de page. Ce processus doit être le plus prioritaire.

6.4.2 Ordonnancement sous L

Politique d'ordonnancement

Tout comme Unix, Linux est un système multiprogrammé utilisant une gestion de priorités dynamiques pour effectuer l'ordonnancement des processus. Les processus peuvent être caractérisés par rapport à la nature dominante de leurs opérations (forte utilisation du CPU pour des calculs ou nombreux accès disque) et par rapport à leur degré d'interactivité (processus interactifs ou batch ou temps réels). Linux reconnait les processus temps réels et leur affecte toujours une priorité plus élevée que celles des processus utilisateurs. En revanche, il n'a pas de moyen de distinguer les processus interactifs des processus batch. Par ailleurs, il favorise les processus qui opérent beaucoup d'I/O.

Linux utilise une politique avec préemption du CPU. Le processus actif doit libérer le CPU dans les cas suivants :

- le processus a consommé son quantum de temps CPU
- un nouveau processus passe dans l'état TASK_RUNNING et sa priorité est supérieure à celle du processus courant. Dans ce cas, le CPU est retiré au processus courant mais il reste la running queue.

Le quantum de base sous Linux est fixé à une vingtaine de tics d'horloge, ce qui correspond à environ 210ms. Cette valeur est admise comme optimisée par rapport au temps de réponse du système.

Pour l'ordonnancement, la ressource temps CPU n'est pas considérée comme un continuum par Linux. Au lieu de cela, le temps CPU est considéré par intervalle (*epoch*). Au début d'un intervalle, l'OS détermine pour chaque processus la quantité de temps CPU à laquelle il a droit pendant l'intervalle. Un processus peut se voir allouer le CPU plusieurs fois à l'intérieur d'un intervalle (tant qu'il n'a pas consommé son dû et qu'il est dans la running queue). L'intervalle prend fin lorsque tous les processus exécutables ont épuisé leur quantum. Ceux qui sont bloqués ne comptent pas.

Calcul du temps CPU

Comment la quantité de temps CPU, pour un intervalle, est-elle déterminée pour chaque processus ? Initialement, un processus dispose d'un quantum de base correspondant à vingt tics d'horloge. Si à la fin de l'intervalle le processus a consommé tout son quantum, pour le prochain intervalle il se voit allouer la même quantité de temps CPU. Sinon, un "bonus" est calculé sur la base du temps restant. Il est ajouté au quantum de base. Ce mécanisme favorise les processus effectuant beaucoup d'I/O. Lorsqu'un processus crée un processus fils, son temps restant est divisé en moitiés, l'une restant pour le père et l'autre allant au fils.

Sélection du processus

Voyons à présent comment s'opère la sélection du processus à exécuter. L'ordonnanceur alloue le CPU à l'un de ceux qui ont la plus grande priorité. En fait, Linux gère deux types de priorités :

- des priorités statiques, assignées aux processus temps réel.
- des priorités dynamiques, calculées pour tous les autres processus et qui sont la somme du quantum de base (aussi appelé priorité de base) et des tics d'horloge restant à consommer dans l'intervalle présent.

Les priorités des processus temps réels sont toujours fixées au dessus des priorités des autres processus. Ainsi, ils sont toujours servis avant les autres.

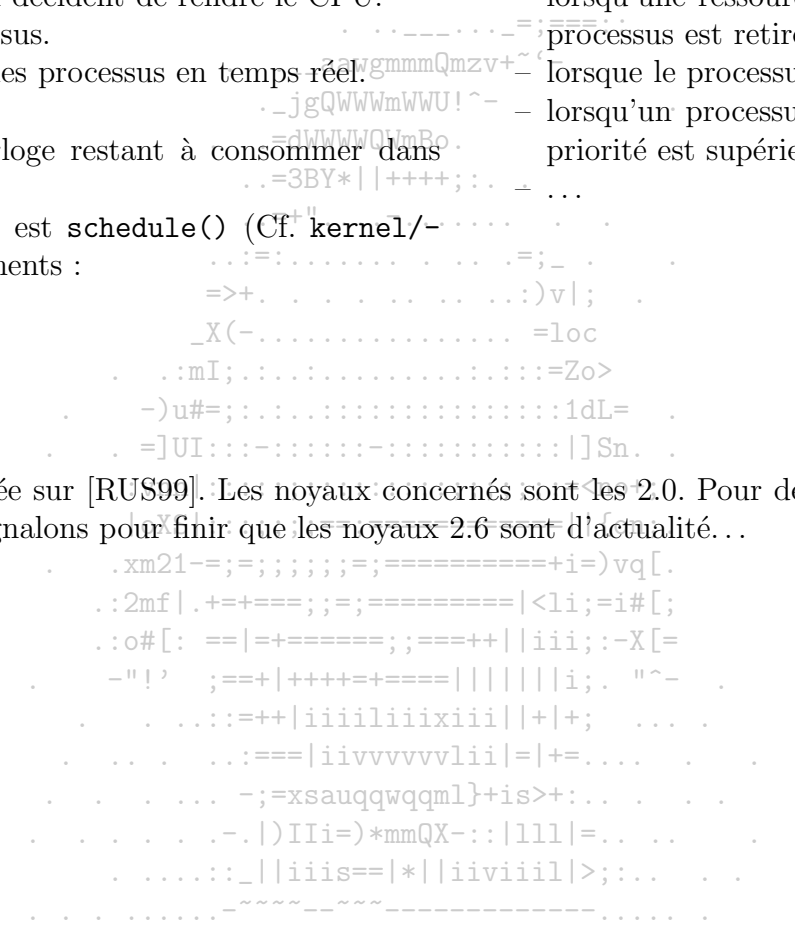
Ces données sont stockées dans l'objet `task_struct` qui contient bien d'autres informations nécessaire à l'ordonnancement :

- `need_resched` est un flag contrôlé à chaque fois qu'une gestion d'interruption prend fin, pour déterminer si l'ordonnancement doit être reconsidéré.
- le type de politique d'ordonnancement peut varier d'un processus à l'autre. Un champ précise cette politique, avec les valeurs possibles suivantes :

  - `SCHED_FIFO` pour les processus temps réel sans limitation d'utilisation de l'UC. Une politique FIFO est utilisée.
  - `SCHED_RR` pour les processus temps réel avec utilisation équitable de l'UC. Une politique round-robin (tourniquet) est utilisée.
  - `SCHED_YIELD` pour les processus qui décident de rendre le CPU.
  - `SCHED_OTHER` pour les autres processus.
- `rt_priority` est la priorité statique des processus en temps réel.
- `priority` est le quantum de base.
- `counter` est le nombre de tics d'horloge restant à consommer dans l'intervalle courant.

La routine effectuant l'ordonnancement est `schedule()` (Cf. `kernel/sched.h`). Celle-ci est appelée à divers moments :

- lorsqu'un processus est sur le point de bloquer, en raison d'une attente d'une ressource. Il est retiré de la running queue et placé dans la file d'attente appropriée à la ressource attendue. Son état devient alors soit TASK_INTERRUPTIBLE, soit TASK_ININTERRUPTIBLE.
- lorsqu'une ressource recquise par un processus devient disponible, un processus est retiré de la file d'attente correspondante...
- lorsque le processus courant a consommé tout son temps CPU.
- lorsqu'un processus est ajouté à la file des processus prêts et que sa priorité est supérieure à celle du processus courant
- ...



Rapellons que notre description est basée sur [RUS99]. Les noyaux concernés sont les 2.0. Pour des informations précises sur les noyaux de la famille 2.4, il est possible de consulter [AIV02]. Signalons pour finir que les noyaux 2.6 sont d'actualité...

6.5 Communication des processus

Les moyens de communication entre processus implémentés par Unix et Linux sont très proches. Ils sont variés et nombreux...

6.5.1 Signaux

Les signaux sont un des moyens les plus anciens de communication sous Unix. Ils sont utilisés pour signaler des événements à un ou plusieurs processus de façon asynchrone. Ils peuvent être générés via le terminal de connexion par une séquence de caractères spéciaux (**Ctrl+C** pour envoyer le signal INTR par exemple), par une erreur d'un processus tentant d'accéder à une adresse non définie dans sa zone mémoire, ou encore par le shell pendant ses opérations de jobs control.

Il y a un ensemble de signaux que seul le kernel peut générer, et les autres signaux que les processus utilisateurs peuvent utiliser. La génération d'un signal peut dépendre de permissions. Le jeu complet de signaux disponibles est listé par **kill -l**.

Les processus peuvent gérer l'arrivée d'un signal donné et même le bloquer (exception faite des signaux SIGSTOP et SIGKILL). Des actions par défaut peuvent être définies dans le noyau.

Les signaux ont deux caractéristiques importantes :

- Les ordres d'émission et de reception de deux signaux sont indépendants.
- Rien ne permet à un processus de savoir combien de fois un signal émis a été emis. La seule information concernant un signal émis jusqu'à son traitement est que le signal a été émis.

Seuls le kernel et le super utilisateur peuvent envoyer à n'importe quel processus un signal. Un processus utilisateur ne peut envoyer des signaux qu'aux processus de même UID et de même GID ou à des processus du même groupe de processus que le sien.

Les signaux ne sont pas présentés au processus dès qu'ils sont émis : le processus doit repasser dans l'état d'exécution. De plus, à la fin de chaque appel système qu'il effectue, un processus contrôle les signaux et le masque de signaux. Si l'action associée à un signal est l'action par défaut définie dans le noyau, le noyau se charge de l'appliquer. Sinon, la routine appelée est gérée par l'appel à SIGACTION qui fait passer le processus en mode noyau.

6.5.2 Pipes ou tubes

Prenons l'exemple suivant :

psselect -p1-100 -r SysEx.ps | a2ps -o - | lpr

La sortie de la commande **psselect -p1-100 -r SysEx.ps** est envoyée en entrée de la commande **a2ps -o -**, et la sortie de cette dernière est envoyée vers l'entrée de la commande d'impression **lpr**. Les commandes travaillent sur un flux unidirectionnel de données comme des "filtres"[5], sans qu'elles le sachent et sans changer leur façon de travailler. Dans cet exemple, c'est le shell qui a la charge de la gestion des pipes.

Un pipe est caractérisé par les fait suivants :

– l'ordre des caractères en entrée est conservé en sortie. La traversée suit le mode FIFO.
– le pipe a une capacité finie.
– il n'est pas possible d'écrire dans un pipe dans lequel aucun processus ne lit.

Lorsque des données sont passées entre deux processus via un pipe, chaque processus utilise sa propre structure de fichier, chaque structure pointant vers le même *i-node*. La figure 6.2 montre que les opérations permises sur le pipe par chacun des processus sont différentes : l'un peut écrire, et l'autre lire

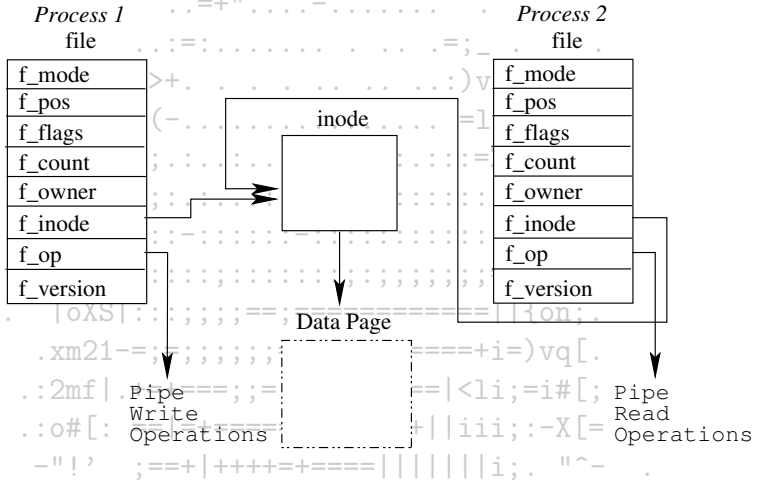


FIG. 6.2 – Mécanisme de pipe entre deux processus

le fichier. Lorsqu'un processus écrit dans le pipe, les données sont copiées vers la page partagée, et lorsque le pipe est lu, les données sont copiées à partir de la page partagée. L'OS doit synchroniser les accès au pipe. Si un seul processus lit le pipe, la lecture est destructive, i.e. les données une fois lues sont retirées du pipe. Mais il peut y avoir plusieurs processus lecteurs et de même plusieurs processus rédacteurs...

Il y a deux types de pipes. Les pipes ordinaires et les pipes nommés. La seule vraie différence est que les tubes ordinaires sont créés à l'ouverture alors que les tubes nommés peuvent avoir été créés préalablement par la commande **mkfifo**. Ils ne sont pas temporaires.

[5]Le concept de filtre est cher au monde Unix. Parmi les principes du bon programmeur : toute commande doit être conçue comme un filtre...C'est une façon de contribuer à la démarche particulière consistant à pouvoir réaliser des routines complexes à partir de routines simples.

6.5.3 Mécanismes d'IPC du système V

L'acronyme IPC s'interprète par Inter Processus Communication. Les mécanismes d'IPC permettent la communication et/ou la synchronisation dans n'importe quel couple de processus locaux (de la même machine).

Les trois mécanismes d'IPC que sont les files de messages, les segments de mémoire partagée et les sémaphores, sont uniquement basés sur la mémoire. Ils n'ont pas de liens avec le système de fichiers, ce qui les sort de la philosophie Unix.

Ces objets ne sont plus désignés localement dans les processus par des descripteurs standards, et de ce fait il n'est plus possible d'utiliser les mécanismes de lecture et d'écriture standards sur de tels objets. Ils partagent tous trois un procédé particulier d'identification/authentification. Les processus y accèdent via des appels systèmes auxquels un identifiant est fourni et les droits d'accès, définis à la façon des droits existant sur les fichiers, sont vérifiés.

Un objet IPC est implémenté par un objet de type `ipc_perm`. Le type `ipc_perm` est une structure rassemblant les identifiants du propriétaire et de l'utilisateur et de leurs groupes, les droits d'accès sur l'objet et la clé de l'objet qui sert à retrouver l'identifiant de l'objet (un processus ne peut accèder directement à un objet IPC avec sa clé. Il doit utiliser son identifiant (ou référence) qui est renvoyée par la fonction *get*). Pour Linux, on pourra consulter `include/linux/ipc.h` pour les détails sur l'implémentation des objets IPCs et `sys/ipc.h` pour Unix.

Files de messages

Une file de messages permet à un ou plusieurs processus de déposer des messages (typés) destinés à un ou plusieurs processus. Le vecteur `msgque` regroupe des pointeurs vers toutes les files de messages existantes. Chaque file, implémentée comme objet de type `msgq_id`, contient un objet `ipc_perm`, des pointeurs vers les messages de la file ainsi que les temps d'accès, de modification...et deux files de messages, l'une pour les rédacteurs et l'autre pour les lecteurs. Les fonctions `msgget`, `msgsnd`, `msgrcv` et `msgctl` (de `sys/msg.h` pour Unix, et `include/linux/msg.h` pour linux) permettent respectivement les opérations de création de file, d'envoi et de réception de message, et de manipulation de file (dont la destruction).

Lorsqu'un processus veut écrire un message dans une file, les droits d'écriture dans cette file sont vérifiés. Si le processus a le droit d'écrire dans la file et s'il reste de la place dans la file, le message est copié à la fin de la file des rédacteurs. S'il n'y a plus de place, le message est placé dans la file des messages en attente d'écriture et le processus endormi jusqu'à ce qu'un message soit lu dans cette file et libère un emplacement.

Le processus de lecture suit un fonctionnement similaire : un processus peut se retrouver bloqué. Il peut choisir de lire le premier message de la file des lecteurs ou bien le premier message d'un type donné. Si aucun message ne correspond aux critères du lecteur, il est endormi.

Sémaphores

Sous Unix (Linux permet d'utiliser les mêmes mécanismes), les sémaphores sont regroupés par tableaux et les opérations sont appliquées à des tableaux de sémaphores. Les objets de type `semid_ds` accueillent les listes de sémaphores et tous les objets de ce type sont répertoriés par le vecteur `semary`. Les sémaphores sont des objets de type `sem`. Les opérations ont lieu via des appels système. Un appel peut spécifier plusieurs opérations, chaque opération étant définie par un triplet :

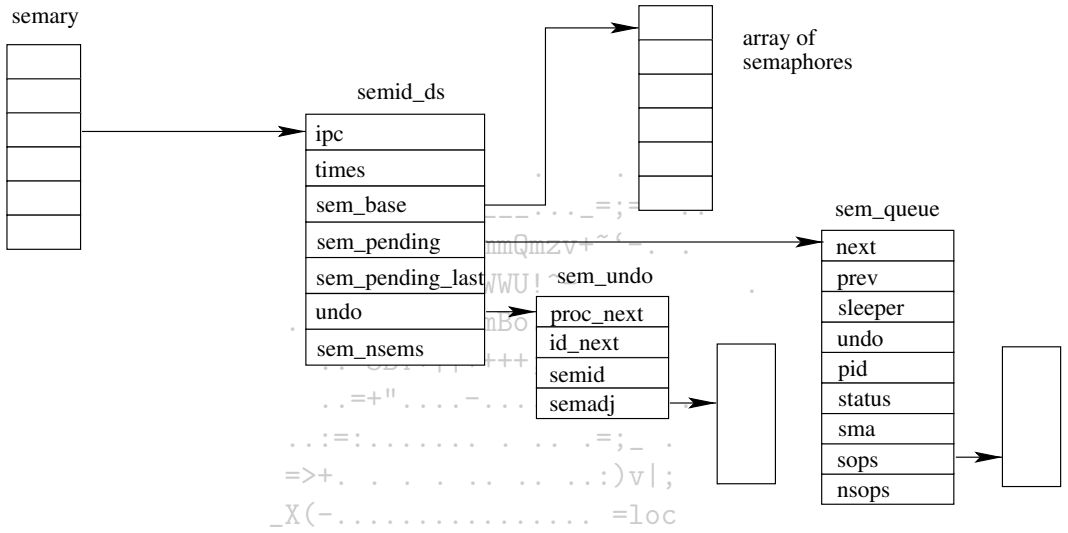


FIG. 6.3 – Implémentation des objets sémaphores

- un index précisant la position du sémaphore dans le tableau.
- la valeur de l'opération, i.e. la valeur à ajouter au sémaphore.
- un ensemble de flags précisant des options sur l'opération.

Il est dit qu'une opération "réussit" si la somme de la valeur du sémaphore et de la valeur de l'opération est strictement positive ou si la valeur de l'opération ainsi que la valeur du sémaphore sont nulles. Lorsqu'un appel est réalisé, le système vérifie que les opérations "réussiront" toutes :

- Si ce n'est pas le cas, et si les flags ne spécifient pas un appel non bloquant, alors les opérations à effectuer sont enregistrées et le processus est placé dans une file d'attente. Un objet `sem_queue` matérialise cette file d'attente et est pointé par l'item `sem_pending` de l'objet `semid_ds`.
- Si toutes les opérations réussissent, le processus n'a pas besoin d'être suspendu. Les valeurs sont ajoutées aux sémaphores concernés et le système vérifie si des processus sont en attente sur les sémaphores modifiés. Si c'est le cas, le système contrôle d'abord si les opérations en attente réussiront ou pas pour un processus donné. Si oui, elles sont appliquées et le processus est retiré de la file d'attente et réveillé (i.e. prêt). La vérification continue pour les autres processus.

Les opérations de bases sont décrites dans `include/linux/sem.h` (resp. `sys/sem.h`) pour Linux (reps. Unix). Trois fonctions permettent ces opérations :

- La fonction `sys_semget()` (resp. `semget()`) permet la création d'un nouveau sémaphore.
- La fonction `sys_semctl()` (resp. `semctl()`) permet de changer la valeur d'un sémaphore.
- La fonction `sys_semop()` (resp. `semop()`) permet d'appliquer un up ou un down sur un sémaphore.

Linux offre aussi la possibilité de manipuler les sémaphores individuellement. Les objets `semaphore` sont alors manipulés par les `up()` et `down()` (Cf. `include/asm/semaphore.h`).

Segments de mémoire partagée

La mémoire partagée consiste à partager des informations entre processus par le biais de segments de mémoire apparaissant dans la mémoire virtuelle de chaque processus impliqué dans ce partage. Comme pour tous les autres objets IPC, l'accès à un segment de mémoire partagée passe par l'utilisation du système de clés et le contrôle des droits d'accès définis sur lui. Une fois que des processus se partagent un tel segment, des mécanismes de synchronisation doivent être mis en œuvre pour assurer la cohérence des opérations d'écriture et de lecture sur cet espace.

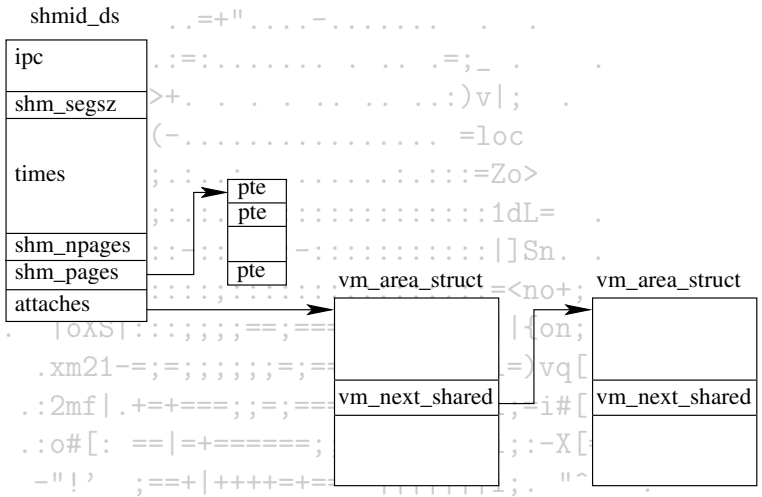


FIG. 6.4 – Implémentation des segments de mémoire partagée

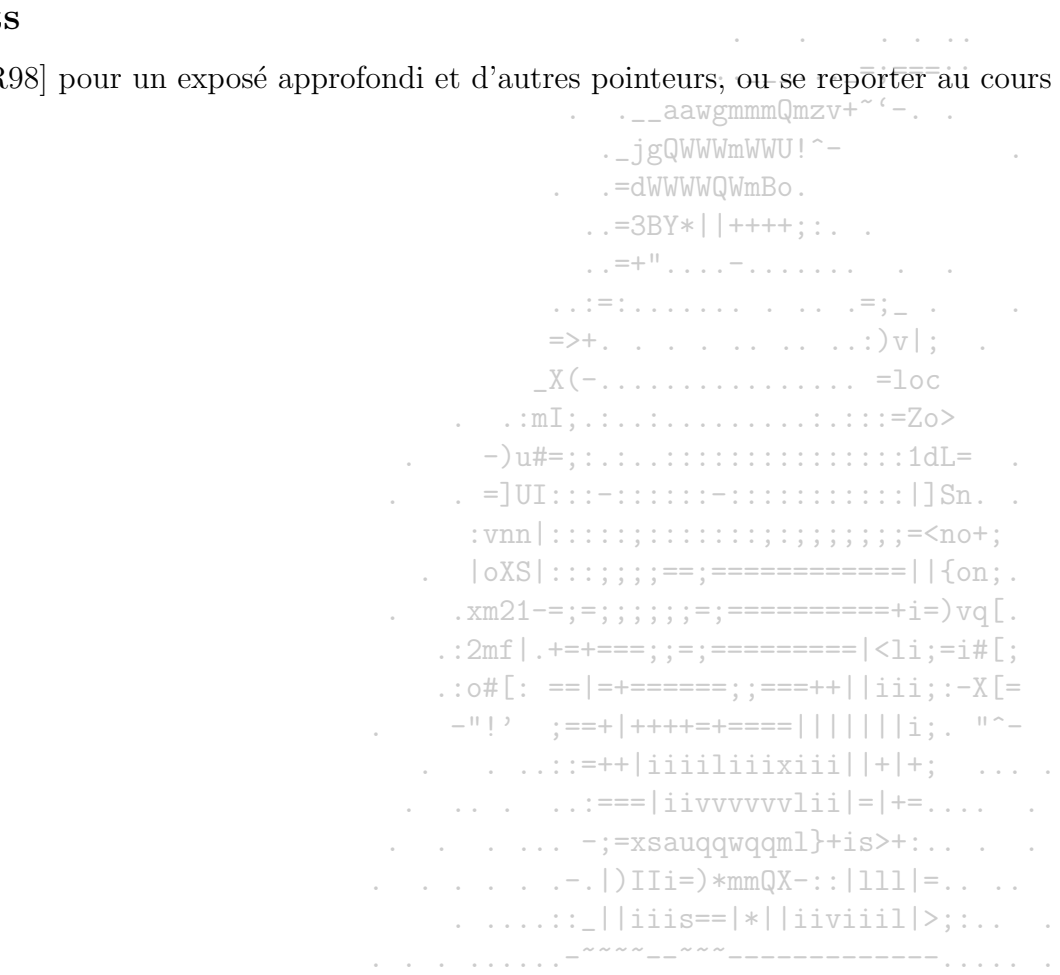
Un segment de mémoire partagée est un objet de type `shmid_ds`. Tous les objets de ce type sont pointés par le vecteur `shm_segs`. Un objet `shmid_ds` indique notamment la taille de l'espace mémoire partagé, le nombre de processus l'utilisant et les adresses des pages le constituant. C'est le créateur de l'objet qui définit les droits d'accès.

Chaque processus souhaitant accéder à un espace mémoire partagé doit s'attacher à cet espace via un appel système. Ceci a pour effet d'ajouter un objet correspondant à cet espace partagé à l'espace d'adressage virtuel du processus. Une page physique correspondant à l'espace mémoire partagé n'est allouée que lorsqu'un des processus tente d'accéder à cette page. Lorsqu'un processus ne souhaite plus partager ce segment de mémoire, il s'en détache et lorsqu'aucun processus n'y est attaché, les pages mémoire associées sont libérées.

Les fonctions `shmget`, `shmat`, `shmdt` et `shmctl` permettent la gestion des segments de mémoire partagée sous Unix (Cf `sys/shm.h`. Pour Linux, consulter `include/linux/shm.h`).

6.5.4 Sockets

Consulter [SFR98] pour un exposé approfondi et d'autres pointeurs, ou se reporter au cours traitant des réseaux.



Deuxième partie

Entrées/sorties

Chapitre 7

Gestion des entrées/sorties

Une des fonctions de l'OS est de contrôler les périphériques d'I/O. Il doit soumettre les instructions aux périphériques, surveiller les interruptions et gérer les erreurs. Il doit également fournir une interface entre ces périphériques et le reste du système. Idéalement, cette interface devrait être simple d'utilisation et, dans la mesure du possible, homogène par rapport à l'ensemble des périphériques.

7.1 I/O physiques

Nous nous bornerons ici à ce qui concerne la programmation des I/O sur un périphérique sans entrer plus dans les détails matériels (gestion de l'alimentation, utilisation des composants...laissés de côté).

7.1.1 Périphériques d'I/O

Les périphériques sont généralement caractérisés par le mode d'accès à leurs données : on distingue les périphériques dont l'accès se passe par blocs (block devices) et ceux pour lesquels l'accès se passe par caractères (character devices).

Pour les premiers, l'information est stockée par blocs de taille fixée (avec une valeur pouvant être comprise entre 512 bytes et 32768 bytes). Chaque bloc a sa propre adresse. Ce qui caractérise ce type de périphérique est qu'une opération de lecture ou d'écriture d'un bloc est indépendante des opérations sur les autres blocs. Sur un disque, il est possible de se positionner à un endroit précis d'un fichier sans avoir à lire les informations situées avant cette position.

Les character devices opèrent sur des flots de caractères, sans structure particulière. Une partie d'un flot n'est pas adressable et par conséquent des opérations de positionnement similaires à celles possible avec les blocks devices sont impossibles. Les imprimantes, les souris et les interfaces réseaux

tombent dans cette catégorie de périphériques.

Bien qu'elle ne permette pas de classer tous les périphériques existants, cette classification suffit à poser les bases d'une modélisation permettant la gestion logicielle par l'OS des I/O. Par exemple, le système de fichiers opère sur des périphériques par blocs virtuels en laissant la gestion des particularités d'un périphérique donné à un pilote associé à ce périphérique.

7.1.2 Controleurs

L'OS ne traite (presque) jamais directement avec un périphérique. Il passe par un intermédiaire appelé un contrôleur de périphérique, le contrôleur pouvant gérer plusieurs périphériques. Les instructions sont reçues par le contrôleur, et ce dernier commande le périphérique visé via une interface commune. En ce qui concerne les disques, les interfaces standards sont l'interface IDE (Integrated Device Electronics) et l'interface SCSI (Small Computer System Interface). Par exemple, un contrôleur SCSI (prenant souvent la forme d'une carte à enficher dans la carte mère) peut être connecté à plusieurs périphériques SCSI par le biais d'une nappe.

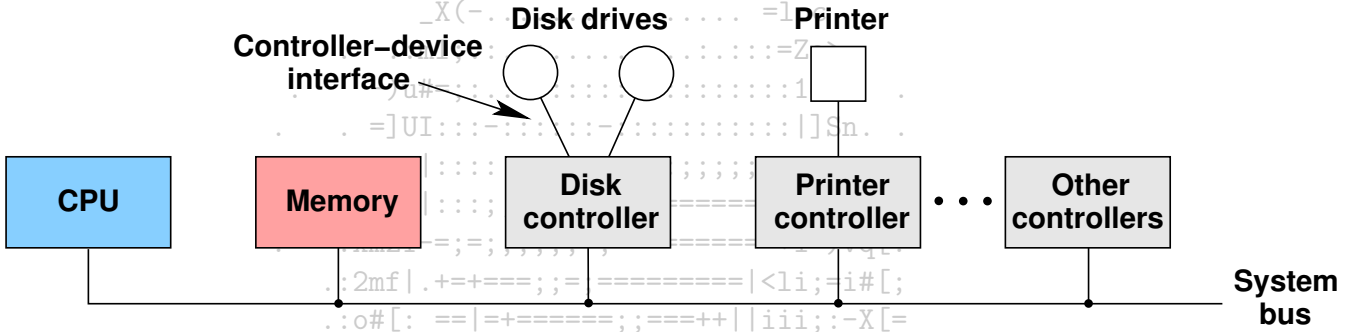


Fig. 7.1 – Connexion simple entre CPU et contrôleurs

Toutes les données sont censées transiter par le CPU qui gère les transmissions, le contrôle et la synchronisation. Sur les "petites" machines (personal computer), le CPU et les contrôleurs sont connectés par un modèle de bus simple, comme illustré par la figure 7.1. C'est par le bus système que sont transmises les données. Les gros systèmes utilisent parfois un modèle de bus multiples et des canaux d'I/O spéciaux pour alléger le CPU de certains transferts de données.

Chaque contrôleur utilise des registres pour échanger des instructions et des données (ces registres correspondent la plupart du temps à des plages d'adresses mémoire). Lorsque les registres sont prêts pour la lecture/écriture, le contrôleur génère une demande d'interruption matérielle (IRQ i.e. Interrupt ReQuest) particulière pour prévenir le CPU. Une interruption matérielle n'est autre qu'un signal électrique qui est récupéré par le contrôleur d'interruptions. A chaque type d'interruption correspond une réaction, cette mise en correspondance étant assurée par le vecteur d'interruptions.

7.1.3 DMA

Lorsque le CPU réclame la lecture d'un bloc d'adresse donnée sur un disque au contrôleur qui en gère l'accès, le contrôleur copie bit à bit les secteurs correspondants au bloc dans son buffer, réalise un contrôle d'erreur sur le bloc lu et génère une interruption. Le CPU peut alors vérifier le résultat des opérations du contrôleur et, en cas de réussite, lire le buffer pour placer son contenu en mémoire.

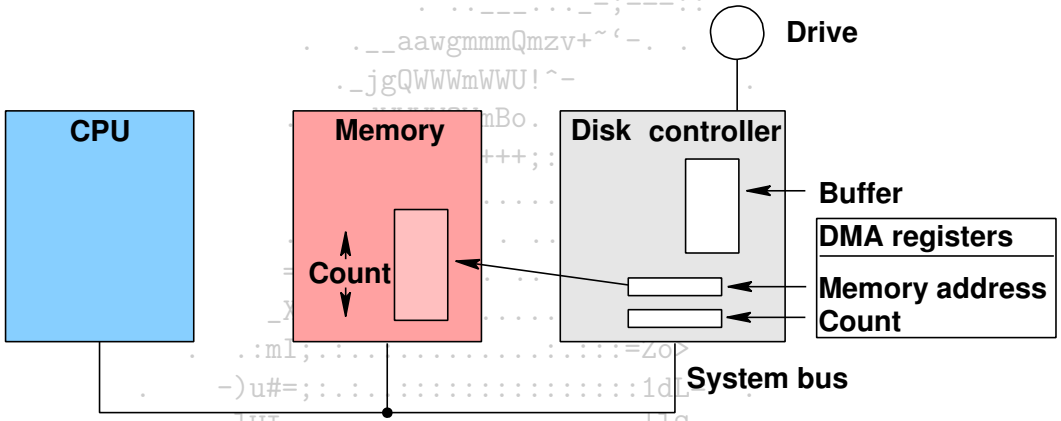


FIG. 7.2 – Transfert DMA

De nombreux contrôleurs de périphériques par blocs permettent un transfert DMA (Direct Memory Access) visant à épargner au CPU la phase de recopie du buffer du contrôleur vers la mémoire. Le transfert DMA s'organise de la façon suivante : le CPU réclame la lecture d'un disque au contrôleur qui en gère l'accès en lui passant, en plus des adresses sur le disque, les adresses mémoire où le bloc doit être placé. Le controleur copie bit à bit les secteurs correspondants au bloc dans son buffer, réalise un contrôle d'erreur et transfert, via le bus, les données vers l'adresse spécifiée. Au fur et à mesure, le compteur DMA est décrémenté du nombre de bits copiés. Lorsque le compteur atteint la valeur zero, le contrôleur génère une interruption, signalant au CPU qu'il peut lire le résultat des opérations et, en cas de succès, que le transfert est achevé.

Le transfert DMA s'avère intéressant dans le cas où le contrôleur est assez rapide pour effectuer l'opération de transfert. Sur une machine où le CPU est beaucoup plus rapide que le contrôleur, l'utilisation du transfert DMA est pénalisante par l'attente que le contrôleur engendre. De plus, sur certains systèmes (Linux...), la mise en œuvre du DMA nécessite des précautions supplémentaires : il faut s'assurer d'un bon interfaçage avec la mémoire virtuelle.

Le DMA n'est pas le seul moyen d'accélerer les I/O à partir de block devices tels que les disques. L'organisation des blocs au moment du formatage d'un disque peut permettre un gain de temps de lecture[1]...

[1] consulter [TW97] pour quelques détails sur l'entrelacement

7.2 I/O logiques

L'enrobage conceptuel qu'opère l'OS autour des périphériques d'I/O passe par la résolution d'une série de problèmes, d'abord proches des interfaces matérielles, puis de plus en plus éloignés au fur et à mesure de la "virtualisation" des périphériques.

Un des objectifs majeur de cette "virtualisation" est la construction d'une interface déconnectée des particularités matérielles des périphériques d'I/O. Imaginons un programmeur devant écrire une routine visant à copier un fichier vers un autre, le premier se trouvant sur un périphérique arbitraire ainsi que le second. Sans virtualisation, il faut construire un programme devant s'adapter, pour la lecture du premier fichier, aux paramètres du premier périphérique (périphérique bloc ou caractères, paramètres associés tels que la taille des blocs pour le premier cas..., système de fichiers...) et faire de même pour l'écriture du second. La virtualisation permet à des commandes telles que les suivantes de rester indépendantes des détails précédents.

```
denis@proxy710 > cp /cdrom/sources/SysEx.pdf ~/pub/OSs/cours/SysEx.pdf
```

```
C:\Documents and Settings\Denis CLOT\Bureau> copy SysEx.pdf F:\SysEx.cop.pdf
```

Un problème proche de cette indépendance est celui de la mise en place d'un mécanisme uniforme de dénomination de fichiers. C'est l'implémentation du système de fichiers qui y répond.

La gestion des erreurs d'I/O signalées par les contrôleurs est aussi un problème important. Lorsqu'une erreur est produite à la lecture d'un bloc, le contrôleur tente de la corriger. S'il n'y parvient pas, l'erreur est transmise au pilote du périphérique. Certaines erreurs de lecture sont souvent causées par une poussière et une opération de relecture suffit souvent à chasser l'impureté. Ainsi, en ordonnant une nouvelle lecture, le pilote de périphérique peut isoler les niveaux plus abstraits de ce genre d'erreur. Il est souhaitable que les couches logicielles les plus proches du matériel aient pour mission de filtrer le plus grand nombre d'erreurs pour protéger les niveaux d'abstraction supérieur.

Le transfert des données peut être définit comme synchrone ou asynchrone. Dans le premier cas, un transfert est bloquant jusqu'à son achèvement. Un programme nécessitant une lecture est bloqué jusqu'à ce que les données soient lues et prêtes pour lui. C'est un mode simple pour la programmation de routines. Le second mode est basé sur les interruptions : un programme nécessitant une lecture en fait la demande et attend une interruption lui indiquant que les données sont prêtes. Pendant l'attente, il peut passer à un autre traitement. Un OS peut définir le mode asynchrone par défaut et simuler un comportement bloquant pour les programmes utilisateurs...

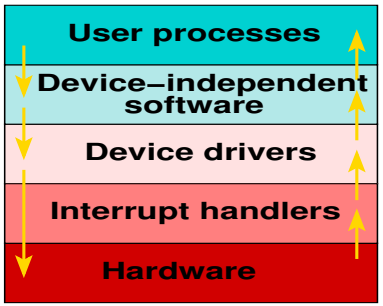


FIG. 7.3 – Couches logicielles

Le dernier problème est celui du partage des périphériques : certains périphériques (disques durs) peuvent être totalement partagés entre les utilisateurs pendant que d'autres (lecteurs de bande) sont reservés à un usage individuel exclusif.

Les solutions à ces différents problèmes peuvent conduire à une structuration logicielle des I/O logiques par couches. Comme l'illustre la figure 7.3, la gestion des interruptions est au plus près du niveau matériel, isolée par les couches supérieures des autres parties du système. La couche de pilotes, qui communique directement avec les périphériques, est la seule à connaître intimement leurs mécanismes. La couche qui lui est supérieure offre aux autres parties du système une interface libérée des spécificités matérielles ...

7.2.1 Gestion des interruptions

Les procédures de gestions d'interruptions sont placées au plus près du matériel, de façon à éviter le plus possible leur gestion aux autres parties du système. Elles constituent une sorte de première couche logicielle au dessous la couche matérielle. Cette couche peut être ignorée par la couche de pilotes lorsque le flux provient de cette dernière couche. Précisons que le schéma proposé représente les procédures de gestion d'interruptions matérielles. L'éventuelle gestion d'erreurs logicielles n'invervient pas à ce niveau.

7.2.2 Pilotes de périphériques

Les pilotes sont les entités qui placent les instructions dans les registres des contrôleurs et qui en vérifient le déroulement. Ils sont la seule partie de l'OS à connaitre le nombre de registres du controleur avec lequel il communique, à connaitre les paramètres physiques des périphériques (pistes, secteurs, cylindres, positionnement de têtes de lectures, facteur d'entrelacement...).

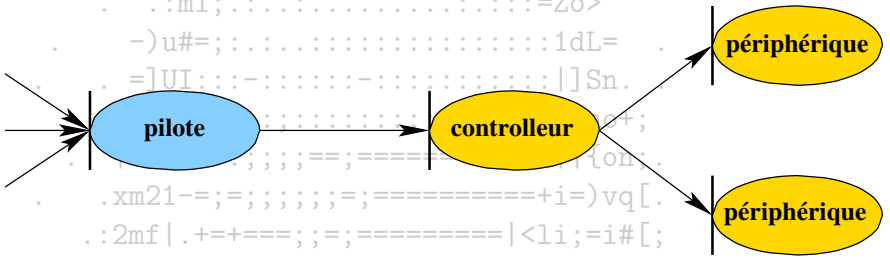


FIG. 7.4 – Abstraction d'un pilote de périphériques[2]

Le but d'un pilote est de fournir à la couche logicielle supérieure une interface de requêtage d'un périphérique dénuée de ses paramètres physiques. Il fournit les éléments définissant le périphérique virtuel associé.

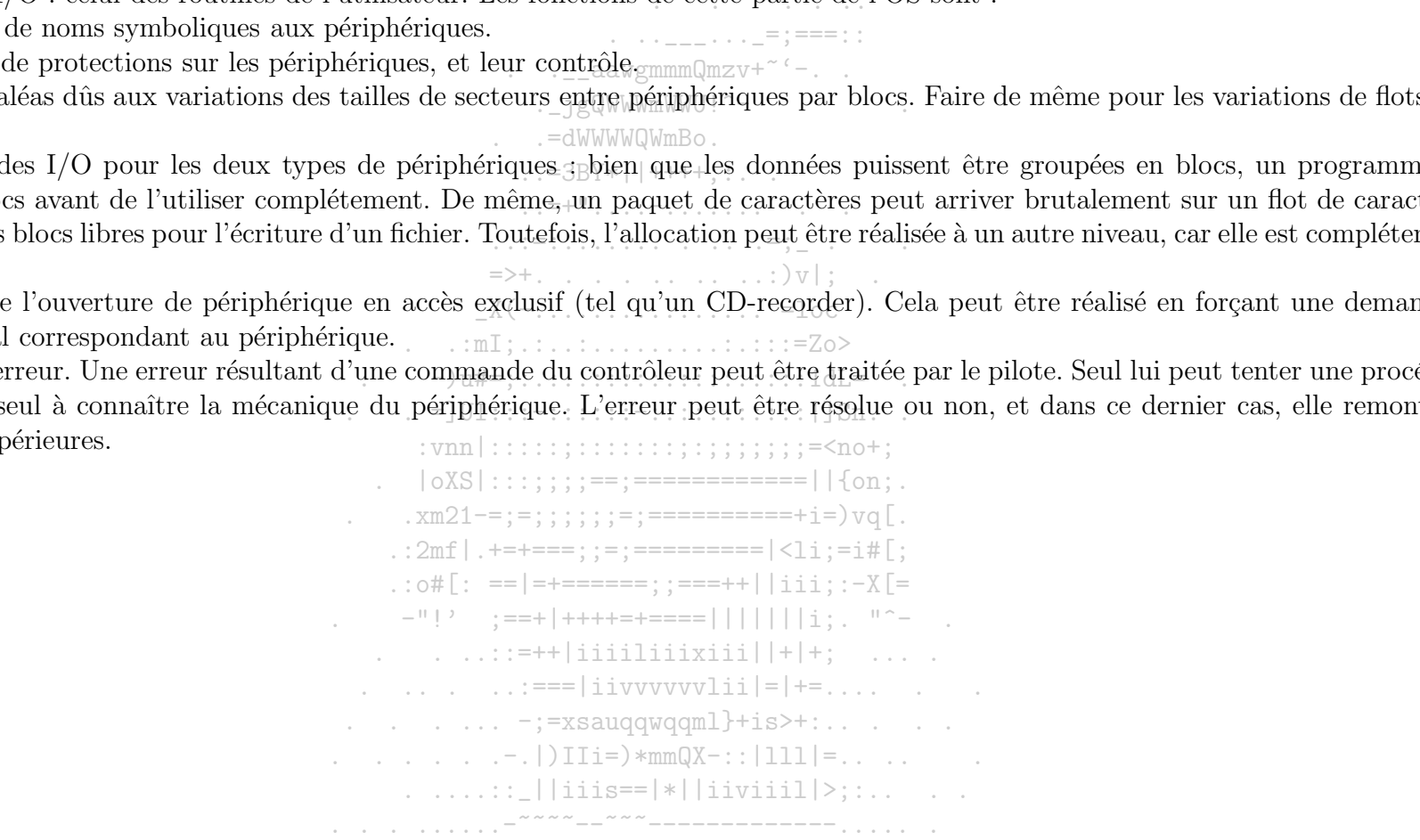
Lorsqu'un pilote traite une requête d'I/O, il doit déterminer l'adresse physique du bloc, vérifier les différents paramètres de l'état du périphérique afin de déterminer la séquence d'instructions qu'il doit soumettre au contrôleur. Les commandes sont passées au contrôleur et le pilote, dans la plupart des cas, se bloque jusqu'à l'aboutissement de ces commandes. Si aucune erreur n'est produite, les données sont transferées vers la couche logicielle supérieure et la prochaine est traitée ou attendue.

[2]Figure tirée de [FAS01]

7.2.3 Interface générale des I/O logiques

La troisième couche logicielle (device independent software), qui fournit une interface génerale des I/O logiques, rassemble les fonctions relatives aux I/O communes aux différents périphériques et indépendantes de leurs physiques, et propose une interface uniforme de ces fonctions pour le dernier niveau d'abstraction des I/O : celui des routines de l'utilisateur. Les fonctions de cette partie de l'OS sont :

- l'attribution de noms symboliques aux périphériques.
- la définition de protections sur les périphériques, et leur contrôle.
- masquer les aléas dûs aux variations des tailles de secteurs entre périphériques par blocs. Faire de même pour les variations de flots des périphériques caractères.
- le buffering des I/O pour les deux types de périphériques : bien que les données puissent être groupées en blocs, un programme peut traiter une partie de blocs avant de l'utiliser complétement. De même, un paquet de caractères peut arriver brutalement sur un flot de caractères.
- la gestion des blocs libres pour l'écriture d'un fichier. Toutefois, l'allocation peut être réalisée à un autre niveau, car elle est complétement indépendante du matériel.
- le contrôle de l'ouverture de périphérique en accès exclusif (tel qu'un CD-recorder). Cela peut être réalisé en forçant une demande d'ouverture du fichier spécial correspondant au périphérique.
- la gestion d'erreur. Une erreur résultant d'une commande du contrôleur peut être traitée par le pilote. Seul lui peut tenter une procédure de correction car il est le seul à connaître la mécanique du périphérique. L'erreur peut être résolue ou non, et dans ce dernier cas, elle remonte vers les couches logicielles supérieures.



7.2.4 Interface utilisateur pour les I/O logiques

Une partie de cet interface est faite des routines regroupées dans des librairies et basées sur les appels systèmes liés aux I/O. Par exemple, les bibliothèques standards du C permettent des opérations d'ouverture de fichiers, de lecture, d'écriture, qui sont de simples appels systèmes. Mais elles proposent également des fonctions plus élaborées sur lesquelles l'utilisateur peut reposer.

fd	=	*creat(name, mode)*	Obsolete way to create a new file
fd	=	*mknod(name,mode, addr)*	Create a regular, special, or directory i-node
fd	=	*open(file, how, ...)*	Open a file for reading, writing or both
s	=	*close(fd)*	Close an open file
n	=	*read(fd, buffer, nbytes)*	Read data from a file into a buffer
n	=	*write(fd, buffer, nbytes)*	Write data from a buffer into a file
pos	=	*lseek(fd, offset, whence)*	Move the file pointer
s	=	*stat(name, &buf)*	Get a file's status information
s	=	*fstat(fd, &buf)*	Get a file's status information
fd	=	*dup(fd)*	Allocate a new file descriptor for an open file
s	=	*pipe(&fd[0])*	Create a pipe
s	=	*ioctl(fd, request, argp)*	Perform special operations on a file
s	=	*access(name, amode)*	Check a file's accessibility
s	=	*rename(old, new)*	Give a file a new name
s	=	*fcntl(fd, cmd, ...)*	File locking and other operations

TAB. 7.1 – Quelques appels systèmes liès aux I/O conformes à POSIX

En plus de ces routines, des mécanismes particuliers visent à régler le partage de certains périphériques dans le contexte de la multiprogrammation. Le spooling est un de ces mécanismes. Il consiste à placer le fichier spécial d'un périphérique sous le contrôle d'un démon et ce démon est chargé d'envoyer vers le périphérique chaque fichier complet présent dans le répertoire de spool. De façon classique, le spooling intervient pour la gestion des impressions, l'envoi de mail...

Chapitre 8

Périphériques standards

Nous allons nous pencher ici sur les détails de quelques périphériques classiques des deux familles de périphériques : d'abord les disques (block devices), puis les terminaux (character devices). Nous finirons avec les horloges qui ne tombent pas vraiment dans l'une de ces familles.

8.1 Disques

8.1.1 Aspects matériels

Un périphérique de type disque est fait d'un empilement d'un ou plusieurs plateaux magnétiques lus par les têtes de lecture portées par un bras. Il y a autant de têtes que de faces de plateaux lues. Les données sont organisées sur des pistes concentriques, elles-mêmes divisées en secteurs. Dans le cas où plusieurs têtes de lecture existent, l'ensemble des pistes pointées par les têtes constitue un cylindre. Deux secteurs consécutifs sont séparés par un intervalle contenant des informations de synchronisation, de repérage et de validation du prochain secteur. Le temps de défilement de cet intervalle permet aussi aux têtes de lectures de commuter d'écriture en lecture et vice versa. C'est au formatage du disque que la taille des secteurs est fixée.

Dans les conceptions les plus simples, il y a autant de secteurs sur les pistes du centre d'un disque que sur les pistes proches du bord. La lecture des données est simple car cette structuration est très claire. Cependant elle engendre une perte de capacité du support car la densité de données décroit alors que l'on s'approche du bord extérieur du disque.

L'utilisation d'une densité homogène de données sur l'ensemble de support conduit bien évidemment à une augmentation du nombre de secteur des pistes alors que l'on s'éloigne du centre. Le repérage d'un secteur repose alors sur un référentiel plus complexe qu'un simple produit cartésien...Les disques modernes tels que les disques IDE sont ainsi construits. Cependant, l'electronique, qui se charge de la lecture d'un secteur, offre à l'OS une interface par laquelle les pistes comportent toutes le même nombre de secteurs. C'est l'électronique du disque qui en permet une gestion basée sur des paramètres

logiques, simplifiés par rapport aux paramètres physiques. Le contrôleur lui-même n'a pas à connaître la géométrie réelle du disque, mais simplement le nombre de têtes, de cylindres et de pistes.

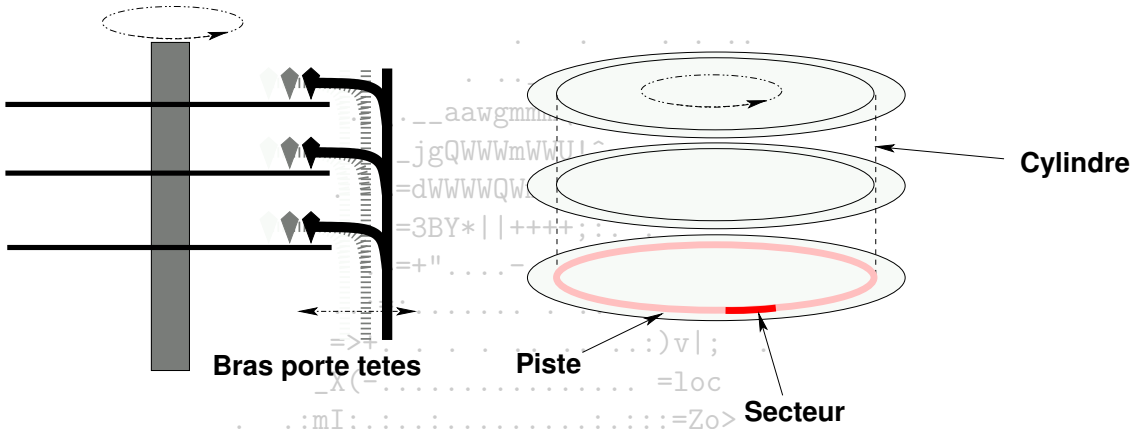


FIG. 8.1 – Mécanique et géométrie simplifiées d'un disque

L'électronique propre aux disques modernes permet ainsi d'intégrer dans leur mécanique un certain nombre d'opérations complexes qui sont déchargées des controleurs. Ces derniers peuvent véritablement communiquer avec le périphérique, lui soumettant une requête et attendant une réponse produite par l'électronique du disque. Ce mode allège l'occupation du controlleur, lui permettant de gérer plusieurs périphériques en alternance et réduisant les temps d'accès aux données.

Le dernier paramètre critique d'un disque est le temps moyen d'écriture et de lecture d'un bloc. Ce temps est fonction de trois paramètres :

- le temps moyen de recherche. C'est le temps nécessaire au bras portant les têtes de lecture pour se positionner sur le bon cylindre.
- le delai rotationel moyen ou temps moyen de latence. C'est le temps nécessaire pour que le secteur souhaité se trouve sous une tête de lecture.
- le temps moyen de transfert réel de l'information.

Les deux derniers paramètres sont intrinsèques au matériel, ainsi que le premier mais en partie seulement. En effet, l'ordre des recherches de secteurs influe directement sur le temps moyen de recherche. Il y a là un vrai problème d'optimisation que nous allons étudier en considérant différentes politiques d'ordonnancement du disque. La raison pour laquelle il est intéressant de se pencher sur ce problème d'optimisation est que le temps moyen de recherche est en général plus important que les deux autres.

Pour être tout à fait rigoureux dans l'étude de ce problème, il faudrait également tenir compte du temps d'attente avant le début du traitement d'une demande de lecture/écriture. C'est un aspect qui n'est pas toujours pris en compte dans la littérature !...

8.1.2 Aspects logiciels

Nous allons considérer différentes politiques d'ordonnancement et comparer leurs performances. Cette mesure de performance se fera sur la différence (en valeur absolue) de pistes entre deux requêtes. Il est admis que le temps de recherche d'un cylindre est proportionnel à cette différence. Nous observerons également un facteur proportionnel au temps d'attente moyen passé dans la file : ce facteur sera le nombre de cylindres parcourus depuis l'arrivée dans la file d'attente jusqu'au positionnement du bras sur le secteur souhaité.

Nous comparerons les performances des différentes politiques sur la séquence suivante de requêtes, données dans leur ordre d'arrivée au pilote, sachant que le cylindre de départ porte le numéro 12 :

34 09 11 14 17 01 36 03 02

Ordonnancement FIFO

La gestion la plus simple du mouvement du bras du disque consiste à satisfaire les requêtes d'accès dans l'ordre où elles surviennent. Cette politique assure une parfaite équitabilité des requêtes. Bien entendu, cet algorithme donne généralement de mauvais temps de réponse. Les performances qui figurent dans la table 8.1 s'avèrent être les plus mauvaises par rapport à celles des autres politiques. Il est prouvé que cette politique est moins performante que les suivantes.

REQUÊTES EN ATTENTE	**départ**	**34**	**09**	**11**	**14**	**17**	**01**	**36**	**03**	**02**	**déplacement total**	**attente moyenne**
politique de déplacement du bras												
FIFO	**12**	**34**	**09**	**11**	**14**	**17**	**01**	**36**	**03**	**02**	**140**	**76**
PCTR	**12**	**11**	**09**	**14**	**17**	**03**	**02**	**01**	**34**	**36**	**62**	**25**
LOOK (vers le haut d'abord)	**12**	**14**	**17**	**34**	**36**	**11**	**09**	**03**	**02**	**01**	**59**	**37**
LOOK (vers le bas d'abord)	**12**	**11**	**09**	**03**	**02**	**01**	**14**	**17**	**34**	**36**	**46**	**19**
C–LOOK (vers le haut)	**12**	**14**	**17**	**34**	**36**	**01**	**02**	**03**	**09**	**11**	**69**	**42**
C–LOOK (vers le bas)	**12**	**11**	**09**	**03**	**02**	**01**	**36**	**34**	**17**	**14**	**68**	**29**

Tab. 8.1 – Exemple de performances de politiques classiques

Ordonnancement PCTR (Plus Court Temps de Recherche ou Shortest Seek First)

Ici, le pilote choisit la requête dont la piste est la plus proche de la position actuelle (Shortest Seek First, plus courte d'abord, variante du PCTE déjà étudié en 3.2.2). Mais les requêtes portant sur des cylindres éloignés peuvent être durablement différées si d'autres requêtes surviennent pendant le traitement de la liste : l'équitabilité peut souffrir de la réduction du temps de réponse.

En fait, les pistes au centre du disque sont mieux servies. Le temps moyen est faible, mais la variance est forte. Plus la charge (i.e. l'utilisation du disque) est forte, et plus le risque de famine pour les pistes excentrées augmente.

Ordonnancement de l'ascenseur ou par balayage (Look)

Pour éviter l'inconvénient de la politique précédente, on déplace la tête dans une direction donnée en traitant toutes les requêtes rencontrées, puis le sens du balayage s'inverse et on traite les requêtes rencontrées, etc...La table propose les performances obtenues pour les deux sens possibles de parcours du bras. Sur cet exemple, cet ordonnancement semble donner les meilleurs résultats. En définitive, le temps moyen de réponse est faible avec une faible variance. Cette politique convient lorsque les files d'attente sont moyennement chargées.

Ordonnancement par balayage circulaire (C-Look)

Une variante de l'ordonnancement par balayage a été proposée par T.J. TEOREY en 1972 (C-Look, C pour circulaire) : la dernière piste est considérée comme adjacente à la première. Lorsqu'elle est arrivée à une extrémité, la tête retourne immédiatement à l'autre extrémité (sans traiter de requête intermédiaire). Cette politique convient bien pour le service de files d'attente fortement chargées.

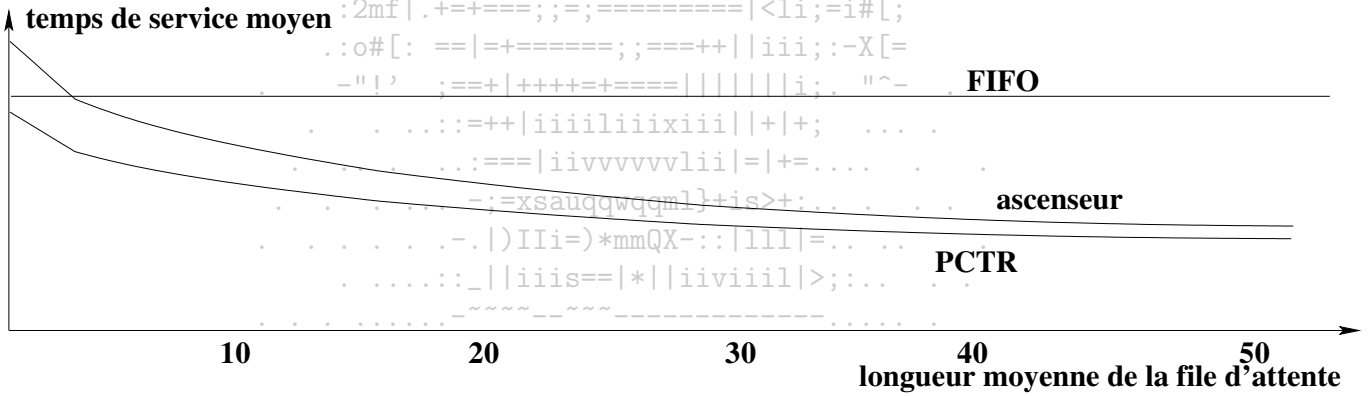


FIG. 8.2 – Performances de politiques classiques

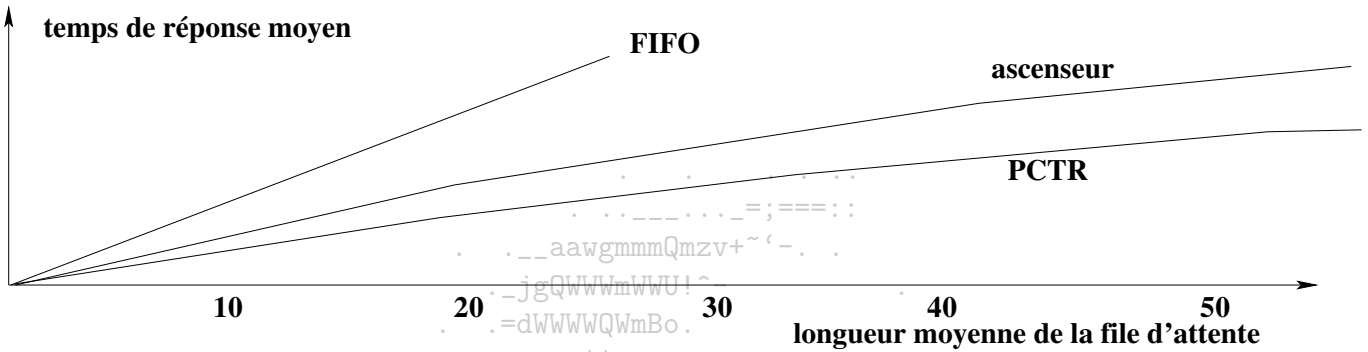


FIG. 8.3 – Temps de réponse (= temps de service + temps d'attente) pour des politiques classiques

Les figures 8.2 et 8.3 montrent qu'en moyenne et en fonction de la longueur de la file d'attente, c'est la politique PCTR qui donnent les meilleurs résultats. Il ne faut pas perdre de vu que cette politique expose à des risques de famine.

La politique suivante prend en considération plus d'hypothèses pour le positionnement du bras.

Ordonnancement PCTL (Plus Court Temps de Latence)

Lorsqu'un disque est fortement sollicité, il se trouve fréquemment plusieurs références à une même piste ou à un même cylindre. Les requêtes peuvent être ordonnées selon le secteur recherché pour réduire le temps de latence. L'algorithme PCTL (plus court temps de latence) traite les requêtes dans l'ordre de défilement des secteurs concernés sous la tête, pour une piste donnée, quel que soit leur ordre d'arrivée. Par exemple, si la tête se trouve au-dessus du secteur n° 2 et que des requêtes concernent les secteurs 11, 5, 8 et 7 d'une même piste, les secteurs 5, 7, 8 et 11 seront lus dans cet ordre, ce qui évitera d'attendre plus d'un tour pour traiter les secteurs 5, 8, 7. On peut parfois gagner en efficacité en entrelaçant les secteurs sur les pistes en fonction de la vitesse de rotation de disques.

Nous n'avons considéré que des politiques déterministes dans cette section. Toutefois, il faut savoir que les modèles probabilistes sont souvent mieux adaptés au cadre du problème car plus réalistes. Les modélisations par files d'attentes permettent de construire des politiques d'ordonnancement plus efficaces et dont les performances peuvent être estimées de façon plus précise (Consulter [BB93]).

8.1.3 Mémoire cache pour les pistes du disque

En général, le temps de recherche est très supérieur au temps de transfert. Il importe donc peu de lire un secteur ou bien une piste complète pour simplifier le fonctionnement du contrôleur. Pour tirer profit de ce fait, il est courant d'utiliser une mémoire cache, i.e. une mémoire à accès rapide, à accès

direct par le contrôleur et par l'UC, appartenant logiquement au disque et dans lequel le contenu de la piste est placé.

Lorsqu'un bloc est sollicité, on vérifie d'abord s'il se trouve dans le cache. S'il y est, sa lecture est immédiate. Dans le cas contraire, le bloc est chargé dans le cache à partir du disque. Lorsque le cache est plein, on retire un des blocs du cache en le recopiant sur disque s'il a été modifié. On peut adapter certains algorithmes de pagination.

Il est préférable d'implanter cette antémémoire dans le contrôleur, plutôt que dans le pilote, pour que le transfert puisse se faire par DMA.

8.1.4 Traitement des erreurs

Le fonctionnement des disques est soumis à de nombreuses sources d'erreurs :

- erreur de programmation (par exemple : accès à un secteur inexistant) nécessitant un arrêt de la requête.
- erreur du contrôle de parité (checksum) : on déclare le secteur endommagé si l'erreur persiste au bout de plusieurs essais. On tient à jour un fichier des secteurs endommagés à ne jamais allouer et à ne jamais copier lors d'une sauvegarde.
- erreur de positionnement du bras : un programme de recalibrage est lancé.
- erreur de contrôleur.

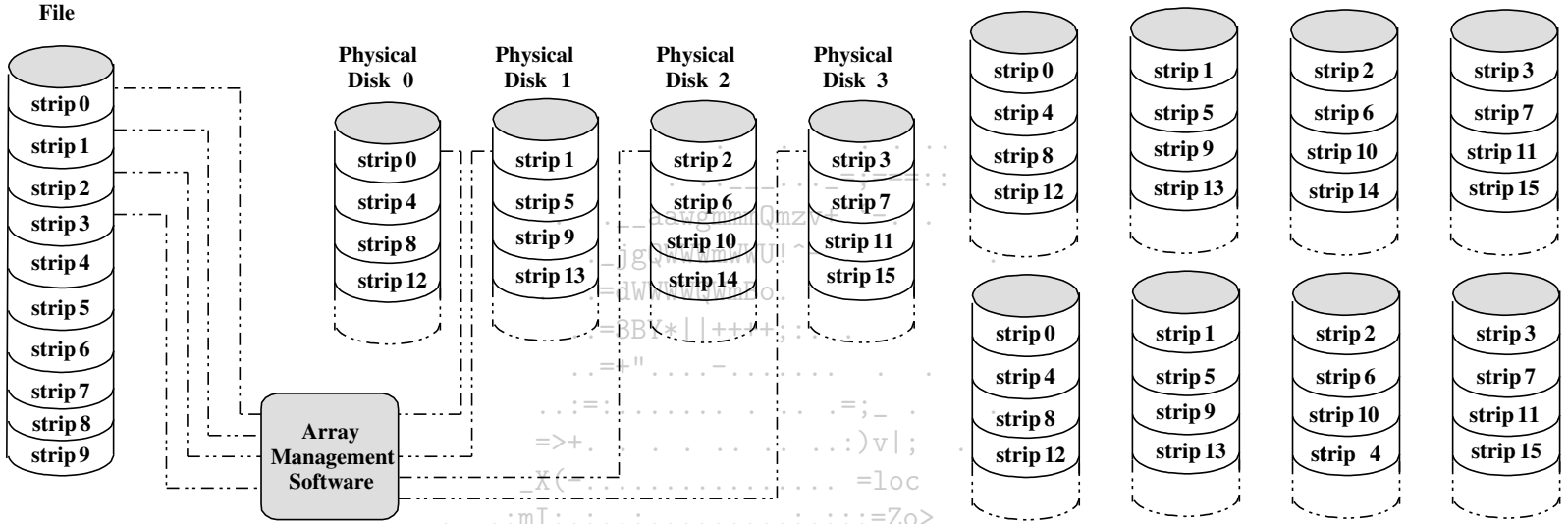
Les erreurs engendrées par la corruption d'un système de fichiers ne sont pas nécessairement des erreurs liés au fonctionnement des disques. Le traitement de ce type d'erreurs sera vu en 9.4.4.

La sauvegarde incrémentale, qui est une bonne parade pour de nombreux types d'erreurs, est une solution économique : selon une périodicité convenue (quotidienne, hebdomadaire), seuls les fichiers modifiés depuis la dernière sauvegarde sont sauvegardés. Les concepts RAID permettent la mise en place de solutions efficaces de protection des données.

RAID

La méthodologie RAID (Redundant Array of Inexpensive Disks) est constituée de différents niveaux de mise en place :

- Mode linéaire : plusieurs périphériques sont regroupés en un seul périphérique virtuel. Les périphériques sont regroupés en file, de sorte qu'un fichier très volumineux remplira d'abord le premier périphérique de la file, puis le second, etc. Ce niveau ne met pas en place de procédure de sauvegarde et n'améliore les performances d'I/O que pour des opérations menées en parallèles sur les périphériques.
- RAID 0 : appelé "stripe"mode (stripe=rayure), il utilise plusieurs disques (idéalement de même taille) pour fragmenter un fichier sur l'ensemble des disques (comme illustré par Fig. 8.4(a)). Comme le niveau précédent, il n'y a pas de procédure de sauvegarde, donc si un disque subit une panne, les données sont perdues. En revanche, les opérations de lecture et d'écriture sont menées en parallèle sur l'ensemble des disques, ce qui permet une amélioration du débit des I/O : si le bus reliant les disques est très rapide, le débit peut être multiplié par le nombre de disques.
- RAID 1 : ce mode utilise deux disques ou plus de tailles identiques. Le mirroir (i.e. la copie exact) d'un disque est mis en place sur tous les autres disques. Tant qu'un disque est valide, les données sont intactes. Les opérations d'écriture sont ralenties pour ce niveau car tous les disques sont

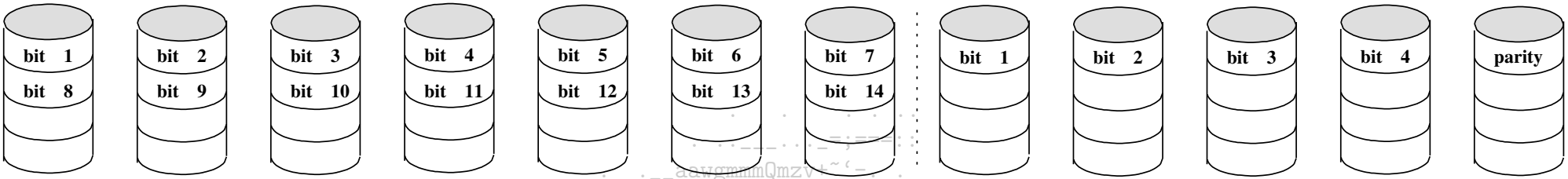


(a) RAID 0 (b) RAID 1+0

FIG. 8.4 – Illustration des niveaux 0 et 1 de RAID

sollicités. Dans le cas de nombre important de disques, des risques de saturations du bus peuvent apparaître. La mise en place d'un contrôleur RAID permet d'alléger les transferts sur le bus. Les opérations de lecture sont en revanche bonnes : le disque ayant la meilleure position pour la lecture d'un bloc donné est utilisé, minimisant le temps de recherche du bloc.

– RAID 2, 3 : La dispersion d'un fichier s'opère au niveau du bit. Les bits sont toutefois considérés par petits paquets afin de leur adjoindre soit un code correcteur d'erreur (RAID 2), soit un bit de parité (RAID 3). Le niveau 2 est peu utilisé en raison de la lourdeur de la gestion du contrôle d'erreur et de la nécessaire synchronisation de l'ensemble des disques. Le nombre de disques pouvant flancher sans entrainer de perte de données dépend des propriétés du code correcteur. Le niveau 3 qui est une simplification du niveau 2 permet de faire face au crash d'un disque unique. Il est lui aussi peu mis en place en pratique.
– RAID 4 : utilise trois disques ou plus. Comme dans les niveaux précédents, des bits de contrôle sont calculés mais le découpage des fichiers se fait par blocs. Supposons que n disques soient réservés à la réception de blocs des fichiers. Les n premiers blocs sont utilisés pour calculer un premier bloc de bits de contrôle. Ce bloc est le résultat de l'opération XOR appliquée aux n blocs. Il en va de même pour les blocs suivants. Ces informations pour la détection d'erreur et la reconstruction sont placées sur un disque spécifique (parity disk). En cas de défaillance d'un disque de données, son contenu



(a) RAID 2 (b) RAID 3

FIG. 8.5 – Illustration de RAID 2 et 3

est reconstitué. Le "parity disk" doit avoir des performances très supérieures à celles des disques stockant les données, car il est toujours utilisé pour maintenir à jour ses informations de contrôle : pour le changement d'un seul bloc, son ancienne valeur doit être lue, ainsi que celle du bloc de contrôle afin de recalculer le nouveau bloc, et les nouvelles valeurs doivent être écrites. Les performances sont en général bonnes pour les opérations de lecture et d'écriture lorsque qu'elles se prêtent à la parallélisation des requêtes, i.e. lorsque les demandes sont importantes. Une opération de lecture ou d'écriture prise individuellement ne présente pas de performance intéressante.

– RAID 5 : ce mode, très employé, utilise trois disques ou plus. Les éléments de contrôle sont dispersés sur les autres disques de façon homogène (round

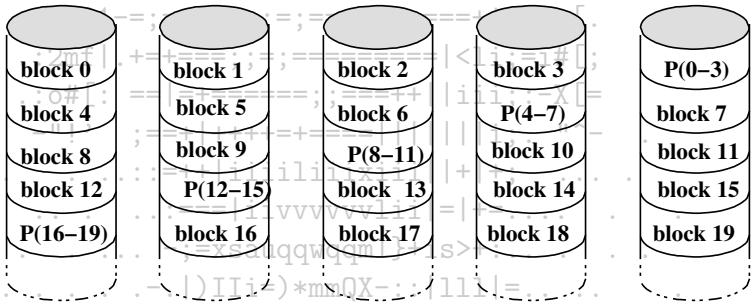


FIG. 8.6 – Illustration du niveau 5 de RAID

robin fashion) (Cg Fig. 8.6). Ainsi, données et bits de contrôle sont dispersés sur tous les disques, permettant de reconstituer le contenu de tout disque qui serait défaillant. Toutefois, si deux disques deviennent hors d'usage simultanément, toutes les données sont perdues. Les performances de lecture et d'écriture sont bonnes, comme au niveau précédent. Toutefois, si ce mode permet de remédier au problème du disque de contrôle du niveau

4, la procédure de récupération des données d'un disque est complexe.
D'autres niveaux existent...Le tableau suivant propose quelques applications pour les différents niveaux :

RAID level	0	1	2	3	4	5
Usage	Video editing and production	Financial, accounting, payroll	No commercial use	Image and video editing	General purpose	Web, databases or file server

8.1.5 Disques virtuels

Un disque virtuel (RAM disk) est un espace de la mémoire principale qui permet l'écriture et la lecture de blocs comme sur un disque classique, mais avec l'avantage d'un accès simplifié et immédiat. Un tel disque est utile pour le placement de programmes ou de données fréquemment utilisés. Pour un système implémentant le montage de système de fichier, un tel disque permet de placer la racine en mémoire afin de déconnecter l'OS du support qui l'initialise...

8.2 Terminaux

Les terminaux sont utilisés pour communiquer avec l'ordinateur. Leurs formes et leurs interfaces matérielles sont trés variées. Les pilotes cachent ces particularités aux autres parties du système et aux utilisateurs.

8.2.1 Aspects matériels

Trois types de terminaux sont distingués selon le mode de communication établi entre eux et l'OS (cf Fig.8.7) :
– les terminaux mappés en mémoire,
– les terminaux à interface RS-232,
– et les terminaux X.

Les terminaux du premier type (clavier, écran) sont directement branchés sur l'ordinateur. Ils s'interfacent via la mémoire RAM vidéo. Le contrôleur vidéo génère, à partir des caractères qu'il lit dans la mémoire RAM vidéo, le signal vidéo permettant d'afficher à l'écran les caractères lus. Sur un IBM PC, l'écran peut être utilisé selon différents modes : en mode caractère ou en mode bitmap. En mode caractère, l'unité d'information est le code ASCII du caractère avec un code de mise en forme (couleur, vidéo inversée, caractère clignotant...). La définition d'un écran de 25 lignes comportant chacune 80 caractères nécessite 4000 bytes de mémoire vidéo. En mode bitmap, l'unité d'information est le pixel. Un écran 768×1024 avec un affichage couleur codant en 24 bit recquiert 2Mb de mémoire vidéo.

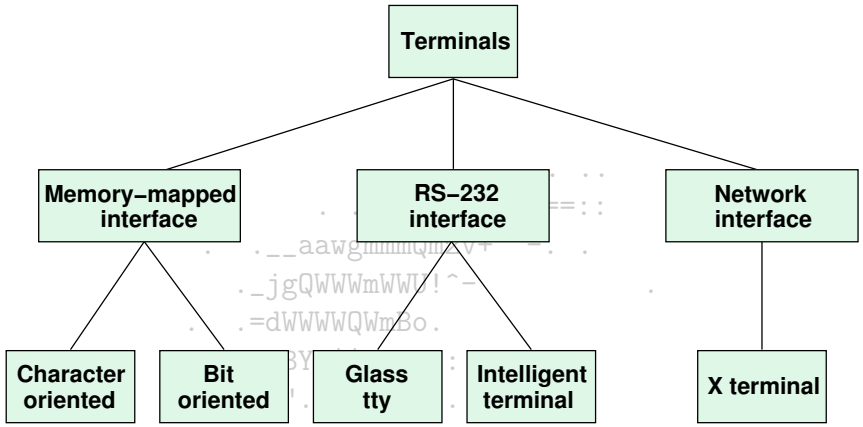
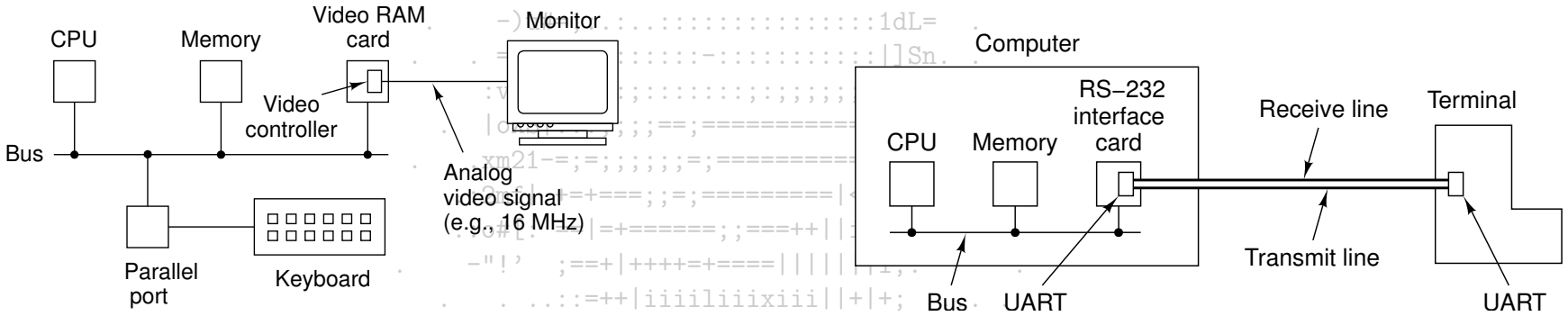


FIG. 8.7 – Types de terminaux



(a) Ecran mappé en mémoire

(b) Terminal RS-232

FIG. 8.8 – Terminaux

Avec un écran mappé en mémoire, le clavier est découplé de l'écran. Son interface passe par un port parallèle ou série et l'envoi d'un caractère passe par un mécanisme d'interruption (le CPU passe la main au pilote du clavier qui lit le code placé dans un port d'I/O). Précisons que le signal envoyé par le biais d'un clavier n'est pas le code du caractère à afficher. C'est un signal qui est interprété par le pilote du clavier.

Les terminaux à interface RS-232 possèdent un clavier et un écran. Ils utilisent une interface série pour communiquer avec la machine distante. Pour la transmission des caractères via la ligne série, une conversion est effectuée pour échanger bit à bit l'information. Cette convertion est assuré par un composant, l'UART (Universal Asynchronous Receiver Transmitter), présent sur le terminal et sur la machine. La ligne série est en général fournie par une ligne téléphonique ou un modem. Cette catégorie de terminal est délaissée au profit des PCs et des terminaux X.

La dernière catégorie de terminaux regroupe les terminaux les plus évolués. Ces derniers sont pourvus d'un CPU, de mémoire, d'un clavier, d'une souris et d'un écran. Les plus répandus sont les terminaux X. La communication avec la machine distante s'effectue via Ethernet.

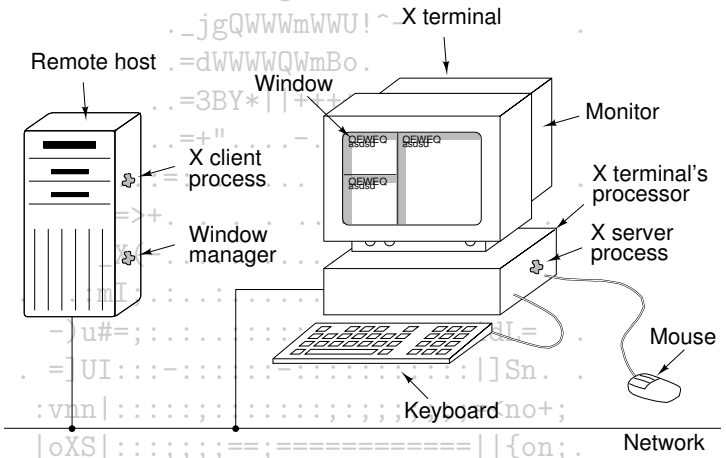


FIG. 8.9 – Terminal X

Un terminal X utilise généralement le système de fenêtrage X du MIT. Le programme collectant les inputs issus du clavier, de la souris et de machines distantes, est le serveur X. Ce serveur communique via le réseau avec des client X et a pour principale fonction de gérer l'affichage découlant des différentes sources d'input. Il peut paraitre surprenant que le serveur X réside dans le terminal et que les clients soient sur les machines distantes, mais il est beaucoup plus léger, en terme de communication, de gérer au sein du terminal son affichage.

8.2.2 Aspects logiciels

Gestion de l'input

La tâche d'un pilote de clavier consiste à collecter l'input généré par le clavier et le transmettre au programme lisant à partir de ce terminal. Deux modes de filtrage de cet input sont possibles :

– mode caractère (raw mode, rebaptisé non canonical mode par POSIX)

– mode ligne (cooked mode, rebaptisé canonical mode par POSIX)

Supposons qu'un utilisateur saisisse *dste*, puis constatant son erreur, corrige avec trois retours arrière et *ate*, pour obtenir la commande *date*, puis valide par un retour chariot. En mode canonique, le programme utilisateur obtiendra cinq codes ASCII correspondant à la commande *date* et le retour chariot. En mode non canonique, le programme utilisateur recevra les onze caractères ASCII. Sous Unix, la fonction ioctl permet de changer le mode d'un périphérique. Il est également possible d'utiliser un mode intermédiaire dans lequel DEL, ^S, ^Q et ^\sont interprétés comme en mode canonique.

Une fois que le signal du clavier a été lu par le pilote, il doit être (éventuellement) converti en code ASCII. La plupart des OS permettent de préciser la correspondance entre les signaux et les codes de caractères afin de satisfaire les spécifités de l'alphabet de la langue native de l'utilisateur.

Le pilote est également chargé de placer dans un tampon les caractères saisis (pour les deux modes). Il existe deux politiques de gestion des tampons associés aux terminaux :

– le pilote réserve un ensemble de tampons de taille standard. A chaque terminal est associé un pointeur sur le premier tampon utilisé ; l'ensemble des tampons associés à un terminal est organisé en une liste chaînée. Cette méthode est utilisée sur les grosses machines possédant beaucoup de terminaux
– au sein de la structure de données associée à chaque terminal dans la mémoire, on implante un tampon propre au terminal. Le pilote est plus simple. Cette solution convient bien aux petits systèmes ou aux PC.

Les pilotes de claviers doivent également s'acquitter de bien d'autres tâches : gérer la présence ou l'absence d'écho à l'écran, gérer les lignes de longueur supérieure à une ligne d'écran, gérer les caractères de tabulation, produire les caractères RC et NL lors d'une validation avec une convention donnée, gérer les caractères spéciaux (CTRL-D, CTRL-Q, CTRL-S, DEL, etc). Sous UNIX, ces caractères peuvent être redéfinis par l'utilisateur (association d'un caractère ou ensemble de caractères à un nom symbolique).

Gestion de l'ouput

Dans le cas d'un terminal RS-232, le programme ou la fonction d'écho de la frappe envoie dans le tampon de l'écran la suite des octets à afficher. Si le tampon est plein ou bien si toutes les données sont transmises, le premier caractère est envoyé au terminal, le pilote est endormi. Une interruption provoque la transmission du caractère suivant, etc.

Dans le cas de terminaux mappés en mémoire, les caractères à afficher sont retirés un à un de l'espace utilisateur et placés dans la RAM vidéo (avec traitement particulier pour certains caractères, comme BELL ou les séquences classiques d'échappement).

Le pilote gère la position courante dans la RAM vidéo, en tenant compte des caractères tels que RC, NL, BS ; il gère aussi le défilement (scrolling) lorsqu'un passage à la ligne en bas d'écran survient. Pour cela, il met à jour un pointeur, géré par le contrôleur vidéo, sur le premier caractère de la première ligne à l'écran. Le pilote gère aussi le curseur en tenant à jour un pointeur sur sa position.

8.3 Horloges

Les horloges (clocks ou timers) sont indispensables au fonctionnement d'un sytème à temps partagé. Ces composants particuliers ne tombent ni dans la catégorie des block devices, ni dans celle des character devices.

8.3.1 Aspects matériels

Deux types d'horloges se retrouvent dans les ordinateurs :

- des horloges simples, fonctionnant sur l'alimentation et générant une interruption à chaque phase, à la fréquence de 50∼60Hz.
- des horloges programmables composées d'un oscillateur à quartz, d'un compteur et d'un registre. Le compteur est décrémenté à chaque pulsation du cristal, et lorsque le compteur veut zéro, une interruption est générée. La grande précision d'un tel oscillateur permet de générer des interruptions à des fréquences comprises entre 5 et 100MHz. Ces horloges peuvent fonctionner selon différents modes. Tout d'abord, en mode simple où, aprés une copie de la valeur placée dans le registre vers le compteur, la valeur du compteur est décrémentée jusqu'à la valeur zéro pour laquelle se produit l'interruption. En mode périodique, après l'émission de l'interruption, la valeur du registre est à nouveau copiée dans le compteur et le processus recommence, permettant l'émission de signaux périodiques, appélés tics d'horloge.

Le grand avantage des horloges programmables est que leur fréquence d'interruption peut être réglée de façon logicielle.

Aux horloges programmables d'un système s'ajoutent en général une horloge à pile, utilisée pour conserver l'heure et la date courantes lorsque la machine est éteinte. Ce n'est pas le seul moyen pour les connaitre : des protocoles réseaux permettent de récuperer cela sur un hôte distant, mais de façon plus simple, l'utilisateur peut être amené à les saisir alors que le système le réclame. Dans le système, ce pointeur temporel est stocké comme étant le nombre de tics d'horloge depuis le premier janvier de l'année 1970 à 12 :00 AM à l'heure de Londres (GMT i.e Greenwich Mean Time ou UTC i.e Universal Coordinated Time).

8.3.2 Aspects logiciels

Les horloges permettent l'émission d'interruptions à des fréquences régulières. Toutes les opérations intervenant sur la dimension temporelle (autres que celle décrites précédemment) doivent être assurées de façon logicielle :

- la maintenance de l'heure, de la date
- le contrôle de l'utilisation du CPU par un processus
- le comptabilisation du temps CPU pour un processus
- la gestion des alarmes programmées par les utilisateurs
- la gestion des alarmes systèmes
- la récolte de données de statistique, de surveillance du système.

Voyons comment certaines des fonctions précédentes peuvent être remplies.

Maintenance de l'heure

Cette fonction est simple à remplir car il suffit d'incrémenter un compteur à chaque tic d'horloge. La chose à considérer est le risque de débordement du compteur : pour une horloge à 60Hz, le nombre de tics dans une journée est de $60 \times 60 \times 60 \times 24$ et dans une année, ce nombre est 1892160000. Un compteur 32 bits permet de distinguer 2^32 valeurs, or $2 < 2\hat{}32/1892160000 < 3$. A l'évidence, un tel compteur ne peut servir à comptabiliser le nombre de tics depuis 1970.

Une solution consiste à utiliser un compteur 64 bits, ce qui permet d'être tranquille pendant quelques milions d'années. Toutefois, cette solution est coûteuse en gestion de compteur car celui-ci est mis à jour de nombreuse fois par secondes. Une économie peut être réalisée en descendant la précision de l'heure au niveau de la seconde. Un compteur 32 bits peut alors servir pendant plus de 136 années sans être débordé. Ceci est réalisé à l'aide d'un horloge programmable qui provoque une interruption toute les secondes. Une dernière solution consiste à utiliser la date de démarrage du système et à compter les tics d'horloge. Lorsque la date est réclamée, les tics d'horloge comptabilisés et la date de boot sont ajoutés pour former la date.

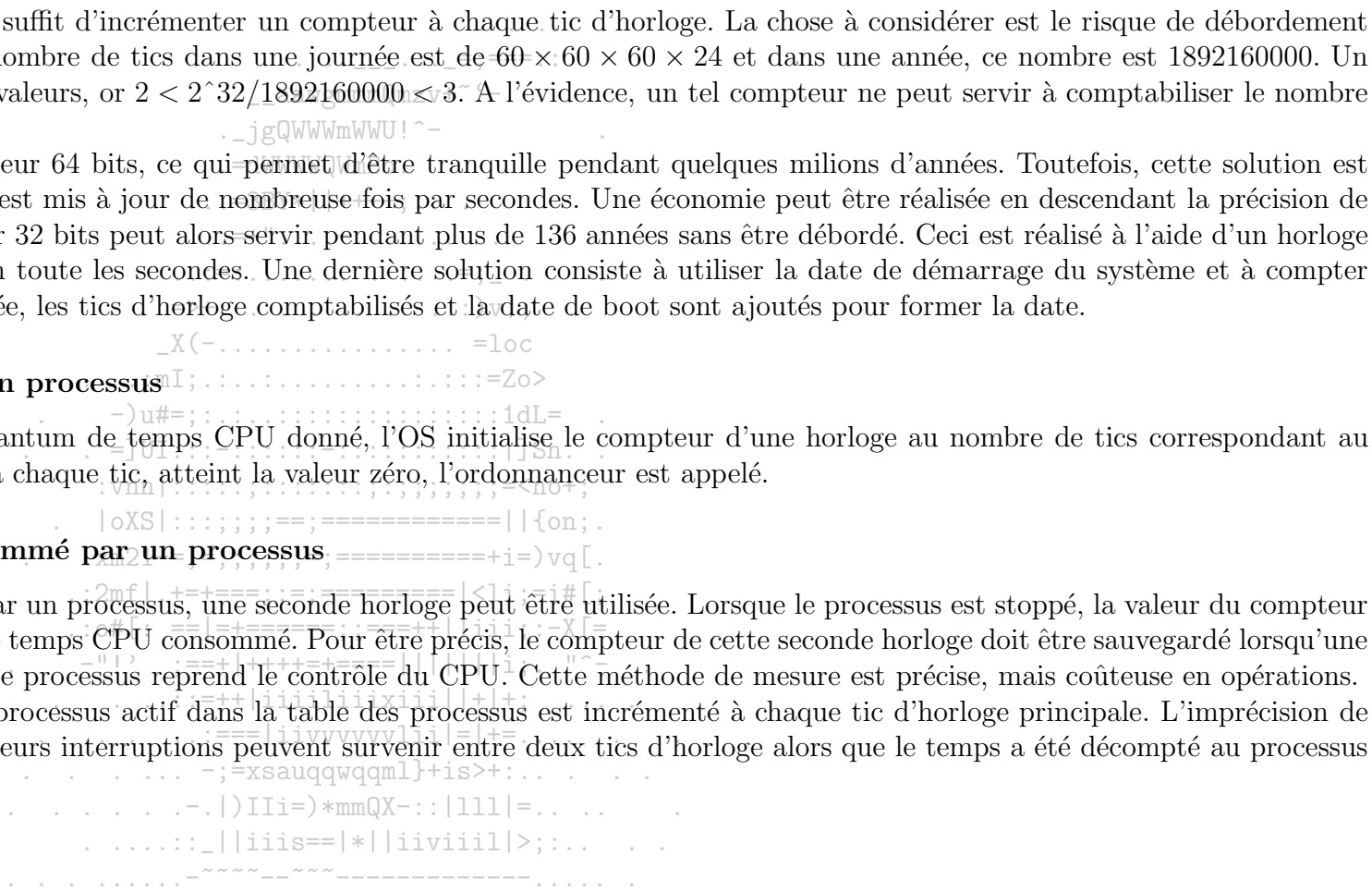
Contrôle de l'utilisation du CPU par un processus

Lorsqu'un processus démarre avec un quantum de temps CPU donné, l'OS initialise le compteur d'une horloge au nombre de tics correspondant au quantum. Lorsque le compteur, décrémenté à chaque tic, atteint la valeur zéro, l'ordonnanceur est appelé.

Comptabilisation du temps CPU consommé par un processus

Pour mesurer le temps CPU consommé par un processus, une seconde horloge peut être utilisée. Lorsque le processus est stoppé, la valeur du compteur de la seconde horloge est lue pour connaitre le temps CPU consommé. Pour être précis, le compteur de cette seconde horloge doit être sauvegardé lorsqu'une interruption se produit, puis restoré lorsque le processus reprend le contrôle du CPU. Cette méthode de mesure est précise, mais coûteuse en opérations.

En général, un champ correspondant au processus actif dans la table des processus est incrémenté à chaque tic d'horloge principale. L'imprécision de cette mesure provient du fait qu'une ou plusieurs interruptions peuvent survenir entre deux tics d'horloge alors que le temps a été décompté au processus qui était actif au dernier tic d'horloge.



Troisième partie

Systèmes de fichiers

Chapitre 9

Les systèmes de fichiers

9.1 Système de fichiers

Le fichier est le concept de base sur lequel repose tout le mécanisme de stockage d'information de façon durable sur un ordinateur. Un fichier regroupe des informations sur un espace logique contigu. Il permet notamment de :

- conserver des données produites par un programme au delà de son existence.
- partager des informations entre plusieurs programmes.
- traiter par un programme un volume de données supérieur à la mémoire centrale disponible.

L'utilisation du concept de fichier nécessite une implémentation faisant face aux problèmes d'organisations (logique et physique) et de gestion des fichiers. L'organisation logique repose en grande partie sur le concept de répertoire. Un système de fichiers (FS pour file system) est une façon de définir cette organisation et cette gestion. Il fournit :

- d'une part une implémentation des concepts de fichiers et de répertoires
- d'autre part une interface permettant une déconnexion entre les concepts et la réalité physique de leur mise en branle. L'utilisateur est déconnecté des caractéristiques des supports physiques de stockage. L'interface définit les opérations applicables aux fichiers et aux répertoires. L'utilisateur manipule donc des fichiers logiques.

C'est à l'OS de faire le lien entre un fichier physique (qui est la représentation physique de l'information) et le fichier logique (qui est un identificateur désignant à la fois le fichier physique et la représentation logique faite des informations portant sur le fichier). L'existence d'un fichier pour l'OS est d'ailleurs conditionnée par la vérification des deux conditions suivantes :

- il existe une représentation physique (sur le support) de l'information contenue par le fichier.
- il existe un identificateur désignant le fichier relié à des informations permettant d'accéder à sa représentation physique.

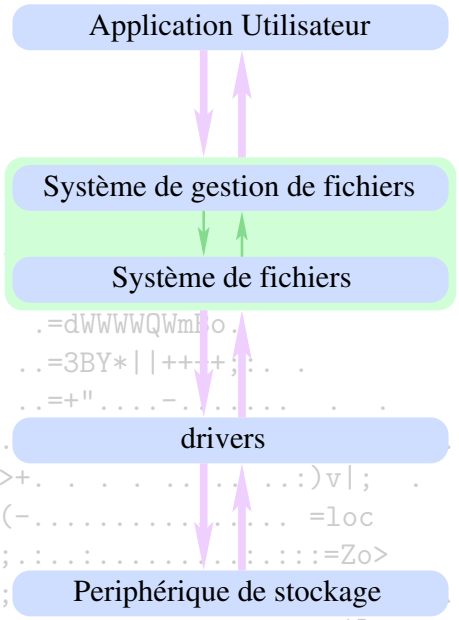


FIG. 9.1 – Du fichier logique vers le fichier physique, and back...

Nous étudierons dans ce chapitre l'aspect logique puis physique des fichiers en général, puis nous aborderons quelques problèmes courants en gestion de système de fichiers tels que le contrôle de la cohérence d'un système, ses performances...L'étude, dans le chapitre suivant, de quelques systèmes de fichiers précis permettra d'affiner l'aspect logique implémenté d'un fichier.

9.2 Logique des fichiers

9.2.1 Fichiers

Définition

Un fichier est un espace contigu d'adresses logiques vers lequel il est possible d'écrire ou à partir duquel il est possible de lire. La façon dont le contenu de ce fichier abstrait est lié au support physique auquel il est attaché ne nous intéresse pas ici et n'est pas (à priori) le souci de l'utilisateur car les opérations mises à sa disposition l'en dispensent.

Nom

Le nom d'un fichier est la chaîne de caractères par laquelle il est référencé. Ce nom est une référence et ne fait absolument pas partie du fichier. Selon le système de fichiers, le modèle de cette chaîne varie.

Type

Dans la plupart des OSs, des types de fichiers sont distingués :

- des fichiers normaux (*regular file*) contenant de données, des codes de programmes. Ces fichiers sont en général des fichiers ASCII ou des fichiers binaires.
- des fichiers systèmes utilisés par l'OS pour ses besoins. Un répertoire est un fichier système utilisé pour la gestion de la structure des fichiers. Certains OSs comme Unix modélisent des I/O vers des périphériques spéciaux (autre que des disques) par le biais de fichiers spéciaux pour s'appuyer sur les opérations déjà définies sur les fichiers dans un but de simplification. Sur un fichier spécial, les opérations définies dépendent de la finalité dudit fichier : si le périphérique est une imprimante, le fichier lu ne peut être «rembobiné»...

Pour cette partie, nous nous intéressons aux éléments fondamentaux d'un système de fichiers, i.e. aux fichiers normaux et aux répertoires.

Attributs

En plus de son nom et des données qu'il contient, un fichier a un ensemble d'attributs. Ces attributs dépendent des concepts implémentés pour la gestion d'un fichier. Les attributs standards sont la localisation du fichier, sa taille, des dates de création, de dernier accès, un propriétaire.... Si des mécanismes de protection sont définis, les droits de lecture, écriture et exécution peuvent être spécifiés. Si différentes structures de fichiers sont supportées par l'OS, la structure peut être précisée.

Opérations

Les variations concernant les opérations implémentées sur les fichiers sont grandes sur l'ensemble des systèmes de fichiers. Le jeu d'instructions définit la ou les structures logiques dont les utilisateurs disposent pour organiser les informations dans un fichier et y accéder. Un fichier pourrait être simplement considéré comme une suite de bits ou bien être vu et construit comme une juxtaposition d'articles de tailles fixes, eux mêmes pouvant être construits selon une certaine structure.

Les opérations sont de deux types : celles opérant au niveau du fichier, et les autres intervenant au niveau de l'information contenue dans le fichier :

- Les opérations élémentaires de manipulation de fichiers sont la création d'un fichier, l'ouverture d'un fichier, la fermeture, la destruction, la lecture et l'écriture.

– Si le système de fichier le permet, le fichier peut être structuré comme une suite d'articles placés consécutivement et des opérations peuvent être définies à ce niveau. Les opérations peuvent alors être la lecture d'un article, l'écriture, la modification, l'insertion et la destruction d'un article.

Le mode par défaut d'accès à l'information d'un fichier est l'accès séquentiel consistant à lire le fichier en commençant par le début, puis en poursuivant la lecture jusqu'à la fin du fichier. L'introduction de structures logiques de fichiers permet souvent d'accompagner l'accès séquentiel d'un mode d'accès indexé[1] sur les articles d'un fichier.

Dans le mode séquentiel, le fichier est lu article par article selon l'ordre d'énumération du fichier : les opérations d'accès ne peuvent utiliser les numéros d'ordre des articles. Par exemple, lorsqu'une lecture commence, un pointeur placé en début de fichier permet la lecture du premier article. Après la lecture du premier article, l'article pointé est le suivant. Etc. Ce type d'accès est incontournable lorsque le support est une bande !

En revanche, dans un accès indexé, il est possible de se positionner sur un article dont le numéro a été fourni à l'opération. L'intérêt est d'autant plus évident si nous considérons l'économie réalisée, par rapport au mode d'accès séquentiel, pour la réalisation d'opérations de lecture et de mise à jour contrôlées sur un petit nombre d'articles.

Sur la plupart des OSs modernes, les fichiers peuvent être accédés en mode indexé. Ceci est principalement dû à la généralisation des disques comme support de stockage.

9.2.2 Répertoires

De façon naïve, un répertoire peut être présentée comme une entité logique rassemblant des fichiers (et selon l'implémentation, des sous-répertoires). Comme illustré ci-dessous, les différentes implémentations du concept de répertoire montrent son rôle dans la structuration des fichiers. Initialement, un

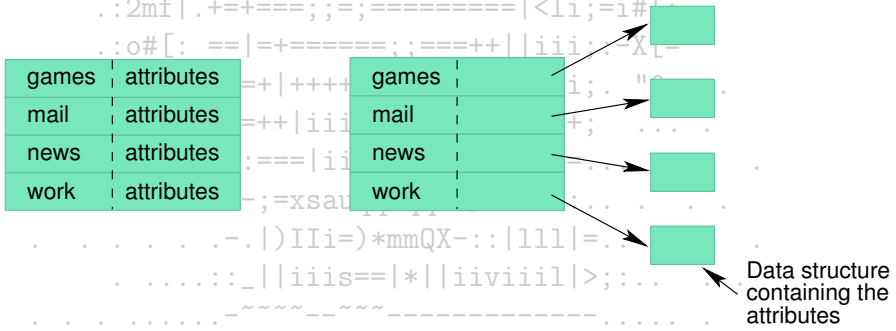


Fig. 9.2 – Structuration des méta-données des fichiers

[1] Le mode d'accès indexé est également dénommé mode d'accès aléatoire.

répertoire était un fichier spécial comportant autant d'"entrées" qu'il contient de fichiers. Chaque entrée contient le nom d'un fichier et d'autres informations (attributs, adresse physique...). Ces informations, en dehors du nom, peuvent être stockées dans une structure de données à part et être simplement pointées dans l'entrée (comme illustré Fig.9.2). Lorsqu'une opération d'ouverture sur un fichier du répertoire est demandée, l'entrée correspondant au fichier est recherchée dans le fichier spécial constituant le répertoire.

Ainsi le but premier d'un répertoire était de permettre un accès rapide aux différents fichiers. L'évolution de l'implémentation des répertoires a été guidée par deux autres objectifs : lever les limitations portant sur les noms pouvant être donnés aux fichiers et permettre à l'utilisateur d'ordonner ses fichiers à sa convenance. Nous verrons que dans certains OSs, un répertoire peut aussi être considéré comme un nœud de l'arborescence des fichiers sur lequel des opérations de montage peuvent être faites.

Structure à un niveau

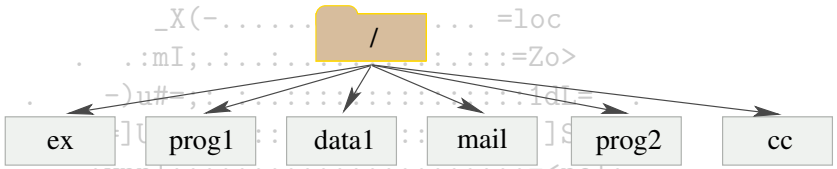


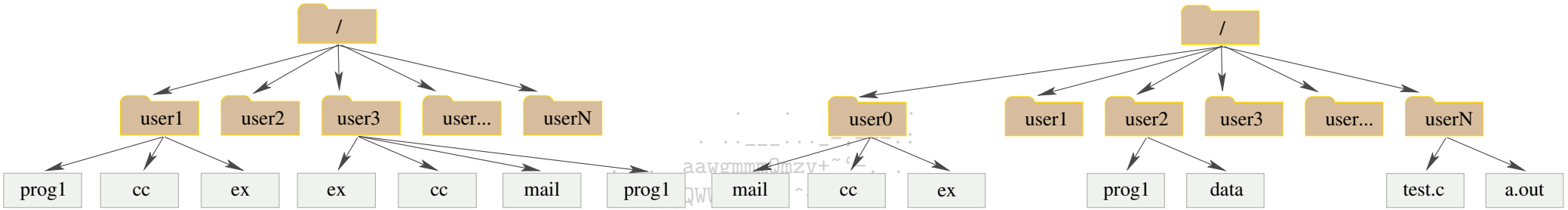
FIG. 9.3 – Système de fichiers plat

La structuration des fichiers la plus pauvre possible est la structure à un niveau unique. Les limitations rencontrées dans l'usage sont fortes : limitation du nombre de fichiers, cohabitation entre plusieurs utilisateurs difficiles, dénommination des fichiers...

Structure à deux niveaux

La structuration à deux niveaux facilite la cohabitation de plusieurs utilisateurs. Le répertoire racine dispose d'autant d'entrées qu'il y a utilisateurs. Ces entrées correspondent aux répertoires des utilisateurs. Un telle structure est en général assortie d'un mécanisme de protection permettant à chaque usager de protéger ses données du regard et des opérations des autres utilisateurs.

Un tel système présente des limitations par rapport au partage de données. Ceci est particulièrement perceptible au niveau des utilitaires. Chaque utilisateur doit avoir sa propre copie de l'exécutable de chaque commande qu'il souhaite utiliser : compilateur, éditeur de texte, client de messagerie...Afin de limiter la redondance engendrée par ces recopies inutiles des utilitaires, un répertoire regroupant les programmes communs peut être créé.



(a) Système basique de fichiers à deux niveaux

(b) Solution pour la partage des utilitaires contournant la réplication systématique...

FIG. 9.4 – Système de fichiers à deux niveaux

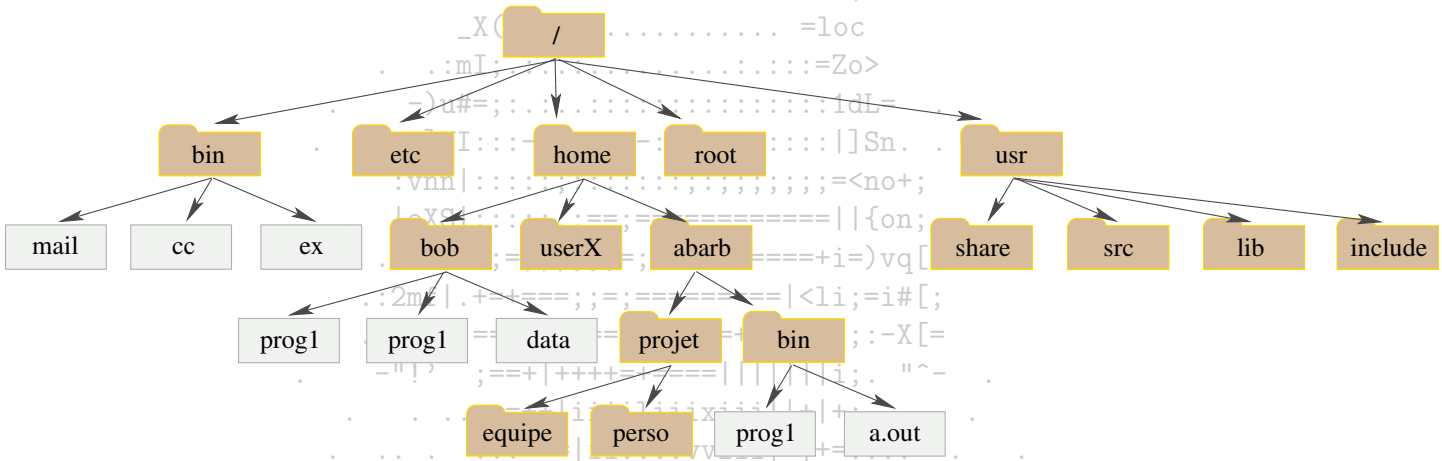


FIG. 9.5 – Système de fichiers avec une structure d'arbre simple

Structure d'arbre

Dans un tel système de fichiers, de nouvelles opérations apparaissent : des opérations de manipulation de répertoires et de fichiers. Ces opérations sont accessibles aux utilisateurs (par défaut), et la hauteur de l'arbre n'est à priori pas limitée. Des opérations de positionnement par rapport à la structure sont également introduites et la désignation d'un fichier peut se faire de façon relative (par rapport au répertoire courant) ou de façon absolue (avec le

nom complet en partant de la racine de l'arbre).

Structure de graphe sans cycle

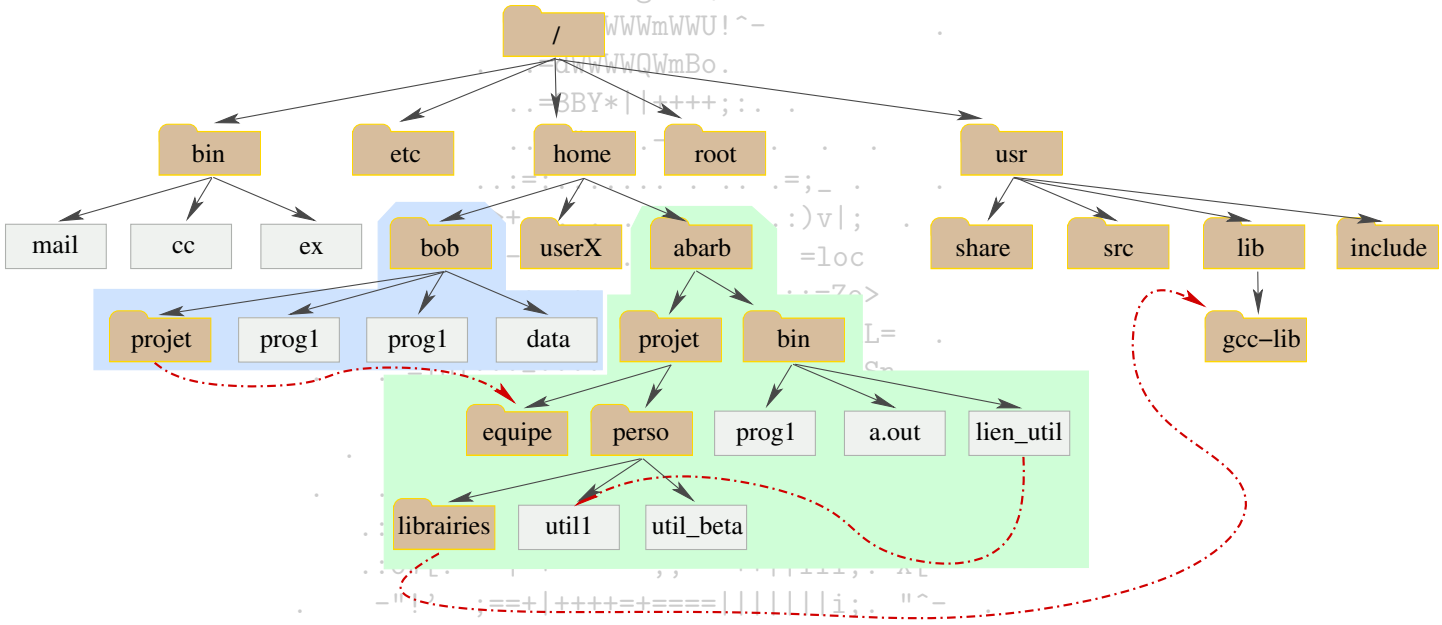


FIG. 9.6 – Système de fichiers avec une structure de graphe sans cycle

Dans cette structure, il est possible de créer des liens dans un répertoire pointant vers des fichiers ou des répertoires situés dans l'arborescence, avec la contrainte de ne pas introduire de cycle dans la structure des fichiers. De fait, un fichier peut porter différents noms et donc être atteint par différents chemins. Ceci pose différents problèmes. Les parties de l'arborescence et les fichiers pointés par des liens doivent être gérés de façon particulière pour ne pas perturber des opérations simples telles que le calcul de l'espace occupé, les procédures de sauvegardes, la recherche de fichiers...L'opération de suppression d'un fichier partagé est plus subtile et peut s'opérer selon différents modes : le fichier peut être supprimé dès qu'un des liens vers le fichier est supprimé, ou seulement lorsque la référence initiale est supprimée, ou lorsque toutes les références au fichier ont disparues.

Structure de graphe général

Dans cette structure, les cycles sont permis. Les problème rencontrés dans les graphes sans cycles sont présents avec une difficulté particulière pour la gestion des suppression de fichiers (Cf. [SGG03]).

9.3 Physique des fichiers

L'OS impose l'utilisation d'un ou plusieurs systèmes de fichiers afin de gérer les données. La partie logique d'un système de fichiers est constituée de l'interface permettant de manipuler les différents concepts présentés ici. Le reste du système de fichier œuvre pour la mise en place des structures et des algorithmes nécessaires à la matérialisation des concepts logiques. La présentation qui suit concerne essentiellement l'implémentation d'un système de fichier sur un disque, ce support étant le plus pratique et le plus courant pour la gestion de fichiers.

Le système de fichiers peut être structuré en plusieurs niveaux, chaque niveau ayant son rôle particulier (Cf. Fig 9.7) Le système de fichier de base

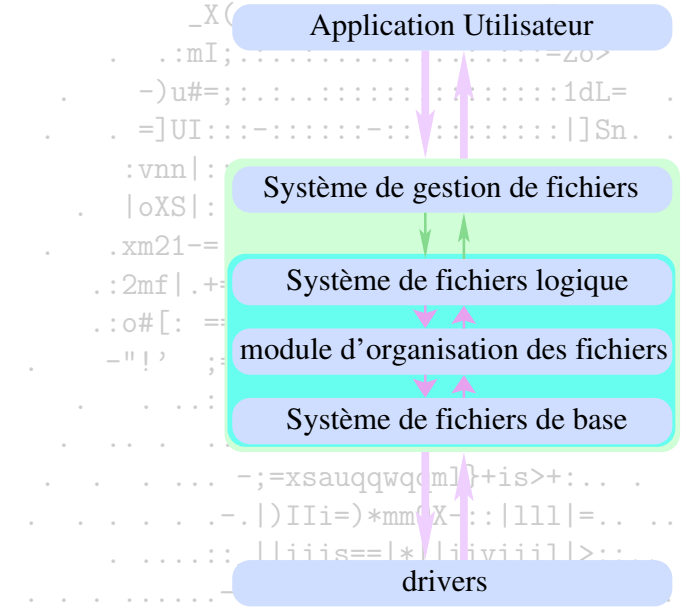


Fig. 9.7 – Couches d'un système de fichiers

lit et écrit les blocs de données en utilisant les adresses physiques sur les périphériques. Ces adresses lui sont fournies par le module d'organisation des

fichiers qui permet de faire la correspondance entre les adresses physiques et logiques. Cette correspondance est déterminée par le type d'allocation pour les fichiers. Le module doit également garder la trace de l'espace libre sur le support. Enfin le système de fichiers logique rassemble l'ensemble des structures et des données permettant la gestion des fichiers logiques, en dehors de leurs contenus. Il gère la structure des répertoires afin de pouvoir fournir au module d'organisation des fichiers les informations asociées à un nom symbolique de fichier. Par ailleurs, il regroupe pour chaque fichier les informations concernant son propriétaire, les permissions définies et la localisation de son contenu dans un bloc de contrôle de fichier (FBC *file block control*). Enfin, c'est lui qui a la charge des mécanismes de sécurité et de protection.

Les OSs, pour la plupart, tolèrent plusieurs systèmes de fichiers : les CD-ROMs sont, pour la plupart, basés sur un système de fichiers standard. En dehors de ces types de média légers, les OSs supportent différents types de systèmes de fichiers sur les disques. La découpe modulaire des systèmes de fichiers permet parfois la mise en commun du système de fichiers de base...

9.3.1 Aperçu de l'implémentation

L'OS utilise des structures présentes sur le support de stockage et des structures gardées en mémoire pour gérer un système de fichiers. Les structures présentes sur une partition (du disque) formatée pour accueillir le système de fichiers associé sont :

- Un bloc de contrôle de boot pour les partitions servant à l'OS pour booter. Ce bloc est en général le premier bloc de la partition en question.
- Un bloc de contrôle de la partition rassemblant les informations telles que le nombre de blocs de la partition, leur taille, le nombre de blocs libres et des pointeurs sur ces blocs, le nombre de FBCs libres et des pointeurs vers ces blocs.
- une structure de répertoire.

Les structures gardées en mémoire sont en général les suivantes :

- Une table rassemblant les informations sur chaque partition montée.
- Une structure contenant les informations sur les répertoires accédés récemment.
- Une table regroupant une copie du FBC pour chaque fichier ouvert.
- Une table rassemblant pour chaque processus des pointeurs vers les entrées des fichiers auquels ils sont liés.

Pour créer un fichier, un appel est fait auprès du système de fichier logique. Ce dernier alloue un FBC, place en mémoire la structure du répertoire concerné et la met à jour sur le disque avec le nom du nouveau fichier et son FBC. Pour cela, il passe la main au module d'organisation et au système de base.

Une fois créé, le fichier peut être utilisé pour stocker des données. Pour cela, il doit être ouvert par l'appel système *open*. L'appel système provoque la recherche du nom passé en paramètre dans le repertoire. Une fois trouvé, son FBC, s'il n'est pas déjà présent dans la table, est copié dans la table recueillant les copies des FBCs et le compteur indiquant le nombre de processus travaillant sur ce fichier est incrémenté. Ensuite, une entrée associée au processus est créée dans la table adéquate, comportant, entre autres choses, un pointeur vers la copie du FBC sus-citée, le mode d'ouverture du fichier, la position dans le fichier...L'appel *open* renvoie un pointeur vers cette dernière entrée. Toutes les opérations portant sur le fichier par le processus seront faites via ce pointeur.

Lorsque le processus ferme le fichier, l'entrée associée au pointeur précédent est supprimée et le compteur indiquant le nombre de processus travaillant sur ce fichier est décrémenté. Quand le compteur retombe à la valeur zéro, l'entrée associée est retirée de la table des fichiers ouverts dans le système.

9.3.2 Implémentation des répertoires

L'implémentation de la gestion des répertoires conditionne l'efficacité et les performances du système de fichiers.

Liste linéaire

Un répertoire est conçu comme une liste linéaire de noms de fichiers avec des pointeurs vers les blocs de données. La recherche d'un nom consiste en une recherche linéaire sur l'ensemble des noms. Elle est simple à mettre en place, mais se révèle peu efficace. La gestion de la libération d'un espace dans la liste peut s'opérer différemment selon les OSs.

Table de hachage

A la liste linéaire de noms de fichiers est associée une table de hachage utilisée pour accélerer la recherche des fichiers. Lorsqu'un fichier est créé, une valeur est calculée à partir de son nom et sert d'index dans une table de hachage. Chaque valeur est associée à un pointeur vers le fichier. Le risque que présente cette méthode est le risque de collision : il faut s'assurer que deux noms différents ne peuvent conduire à la même valeur d'index dans la table de hachage. Par ailleurs, les paramètres de la table de hachage doivent être ajustés au nombre d'entrées dans le répertoire (Cf. [SGG03]).

9.3.3 Méthodes d'allocation des blocs

Allocation continue

Par cette méthode d'allocation, un fichier se voit allouer un ensemble d'adresses contiguës de secteurs sur le disque. L'accès à un fichier ainsi stocké est simple et rapide car les déplacements des têtes de lecture sont très réduits. Toutefois, ce mode d'allocation présente des inconvients :

- Le premier problème est de trouver un espace contigu de taille supérieure ou égale à celle du fichier. Cette allocation repose en partie sur la gestion de l'espace libre qui sera traîtée ci-dessous. Différentes stratégies peuvent être adoptées pour choisir un espace convenable :
  - ↯ First fit : consiste à allouer le premier espace trouvé et satisfaisant la contrainte de taille.
  - ☺ Best fit : consiste à allouer le plus petit espace trouvé et satisfaisant la contrainte de taille.
  - ☹ Worst fit : consiste à allouer le plus grand espace trouvé et satisfaisant la contrainte de taille.

  La dernière stratégie est la plus mauvaise, tant sur le plan de la rapidité que sur celui de l'utilisation du médium de stockage. La première stratégie est en général la plus rapide. La principale limitation de ce mode d'allocation réside dans le problème suivant : l'espace libre peut être suffisant en totalité

pour accueillir un fichier sans qu'aucun espace contigu de la taille désirée le soit. La convergence vers ce problème est accéléré par le phénomène de fragmentation externe[2]. La libération de petits fichiers morcèle l'espace utilisé sans nécessairement restituer de grands blocs. Pour remédier à ce

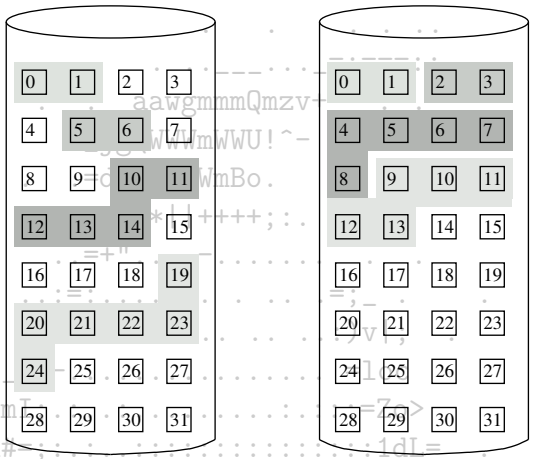


FIG. 9.8 – Allocation continue, avant et aprés recompactage

problème, des procédures de recompactage permettent de rassembler l'espace libre sur un seul ensemble de blocs contigus. Ces procédures sont lentes et pendant leur exécution, le système de fichier de la partition concernée n'est pas disponible.

– Le second problème est la connaissance de la taille du fichier. Il n'est pas possible dans tous les cas de connaître à l'avance la taille du fichier. Par ailleurs, un fichier existant peut être amené à croître. Si trop peu d'espace est alloué, le fichier ne peut augmenter en taille. Certains OSs se contentent dans ce cas de stopper l'application utilisateur en renvoyant une erreur. D'autres cherchent un espace plus grand pour y écrire le fichier et libérer l'ancien espace occupé. Cette procédure est répétée à chaque fois que l'espace n'est plus suffisant, sans que l'utilisateur soit sollicité et ralentissant d'autant le système.

Une astuce, parfois implémentée, permettant de réduire les inconvénients de ce type d'allocation consiste à lier à la fin de l'espace initialement alloué une extension. Mais ce remède n'est que de faible impact sur les faiblesses de la méthode.

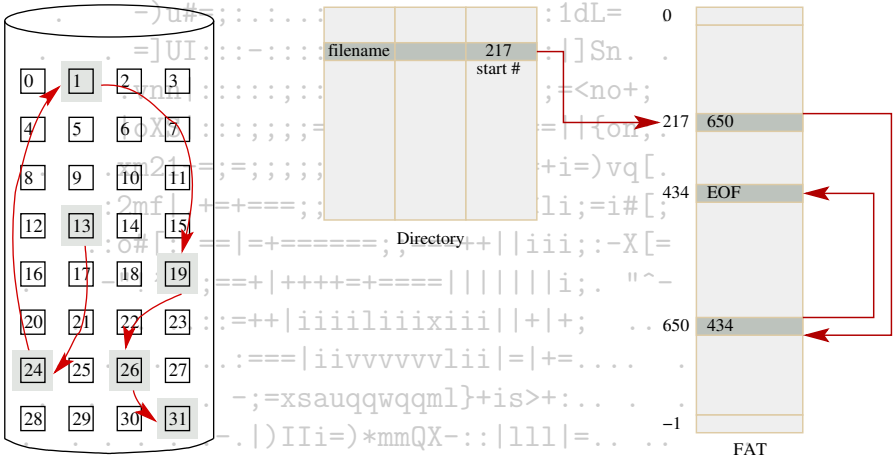
[2]Par opposition à la fragmentation interne. La fragmentation interne résulte du fait que les bits sont alloués par paquets (i.e. pas individuellement) par l'OS. Supposons qu'un OS alloue l'espace d'un disque par paquet de 512 octets. Le stockage d'un fichier de 135 octets engendrera l'allocation d'un bloc dont les 512-135 derniers octets ne seront pas utilisés. La fragmentation interne prend de l'ampleur avec l'augmentation de la taille des blocs et avec le nombre de petits fichiers.

Allocation chaînée

L'allocation chaînée consiste à utiliser une liste chaînée de blocs pour l'allocation d'espace d'un fichier. L'entrée correspondante à un fichier dans son répertoire contient l'adresse du premier et du dernier bloc. Les informations concernant les adresses des blocs intermédiaires sont placées au sein des blocs, grignotant quelques bits : si un bloc comporte 512 octets et si l'adresse d'un bloc est codée sur quatre octets, alors il reste 508 octets pour coder l'information du fichier, et les quatre octets indique l'adresse du bloc suivant. Les adresses des blocs libres sont fournies par le système de gestion de l'espace libre.

Les inconvients de l'allocation continue disparaissent avec cette méthode : il est inutile de connaître à l'avance la taille d'un fichier et il n'y a pas de phénomène de fragmentation externe. En revanche, d'autres problèmes surgissent :

- L'accès indexé ne peut plus être direct. Pour lire le contenu du i^{e}bloc, il est nécessaire de parcourir tous les blocs précédents afin de connaître son adresse physique. Ainsi, ce mode d'accès est inefficace pour l'accès direct.
- Le coût de stockage des adresses de blocs au sein des listes est non négligeable. Il croît avec la taille du fichier. La solution adoptée fréquemment consiste à grouper les blocs en paquets (clusters). Ainsi, l'allocation s'opère au niveau des clusters, allégeant la gestion des adresses (au sein des listes



(a) Allocation chaînée standard

(b) FAT

FIG. 9.9 – Différents modes d'allocation chaînée

chaînées et dans le système de gestion de l'espace libre) et réduisant les déplacements de têtes de lecture. Toutefois, le regroupement en clusters a un coût : une fragmentation interne accrue...

– Le dernier problème est la robustesse de ce chaînage de l'information : la corruption d'un pointeur au sein d'une liste chaînée peut se solder par une redirection vers des blocs d'un autre fichier ou vers des blocs vides et la perte de la fin du fichier. Des structures plus riches que la liste chaînée simple peuvent être employée, au détriment des performances par le coût de leur utilisation...

Une variation efficace et courante sur ce mode d'allocation est la FAT (File Allocation Table), utilisée dans le système MS-DOS. Le système FAT consiste à maintenir en début de partition une table dont chaque entrée correspond à un bloc physique. La table est utilisée pour gérer l'ensemble des listes chaînées, ainsi que les blocs libres (valeur 0) : l'entrée d'un fichier dans son répertoire indique l'adresse du premier bloc, et l'entrée dans la FAT de ce bloc est recherché dans la table. L'adresse du prochain bloc est indiquée, etc. La fin du fichier est codée par une valeur spéciale. Le problème de l'accès direct est bien simplifié par la FAT, car l'adresse du i^{e}bloc logique est facile à déterminer. Toutefois, le placement de la FAT en début de partition peut engendrer beaucoup de déplacements, à moins que la FAT soit placée en cache : d'abord se placer au début de partition, puis chercher l'adresse physique dans la table et enfin se déplacer vers l'adresse lue...

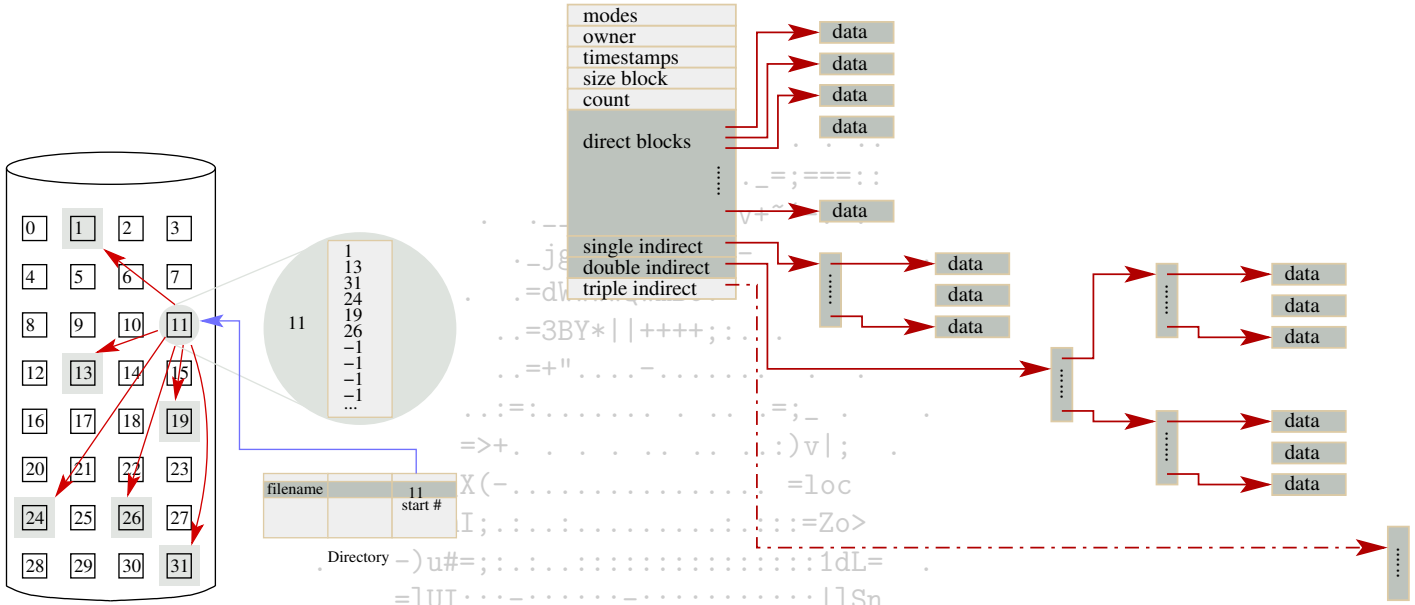
Allocation indexée

Malgré les améliorations apportées par l'allocation chaînée par rapport à l'utilisation du support, les accès aléatoires sont peu efficaces (en l'absence de FAT). L'allocation indexée est basée sur le regroupement des adresses de blocs dans un bloc jouant le rôle d'index. Chaque fichier a son index, la i^{e}entrée donnant l'adresse du i^{e}bloc. Comme l'allocation chaînée, ce type d'allocation est à l'abri de la fragmentation externe. De plus, l'accès direct est facilité par l'usage de l'index. En revanche, les blocs alloués pour l'indexation des blocs d'un fichiers sont rarement remplis. L'espace perdu est d'autant plus important que les blocs sont grands. Par ailleurs les blocs doivent être assez grands pour permettre l'indexation de grands fichiers. Différentes adaptations permettre de réduire la taille des blocs tout en permettant l'indexation de grands fichiers :

– Index chaînés : le dernier indice d'un bloc référence éventuellement un autre bloc servant d'index, et ainsi de suite. La taille de l'index est donc adaptée à la taille du fichier.

– Indexs hiérarchiques : pour deux niveaux d'indexation, les entrées de l'index du bloc pointent vers des blocs d'index indexant les blocs de données. Des niveaux d'indexation peuvent être ajoutés en fonction de la taille maximale que le système doit pouvoir indexer. Supposons par exemple que les blocs soient adressés par des entiers codés sur 32 bits (4 octets). Le nombre de blocs adressables est de $2^{32} \approx 4,29.10^{9}$. Supposons de plus qu'un bloc comporte 4096 octets ($4 \times 1024 = 4 \times 2^{10}$). La capacité maximale adressable est donc de $2^{32+12} \approx 17,59$To. Un niveau d'indexation permet de construire des fichiers de 4096×2^{10} octets ($2^{10+2+10} \approx 4,1$Mo). Deux niveaux d'indexation permettent de distinguer 2^{20} blocs, ce qui correspond à un fichier de 2^{20+12} octets, i.e. 4,29Go.

– Combinaison des solutions précédentes : une autre solution, employée dans la conception des i-nodes (Unix), consiste à combiner les deux approches précédentes. Comme l'illustre la figure 9.10(b), les premiers blocs de données sont indexés directement dans l'i-node. Si le nombre de blocs excède le nombre de pointeurs directs, l'adresse d'un index utilisant un bloc est utilisable, puis une indéxation hiérarchique progressive est utilisée. En



(a) Allocation indexée standard (b) modèle d'i-node

FIG. 9.10 – Différents modes d'allocation indexée

supposant que les pointeurs directs sont au nombre de 12 (et avec les spécifications précédentes), un fichier de $12 \times 2^{10+2} \approx 49.10^3$ octets ne fait pas intervenir l'index chaîné. En utilisant le bloc d'index supplémentaire (l'index chaîné), il est possible de construire un fichier d'une capacité d'au plus $12 \times 2^{10+2} + 4096 \times 2^{10} \approx 4,2$Mo. En utilisant le premier niveau d'indexation hiérarchique, il est possible de construire un fichier d'un capacité d'au plus $12 \times 2^{10+2} + 4096 \times 2^{10} + 4096 \times 2^{20} \approx 4,29$Go. Le dernier niveau permet la construction d'un fichier utilisant au plus $12 + 2^{10} + 2^{20} + 2^{30}$ blocs, i.e. plus de 4To.

Performances

Les performances d'une méthode d'allocation s'évaluent en général sur les deux critères suivants :
- L'utilisation de l'espace du médium de stockage.
- Le temps d'accès aux blocs.

Comme nous l'avons vu, certaines méthodes sont bien adaptées à un type d'accès. Certains OSs jonglent avec les différentes politiques d'allocation et permettent de définir le type d'accès pour un fichier afin de choisir la meilleure option de stockage. Enfin, certains OSs choisissent le type d'allocation en fonction de la taille du fichier. . .Les méthodes performantes s'éloignent des conceptions généralistes pour s'adapter aux particularités d'un fichier.

9.3.4 Gestion des blocs libres

Le système gère une liste des blocs libres pour tout medium de stockage. Cette gestion peut se baser sur différentes structures.

Vecteur (ou carte) de bits

La liste des blocs libres peut être implémentée comme un vecteur de bits. Le i^{e}bit mis a 1 signifie que le i^{e}bloc est libre. Le traitement de vecteur de bits est facilité sur certains processeurs par des opérations dédiées. Toutefois, le nombre de blocs est généralement si grand que la taille du vecteur devient trop lourde, voire impossible à gérer en méloire centrale.

Cette structure est donc souvent délaissée.

Liste chaînée

En plaçant l'adresse du premier bloc libre dans une endroit précis, et en indiquant dans chaque bloc libre l'adresse du prochain bloc libre, une liste chaînée des blocs libres est construite. Cette structure se prête mal à une lecture complète de la liste, mais cette opération est rare : en général, un seul bloc est nécessaire pour l'écriture d'un petit fichier, et le premier de la liste convient.

Regroupement des adresses de blocs libres

Il est possible d'utiliser des blocs chaînés contenant les adresses des blocs libres. Il est alors facile de repérer rapidement les blocs libres.

Comptage

Les allocations et les libérations de blocs libres font souvent intervenir des groupes de blocs contigus. Il est alors intéressant de considérer, non plus les blocs libres individuellement, mais les groupes contigus de blocs libres. La liste de blocs libres est alors une liste donc chaque entrée comporte l'adresse du premier bloc libre d'un ensemble contigu de blocs libres et le nombre de blocs de cet ensemble.

9.4 Concepts avancés sur les systèmes de fichiers

9.4.1 Montage d'un FS

A un système de fichiers correspond une partition. C'est le FS, ou plus précisément les structures assurant la gestion du FS, qui définissent la façon dont l'espace de la partition est utilisé. Afin que les fichiers présents sur une partition puissent être utilisés par l'OS, l'OS doit avoir fait le lien entre la partition physique et la structure de fichiers en cours. L'opération par laquelle s'opère ce lien s'appelle le montage.

Le principe est simple : la désignation physique de la partition et un point de la structure de fichier en cours sont passés en paramètres. Un identifiant pour le FS peut également être précisé si l'OS sait gérer plusieurs FS. L'opération peut être automatique ou manuelle, selon les possibilités de l'OS.

Lorsqu'une opération de montage est demandée, l'OS contrôle la validité des structures du FS, puis lit la structure des fichiers présente sur le médium et l'intégre à la structure existante en indiquant dans la structure en cours la nouvelle association entre le point de montage et la partition montée (et le type de FS, si nécessaire). Les variantes concernant les possibilités de montage dépendent de l'OS : par exemple, certains OSs permettent le montage dans un répertoire non-vide, d'autres non. C'est le rôle de la sémantique de l'OS de définir les spécificités des implémentations des opérations de montage.

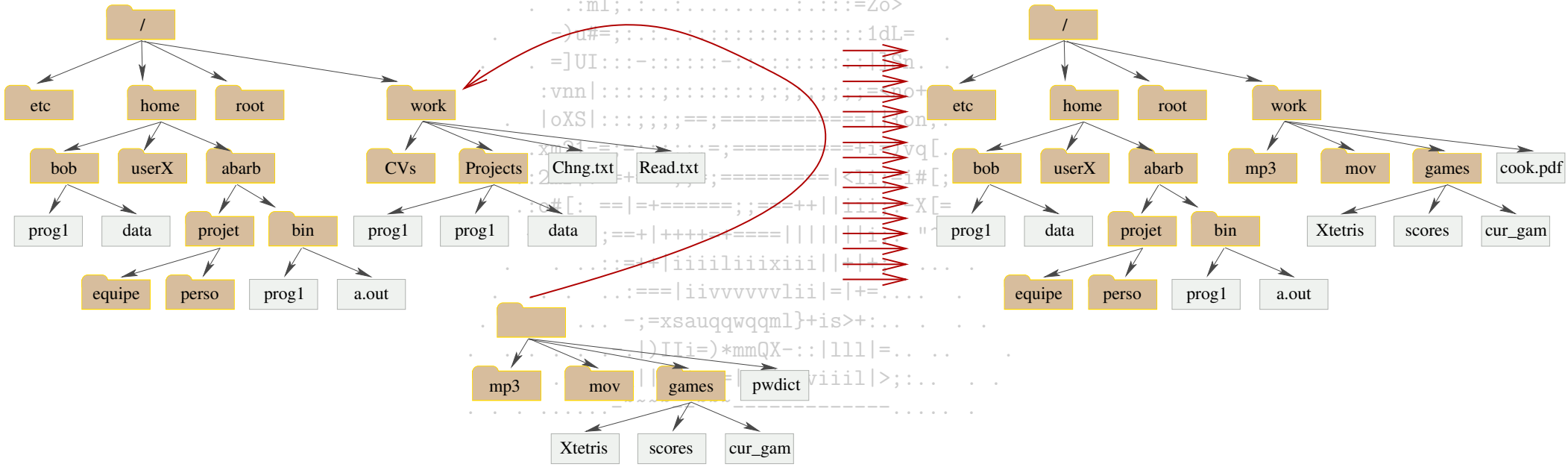


Fig. 9.11 – Opération de montage

Signalons qu'à l'opération de montage est associé l'opération de démontage ayant pour effet de retirer, de la structure courante des fichiers, la structure résidant sur la partition faisant l'objet du démontage.

De nombreux OSs permettent la gestion de différents FSs, notamment pour permettre l'utilisation locale de supports légers tels que le CD-ROM (ce support étant presque toujours utilisé avec le FS iso9660) mais aussi pour faciliter le partage de supports locaux ou distants. Le recours à un système de fichier virtuel (VFS pour virtual file system) est courant pour permettre cette cohabitation. Les applications utilisateurs n'ont accès qu'aux opérations

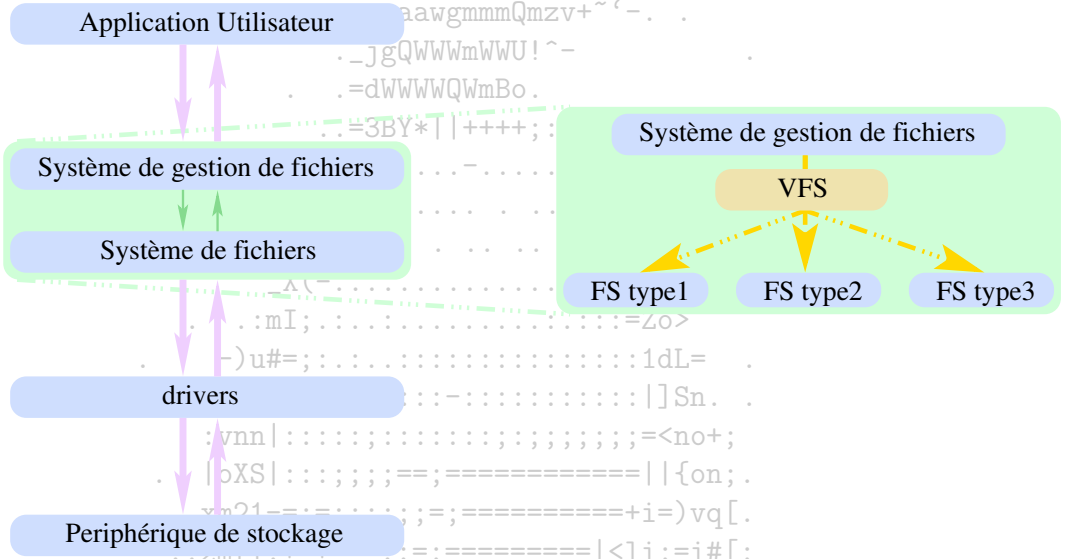


FIG. 9.12 – Utilisation d'un système de fichier virtuel

de bases (open, read, write et close). Le VFS fait le lien entre les opérations demandées par l'utilisateur et les spécificités du FS sur lequel interviennent ces opérations. Il a également pour rôle la distinction des fichiers ouverts issus des différentes sources (locales ou distantes). Chaque fichier ouvert dans le système est associé à un vnode. Cette structure permet d'identifier la source distante d'un fichier ou son FS pour un fichier local. Le système travaille alors sur la base des vnodes.

9.4.2 Partage des fichiers et mécanismes de protection

Les problèmes du partage et de la protection des fichiers sont indissociables. Ils peuvent être pris à différents niveaux de difficulté : d'abord dans le cadre multi-utilisateur, puis en ajoutant à ce cadre le partage via le réseau. Ils sont également accompagnés du problème des accès concurrents.

Par ailleurs, il est important de préciser que ce qui est partagé, ce sont les droits pouvant porter sur un fichier. Les droits sont directement liés aux opérations applicables à un fichier (lecture d'un fichier, écriture, exécution, listage des entrées d'un répertoire, positionnement dans. . .et d'autres en fonction de la sémantique de l'OS).

Cadre multi-utilisateur

Un système qui gère différents utilisateurs passe par le biais d'identificateurs. La plupart de ces systèmes gérent également des groupes d'utilisateurs, et dans ce cadre, chaque utilisateur peut éventuellement appartenir à différents groupes à la fois.

La façon la plus souple pour gérer les droits relatifs à un fichier consiste à construire une liste de contrôle d'accès (ACL) : chaque utilisateur disposant de droits y est listé avec les droits correspondant. En plus de la consommation d'espace qu'engendre les ACLs, les deux points suivants sont des inconvénients majeurs :

- la maintenance et l'utilisation des ACLs peut s'avérer lourde.
- l'espace réservé aux métadonnées d'un fichier doit être ajustable. La gestion de l'espace et des structures du FS s'en trouve plus complexe.

La plupart des systèmes ont été conduits vers la distinction de trois groupes d'usagers, relativement à un fichier (ou répertoire). Cette condensation permet une certaine souplesse et facilite la gestion des droits. Les entités considérées sont les suivantes :

- D'abord le propriétaire du fichier. C'est l'utilisateur qui définit tous les droits sur le fichier.
- Le groupe d'utilisateurs d'un fichier désigne un ensemble d'usagers définis dans le système.
- Tous les autres usagers.

Pour chacun de ces trois groupes d'usagers, des droits sont définis.

De plus en plus de systèmes permettent d'utiliser comme base le système précédent auquel est greffée la possibilité d'étendre les droits par une ACL. La résolution d'éventuels conflits de définitions suit la règle donnant raison à la spécificité : si les membres du groupe d'ont pas d'accès à un fichier, et si un utilisateur membre de ce groupe apparait dans l'ACL du fichier avec des droits de lecture, ce sont les droits définis dans l'ACL qui prévalent.

Ce mécanisme de partage est basé sur l'identité de l'utilisateur sollicitant un accès particulier. Certains systèmes adoptent un mécanisme plus lourd, basé sur l'association de chaque fichier (ou simplement les répertoires) à un mot de passe (ou plusieurs en fonction du niveau de contrôle souhaité). Ce genre de mécanisme est supportable dans les systèmes de fichier construisant une structure simple.

Partage sur un système de fichiers distant

La principale difficulté de l'implémentation d'un système de fichiers distant est son intégration dans la gestion de fichier de l'OS. Le partage de la ressource est basé sur un schéma client/serveur. Le serveur définit des autorisations de montage sur des portions de sa structure de fichiers pour des clients identifiés. L'identification peut se baser sur un nom de machine ou une adresse IP par exemple. L'authentification, via le réseau, comporte de nombreux

risques de faille. La complexité des procédures à mettre en place pour réduire ces risques est telle que la plupart des méthodes d'identification ne sont pas sécurisées.

Une fois le FS distant monté, les opérations sur les fichiers se font via le mécanisme de contrôle de droits de l'utilisateur. Ceci suppose que les informations portant sur l'utilisateur sont les mêmes sur le client et sur le serveur. Afin de faciliter la gestion (et surtout la maintenance) de ce type d'information, des systèmes d'information distribués sont souvent mis en place.

Gestion des accès concurrents sur les fichiers

La gestion des accès concurrents par un FS permettant le partage des fichiers est une de ses caractéristiques profondes. Cette gestion se base sur les concepts de synchronisation vus dans les pages précédentes de ce support. Plus elle est élaborée et plus elle est complexe et lourde à mettre en œuvre, tant à cause des communications qu'elle engendre que de la latence des lectures disques. Certains FSs (Unix FS) partagent l'image physique d'un fichier entre tous les utilisateurs. Ainsi, toute opération effectuée par un utilisateur est visible par tous les utilisateurs qui ont ouvert ce fichier. Le fichier est donc une ressource en accès exclusif, et les opérations se font donc une à une, ce qui ralentit les accès. D'autres (AFS ou Andrew FS) fournissent une image du fichier à l'instant de l'ouverture de ce fichier par un utilisateur. Chaque image est indépendante des autres, et les changements opérés sur un fichier entre l'appel open et l'appel close ne sont visibles qu'après le close par les autres utilisateurs. Ainsi, la gestion des accès concurrentiels sur les images ne ralentit pas les opérations.

9.4.3 Qualités d'un FS

Les disques font partie des éléments les plus lents d'un système. Le FS joue un rôle important sur le bon emploi du support. Les critères de jugement quant à la qualité d'un FS sont l'efficacité de l'utilisation du support et le temps d'accès à l'information,

Comme le montre ce qui suit, les facteurs agissant sur l'évaluation de ces critères sont nombreux. L'implémentation du concept de répertoire influe directement sur ces deux critères. Par exemple, si le répertoire gère un attribut du type «date de dernier accès», le coût de la mise à jour de l'attribut est lourd. La mise en place des structures de gestion du FS peut également être importante. Certains FS préparent ces structures à l'avance, et les disséminent sur le disque pour permettre une proximité entre les méta-données et les données (afin de réduire les temps d'accès). Enfin, la plupart des FS implémentent des méthodes d'allocation sur des structures plus souples et aussi plus complexes que celles présentées : leur gestion est plus complexe, mais cette souplesse peut permettre de gagner sur l'un des critères.

Toutefois, les structures d'un FS ne sont pas les seuls paramètres régissant l'optimisation de l'utilisation d'un support de stockage. Il est possible d'améliorer l'exploitation du FS par diverses techniques :

- des techniques basées sur l'emploi de cache (portant sur les blocs ou les pages).
- la désynchronisation des opérations d'écriture sur les données.
- l'utilisation d'un disque virtuel, placé en mémoire (RAM disque).

9.4.4 Robustesse d'un FS

Les structures que l'OS utilise pour gérer un FS sont en partie sur le support physique et en partie en mémoire centrale. Pour les systèmes utilisant un système de cache pour les répertoires, les structures présentes sur le support physique sont en général moins à jour que celles placées en mémoire centrale, car les opérations d'écriture peuvent être différées. Pour cette raison, un crash du système peut entrainer des pertes d'informations sur les structures décrivant la structure des fichiers si, par exemple, le crash a lieu pendant la mise à jour des structures physiques à partir des données placées en mémoire.

Lors du montage d'une partition, un contrôle (plus ou moins approfondi. Certains OSs font un contrôle rapide qui, en cas de découverte d'incohérence, le font suivre d'une procédure de restitution) des structures du répertoire se déroule. Le croisement des informations des répertoires et des blocs libres permet la détection d'un certain nombre d'erreur. La redondance des méta-données permet également la vérification de la cohérence. Le type d'erreur détectable est fonction de la précision des structures et de leur organisation. . .Remarquons par ailleurs que les méthodes d'allocation des blocs exposent les FS à des risques différents de pertes de données. La robustesse d'un FS est sa capacité à résister aux incohérences qui peuvent surgir dans ses méta-données, en permettant leur détection et leur éradication.

Les procédures de reconstruction d'un système de fichier ne peuvent apporter la garantie d'une restauration complète des métadonnées. Par ricochet, la perte de méta-données peut engendrer la perte de données. Les procédures de sauvegarde de données ou l'installation d'un système RAID sont de bons remparts contre la perte de données. Les procédures de sauvegarde incrémentales sont un moyen simple de réduire les pertes. Elles s'effectuent par cycle :

Jour 1 Sauvegarde de tous les fichiers de la partition.
Jour 2 Sauvegarde des fichiers modifiés depuis le premier jour
Jour 3 Sauvegarde des fichiers modifiés depuis le deuxième jour
⋮
Jour N-1 Sauvegarde des fichiers modifiés depuis le jour précédent. Fin du cycle.

9.4.5 FS journalisé

Le principe du FS journalisé est issu des technologies liées aux bases de données. L'utilisation d'un fichier de log des transactions est efficace pour la vérification de l'intégrité des métadonnées d'un FS.

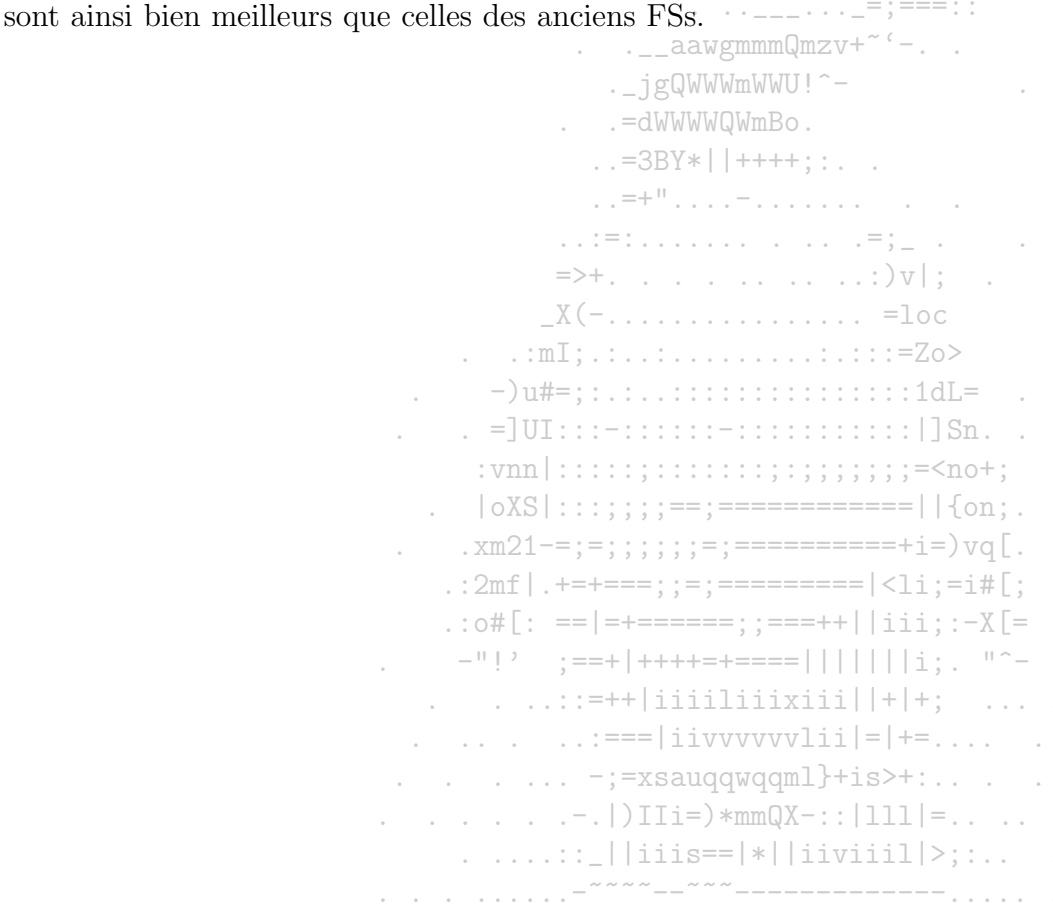
Plus précisément, cela consiste à écrire dans un fichier toute opération portant sur les méta-données (structure d'un répertoire, pointeurs de blocs libres, pointeurs de FBC. . .) un changement. Les "transactions" sont ici les opérations logiques de mise à jour. Toute transaction doit pouvoir se décomposer en un ensemble d'opérations atomiques. Ces opérations atomiques sont déclarées dans le fichier de log comme le corps d'une transaction. L'état de la transaction est suivi par un pointeur sur l'opération en cours. Si toutes les opérations se déroulent normalement, la transaction est dite "commise". Sinon, elle est abandonnée, et les opérations effectuées jusqu'au plantage sont annulées[3]. Une transaction commise est irreversible. Lorsqu'une transaction est accomplie, son entrée dans le fichier de log est retirée. Elle est par conséquent irréversible.

[3]Ceci implique une connaissance des valeurs avant modification de toutes les opérations atomiques effectuées. Pour un exposé détaillé, se reporter à [SGG03]

Le fichier de log sert de tampon, en quelques sorte, pour les transactions. Il peut se trouver sur la partition ainsi contrôlée ou sur une partition distincte.

Ce mode de fonctionnement résiste bien aux crashs. Les transactions présentes dans le fichier de log peuvent être reprises là où elles ont été laissées. Les transactions abandonnées sont annulées et les autres sont poursuivies. Ainsi est préservé l'intégrité des méta-données.

Un autre avantage de la journalisation est qu'elle permet une meilleure répartition des opérations d'écriture, nécessaires aux mises à jours des structures sur disques. Les mises à jour sont asynchrones et régroupées en séquences puis opérées au fil des accès disques, lorsque le déplacement engendré est moindre. Les performances sont ainsi bien meilleurs que celles des anciens FSs.



Chapitre 10

Exemples de systèmes de fichiers

10.1 FAT,VFAT et NTFS

10.2 {Ext2,Ext3,Reiser}fs

Quatrième partie

Mémoire

Chapitre 11

Gestion de la mémoire

Nous allons nous intéresser à présent à la ressource mémoire centrale (centrale pour la distinguer de la mémoire secondaire, i.e. les supports de stockage non-volatile), à son utilisation et à sa gestion. Notre présentation suivra le fil directeur de la multiprogrammation, son but ultime étant l'optimisation de l'utilisation du CPU et des autres ressources par les processus utilisateurs.

Pour pouvoir utiliser un ordinateur en multiprogrammation, l'OS charge plusieurs processus en mémoire centrale (MC). La façon la plus simple consiste à affecter à chaque processus un ensemble d'adresses contiguës. Quand le nombre de tâches devient élevé, pour satisfaire au principe d'équitabilité et pour minimiser le temps de réponse des processus, il faut pouvoir simuler la présence simultanée en MC de tous les processus. D'où la technique de "va et vient" ou recouvrement (swapping), qui consiste à stocker temporairement sur disque l'image d'un processus, afin de libérer de la place en MC pour d'autres processus.

D'autre part, la taille d'un processus doit pouvoir dépasser la taille de la mémoire disponible, même si l'on enlève tous les autres processus. L'utilisation de pages (mécanisme de pagination) ou de segments (mécanisme de segmentation) permet au système de conserver en MC les parties utilisées des processus et de stocker, si nécessaire, le reste sur disque.

Le rôle du gestionnaire de la mémoire est de connaître les parties libres et occupées, d'allouer de la mémoire aux processus qui en ont besoin, de récupérer de la mémoire à la fin de l'exécution d'un processus et de traiter le recouvrement entre le disque et la mémoire centrale, lorsque la mémoire ne suffit pas à contenir tous les processus actifs.

Avant de plonger dans les détails des gestion possibles de cette ressource, observons comment elle peut être utilisée par un programme.

11.1 Utilisation de la mémoire

La mémoire peut être vue comme un ensemble de cases de stockage volatile et référençables par les adresses d'un espace contigu d'adresses. Comment ces adresses mémoire sont utilisées ou référencées dans le code d'un programme ? La réponse dépend de la gestion de la mémoire par l'OS et des étapes par lesquelles passe le code (Cf. Fig.11.1) :

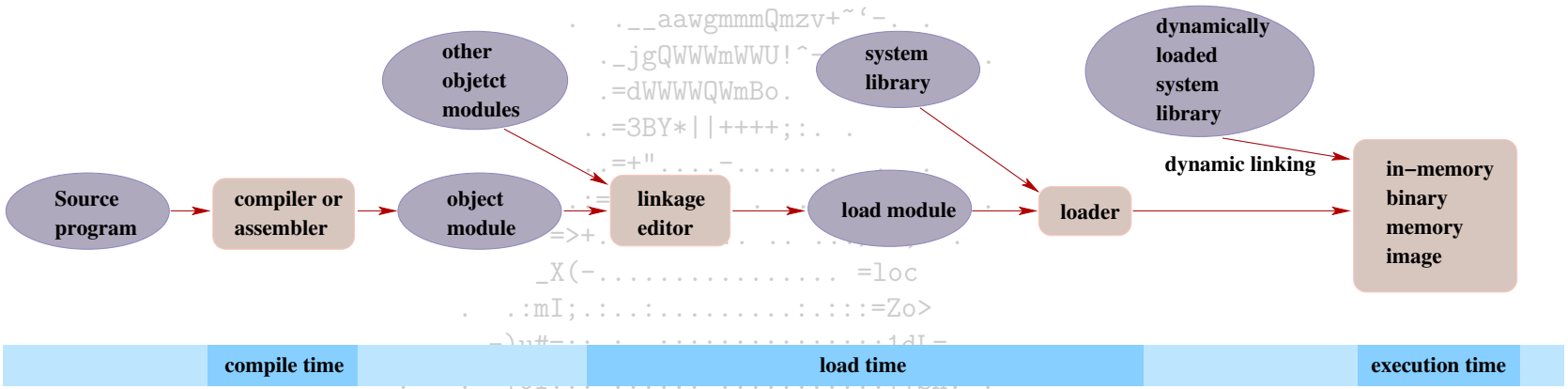


FIG. 11.1 – Processus de génération d'un code

- Si la zone mémoire où résidera le code du programme est connue, alors les adresses générées par la compilation peuvent être celles utilisées pendant l'exécution. Si le code doit être placé à partir de l'adresse R, alors les adresses symboliques du code source seront translatées de la valeur R. Si R change, le programme source doit être recompilé.
- Si la zone n'est pas connue au moment de la compilation, les adresses générées doivent pouvoir être temporaires. Les adresses finales peuvent alors être produites au chargement du code. Si R change, il suffit de recharger le code.
- Enfin, si le processus accueillant le code du programme doit pouvoir être changé de zone mémoire pendant l'exécution, alors les adresses finales doivent être produites pendant l'exécution. Pour cela, des composants matériels sont nécessaires. La MMU (Memory Management Unit) gère la correspondance entre l'espace d'adressage logique (ou virtuel) et l'espace d'adressage physique du code. La correspondance n'est autre qu'une translation d'une valeur placée dans un registre

Le chargement dynamique est une possibilité offerte par les politiques de gestion évoluées qui permet d'alléger la consommation de la ressource mémoire. Il consiste à retarder le chargement des routines jusqu'à leur appel. Si une routine non chargée est appelée, elle est chargée, la table d'adressage du programme est mise à jour, puis le contrôle est passé à la routine. Le chargement dynamique n'est pas automatiquement mis en place par l'OS : c'est au programmeur de prévoir d'utiliser cette possibilité.

Le dernier point intéressant est le concept de liaison dynamique. Certains OSs ne supportent que les liens statiques. Par conséquent, tous les codes de programmes comportent une place pour une copie des librairies qu'ils utilisent. Ainsi, beaucoup d'espace peut être gaspillé. Avec la possibilité de liaison dynamique, une seule copie en mémoire est suffisante : si une routine d'une librairie partagée est appelée, et si elle n'a pas été chargée en mémoire par un autre processus, elle est chargée. Sinon le code chargé est partagé. La gestion des liens dynamiques nécessite l'arbitrage de l'OS car la zone mémoire d'un processus est protégée des autres processus.

Voyons à présent les différentes gestions rencontrées dans les OSs.

11.2 Gestion sans recouvrement, ni pagination

11.2.1 La monoprogrammation

Il n'y a en MC que :
- un seul processus utilisateur,
- le processus système (pour partie en RAM, pour partie en ROM ; la partie en ROM étant appelée BIOS [Basic Input Output System])
- les pilotes de périphériques

Cette technique en voie de disparition est limitée à quelques micro-ordinateurs. Elle n'autorise qu'un seul processus actif en mémoire à un instant donné.

11.2.2 La multiprogrammation

La multiprogrammation est utilisée sur la plupart des ordinateurs : elle permet à plusieurs processus de partager les ressources et à plusieurs utilisateurs de travailler en temps partagé avec la même machine.

Supposons qu'il y ait n processus indépendants en MC, chacun ayant la probabilité p d'attendre la fin d'une opération d'E/S. La probabilité que le processeur fonctionne est $1 - p^n$. Il s'agit d'une estimation grossière (puisque sur une machine monoprocesseur, les processus ne sont pas indépendants - attente de libération de la ressource processeur pour démarrer l'exécution d'un processus prêt). Toutefois, on remarque que plus le nombre n de processus en MC est élevé, plus le taux d'utilisation du processeur est élevé : on a donc intérêt à augmenter la taille de la MC. Une solution simple consiste à diviser la mémoire en n partitions fixes, de tailles non nécessairement égales (méthode MFT [Multiprogramming with a Fixed number of Tasks] apparue avec les IBM 360). Il existe deux méthodes de gestion pour l'utilisation de ces partitions :

- Une file d'attente par partition est créée. Chaque nouveau processus est placé dans la file d'attente de la plus petite partition pouvant le contenir. Inconvénients :
  - on perd en général de la place au sein de chaque partition
  - il peut y avoir des partitions inutilisées (leur file d'attente est vide)

– Une seule file d'attente globale est créée. Il existe deux stratégies :
  • dès qu'une partition se libère, on lui affecte la première tâche de la file qui peut y tenir. Main on peut ainsi affecter une partition de grande taille à une petite tâche et perdre beaucoup de place.
  • dès qu'une partition se libère, on lui affecte la plus grande tâche de la file qui peut y tenir. Mais on pénalise ainsi les processus de petite taille.

11.2.3 Code translatable et protection

Avec le mécanisme de multiprogrammation, un processus peut être chargé n'importe où en MC. Il n'y a plus concordance entre l'adresse dans le processus et l'adresse physique d'implantation. Le problème de l'adressage dans un processus se résout par l'utilisation d'un registre particulier, le registre de base : au lancement du processus, on lui affecte l'adresse de début de la partition qui lui est attribuée. On a alors :

adresse physique = adresse de base (contenu de ce registre) + adresse relative (mentionnée dans le processus)

De plus, il faut empêcher un processus d'écrire dans la mémoire d'un autre. Deux solutions existent :
– des clés de protection : une clé de protection pour chaque partition, dupliquée dans le mot d'état (PSW = Program Status Word) du processus actif implanté dans cette partition. Si un processus tente d'écrire dans une partition dont la clé ne concorde pas avec celle de son mot d'état, il y a refus (génération d'erreur par déroutement).
– un registre limite, chargé au lancement du processus à la taille de la partition. Toute adresse mentionnée dans le processus (relative) supérieure au contenu du registre limite entraîne un refus (génération d'erreur par déroutement).

11.3 Gestion avec recouvrement, sans pagination

Dès que le nombre de processus devient supérieur au nombre de partitions, il faut pouvoir simuler la présence en mémoire de tous les processus pour pouvoir satisfaire au principe d'équité et minimiser le temps de réponse des processus. La technique du recouvrement (swapping) permet de stocker temporairement sur disque des images de processus afin de libérer de la MC pour d'autres processus.

En pratique des partitions de taille variable sont utilisées (au lieu de partitions de tailles immuables), car le nombre, la taille et la position des processus peuvent varier au cours du temps. Nous ne sommes plus limités par des partitions trop grandes ou trop petites comme avec les partitions fixes. Cette amélioration de l'usage de la MC nécessite un mécanisme plus complexe d'allocation et de libération.

11.3.1 Opérations sur la mémoire

Compactage

Le compactage de la mémoire permet de regrouper les espaces inutilisés. Très coûteuse en temps CPU, cette opération est effectuée le moins souvent possible.

Allocation dynamique

S'il y a une requête d'allocation dynamique de mémoire pour un processus, on lui alloue de la place dans le tas (heap) si l'OS le permet, ou bien de la mémoire supplémentaire contiguë à la partition de ce processus (agrandissement de celle-ci). Quand il n'y a plus de place, un ou plusieurs processus sont déplacés :

- soit pour récupérer par ce moyen des espaces inutilisés,
- soit en allant jusqu'au recouvrement. A chaque retour de recouvrement (swap), on réserve au processus une partition un peu plus grande que nécessaire, utilisable pour l'extension de la partition du processus venant d'être chargé ou du processus voisin.

Il existe trois façons de mémoriser l'occupation de la mémoire : les tables de bits (bits maps), les listes chaînées et les subdivisions (buddy).

11.3.2 Gestion de la mémoire par table de bits

La MC est divisée en unités d'allocations de quelques octets à quelques Ko. A chaque unité correspond un bit de la table de bits : valeur 0 si l'unité est libre, 1 sinon. Cette table est stockée en MC. Plus la taille moyenne des unités est faible, plus la table occupe de place.

A un retour de recouvrement (swap), le gestionnaire doit trouver une suite de 0 consécutifs assez longue dans la table pour que la taille cumulée de ces unités permette de loger le nouveau processus à implanter en MC. Le choix parmi les suites possibles peuvent se faire sur l'un des trois critères suivants :

- First fit : la première zone libre rencontrée est choisie. Cet algorithme est rapide.
- Best fit : le meilleur ajustement rencontré est retenu. L'inconvénient de cet algorithme est qu'il a tendance à créer de nombreuses petites unités résiduelles inutilisables. La fragmentation engendrée peut nécessiter un compactage ultérieur.
- Worst fit : le plus grand résidu trouvé est choisi, avec le risque de fractionnement regrettable des grandes unités.

11.3.3 Gestion de la mémoire par liste chaînée

Une liste chaînée des zones libres en MC est utilisée. On applique :

- soit l'un des algorithmes précédents,

– soit un algorithme de placement rapide (quick fit) : on crée des listes séparées pour chacune des tailles les plus courantes, et la recherche est considérablement accélérée.

A l'achèvement d'un processus ou à son transfert sur disque, il faut du temps (mise à jour des liste chaînées) pour examiner si un regroupement avec ses voisins est possible pour éviter une fragmentation excessive de la mémoire.

En résumé, les listes chaînées sont une solution plus rapide que la précédente pour l'allocation, mais plus lente pour la libération.

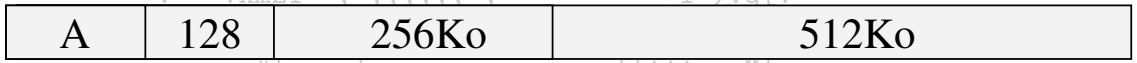
11.3.4 Gestion de la mémoire par subdivisions (ou frères siamois)

Cet algorithme proposé par Donald KNUTH en 1973 utilise l'existence d'adresses binaires pour accélérer la fusion des zones libres adjacentes lors de la libération d'unités.

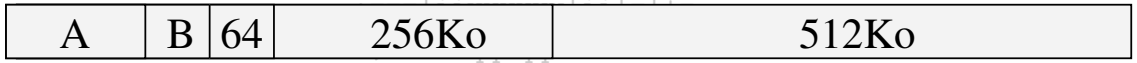
Le gestionnaire mémorise une liste de blocs libres dont la taille est une puissance de 2 (1, 2, 4, 8 octets, ..., jusqu'à la taille maximale de la mémoire).

Par exemple, avec une mémoire de 1 Mo, on a ainsi 251 listes. Initialement, la mémoire est vide. Toutes les listes sont vides, sauf la liste 1 Mo qui pointe sur la zone libre de 1 Mo :

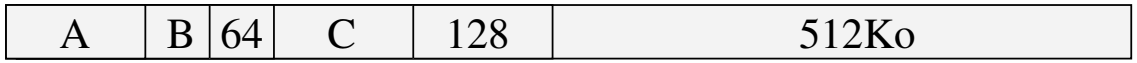
Un processus A demande 70 Ko : la mémoire est fragmentée en deux compagnons (buddies) de 512 Ko ; l'un d'eux est fragmenté en deux blocs de 256 Ko ; l'un d'eux est fragmenté en deux blocs de 128 Ko et on loge A dans l'un d'eux, puisque $64 < 70 < 128$:



Un processus B demande 35 Ko : l'un des deux blocs de 128 Ko est fragmenté en deux de 64 Ko et on loge B dans l'un d'eux puisque $32 < 35 < 64$:



Un processus C demande 80 Ko : le bloc de 256 Ko est fragmenté en deux de 128 Ko et on loge C dans l'un d'eux puisque $64 < 80 < 128$:



A s'achève et libère son bloc de 128 Ko. Puis un processus D demande 60 Ko : le bloc libéré par A est fragmenté en deux de 64 Ko, dont l'un logera D :

D	64	B	64	C	128	512Ko

B s'achève, permettant la reconstitution d'un bloc de 128 Ko :

D	64	128	C	128	512Ko

D s'achève, permettant la reconstitution d'un bloc de 256 Ko, etc...

256	C	128	512Ko

L'allocation et la libération des blocs sont très simples. Mais un processus de taille $2^n + 1$ octets utilisera un bloc de 2^{n+1} octets ! Il y a beaucoup de perte de place en mémoire.

11.4 Gestion avec recouvrement et pagination ou segmentation

La taille d'un processus doit pouvoir dépasser la taille de la mémoire physique disponible, même si l'on enlève tous les autres processus. En 1961, J. FOTHERINGHAM proposa le principe de la mémoire virtuelle : l'OS conserve en mémoire centrale les parties utilisées des processus et stocke, si nécessaire, le reste sur disque. Mémoire virtuelle et multiprogrammation se complètent bien : un processus en attente d'une ressource n'est plus conservé en mémoire, si cela s'avère nécessaire.

La mémoire virtuelle peut faire appel à deux mécanismes : la segmentation ou la pagination. La mémoire est divisée en segments ou pages. Sans recours à la mémoire virtuelle, un processus est entièrement chargé à des adresses contiguës ; avec le recours à la mémoire virtuelle, un processus peut être chargé dans des pages ou des segments non contigus.

11.4.1 La pagination

L'espace d'adressage d'un processus est divisé en petites unités de taille fixe appelées pages. La MC est elle aussi découpée en unités physiques de même taille appelées cadres. Les échanges entre MC et disques ne portent que sur des pages entières. De ce fait, l'espace d'adressage d'un processus est potentiellement illimité (limité à l'espace mémoire total de la machine). On parle alors d'adressage virtuel.

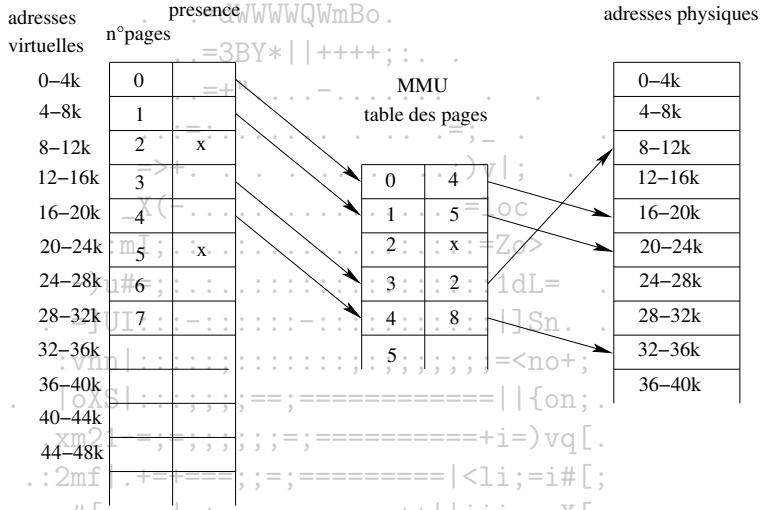
Pour un processus, le système ne chargera que les pages utilisées. Mais la demande de pages à charger peut être plus élevée que le nombre de cadres disponibles. Une gestion de l'allocation des cadres libres est nécessaire.

Dans un SE sans mémoire virtuelle, la machine calcule les adresses physiques en ajoutant le contenu d'un registre de base aux adresses relatives contenues dans les instructions du processus. Dans un SE à pagination, un sous-ensemble inséré entre le CPU et la mémoire, la MMU (Memory Management Unit ou unité de gestion de la mémoire) traduit les adresses virtuelles en adresses physiques.

La MMU mémorise :

– les cadres physiques alloués à des processus (sous forme d'une table de bits de présence)
– les cadres mémoire alloués à chaque page d'un processus (sous forme d'une table des pages)

On dira qu'une page est mappée ou chargée si elle est physiquement présente en mémoire.



Dans l'exemple précédent, les pages ont une taille de 4 Ko. L'adresse virtuelle 12292 correspond à un déplacement de 4 octets dans la page virtuelle 3 (car 12292 = 12288 + 4 et 12288 = 12*1024). La page virtuelle 3 correspond à la page physique 2. L'adresse physique correspond donc à un déplacement de 4 octets dans la page physique 2, soit : (8*1024) + 4 = 8196 .

Par contre, la page virtuelle 2 n'est pas mappée. Une adresse virtuelle comprise entre 8192 et 12287 donnera lieu à un défaut de page. Il y a défaut de page quand il y a un accès à une adresse virtuelle correspondant à une page non mappée. En cas de défaut de page, un déroutement se produit (trap) et le processeur est rendu à l'OS. Le système doit alors effectuer les opérations suivantes :

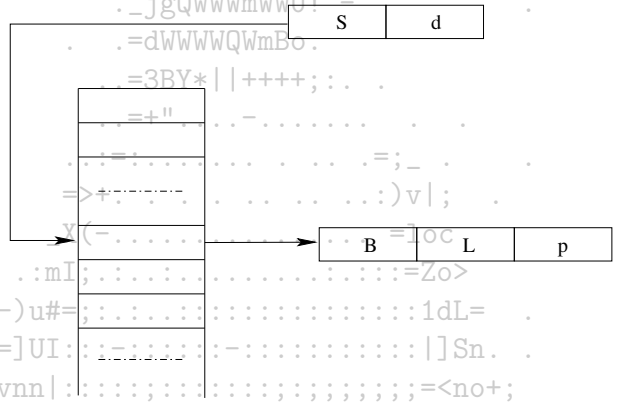
– déterminer la page à charger
– déterminer la page à décharger sur le disque pour libérer un cadre
– lire sur le disque la page à charger
– modifier la table de bits et la table de pages

11.4.2 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 ; une adresse est donnée par un couple (S, d), où S est le numéro du segment et d le déplacement dans le segment, i.e. $d \in [0, L[$.

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

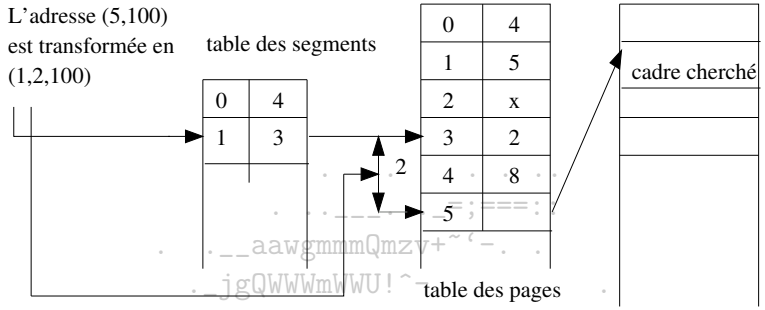
L'adresse physique correspondant à l'adresse virtuelle (S, d) sera donc $B + d$, si $d \leq L$. La segmentation simplifie la gestion des objets communs (rangés chacun dans un segment), notamment si leur taille évolue dynamiquement.

11.4.3 La pagination segmentée

Lorsque le système possède beaucoup de pages, la consultation de la table des pages peut être lente et inefficace s'il y a beaucoup de pages non chargées. La table peut être segmentée alors par processus, pour faciliter la consultation.

Chaque processus possède une table des segments de la table des pages qui le concernent.

Une adresse (P , r) , avec P le numéro logique de page et r le déplacement dans la page, est transformée à la compilation en adresse (S , P' , r), avec S comme numéro d'un segment et P'.

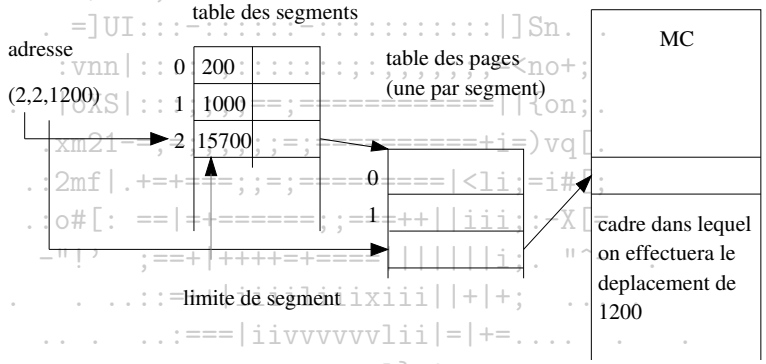


11.4.4 La segmentation paginée

La taille d'un segment peut être importante, d'où un temps de chargement long qui peut en résulter. La pagination des segments peut être une solution.

Une adresse virtuelle (S , d), avec S comme numéro de segment et d déplacement dans le segment, est transformée en (S , P , d'), où P est un numéro de page et d'un déplacement dans la page P.

Par exemple, avec l'hypothèse de pages de 4 Ko, l'adresse logique (2 , 9200) est transformée en (2 , 2, 1200)



Dans tous les cas, l'utilisation d'une mémoire associative, stockant des triplets (S,P,adresse de cadre), gérée comme fenêtre au voisinage du dernier cadre accédé, peut accélérer la recherche des cadres.

11.5 Algorithmes de remplacement de pages

Les méthodes reposent sur 3 stratégies :

- une stratégie de chargement qui choisit les pages à charger et l'instant de chargement
- une stratégie de placement qui choisit un cadre libre pour chaque page à charger
- une stratégie de remplacement destinée à libérer certains cadres. La recherche se fait soit sur l'ensemble des pages (recherche globale), soit sur les pages "appartenant" au processus (recherche locale). Les stratégies de placement sont dépendantes des stratégies de remplacement : on charge nécessairement une page dans le cadre qui vient d'être libéré.

On peut distinguer deux catégories de méthodes :

- pagination anticipée : on charge les pages à l'avance ; mais comment prévoir efficacement ?
- pagination à la demande : le chargement n'a lieu qu'en cas de défaut de page et on ne charge que la page manquante.

On appelle suite de références $w = p_1 p_2 \dots p_n$ une suite de numéros de pages correspondant à des accès à ces pages. Un algorithme A de pagination sur m cadres sera dit stable si la condition suivante est vérifiée :

$$\forall m, \forall w \text{ , Coût(A,m,w)} \leq \text{Coût(A,m+1,w)}$$

Cette notion est importante pour éviter l'écroulement de l'OS (thrashing) par une génération excessive de défauts de pages.

11.5.1 Remplacement de page optimal

L'OS indexe chaque page par le nombre d'instructions qui seront exécutées avant qu'elle ne soit référencée. En cas de nécessité, l'OS retire la page d'indice le plus élevé, c'est à dire la page qui sera référencée dans le futur le plus lointain.

Cet algorithme est pratiquement impossible à appliquer (comment calculer les indices des pages ?). Toutefois, avec un simulateur, on peut évaluer les performances de cet algorithme et s'en servir comme référence pour les suivants. En outre, cet algorithme est stable.

11.5.2 NRU (not recently used)

L'OS référence chaque page par deux bits R (le plus à gauche) et M initialisés à 0. A chaque accès en lecture à une page, R est mis à 1. A chaque accès en écriture, M est mis à 1. A chaque interruption d'horloge, l'OS remet R à 0.

Les bits R-M forment le code d'un index de valeur 0 à 3 en base 10. En cas de défaut de page, on retire une page au hasard dans la catégorie non vide de plus petit index. On retire donc préférentiellement une page modifiée non référencée (index 1) qu'une page très utilisée en consultation (index 2). Cet algorithme est assez efficace.

11.5.3 FIFO (first in, first out)

L'OS indexe chaque page par sa date de chargement et constitue une liste chaînée, la première page de la liste étant la plus anciennement chargée et la dernière la plus récemment chargée. L'OS replacera en cas de nécessité la page en tête de la liste et chargera la nouvelle page en fin de liste.

Deux critiques à cet algorithme :

– ce n'est pas parce qu'une page est la plus ancienne en mémoire qu'elle est celle dont on se sert le moins

– l'algorithme n'est pas stable : quand le nombre de cadres augmente, le nombre de défauts de pages ne diminue pas nécessairement (anomalie de BELADY : L.A. BELADY a proposé en 1969 un exemple à 4 cadres montrant qu'on avait plus de défaut de pages qu'avec 3).

Amélioration 1 : l'OS examine les bits R et M (générés comme ci-dessus) de la page la plus ancienne en MC (en tête de la liste). Si cette page est de catégorie 0, il l'ôte. Sinon, il teste la page un peu moins ancienne, etc. S'il n'y a aucune page de catégorie 0, il recommence avec la catégorie 1, puis éventuellement avec les catégories 2 et 3.

Amélioration 2 : Algorithme de la seconde chance : R et M sont gérés comme ci-dessus. L'OS examine le bit R de la page la plus ancienne (tête de la liste). Si R = 0, la page est remplacée, sinon R est mis à 0, la page est placée en fin de liste et on recommence avec la nouvelle page en tête. Si toutes les pages ont été référencées depuis la dernière RAZ, on revient à FIFO, mais avec un surcoût.

11.5.4 LRU (least recently used)

En cas de nécessité, l'OS retire la page la moins récemment référencée. Pour cela, il 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.

L'algorithme est stable. Mais il nécessite une gestion coûteuse de la liste qui est modifiée à chaque accès à une page.

Variantes : NFU (not frequently used) ou LFU (least frequently used) : l'OS gère pour chaque page un compteur de nombre de références . Il cumule dans ce compteur le bit R juste avant chaque RAZ de R au top d'horloge. Il remplacera la page qui aura la plus petite valeur de compteur. Inconvénient : une page qui a été fortement référencée, mais qui n'est plus utilisée ne sera pas remplacée avant longtemps.

Vieillisement (aging) ou NFU modifié : avant chaque RAZ de R, l'OS décale le compteur de 1 bit à droite (division entière par 2) et additionne R au bit de poids fort. En cas de nécessité, on ôtera la page de compteur le plus petit. Mais en cas d'égalité des compteurs, on ne sait pas quelle est la page la moins récemment utilisée.

Matrice : s'il y a N pages, l'OS gère une matrice $(N, N+1)$. A chaque référence à la page p, l'OS positionne à 1 tous les bits de la ligne p et à 0 tous les bits de la colonne p. En outre, le 1er bit de la ligne p vaut 0 si la page est chargée, 1 sinon. On ôtera la page dont la ligne a la plus petite valeur.

11.5.5 Améliorations

Le dérobeur de pages : dès qu'un seuil minimal de cadres libres est atteint, l'OS réveille un dérobeur de pages . Celui-ci supprime les pages les moins récemment utilisées, par algorithme d'aging, et les range dans la liste des cadres libres, ceci jusqu'à un seuil donné. L'OS vérifie qu'une page référencée sur

défaut de page n'est pas dans la liste.

Problèmes d'entrée/sortie : on a intérêt, soit à verrouiller les pages concernées par les E/S pour éviter de les ôter, soit à effectuer les E/S dans des tampons du noyau et à copier les données plus tard dans les pages des utilisateurs.

Pages partagées : lorsque plusieurs processus exécutent le même ensemble de code, on a intérêt à utiliser des pages de code partagées, plutôt que des pages dupliquées. Mais on aura à prévoir une gestion spécifique des pages partagées pour éviter des défauts de pages artificiels.

11.5.6 Taille optimale des pages

Soit t la taille moyenne des processus et p (en octets) la taille d'une page. Soit e (en octets) la taille de l'entrée de chaque page dans la table des processus. Le nombre moyen de pages par processus est t/p. Ces pages occupent $t \times e/p$ octets dans la table des processus. La mémoire perdue en moyenne dans la dernière page d'un processus est $p/2$ (fragmentation interne). La perte totale q est donc :

$$q = t \times e/p + p/2$$

q est minimal si $\frac{dq}{dp} = 0$, c'est à dire : $p = (2t * e)^{0.5}$

Exemple : t = 64 Ko, e = 8 octets, donc p = 1 Ko Les valeurs courantes sont 1/2 Ko, 1 Ko, 2 Ko ou 4 Ko

11.5.7 Exemples d'application

Retour sur le coût d'une méthode

À une suite d'adresses logiques référencées $a_1 a_2 \ldots a_n$ correspond une suite de références de pages $p_1 p_2 \ldots p_n$, elle-même associée une suite d'états de la mémoire principale $e_1 e_2 \ldots e_n$, états définis par les affectations des pages aux cadres de la mémoire. Un algorithme de pagination définit une fonction de transition, qui, a un état e_i et une référence p_{i+1} fait correspondre le nouvel état e_{i+1}. Lors de la transition, il y a un certain nombre (≥ 0) de pages déchargées et un nombre (≥ 0) de pages chargées par le système. Ces opérations engendrent un coût qui est fonction du temps nécessaire aux opérations de déchargement et de chargement. Remarquons que les pages chargées sont, en moyenne, rarement modifiées. Pour les pages intactes, il n'est pas nécessaire de les reporter vers la mémoire secondaire lors que leur déchargement, car il existe déjà des copies de ces pages. Ainsi le temps des opérations de déchargement est souvent négligeable, comparé à celui du chargement. Le coût d'une transition peut donc être approché en ne tenant compte que du temps de chargement des pages.

Le temps nécessaire au chargement d'une page fait intervenir :

– le temps d'attente correspondant à l'attente dans la file du périphérique d'I/O, le temps de recherche sur le prériphérique et le temps de latence
– la durée de transfert de la page

Il est évident que d'une adresse de page à l'autre, des variations sur ces quantités existent. Afin de simplifier le calcul du coût des opérations de chargement, il est considéré que le chargement d'une page est indépendant de son adresse. Le coût est modélisé de telle sorte qu'il ne dépende que du nombre de pages à charger : ci-dessous, notre fonction de coût de transition est noté c et a pour unique argument x, le nombre de pages. Ce coût n'est à priori pas linéaire[1].

Considérons un algorithme opérant sur une suite de références dans une mémoire de taille m. Le coût de cet algorithme peut être défini comme la somme des coûts des transitions permettant de passer de l'état initial à l'état final. C'est cette somme qui permet d'étudier la stabilité d'un algorithme de pagination.

Nous nous intéresserons principalement aux méthodes de pagination à la demande, avec la fonction de coût pour les transitions définie comme suit :

$$c(x) = \begin{cases} 0 & , \quad \text{si } x = 0 \\ 1 & , \quad \text{si } x = 1 \end{cases}$$

Considérons la suite d'adresses logiques suivantes : 00011, 00101, 01000, 01111, 00010, 00100, 10001, 00010, 00101, 01011, 01100, 10010. Sachant qu'une page est de taille 4, déterminez la suite de références de pages associées à cette suite d'adresses.

Gestion FIFO

Déterminez pour la méthode d'allocation FIFO la suite d'états de la mémoire sachant qu'elle permet d'accueillir trois pages. À l'état initial, les pages de la mémoire sont vides. Quel est le coût de la méthode pour cette configuration. Refaites vos calculs pour une mémoire augmentée d'une page. Quelle observation faites-vous ?

Gestion par allocation optimale

Mêmes questions.

Gestion par allocation LRU

Mêmes questions, en effectuant la gestion du temps de façon libre, puis en utilisant la représentation matricielle décrite plus haut.

[1] Cette remarque est importante pour les méthodes de pagination par anticipation.

11.6 Le problème d'écroulement d'un système (trashing)

11.6.1 Présentation du problème

Dans un système multiprogrammé, un des buts est de maximiser le taux d'utilisation du CPU. Les premier systèmes multiprogrammés ont révélé le risque d'écroulement du système lorsque le degré de multiprogrammation est trop poussé. Le scenario classique conduisant à cette situation est le suivant : l'OS, qui surveille le taux d'utilisation du CPU, juge ce taux améliorable. Pour l'augmenter, il introduit un nouveau processus et lui alloue des cadres en retirant des cadres à d'autres processus. Si le nouveau processus réclame d'autres cadres, des défauts de pages sont engendrés, entrainant des échanges entre mémoire centrale et mémoire secondaire. Les processus à qui les cadres ont été retirés ont eux aussi besoin de cadres, entrainant davantages d'échanges. Ainsi, le taux d'utilisation du CPU diminue, et l'OS peut alors décider d'injecter un nouveau processus, agravant encore la concurrence des processus sur

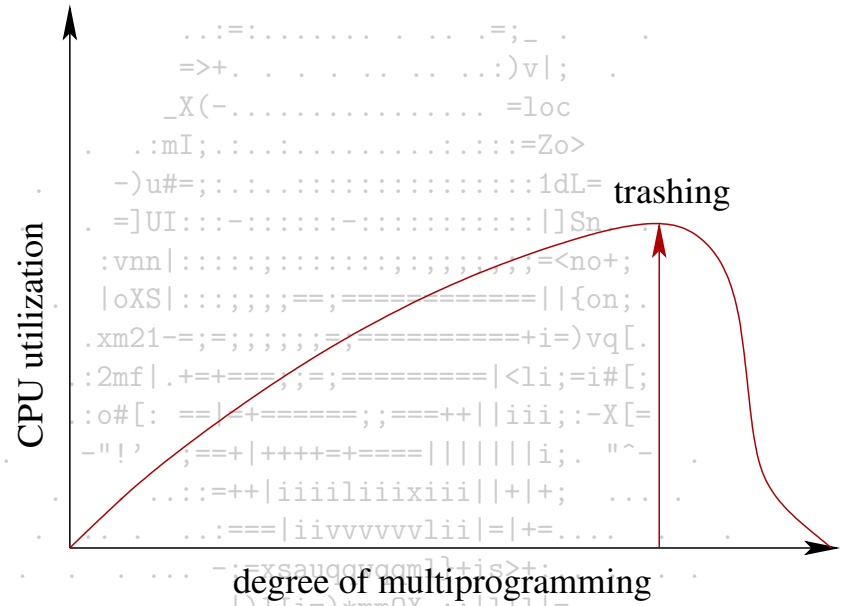


FIG. 11.2 – Trashing

les cadres. Le taux d'utilisation du CPU par les processus s'écroule et le système se noie dans la gestion des cadres.

Pour éviter ce phénomène, certains OSs utilisent le modèle du working-set, basé sur le principe de localité. Selon le principe de localité, les adresses référencées par un processus ne sont pas réparties uniformément, mais tendent à se regrouper à la fois dans le temps et dans l'espace :

– Si un ensemble d'adresses a été référencé dans un passé immédiat, la probabilité pour que les adresses de cet ensemble soient référencées dans un futur immédiat est grande. C'est la localité dans le temps. La présence de boucle favorise ce type de localité.
– Sur un intervalle de temps moyen, les références portent en majorité sur des adresses voisines.

11.6.2 Le modèle du Working Set

Le modèle working-set est une tentative proposée par P.J. Denning en 1968 pour comprendre les performances d'un système paginé en multiprogrammation. Il s'agit de raisonner non pas en terme de processus isolés, mais d'effet des autres processus présents sur chacun d'eux, de corréler allocation de mémoire et allocation du processeur.

Conceptuellement, le working-set, ou ensemble de travail, peut être défini de la façon suivante :

Soit Δ un entier supérieur ou égal à un et $w = p_1 p_2 \dots p_n$ une suite de références. Pour $k = 1, 2, \dots n$, l'ensemble de travail, noté $W(\Delta, k)$, est défini comme l'ensemble des pages distinctes référencées dans la suite partielle de références $p_{k-\Delta+1} p_{k-\Delta+2} \dots p_{k-1} p_k$.

Denning a montré que la courbe de la fonction $|W(., k)|$ avait un graphe tel que celui-ci :

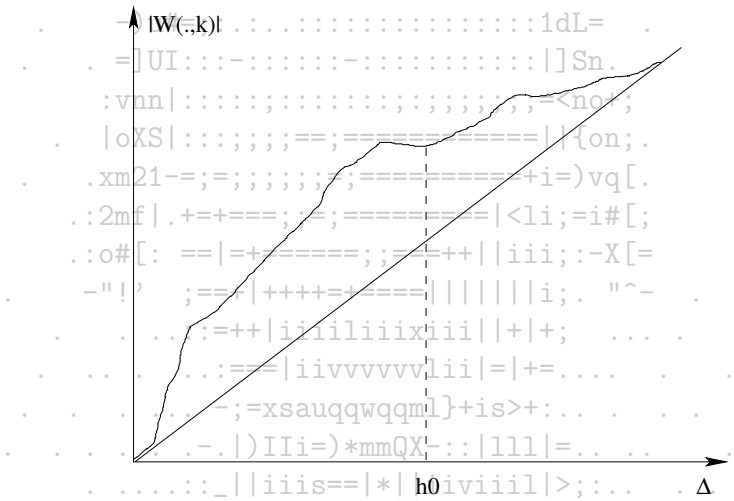


FIG. 11.3 – Evolution de la taille du working-set

Plus Δ croît et moins on trouve de pages en excédent dans le working-set. Cela permet d'établir une valeur raisonnable de h (h0) tel qu'une valeur de h supérieure à h0 n'accroîtrait pas beaucoup le working-set.

L'ensemble de travail peut être considéré comme une fenêtre faisant apparaître les Δ dernières références. L'entier Δ est appelé la taille de la fenêtre. Si Δ est trop petit, aucun profit ne sera tiré des possibles redondances dans les références. En revanche, si Δ est assez grand, il permettra de tirer profit de références déjà présentes en mémoire et travailler sans défaut de pages jusqu'à la disparition de la redondance. Supposons que la taille du working-set de chaque processus soit connue à un instant donné. La demande totale du nombre de cadres est alors connue à cet instant. Si cette demande de cadre dépasse le nombre de cadres présents en mémoire, le système s'oriente vers une situation d'écroulement.

La gestion des pages repose alors sur le principe suivant : l'unité centrale n'est attribuée qu'à un processus dont l'ensemble de travail est entièrement chargé en mémoire centrale. En conséquence, aucune page de cet ensemble n'est choisie comme victime pour un remplacement. Si la mémoire centrale est entièrement occupée par des ensembles de travail et qu'un remplacement est nécessaire, l'OS suspend un des processus. Toutefois, cette stratégie est maladroite et lourde pour prévenir de l'effondrement d'un système.

11.6.3 Contrôle du taux de défaut de page

Une approche plus directe consiste à contrôler la fréquence des défauts de pages. L'écroulement est marqué d'une fréquence de défauts de pages très élevée. Cet indicateur peut être utilisé pour contrôler l'entrée de nouveaux processus dans le système : si le taux des défauts est trop élevé, non seulement, il n'est pas raisonnable d'autoriser un nouveau processus, mais il est souhaitable d'en retirer un provisoirement afin d'augmenter le nombre de cadres par processus. Si, au contraire, le taux est faible, on peut penser que les cadres sont sous-exploités et que l'introduction d'un nouveau processus permettrait une meilleure exploitation par une redistribution de certains cadres.

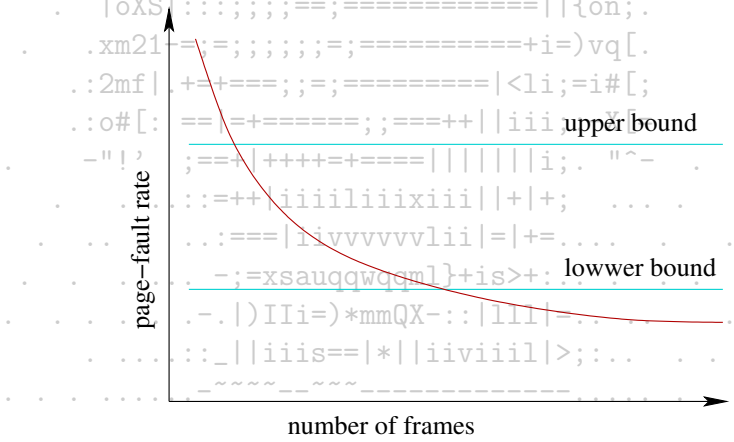


Fig. 11.4 – Fréquence des défauts de pages

Chapitre 12

Exemples de gestion

12.1 Exemple : gestion de la memoire par Windows NT

Avec les adresses exprimées sur 32 bits, Windows NT offre à chaque processus 2^{32} octets = 4 Go de mémoire virtuelle : 2 Go pour l'OS (threads en mode noyau) et 2 Go pour les programmes (threads en mode utilisateur et en mode noyau) paginés en pages de 4 Ko.

Windows NT protège sa mémoire grâce aux mécanismes suivants :

– espace d'adressage virtuel différent pour chaque processus
– fonctionnement en mode noyau pour l'accès au code et aux données du système

12.1.1 Le gestionnaire de la mémoire virtuelle

Il optimise l'utilisation de la mémoire selon plusieurs techniques :

– évaluation différée : plusieurs processus peuvent accéder aux mêmes données en lecture, sans les dupliquer. Si l'un des processus veut modifier des données, la page physique est recopiée ailleurs dans la mémoire ; l'espace d'adressage virtuel du processus est mis à jour
– protection de mémoire par page : chaque page contient le mode d'accès autorisé en mode noyau et en mode utilisateur (lecture, lecture/écriture, exécution, pas d'accès, copie à l'écriture, page de garde)

Le gestionnaire de la mémoire virtuelle fournit des services natifs aux processus :

– lecture/écriture en mémoire virtuelle
– blocage des pages virtuelles en mémoire physique
– protection des pages virtuelles

– déplacement des pages virtuelles sur disque
– allocation de la mémoire en deux phases : réservation (ensemble d'adresses réservées pour un processus) et engagement (mémoire retenue dans le fichier d'échange avec le disque du gestionnaire de la mémoire virtuelle)

Le gestionnaire de pages utilise des techniques classiques :
– pagination à la demande : à la suite d'un défaut de pages, on charge les pages requises plus quelques pages voisines
– placement des pages virtuelles en mémoire : à la première page physique disponible
– remplacement selon la stratégie FIFO d'une page virtuelle d'un processus, suite à un défaut de page
– table des pages à deux niveaux

En outre, lorsqu'une page est requise suite à un défaut de page, et qu'elle est non utilisée, un indicateur de transition permet de déceler si elle peut devenir utilisable sans la faire transiter par le disque.

Toute page physique peut prendre l'un des 6 états suivants utilisés dans la base de données des pages physiques :
– valide : page utilisée par un processus
– à zéro : libre et initialisée avec des 0 pour des raisons de sécurité ("niveau C2")
– modifiée : son contenu a été modifié et n'a pas encore été recopié sur disque
– mauvais : elle a généré des erreurs au contrôle
– en attente : retirée d'un ensemble de pages de travail

Le gestionnaire des pages étant réentrant, il utilise un verrou unique pour protéger la base de données des pages physiques et n'en autoriser l'accès qu'à une tâche (thread) simultanée.

12.1.2 La mémoire partagée

Définition : c'est de la mémoire visible par plusieurs processus ou présente dans plusieurs espaces d'adressage virtuels

Windows NT utilise des objets-sections pour utiliser des vues de fenêtres de la zone de mémoire partagée à partir de différents processus. La mémoire partagée est généralement paginée sur disque.

12.2 La pagination dans le systeme Linux

Dans le système Linux, la gestion mémoire est conçue selon un modèle de pagination à trois niveaux reposant sur les tables suivantes :
– la table globale (page global directory) : elle contient les numéros des pages contenant des tables intermédiaires ;
– la table intermédiaire (page middle directory) : elle contient les numéros des pages contenant des tables des pages ;
– la table des pages (page table) : elle contient les adresses des pages mémoires contenant le code ou les données utilises par les processus.

Ce modèle est réduit à une pagination a deux niveaux sur les machines basées sur la famille de processeurs Pentium, en faisant disparaître les tables intermédiaires[1].

En effet, ces processeurs, qui possèdent un adressage sur 32 bits, peuvent adresser un espace virtuel de 4 Go. Avec des pages mémoires de 4 Ko, la table des pages, dans une pagination à un niveau, contiendrait 220 entrées. Pour éviter une aussi grande table des pages, la mémoire est gérée avec une pagination à deux niveaux.

Comme on trouve dans une table des pages 1 024 entrées de 4 octets, une table de pages tiendra exactement dans une page mémoire et décrit un espace de 1 024 pages de 4 Ko, soit 4 Mo. Ainsi un programme d'une taille inférieure ou égale à N x 4 Mo nécessitera N tables de pages[2]. Mais un programme chargé en mémoire n'aura pas nécessairement ses N tables en mémoire : en effet, il lui suffit pour s'exécuter, d'avoir sa table des hyperpages et la table des pages correspondant aux pages en cours d'utilisation. Le bit de présence P dans l'entrée k de la table des hyperpages indique si la table des pages k est chargée en mémoire ou non[3].

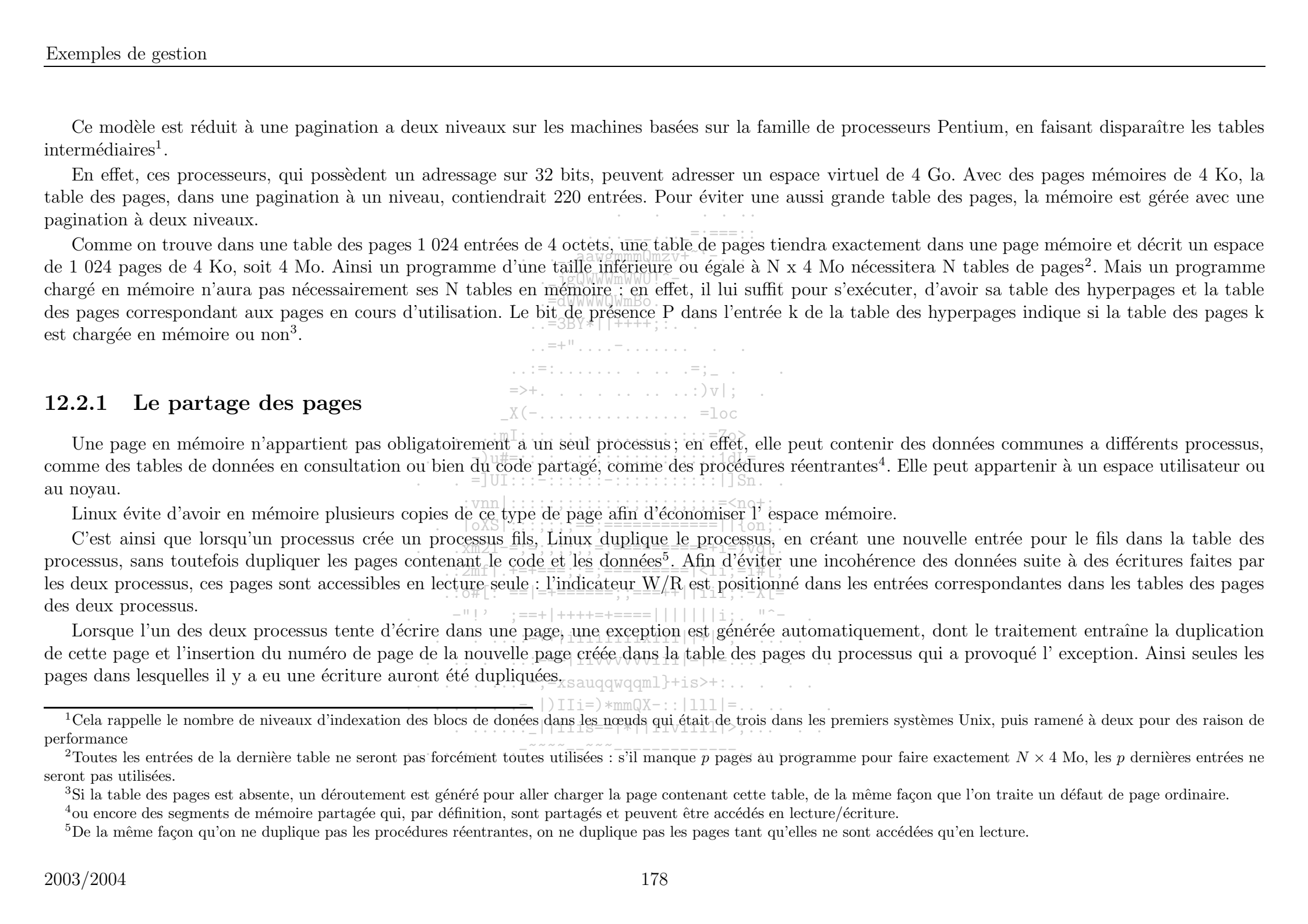
12.2.1 Le partage des pages

Une page en mémoire n'appartient pas obligatoirement à un seul processus ; en effet, elle peut contenir des données communes a différents processus, comme des tables de données en consultation ou bien du code partagé, comme des procédures réentrantes[4]. Elle peut appartenir à un espace utilisateur ou au noyau.

Linux évite d'avoir en mémoire plusieurs copies de ce type de page afin d'économiser l' espace mémoire.

C'est ainsi que lorsqu'un processus crée un processus fils, Linux duplique le processus, en créant une nouvelle entrée pour le fils dans la table des processus, sans toutefois dupliquer les pages contenant le code et les données[5]. Afin d'éviter une incohérence des données suite à des écritures faites par les deux processus, ces pages sont accessibles en lecture seule : l'indicateur W/R est positionné dans les entrées correspondantes dans les tables des pages des deux processus.

Lorsque l'un des deux processus tente d'écrire dans une page, une exception est générée automatiquement, dont le traitement entraîne la duplication de cette page et l'insertion du numéro de page de la nouvelle page créée dans la table des pages du processus qui a provoqué l' exception. Ainsi seules les pages dans lesquelles il y a eu une écriture auront été dupliquées.



[1] Cela rappelle le nombre de niveaux d'indexation des blocs de données dans les nœuds qui était de trois dans les premiers systèmes Unix, puis ramené à deux pour des raison de performance

[2] Toutes les entrées de la dernière table ne seront pas forcément toutes utilisées : s'il manque p pages au programme pour faire exactement $N \times 4$ Mo, les p dernières entrées ne seront pas utilisées.

[3] Si la table des pages est absente, un déroutement est généré pour aller charger la page contenant cette table, de la même façon que l'on traite un défaut de page ordinaire.

[4] ou encore des segments de mémoire partagée qui, par définition, sont partagés et peuvent être accédés en lecture/écriture.

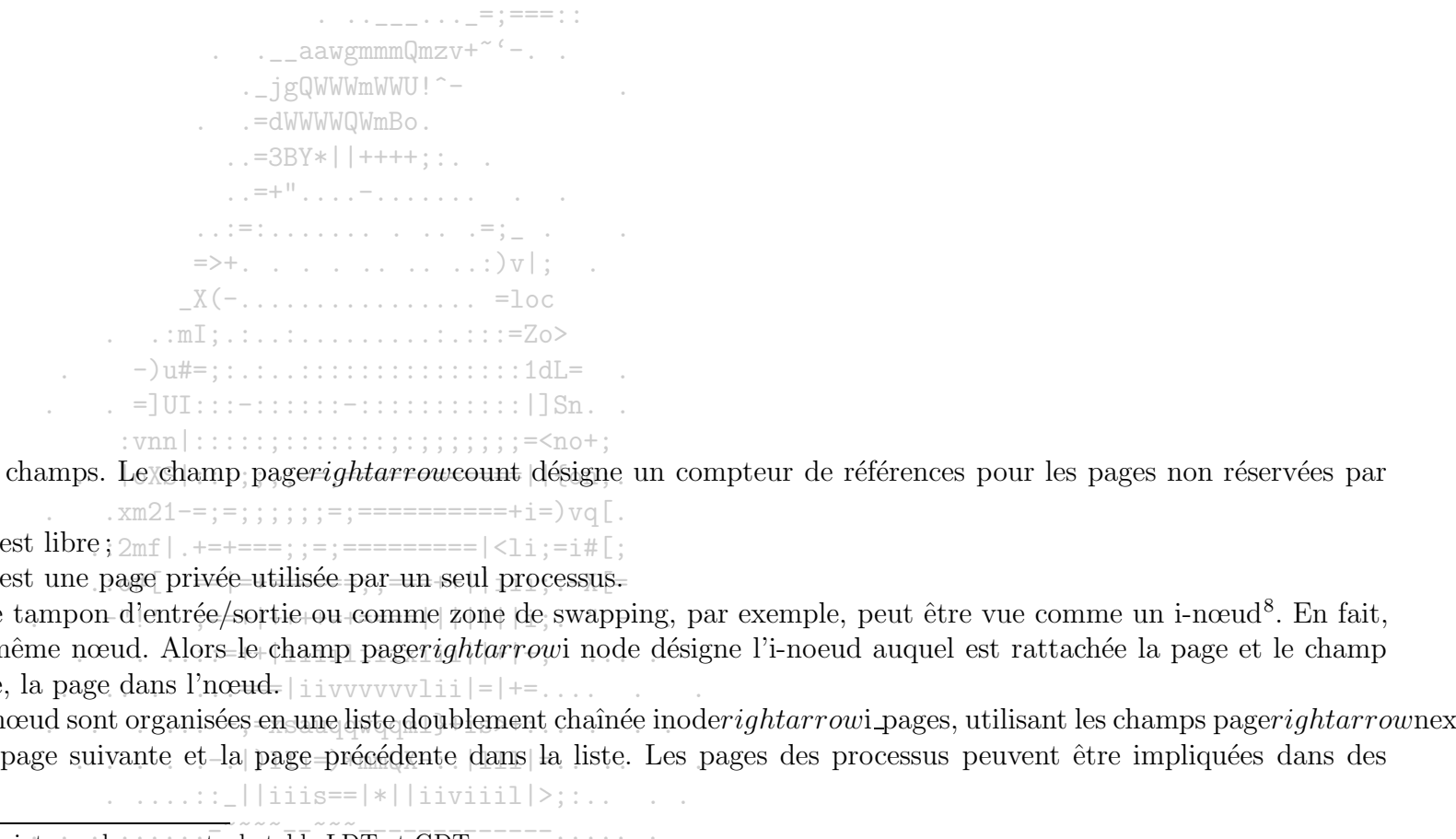
[5] De la même façon qu'on ne duplique pas les procédures réentrantes, on ne duplique pas les pages tant qu'elles ne sont accédées qu'en lecture.

12.2.2 Les descripteurs de pages

Linux tient a jour un tableau de descripteurs[6], pointe par la variable mem_map[7], décrivant chacun une page de la mémoire centrale. La constante PAGESIZE définie dans <limits. h> donne la taille d'une page mémoire. Dans les architectures Intel, cette taille est de 4 Ko.

La structure page, déclarée dans le fichier <linux/mm. h>, définit les champs d'un descripteur.

```
typedef struct page {
struct page *next;
struct page *prev;
struct inode *inode;
unsigned long offset;
struct page *next_hash;
atomic_t count;
unsigned long flags;
struct wait_queue *wait;
struct page **pprev_hash;
struct buffer_head *buffers;
} mem_map_t;
```



Voyons la signification de ces différents champs. Le champ page*rightarrow*count désigne un compteur de références pour les pages non réservées par les processus :

- page→count = 0 signifie que la page est libre ;
- page→count = 1 signifie que la page est une page privée utilisée par un seul processus.

Une page utilisée par le système comme tampon d'entrée/sortie ou comme zone de swapping, par exemple, peut être vue comme un i-nœud[8]. En fait, plusieurs pages peuvent être associées au même nœud. Alors le champ page*rightarrow*i node désigne l'i-noeud auquel est rattachée la page et le champ page*rightarrow*offset est le déplacement de, la page dans l'nœud.

Toutes les pages qui appartiennent a un nœud sont organisées en une liste doublement chaînée inode*rightarrow*i_pages, utilisant les champs page*rightarrow*nex et page*rightarrow*prev pour référencer la page suivante et la page précédente dans la liste. Les pages des processus peuvent être impliquées dans des opérations d' entrées/sorties :

[6]Ne pas confondre ces descripteurs avec les descripteurs de segments de table LDT et GDT.

[7]Déclarée dans le fichier source `mm/memory.c`.

[8]Tout comme un fichier ou un périphérique. Un fichier, un périphérique, un répertoire, un tube simlpe ou nommé, un socket sont des objets Unix sur lesquels on fait des opérations de lecture, écriture. Chacun est désigné par un numéro dans la table des nœuds ; un nœud, comme index i-node, étant une structure regroupant les informations décrivant l'objet en question et des pointeurs sur des données.

– les pages des nœuds peuvent nécessiter des lectures a partir du disque ;
– les pages des nœuds qui ont été modifiées et qui sont partagées (MAP_SHARED) nécessitent d'être écrites sur le disque ;
– les pages privées qui ont été modifiées nécessitent d'être recopiées sur le disque (swapped out)

Le champ page*rightarrow*wait est une file d'attente de tous les processus qui attendent la fin d'une E/S sur cette page.

Le champ flag contient des informations sur l' accessibilité de la page, reprises pour certaines dans le tableau suivant :

Symbole binaire	Position	Signification
PG_locked	0	La page est verrouillée en mémoire pour cause d'E/S dans la page
PG_error	1	Une E/S faite dans la page s'est terminée en erreur
PG_referenced	2	La page a été référencée
PG_dirty	3	La page a été modifiée
PG_uptodate	4	Le contenu de la page est valide (à jour)
PG_reserved	31	La page est réservée

Le bit PG_referenced est utilisé par l'algorithme de remplacement basé sur technique LRU lors du remplacement d'une page.

12.2.3 Opérations sur les pages mémoires Linux

La prise en charge de la pagination par le système est transparente pour l'utilisateur qui continue à voir un espace adressable linéaire et travaille sur des zones memoire qui peuvent être contenues dans une ou plusieurs pages, et pourquoi pas recouvrir exactement une page.

Projection en mémoire

La projection du contenu d'un fichier vers la mémoire peut être faite par la primitive mmap.

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t len, int prot, Int flags, int fd, off_t offset);
```

projette len octets à partir de la position offset d'un fichier spécifié par le descripteur fd en mémoire dans l'espace d'adressage du processus courant, de préférence à l'adresse mémoire start. En général start est mis à NULL car il sert uniquement d'indication au système, qui peut utiliser une autre adresse de début de page.

L'adresse réelle où sera projeté le fichier est retournée par la primitive mmap. Le paramètre prot définit les protections mémoire à appliquer sur cette zone mémoire et prend les mêmes valeurs que dans la primitive mprotect(). flag spécifie les modalités de protection. start doit correspondre à un début de page et offset à un début de bloc. mmap retourne un pointeur sur la zone de projection. En cas d'échec, La valeur MAP_FAILED (-1) est retournée avec positionnement de errno.

```
#include <unistd.h>
#include <sys/man.h>
void *munmap(void *start, size_t len);
```

supprime la projection en mémoire d'un fichier pointée par l'adresse start. start est l'adresse début de zone mémoire retournée par mmap et de longueur len. En cas de succès, la primitive retourne 0, et retourne -1 en cas d'erreur. errno contient la valeur EINVAL.

```
#include <unistd.h>
#include <sys/mman.h>
void *mremap(void *old_start. size_t old_len. size_t new_len. unsigned long flags);
```

modifie (étend ou réduit) la taille d'une zone mémoire déjà projetée. Ceci petit entraîner un déplacement de la zone mémoire, notamment si celle-ci devient plus grande. old_start est l'adresse de la zone initiale qui est une adresse de début de page et old_len est la taille de cette zone qui est égale à la taille d'un bloc. new_len est la nouvelle taille de ce bloc. flags est égal a l'option MREMAP_MAYMOVE qui indique que le noyau est autorisé à modifier l'adresse début de la zone si le redimensionnement de la zone ne peut se faire a l'ancienne adresse. mremap retoume un pointeur sur la nouvelle zone mémoire qui peut être différent de old_start. En cas d'échec, la valeur -1 est retournée avec le positionnement de errno a la valeur appropriée.

Protection de pages mémoire

Un processus peut définir une zone mémoire en lui associant des droits d' accès grâce à la primitive mprotect.

```
#include <sys/mman.h>
int mprotect (caddr_t addr, size_t len. int prot);
```

- addr : est l'adresse de la zone mémoire dont il faut modifier les droits d'accès
- len : est la taille de la zone en octets ;
- prot : définit le type d'accès comme indique dans le tableau ci-dessous ;

La fonction retourne 0 en cas de succès ou la valeur -1 en cas d'erreur. Dans ce dernier cas, la variable errno peut prendre les valeurs suivantes :

Si un processus essaie de faire tine opération non autorisée sur la zone mémoire, par exemple écrire dans une zone accessible en lecture seule, alors le signal SIGSEGV est envoyé au processus, entraînant, par défaut, sa terminaison.

Valeur de PROT	Signification
PROT_NONE	La zone est marquée comme inaccessible
PROT_READ	La zone est marquée comme accessible en lecture
PROT_WRITE	La zone est marquée comme accessible en écriture
PROT_EXEC	La zone peut contenir du code (elle est exécutable)

Verrouillage de pages mémoire

Un processus privilégié peut verrouiller et déverrouiller des pages mémoire.

```
#include <sys/mman.h>
int mlock (const void *addr. size_t len);
```

permet de verrouiller une zone en mémoire d'adresse addr et de longueur len (en octets). Toutes les pages qui contiennent une partie de cette zone restent résidentes en mémoire (ne seront pas transférées en mémoire secondaire) jusqu'au déverrouillage de cette zone par munlock ou munlockall, ou bien jusqu'à ce que le processus ayant fait le verrouillage soit terminé ou ait demarré un autre programme avec la primitive exec...Les processus fils créés à l'aide de la primitive fork n'héritent pas des verrous des pages.

Le verrouillage mémoire peut être utilisé par les algorithmes d'ordonnancement temps réel. En effet, verrouiller une page c'est garantir que la page reste en mémoire, donc le processus qui la contient reste en mémoire, augmentant ses chances de se terminer dans les délais, les sauvegardes temporaires sur disque étant évitées. Les pages qui sont verrouillées plusieurs fois par des appels à munlock ou munlockall sont déverrouillées par un seul appel à munlock ou munlockall.

```
#include <sys/rnrnan.h>
int munlock (const void *addr. size_t len);
```

permet de déverrouiller une zone mémoire d'adresse addr et de longueur len (en octets).

```
#include <sys/mman.h>
int mlockall (const void *addr. size_t len);
```

permet de verrouiller toutes les pages mémoire (pages de code, de données et de pile, les bibliothèques partagées, les segments de mémoire partagée et les fichiers projetés en mémoire) appartenant à l'espace d'adressage du processus appelant. Si la primitive s'est exécutée avec succès, alors on garantit que toutes ces pages restent résidentes en mémoire (ne seront pas transférées en mémoire secondaire) jusqu'au déverrouillage par munlockall ou bien jusqu'à

Valeur de flag	Signification
MCL_CURRENT	Vérouille toutes les pages chargées en mémoire dans l'espace d'adressage du processus courant
MCL_FUTURE	Verrouillera (dans le futur) toutes les pages qui seront chargées en mémoire dans l'espace d'adressage du processus courant. Ces pages peuvent concerner de nouvelles pages qui résultent d'une extension du tas ou de la pile du processus ou bien de l'espace mémoire nécessaire au chargement de nouveaux fichiers.

ce que le processus ait terminé ou démarré un autre programme avec la primitive exec. . .Les processus fils créés à l' aide de la primitive fork n'héritent pas des verrous des pages.

flag peut prendre les valeurs suivantes : Si MCL_FUTURE est spécifié et que le nombre de pages verrouillées dépasse le nombre autorise de pages à verrouiller, alors l'appel échoue et une erreur du type ENOMEM est retournée. Si ces nouvelles pages correspondent a l'extension de la pile, alors le système refuse cette extension et envoie un signal du type SIGSEGV.

```
#include <sys/mman.h>
int munlockall (const void *addr. size_t len);
```

permet de déverrouiller tout l'espace d'adressage du processus courant. Elle annule l'effet de plusieurs appels à mlock et/ou mlockall.

Synchronisation de pages mémoire

```
#include <unist. h>
#include <sys/mman.h>
int msync (const void *start size_t len int flags)
```

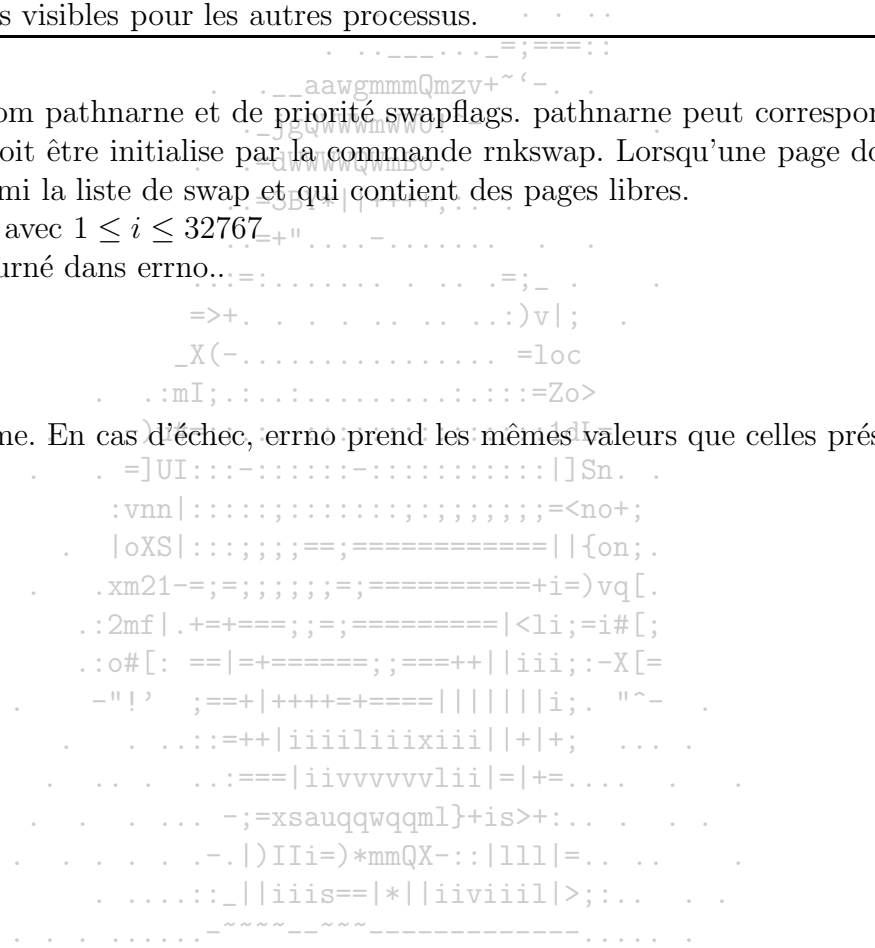
permet de réécrire sur disque des modifications effectuées dans la zone mémoire qui contient un fichier projeté. Autrement dit, la partie du fichier correspondant a la zone qui commence à l'adresse start et de longueur len est mise a jour. Sans l'appel à cette primitive, il n'y aura aucune garantie que les modifications soient portées sur le disque avant l'appel à munmap(). La projection en mémoire doit être faite au préalable avec l'option MAP_SHARED.

msync retourne 0 en cas de succès et -1 sinon.

Gestion des périphériques de swap

```
#include <unistd.h>
\#incl ude<linux/swap.h>
int swapon (const char *pathname int swapflags)
```

Valeur de flags	Signification
MS_SYNC	La mise a jour est asynchrone : la sortie de la primitive n'a lieu que lorsque les modifications sont réellement portées sur le disque
MS_ASYNC	La mise à jour est asynchrone : la sortie de la primitive n'attend pas la mise à jour sur disque
MS_INVALIDATE	On invalide les autres projections du fichier, de manière à provoquer la relecture des données des qu'un processus y accède. Les mises à jour sont alors visibles pour les autres processus.



permet l'activation du périphérique de nom pathnarne et de priorité swapflags. pathnarne peut correspondre à un périphérique géré en mode bloc ou à un fichier ordinaire. Dans tous les cas, il doit être initialise par la commande rnkswap. Lorsqu'une page doit être sauvegardée (swappée), Linux choisit le périphérique qui est le plus prioritaire parmi la liste de swap et qui contient des pages libres.

swapflags = SWAP_FLAG_PREFER + i avec $1 \leq i \leq 32767$

En cas d'échec, un code d'erreur est retourné dans errno..

```
#include <unistd. h>
int swapoff(const char *pathname);
```

désactive le périphérique nommé pathname. En cas d'échec, errno prend les mêmes valeurs que celles présentées dans la primitive swapon.

Index

Bibliographie

[AIV02] Tigran AIVAZIAN. Linux kernel 2.4 internals. available at `www.mc.man.ac.uk/LDP/LDP/lkii/lki.html`, August 2002.

[BA86] M. BEN-ARI. Processus concurrents. Masson, 1986.

[BB93] J. BEAUQUIER and B. BERARD. Systèmes d'exploitation. Schaum's Montreal et Ediscience Paris, 1993.

[BK95] Morris I. BOLSKY and David G. KORN. The new Kornshell - Command and programming language. Prentice Hall PTR, 1995.

[FAS01] Jean-Philippe FASSINO. THINK : vers une architecture de systèmes flexibles. PhD thesis, École nationale supérieure des télecommunications, décembre 2001.

[KRA87] Sacha KRAKOWIAK. Principes des systèmes d'exploitation des ordinateurs. Dunod informatique, 1987.

[REV98] D. REVUZ. Cours système. available at `www-igm.univ-mlv.fr/~dr/CS/`, novembre 1998.

[RUS99] David A. RUSLING. The linux kernel. available at `www.tldp.org/guides.html`, January 25 1999.

[SFR98] W. Richard STEVENS, Bill FENNER, and Andrew M. RUDOFF. UNIX network programming - The sockets networking API, volume 1. Addison Wesley, third edition, 1998.

[SGG03] Abraham SILBERSCHATZ, Peter B. GALVIN, and Greg GAGNE. Operating system concepts. John Wiley & Sons, Inc, six edition, 2003.

[STE93] W. Richard STEVENS. Advanced programming in the UNIX network environnement. Addison-Wesley, 1993.

[STE99] W. Richard STEVENS. UNIX network programming - Interprocess communication, volume 2. Prentice Hall PTR, second edition, 1999.

[TAN99] A. S. TANENBAUM. Structured computer organization. Prentice-Hall international, fourth edition, 1999.

[TW97] A. S. TANENBAUM and A.S. WOODHULL. Operating Systems - Design and implementation (2nd ed.). Pearson education, 1997.