

Précis de système d'exploitation

Département
Informatique et Mathématiques appliquées
E.N.S.E.E.I.H.T.

Gérard Padiou

16 juillet 2012

Préambule

Le document suivant n'est qu'un **résumé très condensé** des principes de conception et des mécanismes mis en œuvre dans les systèmes d'exploitation. La connaissance approfondie d'un système d'exploitation est un vaste sujet. Pour les lecteurs intéressés, la lecture des livres suivants est recommandée :

- Un livre complet présentant de façon approfondie tous les principes de conception, les mécanismes et les aspects techniques des systèmes d'exploitation :
Andrew S. Tanenbaum, *Systèmes d'exploitation*, 3-ième édition, PEARSON Education, 2003.
- Un livre lui aussi très approfondi mais en langue anglaise :
Abraham Silberschatz, Peter Baer Galvin et Greg Gagne, *Operating System Concepts*, 7-ième édition, John Wiley and Sons Inc., 2004.
- Un livre dédié au système d'exploitation qui marque l'âge de la maturité dans la technologie de ces systèmes, en l'occurrence le système *Unix* :
Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice Hall International, 1986.
- Un livre plus spécifiquement dédié au système d'exploitation *Windows* et qui en présente clairement les concepts et leur mise en œuvre :
Gary Nutt, *Operating System Projects using Windows NT*, Addison Wesley Longman Inc., 1999.
- Enfin, un livre plus spécialement dédié à la description de la notion de mémoire virtuelle, concept fondamental des systèmes d'exploitation et qui reste encore aujourd'hui parmi les logiciels les plus complexes jamais développés. La référence proposée ci-dessous décrit de façon approfondie la gestion des mémoires virtuelles dans le système *Linux* :
Mel Gorman, *Understanding the Linux® Virtual Memory Manager*, Bruce Perens' Open source series, Pearson Education, Inc., 2004.

Remerciements

Ces quelques pages ne seraient pas ce qu'elles sont aujourd'hui sans les nombreux lecteurs ou lectrices qui ont détecté les erreurs, oublis, points obscurs tout au long des mises à jour annuelles.

Il y a bien sûr parmi eux des générations d'élèves du département, mais aussi, les collègues qui ont participé à l'enseignement des systèmes d'exploitation de façon récurrente ou temporaire. En tout premier lieu, je tiens à remercier Philippe Quéinnec, correcteur redoutable, Philippe Mauran, pour la reformulation des passages trop énigmatiques, Patrick Amestoy pour ses interrogations opiniâtres et fort utiles, Mamoun Filali pour son rôle patient et infatigable de gourou et Michel Charpentier pour ses critiques toujours constructives.

Table des matières

1	Introduction	5
1.1	Une drôle d'histoire	5
1.2	Une définition un peu plus précise	7
1.3	Un système : un logiciel pas comme les autres!	8
1.3.1	Un logiciel réactif	8
1.3.2	Un logiciel nécessaire	8
1.3.3	Un logiciel qui partage et cohabite avec d'autres	8
1.3.4	Un logiciel qui résiste aux fautes des autres	9
1.3.5	Un programme au service des autres	11
1.3.6	Un programme parallèle	12
1.3.7	Un programme de taille respectable	12
1.3.8	Un programme discret ?	12
1.3.9	Conclusion	12
1.4	La notion de programme : le point de vue du noyau	13
1.4.1	La notion de format binaire objet	14
1.4.2	Image binaire exécutable	15
1.4.3	Autre format de programme exécutable	15
1.5	Principes de conception	16
1.5.1	Conception selon une approche hiérarchique	16
1.5.2	La notion de ressource virtuelle	17
1.5.3	Le principe de liaison dynamique	17
1.6	Un mini-système	17
1.6.1	L'architecture matérielle	18
1.6.2	La spécification du système d'exploitation	18
1.6.3	L'interpréteur de commande	19
1.6.4	Exercices	20
2	Les processus	21
2.1	Les processus	21
2.1.1	Origine et motivation	21
2.1.2	Le concept de processus	22
2.1.3	Les opérations sur les processus	23
2.1.4	L'ordonnancement des processus	24
2.2	L'environnement d'exécution d'un processus	26
2.2.1	La communication par événements asynchrones	26
2.2.2	La communication par flots de données	27
2.3	Conclusion	28

3 Les fichiers	31
3.1 Introduction	31
3.2 Le concept de fichier	33
3.2.1 Définition	33
3.2.2 Méthode d'accès séquentielle	33
3.2.3 Méthode d'accès directe	34
3.2.4 Autres méthodes d'accès	34
3.3 Organisation de l'espace logique des fichiers	35
3.3.1 La désignation des fichiers	36
3.3.2 La protection des fichiers	36
3.4 Organisation de l'espace physique des ressources	39
3.4.1 La désignation des ressources matérielles ou périphériques	39
3.4.2 La connexion des périphériques	39
3.4.3 La notion de volume	40
3.5 La projection de l'espace des fichiers sur l'espace des ressources	41
3.5.1 Principes généraux de conception	41
3.5.2 Le niveau répertoire : implantation de l'arborescence de fichiers	41
3.5.3 Le niveau fichier : implantation des fichiers	41
3.5.4 Le niveau bloc : la gestion de l'allocation/libération d'espace aux fichiers	44
3.5.5 La destruction des fichiers	45
3.5.6 La gestion des échanges	45
3.6 Conclusion	46
4 Les mémoires virtuelles	47
4.1 Mémoires virtuelles	47
4.1.1 Le problème du partage mémoire	47
4.1.2 Mémoire virtuelle	48
4.1.3 Principe de base	48
4.1.4 La pagination	49
4.1.5 La segmentation	49
4.1.6 La technique mixte segmentation-pagINATION	50
4.1.7 Le couplage des programmes	51
4.1.8 Les stratégies d'allocation de pages	51
4.1.9 Conclusion	55
5 Conclusion	57
Appendices	59
A Exercices	61
A.1 Des questions d'ordre général	61
A.2 À propos des processus	62
A.3 À propos des fichiers	63
B Encore mieux, exercices corrigés...	65
B.1 Généralités	65
B.2 Processus	66
B.3 Les fichiers	67

Chapitre 1

Introduction

1.1 Une drôle d'histoire...

L'histoire des systèmes d'exploitation se confond intimement avec l'histoire de la programmation, de l'algorithme et du génie logiciel. En effet, sans « système » (d'exploitation), pas d'utilisation possible d'un ordinateur. Pourtant, cet artefact reste bien mystérieux. En fait, un informaticien amateur ou professionnel peut souvent se contenter d'une vision très extérieure.

Il est facile de résumer la perception la plus courante :

- Lorsque l'on met sous tension un ordinateur (personnel par exemple), on entend le disque dur s'agiter, l'écran passe par différentes couleurs, puis quelques lignes truffées de noms étranges et de nombres défiler. Après un laps de temps variable (1mn environ), un ensemble d'icônes et de fenêtres apparaissent : ça y est, on peut cliquer !! LE système est chargé.
- Dans le cas où l'on vient d'acheter un magnifique engin avec, selon le vendeur, 500 Giga de disque dur et 2 Giga de mémoire centrale, on souhaite évidemment vérifier ces petites choses. Après quelques clics de souris et un peu de flair, on arrive à obtenir les informations recherchées : et là, bizarrerie, l'espace libre sur le disque dur est de 420 Giga. Où sont passés les 80 Giga manquants ? La mémoire centrale ne fait plus quant à elle, que 1,5 Giga. Où sont passés les 500 Mega manquants. Le néophyte se précipitera sur son téléphone pour traiter son vendeur de tous les noms d'oiseaux. Il entendra alors la phrase miracle mais encore énigmatique : « C'est LE système ».
- Pour l'usager confirmé, le système apparaît sous une forme désagréable : après des heures d'effort, un programme d'enfer commence à faire ce qu'on voulait. Mais, soudain, lors d'un énième test, une petite fenêtre de couleur blanche (blème) apparaît avec un message sibyllin :

« system error at #17AF486 »

Puis, plus rien. Les clics se perdent... L'écran reste désespérément figé. LE système est planté¹. Une seule solution : redémarrer.

- Pour l'usager confirmé, le système, c'est aussi celui dont on cause, entre amis, au coin du feu :

« Et toi ? tu as installé la version OS 8.5.1 ? »
« Non, je préfère rester en 7.5.3 ; j'assure »
« Oui, mais en 8.5.1, tu peux, blablabla...»

1. Terminologie professionnelle : on dit aussi plus précisément et par exemple, Windows Vista s'est planté

En résumé, le système, ça met du temps à démarrer, ça consomme des ressources (disque, mémoire), ça se « plante », sans lui, on ne peut plus rien faire, ça évolue de version en version vers un mieux toujours rêvé mais rarement vérifié.

On peut faire de l'informatique avec cette connaissance limitée d'un système d'exploitation. Cependant, il peut être intéressant d'en savoir un peu plus. Cela permet entre autre chose de pouvoir exploiter au mieux les possibilités offertes par un système ou de comprendre pourquoi ça ne marche pas.

Les systèmes d'exploitation ont été particulièrement importants dans le développement de l'informatique. Ce sont eux qui ont posé, dans leur conception et réalisation, les plus gros problèmes de programmation du fait de leur taille et de leur complexité. Ils ont donc beaucoup marqué, influencé, fait progresser le génie logiciel : principe de modularité, d'encapsulation par exemple, gestion de versions, cycle de développement, programmation par composants (« programming in the large ») avec les langages de script, mais aussi la programmation parallèle (synchronisation de processus), la notion de ressource virtuelle permettant de construire des abstractions des ressources réelles (machine virtuelle, mémoire virtuelle, imprimante virtuelle), la sécurité, la tolérance aux fautes, les bases de données, ...

Il est donc normal de leur rendre « hommage » !

Au passage, une remarque importante : il est à noter que les ressources matérielles que doit gérer un système d'exploitation ont vu leur technologie profondément évoluer au cours de ces 40 dernières années.

En effet, la puissance des processeurs a doublé tous les 18 mois depuis les années 70 par suite de l'intégration d'un nombre de plus en plus grand de transistors dans les microprocesseurs (la densité des transistors sur une puce a doublé tous les deux ans de 1971 à 2001 selon la loi dite de Gordon Moore).

La capacité des mémoires de masse et disques en particulier a aussi connu une augmentation extraordinaire. D'une capacité de quelques Mega (1 à 5 Mega) dans les années 70, une unité de disque offre aujourd'hui pour un prix moindre et sur un support matériel bien plus petit, une capacité de 500 Giga, soit 100 000 fois plus. Si des progrès similaires avaient été faits dans le domaine de l'automobile, une voiture qui consommait 10 litres au 100 kms, ne devrait plus consommer que 0,1 millilitre ... ou bien elle devrait rouler à 10 millions de km/h au lieu de 100 km/h ...

Il n'est pas étonnant qu'avec une telle évolution des performances des ordinateurs, ceux-ci aient eu un impact si important dans tous les domaines de la société au point de parler aujourd'hui de civilisation numérique ou d'économie numérique.

De façon étonnante, malgré cette évolution extrêmement rapide, les principes et concepts des systèmes d'exploitation se sont montrés très stables depuis les années 70.

Enfin, l'apparition des réseaux d'ordinateurs a lancé un nouveau défi à la communauté « système ». Après une vingtaine d'années d'effort de recherche, les systèmes d'exploitation répartis sont arrivés à maturité.

Mais ceci est une autre histoire ...

Dans ce qui suit, et compte tenu du contexte général de ce document, nous utiliserons l'abréviation « *système* » pour désigner un système d'exploitation.

1.2 Une définition un peu plus précise

Un système est, nous venons de le voir, un ensemble de modules, de composants logiciels indispensable pour pouvoir utiliser un ordinateur. Comme tout programme, il faut donc le démarrer, et c'est cette phase de démarrage qui se déroule lorsque l'on place l'ordinateur sous tension. En fait, comme tout programme exécutable, une partie du système doit être installée en mémoire centrale pour pouvoir s'exécuter : on appelle ce programme, cœur du système, le noyau².

Une fois ce noyau installé et actif, quelle fonction doit-il réaliser ? En fait, elle peut paraître élémentaire. Ce que veut l'usager, c'est pouvoir exécuter des programmes soit disponibles tels que les compilateurs, les éditeurs de textes, les interpréteurs,..., soit écrits par l'utilisateur lui-même. La fonction du système est d'assurer l'enchaînement de l'exécution de ces programmes à la demande de l'usager. Autrement dit, le système transforme l'architecture matérielle en une machine à exécuter des programmes. Cependant, cette tâche est plus ardue qu'il n'y paraît. Nous allons préciser pourquoi.

En résumé, un système est un logiciel qui doit remplir les objectifs suivants :

- Offrir une interface agréable aux usagers pour développer et exécuter leurs programmes. Cela passe par une interface simple pour soumettre les demandes d'exécution de programmes appelées *commandes* d'une part et, d'autre part, par un ensemble d'opérations mises à la disposition de tout programmeur pour réaliser des traitements ou gérer des données rémanentes³. Ces opérations sur des abstractions bien définies s'appellent des primitives et définissent une machine logique caractéristique du système ou de la famille de systèmes que l'on utilise.
- Offrir une interface indépendante de l'architecture matérielle : les caractéristiques des ressources de l'architecture tels que type de processeur, de disque, d'interface (SCSI, Firewire, USB par exemple),...doivent être le plus possible masquées. L'idéal est de pouvoir développer des programmes dans un environnement système donné (Unix par exemple) et porter ces programmes par simple recompilation sur différentes configurations et/ou marques de machines dès lors qu'elles utilisent le même système. Cette propriété de portabilité est extrêmement importante pour les applications. En fait, le système implante une machine virtuelle aux propriétés plus attractives que la machine réelle. C'est ainsi que l'on pourra parler de "machine" Windows pour signifier que l'on utilise une machine dotée d'un système Windows. Cette indépendance vis-à-vis du matériel est une des propriétés fondamentales d'un système d'exploitation.
- Assurer une gestion optimale des ressources disponibles. Autrement dit, le programme système doit être le plus économique possible. Il ne doit pas consommer trop de ressources pour sa propre exécution. C'est hélas un problème majeur comme nous l'avons vu avec les disques durs qui se réduisent et l'espace mémoire qui disparaît. De plus, le système doit exécuter les programmes demandés avec le maximum d'efficacité. Pour cela, l'utilisation du parallélisme potentiel offert par une architecture est un objectif très important. Dans presque toute architecture, les échanges mémoire centrale/périphérie (appelées historiquement les entrées/sorties) peuvent s'exécuter en parallèle avec l'exécution d'instructions sur le(s) processeur(s) central(aux).
- Contrôler les actions de l'usager (ou des usagers) afin de garantir, en tout premier lieu, l'intégrité des informations rémanentes placées sur disque dans des fichiers. De façon plus générale, le système doit garantir l'intégrité et la confidentialité des informations de ses usagers.

2. En anglais, kernel ou nucleus ou supervisor.

3. On appelle information rémanente, des données ou programmes dont la durée de vie est indépendante de l'exécution d'un programme, par opposition aux informations temporaires liées à la durée de vie d'une exécution.

1.3 Un système : un logiciel pas comme les autres !

Un système d'exploitation est donc en première approximation un ensemble de logiciels qui vont offrir un support, un environnement d'exécution aux programmes applicatifs. On peut cependant distinguer deux contextes possibles d'utilisation de cet environnement :

- un contexte de développement de programmes : le système est utilisé par des développeurs pour programmer et mettre au point de nouvelles applications dans un contexte métier particulier : calcul scientifique, gestion, multimédia, téléphonie, etc
- un contexte d'exploitation : le système est utilisé comme support d'exécution d'un ensemble d'applications dédiées : bureautique, transactions bancaires, contrôle d'un réacteur, acquisition d'images satellites, video à la demande, etc.

Ces deux modes d'utilisation ne sont bien sûr pas exclusifs et, pour le système d'exploitation lui-même, les fonctions de base à assurer sont identiques. Seuls des réglages de configuration ou des aspects « interface usager » viendront particulariser et adapter au mieux le système à son contexte d'utilisation.

Un logiciel système possède cependant quelques particularités par rapport à la plupart des programmes applicatifs. Nous allons lister ces principales originalités.

1.3.1 Un logiciel réactif

Un système interagit avec ses usagers. Il ne réalise donc pas le calcul d'une certaine fonction dotée de paramètres données comme le font les programmes fonctionnels. Son interaction avec l'environnement prend la forme d'un processus sans fin. Pire, on souhaite même qu'un système n'ait pas la mauvaise idée de s'arrêter (plantage!) car son bon fonctionnement est indispensable à l'usager. On peut simplement arrêter le fonctionnement du système lorsqu'on le souhaite.

1.3.2 Un logiciel nécessaire

Comme on vient de le souligner, le système est pratiquement indispensable pour utiliser efficacement et sans trop de difficultés un ordinateur. En particulier, le système doit être chargé au démarrage. En fait, c'est la partie noyau qui est recopiée en mémoire centrale depuis en général le disque dit *système*. Cette phase de chargement s'appelle le bootstrapping. Lors de la mise sous tension, le contrôle est donné à un programme de chargement initial (bootstrap).

En fait, c'est aujourd'hui, la plupart du temps, un metteur au point minimal résidant en mémoire morte. Celui-ci permet de visualiser/modifier par exemple les mots de la mémoire centrale, les registres du processeur, mais aussi de charger le fichier contenant la version exécutable du noyau.

Après le chargement du noyau, le contrôle est donné à un ou plusieurs programme(s) script(s)⁴ qui termine(nt) l'initialisation du système par le lancement d'activités assurant des fonctions systèmes comme par exemple la connexion d'un usager (phase de *login*) ou la gestion des imprimantes.

1.3.3 Un logiciel qui partage et cohabite avec d'autres

C'est certainement l'une des particularités majeures qu'il faut bien appréhender. Un système, et plus spécifiquement sa partie noyau, est un programme qui doit assurer l'exécution d'autres programmes. Donc, lorsqu'un système active un programme usager, il existe forcément deux programmes en mémoire centrale : le noyau et celui de l'usager.

4. Nous reviendrons sur cette notion de programme et langage de script.

Il y a donc partage des ressources de l'ordinateur entre le noyau et les programmes usagers. Réciproquement, les programmes usagers ont en commun (ils partagent) le noyau qui est donc vu comme une ressource commune. L'allocation équitable des ressources aux différents programmes usagers est un rôle fondamental du système. Il doit attribuer aux programmes usagers les ressources nécessaires à leur exécution : espace mémoire centrale pour charger le programme en mémoire, temps processeur pour l'exécution des instructions, accès aux périphériques unités de disque, lecteur/graveur de CD ou DVD, clé USB, scanner, imprimante, modem,...

À titre d'exemple, lorsqu'un système gère plusieurs usagers et plusieurs programmes par usager, il doit prendre soin de répartir équitablement les ressources. Il ne faut pas qu'un programme reste, par exemple, indéfiniment ignoré par le système.

Un des premiers défis relevé par les concepteurs de système a été de répartir les ressources d'un système monoprocesseur entre différents usagers interactifs. Ces systèmes dits « temps partagé » avaient pour ambition de faire croire à leurs usagers qu'ils étaient chacun seul à utiliser l'ordinateur. L'astuce adoptée était de provoquer une commutation de programme actif régulièrement par quantum de temps (100 millisecondes par exemple) de façon à ce que tous les programmes s'exécutent à tour de rôle. L'échelle de temps de l'usager étant beaucoup plus lente, celui-ci pouvait avoir l'impression de disposer de l'ordinateur pour lui seul (si le système était bien fait et les usagers peu nombreux ou peu actifs). Cette méthode reste toujours d'actualité dans la plupart des systèmes (non temps réel).

Ce partage de ressources et cette cohabitation posent aussi un problème de protection des programmes les uns envers les autres. En particulier, le noyau ne doit pas être mis en danger par les programmes usagers qu'il supervise.

1.3.4 Un logiciel qui résiste aux fautes des autres

Lorsqu'un programme a été chargé en mémoire par le noyau et que son exécution commence, le noyau perd le contrôle au profit de ce programme. Ce dernier peut alors provoquer des erreurs. On peut distinguer deux types d'erreurs dues aux fautes de programmation :

- les erreurs d'exécution détectées par le processeur : par exemple, instruction inconnue, opérande à une adresse inexistante, division par zéro, ... De telles erreurs sont détectées par le processeur et un déroutement se produit vers une routine de traitement. Pour que le système puisse continuer sa tâche (et non pas s'arrêter), il faut donc que la routine de traitement du déroutement soit partie intégrante du noyau et assure par exemple l'arrêt définitif du programme fautif, l'émission d'un message d'erreur et le passage à l'exécution d'un autre programme usager s'il en existe au moins un.
- les erreurs dues à des tentatives d'accès à des ressources contrôlées par le noyau ou allouées à d'autres programmes. L'une de ces ressources est l'espace alloué à chaque programme en mémoire centrale. En effet, un programme usager en exécution peut par erreur lire, écrire ou se brancher au hasard dans la zone mémoire où est chargé le noyau. Une lecture n'est pas trop grave excepté sous l'angle de la confidentialité. Une écriture est par contre beaucoup plus destructrice : une ou plusieurs instructions ou données du noyau peuvent ainsi disparaître. Dans ce cas, si le noyau doit exécuter la séquence détruite, le résultat est imprévisible, ou plutôt fort prévisible : le système se plante ! Il en sera en général de même pour un branchement, car on ne sait pas trop ce qui sera alors exécuté. Pour éviter de telles défaillances du système, il faut mettre en œuvre des mécanismes de protection mémoire.

Pour faire face à ces erreurs, un ensemble de mécanismes matériels ont été conçus et implantés dans les architectures des processeurs. Nous précisons le rôle de chacun de ces mécanismes.

Protection mémoire

Lorsqu'un programme usager a le contrôle, autrement dit, il s'exécute, il faut confiner son espace mémoire adressable de telle sorte qu'il ne puisse ni lire ni écrire dans les zones qui ne lui appartiennent pas. Il faut que le système partage la mémoire centrale en zones distinctes attribuées à chaque programme en cours d'exécution. Si un programme tente d'accéder hors de son domaine chez ses voisins, le mécanisme de protection mémoire provoquera un déroutement de telle façon qu'une nouvelle fois, le noyau puisse reprendre le contrôle et, par exemple, arrêter le programme fautif.

Modes d'exécution

Ces mécanismes de protection de la mémoire sont délicats à mettre en œuvre. En effet, il faut un mécanisme programmable par le noyau de telle façon que les zones protégées puissent évoluer selon les programmes. Dans ce cas, la protection ne sera réelle que si un programme usager ne peut la modifier à son profit. Un programme usager doit donc s'exécuter dans un mode qui lui interdit l'exécution des instructions permettant de modifier les protections de la mémoire. Seul le noyau doit pouvoir le faire. Ceci conduit à distinguer pour un processeur deux modes d'exécution (au minimum) :

- un mode usager (esclave), dans lequel certaines instructions du jeu d'instructions du processeur sont interdites. Toute tentative d'exécution provoquera une exception (déroulement) contrôlable par le noyau ;
- un mode superviseur (maître) dans lequel toutes les instructions du processeur sont exécutables. Les instructions exécutables seulement dans le mode superviseur sont appelées des instructions privilégiées.

Cette notion de mode assure la validité du système de protection mémoire à condition de contrôler le passage entre les deux modes. Lorsqu'un programme usager s'exécute, le processeur est en mode esclave. Cependant lorsqu'il appelle une primitive du noyau, il faut passer en mode maître de telle façon que la primitive puisse accéder à toutes les ressources sans restriction. Lors de la terminaison de la primitive, la sortie du noyau et le retour dans le programme appelant doivent s'accompagner inversement du passage du mode maître au mode esclave.

De façon plus générale, la notion de mode est utilisée pour limiter les droits d'accès d'un programme en exécution aux ressources du système. En effet, certaines instructions du processeur peuvent avoir un effet important sur le fonctionnement global du système. À titre d'exemple, si un programme peut exécuter une instruction qui le rend le plus prioritaire et non interruptible, ce programme risque de perturber fortement l'exécution des autres et en particulier monopoliser les ressources du système à son profit pour une durée indéfinie. De telles instructions ne seront donc autorisées qu'en mode superviseur.

Attention : un programme usager ne doit évidemment pas pouvoir programmer le changement de mode du processeur. On reviendrait à la case départ....

Interruptions

Un autre genre de faute peut être tout simplement le phénomène de boucle infinie. Un programme peut entrer dans une boucle permanente. Dans ce cas, seule l'intervention de l'usager ou du système au bout d'un délai plus ou moins long, peut permettre d'interrompre l'exécution du programme fautif. Il faut pour cela disposer d'un mécanisme matériel permettant d'interrompre l'exécution de la boucle par un signal externe au programme. C'est une des utilisations de la notion d'interruption.⁵

Conclusion

En conclusion, la tolérance aux fautes des programmes usagers et la protection des programmes les uns par rapport aux autres imposent donc de disposer de mécanismes matériels spécifiques : un système de gestion des exceptions (interruptions et déroutements), un mécanisme de protection mémoire et au moins deux modes d'exécution du processeur. Le tableau ci-dessous fait le lien entre fautes et mécanismes nécessaires à leur correction.

5. Les interruptions jouent par ailleurs un rôle essentiel dans la gestion des entrées/sorties.

Faute du programme	Mécanisme nécessaire
Accès hors limite	Protection mémoire + modes maître/esclave
Boucle infinie	Interruption
Instruction inexécutable	Déroutement
Intégrité des ressources gérées par le système	Instructions privilégiées

1.3.5 Un programme au service des autres

Un système est un programme au service des autres. En effet, lorsqu'un programme usager s'exécute, il appelle en général différents services offerts sous la forme de primitives par le noyau. Par exemple, lorsque l'on veut lire ou écrire un fichier, lorsque l'on veut obtenir de l'espace mémoire, si l'on veut connaître la date, le répertoire courant, si l'on veut attendre la fin d'un autre programme, etc.

L'appel de tels services, entre un programme usager et le noyau, obéit au schéma procédural. Cependant, l'implantation d'un appel de primitive nécessite un mécanisme spécifique. En effet, si le programme usager peut appeler une primitive du noyau par une instruction de branchement classique, cela signifie que tout programme a le droit d'appeler le noyau en un point quelconque (et pas seulement à un point d'entrée d'une primitive). Ceci est évidemment inacceptable comme nous l'avons vu. De plus, la commutation de mode n'est pas effectuée. Par conséquent, il faut pouvoir, certes autoriser les appels des primitives du noyau, mais il faut aussi protéger le noyau des appels dans le vide.

Pour restreindre les appels à des points d'entrées prédéfinis, on a donc recours à un mécanisme de type déroutement programmé. Sur tout processeur, il existe une instruction permettant de provoquer un déroutement volontaire (explicitement programmé). Le format le plus classique d'une telle instruction est le suivant : *TRAP n* où *n* est un numéro qui sert d'index dans une table de branchements. Lorsqu'une telle instruction est exécutée, il y a branchement à l'adresse trouvée dans la table à l'index spécifié, celle-ci fournissant le point d'entrée de la primitive. Il suffit donc que le noyau contienne une telle table correctement initialisée.

Bien entendu, cette table doit être protégée de telle façon qu'elle ne puisse être modifiée par un programme utilisateur.

Enfin, nous avons vu, que grâce au noyau, un système construit une machine étendue (virtuelle) dotée de primitives qui implantent des abstractions simplifiant la programmation. Par exemple, le concept de fichier permet de s'abstraire de la connaissance des paramètres internes de programmation d'une unité de disque (adresses des registres de contrôle, adresse physique de l'unité, codage des ordres d'écriture/lecture, positionnement de tête,...). Le système construit une sorte de machine virtuelle à fichiers ayant pour composant et support physique, l'unité de disque.

Cependant, il est important de garantir l'intégrité de la structure de fichiers mémorisée sur le support physique. Or, un programmeur mal intentionné (le hacker de base) peut très bien arriver à connaître suffisamment d'informations sur la programmation de plus bas niveau de l'unité. Dans ce cas, il peut arriver à écrire un programme qui pourra lire ou pire écrire un ou plusieurs secteurs du disque SANS avoir recours à l'appel d'une primitive d'accès aux fichiers. Il « court-circuite » le noyau. Dans ce cas, l'intégrité de l'espace de fichiers ne peut plus être garantie par le système.

Il est donc extrêmement important qu'un programme usager ne puisse pas exécuter une telle opération directe d'accès. Pour cela, deux approches sont possibles :

- Si les interfaces des contrôleurs sont projetés en mémoire (les registres de contrôle sont visibles comme des mots à des adresses précises et fixes de la mémoire centrale) : dans ce cas, il suffit de protéger cet espace et d'empêcher ainsi l'accès au contrôleur.
- Si les interfaces des contrôleurs sont programmables à l'aide d'instructions spécifiques du processeur, il suffit que ces instructions soient privilégiées. Un programme qui s'exécute en mode esclave ne pourra pas exécuter de telles instructions.

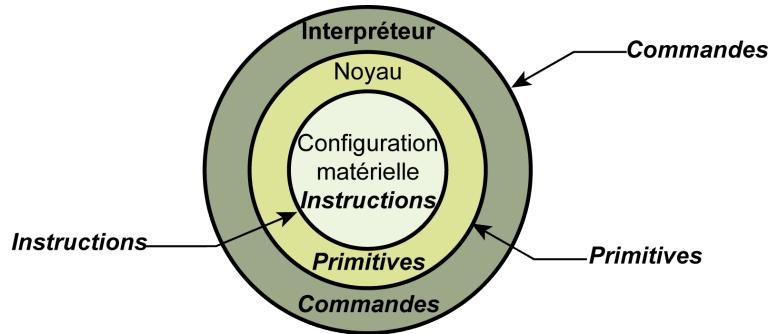


FIGURE 1.1 – Principe de structuration en couches

1.3.6 Un programme parallèle

Les systèmes d’exploitation sont indéniablement à l’origine de la programmation parallèle. C’est leur conception et réalisation qui a conduit à découvrir les difficultés du partage de ressources entre des activités concurrentes.

En effet, les architectures de calculateur ont offert la possibilité de traitements parallèles dès les années 1960. Les échanges mémoire centrale/périphérique (appelés entrées/sorties) ont été confiés à des processeurs dédiés (canaux) permettant d’exécuter en parallèle un programme sur le(s) processeur(s) central(aux). Ce principe est d’autant plus efficace que l’échange est long. Il y a en effet une différence de temps d’accès et de débit extrêmement importante entre la mémoire centrale, les mémoires de masse telles que les disques et les périphériques tels que les consoles, les imprimantes.

Le concept fondamental de processus a été proposé pour structurer et contrôler ces activités parallèles. Les techniques de synchronisation et communication entre processus constituent un volet clé du domaine informatique.

1.3.7 Un programme de taille respectable

Même si l’on se limite au noyau, ce programme est de taille respectable (2 Mega-octets par exemple). Il comporte différentes classes de modules :

- Les primitives qui sont les opérations (sous-programmes) appelées par les programmes usagers (ou services système) ;
- Les pilotes d’entrées/sorties (« device drivers ») qui implantent les interfaces de programmation des différents contrôleurs d’unités périphériques ;
- Les routines de traitement des exceptions : déroutements ou interruptions.
- Les programmes de services qui assurent des fonctions spécifiques : service d’impression, service de connexion…

1.3.8 Un programme discret ?

Un système a pour objectif de mettre à la disposition des programmes usagers les ressources de l’ordinateur qu’il contrôle. Pour assurer sa tâche, nous avons vu qu’il est « embarqué dans le même bateau » et qu’il consomme lui-même inévitablement des ressources. Il doit cependant rester le plus discret possible par un temps de supervision (« overhead ») minimal. C’est un objectif difficile qui n’est pas toujours atteint.

1.3.9 Conclusion

Un système d’exploitation comporte un composant essentiel appelé noyau. Celui-ci est certes « un programme », dans la mesure où il se compose de primitives, de pilotes, de routines, tous ces composants étant fondamentalement des « programmes », du code exécutable. Cependant, nous venons de voir

que la fonction que doit assurer ce noyau, en l'occurrence, l'exécution des autres programmes, nécessite des mécanismes particuliers au niveau de l'architecture matérielle et pose des difficultés particulières de réalisation. C'est pourquoi, le développement de ces noyaux, et plus généralement de ces systèmes d'exploitation, a soulevé de nombreux problèmes de génie logiciel et, par conséquent, a fait progressé sans nul doute l'art ou la science de la programmation.

1.4 La notion de programme : le point de vue du noyau

Nous avons beaucoup parlé de « programmes ». Nous avons vu, en particulier, que la fonction d'un noyau était d'assurer l'exécution correcte des programmes qui étaient soumis par les usagers au système. Il est donc indispensable de préciser cette notion de programme. Plus précisément, qu'est-ce qu'un programme pour le noyau ? Cette question est primordiale et elle constitue même une marque spécifique d'un système. En effet, tous les systèmes d'exploitation ne traitent pas le même format de programme.

Auparavant, rappelons la notion de programme telle qu'elle se présente pour un programmeur d'application. Prenons pour exemple un programme source écrit en C :

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("coucou");
    return 0 ;
}
```

Ce programme va être compilé par le compilateur C qui va engendrer dans un fichier une représentation nouvelle du programme : en l'occurrence, la version dite binaire objet ré-éritable. Un tel fichier contient une description structurée du programme à exécuter. Ce format de description est très spécifique d'un système d'exploitation (ou d'une famille de).

La figure 1.2 présente cette structure distinguant l'en-tête et une suite de sections ainsi que des tables diverses qui seront utilisées par l'éditeur de liens. Le code engendré dans une section spécifique (dite « `text` ») n'est cependant pas encore tout à fait exécutable. Des opérandes, en particulier, dans les instructions de branchement, restent à définir.

Une « décompilation » de la seule section de code du programme binaire objet de l'exemple, donne le code suivant en assembleur pour un processeur Intel :

Adresse	code	opérandes	opérandes avec symboles
main :			
0000000000000000	pushq	%rbp	
0000000000000001	movq	%rsp,%rbp	
0000000000000004	subq	\$0x10,%rsp	
0000000000000008	movl	%edi,0xfc(%rbp)	
000000000000000b	movq	%rsi,0xf0(%rbp)	
000000000000000f	leaq	0x00000000(%rip),%rdi	LC0(%rip),%rdi
0000000000000016	movl	\$0x00000000,%eax	\$_main, %eax
000000000000001b	callq	0x00000020	_printf
0000000000000020	movl	\$0x00000000,%eax	\$_main, %eax
0000000000000025	leave		
0000000000000026	ret		

Le code produit par le compilateur est dans un format binaire objet ré-éritable, c'est-à-dire qu'il peut être traité par un éditeur de liens pour obtenir un programme binaire objet exécutable. Ainsi, après la phase d'édition des liens, on obtient le code final suivant :

Adresse	code	opérandes	opérandes avec symboles
start :			
00000000100000ec4	pushq	\$0x00	
00000000100000ec6	movq	%rsp,%rbp	
00000000100000ec9	andq	\$0xf0,%rsp	
00000000100000ecd	movq	0x08(%rbp),%rdi	
00000000100000ed1	leaq	0x10(%rbp),%rsi	
00000000100000ed5	movl	%edi,%edx	
00000000100000ed7	addl	\$0x01,%edx	
00000000100000eda	shll	\$0x03,%edx	
00000000100000edd	addq	%rsi,%rdx	
00000000100000ee0	movq	%rdx,%rcx	
00000000100000ee3	jmp	0x100000ee9	
00000000100000ee5	addq	\$0x08,%rcx	
00000000100000ee9	cmpq	\$0x00,(%rcx)	
00000000100000eed	jne	0x200000ee5	
00000000100000eef	addq	\$0x08,%rcx	
00000000100000ef3	callq	0x100000f00	_main
00000000100000ef8	movl	%eax,%edi	
00000000100000efa	callq	0x100000f28	symbol stub for : _exit
00000000100000eff	hlt		
_main :			
00000000100000f00	pushq	%rbp	
00000000100000f01	movq	%rsp,%rbp	
00000000100000f04	subq	\$0x10,%rsp	
00000000100000f08	movl	%edi,0xfc(%rbp)	
00000000100000f0b	movq	%rsi,0xf0(%rbp)	
00000000100000f0f	leaq	0x0000001e(%rip),%rdi	LC0(%rip),%rdi
00000000100000f16	movl	\$0x00000000,%eax	\$_main, %eax
00000000100000f1b	callq	0x100000f2e	symbol stub for : _printf
00000000100000f20	movl	\$0x00000000,%eax	\$_main, %eax
00000000100000f25	leave		
00000000100000f26	ret		

On peut constater que l'éditeur de liens a ajouté un code commençant à l'étiquette `start` et se déroulant jusqu'à l'étiquette `_main` qui débute la génération du code du programme correspondant. On remarquera aussi l'instruction de branchement de type sous-programme (`callq`) pour appeler le programme principal. Néanmoins, le code du sous-programme `printf` n'est toujours pas présent. En fait, il n'est pas inclus dans le binaire objet exécutable car il sera couplé dynamiquement lors du chargement du programme en mémoire. Autrement dit, il apparaitra bien dans l'image binaire mémoire qui sera construite par le chargeur à partir de ce binaire objet exécutable, mais cette implantation des sous-programmes de la bibliothèque C ne se fera qu'en mémoire virtuelle au moment de l'implantation de l'image binaire exécutable⁶.

1.4.1 La notion de format binaire objet

La notion de format binaire objet exécutable ou ré-éditable est donc fondamentale dans un système d'exploitation. C'est la représentation « officielle » de tout programme qui peut être développé et mis en œuvre sous un système donné. La structure logique d'un binaire objet ré-éditable ou exécutable est la même.

En résumé, le format de description d'un programme pour un système donné est :

- Spécifique d'une famille de systèmes : Unix, Windows, ... ;
- Portable : format identique \forall le processeur ;

6. Se reporter au chapitre sur les mémoires virtuelles pour plus de détails sur cette notion de couplage en 4.1.7.

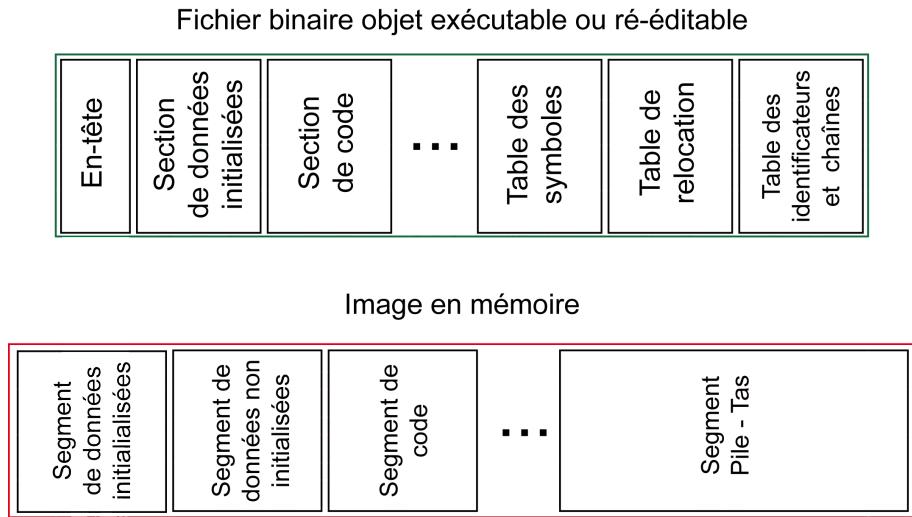


FIGURE 1.2 – Programme binaire exécutable et image

- Existe sous deux formes de format commun : exécutable et ré-éditable (statiquement ou dynamiquement).
- À titre d'exemples, nous citons quelques formats selon les systèmes :
- Les vieux formats Unix : Formats **a.out** et **COFF** (Common Object File Format)
 - La famille Unix (Linux, Solaris, Irix, System V, BSD) : Format **ELF** (Executable and Linking Format)
 - La famille Windows : Format **PE** (Portable Executable Format) et **COFF** .
 - La famille Mac OS X d'Apple : le format **macho**...

1.4.2 Image binaire exécutable

Une dernière forme de programme existe : l'image du programme qui sera implantée en mémoire centrale par le chargeur du noyau pour que le programme puisse enfin être exécuté. Cette image mémoire est donc construite à partir du binaire objet exécutable lors du chargement en mémoire. La figure 1.2 montre les deux représentations de programme.

En fait, cette image sera implantée très souvent dans une mémoire virtuelle par couplage (voir 4.1.7). De façon similaire au format binaire objet, une image binaire d'un programme est structurée en segments. Chaque segment contient soit du code, soit des données. Un segment est réservé pour l'allocation dynamique de mémoire durant l'exécution du programme : allocation d'espace selon un schéma de pile lors des appels de procédure, allocation d'espace selon un schéma de tas pour les structures de données dynamiques accédées par pointeur.

1.4.3 Autre format de programme exécutable

Pour le système d'exploitation, il existe un autre format de programme exécutable. En effet, nous avons vu que les requêtes pour soumettre un programme à exécuter peuvent se faire grâce à un langage de commandes (ou langage de script) par l'intermédiaire d'un interpréteur. Cet interpréteur considère une commande soit comme immédiatement interprétable par lui-même (cas des commandes dites « built in ») soit comme un nom de programme exécutable. Or, bien entendu, pour l'interpréteur, deux possibilités sont offertes :

- soit il s'agit d'un programme binaire objet exécutable : c'est le format que nous venons d'analyser et il suffira de provoquer le chargement en mémoire de ce programme exécutable ;
- soit il s'agit d'un programme de script, c'est-à-dire un programme écrit dans le langage textuel de l'interpréteur. Ce programme est tout autant exécutable, mais cette fois-ci, par l'interpréteur (et non

pas par le processeur natif).

Conséquence : ces deux formes de programmes sont donc exécutables en ce sens que un fichier qui contient l'un ou l'autre de ces programmes peut être accédé en mode « exécution ».

1.5 Principes de conception

Un système présente donc des particularités importantes. Cependant, la réalisation d'un tel logiciel s'appuie sur des principes de conception qui ont été ensuite très utiles en génie logiciel de façon générale. Nous insistons sur les principes de conception essentiels.

1.5.1 Conception selon une approche hiérarchique

Un principe de conception fondamental consiste à construire une hiérarchie de machines abstraites, virtuelles. Chacune implante un jeu d'instructions, de commandes qui sera implanté à l'aide de la machine de niveau inférieur. On représente graphiquement cette approche méthodologique sous la forme d'une cible comme la figure (1.1).

Au niveau le plus bas, on trouve l'architecture matérielle qui apporte les ressources utilisables et un jeu d'instructions correspondant à une famille de processeurs (RISC par exemple). À ce niveau les instructions sont très élémentaires et leurs opérandes aussi (octets, entiers, réels flottants...).

Au niveau suivant, le cœur d'un système est constitué par le programme noyau. Celui-ci permet d'étendre le jeu d'instructions précédent par un ensemble de primitives qui implante les abstractions clés qui permettront au système de gérer les traitements et les données des usagers. Pour les traitements, la notion de processus permet de contrôler l'exécution d'un programme séquentiel. Pour les données, la notion de fichier permet de s'abstraire des particularités physiques des périphériques et de contrôler l'accès aux informations de chaque usager. Les primitives d'un noyau sont des sous-programmes qui peuvent donc être appelés durant l'exécution d'un programme. L'interface d'appel est cependant particulière via la notion de déroutement programmé (*TRAP*). Ils peuvent comporter de quelques centaines à plusieurs milliers d'instructions machine.

Au niveau suivant, un interpréteur de commande va permettre de dialoguer avec le système pour soumettre des demandes d'exécution de programmes. Un interpréteur de commandes est en fait un programme système qui, une fois actif, devient un processus interpréteur. Dans le système Unix, la famille des langages de scripts *shell* permet de demander l'exécution d'un ou plusieurs programmes éventuellement en parallèle. L'exécution d'une commande est alors la plupart du temps synonyme de l'exécution d'un programme sous la forme d'un processus. Seules quelques commandes très élémentaires, dites « built-in », sont directement interprétées. À ce niveau, on a donc bien obtenu une machine capable d'assurer l'exécution de programmes quels qu'ils soient.

Les langages de script sont particulièrement importants pour le développement rapide de programme, pour le prototypage. Ils sont aussi des langages de programmation « à large échelle » dans la mesure où les objets opérandes sont des entités à gros grain, essentiellement des fichiers de données ou des fichiers exécutables (contenant des programmes exécutables ou interprétables).

À ceci peut s'ajouter un sous-système de fenêtrage tel que X11. Dans ce cas, un processus serveur assure le contrôle de la ressource écran et accepte de gérer les interactions avec l'usager via la souris et un curseur d'écran.

Chaque frontière entre niveaux adjacents de la hiérarchie précédente comporte une rupture d'échelle. Le tableau ci-dessous résume la structuration à trois niveaux des systèmes actuels.

Niveau	Ressource	Type d'interface	Durée d'une opération
Matériel	Processeur	Instruction	μ sec
Noyau	Primitives	Sous-programme	> millisec
Interpréteur	Commandes	Programmes	quelconque

1.5.2 La notion de ressource virtuelle

La notion de ressource virtuelle est apparue assez tôt dans la conception des systèmes. Cette notion s'inscrit naturellement dans l'effort général des concepteurs de système pour offrir aux programmes des usagers un environnement de programmation masquant les caractéristiques matérielles et parfois même leur limitations ou imperfections.

Une première application de l'idée de ressource virtuelle a été la notion « d'imprimante virtuelle ». Au lieu que chaque programme édite directement ses résultats sur une imprimante souvent unique et lente, les résultats sont redirigés dans un fichier sans modification du programme. Cette redirection est transparente. Un programme spécifique assure l'édition effective du contenu de chaque fichier d'impression séquentiellement. Cette stratégie est appelée *spooling*⁷. La démarche est donc d'attribuer une imprimante virtuelle à chaque programme. Celle-ci est privée au programme (pas de problème de concurrence) et plus rapide que la ressource réelle (la rapidité d'écriture sur disque est beaucoup plus grande que la vitesse d'impression). La ressource virtuelle a des propriétés meilleures que celle de l'objet réel.

Cette idée a été en particulier utilisée pour la gestion de la mémoire centrale. Le partage de la mémoire disponible entre plusieurs programmes (multiprogrammation) pose des problèmes d'allocation de la ressource mémoire. Le concept de mémoire virtuelle permet d'optimiser l'allocation de la mémoire réelle et, dans certains cas, autorise par exemple l'exécution de programmes ayant une taille supérieure à la mémoire réelle !

Une ressource virtuelle est donc une abstraction d'une ressource réelle qui « améliore » les propriétés de la ressource réelle. Par ailleurs, le nombre de ressources virtuelles utilisées peut être dynamique. Ainsi, de multiples exemplaires d'une ressource qui n'existe physiquement qu'en un seul exemplaire peuvent être créés à la demande.

Aujourd'hui, la notion de machine virtuelle est très développée. Vieille idée, elle réapparaît avec la puissance énorme des serveurs actuels. Cette notion permet de transformer une machine unique réelle en un ensemble de machines virtuelles qui, chacune, peuvent exécuter un système d'exploitation éventuellement différent. Cette approche connaît un fort développement permettant par exemple de faire tourner deux systèmes d'exploitation, Linux et Windows, simultanément sur une même machine ou de faire cohabiter une machine virtuelle d'exploitation et une machine virtuelle de développement sur la même machine réelle.

1.5.3 Le principe de liaison dynamique

Lorsqu'un programme s'exécute, il demande dynamiquement des services au noyau. Par exemple, un programme veut accéder à un fichier pour le lire. Par l'appel de la primitive *open*, le noyau va créer la connexion dynamique entre le fichier et le programme usager et renvoyer à l'appelant un identificateur de connexion (qui est souvent un simple index) qui sera ensuite utilisé lors des opérations de lecture. L'opération *close* rompra cette liaison. C'est donc une liaison dynamique qui est créée entre le programme en exécution et la ressource fichier qu'il veut utiliser.

Cette fonction de liaison, connexion dynamique est une des tâches de base du noyau. En particulier, toutes les communications du processus avec son environnement externe (fichiers, pseudo-périphériques, autres processus) nécessitent préalablement une opération de liaison dynamique qui s'accompagne en général d'un contrôle sur les droits d'accès de l'appelant.

L'optimisation de l'utilisation des ressources conduit le noyau à adopter le plus possible une stratégie consistant à retarder la liaison effective le plus tard possible (*Delay Binding Time*). À titre d'exemple, dans un système de gestion de mémoire virtuelle, la liaison d'une zone de mémoire réelle (appelée page) à un programme peut être retardée jusqu'à l'ultime moment d'une référence dans cette zone.

1.6 Un mini-système

Pour préciser les principaux problèmes auxquels se heurte le concepteur de système d'exploitation, on considère un exemple de mini-système. Nous présentons d'abord l'architecture matérielle, puis la machine

7. de l'abréviation SPOOL : Simultaneous Peripheral Operations On-line

logique, abstraite qui est implantée pas le système d'exploitation et qui va rendre utilisable, « exploitable » l'architecture matérielle. Pour simplifier, on considère un système très simple qui offre une interface d'usage minimaliste, en l'occurrence, un écran/clavier qui permet de soumettre à un interpréteur des commandes.

1.6.1 L'architecture matérielle

Une architecture matérielle simple est celle des ordinateurs personnels (PC) qui comportent un bus unique auquel sont connectés les différents modules : processeur, mémoire et interfaces diverses : Ethernet, FireWire, USB, séries (RS232), parallèle,...

La figure (1.3) présente le schéma d'une telle architecture ne comportant comme ressource périphérique qu'un écran-clavier et un disque fixe.

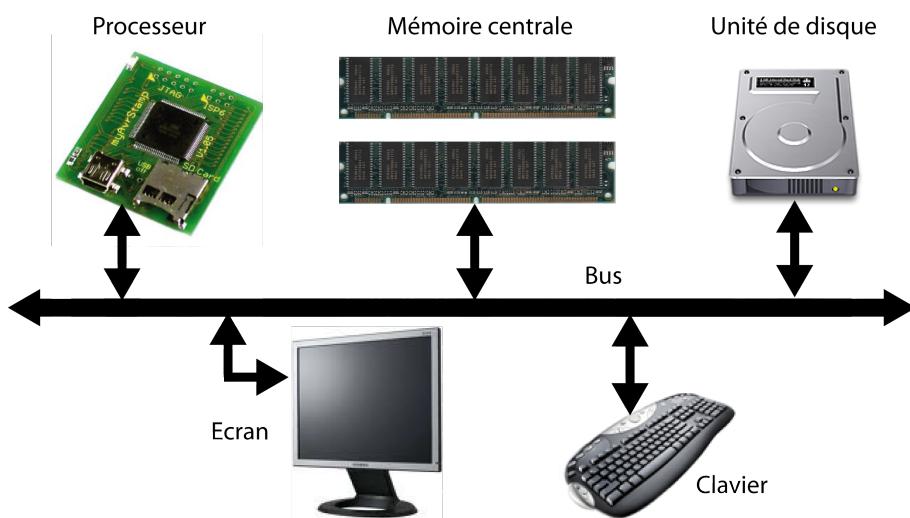


FIGURE 1.3 – Architecture minimale standard

1.6.2 La spécification du système d'exploitation

Pour utiliser l'architecture précédente, il faut définir un protocole de communication entre l'usager devant son écran-clavier et cette architecture. C'est le rôle du système de réaliser une telle interface de dialogue. La fonction de base qui doit être assurée via ce protocole est de permettre à l'usager de soumettre des demandes d'exécution de programmes. Autrement dit, le «système» aura pour tâche d'assurer la bonne exécution des programmes demandés par l'usager séquentiellement.

L'implantation de cette interface de dialogue peut prendre la forme minimalisté suivante : après chargement et initialisation du système, l'usager voit apparaître un prompt (par exemple le symbole >) lui indiquant la disponibilité du système. Il peut alors taper une commande sur une ligne et terminer par un *return*.

Toute commande est en fait une suite de chaînes de caractères séparés par des blancs. La première chaîne doit être un identificateur qui désigne un nom de fichier contenant un programme exécutable⁸. Les chaînes suivantes constituent les arguments fournis au lancement de l'exécution du programme par le « système ». En effet, le système interprète la commande frappée par l'usager comme une demande d'exécution du programme exécutable référencé.

On suppose qu'il existe sur le disque, un ensemble de fichiers contenant des programmes exécutables constituant les commandes de base offertes aux usagers. Ceci suppose évidemment que le volume disque est

8. Dans le cas d'un vrai système, l'interpréteur de commande exécute lui-même un ensemble prédéfini de commandes qualifiées de « built-in ».

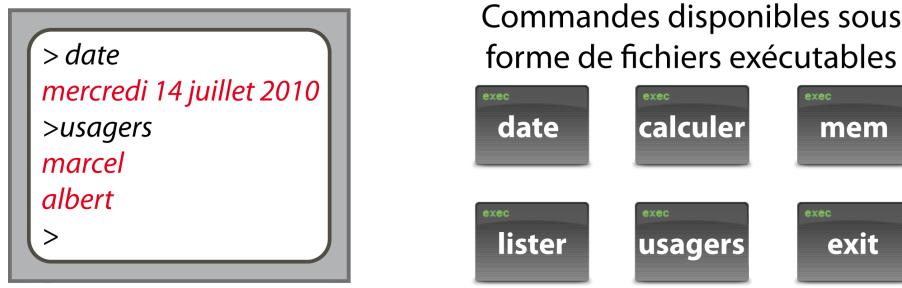


FIGURE 1.4 – Machine abstraite usager

bien vu comme un espace de fichiers et que l’interpréteur va chercher les fichiers binaires objets exécutables implantant les commandes dans un répertoire prédéfini.

Nous donnons quelques exemples de commandes possibles :

<i>lister</i>	liste les commandes de base qui existent ;
<i>date</i>	affiche la date présente ;
<i>expliquer <nom de commande></i>	explique la fonction de la commande référencée ;
<i>calculer <expression arithmétique></i>	calcule l’expression arithmétique fournie en paramètre et affiche le résultat ;
<i>mem</i>	affiche l’espace mémoire centrale disponible ;
<i>usagers</i>	liste les usagers connus du système ;
<i>exit</i>	arrêt du système ;
...	

La figure (1.4) illustre cette vision, à un niveau logique, de la « machine abstraite » usager. On ne considère ici que l’interface usager via un écran/clavier.

1.6.3 L’interpréteur de commande

La gestion du dialogue usager-système s’appuie classiquement sur un programme appelé interpréteur de commandes. L’algorithme de principe d’un tel interpréteur, décrit en pseudo-Java, est le suivant :

```
Interpréter() {
    while (true) {
        écran.Afficher("'");
        Commande c = Ligne.Lire();
        if (c.Valide()) c.Exécuter();
        else écran.Afficher(c.Erreur);
    }
}
```

Quant à l’exécution d’une commande valide, elle peut être raffinée dans la classe *Commande* par la description de la méthode *Exécuter* :

```
class Commande {
    ...
    void Exécuter() {
        Proc prog = Programme.Charger(this.nom);
        prog(this.arg);
    }
    boolean Valide() {...}
    ...
}
```

Quelques remarques :

- les commandes se traduisent pour l'interpréteur par le lancement d'un programme qui est activé par un appel de type procédural après chargement préalable en mémoire centrale ;
- l'interpréteur fait appel à des opérations « d'utilité publique » telles que `Afficher`, `Lire`, `Charger` qui appellent des primitives du noyau. Ces primitives constituent un ensemble d'opérations sur des abstractions telles que fichiers, processus, mémoire, canaux, ... qui définissent une machine « système », à la fois plus simple et puissante, pour développer des programmes. Un noyau implante ainsi une machine étendue « système » offrant les mêmes services à tous les programmes usagers.

1.6.4 Exercices

L'interpréteur précédent doit assurer un « service » permanent et fiable. Pour ce faire, plusieurs obstacles doivent être surmontés. En effet, des fautes de programmation dans les programmes « *commandes* » et plus généralement, dans un vrai système, les programmes des usagers, peuvent provoquer des erreurs conduisant à l'arrêt du système.

1. Énumérer différentes anomalies qui pourraient survenir durant l'exécution du programme chargé et lancé par l'interpréteur ;
2. Pour chacune, proposer un mécanisme pour récupérer l'erreur et assurer la survie du système ;
3. L'interface du système jouet proposé est enrichie par la notion d'usager. Un usager ne peut exécuter des commandes qu'après une phase d'identification. Modifier l'interpréteur pour assurer ce contrôle d'accès.

Chapitre 2

Les processus

2.1 Les processus

2.1.1 Origine et motivation

Une tâche fondamentale d'un système d'exploitation est d'assurer l'exécution de programmes divers et variés...

Ces programmes peuvent être de différentes natures :

- programme développé par un programmeur pour une application ;
- programme assurant une fonction dans la production même d'autres programmes : compilateurs, interpréteurs, éditeur de liens, metteur au point ;
- programme dit "système" assurant une tâche dans le fonctionnement même du système : interpréteur de commande par exemple.

Les concepteurs de systèmes d'exploitation se sont très vite aperçus que l'on pouvait envisager de gérer l'exécution de plusieurs programmes en parallèle. En effet, une architecture classique de machine, même mono-processeur, autorise le parallélisme des échanges d'information entre les périphériques (disques, bandes imprimantes, scanners, CD,...) et la mémoire centrale d'une part, et d'autre part, l'exécution d'un programme par le(s) processeur(s) avec cependant un problème de contention et de partage d'accès à la mémoire centrale qui apparaît alors comme une ressource commune partagée. La figure (2.1) illustre ce parallélisme des « entrées/sorties » avec l'exécution des programmes.

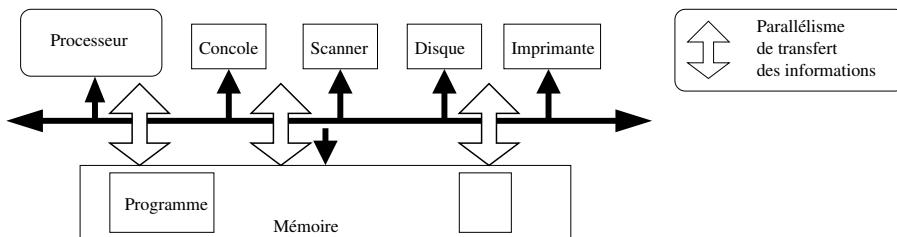


FIGURE 2.1 – Parallélisme entre entrées/sorties et exécution d'un programme

Or, on a grand intérêt à exploiter ce parallélisme. Prenons l'exemple d'un interpréteur de commandes. Son schéma d'exécution comporte des lectures répétitives du clavier pour lire la commande de l'usager. Tant que la ligne de commande n'a pas été frappée, le programme interpréteur n'a rien d'autre à faire que de tester de façon continue si le caractère de fin de ligne a bien été lu et placé dans un octet de la mémoire centrale. Ce test incessant est une boucle qui va être exécutée durant plusieurs secondes. Autrement dit, le processeur exécute plusieurs millions de fois la même instruction de test. Le processeur pourrait donc être

plus efficacement utilisé si l'on pouvait assurer l'exécution d'un autre programme pendant cette période où l'interpréteur est « logiquement » bloqué. Pour cela, plusieurs conditions sont néanmoins nécessaires :

- il faut savoir suspendre momentanément l'exécution d'un programme : le programme interpréteur doit être arrêté, son état courant d'exécution doit être sauvegardé en mémoire de telle sorte que l'on puisse continuer son exécution ultérieurement. Ceci implique de savoir sauvegarder/restaurer un contexte d'exécution de programme. C'est pourquoi, tout processeur possède un contexte minimal d'exécution rassemblant les informations caractéristiques et suffisantes pour pouvoir, si elles sont rechargées dans les registres du processeur, reprendre l'exécution en son point de sauvegarde. Cet ensemble d'informations se présente sous la forme d'un « mot d'état programme » (PSW : Program Status Word), car la sauvegarde de l'état en mémoire centrale (en général par empilement) occupe un (double)-mot ;
- l'interface de supervision des entrées/sorties doit pouvoir émettre une interruption vers le processeur lorsqu'un échange de données est terminé afin que le noyau système puisse enregistrer que le programme ayant lancé l'échange peut désormais continuer. Cette gestion des événements asynchrones de fin de lecture ou d'écriture met en jeu intensivement le mécanisme d'interruption. On peut remarquer que le traitement d'une interruption implique aussi la sauvegarde de l'état du programme qui est interrompu. Le processeur commute de l'exécution du programme (interrompu) à l'exécution de la routine de traitement de l'interruption. En fin de routine, le contrôle doit être rendu au programme interrompu et donc le processeur commute de l'exécution de la routine à l'exécution du programme interrompu.

Ainsi, d'un point de vue logique, le processeur(s) ne s'arrête(nt) jamais, mais il(s) commute(nt) fréquemment d'un programme à un autre, ce programme pouvant être un programme applicatif ou un programme « système » tel que l'interpréteur de commandes. Le noyau, quant à lui, sert d'intermédiaire et de superviseur de ces commutations (via l'exécution de primitives ou de routines d'exception). C'est pourquoi, les concepteurs de système ont eu l'idée d'abstraire, sous la forme d'un concept et aussi d'un objet, la gestion de l'exécution d'un programme sous le contrôle d'un noyau d'exploitation. Ce concept est celui de processus.

2.1.2 Le concept de processus

Une processus modélise l'exécution d'un programme. Il capte le caractère dynamique d'un traitement réalisé par un programme. Il ne doit donc pas être confondu avec la notion de programme qui capte l'aspect statique et descriptif d'un traitement. On peut par exemple comparer le programme à un DVD et le processeur au lecteur de DVD. Alors, le processus sera . . . le visionnage du DVD. S'il y a une interruption de la séance de cinéma, on sait qu'il faut reprendre la projection au point où elle avait été interrompue. C'est la même chose pour l'exécution d'un programme.

Un processus est donc une entité dynamique qui a une durée de vie limitée (la durée de l'exécution du programme) et dont on peut caractériser le comportement à un certain niveau d'abstraction par un diagramme de transitions d'état. La figure (2.2) illustre les règles comportementales d'un processus en distinguant 3 états :

- Dans l'état actif, le processus exécute des instructions d'un programme. Il monopolise un processeur.
- Dans l'état prêt, le processus est arrêté. Il ne lui manque que la ressource processeur pour devenir actif. Autrement dit, il suffira de lui allouer le (un) processeur et de charger le contexte d'exécution sauvegardé pour que celui-ci poursuive l'exécution du programme dont il a la charge.
- Dans l'état bloqué, le processus est aussi arrêté. Mais, pour se poursuivre, une condition logique devra de plus devenir vraie. Par exemple, le processus interpréteur attendant qu'une ligne de commande soit frappée est bloqué tant que cette ligne n'est pas disponible. La sortie de l'état bloqué pourra conduire le processus soit dans l'état actif, soit dans l'état prêt. Ce choix est fixé par la stratégie d'ordonnancement des processus vis-à-vis de la ressource processeur.

On désigne souvent l'ensemble composé des processus prêts et actifs comme l'ensemble des processus exécutables.

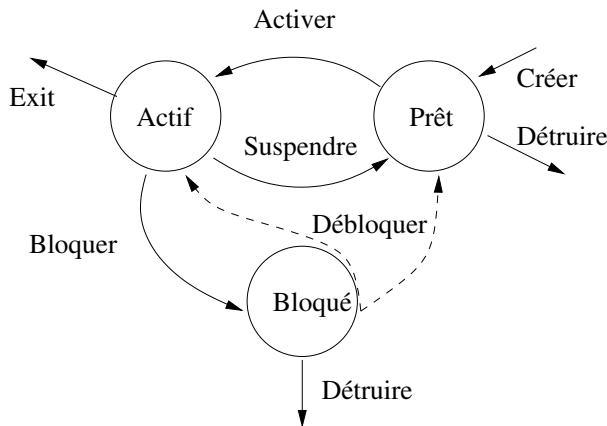


FIGURE 2.2 – Diagramme de transitions d'état d'un processus

2.1.3 Les opérations sur les processus

Le diagramme de transitions fait apparaître les primitives applicables aux objets processus. Celles-ci définissent la classe ou le type des objets processus du point de vue du noyau de gestion des processus.

Créer

La première opération indispensable est l'opération de création d'un processus. Pour le noyau, un objet processus est un objet comme un autre. La création d'un processus apparaît donc comme la création d'un objet de la classe processus avec ses attributs et méthodes (opérations). Parmi les attributs d'un processus, on trouve par exemple, le nom du fichier d'où est issu le programme associé au processus, un nom interne de processus (numéro par exemple), l'état, le contexte initial qui sera chargé dans les registres du processeur pour débuter l'exécution, l'espace mémoire alloué pour le chargement et l'exécution du programme, certains paramètres d'exécution (priorité, délais de garde,...), le nom de l'usager pour lequel le processus « travaille », l'environnement fichier accessible (répertoire de travail), etc. Ainsi, un objet processus peut nécessiter un descripteur de plusieurs Koctets. Dans le système Unix, on distingue même une zone *noyau* du descripteur de processus (structure *sproc*) et une zone *usager* (structure *uproc*). La première est allouée dans l'espace mémoire du noyau, alors que la seconde est dans l'espace mémoire utilisateur.

Lorsqu'on crée un processus, deux approches sont possibles : soit le nouvel objet est créé de toute pièce, soit le nouvel objet est une copie d'un processus courant. Cette dernière approche est par exemple adoptée par le système Unix. Elle présente l'avantage de pouvoir décomposer la création d'un processus en deux étapes : une première étape concerne l'aspect parallèle avec la création d'un processus sosie et une deuxième étape concerne l'aspect traitement (programme) avec la commutation du programme exécuté par le processus sosie. De plus, le processus fils créé hérite naturellement de tout l'environnement d'exécution du processus père (créateur). Ceci évite d'avoir à préciser explicitement un grand nombre de paramètres de création.

Enfin, la gestion globale des processus par le noyau comporte souvent la structuration de l'ensemble des processus existants selon une arborescence fondée sur la relation de création. Cette structure permettra en particulier à un processus père d'attendre la terminaison d'un ou plusieurs de ses fils assurant ainsi une synchronisation globale entre processus.

Activer

Un processus prêt peut devenir actif par l'opération *Activer*. Cette opération consiste essentiellement, à charger dans les registres du processeur le contexte sauvegardé en mémoire. La ressource processeur est ainsi allouée au processus pour exécuter réellement un programme. La période active du processus se termine soit par un blocage, soit par une préemption du processeur au bout d'un délai fixé maximal (voir opération

Suspendre). Cette suspension forcée d'un processus permet de répartir plus équitablement la ressource processeur entre les différents processus prêts candidats.

Suspendre

Si la commutation d'un processus à un autre n'est causée que par le blocage d'un processus actif, un processeur peut être monopolisé par un processus pendant une longue période. Par exemple, si le programme exécuté comporte de longues étapes de calcul sur des données en mémoire centrale, il faudra attendre une lecture ou écriture sur disque pour qu'une commutation se produise. Pour mieux répartir le temps processeur entre les processus, un processeur est en général alloué au processus pour une durée maximale fixée appelée quantum. Si le processus atteint la fin de quantum, alors une interruption est provoquée et le processus actif sur le processeur interrompu est suspendu. Libérant ainsi le processeur, un autre processus prêt peut être activé.

Bloquer

Cette opération comporte une première étape identique à l'opération *Suspendre*. Mais, le processus ne peut plus être candidat à la ressource processeur. Il devra être remplacé dans l'état prêt, éventuellement directement dans l'état actif, par une opération explicite *Débloquer*. Les périodes pendant lesquelles un processus est bloqué correspondent aux périodes pendant lesquelles la poursuite de l'activité du processus est conditionnée par la disponibilité d'une ressource ou l'occurrence d'un événement (fin d'entrée/sortie par exemple).

Cette opération a un effet de bord important : elle libère la ressource processeur pour un autre processus prêt. Par conséquent, elle conduit le noyau à enchaîner par une opération *Activer* (si possible).

Débloquer

Cette opération replace un processus parmi les processus exécutables. Elle est invoquée en général sur occurrence d'un événement asynchrone provoqué par un autre processus, une fin d'entrée/sortie ou une fin de quantum.

Exit

La terminaison d'un programme se traduit par la fin du processus mais pas l'arrêt du processeur. Par conséquent, la fin du programme doit se traduire par un appel explicite au noyau d'exécution via une primitive *Exit*. L'objet processus correspondant pourra alors être détruit sauf contraintes particulières. Une commutation de processus s'ensuivra automatiquement.

Détruire

Si un programme ne se termine pas normalement ou si une raison externe nécessite de détruire un processus (manque de ressource, surcharge du système), on pourra être amené à détruire un processus non actif.

2.1.4 L'ordonnancement des processus

Il faut bien distinguer trois niveaux d'ordonnancement des processus. La figure (2.3) illustre ces niveaux résumés dans le tableau suivant :

Processus	Ordonnancement	Gestion des transitions
Exécutables	à court-terme	Prêt \Leftrightarrow Actif
Synchronisés	à moyen-terme	Exécutable \Leftrightarrow Bloqué
Régulés	à long-terme	Dormant \Leftrightarrow Synchronisé

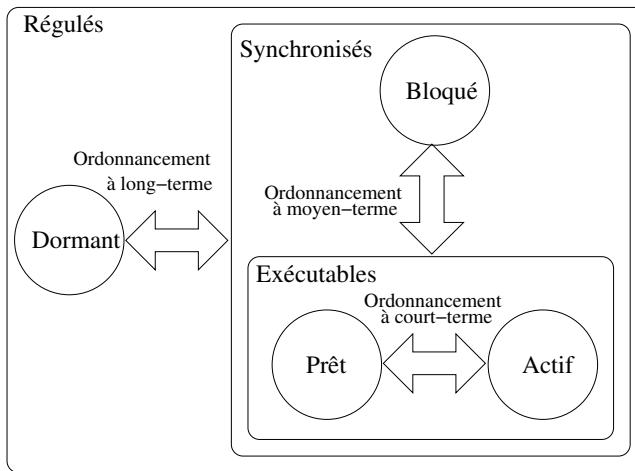


FIGURE 2.3 – Niveaux d'ordonnancement

Le niveau des processus exécutables gère l'ordonnancement à court terme qui consiste à contrôler l'allocation de la ressource processeur. Il s'agit de répartir selon une stratégie d'ordonnancement adéquate le temps processeur disponible entre les différents processus exécutables c'est-à-dire dans l'état prêt ou actif. En général, le noyau gère donc une file d'attente des processus prêts (file simple chronologique *fifo* ou à priorité) et ceux-ci passent dans l'état actif dès qu'un processeur (et par conséquent du temps processeur) est disponible. Le choix d'un processus dans la file détermine ce qui se passe dans le système pour quelques centaines de millisecondes. On parle donc d'ordonnancement à court-terme.

Le niveau des processus synchronisés contrôle le blocage/déblocage des processus intervenant lors de l'allocation/libération des autres ressources que processeur nécessaires à l'exécution du processus. À chaque demande de ressource physique ou logique peut être associée une file d'attente de processus bloqués et la stratégie d'ordonnancement consistera donc ici à choisir quel(s) processus débloquer lorsque la ressource attendue sera disponible. Ce choix a donc un impact sur ce qui s'exécute dans le système dans les quelques secondes qui suivent. On parle d'ordonnancement à moyen-terme.

La figure 2.4 présente un diagramme des événements qui peuvent s'enchaîner sur un processeur unique durant une période d'activité du système : on suppose qu'il existe trois programmes d'usagers en cours d'exécution, soit donc trois processus usagers. Ces programmes font appel aux primitives du noyau, ici pour lire des fichiers (primitive *read*) par exemple. Les interruptions de fin de quantum provoquent des commutations de processus. Une fin d'entrée-sortie signale ici la fin de la lecture demandée par le processus P_1 ce qui permet au noyau de redonner le contrôle du processeur au processus P_1 .

Enfin, le niveau des processus régulés gère l'ordonnancement à long terme qui consiste à contrôler la création/destruction des processus. En effet, la décision de créer un nouveau processus peut être soumise à condition : par exemple, le nombre de processus existants est inférieur à une limite fixée. La création d'un processus peut avoir un impact sur ce qui se passe dans le système pendant une durée longue : si le processus doit exécuter un calcul de plusieurs heures ...

En résumé, l'ordonnancement des processus est une tâche complexe comportant différentes stratégies d'ordonnancement selon le niveau où l'on se place ou selon les ressources mises en jeu. Une difficulté de base réside dans la nécessité de partager équitablement les ressources entre les processus demandeurs. Un processus ne doit pas rester bloquer indéfiniment en attente d'une ressource. De nombreuses recherches ont été réalisées sur ce sujet pour optimiser le flot global de processus exécutés par le système tout en satisfaisant les contraintes temporelles d'exécution (temps de réponse) propres à chaque processus.

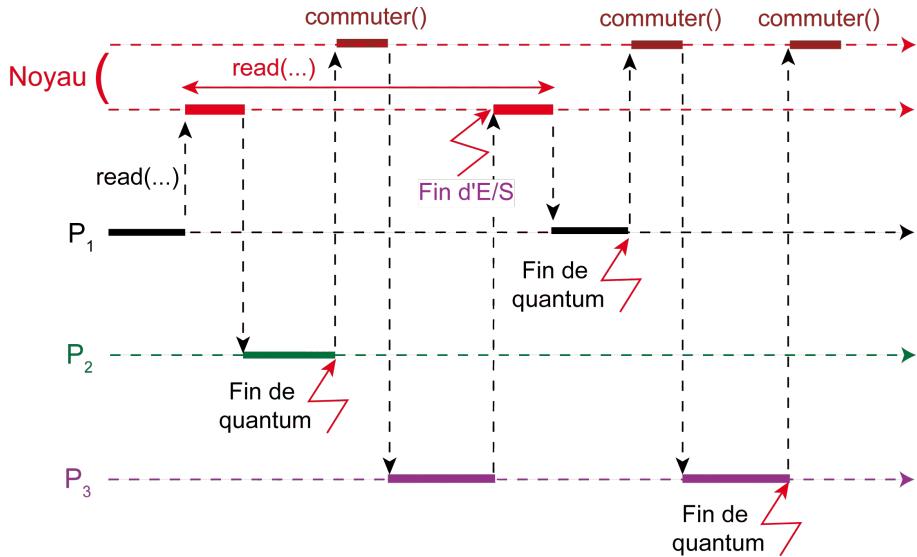


FIGURE 2.4 – Diagramme événementiel de l'activité d'un système pour un processeur fixé

2.2 L'environnement d'exécution d'un processus

Le noyau d'exécution assure et contrôle le déroulement d'un processus de façon à garantir sa bonne exécution sans mettre en danger ni l'exécution des autres processus, ni le fonctionnement correct du noyau lui-même. Pour ce faire, il doit fournir une sorte de « machine virtuelle » la plus portable possible. En effet, il est intéressant de pouvoir écrire des programmes qui s'appuient sur les services offerts par le système d'exploitation plutôt que sur les caractéristiques particulières de tel ou tel processeur ou périphérique. Entre autre chose, la machine « support » de l'exécution du processus doit être capable :

- de traiter les exceptions dues à des erreurs de programmation contenues dans un programme de façon à éviter l'arrêt du système global : seul le processus ayant provoqué l'erreur sera interrompu ;
- de communiquer des événements asynchrones externes vers le processus : par exemple, les interruptions d'entrée/sortie ;
- d'échanger des flots d'informations via les ressources périphériques, fichiers et/ou d'autres processus.

2.2.1 La communication par événements asynchrones

Un processus doit disposer d'un système de gestion d'exceptions lui permettant de réagir à des événements internes à l'exécution même du programme (déroutement) ou à des événements externes telles que les interruptions. Or, les concepteurs de processeurs dotent leurs architectures de systèmes d'exception très propriétaires. Bien que l'on retrouve de nombreux points communs, il existe des différences : nombre de niveaux d'interruptions, nombre exact d'interruptions, masquage ou invalidation des exceptions, etc. Face à cette hétérogénéité des processeurs, les concepteurs de système définissent un système standard d'exception.

La figure (2.5) illustre l'idée d'un système d'exception portable qui interagit avec le processus en cours d'exécution.

L'occurrence d'une exception provoquera une action par défaut : en général, le processus cible est arrêté et détruit. Un code de terminaison peut permettre à un autre processus de tester si le processus s'est terminé normalement ou sur exception.

Néanmoins, le noyau offre une interface de programmation sous la forme d'un ensemble de primitives qui permettent de programmer le système d'exception en offrant par exemple la possibilité :

- de masquer la plupart des exceptions ;

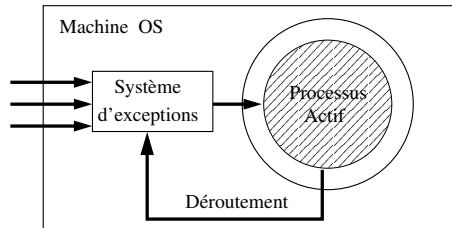


FIGURE 2.5 – Système d’exception portable

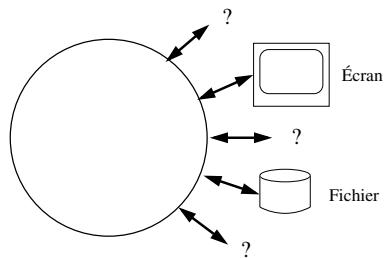


FIGURE 2.6 – Environnement de communication d’un processus

- de connecter une routine de traitement d’exception qui sera exécutée par le processus lui permettant ainsi de continuer son exécution après récupération de l’exception ;
- d’ignorer une exception.

2.2.2 La communication par flots de données

Pour exécuter un programme, un processus va consommer un ensemble de ressources du système : de la mémoire centrale, du temps processeur mais aussi des ressources d’un niveau plus abstrait tel que par exemple des fichiers et/ou les dispositifs d’entrées/sorties plus spécialisés tels que : imprimante, scanner, CD-Rom, etc.

La connexion du processus à ces moyens de communication externe est dynamique. En effet, au cours de l’exécution du programme, celui-ci peut demander l’ouverture d’un flot de données avec un fichier ou une ressource périphérique précise. Le noyau d’exécution doit donc gérer, contrôler et permettre la communication du processus avec son environnement via un ensemble de canaux potentiels. Pour qu’une communication s’établisse réellement, le programme doit contenir des appels au noyau pour ouvrir et connecter ces canaux à une ressource effective qui peut être : un fichier, un (pseudo-)périphérique ou même un autre processus (via un tube ou pipe sous le système Unix par exemple).

Cette approche assure une indépendance (relative, il ne sera pas possible de lire sur une imprimante !) des programmes par rapport à l’environnement réel de leur exécution. En effet, les connexions des canaux peuvent être modifiées sans intervenir dans le code du programme. Cette possibilité est appelée redirection. à titre d’exemple, le système Unix fournit par défaut trois canaux connectés :

- *stdin* qualifiée d’entrée standard est connectée par défaut, pour un processus interactif, au clavier de l’usager connecté,
- *stdout* qualifiée de sortie standard est connectée à la fenêtre de lancement du processus,
- *stderr* deuxième sortie standard, dédiée par convention aux messages d’erreurs, connectée comme *stdout* à la fenêtre de lancement.

Ces trois connexions peuvent être modifiées : par exemple, l’entrée standard peut être redirigée sur un fichier de données et la sortie standard redirigée sur un autre fichier destiné à contenir les résultats.

Une possibilité offerte par ce mécanisme de redirection est de définir un schéma de communication entre processus via la notion de pipe ou tube. Un objet pipe est une ressource permettant de faire communiquer

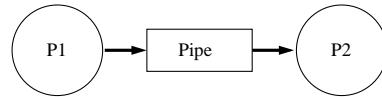


FIGURE 2.7 – Communication par pipe

deux processus selon un schéma de producteur-consommateur. La figure (2.7) illustre cette connexion par pipe entre un processus producteur et un processus consommateur. Le processus producteur écrit via un canal connecté au pipe un flot d'octets. En parallèle, le processus producteur lit et consomme donc le flot d'octets produit. La consommation des données produites se fait dans l'ordre chronologique de production. Bien que le schéma de production-consommation soit asynchrone et parallèle, le producteur peut être bloqué sur la primitive d'écriture si la capacité de mémorisation du pipe est atteinte. En effet, la ressource pipe est essentiellement constituée par un tampon mémoire attribué au pipe par le noyau lors de la création de l'objet. Celui-ci n'est pas de capacité illimitée (quelques Koctets en général). De même, le processus consommateur sera bloqué en lecture si rien n'a été produit lors d'une demande de lecture.

Le mécanisme de pipe permet de réaliser très simplement un traitement parallèle de type pipeline. Une suite de processus exécute des traitements complémentaires où chacun exploite un flot de données entrant et produit un flot résultat sortant consommé par le suivant. Les programmes ayant ce comportement générique de consommateur/producteur peuvent être considérés comme des filtres. Le système Unix offre ainsi en standard un ensemble de commandes de filtrage (*grep*, *tr*, *tail*, *head*, *cat*, *cut*, *join*,...) qui permettent de programmer très rapidement des applications complexes sur des flots de données.

2.3 Conclusion

La notion de processus intervient comme l'objet fondamental géré par un noyau d'exécution. Il permet d'assurer le suivi et le contrôle de l'exécution d'un programme de façon sûre vis-à-vis des autres programmes ou du noyau ainsi que d'exploiter le parallélisme possible entre les traitements en cours optimisant ainsi l'utilisation d'une architecture matérielle.

La définition d'un système d'exception et d'un environnement de communication par flots, indépendant des aspects matériels, assure par ailleurs une bonne portabilité des applications. Les programmes développés peuvent être écrits pour une machine virtuelle « système » plutôt que pour la machine matérielle Sun, Intel ou Motorola.

La gestion des processus pose cependant des problèmes difficiles de programmation parallèle. En effet, ceux-ci entrent en concurrence pour accéder aux ressources du système dont ils ont besoin (mémoire, processeur, fichiers,...) et ils doivent se coordonner pour échanger des données (exemple du schéma producteur-consommateur). Les progrès technologiques aussi bien que ceux enregistrés dans le domaine de la programmation et de l'algorithme, ont conduit à introduire un deuxième niveau de parallélisme : un processus n'exécute plus un seul programme séquentiellement, mais peut dérouler un calcul comportant plusieurs activités parallèles. On parle alors de processus légers (ou threads). Cependant, un processus (lourd par opposition) reste l'unité d'allocation de ressources pour l'exécution d'un programme désormais éventuellement parallèle.

Cet aspect a fait l'objet de longues recherches sur le thème de la synchronisation des processus (lourds ou légers). Ce sujet sera donc développé dans un module spécifique.

Exemple de primitives de création de processus

Côté Windows

Une primitive unique

Nécessité de fournir pas mal de paramètres même si certains peuvent être implicites

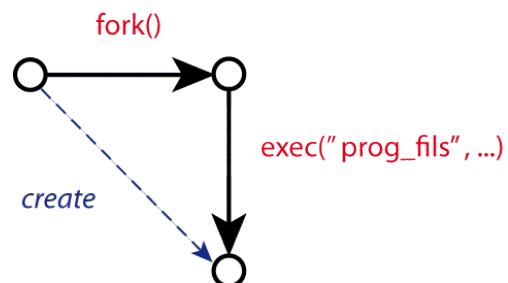
```
BOOL CreateProcess (
    LPCTSTR lpApplicationName, // programme exécutable
    LPTSTR lpCommandLine,     // ligne de commande
    LPSECURITY_ATTRIBUTES lpProcessAttributes
    LPSECURITY_ATTRIBUTES lpThreadAttributes
    BOOL bInheritHandles,    // indicateurs d'héritage
    DWORD dwCreationFlags,   // priorité, nouvelle fenêtre, ...
    LPVOID lpEnvironment,   // environnement
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo, // fenêtre, redirections
    LPPROCESS_INFORMATION lpProcessInformation // résultat
);
```

Côté Unix

Séparation en 2 primitives :

- Crédit du processus fils sosie du père (héritage implicite)
- Commutation de programme : un processus peut changer de programme.

```
if (fork()) /* code exécuté par le fils */
    exec("prog fils", ...);
} else {
    /* code exécuté par le père */
}
```



Chapitre 3

Les fichiers

3.1 Introduction

La notion de fichier est l'abstraction fondamentale introduite dans les systèmes d'exploitation pour gérer les données des utilisateurs acquises ou produites via les divers périphériques connectés : disques, écran/clavier, imprimantes, scanners, modems, etc.

La gestion de ces données comporte deux aspects complémentaires :

- un aspect communication : les données doivent être échangées entre la mémoire centrale où réside le programme en cours d'exécution et les ressources externes ou périphériques qui permettent d'obtenir ou de produire des données ;
- un aspect mémorisation, conservation : les données échangées « appartiennent » à différents usagers qui souhaitent pouvoir retrouver leurs données dans le temps. Cette permanence, rémanence des données par rapport à la durée des traitements (c'est-à-dire des processus) et leur protection contre des accès erronés accidentels ou/et malveillants constituent des éléments essentiels dans la gestion des fichiers.

Comme le montre la figure 3.1, le concepteur d'un système de gestion de fichiers est confronté au problème suivant : comment fournir une interface de programmation simple pour programmer les échanges de données avec un environnement composé de ressources matérielles fort diverses : disques ou équivalents (clé USB), amovibles ou pas, mais aussi imprimantes, scanners, modems, appareil photo, etc ?

Concevoir un système de gestion de fichiers pose les trois problèmes fondamentaux suivants :

- l'organisation, la structuration d'un espace logique de fichiers ;
- l'organisation, la structuration de l'espace physique des ressources ;
- la relation entre ces deux espaces : comment les objets fichiers de l'espace logique sont implantés par l'espace physique des ressources supports ?

Ces ressources présentent des similitudes mais bien des différences, aussi bien dans leur fonctionnalité, on ne peut pas (re)lire des données avec une imprimante, que dans leur capacité de mémorisation : 256Mo pour une clé USB par exemple, 100Go pour un disque, ou la vitesse des contrôleurs (10Mo/s pour une interface IDE par DMA, jusqu'à 640Mo/s pour une interface Ultra-5 SCSI).

Le système de gestion de fichiers va avoir pour tâche première de parvenir à « homogénéiser » toutes ces ressources, en gommant le plus possible leurs différences. On applique en cela le principe de transparence : offrir au programmeur une abstraction, une ressource virtuelle, dont les propriétés sont, le plus possible, indépendantes des spécificités matérielles. Il ne faut pas avoir à modifier un programme parce que l'on veut écrire des données sur une clé USB au lieu de les écrire sur un disque dur. On ne veut pas non plus avoir à connaître sur quels secteurs du disque seront écrites les données.

La figure 3.2 illustre la liaison dynamique d'un processus avec une ressource logique « fichier ». Par un appel de la primitive `open`, le noyau établit une liaison dynamique entre le processus et un fichier. Une fois cette connexion établie, les échanges par les primitives `read` ou `write` sont possibles : pour le programmeur, le fichier apparaît comme une source (en lecture) ou un puits (en écriture) de données. Il peut envoyer ou recevoir un « flot » de données.

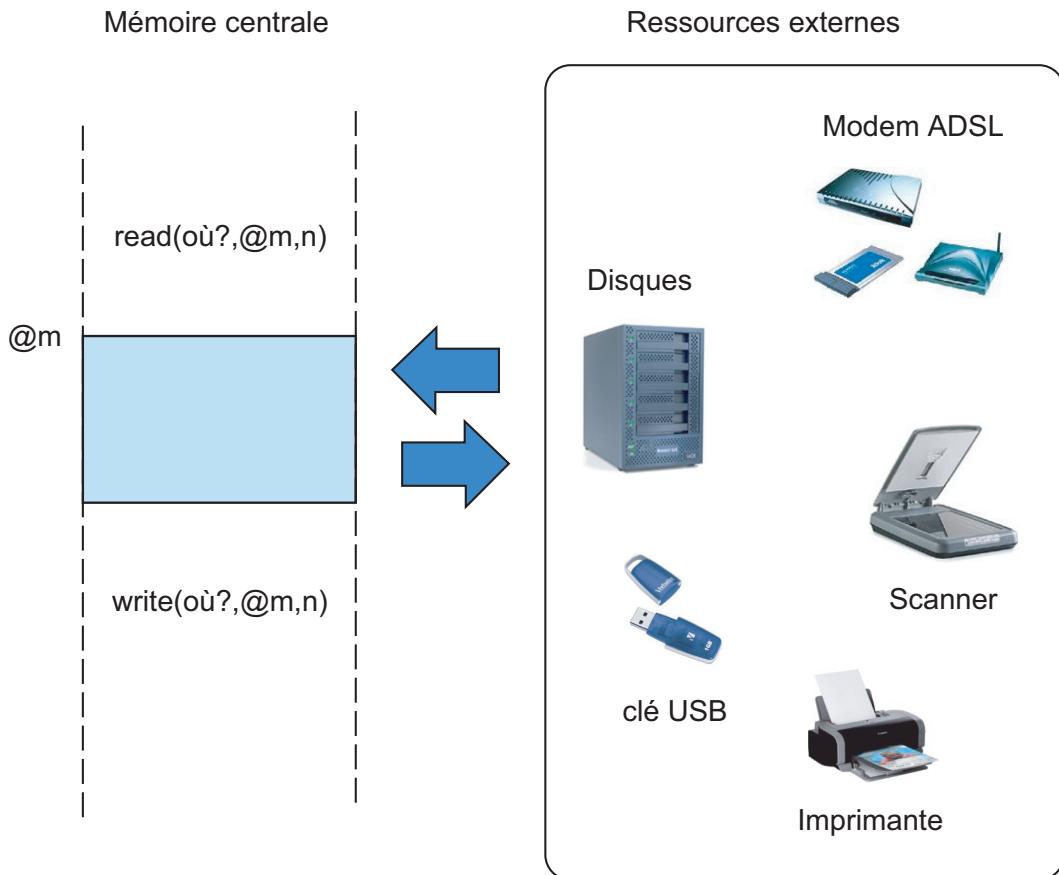


FIGURE 3.1 – La gestion des entrées/sorties

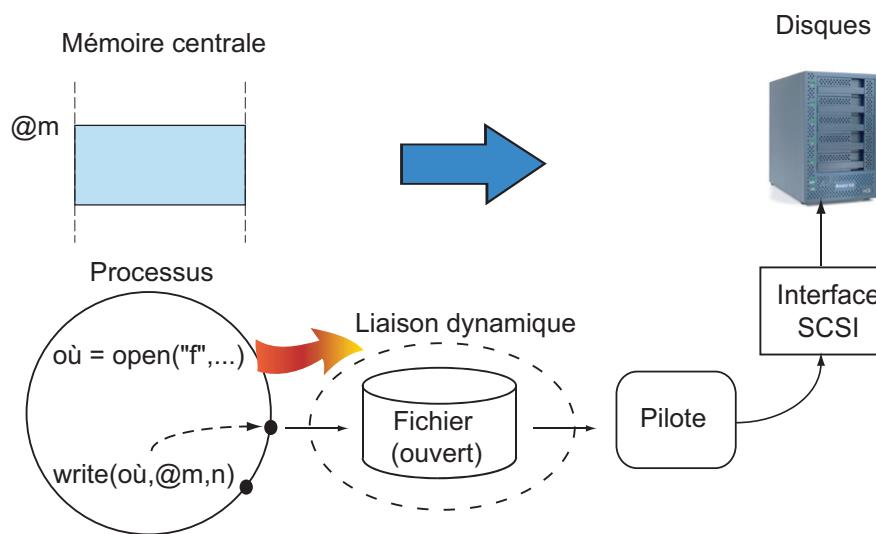


FIGURE 3.2 – Liaison dynamique d'un processus à un fichier

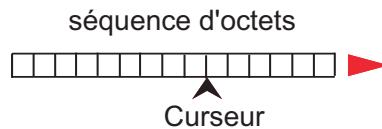


FIGURE 3.3 – Fichier

Le noyau prend en charge la totalité de l'échange jusqu'à la couche physique : le fichier ouvert repère l'implantation des blocs alloués au fichier sur le disque et l'écriture de nouveaux blocs sera réalisée finalement par un pilote (driver) assurant la programmation du contrôleur gérant les accès physiques au disque cible. C'est à ce niveau, et seulement à celui-ci, que la connaissance des caractéristiques matérielles du contrôleur (interface IDE, SCSI, FireWire, USB, etc) et du disque sont prises en compte et masquées au programme usager.

Nous allons voir plus précisément, la structuration interne des systèmes de gestion de fichiers. Auparavant, nous présentons une définition générale de la notion de fichier.

3.2 Le concept de fichier

3.2.1 Définition

Sous sa forme la plus élémentaire, le concept de fichier peut être décrit comme une séquence d'octets dotée d'un curseur et d'opérations permettant de la lire et de la modifier (figure 3.3). Le rôle du curseur est de fixer le point de début de lecture ou d'écriture.

$$\boxed{\text{fichier} = \text{séquence d'octets} + \text{curseur}}$$

Les opérations d'accès aux données d'un fichier constituent la *méthode* d'accès à ce fichier. On distingue plusieurs méthodes d'accès : séquentielle, séquentielle indexée, directe, partitionnée. Elles ont toutes en commun les deux opérations de lecture (`read`) et d'écriture (`write`). Cependant le déroulement d'une opération de lecture ou d'écriture dépendra de la méthode d'accès utilisée.

3.2.2 Méthode d'accès séquentielle

Comme son nom l'indique, une telle méthode d'accès va permettre de lire ou écrire dans un fichier selon une approche séquentielle. Chaque écriture débute à l'emplacement pointé par le curseur courant de lecture ou d'écriture et si l'opération lit ou écrit n octets, la position du curseur est déplacée (incrémentée) de n octets.

On obtient donc la sémantique suivante :

- pour une lecture, la première lecture lit à partir du premier octet (curseur en début de fichier) et chaque lecture suivante commencera après le dernier octet lu par l'opération précédente ;
- pour une écriture, deux possibilités existent lors de l'ouverture,
 - ou bien le curseur est positionné en début de fichier : la première opération d'écriture écrit donc des octets à partir du début du fichier. Si le fichier existait déjà, son contenu va être modifié au fur et à mesure des écritures ;
 - ou bien le curseur est positionné en fin de fichier : la première opération d'écriture écrit donc des octets qui viennent se concaténer à la séquence d'octets qui existait. Bien évidemment, cette deuxième possibilité suppose que le fichier existait déjà et avait un contenu non vide.

Exemple de primitives de gestion du curseur sous Unix

```
int read (n°canal, @m, n)
      write (n°canal, @m, n)
```

Les paramètres précisent :

- le numéro de canal identifiant la connexion dynamique établie entre le processus et le fichier lors de l'opération d'ouverture ;
- l'adresse de la zone accédée en mémoire centrale¹ ;
- le nombre d'octets à lire ou écrire.

La primitive renvoie le nombre effectif d'octets lus ou écrits.

3.2.3 Méthode d'accès directe

Cette méthode d'accès autorise la modification de la position courante du curseur. Avant toute opération de lecture ou d'écriture, le curseur courant peut être déplacé en un point quelconque de la séquence. Celle-ci peut d'ailleurs dans ce cas comporter des zones de contenu « indéfini » (trous) dans la mesure où, par exemple, un programme peut écrire systématiquement n octets et déplacer le curseur courant de $2n$ octets. Le contenu de la séquence sera une sorte de « pointillés » : n octets de contenu, n octets indéfinis, n octets de contenu, etc.

D'un point de vue pratique, pour une telle méthode d'accès, soit on dispose d'une primitive spécifique dédiée à la gestion du curseur, soit un paramètre supplémentaire peut être introduit dans la primitive de lecture ou d'écriture pour préciser la position choisie de début de lecture ou d'écriture.

Exemple de primitives de lecture et d'écriture sous Unix

```
int lseek(n° de canal, pos, origine)
```

Les paramètres précisent :

- le numéro de canal identifiant la connexion dynamique établie entre le processus et le fichier lors de l'opération d'ouverture ;
- la position finale du curseur interprétée différemment selon la valeur du 3-ième paramètre ;
- l'origine utilisée pour fixer la position finale du curseur : soit le début du fichier, soit la position courante du curseur, soit la fin du fichier.

3.2.4 Autres méthodes d'accès

D'autres méthodes d'accès ont été proposées. Elle consiste à voir un fichier comme composé non plus d'une suite d'octets, mais d'une suite d'entités de plus grosse granularité. Deux entités, deux niveaux d'abstraction de données ont en particulier été proposés : l'enregistrement et la partition. Nous décrivons deux exemples.

La méthode d'accès séquentielle indexée

Dans cette méthode, un fichier est vu comme une suite d'enregistrements repérables, de façon associative, par une clé contenue dans l'enregistrement.

fichier = séquence d'enregistrements

Un enregistrement est une structure de donnée quelconque dotée d'un champ clé.

```
struct enregistrement {  
    type_clé clé ;  
    ...  
}
```

En lecture, il est ainsi possible de préciser l'enregistrement désiré en spécifiant sa clé. Afin d'accélérer la recherche d'un enregistrement à partir de la donnée de sa clé, de nombreuses implantations ont été proposées, notamment en introduisant des tables d'index (de clés).

1. Attention : la zone mémoire est écrite lors d'un `read` et lue lors d'un `write`.

La méthode d'accès partitionnée

Dans cette méthode, un fichier est vu comme une suite de partitions : en fait, le contenu d'une partition est en général issu d'un fichier. On pourrait donc aussi voir un fichier partitionné comme un fichier de fichiers. Ce genre de fichiers est surtout utilisé pour créer des archives de fichiers ou des bibliothèques (de binaires rééditables par exemple).

fichier = séquence de partitions

Une partition est donc un « sous-fichier ». Toute partition est identifiable par le nom du fichier origine.

Les opérations d'accès permettent d'ajouter une partition (un fichier), d'extraire/recopier une partition dans un fichier en fournissant son nom, de lister le nom des partitions, etc.

Pour faciliter ces opérations, un fichier partitionné, encore appelé archive, comporte une table répertoriant les différentes partitions sous la forme d'entrées (nom de partition, index de début). Ainsi, la lecture d'une partition donnée débute par la recherche du nom de la partition dans la table répertoire pour trouver le début de la partition recherchée dans le fichier archive.

Exemple de fichiers partitionnés Le système Unix implante la notion de fichiers partitionnés sous la forme d'archives. Une commande de base `tar` permet de gérer des fichiers archives. Par exemple, pour archiver dans un fichier archive `source.tar`, les sources de programmes Java contenues dans un répertoire `projet`, on exécute une commande :

```
tar cvf source.tar projet/*.java
```

avec l'option `c` pour create, `v` pour verbose et `f` pour file.

3.3 Organisation de l'espace logique des fichiers

La figure 3.4 illustre une organisation classique de l'espace logique de fichiers : la structure d'arbre. Les noeuds internes de l'arbre sont appelés répertoires et les feuilles de l'arbre sont les fichiers de données²

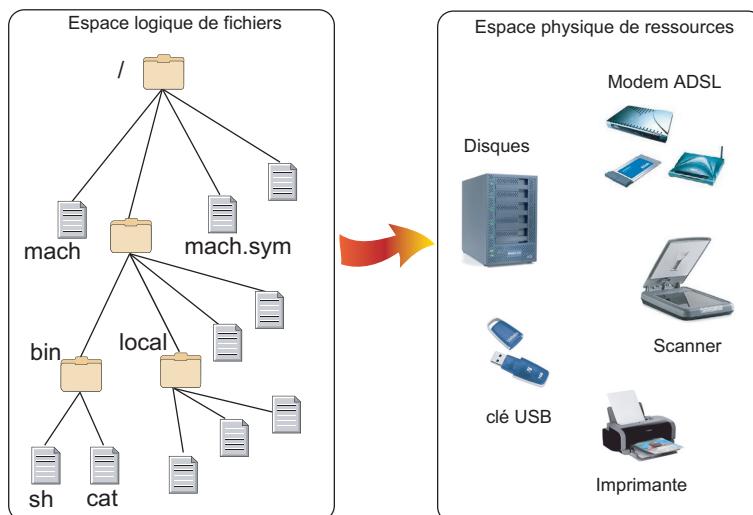


FIGURE 3.4 – Projection de l'espace logique sur l'espace physique

2. Ces feuilles peuvent aussi désigner des ressources physiques ainsi intégrées dans l'arbre de fichiers.

3.3.1 La désignation des fichiers

Cette structuration de l'espace des fichiers en arbre permet de désigner les fichiers en utilisant la notion de chemin d'accès. Les répertoires et les fichiers reçoivent des noms symboliques (chaînes de caractères) et la désignation d'un fichier peut ainsi être définie sous la forme d'un chemin absolu à partir de la racine ou d'un chemin relatif à partir un noeud quelconque de l'arbre, ce noeud étant pris comme racine d'un sous-arbre de fichiers.

Par exemple, dans la figure 3.4, le chemin d'accès `/usr/bin/ssh` désigne sans ambiguïté un fichier exécutable et le chemin relatif `bin/ssh` peut désigner le même fichier à partir du répertoire courant `/usr`.

3.3.2 La protection des fichiers

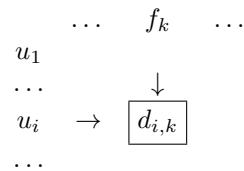
La protection des fichiers est une fonction fondamentale des systèmes de fichiers. Il s'agit d'assurer que le contenu des fichiers et l'existence même de ces fichiers n'est pas remise en cause par n'importe quel usager (\equiv n'importe quel programme exécuté par un usager). En effet, dans une arborescence de fichiers, les fichiers ont pu être créés par différents usagers. Un usager utilisant un système dans le cadre d'une session après une phase d'identification (`login`) possède normalement l'accès à un répertoire de base (accessible dans une variable d'environnement `$HOME` par exemple) à partir duquel il créera des sous-répertoires et fichiers, autrement dit un sous-arbre de fichiers dont le créateur est un même usager.

Protéger des fichiers nécessite donc d'assurer une politique de sécurité d'accès à ces fichiers. Par exemple, les fichiers créés par un usager ne doivent pas pouvoir être détruits ou modifiés par n'importe quel autre usager. A la base de ce problème, deux entités entrent en jeu :

- d'une part, les usagers du système de fichiers ;
 - d'autre part, les objets du système de fichiers, c'est-à-dire les répertoires et les fichiers proprement-dits.
- Ceux-ci sont accessibles via un ensemble d'opérations qui définissent de façon classique leur type.

Le problème à résoudre est donc, pour toute opération *op* (*comment*) sur un objet *obj* (*quoi*) par un usager (*qui*), autoriser ou non l'opération. Le *comment* est défini par une liste d'opérations correspondant au type d'objet (fichier ou répertoire ici), le *quoi* par un nom d'objet et le *qui* par la connaissance du nom d'usager associé au processus tentant exécuter l'opération.

Pour répondre à cette question, il faut donc, pour tout usager et tout objet, fixer et/ou connaître les opérations autorisées, que l'on qualifie habituellement de *droit d'accès* : on a le droit ou non de lire, d'écrire dans un fichier, le droit ou non de traverser un répertoire, d'y créer/détruire un fichier, etc. Ceci peut être schématisé sous la forme d'une matrice (*usager, fichier*) dont les éléments sont les droits d'accès comme l'illustre le schéma suivant.



Il existe deux implantations possibles à partir de ce modèle général. L'une adopte une approche « orientée » ligne, l'autre une approche « orientée » colonne.

L'approche par ligne consiste à associer, à tout usager, un ensemble de droits d'accès aux objets qu'il pourra accéder. On parle alors de capacités. Un fichier ne sera accessible à un processus s'exécutant pour le compte d'un usager qu'à condition de disposer d'une capacité associée à cette objet. La politique de distribution des capacités constitue alors le point critique pour assurer une protection fiable des objets. Par

exemple, un processus ne devra pas pouvoir se forger lui-même des capacités. Seul le noyau décidera de fournir au processus une capacité sur un objet.

L'approche par colonne consiste à associer, à tout objet, l'ensemble des usagers qui peuvent y accéder. Cette approche est simplifiée en structurant les usagers en groupes. On distingue alors généralement 3 classes d'usagers :

1. la classe du propriétaire ayant créé l'objet, ici le fichier ;
2. la classe des usagers appartenant au même groupe que le propriétaire ;
3. la classe réunissant tous les autres usagers. Autrement dit, tous les usagers qui n'appartiennent pas au groupe du propriétaire.

Il suffit donc dans ce cas de distinguer 3 droits d'accès : ceux du propriétaire, ceux attribués au groupe du propriétaire et enfin, ceux attribués aux autres usagers. Si l'on distingue 3 types d'accès pour un fichier : lire, écrire, exécuter, il suffit donc de 3 fois 3 bits pour décrire les droits d'accès de tous les usagers à un fichier donné.

Mise en œuvre du contrôle d'accès On envisage le cas d'une approche orientée colonne. Lorsqu'un processus tente de connecter un canal à un fichier via la primitive `open`, le noyau va décider d'accepter l'ouverture du fichier à partir des paramètres suivants :

1. pour quel usager « roule » (s'exécute) le processus appelant ?
2. quel type d'accès est demandé ?
3. quels sont les droits d'accès à ce fichier ? Ceux-ci sont accessibles dans le descripteur de fichier.

Le contrôle consiste alors à vérifier, à partir des droits d'accès spécifiés, que l'usager est autorisé à accéder au fichier avec le type d'accès qu'il demande. Pour cela, il est nécessaire de savoir quel est la classe de l'usager tentant un accès au fichier. Or, celle-ci dépend de l'identification de l'usager qui a créé le fichier, qui est donc, de fait, le propriétaire du fichier. C'est pourquoi, l'identité du propriétaire du fichier est mémorisée dans le descripteur de fichier.

Cette stratégie de contrôle d'accès présente cependant une insuffisance importante pour une classe relativement fréquente d'applications, celles mettant en jeu des données rémanentes partagées par différents usagers clients, ce qu'on appelle classiquement une base de données.

Exposé du problème On considère des applications gérant des données concernant des stocks, des commandes, des réservations, etc. Autrement dit, ce sont des applications qui gèrent une base de données partagée.

Le développement de telles applications nécessitent

- d'une part de définir un ensemble d'informations structurées qui seront conservées dans des fichiers ;
- d'autre part, de développer un ensemble de programmes applicatifs pour manipuler ces données et assurer leur mise-à-jour cohérente.

Un tel développement va conduire à un ensemble de fichiers contenant des données $D = \{f_1, \dots, f_n\}$ et à un ensemble de fichiers contenant des programmes exécutables de type transactionnel³ $T = \{t_1, \dots, t_m\}$. La figure 3.5 illustre les composants du problème.

Analyse du problème On peut supposer que, lors du développement de l'application, les programmeurs appartiennent au même groupe d'usagers. La protection d'accès à appliquer est donc simple :

3. On appelle transaction, un traitement présentant des garanties d'atomicité et de persistance des informations accédées vis-à-vis des fautes éventuelles pouvant survenir durant son exécution.

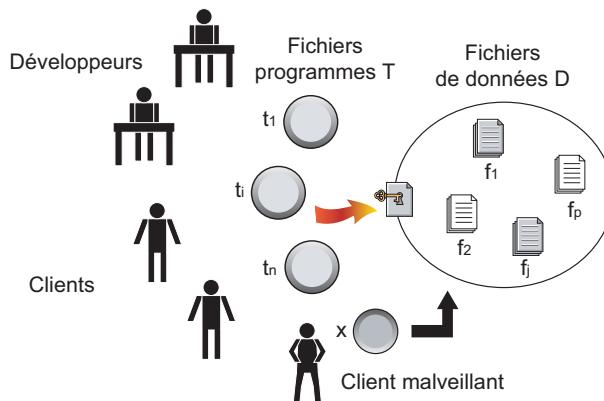


FIGURE 3.5 – Problème de protection des données d'une application

- autoriser l'accès aux fichiers de données par tout membre du groupe de développeur et interdire tout accès aux autres usagers ;
- autoriser l'accès en exécution aux fichiers de programmes par tout membre du groupe de développeur et interdire tout accès aux autres usagers.

Lorsqu'un développeur lance l'exécution d'un programme, le processus a pour usager, ce développeur qui, appartenant au groupe des développeurs, peut donc accéder aux fichiers de données.

Néanmoins, une fois mis au point, les programmes développés doivent pouvoir être exécutés par des usagers clients de l'application quel que soit leur identité. Il est, en effet, peut intéressant de réservier l'usage de l'application finale aux seuls développeurs de celle-ci !

Pour cela, on va donc modifier les droits d'accès aux fichiers programmes de l'ensemble T en autorisant leur exécution par tout usager. Mais cette modification ne résout pas complètement le problème : en effet, si un usager n'appartenant pas au groupe des développeurs tente d'exécuter un des programmes t_j , lorsque le processus correspondant tentera d'accéder à un fichier de données, l'opération d'ouverture échouera : le processus appelant ne travaille pas pour un usager du groupe des développeurs et donc tout accès au fichier de données lui est interdit.

On pourrait alors envisager de donner le droit d'accès aux fichiers de données de l'ensemble D pour tous les usagers. Le programme exécuté pourrait donc accéder cette fois-ci aux fichiers de données dont il a besoin. Mais, dans ce cas, non seulement les programmes de l'ensemble T peuvent accéder aux fichiers de données D , mais **n'importe quel programme écrit par un usager quelconque** le peut aussi. Autrement dit, on ne peut garantir que les données ne seront accédées que via les programmes validés et écrits par les développeurs.

Une solution Le problème à résoudre est de garantir que seuls les programmes validés pourront exécuter des accès aux fichiers de données même lorsqu'ils sont exécutés par des usagers qui n'appartiennent pas au groupe des développeurs. La stratégie adoptée consiste à provoquer un changement d'usager du processus exécutant tout programme de l'ensemble T . Le processus prend temporairement l'identité du développeur propriétaire du fichier programme qu'il exécute. Par conséquent, quelle que soit l'identité initiale de l'usager ayant créé le processus, celui-ci commutera d'usager courant lors d'une exécution du fichier applicatif. Cette commutation d'usager ne sera effectuée que si un bit indicateur spécifique a été positionné dans le descripteur du fichier programme.

A titre d'exemple, sous Unix, il existe en fait deux bits indicateurs : l'un pour déclencher la commutation d'usager et l'autre pour déclencher la commutation de groupe d'usagers. Le processus peut ainsi indépen-

damment commuter d'usager et/ou de groupe avant d'exécuter un programme.

Grâce à ce mécanisme, il est possible de conserver l'interdiction d'accès aux fichiers de données D par les usagers clients via des programmes quelconques tout en assurant que ces mêmes clients pourront utiliser les programmes applicatifs validés qui constitueront bien pour eux les seules opérations possibles utilisant les données applicatives contenues dans les fichiers de D .

3.4 Organisation de l'espace physique des ressources

On aborde ici le problème de la gestion des ressources physiques qui pourront être utilisées comme support de mémorisation des données des fichiers. Un principe de conception essentiel consiste à banaliser le plus possible les ressources physiques en masquant leurs caractéristiques en terme de capacité, de performance d'accès, de type d'accès, etc.

3.4.1 La désignation des ressources matérielles ou périphériques

Une première idée et étape consiste à intégrer les ressources physiques dans l'espace logique même des fichiers. Cette approche apporte deux avantages : d'une part, les ressources matérielles peuvent être désignées selon les mêmes règles que les fichiers c'est-à-dire dans le même espace de noms et, d'autre part, l'accès à une ressource pourra se faire en utilisant les méthodes d'accès des fichiers à condition, toutefois, d'en avoir le droit : en effet, un usager ne doit pas avoir la possibilité d'écrire un volume disque séquentiellement, secteur par secteur.

Exemple Unix Les ressources matérielles ou périphériques sont localisées dans le répertoire `/dev`. Les ressources sont identifiées par un nom de base précisant leur nature et fixant le type de leur pilote, suivi d'un numéro. Par exemple, `tty01` désigne un « terminal usager » connecté à l'entrée standard (`stdin`) et à la sortie standard (`stdout`), c'est-à-dire la plupart du temps le clavier et une fenêtre.

Les unités de disque peuvent apparaître sous des noms divers associés souvent à leur interface de connexion, `scsi` par exemple.

Remarque La ressource mémoire centrale elle-même est intégrée dans l'espace de noms des fichiers : `mem` désigne la mémoire usager, `kmem` celle du noyau.

3.4.2 La connexion des périphériques

Il existe une très grande diversité de périphériques. Il est possible de les caractériser (mais aussi distinguer) par leurs propriétés : capacité, vitesse de transfert, support final de l'information, sens des échanges, mode de transfert (par bloc ou par octets).

On peut considérer qu'il existe 2 classes de périphériques :

- les périphériques qui ne constituent que des sources pour capter des données (clavier, scanner, carte réseau, etc) ou des puits pour sauvegarder, diffuser des données (imprimante, table graphique, carte réseau, etc).
- les périphériques qui sont capables de mémoriser des informations pour qu'elles puissent être accéder ultérieurement, appelés historiquement mémoire de masse : les disques, les lecteurs/graveurs de CD's ou DVD's, les clés USB, les dérouleurs de bandes magnétiques, etc. Ce type de périphérique constitue le support privilégié des systèmes de fichiers.

On peut distinguer de plus, les périphériques qui acceptent le chargement dynamique des supports de mémorisation qui sont amovibles (par exemple, un lecteur/graveur de DVD's) de ceux dotés d'un support

de mémorisation fixe : disques de grande capacité. Par ailleurs, l'unité périphérique elle-même peut être connectée de façon permanente ou temporaire (pour une clé USB, un appareil photo numérique, caméra numérique par exemple, la connexion est la plupart du temps dynamique pour une période courte).

Historiquement, la connexion des périphériques était très statique. L'ajout/retrait d'une unité nécessitait l'arrêt préalable du système, éventuellement sa reconfiguration (ajout éventuel du pilote correspondant à la nouvelle unité) et un redémarrage. Aujourd'hui, avec les unités amovibles de type clé USB, la connexion d'une nouvelle unité est une opération simple ne nécessitant ni arrêt, ni reconfiguration. Seule la connexion d'une unité d'un type nouveau dont le pilote n'est pas présent dans la configuration courante nécessite une installation et un redémarrage (en général) du système.

La programmation des différents types de périphériques passe par des interfaces standardisées caractérisées, en particulier, par leur vitesse de transfert, le protocole d'échange synchrone ou asynchrone, leur type série ou parallèle. Pour les connexions d'unités de disques, on utilise surtout les interface parallèles SCSI (avec ses nombreuses variantes Ultra-5 à 640 Mo/S, Ultra-4 à 320 Mo/s, Ultra-3 à 160 Mo/s, ...) ou FireWire (norme IEEE1394 de 50 à 100Mo/s). L'interface USB de 12 et 60Mo/s est devenu très populaire pour la connexion de nombreux types de périphériques (clavier, appareils photos numériques, clé du même nom, ...). Le protocole de transfert SCSI parallèle gère très bien les périphériques multiples : plusieurs disques connectés en série. Le point fort du Firewire est son caractère isochrone bien adapté aux échanges de flux vidéo⁴.

Pour qu'un périphérique soit utilisable, il faut qu'il puisse être connecté via une des interfaces disponibles de l'architecture matérielle et que le logiciel de programmation des échanges, appelé pilote ou driver en anglais, soit disponible et installé dans le noyau. Un pilote implante les opérations d'échange et de contrôle du périphérique. Il se décompose en deux parties :

- les opérations appelées via les programmes usagers : lorsqu'un programme appelle la primitive `read`, celle-ci va finalement sous-traiter l'opération effective au pilote qui contient la routine de lecture adaptée au périphérique adressé ;
- les routines de traitement d'interruption de fin d'entrées/sorties : l'échange mémoire ↔ périphérique se faisant en parallèle avec l'exécution des processus sur l'unité centrale (les unités centrales), la fin d'un échange doit être signalée par le contrôleur du périphérique. Pour ce faire, on utilise les interruptions. Lorsqu'un échange se termine, le contrôleur émet une interruption vers l'unité centrale. Le pilote doit donc contenir les routines de traitement d'interruption correspondantes.

Généralement, un pilote peut gérer un type de périphériques. Autrement dit, si plusieurs périphériques de même type sont connectés, un seul pilote existe. Une adresse passée en paramètre des opérations permet d'identifier l'unité concernée.

3.4.3 La notion de volume

La notion de volume est introduite pour masquer la grande diversité des supports utilisables comme mémoires de masse des systèmes de fichiers. Un volume est une abstraction de la notion de disque : il peut être vu comme un tableau de blocs ou fragments (paquets de blocs) de taille fixée.

$$\boxed{\text{volume} = \text{blocs}[capacité] \text{ ou } \text{fragments}[capacité]}$$

La représentation d'un volume pourra être une ressource donnée : disque, CD, DVD, clé USB, ou seulement une partition d'une unité dans le cas des disques de grandes capacité. L'unité physique est découpée en plusieurs partitions, zones, supports de différents volumes. Le partitionnement d'un disque

4. Le mode de transfert isochrone permet de garantir des temps de transmission constants. Il fonctionne par réservation d'unités d'espace-temps de dimension données et cycliques. Il n'y a donc plus besoin d'accusés de réception, l'échange utilisant une ressource « espace-temps » réservé dont le débit et la bande passante sont fixes et garantis.

en plusieurs volumes est une opération de configuration qui doit être réalisée préalablement à l'usage de l'espace disque.

La propriété fondamentale d'un volume est de pouvoir être le support d'un système de fichiers : autrement dit, une sous-arborescence de fichiers pourra être déployée sur un volume. Pour ce faire, une opération d'initialisation du système de fichiers sur le volume est nécessaire. Cette opération définit le nom du système de fichiers, les restrictions d'accès, l'espace libre disponible, la table des descripteurs de fichiers, des dates (de création, de dernière modification). A titre d'exemple, la commande `mkfs <nom de volume>`, sous Unix, permet d'initialiser un système de fichiers sur un volume.

3.5 La projection de l'espace des fichiers sur l'espace des ressources

Nous nous intéressons ici à l'implantation d'un sous-arbre de fichiers sur un volume. On parle souvent dans ce cas d'un (sous-)système de fichiers ayant pour support l'espace disque offert par le volume. Il s'agit donc de représenter l'ensemble des fichiers de l'arbre y compris les répertoires.

3.5.1 Principes généraux de conception

Stuart E. Madnick a proposé en 1970 une structuration en couches pour l'implantation des systèmes de fichiers[4]. Le tableau ci-dessous présente cette structuration :

Abstraction	Niveau	Fonction
Répertoires	6	Désignation (chemins via les répertoires)
	5	Localisation des descripteurs
Fichiers	4	Contrôle d'accès (listes ou capacités)
	3	Accès au contenu (lire, écrire)
Blocs	2	Allocation et caches
	1	Gestion des échanges

3.5.2 Le niveau répertoire : implantation de l'arborescence de fichiers

L'implantation de la structure d'arbre repose sur une idée simple : les noeuds, c'est-à-dire les répertoires, sont représentés par des fichiers. Autrement dit, **nœuds ou feuilles, tout est fichier**. Pour obtenir la structure d'arbre, le contenu d'un répertoire est donc tout simplement un tableau de couples (nom, i) dans lequel le nom est une chaîne de caractères identifiant le fichier et l'indice repère le descripteur de fichier associé.

Cette approche implique que l'espace des fichiers proprement dit est représenté sous la forme d'un simple tableau (ou table) de descripteurs (appelés **i-node** pour « internal node » sous Unix). Lorsqu'un chemin d'accès est fourni en paramètre de la primitive `open`, il faut donc éventuellement accéder à plusieurs répertoires, donc fichiers jusqu'à obtenir le numéro de descripteur final du fichier cible.

3.5.3 Le niveau fichier : implantation des fichiers

D'après ce qui précède, un volume n'est donc que le support de fichiers. Il contient donc seulement une table de descripteurs de fichiers et il suffit de connaître le numéro de ce descripteur pour pouvoir accéder au fichier.

Exemple de descripteur sous Unix : le **i-node** Lorsqu'un fichier est créé, il faut donc réserver un descripteur libre. Au fur et à mesure que des données vont être écrites dans le fichier, il faudra allouer de l'espace disque pour recevoir ces données. La solution retenue est une allocation dynamique par blocs ou fragments de taille multiple d'un secteur puisque la taille d'un fichier n'est pas initialement connue.

Ceci pose naturellement deux problèmes :

- les blocs étant alloués dynamiquement, ils ne sont pas contigus sur le disque. Par conséquent, il faut gérer une carte donnant pour chaque numéro logique de bloc de fichier, le bloc physique lui correspondant. La taille d'un fichier n'étant pas connue, la taille d'une carte ne l'est pas non plus.
- Il est nécessaire de connaître les blocs libres. Il faut donc une structure de données permettant de connaître les numéros de blocs libres.

Selon les systèmes de fichiers, les solutions ne sont pas tout à fait les mêmes. A titre d'exemple, nous présentons les solutions Windows et Unix.

L'implantation Windows

L'idée de base consiste à utiliser un tableau unique pour toutes les cartes, ce tableau ayant un nombre d'entrées égal au nombre de blocs disque utilisables. Pour des raisons de robustesse, ce tableau est dupliqué.

Ce tableau sert à planter un chaînage de type liste entre les blocs logiques successifs d'un fichier. En effet, le nombre d'entrées étant égal au nombre de blocs physiques, l'indice d'une entrée peut être utilisé comme numéro de bloc physique et, dans une entrée occupée i du tableau (associé au bloc i), on trouve la localisation du bloc logique suivant de i . En connaissant le numéro du premier bloc physique du fichier, il est donc possible de retrouver tous ses blocs. La figure 3.7 illustre cette représentation des cartes.

Cette implantation présente l'avantage de mixer à la fois les cartes des fichiers et la carte des blocs libres : en effet, tout bloc libre peut être repéré dans le tableau par une entrée ayant une valeur codant l'état libre. Elle a le désavantage d'être en fait une structure de liste et donc d'être peu performante vis-à-vis des méthodes d'accès direct : il faut parcourir la liste jusqu'au bloc demandé.

L'implantation Unix

L'idée de base est de placer une carte de taille fixe dans chaque descripteur de fichier. Une carte comporte 13 entrées : les 10 premières peuvent pointer les 10 premiers blocs du fichier s'ils existent. Typiquement, tous les blocs d'un petit fichier (une dizaine de Ko) sont donc directement accessibles. Ensuite, la 11-ième entrée pointe un bloc qui contiendra un nombre fixe de pointeurs vers les blocs suivants du fichier : par exemple, si les blocs ont une taille de 4K octets, on pourra placer dans un tel bloc, les pointeurs vers les K blocs suivants du fichier. La 12-ième entrée adopte le même schéma mais avec une double indirection. Finalement, l'usage du 13-ième conduit à 3 indirections successives. La figure 3.8 illustre cette représentation des cartes.

Bien que cette approche permette de représenter des fichiers de taille importante, une taille maximale existe. Les concepteurs d'Unix avaient adopter cette approche compte-tenu du fait que le système Unix était pour eux un support au développement de programmes. C'est pourquoi, 90 % de leur fichiers étaient

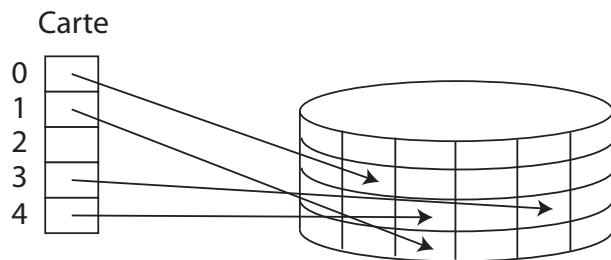


FIGURE 3.6 – Localisation du contenu d'un fichier : notion de carte

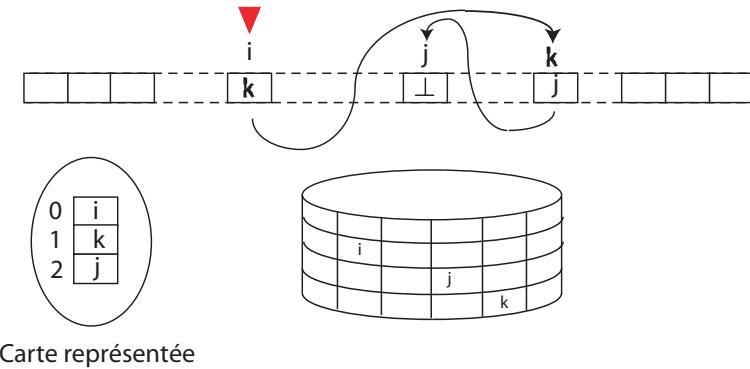


FIGURE 3.7 – Localisation du contenu d'un fichier : approche Windows

Carte à 13 entrées

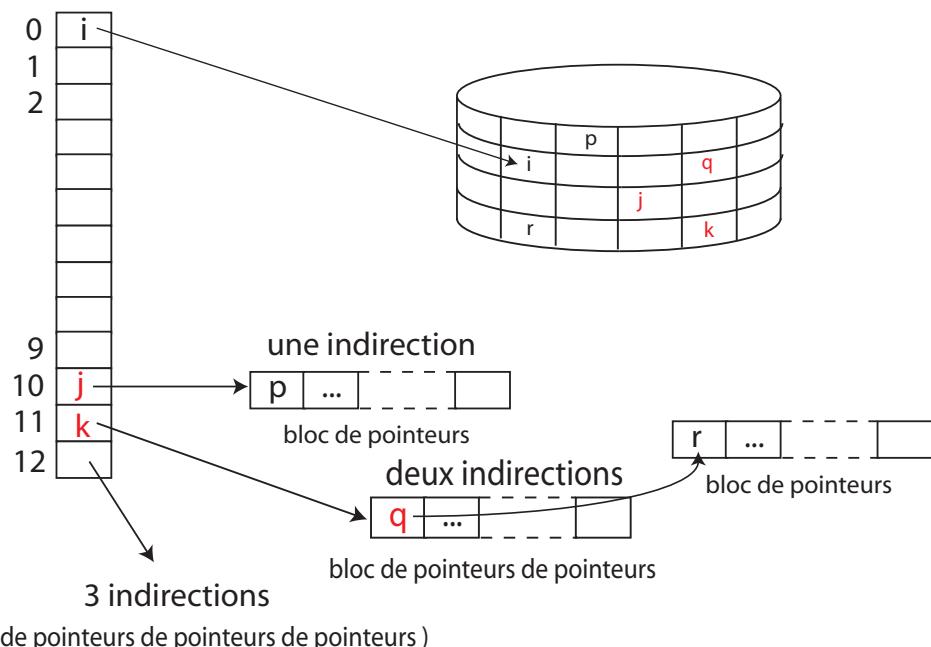


FIGURE 3.8 – Localisation du contenu d'un fichier : approche Unix

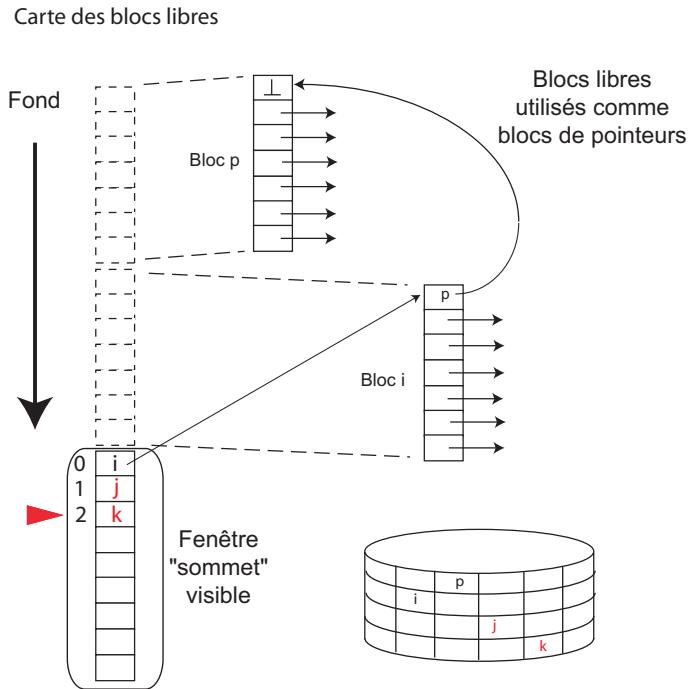


FIGURE 3.9 – Carte (pile) des blocs libres : approche Unix

de petite taille et n’impliquaient que rarement l’usage des indirections.

Par contre, aujourd’hui, l’usage de fichiers contenant de la vidéo par exemple conduit souvent à de très importants volumes de données. Par conséquent, cette approche n’est plus adaptée.

3.5.4 Le niveau bloc : la gestion de l’allocation/libération d’espace aux fichiers

L’allocation dynamique des blocs impose, nous avons vu, de connaître la « cartographie » des blocs libres sur un volume. Le système Windows, avec une table de taille égale au nombre de blocs combine les cartes de chaque fichier ainsi que celle des blocs libres. En effet, lorsqu’un bloc *i* est libre, l’entrée correspondante contient une balise spécifique indiquant que le bloc est libre. Le système Unix ayant opté pour des cartes implantées dans chaque descripteur de fichier, doit donc par ailleurs gérer une carte des blocs libres. Cette carte est localisée dans le descripteur du volume appelé « super-bloc ».

La structure de donnée utilisée est une pile dont la représentation se répartit dynamiquement dans les blocs libres eux-mêmes au fur et à mesure des allocations/libérations de blocs. Le descripteur de volume contient en fait une fenêtre « sommet » visible de la pile. La taille de cette fenêtre est exactement égale au nombre de numéros de blocs que l’on peut mémoriser dans un bloc. En effet, des blocs libres interviennent dans la représentation de la pile pour contenir des numéros de blocs libres. Ceux-ci sont chaînés dans une liste.

La figure 3.9 illustre cette représentation éclatée de la pile des blocs libres. Seule une fenêtre de taille fixe localisée dans le super bloc contient une partie sommet visible de la pile. Le premier élément (base) de cette fenêtre contient un numéro de bloc libre qui, lui-même, ne contient que des numéros de blocs libres. Lorsque la fenêtre ne contient plus que ce dernier numéro de bloc *i* et qu’une demande d’allocation d’un bloc arrive, la fenêtre est re-remplie avec le contenu du bloc *i* et ce bloc *i* est alloué pour satisfaire la requête. L’état de la fenêtre est alors à nouveau « plein » et permettra de satisfaire d’autres requêtes futures d’allocation. Si, par contre, lors d’une libération d’un bloc *j*, la fenêtre est trouvée pleine, le bloc libéré *j* est rempli avec le

contenu de la fenêtre (pleine) et la fenêtre ne contient plus alors, qu'à sa base, le numéro du bloc libéré j . A partir de cet état, d'autres blocs libérés peuvent donc être à nouveau empilés dans la fenêtre.

3.5.5 La destruction des fichiers

L'usage d'un fichier obéit à une sorte de protocole standard constitué d'une séquence d'appels de primitives.

Fonctionnalité	Exemple de primitives sous Unix
Connexion entre le processus et la ressource fichier	<code>int c = open("nom-fichier", O_RDONLY)</code>
Accès en lecture ou écriture via la connexion ouverte	<code>int nblus = read(c,zone, n)</code>
Déconnexion du fichier (fermeture de la liaison)	<code>int nbegr = write(c,zone, n)</code> <code>close(c)</code>

Aucune opération de destruction n'apparaît. Au niveau du langage de commande, la destruction d'un fichier est souvent assimilée à la commande `rm` ou `delete`. En fait, une telle commande a pour fonction première d'extraire le nom du fichier fourni en paramètre du répertoire où il se trouve. La destruction du fichier sera automatiquement assurée par le noyau du système lorsque deux conditions seront satisfaites pour un fichier donné F :

- Aucune connexion de processus n'existe sur ce fichier F .
- Le fichier est devenu anonyme : autrement dit, aucun répertoire ne contient d'entrée pointant le descripteur du fichier F ;

Si ces deux conditions sont satisfaites, le fichier n'est plus utilisé et plus utilisable puisqu'anonyme. Par conséquent, le noyau peut récupérer (libérer) les éléments intervenant dans la représentation du fichier, en l'occurrence, son descripteur et les blocs contenant les données du fichier.

Cette approche consistant à automatiser une opération en confiant cette opération au noyau plutôt qu'à l'usager est un principe fondamental pour la gestion des objets partagés rémanents. Rappelons que le même principe est adopté pour la destruction des pipes par exemple.

3.5.6 La gestion des échanges

L'optimisation de la gestion des échanges mémoire centrale ↔ périphériques est un point crucial d'un système de gestion de fichiers. En effet, de tels échanges (classiquement appelés entrées/sorties) peuvent se dérouler en parallèle avec les processus s'exécutant sur le ou les processeurs. Il s'agit donc de maximiser le parallélisme possible en essayant d'optimiser l'usage des canaux d'entrées/sorties.

Cette optimisation consiste, en entrée, à « prévoir » les données qui vont être lues par les processus : de cette façon, lorsque le processus émet une demande de lecture, l'opération peut être très rapide puisque réduite à une recopie en mémoire centrale d'une zone source à une zone destination. Cette approche est réalisable dans tous les cas d'accès séquentiel à l'information.

En sortie, lorsqu'un processus émet une requête d'écriture, celle-ci se traduit par l'écriture des données dans un tampon (des tampons) mémoire(s) du noyau. Ainsi, le noyau pourra optimiser l'exécution de ces requêtes d'écriture, notamment en les ordonnancant de manière la plus efficace en terme de temps d'exécution.

Dans les deux cas, le noyau utilise un espace mémoire composé de blocs de taille adaptée aux caractéristiques physiques des disques. Cet espace, appelé cache, est partagé entre les processus demandeurs et les processeurs d'entrées/sorties qui exécutent les échanges.

Ce type de gestion des échanges n'exclut pas l'usage de primitives exécutant les accès de manière directe sans passer par des caches.

3.6 Conclusion

Le système de gestion des fichiers représente un module très volumineux et complexe d'un système d'exploitation. Historiquement, partie intégrante du noyau d'un système d'exploitation, les concepteurs ont réussi à les considérer comme de simples services. Cette technologie des noyaux a conduit au développement de « micro-noyaux » (encore appelés kernels ou nucleus) ne contenant pas de système de gestion de fichiers à proprement parler. Un tel noyau ne contient plus que la gestion des processus, de la mémoire virtuelle et de la communication entre processus. Un service de fichiers est implanté par un processus dédié auquel s'adresse, par requête, tous les processus utilisant des fichiers.

Un avantage important d'une telle approche est qu'elle offre la possibilité de faire cohabiter plusieurs systèmes de fichiers de façon très simple : il suffit d'activer plusieurs processus serveurs dédiés aux fichiers. Citons les systèmes Chorus ou Mach comme exemples de tels micro-noyaux⁵.

Enfin, les systèmes de fichiers sont en étroite relation avec les systèmes de mémoires virtuelles : en effet, le contenu d'une mémoire virtuelle est défini à partir d'une image exécutable d'un programme contenue dans un fichier. Au cours de l'exécution d'un processus, le contenu des pages virtuelles sera représenté en mémoire réelle mais aussi en mémoire secondaire dans les systèmes de pagination à la demande. Dans ce cas, une zone de sauvegarde, appelée zone de swap, est alors souvent implantée sur un volume dédié à cet usage.

Les mécanismes de mémoire virtuelle peuvent même être « détournés » pour lire/écrire un fichier : en effet, puisque le système de mémoire virtuelle implante comme primitive de base le couplage du contenu d'un fichier dans une mémoire virtuelle, il suffit d'exploiter cette fonctionnalité pour accéder à un fichier : le fichier, une fois couplé en mémoire virtuelle, toute écriture ou lecture dans la zone d'adresse correspondante équivaut à écrire/lire dans le fichier. Le système de mémoire virtuelle se charge de façon transparente d'exécuter les échanges nécessaires entre la mémoire réelle et le fichier sans aucune instruction de la part du programme usager.

5. Le terme micro-noyau est tout relatif : le noyau en question reste volumineux et complexe.

Chapitre 4

Les mémoires virtuelles

4.1 Mémoires virtuelles

L'utilisation optimale de la mémoire centrale d'un ordinateur a fait l'objet d'études approfondies dès l'origine de l'informatique. En effet, la taille de la mémoire centrale a constitué pendant longtemps une ressource « rare ». Si aujourd'hui, le paysage est un peu changé, les technologies développées restent néanmoins à l'ordre du jour pour d'autres bonnes raisons.

4.1.1 Le problème du partage mémoire

Dans tout système même monoprogrammé, la mémoire centrale doit être partagée entre plusieurs programmes (au moins le noyau et un programme utilisateur). Le résistance du noyau aux fautes du programme usager passe par une protection efficace de la zone mémoire contenant le noyau. Il faut donc disposer de mécanismes de protection mémoire pour protéger les programmes les uns des autres.

De façon plus générale, la multi-programmation implique le chargement en mémoire de plusieurs programmes. Le placement en mémoire centrale de ces programmes pose plusieurs problèmes :

- la protection réciproque de ces programmes les uns par rapport aux autres doit être assurée. Lorsqu'un processus exécute l'un des programmes, les seules adresses acceptables doivent être celles confinées à la zone mémoire explicitement allouée au programme et contenant les différentes sections du programme : section(s) de code, section(s) de données statiques, section pile-tas. Il faut donc disposer d'un mécanisme de protection mémoire permettant au noyau de fixer une zone mémoire référencable avant toute activation d'un processus ;
- l'adresse de chargement n'est plus fixée lors de l'édition des liens. Le binaire exécutable représentant le programme à charger en mémoire devra pouvoir être exécutable quelle que soit l'adresse de chargement finalement trouvée par le noyau. Deux solutions sont alors possibles : soit ce binaire exécutable ne contient pas d'instructions utilisant un mode d'adressage absolu et le code obtenu est donc intrinsèquement translatable et chargeable n'importe où en mémoire, soit certaines instructions utilisent un mode d'adressage absolu et il faudra donc, lors du chargement, modifier ces champs adresses pour les rendre cohérents par rapport à l'adresse de chargement ;
- la taille de l'espace d'exécution allouée au programme doit être connue dès le chargement du programme. Il faut en effet trouver une zone mémoire de taille suffisante pour y charger le programme sous sa forme binaire. Les processus pouvant se terminer n'importe quand et donc dans n'importe quel ordre, des zones libres peuvent apparaître entre deux zones occupées. Le noyau devra gérer une cartographie de sa mémoire centrale en maintenant une liste des zones de mémoire libres. Chaque zone devra être repérée par son adresse de début et sa taille. Une stratégie de fusion des zones libres contiguës devra être mise en œuvre pour éviter l'émission de l'espace en zones de plus en plus petites. Cependant une technique de type ramasse-miettes ne peut être envisagée, celle-ci posant le problème de la réimplantation des programmes à une adresse de chargement différente durant leur exécution.

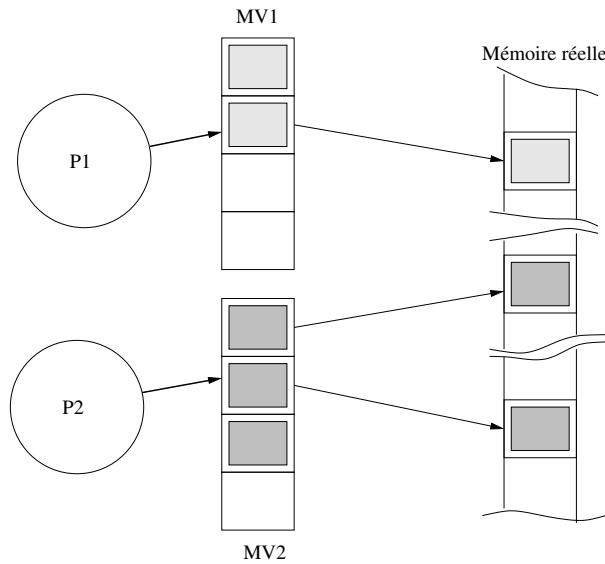


FIGURE 4.1 – Représentation de mémoires virtuelles

En résumé, le chargement de plusieurs programmes dans une même mémoire pose de nombreux problèmes. C'est pourquoi, les concepteurs de système d'exploitation ont eu l'idée d'une abstraction permettant de masquer ce partage, en l'occurrence, la notion de mémoire virtuelle.

4.1.2 Mémoire virtuelle

Le mécanisme de mémoire virtuelle a pour objectif de permettre une exécution des programmes dans un contexte virtuellement monoprogrammé. Tout processus exécutera un seul programme “chargé” dans une mémoire privée au processus. Cette mémoire dite virtuelle devra finalement être implantée physiquement grâce aux ressources mémoires centrale et secondaire (disques). Cependant, le problème du partage est masqué et c'est le noyau de gestion de mémoire virtuelle qui prendra en charge la représentation des différentes mémoires virtuelles à l'aide des ressources mémoires disponibles, tant mémoire centrale que mémoire secondaire.

4.1.3 Principe de base

L'idée fondamentale est de dissocier un espace virtuel fournissant essentiellement un espace adressable de la représentation effective de cet espace en mémoire centrale. La figure (4.1) illustre l'association d'une mémoire virtuelle à chaque processus, ces mémoires étant alors représentées par des blocs de mémoire réelle.

Le mécanisme de mémoire virtuelle simplifie le problème du chargement d'un programme en mémoire puisque tout programme binaire exécutable peut être engendré en tenant compte de son emplacement en mémoire virtuelle : en effet, dans un système à mémoire virtuelle, un seul programme est présent dans la mémoire et donc, on peut décider de prévoir une implantation standard dans cette mémoire. Les programmes peuvent faire des références en adressage absolu sans poser de difficulté au moment du chargement.

Lorsqu'un processus est actif, celui-ci engendre des références dans la mémoire virtuelle qui lui est affectée et dans laquelle se trouve le programme qu'il exécute. Ces adresses virtuelles doivent finalement permettre d'accéder un mot mémoire réel et donc être converties en adresses réelles. Pour ce faire, il faut donc avoir défini une technique de projection d'un espace virtuel d'adressage sur l'espace mémoire réel. Deux techniques de projection ont été proposées, la pagination et la segmentation. Le mariage possible des deux techniques apporte une solution finale bénéficiant des avantages des deux techniques de base.

Le mécanisme de conversion d'adresse, mis en jeu lors de toute référence mémoire, doit être très performant. C'est pourquoi des circuits dédiés appelés *UGM*, unités de gestion mémoire (*MMU*, Memory

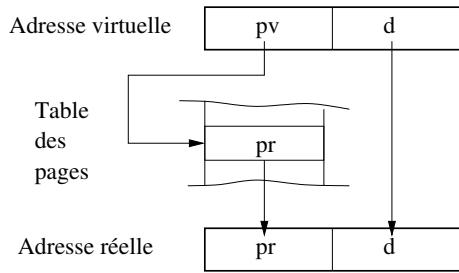


FIGURE 4.2 – Conversion d’adresse en pagination

Management Unit), ont été développés.

4.1.4 La pagination

La pagination repose sur les règles suivantes :

- L'espace d'adressage virtuel est découpé en blocs de taille fixe appelés pages ;
- L'espace d'adressage réel est aussi découpé en pages de même taille ;
- La taille d'un page est une puissance de 2 et s'élève en général à quelques K-octets (4, 8 ou 16 par exemple).

Conversion d’adresse

Toute adresse virtuelle est décomposable en deux champs : un numéro de page et un déplacement dans cette page. La conversion d'une telle adresse en adresse réelle reposera sur une **table des pages** qui mémorise pour tout numéro de page virtuelle valide, le numéro de page réelle correspondant. Le schéma (4.2) décrit ce calcul.

Avantages et inconvénients

Les avantages de la pagination tiennent essentiellement aux deux propriétés suivantes :

- les pages étant de taille fixe en mémoire réelle, l'allocation de l'espace mémoire réel est simple. Il suffit de trouver un nombre suffisant de pages libres pour la taille de mémoire virtuelle demandée (nous verrons que l'on peut faire mieux avec l'approche d'allocation dynamique) ;
- la conversion d'adresse est simple.

Par contre, le principal inconvénient est le découpage arbitraire en tranches du programme “paginé”. En effet, une page peut très bien ne contenir que quelques mots utiles occupés par la fin d'une section de code par exemple. Il y a donc perte d'espace dans de tels cas. Plus fondamentalement, la structure du programme en section(s) de code, de données, de pile,... n'est pas connue de la pagination. Or, on peut exploiter cette structuration du programme pour découper le programme en zones ayant un contenu bien spécifique. C'est ce que permet l'approche suivante, en l'occurrence, la segmentation.

4.1.5 La segmentation

La segmentation repose sur les règles suivantes :

- L'espace d'adressage virtuel est découpé en blocs de taille **fixe** appelés segments ;
- L'espace d'adressage réel n'est pas structuré et reste un espace de mots ;
- La taille d'un segment est une puissance de 2 et s'élève en général à quelques centaines de K-octets (64, 128, 256 par exemple).

On remarquera que les segments sont de taille fixe dans l'espace virtuel. C'est leur projection en mémoire réelle qui va être de taille variable pour s'adapter au contenu exact du segment.

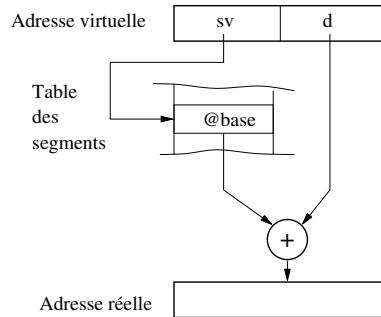


FIGURE 4.3 – Conversion d'adresse en segmentation

Conversion d'adresse

Toute adresse virtuelle est décomposable en deux champs : un numéro de segment et un déplacement dans ce segment. La conversion d'une telle adresse en adresse réelle reposera sur une **table des segments** qui mémorise pour tout numéro de segment virtuel valide, l'adresse de début et la longueur effective du segment alloué en mémoire réelle. Le schéma (4.3) décrit le calcul de l'adresse réelle.

Avantages et inconvénients

Le principal avantage de la segmentation est de permettre un découpage du programme en segments correspondant logiquement au découpage en sections du programme. Ceci optimise l'allocation mémoire puisque l'espace alloué est exactement l'espace nécessaire utilisé.

Par contre, l'allocation de la mémoire réelle est plus délicate puisqu'il faut allouer des blocs de taille variable. C'est pourquoi, on a souvent recours à une technique mixte exploitant les deux techniques à la fois.

4.1.6 La technique mixte segmentation-pagINATION

La segmentation repose sur les règles suivantes :

- L'espace d'adressage virtuel est découpé en blocs de taille **fixe** appelés segments ;
- Les segments sont découpés en pages ;
- L'espace d'adressage réel est découpé en pages ;

La mémoire réelle retrouve une structure de pages qui facilitera l'allocation. On conserve cependant la possibilité de coupler le programme en mémoire virtuelle selon un découpage obéissant à la structure logique du programme (segment(s) de code, segment pile, segment tas, segment de données statique,...). On combine donc les avantages des deux techniques au prix d'une conversion d'adresse un peu plus complexe et de tables un peu plus nombreuses (tables de pages associées à chaque segment).

Conversion d'adresse

Toute adresse virtuelle est décomposable en trois champs : un numéro de segment, un numéro de page et un déplacement dans cette page. La conversion d'une telle adresse en adresse réelle reposera sur les **table des segments** qui mémorise pour tout numéro de segment virtuel valide, l'adresse de début et la longueur effective du segment alloué en mémoire réelle. Le schéma (4.4) décrit le calcul de l'adresse réelle.

Avantages et inconvénients

Le mécanisme de segmentation-pagINATION est évidemment plus couteux que la simple pagination au point de vue conversion d'adresse et espace mémoire (tables de pages pour chaque segment), mais il permet de concilier le respect de la structure logique du programme et l'allocation simple de la mémoire réelle. C'est une technique largement utilisée notamment sur les architectures de grande puissance (mainframes).

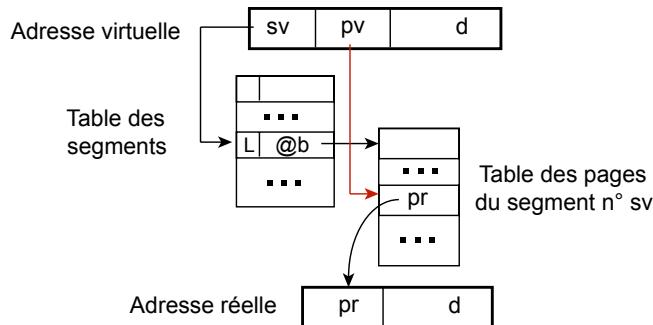


FIGURE 4.4 – Conversion d'adresse en segmentation-pagINATION

4.1.7 Le couplage des programmes

L'opération de couplage consiste à fixer l'installation du programme exécutable dans la mémoire virtuelle. Il s'agit en fait d'affecter un contenu à chaque segment ou page.

La version binaire objet exécutable d'un programme est décrite dans un format souvent spécifique au système d'exploitation utilisé. Néanmoins, cette description est toujours structurée en différentes sections contenant : une ou plusieurs sections de code, une ou plusieurs sections de données statiques, une section tas-pile.

Pour un système de segmentation, chaque section correspondra à un segment. Pour un système de pagination, chaque section sera évidemment découpée en pages.

La segmentation d'un programme offre la possibilité de partage de code. En effet, une section de code d'un programme doit être protégée contre l'écriture de façon à éviter la destruction du code en cours d'exécution. Un segment de code est donc une zone ayant un contenu constant quel que soit le processus qui l'utilise (l'exécute). Le mécanisme de segmentation permet d'éviter de charger plusieurs fois le même code si plusieurs processus exécutent le même programme.

À titre d'exemple, la figure (4.5) illustre le partage du code d'un compilateur *C* par deux processus exécutant chacun une compilation. Les segments de code présents dans chaque mémoire virtuelle sont associés au **même** segment en mémoire réelle. Par contre, et bien évidemment, les segments de données et en particulier le segment pile de chaque processus est associé à un segment réel distinct.

4.1.8 Les stratégies d'allocation de pages

Le mécanisme de mémoire virtuelle peut être utilisé pour assurer seulement une bonne protection des programmes (des processus) entre eux. Dans ce cas, l'optimisation de l'allocation de la mémoire réelle n'étant pas recherchée, le programme exécutable couplé en mémoire virtuelle est chargé effectivement en mémoire réelle avant de commencer l'exécution du processus. Autrement dit, l'espace mémoire réelle nécessaire pour représenter toutes les pages (ou tous les segments) occupé(e)s par le programme sont allouées préalablement à l'activation du processus et restent fixes durant toute l'exécution.

Cependant, une optimisation supplémentaire peut être réalisée en adoptant une stratégie de pages à la demande. Dans cette stratégie, la liaison entre une page virtuelle et une page réelle pourra être remise en cause tout au long de l'exécution du programme. L'optimisation réalisée consiste à retarder jusqu'au dernier moment, celui de la première référence à la page virtuelle (delay binding time) la liaison d'une page réelle à la page virtuelle référencée. C'est en effet à cet instant seulement qu'il devient indispensable d'avoir une représentation de la page virtuelle sous forme d'une page réelle en mémoire centrale. L'unité de gestion mémoire détectera automatiquement l'absence de liaison et provoquera une exception. Contrairement au cas habituel, cette exception ne provoquera pas l'arrêt du processus fautif, mais au contraire sera interprétée comme une "demande" de page. Cette exception est appelée "défaut de page". La routine de traitement des défauts de page est un élément essentiel dans un tel système.

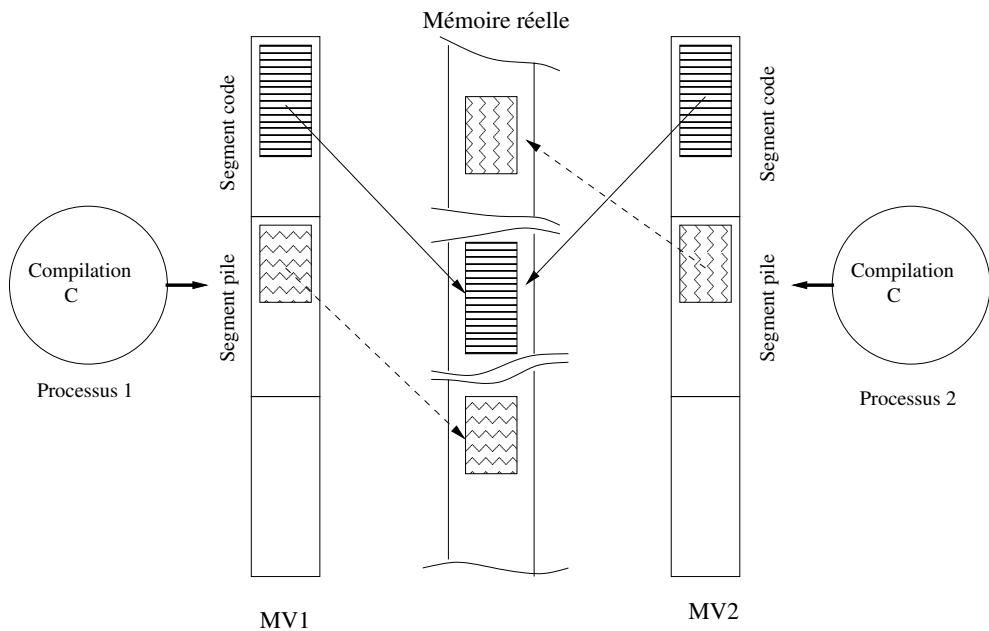


FIGURE 4.5 – Partage de code

Si une page virtuelle présente en mémoire centrale n'est pas référencée, elle est inutile. On peut donc envisager de casser cette liaison pour libérer la page réelle au profit d'une autre page virtuelle absente mais référencée. Cette approche consiste donc à faire du « remplacement de pages ».

Cependant, une page ne peut être libérée sans tenir compte de son contenu. Cette page contient l'état courant d'un programme et doit donc être sauvegardée si son contenu doit être remplacé au profit d'un autre programme. Il faut disposer d'un espace disque adéquat pour conserver une copie du contenu des pages virtuelles qui ne sont plus (ou pas) présentes en mémoire centrale. Cette espace disque, appelé zone (espace) de swap, constitue une sorte d'extension de la mémoire disponible pour représenter les mémoires virtuelles.

Deux problèmes importants doivent être maîtrisés dans une telle approche.

- Le premier tient à l'existence éventuelle de plusieurs copies d'une même page virtuelle à un instant donné. En effet, une page virtuelle peut désormais avoir un contenu dans une page de mémoire centrale et aussi une copie sur disque en zone de swap. Il faudra donc gérer correctement la cohérence de ces deux copies.
- Le second concerne le contrôle de l'efficacité du mécanisme de remplacement de page. En effet, un remplacement de page peut induire jusqu'à deux échanges mémoire-disque. De tels échanges ont un coût. Il faut donc s'assurer que ce mécanisme n'absorbe pas trop de ressources du système global.

Stratégie de remplacement de pages

Dans un système adoptant le principe de chargement de page à la demande, une routine de traitement des défauts de page met en œuvre le remplacement de pages.

Activée sur l'occurrence d'une exception « page absente », la routine de traitement du défaut de page déroule l'algorithme suivant :

```

TraiterDéfaut() {
    mv = MV.Identifier();           /* trouver la mémoire virtuelle en cause */
    if (mv.DéfautValide()) {
        p = MemRéelle.Allouer();   /* allocation d'une page réelle */
        if (p.Écrire()) p.Vider(); /* Vidage de la page choisie */
        p.Charger(mv);           /* Chargement de la page avec son nouveau contenu */
    }
}

```

La routine vérifie d'abord si le défaut de page est valide. En effet, la référence ayant provoquée le défaut de page peut être vraiment erronée si elle référence une page interdite, non couplée par exemple.

S'il s'agit bien d'un défaut de page valide, une page réelle doit être trouvée. Si la page obtenue appartient à une autre mémoire virtuelle et vait été écrite, alors, il faut vider son contenu en zone de swap. Puis, le nouveau contenu de la page peut être chargé et la table des pages associée à la mémoire virtuelle *mv* est modifiée pour rendre présente la page virtuelle qui possède désormais une représentation en mémoire centrale.

La procédure *Allouer* n'est jamais bloquante et fournit toujours une page réelle, celle-ci ayant peut-être due être préemptée à un autre processus. Cette stratégie de préemption est adoptée car un tel système appartient au modèle générique présentant un risque d'interblocage. En effet, les processus via les défauts de pages demandent des ressources critiques de type page dynamiquement. S'ils gardent les pages qu'ils possèdent lorsqu'ils se bloquent en attente de pages, les conditions nécessaires à un interblocage sont vérifiées. Par conséquent, la solution retenue est de préempter des pages aux processus lorsque cela devient nécessaire.

Une question importante est alors de choisir la page à préempter. Ce choix doit minimiser le risque de défaut de page futur. Pour ce faire, la meilleure page serait donc la page qui sera référencée dans le futur le plus lointain. Or, on ne connaît pas l'avenir...

La seule possibilité est donc d'extrapoler le futur en regardant le passé. Plusieurs propositions ont été longuement étudiées. Nous retiendrons :

- Compter sur le hasard : Cette solution n'est pas la meilleure mais elle n'est pas catastrophique ;
- Choisir la page réelle qui contient la page virtuelle présente qui est chargée depuis le plus longtemps. C'est une stratégie *FIFO* vis-à-vis de l'instant de chargement en mémoire réelle des pages virtuelles. Cette solution n'est pas forcément adéquate car les pages les plus anciennes peuvent correspondre aux pages les plus référencées ;
- Choisir la page non référencée depuis le plus longtemps par rapport à l'instant présent. C'est la page la plus "oubliée". Ce choix est souvent juste mais le problème est cependant de savoir trouver cette page. Cette stratégie, appelée *LRU* pour "Least-Recently-Used", est en effet difficile à mettre en œuvre car il faudrait dater toutes les références. Cependant une bonne approximation peut être implantée à un coût raisonnable. Nous en donnons le principe dans ce qui suit.

Stratégie *NUR* (*Not-Recently-Used*)

L'idée est de considérer seulement un passé proche du système. Plus exactement, il s'agit d'observer les événements "référence à" et "écriture de" pour chaque page réelle. Pour cela, deux bit indicateurs sont gérés par l'*UGM* dans chaque descripteur de table de page. Pour éviter que toutes les pages soient finalement dans l'état "référencée" et y restent, dès que cet état est atteint, l'*UGM* remet ces indicateurs de référence à zéro¹. Ainsi, les indicateurs de référence donne une vision des références "récentes", d'où le nom donné à cette stratégie.

Les pages réelles se répartissent en 4 catégories. Le choix d'une page se fera dans le premier ensemble non vide trouvé :

1. l'ensemble des pages non référencées et non écrites ;
2. l'ensemble des pages non référencées et écrites ;

¹ Bien sûr, les indicateurs d'écriture ne peuvent être remis à zéro que lorsque la page aura été sauvegardée en zone de swap.

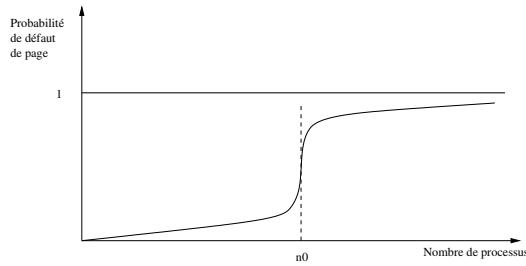


FIGURE 4.6 – Phénomène d’écroulement

3. l'ensemble des pages référencées et non écrites ;
4. l'ensemble des pages référencées et écrites.

On remarquera que les deux premiers états sont possibles compte tenu de la gestion de l'indicateur de référence.

Enfin, la priorité est donnée au fait que la page soit référencée ou non. En effet, une page non référencée mais écrite sera choisie en priorité sur une page référencée mais non écrite.

Contrôle de l’écroulement (thrashing)

Quelle que soit la technique de remplacement de page envisagée, un problème d’écroulement du système se pose. En effet, plus le nombre de mémoires virtuelles augmente , plus le nombre de pages présentes par mémoire va diminuer. Si le nombre de pages présentes en mémoire centrale pour chaque mémoire virtuelle devient trop petit, beaucoup de défauts de page vont survenir et par conséquent, beaucoup de remplacements de page.

On comprends qu’au delà d’un certain seuil, les ressources du système global vont être presque exclusivement consacrées à gérer le remplacement de pages. Les performances du système global vont alors s’effondrer. Ce phénomène d’écroulement (thrashing) doit donc être contrôlé et évité si possible. Les concepteurs de systèmes de mémoire virtuelle avec pages à la demande ont donc été confronté à ce problème. La figure (4.6) illustre le phénomène.

Ce problème n'est cependant pas simple. En effet, le taux de défauts de page engendré par l'exécution d'un processus est lié aux schémas de références mémoires qu'il provoque. À titre d'exemple, un processus exécutant une boucle sur quelques instructions référençant peu de données peut très bien ne pas provoquer de défaut de page durant toute l'exécution de cette boucle à condition de lui accorder deux pages présentes en mémoire : celle contenant les instructions de la boucle et celle contenant les données. Par contre, si le processus exécute un boucle mettant en jeu une matrice de grande taille provoquant des références sur de nombreuses pages virtuelles différentes, le nombre de défauts de page va être important si les pages contenant la matrice ne sont pas maintenues présentes.

Notion d'espace d'exécution (working set)

On constate donc qu'il faudrait connaître par avance les besoins en espace mémoire du processus pour garantir un taux de défauts de page raisonnable. À tout instant, il existe ainsi un ensemble de pages présentes qui caractérise et garantit une exécution du processus avec un taux raisonnable de défaut de pages. Cet ensemble est appelé espace d'exécution (working set). La taille de cet espace fixe en quelque sorte l'espace "vital" nécessaire au processus pour s'exécuter correctement. Il faut pouvoir cependant évaluer cette taille qui évolue dynamiquement. Pour ce faire, on ne peut une nouvelle fois qu'extrapoler un comportement futur à partir d'un comportement passé. La stratégie adoptée consiste à évaluer périodiquement le nombre de défauts de page qui ont été provoqués par un processus. Si ce taux augmente, le nombre de pages présentes pour ce processus, autrement dit la taille de son espace d'exécution doit être augmenté. Si ce taux diminue, le nombre de pages présentes pourra être réduit.

Pour éviter le phénomène d'écroulement, il suffit d'assurer l'invariant suivant :

$$\forall t \sum_{p=1}^{p=n} WS(p, t) \leq M$$

où $WS(p, t)$ représente la taille de l'espace d'exécution du processus p à l'instant t et M la taille en pages de la mémoire réelle disponible initialement.

En réalité, il n'est évidemment pas possible de connaître la taille de l'espace d'exécution à chaque instant et c'est une approximation qui est périodiquement calculée.

En particulier, cet invariant servira à contrôler l'introduction d'un nouveau processus impliquant la création d'une nouvelle mémoire virtuelle.

4.1.9 Conclusion

Même si l'espace mémoire centrale est aujourd'hui une ressource moins critique qu'il y a quelques années, les mémoires virtuelles constituent à la fois une abstraction et un mécanisme fondamental dans la gestion de la mémoire. En particulier, leur rôle de protection des traitements des différents usagers et du noyau est indispensable pour obtenir un système fiable et sûr.

Enfin, on trouvera dans [6] et [8], une description détaillée des notions précédentes. Pour les amateurs, un livre complet est dédié à la description de la gestion des mémoires virtuelles dans le système Linux sous le titre : *Understanding the Linux® Virtual Memory Manager*[3].

Chapitre 5

Conclusion

Les principes de conception des systèmes d'exploitation sont maintenant bien établis. Les concepts de processus pour l'exécution des programmes et de fichiers pour la gestion des données rémanentes sont les briques de base stables des noyaux de système.

Cependant, le développement d'un fort parallélisme dans les architectures des processeurs conduit aujourd'hui à introduire la notion de processus léger ou activité ou *thread*. Un processus apparaît alors plutôt comme une unité d'allocation de ressources (processeur, mémoire, programme, fichiers, etc), une sorte de machine virtuelle qui permet l'exécution de plusieurs activités séquentielles au sein d'un même processus. Autrement dit, un processus peut comporter plusieurs points de contrôle actifs, plusieurs activités en parallèle, celles-ci partageant le même programme et le même espace mémoire. L'avantage est bien entendu ce partage de ressources qui permet notamment une optimisation des communications entre activités par l'usage de leur mémoire commune. C'est aussi un inconvénient dans la mesure où les activités sont moins bien protégées les unes des autres que dans le cadre de processus distincts.

Les noyaux doivent donc aujourd'hui gérer deux niveaux d'allocation de ressources : l'allocation des ressources aux processus dits «lourds» et l'allocation des ressources aux activités appelées aussi processus «légers». En particulier, l'ordonnancement des activités doit être bien maîtrisé de façon à répartir équitablement le temps processeur offert par les processeurs des architectures multi-cœurs entre les différentes activités selon qu'elles appartiennent au même processus ou à des processus différents.

La synchronisation des processus lourds ou légers a pour objectif de régler les conflits d'accès aux ressources critiques qu'ils sont amenés à partager bien malgré eux ou de coordonner la coopération ou communication entre eux selon des protocoles bien définis assurant une cohérence des traitements. La réalisation de cette synchronisation s'appuie sur un ensemble de mécanismes : verrous, sémaphores, moniteurs, rendez-vous, transactions, etc. Une étude des différents problèmes génériques de synchronisation constitue une tâche difficile et complexe d'un point de vue algorithmique (parallèle). Les ordinateurs ayant de plus en plus de processeurs, de nouvelles études sont nécessaires pour améliorer les techniques de programmation parallèle fondées sur ces mécanismes, ceci incluant la découverte de nouveaux mécanismes mieux adaptés.

Les systèmes de fichiers connaissent aussi des évolutions en ce qui concerne leur implantation compte tenu de la croissance des capacités offertes par les unités de mémoire de masse (de la clé USB au disque de plusieurs Tera-octets). Le nombre de fichiers présents sur un disque est en effet de plus en plus énorme et nécessite des algorithmes très spécifiques pour optimiser la localisation des descripteurs de fichiers.

Par ailleurs, les fichiers deviennent eux-même de plus en plus gros avec notamment les fichiers contenant des images et surtout de la vidéo. Là encore, une implantation spécifique de ces fichiers sur l'espace disque disponible peut optimiser l'accès aux données et améliorer les performances du «système».

Le parallélisme interne des architectures et la connexion de centaines, voire de milliers d'ordinateurs via un réseau à haut débit fait apparaître aussi un autre besoin : des systèmes d'exploitation capables de répartir entre usagers non pas les ressources d'une architecture centralisée, mais celles d'une architecture répartie composée de ces milliers de machines constituant une grille de calcul ou «cluster». Des systèmes d'exploitation de grilles de calcul sont en cours de développement. Citons par exemple *XtreemOS* développé

à partir de Linux[2].

Enfin, une tendance actuelle est le développement de systèmes informatiques embarqués. De nombreux objets contiennent maintenant des architectures informatiques complexes ayant besoin d'un noyau de système d'exploitation pour fournir le support d'exécution aux applications qui pourront être disponibles sur ces objets : on pense naturellement aux téléphones (I-phone, Blackberry's, etc) ou aux diverses tablettes (I-pad par exemple), baladeur (i-Pod) et consoles de jeux. Ce phénomène ne se limite pas aux objets de type grand public mais prend aussi de l'ampleur dans tous les domaines : robotique, instrumentation, etc.

Conséquence de cette évolution : le nombre de noyaux de système d'exploitation ne cesse d'augmenter. Des propositions de standardisation viendront sûrement, mais on en est encore à la période « farwest ». Citons *Android* de Google, *iOS 4* d'Apple pour ses téléphones et tablettes, *Windows 7 Mobile* de Microsoft. Tous ces noyaux de système intègrent la gestion du parallélisme et un ordonnancement dit Temps réel des applications. Ils sont surtout très bien adaptés pour gérer des interfaces usagers sophistiquées : images 2D ou 3D, icônes, fenêtrage, menus, etc.

En conclusion, il y a donc un bel avenir pour les systèmes d'exploitation... .

Appendices

Annexe A

Exercices

Les questions suivantes ont été posées en examen et rassemblées au fil du temps. Elles peuvent permettre de faire le point sur la compréhension des principes, concepts et mécanismes présentés.

A.1 Des questions d'ordre général

1. Quelle fonction de base est assurée par un interpréteur de commandes ? Comment le jeu de commandes est-il extensible par un usager quelconque ?
2. Qu'appelle-t-on un script ?
3. Le dialogue d'un usager avec le système via un interpréteur de commandes est-il implanté par un processus ? Justifier votre réponse.
4. D'un point de vue sécurité, quel risque existe-t-il pour un usager *A* d'exécuter un programme développé par un autre usager *B* ayant des droits distincts de *A* ?
5. Pourquoi, les primitives du noyau d'un système d'exploitation ne sont pas réentranttes ?
6. Quelle est la différence entre une primitive et une commande ?
7. Quelle est la différence sémantique entre une interruption et un déroutement ?
8. A quoi servent les déroutements programmés (instruction TRAP) ?
9. Dans un système d'exploitation, quels sont les deux concepts utilisés pour la gestion d'une part, des traitements (exécution des programmes) et d'autre part, des données ?
10. À quoi sert la phase de connexion (login) des usagers ?
11. Tout système d'exploitation possède un langage de commande. Par exemple, le système Unix propose le langage `shell` (et ses nombreuses variantes). Un tel langage est-il interprété ou compilé ? Justifiez votre réponse.
12. Un système d'exploitation doit résister aux erreurs des programmes usagers qu'il exécute. Par exemple, un programme contient une instruction qui tente d'écrire dans le mot mémoire d'adresse 0. Que se passe-t-il ? Quel mécanisme matériel est mis en œuvre ?
13. Un système d'exploitation doit résister aux erreurs des programmes usagers qu'il exécute. Citez deux exemples d'erreur possible et le(s) mécanisme(s) matériel(s) qui permettent de garantir une récupération de telles erreurs et leur contrôle par le noyau du système.
14. Dans un système d'exploitation, qu'appelle-t-on superviseur ?
15. Un programme binaire exécutable peut-il être exécuté sans modification par des processeurs matériels différents, par exemple un processeur IBM Power PC et un Intel Pentium ? Justifiez votre réponse.
16. Un programme binaire exécutable peut-il être exécuté sans modification sous des systèmes d'exploitation différents, par exemple Windows et Linux ? Justifiez votre réponse.

17. Un système d'exploitation doit résister aux erreurs des programmes usagers qu'il exécute. Précisez pour les erreurs suivantes s'il s'agit d'un déroutement ou d'une interruption et indiquez, dans chaque cas, la réaction du noyau :
 - exécution d'une instruction TRAP ;
 - fin d'un échange disque via un contrôleur SCSI ;
 - exécution d'une instruction privilégiée en mode programme ;
 - fin de quantum.
18. Quelles sont les ressources qui doivent être virtualisées pour que plusieurs instances de systèmes d'exploitation puissent s'exécuter de façon sûre ?
19. Quel risque potentiel existe-t-il lorsqu'on exécute un programme que l'on n'a pas développé ?
20. Pourquoi les langages de scripts sont-ils qualifiés de langage de programmation à gros grain (« in the large »). Justifier votre réponse par un exemple en shell.
21. Pourquoi, les primitives du noyau d'un système d'exploitation ne sont pas réentranttes ?
22. Expliquer pourquoi il peut être intéressant de lancer deux instances d'un même système d'exploitation en utilisant la virtualisation.

A.2 À propos des processus

1. Dans un noyau de gestion des processus exécutables (ordonnanceur), les processus prêts possèdent une priorité fixe attribuée lors de leur création. Le processus prêt qui devient actif lors de la libération du (d'un) processeur est donc toujours le plus prioritaire. Préciser sous quelles conditions, un processus peu prioritaire peut ne jamais devenir actif. Proposer un algorithme d'ordonnancement pour éviter cette anomalie tout en gardant la notion de priorité.
2. Un processus est interrompu par le superviseur au bout d'un quantum de temps fixé même lorsqu'il peut continuer à s'exécuter. Proposer une stratégie adaptative de la durée d'un quantum de façon à minimiser le nombre d'interruptions sur fin de quantum. Attention, cette stratégie ne doit pas amener à une durée de quantum trop grande : une valeur maximale devra être fixée. Pourquoi ?
3. Comment peut-on comptabiliser le temps processeur utilisé par un processus particulier ? Expliquer les mécanismes mis en jeu.
4. Des processus de deux usagers distincts ne partagent pas a priori un espace mémoire usager commun. Expliquez pourquoi ? Le peuvent-ils s'ils « roulent pour le » (s'exécute pour le compte du) même usager ? Justifier votre réponse.
5. Combien de processus au maximum peuvent être actifs sur un biprocesseur ?
6. Les processus exécutables sont soit actifs, soit prêts. Quelle différence essentielle existe-t-il entre ces deux états : actif et prêt ?
7. Donner 3 exemples de situations conduisant un processus à l'état bloqué.
8. Pourquoi un processus est interrompu par le superviseur au bout d'un quantum de temps fixé même lorsqu'il pourrait continuer à s'exécuter ?
9. Quelle est l'utilité de connaître pour quel usager un processus s'exécute ? Donner au moins deux raisons.
10. Quelle propriété (de sûreté) doit assurer un bon algorithme d'ordonnancement à long terme des processus ?
11. Un processus consomme des ressources pour s'exécuter. Citez 2 types de ressources qui lui sont indispensables.
12. Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ? Justifiez votre réponse en expliquant le mécanisme de commutation de processus.

13. Sur un quadriprocesseur, combien de processus au maximum peuvent être simultanément dans l'état actif ?
14. Sur une architecture monoprocesseur, deux compilations sont lancées en parallèle. Expliquez pourquoi l'exécution « parallèle » de ces deux compilations durera moins longtemps que leur exécution séquentielle bien qu'il n'existe qu'un seul processeur central.
15. Sur une architecture monoprocesseur, par quel mécanisme assure-t-on qu'un processus ne monopolise pas trop longtemps le processeur ?
16. Un même processus peut-il exécuter plusieurs programmes (binaires exécutables) simultanément ?
17. Un ensemble I de processus exécute des programmes qui engendrent beaucoup de lectures et d'écritures dans des fichiers (compilations par exemple) et un autre ensemble C de processus exécute des programmes qui engendrent des calculs longs. L'ordonnanceur a-t-il intérêt à donner une plus forte priorité aux processus de l'ensemble I ou de l'ensemble C ? Justifiez votre réponse.
18. Comment un processus peut-il communiquer avec d'autres ? Expliquez en particulier le mécanisme de connexion dynamique de ressources.
19. Expliquer pourquoi il est intéressant d'exécuter plusieurs processus même lorsqu'il n'existe qu'un seul processeur réel.
20. Le système Windows offre une primitive `CreateProcess` pour créer un nouveau processus. En conséquence, un processus peut-il changer de programme durant son exécution ? En est-il de même dans le système Unix ? Justifiez votre réponse.
21. Dans un système multi-processeur, un processus peut-il s'exécuter :
 - sur différents processeurs simultanément ?
 - sur différents processeurs alternativement (par exemple, pour deux processeurs : $P_1, P_2, P_1, P_2, \dots$) ?
 - sur différents processeurs aléatoirement (le scheduler choisit dynamiquement) ?
 - ou sur un seul durant toute son exécution ?
22. Un processus applicatif est lancé par mégarde en mode superviseur. Quel risque en découle pour le système d'exploitation ?
23. Donner et expliquer une stratégie d'allocation du processeur aux processus prêts assurant la répartition équitable du temps processeur entre les processus exécutables.
24. Avec le système Windows, un processus peut-il changer de programme ? Justifier votre réponse.
25. Pour quelle raison un processus actif risque de ne pas consommer la totalité de son quantum de temps ?
26. Dans un ordonnanceur, le concepteur a choisi de provoquer l'allongement de la durée du prochain quantum de temps attribué à un processus à chaque fois que le processus a consommé complètement son quantum courant. Cette stratégie vous semble-t-elle acceptable ?
27. Citer des ressources logiques que peuvent utiliser plusieurs processus pour partager des données ?

A.3 À propos des fichiers

1. Qu'appelle-t-on méthode d'accès dans les systèmes de gestion de fichiers ? Illustrer par des exemples.
2. Un système de fichiers peut-il gérer plusieurs volumes ? Justifiez votre réponse.
3. Dans un système de fichiers, le nombre de fichiers est-il illimité ? Justifiez votre réponse.
4. Expliquez ce qu'est une méthode d'accès à un fichier. Donnez deux exemples différents de méthode d'accès.
5. Par quoi est représenté un répertoire dans un système classique de gestion de fichiers ?
6. Donnez deux exemples d'incohérence possible sur un volume disque entre la description des blocs libres et des blocs de fichiers.
7. Quels mécanismes permettent d'assurer qu'un programme exécuté par un processus usager ne peut pas écrire ou lire un secteur disque ?

8. Donner une différence fondamentale entre le mode de désignation des fichiers dans le système Windows et dans le système Unix.
9. Quel contrôle élémentaire de cohérence doit-on faire pour vérifier qu'un volume et les fichiers qu'il contient sont dans un état cohérent ?
10. Un système d'exploitation doit garantir l'intégrité du ou des systèmes de fichiers qu'il gère. En particulier, il faut empêcher qu'un programme usager quelconque puisse écrire des secteurs du (pseudo-)volume support d'un système de fichiers. Donner une implantation possible : préciser les mécanismes matériels mis en jeu, la politique de sécurité mise en œuvre.
11. Deux processus accèdent au même fichier : le premier écrit dans le fichier sans problème. Le second tente de lire le fichier, mais une exception se produit et le processus est avorté. Expliquez pourquoi.
12. Quelle propriété de la méthode d'accès séquentielle exploite-t-on pour optimiser le parallélisme entre les échanges disque-mémoire et l'exécution des processus ?
13. Pourquoi le parcours d'un chemin d'accès à un fichier, par exemple `/users/1AI/gr1/dupont/prog.c` nécessite-t-il l'ouverture et la lecture de plusieurs fichiers ? Précisez lesquels.
14. Lorsqu'un processus s'exécute, sur quel attribut associé au processus se base-t-on pour vérifier les droits d'accès de ce processus aux ressources fichiers qu'il cherche à accéder ?
15. Dans le système Windows, expliquer comment les blocs libres d'un volume disque peuvent être connus.
16. Sous Unix, il n'existe pas de primitive permettant de détruire un fichier. Expliquer pourquoi et comment le noyau assure néanmoins la destruction des fichiers.
17. La notion de liste d'accès utilisée comme mécanisme de protection des fichiers est-elle une approche par ligne ou par colonne en se basant sur la modélisation classique matricielle $Usager \times Fichier \rightarrow Droits$?
18. Comment garantit-on qu'un processus usager ne peut pas écrire des données dans un secteur n'importe où et directement sur un volume disque ?
19. On ne peut lire moins d'un secteur de quelques octets sur un volume disque. Cela est-il un inconvénient lorsqu'un programme lit séquentiellement dans un fichier par paquet de quelques octets ?
20. Pourquoi les langages de scripts sont-ils qualifiés de langage de programmation à gros grain (« in the large »). Justifier votre réponse par un exemple en shell
21. Sous Unix, il n'existe pas de primitive permettant de détruire un fichier. Expliquer pourquoi et comment le noyau assure néanmoins la destruction des fichiers.
22. La notion de liste d'accès utilisée comme mécanisme de protection des fichiers est-elle une approche par ligne ou par colonne en se basant sur la modélisation classique matricielle $Usager \times Fichier \rightarrow Droits$?

Annexe B

Encore mieux, exercices corrigés...

B.1 Généralités

1. Dans un système d'exploitation, quels sont les deux concepts utilisés pour la gestion d'une part, des traitements (exécution des programmes) et d'autre part, des données ?

Réponse La notion de processus pour les traitements et la notion de fichier pour les données.

2. Un système d'exploitation doit résister aux erreurs des programmes usagers qu'il exécute. Citez deux exemples d'erreur possible et le(s) mécanisme(s) matériel(s) qui permette(nt) de garantir une récupération de telles erreurs et leur contrôle par le noyau du système.

Réponse Deux exemples d'erreurs :

- un programme boucle : il faut alors pouvoir l'interrompre. Le mécanisme matériel est l'interruption.
 - une programme adresse un mot mémoire inexistant : le processeur provoque un déroutement vers une routine de traitement de cette exception.
3. Sur une architecture monoprocesseur, deux compilations sont lancées en parallèle. Expliquez pourquoi l'exécution « parallèle » de ces deux compilations durera moins longtemps que leur exécution séquentielle bien qu'il n'existe qu'un seul processeur central.

Réponse Durant une compilation de nombreuses lectures et écritures de fichiers ont lieu. Ces échanges bloquent logiquement les processus de compilation. Pendant qu'un processus est bloqué en attente d'une fin d'entrée/sortie, l'autre processus peut s'exécuter (jusqu'à ce qu'il se bloque lui-même éventuellement). L'exécution « parallèle » des 2 processus sera donc plus courte.

4. Un système d'exploitation doit garantir l'intégrité du ou des systèmes de fichiers qu'il gère. En particulier, il faut empêcher qu'un programme usager quelconque puisse écrire des secteurs du (pseudo-)volume support d'un système de fichiers. Donner une implantation possible : préciser les mécanismes matériels mis en jeu, la politique de sécurité mise en œuvre.

Réponse Pour qu'un programme usager ne puisse pas accéder directement aux secteurs d'un disque, on utilise la notion de mode d'exécution et d'instruction privilégiée. Un programme usager s'exécute en mode usager et, dans ce mode, certaines instructions lui sont interdites. Ces instructions, dites privilégiées, comprennent donc les instructions permettant de programmer le contrôleur du disque. En conséquence, seules les primitives du noyau, qui sont exécutées en mode privilégié, peuvent lancer des échanges au niveau physique.

5. À quoi sert la phase de connexion (login) des usagers ?

Réponse Cette phase permet d'identifier l'usager de façon à déterminer les ressources qu'il pourra accéder, en particulier en ce qui concerne l'espace de fichiers.

6. Comment un processus peut-il communiquer avec d'autres ? Expliquez en particulier le mécanisme de connexion dynamique de ressources.

Réponse Un processus communique avec son environnement par des flots de données en entrée ou en sortie via des « portes d'échange ». Le noyau offre à tout processus un ensemble de flots « ouverts » par défaut sur des ressources permettant l'échange d'informations : fichiers, pipes, etc. La connexion à ces ressources est dynamique (primitive `open`, `pipe` et `close`).

B.2 Processus

1. Combien de processus au maximum peuvent être actifs sur un biprocesseur ?

Réponse : Deux puisqu'un processus, dans l'état actif, s'exécute sur un processeur.

2. Les processus exécutables sont soit actifs, soit prêts. Quelle différence essentielle existe-t-il entre ces deux états : actif et prêt ?

Réponse : Les processus exécutables sont dans l'état actif ou prêt. Dans l'état prêt, un processus attend qu'un processeur lui soit alloué pour s'exécuter. La transition vers l'état actif est conditionnée par cette allocation. En résumé, dans l'état prêt, le processus attend la ressource processeur, dans l'état actif, il a obtenu cette ressource et il s'exécute.

3. Donner 3 exemples de situations conduisant un processus à l'état bloqué.

Réponse : Un processus passe de l'état actif à l'état bloqué lorsqu'un condition logique fausse empêche la poursuite de son exécution, par exemple :

- Le processus a exécuté une demande de lecture bloquante du clavier : il faut qu'une ligne de texte soit effectivement tapée pour que l'opération puisse être exécutée ;
- Le processus attend la terminaison d'un processus fils (Exemple Unix, via une primitive `wait`).
- Le processus écrit en mode bloquant sur un canal connecté à un pipe plein.

4. Pourquoi un processus est interrompu par le superviseur au bout d'un quantum de temps fixé même lorsqu'il pourrait continuer à s'exécuter ?

Réponse : Un processus est interrompu par le superviseur au bout d'un quantum de temps fixé pour assurer un partage équitable du temps processeur disponible entre les différents processus. S'il existe des processus interactifs prêts, ceux-ci doivent pouvoir être actifs dans un délai raisonnable (problème de temps de réponse). Sur un mono-processeur, un seul processus exécutant des calculs pourrait par exemple, s'il n'était pas interrompu, s'exécuter durant plusieurs secondes, voire plusieurs minutes, bloquant ainsi tous les autres processus prêts.

5. Quelle est l'utilité de connaître pour quel usager un processus s'exécute ? Donner au moins deux raisons.

Réponse : Lorsqu'un processus s'exécute, il utilise les ressources du système matériel et logiciel hôte. Une première utilité est donc de pouvoir imputer, comptabiliser les ressources utilisées par un processus pour le compte d'un usager précis. Mais, la connaissance de l'usager permet surtout de limiter et contrôler les ressources que celui-ci pourra utiliser. L'une des ressources les plus importantes est le système de fichiers. Les droits d'accès aux fichiers dépendent de l'utilisateur qui tente de les utiliser. Cette notion d'usager permet donc avant tout de mettre en œuvre une politique de protection des fichiers appartenant à des usagers distincts.

On considère une architecture biprocesseur dotée d'un bus mémoire et d'un bus d'entrées-sorties. On suppose que l'on dispose sur cette architecture d'un noyau d'exécution pour gérer des processus.

6. Précisez combien de processus au maximum peuvent être dans l'état actif à un instant donné sur une telle architecture ;

Réponse Deux processus au plus peuvent être actifs puisqu'il n'y a que deux processeurs.

7. Un processus dans l'état bloqué possède-t-il la ressource processeur ?

Réponse Dans l'état bloqué, un processus attend qu'une condition logique (ressource disponible, événement arrivé,...) soit satisfaite avant de pouvoir continuer son exécution. Par conséquent, il ne peut pas, dans cet état, exécuter des instructions et il ne possède donc pas la ressource processeur qui lui serait parfaitement inutile.

8. Proposez une stratégie simple d'allocation de la ressource processeur de façon à assurer que tout processus prêt finira par obtenir un processeur ;

Réponse La ressource processeur doit être allouée aux processus prêts. Pour éviter tout risque de famine, une solution simple consiste à placer les processus prêts dans une file unique FIFO. Ainsi, le processus prêt depuis le plus longtemps sera toujours sélectionné dès qu'un processeur sera libre. On assure ainsi une équité forte entre les processus prêts.

9. Par quelle stratégie peut-on éviter qu'un processus monopolise la ressource processeur pendant une durée trop longue ? Quel mécanisme matériel entrera en jeu pour mettre en œuvre une telle stratégie ?

Réponse La stratégie la plus simple consiste à fixer une valeur maximale de durée d'exécution pour un processus actif. Le processus actif sera interrompu s'il dépasse cette durée maximale. Autrement dit, lorsqu'un processus devient actif, c'est en réalité un quantum de temps processeur qui lui a été alloué (et non pas la ressource processeur pour une durée indéterminée). Le mécanisme matériel est la notion d'interruption.

10. Précisez grâce à quel mécanisme un processus peut appeler les primitives du noyau mais ne peut pas se brancher n'importe où dans le code du noyau ?

Réponse L'appel d'une primitive du noyau d'un système ne peut se faire que par un mécanisme spécifique et non pas par une simple instruction de branchement. En effet, la zone mémoire contenant le programme noyau est protégée contre de tels branchements directs. Le mécanisme le plus utilisé est le déroutement programmé. Une instruction spécifique de format suivant : TRAP *n*. Elle provoque un déroutement lorsqu'elle est exécutée. L'adresse du point d'entrée de la routine de traitement du déroutement est alors lue dans une table d'implantation préédéfinie, l'opérande spécifiée *n* sélectionnant le numéro de l'entrée dans la table. Ainsi, les seuls points d'entrée dans des primitives du noyau sont ceux placés dans cette table. Cette table étant protégé contre tout accès par un programme usager, on garantit bien la protection du noyau contre les branchements intempestifs.

B.3 Les fichiers

1. Qu'appelle-t-on méthode d'accès ? Donnez deux exemples de méthodes avec leurs principales propriétés.

Réponse On appelle méthode d'accès, l'ensemble des procédures définies pour accéder à un fichier. On distingue par exemple :

- la méthode d'accès séquentielle, qui autorise à lire ou écrire un fichier par blocs de taille variable mais toujours à partir du dernier octet lu ou écrit ;
- la méthode d'accès direct, qui autorise à lire ou écrire dans un fichier en fixant arbitrairement le point de début de lecture ou d'écriture.

2. Que contient un fichier répertoire ?

Réponse Un fichier répertoire assure la correspondance entre le nom symbolique du fichier et sa désignation interne par le noyau (en l'occurrence un numéro). On peut donc considérer un répertoire comme une liste de couples (nom symbolique, numéro interne).

3. Lorsqu'un fichier existant est lu séquentiellement, comment le noyau peut-il optimiser les lectures ?

Réponse Lorsqu'un fichier est lu séquentiellement, le noyau peut optimiser les lectures de deux façons :

- d'une part, en lisant le fichier par blocs de taille fixe dans des tampons mémoire du noyau.. Ainsi, même si le programme usager lit le fichier octet par octet, une lecture en mémoire secondaire n'aura lieu que lorsque tous les octets lus dans un tampon auront été consommés ;
- d'autre part, le noyau peut anticiper les lectures. Il peut déclencher le chargement dans des tampons de plusieurs blocs consécutifs puisque le programme usager lit le fichier séquentiellement. Cette anticipation permet d'optimiser le parallélisme entre l'activité du programme usager et les échanges avec la mémoire secondaire.

4. Une opération de fermeture d'un fichier (primitive `close`) peut-elle entraîner la destruction de ce fichier ?

Réponse La fermeture d'un fichier rompt la liaison entre un processus et un fichier. Le fichier n'est pas détruit tant qu'il est accessible par un chemin d'accès ou ce qui revient au même tant qu'il porte un nom présent dans un répertoire. La destruction n'aura donc lieu que si le fichier n'a plus de nom (ce qui est un événement indépendant de la rupture de liaison).

5. Un processus ouvre un fichier en écriture et écrit des chaînes de 10 octets séquentiellement dans ce fichier. Le noyau utilise un cache mémoire de 1K octets en écriture. Le système s'arrête pour cause de coupure de l'alimentation électrique. Expliquez pourquoi certaines écritures risquent d'être perdues.

Réponse Les écritures présentes dans le cache mémoire mais non encore recopiées en mémoire secondaire seront perdues puisque la mémoire centrale est volatile.

6. Expliquez pourquoi l'exécution d'une primitive `read` n'entraîne pas forcément une lecture physique de secteur(s) disque.

Réponse Parce qu'un fichier est lu par bloc et qu'une primitive `read` peut très bien ne demander que la lecture de quelques octets. Le contenu d'un bloc complet est donc présent dans un cache mémoire et prêt à être lu. Par ailleurs, en cas de méthode d'accès séquentiel, le noyau lit par avance dans un espace tampon les premiers blocs du fichier lors de son ouverture. Lorsque la primitive `read` sera appelée, les données à lire seront donc déjà en mémoire centrale (mécanisme de cache).

7. Plusieurs processus peuvent-ils exécuter le même programme en parallèle avec des données d'entrée distinctes ? Justifiez votre réponse.

Réponse Oui, car chaque processus s'exécute dans un espace mémoire qui lui est propre.

8. Quelle différence existe-t-il entre l'exécution en mode bloquant et en mode non bloquant de la primitive `read` ?

Réponse En mode bloquant, la primitive `read` ne se termine que lorsque l'opération de lecture a pu être exécutée (bien ou mal). En mode non bloquant, l'opération est simplement tentée immédiatement. Si aucune donnée ne peut être lue, l'opération se termine quand même.

Bibliographie

- [1] Maurice J. Bach. *The design of the UNIX operating system*. Prentice Hall International, 1986.
- [2] Coppola, M. and Jégou, Y. and Matthews, B. and Morin, C. and Prieto, L.P. and Sanchez, O.D. and Yand, E.Y. and Yu, H. Virtual Organization Support within a Grid-wide Operating System. *IEEE Internet Computing*, 12(2) :20–28, March 2008.
- [3] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open source series. Pearson Education, Inc., 2004.
- [4] S. E. Madnick. Design strategies for file systems. Technical Report TR-78, Massachusetts Institute of Technology, Cambridge, MA, USA, October 1970.
- [5] Gary Nutt. *Operating system projects using Windows NT*. Addison Wesley Longman, Inc., 1999.
- [6] J-L. Peterson and A. Silberschatz. *Operating System Concepts*, pages 131–187. Addison-Wesley Publishing Company, 1983.
- [7] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, Seventh Edition*. John Wiley and Sons, Inc., 2004.
- [8] A. Tanenbaum. *Systèmes d'exploitation, 2°Edition*, pages 201–279. Pearson Education France, 2003.
- [9] Andrew S. Tanenbaum. *Systèmes d'exploitation, 3-ième Edition*. PEARSON Education, 2003.

Index

Fichiers

- définition, 33
- désignation, 36
- destruction, 45
- exemple Unix, 39
- implantation, 41
- méthodes d'accès, 34
- protection, 36
- répertoire, 41

Interactions

- événementielles, 26
- par flots de données, 27
- redirections, 27

Interpréteur de commandes, 15, 18

Langage de script, 15

Liaisons dynamiques, 16

- flots de données, 27
- pages à la demande, 51

Mécanismes de base, 9

Mémoires virtuelles

- allocation de pages, 51
- couplage, 51
- définition, 48
- écroulement, 54
- pagination, 49
- principes, 48
- remplacement de pages, 52
- segmentation, 49
- segmentation-pagination, 50

Parallélisme, 11, 21

- threads, 28, 57

Primitives, 10

Principes de conception, 15

Processus

- commutation, 22
- définition, 22
- environnement, 26
- états, 22
- opérations, 23
- ordonnancement, 24

Programme

- binaire objet exécutable, 12
- binaire objet ré-éditable, 12
- format binaire objet, 14
- image binaire, 14
- notion de, 12

Système d'exploitation

- définition, 6, 7
- modes d'exécution, 9
- noyau, 6
- protection mémoire, 9
- temps partagé, 8
- tolérance aux fautes, 9

Virtualisation, 16