Ezenku



Zeste de savoir

# Les systèmes d'exploitation

12 août 2019

# Table des matières

Vous utilisez tous les jours un système d'exploitation sans forcément savoir ce qui se cache derrière. Vous avez certainement l'habitude de Windows, de Linux, de Mac OS, ou d'un autre système d'exploitation moins courant. Ce cours va vous expliquer ce qu'est un système d'exploitation, à quoi il sert, ce qu'il fait et comment il est fabriqué.

**Pré-requis :** des connaissances minimales concernant le fonctionnement d'un ordinateur sont requises pour suivre ce cours. Vous devez savoir des choses comme ce qu'est la mémoire, le processeur, les entrées-sorties, etc. Les notions d'adresse, de registres, d'instructions machine, et d'autres notions de cet acabit seront notamment utilisées dans ce cours.

# 1. Mécanismes d'abstraction des périphériques

Au tout début de l'informatique, avant l'invention des systèmes d'exploitation, les programmes étaient conçus pour un ordinateur en particulier et la compatibilité entre les différents ordinateurs était médiocre. Avec le temps, les informaticiens se sont rendus compte qu'il était nettement mieux de faire en sorte qu'un programme puisse s'exécuter sur plusieurs ordinateurs avec des matériels différents. Il a donc fallu créer des techniques qui permettent à un ou plusieurs programmes de s'exécuter sur toutes sortes de matériels différents : c'est ce qu'on appelle l'**abstraction matérielle**.

Pour cela, les informaticiens ont inventé différents programmes chargés de la gestion du matériel et qui permettent de plus ou moins faire passer le problème de la compatibilité à la trappe. Le système d'exploitation de notre ordinateur est l'un d'entre eux.

## 1.1. Pourquoi utiliser un OS ?

Sans système d'exploitation, un seul et unique programme dans l'ordinateur se charge de tout faire, y compris manipuler le matériel. Dès lors, un programmeur qui souhaite se passer de système d'exploitation doit tout programmer, y compris les parties du programme en charge de la gestion des périphériques, des ports d'entrée-sortie, de la mémoire... Vu qu'on ne gère pas les périphériques et/ou la mémoire de la même façon sur tous les ordinateurs, ce que le programmeur aura réalisé sera difficilement portable sur d'autres ordinateurs, si tant est que cela soit possible.

### 1.1.1. Programmes systèmes et applicatifs

Mais plutôt que de créer un seul et unique programme informatique chargé de tout gérer, on peut segmenter ce programme en plusieurs programmes séparés. Ces programmes peuvent être divisés en deux types :

- les **programmes systèmes** gèrent la mémoire, les périphériques, et les autres programmes applicatifs ;
- les **programmes applicatifs** sont des programmes qui délèguent la gestion de la mémoire et des périphériques aux programmes systèmes.

Pour information, voici une liste des tâches qui sont déléguées aux programmes systèmes :

- gérer une partie de ce qui concerne la mémoire ;
- la gestion de l'exécution de plusieurs programmes sur un même ordinateur ;
- permettre à plusieurs programmes d'échanger des données entre eux si besoin est ;

— gérer tous les périphériques de l'ordinateur ;
— la gestion des fichiers, du réseau, du son et de l'affichage ;
— ...

Généralement, les programmes systèmes sont tous regroupés dans ce qu'on appelle un **système d'exploitation**, aussi appelé OS (*operating system* en anglais). En plus de ces programmes systèmes, le reste de l'OS est composé de programmes applicatifs, dont certains permettent d'afficher une interface qui permet à l'utilisateur de pouvoir utiliser l'ordinateur comme il le souhaite. Évidement, les programmes systèmes ne sont pas les mêmes sur tous les systèmes d'exploitation : c'est une des raisons qui font que certains programmes ne sont compatibles qu'avec *(mettez ici le nom de n'importe quel OS)*.


FIGURE 1.1. – Image de Juju2004, CC-BY-SA 3.0, wikicommons

? L'intérêt de cette séparation ?

Très simple : plutôt que de devoir reprogrammer à chaque fois la gestion de la mémoire et des périphériques, ce travail de programmation est déjà fait. Les programmeurs peuvent se contenter de créer des programmes applicatifs et n'ont pas besoin d'écrire les programmes systèmes.

Cependant, un système d'exploitation ne sait pas utiliser tous les périphériques et ports d'entrée-sortie existants. Pour cela, on a inventé les **pilotes de périphériques**, des programmes systèmes qui permettent à un OS de communiquer avec un périphérique. Évidement, la façon dont le pilote de périphérique va communiquer avec le système d'exploitation est standardisée pour faciliter le tout.

### 1.1.2. Appels système

L'ensemble des opérations qui permettent à notre programme applicatif d'exécuter des programmes systèmes au besoin s'appelle un **appel système**. Pour en effectuer, les programmes applicatifs vont utiliser des fonctionnalités du processeur qu'on appelle des **interruptions**.

#### 1.1.2.1. Interruptions

? Holà, c'est quoi une interruption ?

C'est une fonctionnalité de notre processeur qui va permettre d'arrêter temporairement l'exécution d'un programme pour en exécuter un autre. Ces interruptions ont pour but d'interrompre

un programme, effectuer un traitement et de rendre la main au programme stoppé. L'interruption va exécuter un petit programme auquel on a donné le nom technique de **routine d'interruption**.

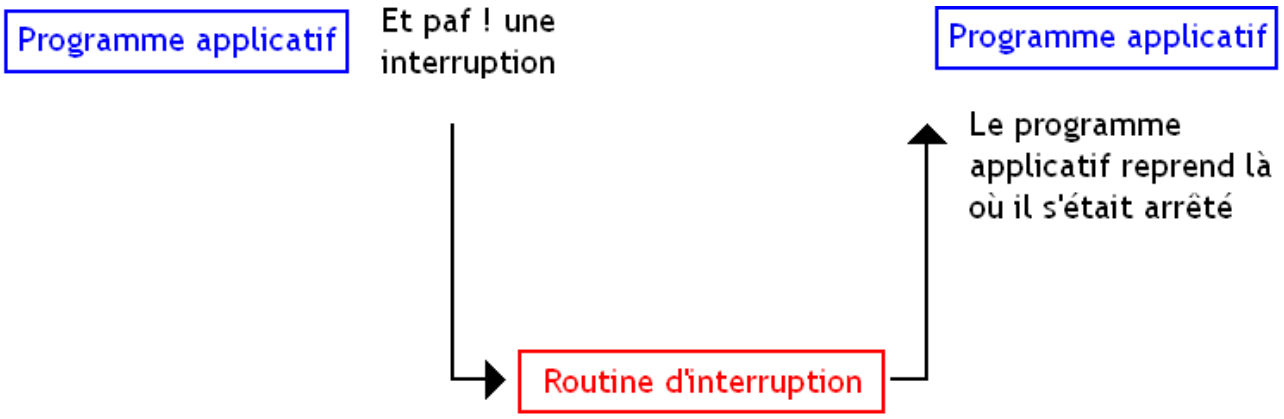


FIGURE 1.2. – Interruption

L'OS et nos pilotes fournissent toutes les routines d'interruptions de bases pour que notre matériel fonctionne : vous voulez écrire une donnée sur le disque dur ? Un programme système exécutant des interruptions est fourni par votre OS. C'est ainsi que les programmes applicatifs peuvent appeler à la demande un programme système : en exécutant l'interruption qui va bien.

#### 1.1.2.2. Vecteur d'interruption

Devant la multiplicité des périphériques, on se doute bien qu'il existe plusieurs routines d'interruption : un programme envoyant un ordre au disque dur sera différent d'un programme agissant sur une carte graphique. Mais il faut bien décider quelle est l'interruption à exécuter suivant la situation : par exemple, exécuter l'interruption de gestion du clavier alors qu'on souhaite communiquer avec le disque dur donnerait un résultat plutôt comique.

Pour retrouver la routine en mémoire, certains ordinateurs utilisent un tableau qui stocke les adresses de chaque routine d'interruption appelé le **vecteur d'interruption**. Ce vecteur d'interruption est initialisé par le BIOS au démarrage de l'ordinateur, mais les pilotes et l'OS vont fournir leurs propres routines. Pour que celles-ci soient exécutées, il suffit à l'OS de « détourner l'interruption », c'est-à-dire de remplacer et mettre à jour le vecteur d'interruption avec les adresses des routines de l'OS. En clair, le vecteur d'interruption ne contiendra plus l'adresse servant à localiser la routine du BIOS, mais renverra vers l'adresse localisant la routine de l'OS.

#### 1.1.2.3. Comment profiter de ces interruptions ?

Pour déclencher une interruption, les programmes applicatifs exécutent une instruction machine spéciale. Sur les processeurs x86, on utilise l'instruction machine `int`. Cette instruction machine a besoin de quelques petites informations pour faire ce qui est demandé et notamment de savoir quelle interruption exécuter. Pour cela, elle a besoin du numéro ou de l'adresse de l'interruption dans le vecteur d'interruption.

Une grande partie de ces routines a besoin qu'on leur fournisse des paramètres, des informations pour qu'elles fassent leur boulot. Par exemple, une routine devant afficher une lettre à l'écran aura besoin qu'on lui passe en entrée la lettre à afficher. Pour chaque routine, il suffira de copier ces paramètres (ou un pointeur vers ceux-ci) dans des petites mémoires ultra-rapides intégrées dans le processeur qu'on appelle les **registres**.

## 1.2. Noyau d'un OS

On l'a vu auparavant, les programmes systèmes et les programmes applicatifs n'ont pas vraiment les mêmes droits : certains peuvent accéder à la mémoire et d'autres non. Pour le moment, on a aucune certitude qu'un programme applicatif n'accédera pas aux périphériques ou fasse des manipulations dangereuses avec la mémoire. Il faut donc trouver un moyen de protéger le matériel contre les actions dangereuses des programmes applicatifs et faire en sorte que ceux-ci restent à leur place. Ce moyen, c'est une technique incorporée dans les processeurs actuels : les **anneaux mémoires**.

### 1.2.1. Espace noyau et espace utilisateur

Les anneaux mémoire sont des zones de mémoire avec des droits d'accès au matériel différents. Pour obtenir une séparation entre programmes systèmes et programmes applicatifs, il faut au minimum deux niveaux de privilèges distincts : un anneau pour les programmes systèmes et un autre pour les programmes applicatifs. L'anneau des programmes systèmes est ce qu'on appelle **l'espace noyau**, alors que l'autre est appelé **l'espace utilisateur**. Il peut y en avoir plus sur certains ordinateurs, par exemple, un processeur x86 32 bits supporte 4 niveaux de privilèges, même si seulement deux sont utilisés en pratique.

Sur certains systèmes, la mémoire n'est pas séparée comme suit, tout dépend de comment le CPU gère la mémoire !

#### 1.2.1.1. Noyau d'un OS

Tout programme qui s'exécute avec le niveau de privilège de l'espace noyau va pouvoir faire tout ce qu'il souhaite : accéder aux périphériques et aux ports d'entrée-sortie, manipuler l'intégralité de la mémoire, etc. Tous les programmes de notre système d'exploitation placés dans l'espace noyau sont ce qu'on appelle **le noyau du système d'exploitation**. La moindre erreur de programmation d'un programme en espace noyau a des conséquences graves : tous vos écrans bleus ou vos *kernel panic* (l'équivalent chez les OS UNIX) sont dus à une erreur en espace noyau (généralement par un pilote de carte 3D ou un problème matériel).

Les programmes en espace utilisateur ne peuvent pas écrire ou lire dans la mémoire des autres programmes ou communiquer avec un périphérique, il doivent déléguer cette tâche à un programme système via une interruption. L'avantage, c'est qu'une erreur commise par un programme en espace utilisateur n'entraine pas d'écrans bleus. C'est donc un gage de sûreté et de fiabilité.

Dans l'espace utilisateur, diverses instructions machines du processeur ne sont pas autorisées car jugées trop dangereuses. Par exemple, sur un processeur à architecture x86, les instruction `in` et `out`, qui permettent respectivement de lire ou écrire un octet depuis un périphérique sont interdites. Dans le même genre, certaines routines d'interruptions ne sont pas exécutables directement en espace utilisateur.

#### 1.2.1.2. Implémentation

Les anneaux mémoires sont gérés par un circuit qui gère tout ce qui a rapport avec la gestion de la mémoire : la **Memory Management Unit**, abréviée en MMU. Le niveau de privilège d'un programme en cours d'exécution est connu grâce à des bits contenus dans des registres spéciaux placés dans le processeur, ce qui permet à la MMU de détecter les violations de droits d'accès. Un programme ne peut changer d'anneau mémoire au cours de son exécution.

Lorsqu'un programme effectue une instruction ou demande l'exécution d'une fonctionnalité du CPU interdite par son niveau de privilège, l'unité de gestion mémoire (la MMU, donc) va déclencher une interruption d'un type un peu particulier : une **exception matérielle**. Généralement, le programme est arrêté sauvagement et un message d'erreur est affiché. Vous savez maintenant d'où viennent vos messages d'erreurs sous votre OS préféré.

### 1.2.2. Principaux types de noyaux

Exécuter un appel système est très lent. Dès lors, exécuter des instructions juste pour changer de niveau de privilège et demander l'exécution d'une routine, devrait de préférence être évité. On se retrouve donc avec deux contraintes : les performances et la sécurité. Avoir de bonnes performances nécessite de diminuer le nombre d'appels systèmes, ce qui se fait en laissant des programmes dans le noyau. Par contre, la sécurité s'obtiendra en mettant un maximum de trucs en espace utilisateur, ce qui augmente le nombre d'appels systèmes. On peut ainsi classer les noyaux en plusieurs types, selon leurs priorité et la façon dont ils gèrent ce genre de problèmes.

#### 1.2.2.1. Noyaux mégalithiques

Avec les **noyaux mégalithiques**, tout l'OS est dans l'espace noyau.

#### 1.2.2.2. Noyau monolithique

Dans le cas du **noyau monolithique**, un maximum de programmes systèmes est placé dans l'espace noyau. Avec ce genre d'organisation, très peu d'appels systèmes sont effectués, ce qui est un avantage en terme de performances. Mais cela est aussi un désavantage en terme de sûreté.

Un **noyau modulaire** est un noyau monolithique qui est divisé en plusieurs parties bien distinctes nommées les modules. Par exemple, chaque pilote de périphérique sera stocké dans un module séparé du reste du noyau. Avec ce genre d'organisation, on peut ne charger que ce dont on a besoin au lancement de l'ordinateur (par exemple, cela permet de ne pas charger le pilote d'un périphérique qui n'est pas branché sur l'ordinateur). Cela permet aussi de rajouter plus facilement des modules dans le noyau sans avoir à refaire celui-ci depuis le début.


FIGURE 1.3. – Image de Julien Sopena, wikicommons, GFDL et CC-BY-SA 3.0

#### 1.2.2.3. Micro noyau

Pour gagner en sureté de fonctionnement, certains créateurs de systèmes d'exploitation ont décidé de ne laisser dans le noyau que les programmes qui ont absolument besoin d'un niveau de privilège élevé. Ces **micro-noyaux** sont souvent très légers : peu de programmes systèmes sont présents dans le noyaux, les autres étant évacués dans l'espace utilisateur. L'avantage, c'est qu'un *bug* a plus de chances de se retrouver dans l'espace utilisateur. Mais cela implique de nombreux appels systèmes entre les programmes systèmes en espace utilisateur et ceux en espace noyau, ce qui réduit les performances.


FIGURE 1.4. – Image de Raphael Javaux, GFDL et CC-BY-SA 3.0, wikicommons

#### 1.2.2.4. Noyau hybride

Dans les **noyaux hybrides**, on garde la même philosophie que pour les micro-noyaux, en étant un peu plus souple : on évacue un maximum de programmes systèmes dans l'espace utilisateur. Néanmoins, certains programmes systèmes, très demandeurs en appels systèmes sont placés en espace noyau. Cela évite de plomber les performances en générant trop d'appels systèmes.


FIGURE 1.5. – Image de Raphael Javaux, GFDL et CC-BY-SA 3.0, wikicommons

#### 1.2.2.5. Exokernels

Enfin, une dernière catégorie de noyaux existe : les ***exokernels***. Celle-ci consiste à extraire le plus de programmes systèmes du noyau tout en conservant la sécurité de fonctionnement en utilisant les anneaux mémoires. En clair, même des programmes systèmes deviennent des

programmes en espace utilisateur qui utilisent un ensemble minimal de fonctionnalités fournies par un noyau réduit à son strict minimum.

Les différentes abstractions matérielles, qui permettent de simplifier la façon dont un programme va devoir gérer la mémoire, le disque dur, le processeur et tout le reste sont donc placés dans l'espace utilisateur. Seul un minuscule noyau existe, qui se charge simplement de contenir quelques routines qu'on a pas pu évacuer dans l'espace utilisateur. Par exemple, dans le cas du disque dur : un exokernel n'implémentera pas de quoi camoufler l'organisation physique du disque dur (cylindre, têtes et pistes) et ne gérera pas de systèmes de fichiers.

L'avantage, c'est qu'un programmeur peut reprogrammer tout seul la gestion de la mémoire, du disque dur ou du reste du système d'exploitation. Dans certains cas particuliers, cela permet soit d'avoir une meilleure sécurité, soit de pouvoir programmer en étant bien plus proche du matériel ou en optimisant nettement mieux notre application.

# 2. Systèmes de fichiers

Vous avez déjà remarqué que lorsque vous éteignez votre ordinateur, le système d'exploitation et les programmes que vous avez installés… ne s'effacent pas. On dit que la mémoire dans laquelle votre OS et vos programmes sont placés est une mémoire de masse. Tout ordinateur contient au moins une mémoire non-volatile, afin de conserver des données quand on l'éteint. Sur nos ordinateurs, ce rôle est le plus souvent rempli par un disque dur. Dans d'autres cas, il l'est par des mémoires Flash. Dans ce qui va suivre, nous allons parler brièvement de l'organisation des données sur une mémoire de masse et comment notre OS gère le tout.

## 2.1. Fichiers et répertoires

On sait donc stocker nos données en utilisant un disque dur. C'est un bon début. Mais le seul problème, c'est que toutes les données que l'on met sur notre disque dur sont codées en binaire. Il est donc impossible de savoir si une portion de disque dur stocke une vidéo, du texte, votre jeu vidéo préféré, ou un programme exécutable. Pour retrouver nos données, il a bien fallu inventer un moyen pour les organiser et pouvoir remettre la main dessus si besoin est. Pour cela, on regroupe nos informations dans ce qu'on appelle des **fichiers**. Ces fichiers sont généralement de simples morceaux de disque dur ou d'une mémoire de masse, sur lesquels un programme peut écrire ce qu'il veut.

Chaque fichier reçoit un nom, qui permet de l'identifier et de ne pas le confondre avec d'autres. L'utilisateur peut évidemment renommer les fichiers, choisir le nom des fichiers et d'autres choses dans le genre. En plus du nom, le système d'exploitation peut mémoriser des informations supplémentaires sur le fichier : la date de création, la quantité de mémoire occupée par le fichier, etc. Ces informations sont appelées des **attributs** de fichier.

### 2.1.1. Formats de données

Les données d'un fichier sont structurées de la manière établie par le programme qui l'utilise. C'est donc lui qui détermine comment ranger les données dans le fichier. Mais qu'on se rassure : les programmeurs ne font généralement pas n'importe quoi (ou alors, ce n'est pas volontaire). Il existe des normes qui décrivent comment les données doivent être organisées dans un fichier. Ces normes sont appelées des **formats de fichiers**.

Identifier le format d'un fichier est relativement simple : il suffit de faire finir le nom du fichier par une extension qui indique son format. Généralement, cette extension de nom commence par un point, suivi d'une abréviation. Par exemple, le .JPEG, le .WAV, le .MP3, le .TXT, etc. sont tous des formats de fichiers comme les autres. Ces formats de fichiers ne sont pas gérés par le système d'exploitation, c'est aux programmes applicatifs de ranger les données convenablement dans les fichiers. Tout ce que peut faire l'OS, c'est lier un format de fichier à une application : il

sait qu'un .PDF doit s'ouvrir avec un lecteur de fichiers .PDF, que les .WAV peuvent s'ouvrir avec votre lecteur vidéo favori, etc.

### 2.1.2. Répertoires

Sur les premiers systèmes d'exploitation, on ne pouvait pas ranger les fichiers, ils étaient tous placés sur le disque dur sans organisation. Quand le nombre de fichier était relativement faible, cela ne posait pas trop de problèmes. Maintenant, compter le nombre de fichiers que vous avez sur votre ordinateur et imaginez qu'on mette tout sur votre bureau - évitez de perdre conscience, si vous le faites vraiment. *Hé* oui, heureusement que les dossiers et **répertoires** sont là pour organiser nos fichiers !

Ces répertoires sont organisés en hiérarchie : un répertoire peut contenir d'autres répertoires et ainsi de suite. Évidemment, cette hiérarchie commence par un répertoire maître, tout en haut de cette hiérarchie.

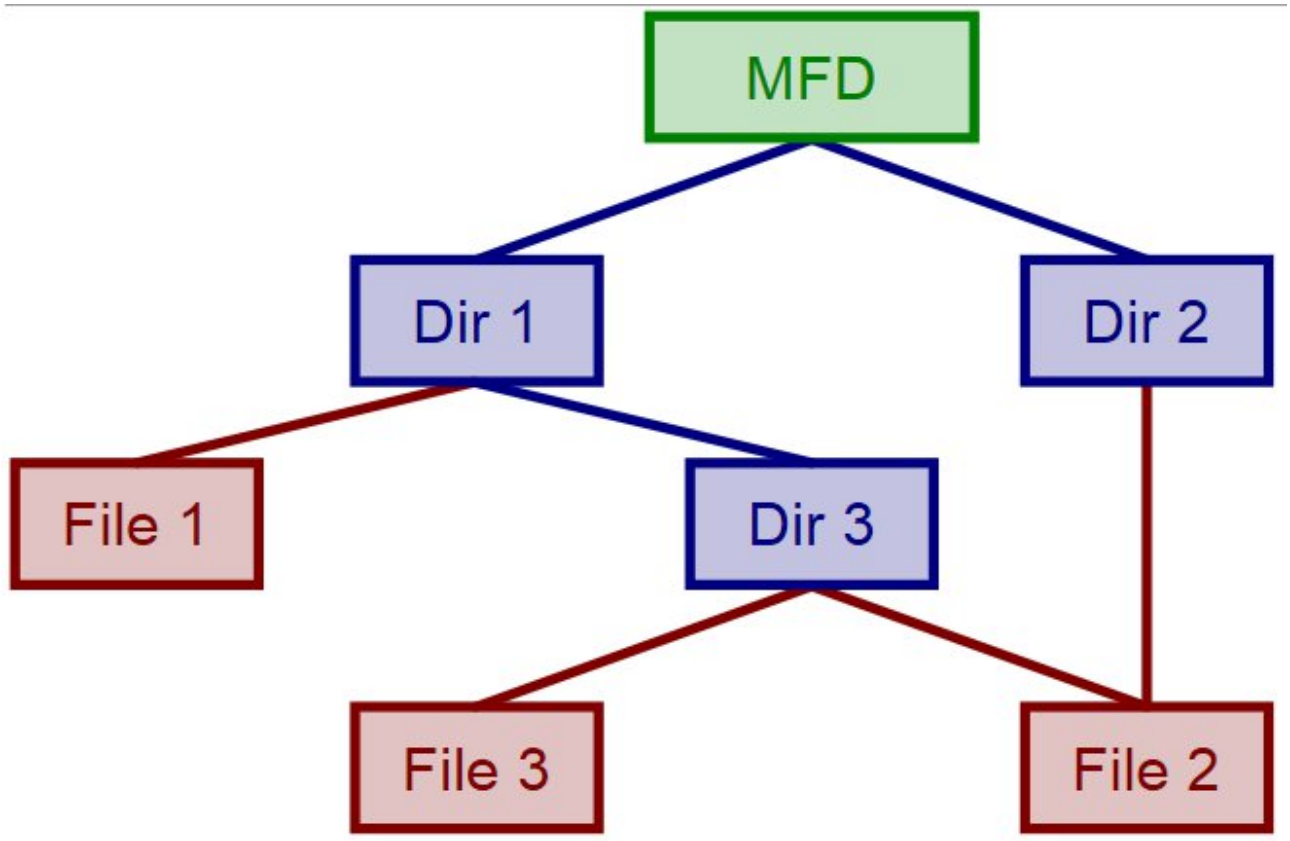


FIGURE 2.1. – Organisation hiérarchique des répertoires

Mais ces répertoires, comment sont-ils représentés sur le disque dur ? Et bien tenez-vous bien : ce sont des fichiers comme les autres...

## 2.2. Gestion des fichiers par l'OS

Un fichier est donc une portion d'une mémoire de masse : disque dur, *Solid State Drive*, CD-ROM, DVD-ROM, etc. Mais comment sont rangés ces blocs de données sur le disque dur ? Cela dépend de l'OS et plusieurs méthodes existent. Ces méthodes sont appelées des **systèmes de fichiers**. Un système de fichier gére deux choses : le stockage des fichiers et leur utilisation.

### 2.2.1. Stockage

Pour commencer, il faut savoir que toutes les mémoires de masse sont découpées en blocs de taille fixe : les **secteurs**. Dans le cas le plus fréquent, ces secteurs sont tous identifiés par un numéro qu'on appelle l'adresse LBA. Certains disques durs étaient adressés avec autre chose qu'un simple numéro, mais je vais passer ce cas là sous silence, l'OS se chargeant souvent de la conversion vers une adresse LBA. Les systèmes de fichiers actuels ne travaillent pas toujours sur la base des secteurs, mais utilisent des *clusters*, des groupes de deux ou trois secteurs consécutifs qui ont pour adresse celle de leur premier secteur.

#### 2.2.1.1. Allocation par bloc

La première méthode consiste à trouver un bloc de secteurs adjacents (consécutifs si vous préférez), capable d'accepter l'ensemble du fichier. Dans ces conditions, repérer un fichier sur le disque dur est relativement simple : il suffit de mémoriser le premier secteur et le nombre de secteurs utilisés.

Cette méthode a de nombreux avantages, notamment pour les performances des lectures et des écritures à l'intérieur d'un fichier. C'est d'ailleurs la voie royale pour le stockage de fichiers sur les CD-ROM ou les DVD, où les fichiers sont impossible à supprimer. Mais sur les disques durs, où les fichiers peuvent être supprimés, les choses changent.

À force de supprimer ou d'ajouter des fichiers de taille différentes, on se retrouve avec des blocs de secteurs assez petits, coincés entre deux fichiers proches. Ces secteurs sont inutilisables, vu que les fichiers à stocker sont trop gros pour rentrer dedans. Une telle situation est ce qu'on appelle de la **fragmentation**. Le seul moyen pour récupérer ces blocs est de compacter les fichiers, pour récupérer l'espace libre. C'est l'opération de défragmentation.


FIGURE 2.2. – Defragmentation

#### 2.2.1.2. Allocation par liste de blocs

Sur d'autres systèmes de fichiers, les clusters d'un fichier ne sont pas forcément consécutifs sur le disque dur. Le système d'exploitation doit garder, pour chaque fichier, la liste des clusters qui correspondent à ce fichier. Cette liste est ce qu'on appelle une liste chainée : chaque cluster indique quel est le cluster suivant (plus précisément, l'adresse du cluster suivant). L'accès à un fichier se fait cluster par cluster : l'accès à un cluster bien précis demande de parcourir le fichier depuis le début, jusqu'à tomber sur le cluster demandé.

Avec cette méthode, la fragmentation disparait. Mais cela ne signifie pas que la défragmentation devient inutile : elle permet de regrouper des blocs dispersés sur le disque dur en un seul bloc de secteurs consécutifs. L'accès aux données est alors plus rapide sur les disques durs, pour des raisons matérielles.

#### 2.2.1.3. Allocation par FAT

On peut améliorer la méthode précédente, en regroupant toutes les correspondances (cluster -> cluster suivant) dans une table en mémoire RAM. Cette table est appelée la FAT, pour **File Allocation Table**. Avec cette méthode, l'accès aléatoire est plus rapide car il demande uniquement de parcourir cette table en mémoire RAM, ce qui est plus court que de parcourir le disque dur. Malheureusement, cette table a une taille assez imposante pour des disques durs de grande capacité : pour un disque dur de 200 gibioctets, la taille de la FAT est de plus de 600 mébioctets.

#### 2.2.1.4. Allocation par i-node

Une autre méthode consiste à ne pas stocker les correspondances entre un cluster et le suivant, mais simplement les adresses des clusters les unes à la suite des autres : on obtient ainsi un ***inode***. Un *inode* se contente ainsi, dans sa forme la plus simple, de lister les adresses des secteur d'un fichier. D'autres organisations plus compliquées reposant sur ce principe sont utilisées sur les systèmes d'exploitation UNIX.

### 2.2.2. Utilisation

Pour ouvrir un fichier, un programme demande au système d'exploitation pour y avoir accès. Si aucune erreur n'a lieu, le programme peut alors lire ou écrire dans le fichier : le système d'exploitation s'occupera de localiser les informations du fichier sur le disque dur. Une fois son travail terminé, le programme va alors fermer le fichier et dire au système d'exploitation qu'il n'en a plus besoin.

Reste qu'ouvrir un fichier n'est pas une opération qui se fait sans formalités : le système d'exploitation vérifie la demande avant d'effectuer l'ouverture (le fichier existe t-il ? Le processus a t-il le droit d'y accéder ?). Dans certaines situations, ce droit lui est refusé et l'OS lève une erreur.

# 3. Allocation mémoire

De nombreuses instructions ou fonctionnalités d'un programme nécessitent de connaitre une adresse dès la compilation (les branchements par exemple). Sur la plupart des OS actuels, l'adresse à laquelle on charge un programme en mémoire n'est presque jamais la même d'une exécution à l'autre. Ainsi, les adresses de destination des branchements et les adresses des données ne sont jamais les mêmes. Il nous faut donc trouver un moyen pour faire en sorte que nos programmes puissent fonctionner avec des adresses qui changent d'une exécution à l'autre. Ce besoin est appelé la ***relocation***.

De plus, un autre problème se pose : un programme peut être exécuté sur des ordinateurs ayant des capacités mémoires diverses et variées et dans des conditions très différentes. Par exemple, un programme doit pouvoir fonctionner sur des ordinateur ayant peu de mémoire sans poser de problèmes. Après tout, quand on conçoit un programme, on ne sait pas toujours quelle sera la quantité mémoire que notre ordinateur contiendra et encore moins comment celle-ci sera partagée entre nos différents programmes en cours d'exécution : s'affranchir de limitations sur la quantité de mémoire disponible est un plus vraiment appréciable. Ce besoin est appelé l'**abstraction matérielle de la mémoire**.

Généralement, plusieurs programmes sont présents en même temps dans notre ordinateur et doivent se partager la mémoire physique. Si un programme pouvait modifier les données d'un autre programme, on se retrouverait rapidement avec une situation non-prévue par le programmeur. Cela a des conséquences qui vont de comiques à catastrophiques et cela fini très souvent par un joli plantage. Il faut donc introduire des mécanismes de **protection mémoire**.

Pour éviter cela, chaque programme a accès à des portions de la mémoire dans lesquelles lui seul peut écrire ou lire. Le reste de la mémoire est inaccessible en lecture et en écriture, à part pour la mémoire partagée entre différents programmes. Ces droits sont différents pour chaque programme et peuvent être différents pour la lecture et l'écriture : on peut mettre certaines portion de mémoire en lecture seule, en écriture seule, etc. Toute tentative d'accès à une partie de la mémoire non-autorisée déclenchera une exception matérielle (rappelez-vous le chapitre sur les interruptions) qui devra être traitée par une routine du système d'exploitation. Généralement, le programme fautif est sauvagement arrêté et un message d'erreur est affiché à l'écran.

Ces détails concernant l'organisation et la gestion de la mémoire sont assez compliqués à gérer et faire en sorte que les programmes applicatifs n'aient pas à s'en soucier est des plus agréable. Ce genre de problème a eu des solutions purement logicielles : on peut citer par exemple l'*overlaying* . Mais d'autres techniques règlent ces problèmes directement au niveau du matériel : on parle de **mémoire virtuelle**.

## 3.1. Mono-programmation

Sur les ordinateurs mono-programmés, capables d'exécuter un seul programme à la fois, la technique la plus souvent utilisée est la technique du **moniteur résident**. Cela consiste à découper la mémoire en deux parties :

— une portion de taille fixe et constante qui sera réservée au système d'exploitation ainsi qu'à quelques autres machins (par exemple la gestion des périphériques) ;
— et le reste de la mémoire qui sera réservé au programme à exécuter.


FIGURE 3.1. – Moniteur résident

### 3.1.1. Chargement d'un programme

Au démarrage d'un programme, un morceau du système d'exploitation, le **chargeur de programme** (***loader*** en anglais), rapatrie le fichier exécutable en RAM et l'exécute. Avec une telle allocation mémoire, cette tâche est relativement simple : l'adresse du début du programme est toujours la même, les *loaders* de ces systèmes d'exploitation n'ont donc pas à gérer la *relocation*, d'où leur nom de ***loaders* absolus**.

Les fichiers objets/exécutables de tels systèmes d'exploitation ne contenaient que le code machine : on appelle ces fichiers des ***Flat Binary***. On peut citer l'exemple des **fichiers .COM**, utilisés par le système d'exploitation MS-DOS (le système d'exploitation des PC avant que Windows ne prenne la relève).

### 3.1.2. Protection mémoire

Protéger les données de l'OS contre une erreur ou malveillance d'un programme utilisateur est nécessaire. La solution la plus courante est d'interdire les écritures d'un programme utilisateur aux adresses inférieures à la limite basse du programme applicatif (la limite haute de l'OS, en terme d'adresses). Pour cela, on va implanter un **registre limite** dans le processeur : ce registre contient l'adresse à partir de laquelle un programme applicatif est stocké en mémoire.

Quand un programme applicatif accède à la mémoire, l'adresse à laquelle il accède est comparée au contenu du registre limite. Si cette adresse est inférieure au contenu du registre limite, le programme cherche à accéder à une donnée placée dans la mémoire réservée au système : l'accès mémoire est interdit, une exception matérielle est levée et l'OS affiche un message d'erreur. Dans le cas contraire, l'accès mémoire est autorisé et notre programme s'exécute normalement.

## 3.2. Multi-programmation

Vous avez sûrement déjà remarqué que vous pouvez parfaitement lancer plusieurs programmes en même temps sur votre ordinateur. Par exemple, vous pouvez télécharger quelque chose grâce à Firefox et regarder une vidéo en même temps sans que cela pose problème. C'est parce que votre système d'exploitation peut exécuter plusieurs programmes en même temps sur le même ordinateur. On dit qu'il est **multitâches**.

Permettre à plusieurs programmes de s'exécuter « en même temps » sur un ordinateur porte un nom : la **multiprogrammation**. Au départ, la multiprogrammation a été inventée pour masquer la lenteur des périphériques. Si un programme doit attendre une donnée en provenance d'un périphérique, on peut exécuter un autre programme durant ce temps d'attente.

Mais l'apparition de la multiprogrammation a posé de nombreux problèmes, notamment au niveau du partage de la mémoire, car sur les premiers OS qui utilisaient la multiprogrammation, les différents programmes devaient se partager la mémoire physique. Chaque programme recevait donc un bloc de mémoire RAM pour lui tout seul (dans ce qui va suivre, nous allons appeler ces blocs des **partitions mémoire**).


Figure 3.2. – Partition mémoire

### 3.2.1. Gestion des partitions libres

Lorsqu'on démarre un programme, l'OS doit trouver un bloc de mémoire vide pour accueillir le programme exécuté. Pour cela, l'OS doit savoir quelles sont les portions de la mémoire inutilisées. Et cela peut se faire de deux manières.

#### 3.2.1.1. Table de bits

La première solution consiste à découper la mémoire en blocs de quelques kibioctets. Pour chaque bloc, l'OS retient si ce bloc est occupé ou libre. Cela prend à peine un bit par bloc : 0 pour un bloc occupé et 1 pour un bloc libre. L'ensemble des bits associé à chaque bloc est mémorisé dans un tableau de bits. Cette technique ne gaspille pas beaucoup de mémoire si l'on choisit bien la taille des blocs. Mais la recherche d'un segment libre demande de parcourir le tableau de bit du début à la fin (en s'arrêtant quand on trouve un segment suffisant), or il s'agit d'une opération très lente.

#### 3.2.1.2. Liste des partitions libres

Une autre méthode consiste à conserver une liste chainée des partitions mémoire. Cette liste est généralement triée suivant les adresses des partitions/blocs libres : des partitions qui se suivent dans la mémoire se suivront dans la liste. Cela permet de faciliter la fusion des blocs libres : si un programme libère un bloc, ce bloc peut être fusionné avec d'éventuels blocs libres adjacents pour donner un seul gros bloc.

Il est aussi possible d'utiliser deux listes séparées pour les trous et les programmes, mais cela complexifie fortement le programme utilisé pour gérer ces listes. On peut aussi en profiter pour trier les listes non par adresse, mais par taille : les plus gros ou plus petits trous seront alors disponibles en premier dans la liste. Cela permet de choisir rapidement les blocs les plus gros ou les plus petits capables d'accueillir un programme, certains algorithmes de sélection du segment seront alors plus rapides.

### 3.2.2. Choix de la partition

Lors de la recherche d'un segment de mémoire capable d'accueillir un programme, on tombe souvent sur plusieurs résultats, autrement dit plusieurs segments peuvent recevoir le programme démarré.

Si on choisit mal la partition, on peut obtenir un phénomène de **fragmentation externe**, c'est-à-dire que l'on dispose de suffisamment de mémoire libre, mais que celle-ci est dispersée dans beaucoup de petits segments vides qui peuvent difficilement stocker une partition à eux tous seuls. Si aucun bloc de mémoire vide n'est suffisamment gros pour combler une demande d'un programme, le système d'exploitation va devoir déplacer les partitions pour les compacter et créer des espaces vides plus gros. En clair, il va devoir déplacer des segments entiers dans la mémoire, ce qui prend beaucoup de temps inutilement.

#### 3.2.2.1. Algorithme de la première zone libre

La solution la plus simple est de placer le programme dans la première partition capable de l'accueillir. Si le programme n'utilise pas tout le segment, on découpe la partition de manière à en créer une pour la zone non-occupée par le programme lancé.

Une variante permet de diminuer le temps de recherche d'un segment adéquat. L'idée est que quand on trouve un segment libre, les segments précédents ont de fortes chances d'être occupés ou trop petits pour contenir un programme. Dans ces conditions, mieux vaut commencer les recherches futures à partir du segment libre. Pour cela, lors de chaque recherche d'un segment libre, l'OS mémorise là où il s'est arrêté et il reprend la recherche à partir de celui-ci.

#### 3.2.2.2. Algorithme du meilleur ajustement

On pourrait penser que quitte à choisir un bloc, autant prendre celui qui contient juste ce qu'il faut de mémoire. Cette méthode demande de parcourir toute la liste avant de faire un choix, ce qui rend la recherche plus longue. Avec une liste de trous triée par taille, cet algorithme est particulièrement rapide. Mais bizarrement, cette méthode laisse beaucoup de partition trop

petites pour être utilisables. En effet, il est rare que les programmes utilisent toute la partition attribuée et laissent souvent du « rab » ce qui fait que la fragmentation externe devient alors importante.

#### 3.2.2.3. Algorithme du plus grand segment

Pour éviter le problème décrit plus haut, on peut procéder de manière à ce que les partitions laissée lors de l'allocation soient les plus grosses possibles pour augmenter leurs possibilités de réutilisation. L'algorithme utilisé consiste donc à choisir le plus grand segment libre capable d'accueillir le programme lancé. Avec une liste de partitions triée par taille, cet algorithme est particulièrement rapide.

### 3.2.3. Protection mémoire

Second problème : comment éviter qu'un programme aille modifier les données d'un autre programme ? Pour cela, l'OS doit vérifier que les accès mémoire d'un programme restent bien dans sa partition. Pour cela, il faut vérifier les débordements par le haut et par le bas, ce qui peut être fait de deux manières.

#### 3.2.3.1. Registres de base et limite

La première solution utilise deux registres :

— un **registre de base** ;
— et un **registre limite**.

Le registre de base stocke l'adresse à laquelle commence la partition et le registre limite conserve l'adresse à laquelle elle se termine. Une implémentation naïve de la protection mémoire consiste à vérifier pour chaque accès mémoire que l'adresse à laquelle le programme veut lire est bien située dans l'intervalle délimité par le registre de base et le registre limite.

#### 3.2.3.2. Protection keys

Toutefois, il existe une autre solution : attribuer à chaque programme une **clé de protection**, qui consiste en un nombre unique (chaque programme a donc une clé différente de ses collègues). La mémoire est fragmentée en blocs de même taille, généralement de 4 kibioctets et le processeur mémorise, pour chacun de ses blocs, la clé de protection du programme qui a réservé ce bloc. À chaque accès mémoire, le processeur compare la clé de protection du programme en cours d'exécution et celle du bloc de mémoire de destination. Si les deux clés sont différentes, alors un programme a effectué un accès hors des clous et il se fait sauvagement arrêter.

### 3.2.4. Relocation

Chaque partition n'a pas une adresse de base fixe, celle-ci dépendant des allocations précédentes. Or, chaque accès à une donnée, chaque branchement, chaque instruction est localisée en mémoire et vouloir y accéder signifie accéder à la bonne adresse. Un programme doit donc connaitre celle-ci dès sa conception. Pour cela, le compilateur considère que le programme commence à l'adresse zéro et laisse l'OS corriger les adresses du programme en fonction de la première adresse de sa partition : il suffit d'additionner cette adresse de début aux adresses manipulées par le programme. Il existe deux grandes méthodes pour faire cette simple addition : la ***relocation*** et le ***position indépendant code***.

#### 3.2.4.1. Relocation

Avec la *relocation*, la correction est réalisée au lancement du programme par l'OS. Dans d'autres cas, la *relocation* peut être réalisée automatiquement par le processeur : il suffit d'additionner l'adresse à lire ou écrire avec le registre de base, pour obtenir l'adresse réelle de la donnée ou de l'instruction dans la mémoire.

#### 3.2.4.2. Position indépendant code

Le *position indépendant code* est une autre solution qui consiste à créer des programmes dont le contenu est indépendant de l'adresse de base et qui peuvent être lancés sans *relocation*. Mais cette méthode demande d'utiliser des techniques logicielles, aujourd'hui incorporées dans les compilateurs et les *linkers*. Mais je passe ce genre de détails sous silence, nous sommes dans un tutoriel sur les OS et non dans un tutoriel sur les *linkers* et les assembleurs.

### 3.2.5. Allocation mémoire

Utiliser des partitions mémoire permet d'implémenter ce que l'on appelle l'**allocation mémoire** : le programme pourra demander à l'OS d'agrandir ou rétrécir sa partition mémoire suivant les besoins. Si notre programme a besoin de plus de mémoire, il peut en demander à l'OS. Et quand il n'a plus besoin d'une portion de mémoire, il peut libérer celle-ci et la rendre à l'OS. Dans les deux cas, le système d'exploitation fournit des appels système pour agrandir ou rétrécir la partition mémoire.

Cela pose toutefois quelques problèmes. Prenons par exemple le cas suivant : deux programmes sont lancés et sont stockés dans deux partitions en mémoire. Ces programmes vont alors régulièrement avoir besoin de mémoire et vont prendre de la mémoire. Imaginez qu'un programme ait tellement grossit qu'on en arrive à la situation suivante :


FIGURE 3.3. – Problèmes d'allocation mémoire

? Imaginez maintenant que le programme n° 1 ait besoin de plus de mémoire, que se passe-il ?

Je suppose que vous voyez bien qu'il y a un problème : il n'y a pas de mémoire libre à la suite du programme n° 1. Pour le résoudre, notre système d'exploitation va devoir déplacer au moins un programme et réorganiser la façon dont ceux-ci sont répartis en mémoire. Ce qui signifie qu'au moins un des deux programme sera déplacé.

## 3.3. Mémoire virtuelle

Avec la mémoire virtuelle, tout se passe comme si notre programme était seul au monde et pouvait lire et écrire à toutes les adresses disponibles à partir de l'adresse zéro. Chaque programme a accès à autant d'adresses que ce que le processeur peut gérer. Dès lors, on se moque du fait que des adresses sont réservées aux périphériques, de la quantité de mémoire réellement installée sur l'ordinateur ou de la mémoire prise par d'autres programmes en cours d'exécution. Pour éviter que ce surplus de fausse mémoire pose problème, on utilise une partie des mémoires de masse (disque durs) d'un ordinateur en remplacement de la mémoire physique manquante.

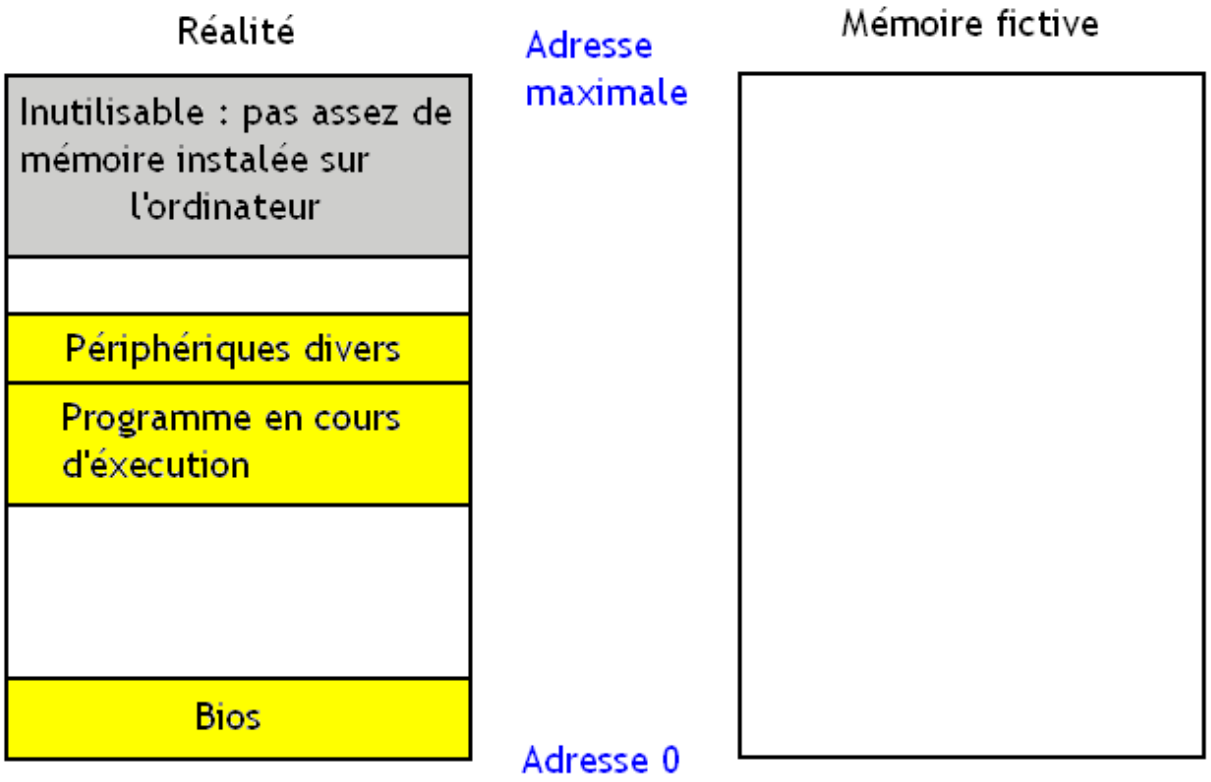


FIGURE 3.4. – Espace d'adressage

Bien sûr, les adresses de cette fausse mémoire vue par le programme sont des adresses fictives qui devront être traduites en adresses mémoire réelles pour être utilisées. Les fausses adresses sont ce

qu'on appelle des **adresses logiques** qui seront les adresses manipulées par notre programme. Par contre, les adresses réelles seront ce qu'on appelle des **adresses physiques**.

Pour implémenter cette technique, il faut rajouter un circuit qui traduit les adresses logiques en adresses physiques : ce circuit est appelé la **Memory Management Unit**. Il faut préciser qu'il existe différentes méthodes pour gérer ces adresses logiques et les transformer en adresses physiques : les principales sont la segmentation et la pagination. La suite du tutoriel va détailler ces deux techniques et quelques autres.

### 3.3.1. Segmentation

La **segmentation** consiste à découper la mémoire virtuelle (la fausse mémoire vue par un programme) en blocs de mémoire de taille variable qu'on appelle **segments**. Pour donner un aperçu de la technique de segmentation, je vais prendre l'exemple de la segmentation sur les processeurs x86. La mémoire virtuelle est découpée en $2^{16}$ segments de $2^{32}$ bits, les adresses font 48 bits. Pour identifier un segment, seuls les 16 bits de poids forts de l'adresse 48 bits sont utiles : ils forment ce qu'on appelle un **sélecteur de segment**. Les 32 bits servent à identifier la position de la donnée dans un segment et sont appelés l'***offset***.

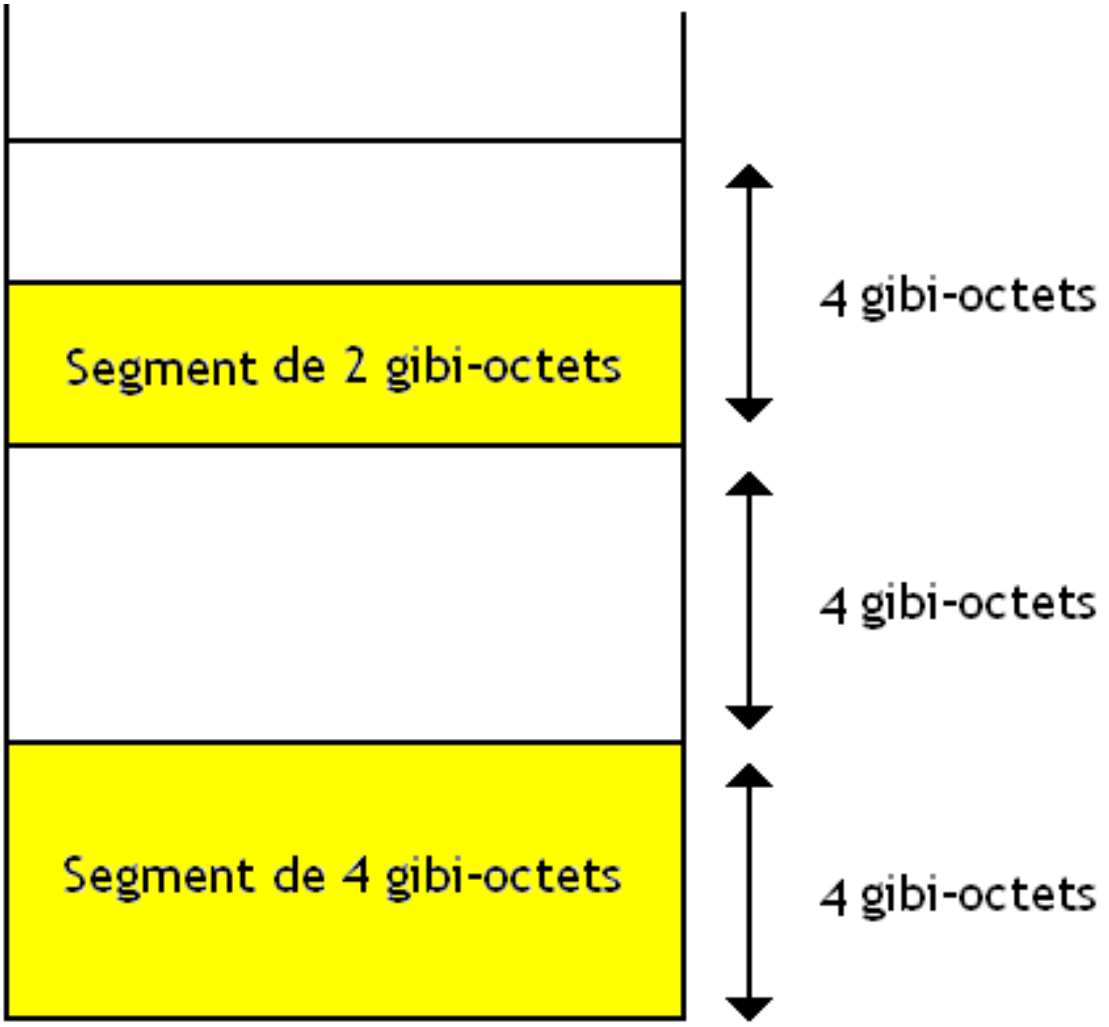

FIGURE 3.5. – Espace d'adressage avec la segmentation x86

#### 3.3.1.1. Relocation

Chaque segment peut être placé n'importe où en mémoire physique.

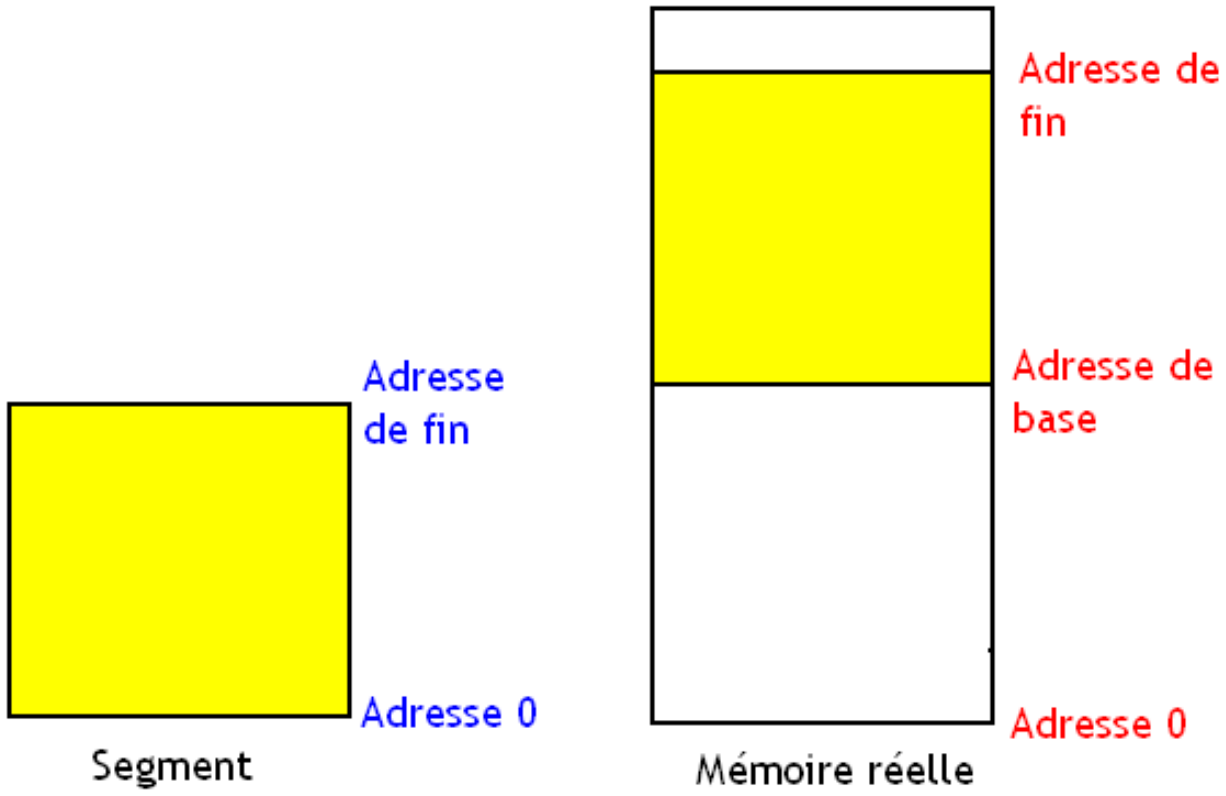


FIGURE 3.6. – Allocation des segments en mémoire physique

Effectuer la *relocation* demande de connaitre l'adresse de base de chaque segment. Cette correspondance segment-adresse de base est stockée dans une table de correspondance en mémoire RAM ou dans des registres du processeur.

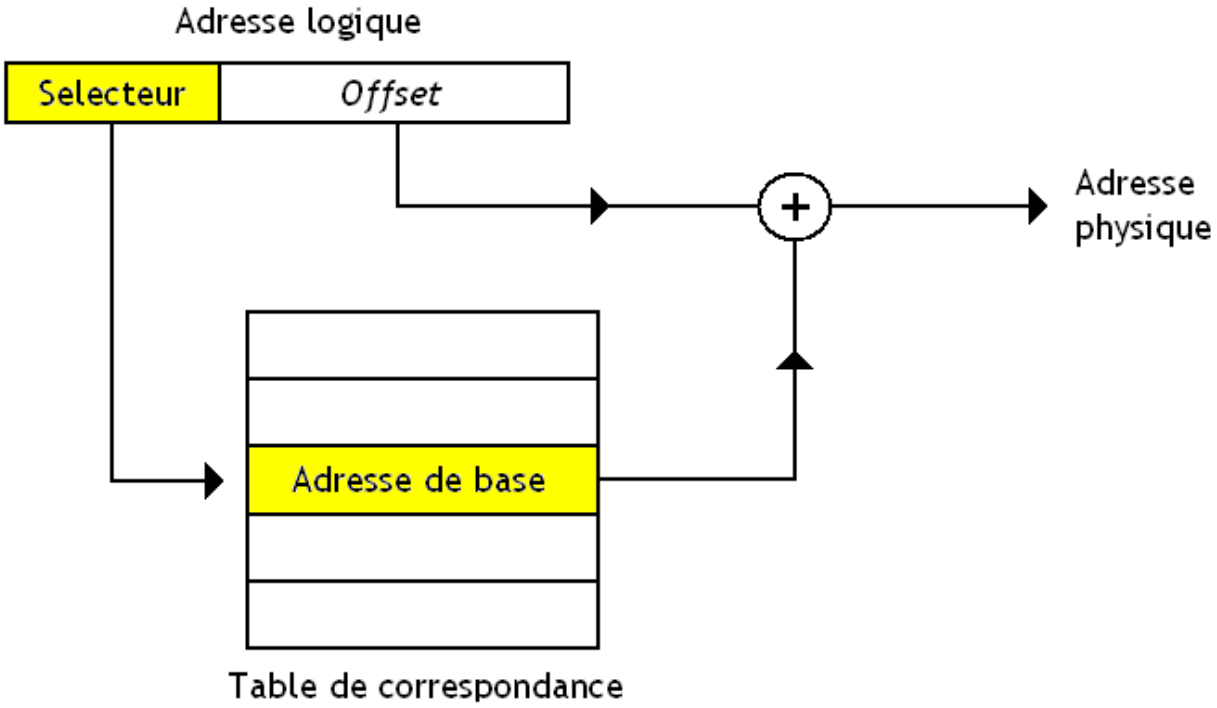


FIGURE 3.7. – Table des segments

Cette table de correspondance est unique pour chaque programme : la même adresse logique ne donnera pas la même adresse physique selon le programme. Vu que les tables des segments sont différentes pour chaque programme, il faut pouvoir mettre à jour le contenu des registres qui mémorisent la table des segments. Cette opération est réalisée lors d'un changement de contexte par l'OS.

Mais il y a moyen de faire plus simple. Quand on accède à un segment, on peut être sur que notre processeur va effectuer un grand nombre d'accès dans ce segment avant d'en changer : on peut éviter d'avoir à préciser le sélecteur à chaque fois. Au lieu d'utiliser une table de correspondance, on mémorise l'adresse physique du segment en cours de traitement dans un registre, le **registre de base**. Tous les accès mémoire suivants utiliseront cette adresse et n'auront pas à préciser le sélecteur de segment.

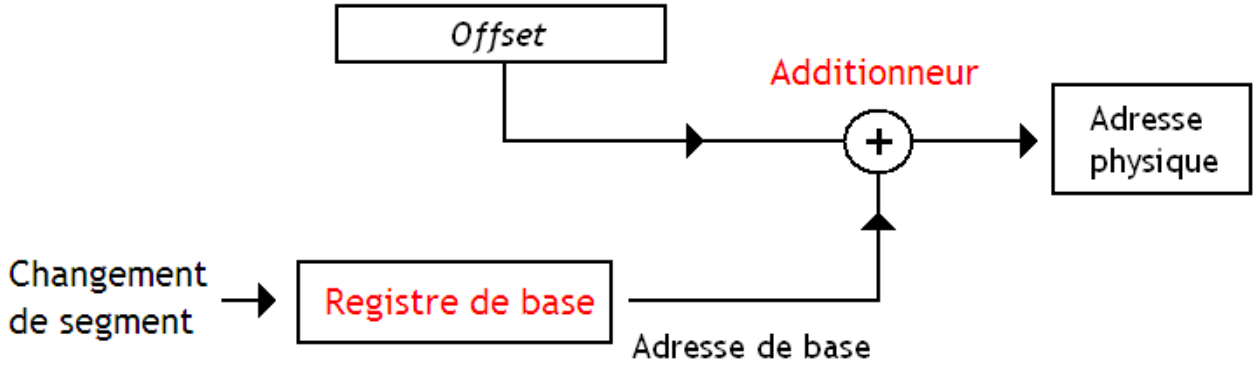


FIGURE 3.8. – Registre de base de segment

#### 3.3.1.2. Protection mémoire

La segmentation permet aussi d'interdire certaines manipulations dangereuses sur la mémoire et de garantir la protection mémoire. La première méthode consiste à tester les accès hors-segment : si jamais l'accès mémoire se fait à une adresse en-dehors du segment en cours de traitement, le processeur lève alors une exception matérielle qui est traitée par le système d'exploitation de l'ordinateur.

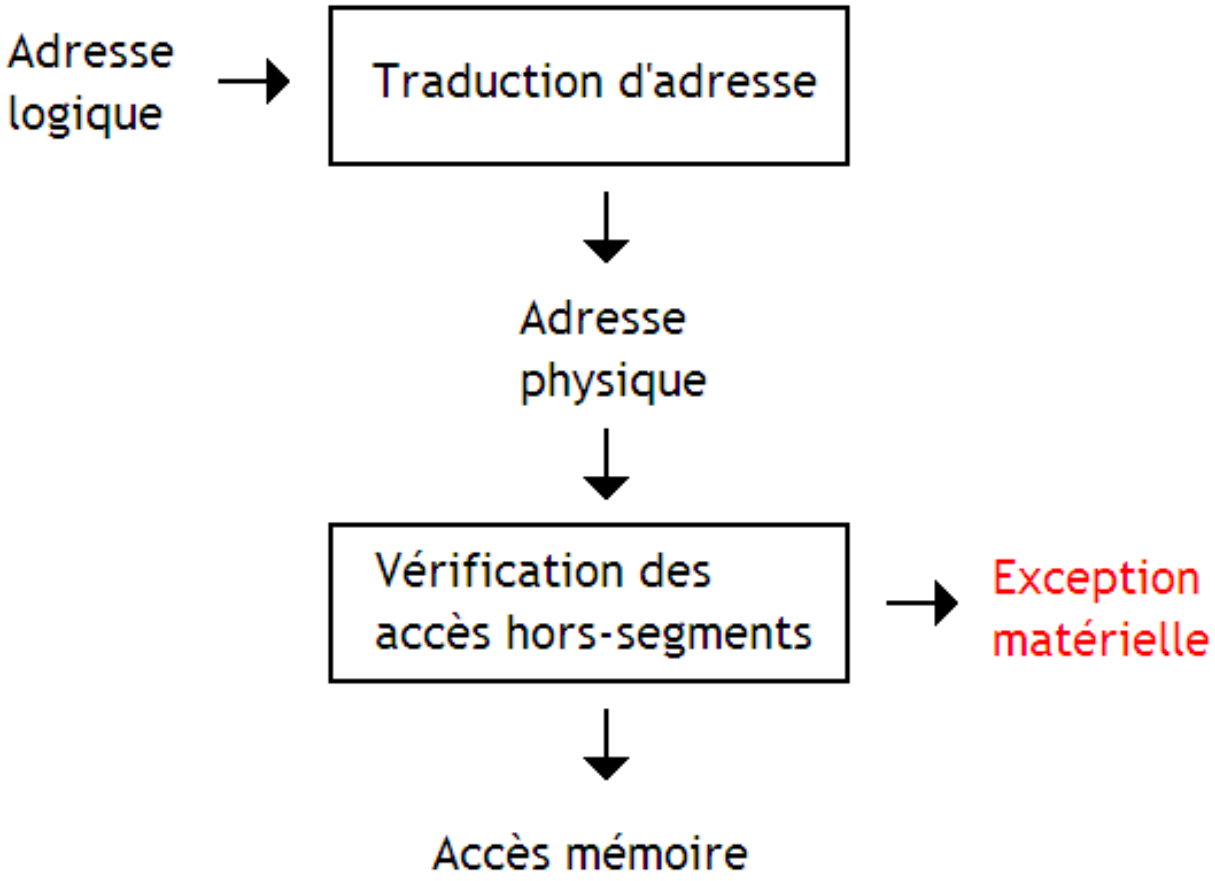


FIGURE 3.9. – Accès hors-segment

Pour effectuer cette vérification, la MMU doit comparer l'adresse de fin du segment et l'adresse à laquelle on veut accéder. Une autre solution demande de comparer l'*offset* avec la longueur du segment. Pour cela, la MMU doit se souvenir de l'adresse de fin de chaque segment ou de sa longueur, cette information étant placée dans la table des segments.

Vient ensuite la gestion des droits d'accès : chaque segment se voit attribuer un certain nombre d'autorisations d'accès qui indiquent si l'on peut lire ou écrire dans un segment, si celui-ci contient des données ou des instructions, etc. Ces informations sont rassemblées, avec les adresses et d'autres informations sur les segments, dans ce qu'on appelle un **descripteur de segment**. Pour se simplifier la tâche, les concepteurs de processeurs et de systèmes d'exploitation ont décidé de regrouper ces descripteurs dans une portion de la mémoire, spécialement réservée pour l'occasion : la **table des descripteurs de segment**.

### 3.3.2. Pagination

De nos jours, la segmentation est obsolète et n'est plus utilisée. À la place, les OS et processeurs utilisent la **pagination**. Avec la pagination, la mémoire virtuelle et la mémoire physique sont découpées en blocs de taille fixe (contrairement aux segments de tailles variables) : ces blocs sont appelés des **pages mémoires**. Toute page de la mémoire virtuelle peut être placée dans n'importe quelle page de la mémoire physique : une page en mémoire physique correspond à une page en mémoire virtuelle. La taille des pages varie suivant le processeur et le système d'exploitation et tourne souvent autour de 4 kibioctets.

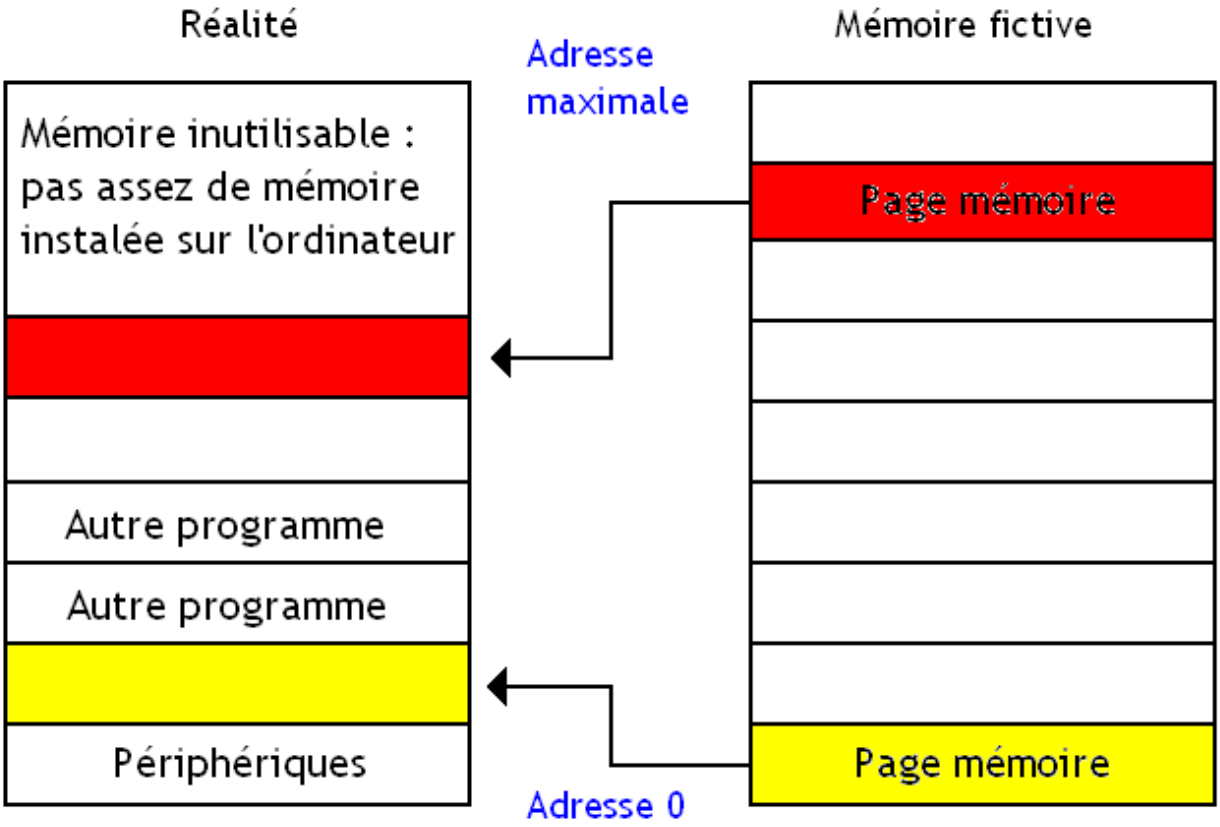


FIGURE 3.10. – Principe de la pagination

#### 3.3.2.1. Translation d'adresse

Par contre, le contenu d'une page en mémoire fictive est rigoureusement le même que le contenu de la page correspondante en mémoire physique : on peut ainsi localiser une donnée par sa position dans la page. Une adresse (logique ou physique) se décompose donc en deux parties : un numéro de page qui identifie la page et une autre permettant de localiser la donnée dans la page. Traduire l'adresse logique en adresse physique demande juste de remplacer le numéro de la page logique en un numéro de page physique.

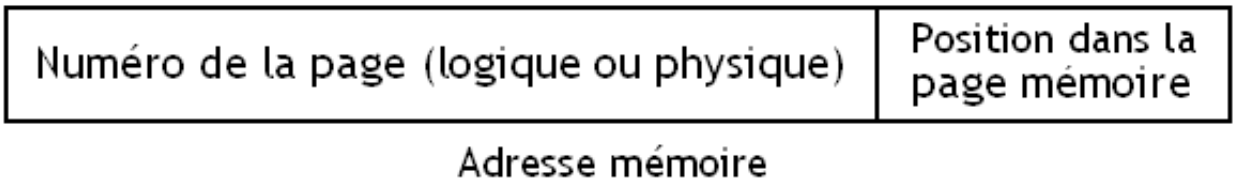


FIGURE 3.11. – Adresse logique/physique avec la pagination

Pour faire cette traduction, il faut se souvenir des correspondances entre numéro de page et adresse de la page en mémoire fictive. Dans le cas le plus simple, ces correspondances sont stockées dans une sorte de table, nommée la **table des pages**. Ainsi, pour chaque numéro (ou chaque adresse) de page logique, on stocke l'adresse de base de la page correspondante en mémoire physique. La table des pages est unique pour chaque programme vu que les correspondances entre adresses physiques et logiques ne sont pas les mêmes. Cette table des pages est souvent stockée dans la mémoire RAM, à un endroit bien précis connu du processeur.

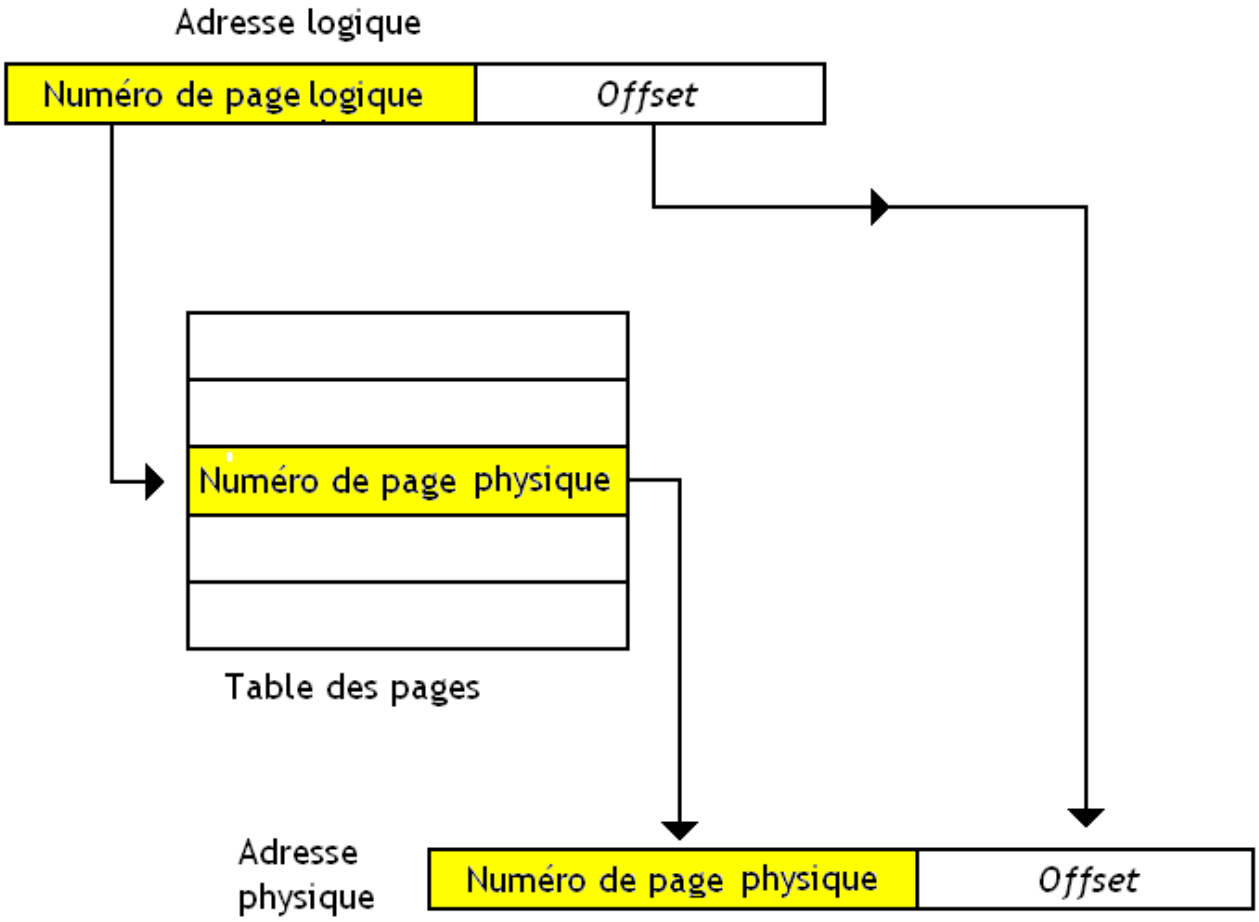


FIGURE 3.12. – Table des pages

Mais la technique précédente a un léger défaut : la table des pages bouffe beaucoup de mémoire. Pour éviter cela, les concepteurs de processeurs et de systèmes d'exploitation ont remarqué que les adresses les plus hautes et/ou les plus basses sont les plus utilisées, alors que les adresses

situées au milieu de l'espace d'adressage sont peu utilisées en raison du fonctionnement de la pile et du tas. Ainsi, les concepteurs d'OS ont décidés de découper l'espace d'adressage en plusieurs sous-espace d'adressage de taille identique qui ont tous leur propre table des pages. Si un sous-espace d'adressage n'est pas utilisé, il n'y a pas besoin d'utiliser de la mémoire pour stocker sa table des pages associée .


FIGURE 3.13. – Exemple d'un espace d'adressage divisé en 4

Sur certains systèmes, la taille d'une table des pages serait beaucoup trop grande en utilisant les techniques vues au-dessus. Pour éviter cela, on a inventé les **tables des pages inversées**. Elles se basent sur le fait que la mémoire physique réellement présente sur l'ordinateur est souvent plus petite que la mémoire virtuelle. Pour faciliter le processus de recherche dans la table, les concepteurs de système d'exploitation peuvent stocker celle-ci avec ce que l'on appelle une table de hachage.


FIGURE 3.14. – Déroulement d'un accès mémoire avec une table des page inversée

#### 3.3.2.2. Remplacement des pages mémoires

La mémoire physique contient moins de pages que la mémoire fictive et il faut trouver un moyen pour que cela ne pose pas de problème. La solution consiste à utiliser des mémoires de stockage comme mémoire d'appoint. Si on a besoin de plus de pages mémoire que la mémoire physique n'en contient, certaines pages mémoire vont être déplacées sur le disque dur pour faire de la place.

Les pages localisées sur le disque dur doivent néanmoins être chargées en RAM avant d'être utilisables. Lorsque l'on veut traduire l'adresse logique d'une page mémoire déplacée sur le disque dur, la MMU ne va pas pouvoir associer l'adresse logique à une adresse en mémoire RAM. Elle va alors lever une **exception matérielle** dont la routine rapatriera la page en mémoire RAM.

Charger une page en RAM ne pose aucun problème tant qu'il existe de la RAM disponible : on peut charger la donnée dans une page inoccupée. Mais si toute la RAM est pleine, il faut déplacer une page mémoire en RAM sur le disque dur pour faire de la place. Tout cela est effectué par la fameuse routine du système d'exploitation dont j'ai parlé plus haut.

Il existe différents algorithmes qui permettent de décider quelle page supprimer de la RAM. Ces algorithmes ont une importance capitale en terme de performance : si on supprime une donnée dont on aura besoin dans le futur, il faudra recharger celle-ci, ce qui prend du temps. Pour éviter cela, le choix de la page doit être fait avec le plus grand soin.

Voici une liste non exhaustive de ces algorithmes :

— Aléatoire : on choisit la page au hasard.
— FIFO : on supprime la donnée qui a été chargée dans la mémoire avant toute les autres.
— LRU : on supprime la donnée qui été lue ou écrite pour la dernière fois avant toute les autres.
— LFU : on vire la page qui est lue ou écrite le moins souvent comparé aux autres.
— etc.

Ces algorithmes ont chacun deux variantes : une locale et une globale. Avec la version locale, la page qui va être rapatriée sur le disque dur est une page réservée au programme qui est la cause du manque de pages libres. Avec la version globale, le système d'exploitation va choisir la page à virer parmi toutes les pages présentes en mémoire vive.

Sur la majorité des systèmes d'exploitation, il est possible d'interdire le déplacement de certaines pages sur le disque dur. Ces pages restent alors en mémoire RAM durant un temps plus ou moins long, parfois en permanence. Cette possibilité simplifie la vie des programmeurs qui conçoivent des systèmes d'exploitation : essayez donc d'exécuter une interruption alors que la page contenant son code est placée sur le disque dur.

#### 3.3.2.3. Protection mémoire

Avec la pagination, chaque page a des droits d'accès : on peut autoriser ou interdire la lecture ou l'écriture, voir l'exécution du contenu d'une page. Pour cela, il suffit de rajouter des bits dans la table des pages pour mémoriser ces autorisations. Suivant la valeur de ces bits, la page sera accessible ou non.

De plus, chaque page est attribuée à un programme en particulier puisqu'un programme n'a pas besoin d'accéder aux données d'une page réservée à un autre. Autant limiter les erreurs potentielles en spécifiant à quel programme appartient une page. Pour cela, des bits permettant d'identifier le programme possesseur de la page sont ajoutés en plus des adresses et des bits de gestion des droits d'accès en lecture/écriture dans la table des pages.

# 4. Processus et threads

Entre un fichier exécutable sur le disque dur et un programme en cours d'exécution, il y a un monde. Le fichier exécutable doit être copié en mémoire RAM par le chargeur de programme, le système d'exploitation doit gérer tout ce qui a trait à l'allocation mémoire, configurer certaines de ses structures et j'en passe et des meilleures. Le résultat final de ces opérations est ce qu'on appelle un **processus**, un programme chargé en mémoire et en cours d'exécution sur l'ordinateur.

Pour manipuler des processus, l'OS doit mémoriser des informations sur eux : à quel fichier exécutable correspond le processus, où se situe-t-il en mémoire et ainsi de suite. Toutes ces informations sont stockées dans ce qu'on appelle un ***Process Control Block***, une portion de la mémoire dans laquelle le système d'exploitation va stocker toutes les informations attribuées à un processus.

## 4.1. Vie et mort des processus

### 4.1.1. Création de processus

Les processus peuvent naître et mourir. La plupart des processus démarre quand l'utilisateur demande l'exécution d'un programme, n'importe quel double-clic sur une icône d'exécutable en est un bon exemple. Mais, le démarrage de la machine en est un autre exemple. En effet, il faut bien lancer le système d'exploitation, ce qui demande de démarrer un certain nombre de processus. Après tout, vous avez toujours un certain nombre de services qui se lancent avec le système d'exploitation et qui tournent en arrière-plan.

À l'exception du premier processus, lancé par le noyau, les processus sont toujours créés par un autre processus. On dit que le processus qui les a créés est le processus parent. Les processus créés par le parent sont appelés processus enfants. On parle aussi de père et de fils. Certains systèmes d'exploitation conservent des informations sur qui a démarré quoi et mémorisent ainsi qui est le père ou le fils pour chaque processus. L'ensemble forme alors une **hiérarchie de processus**.

Sur la majorité des systèmes d'exploitation, un appel système suffit pour créer un processus. Mais certains systèmes d'exploitation (les unixoïdes, notamment) ne fonctionnent pas comme cela. Sur ces systèmes, la création d'un nouveau processus est indirecte : on doit d'abord copier un processus existant avant de remplacer son code par celui d'un autre programme. Ces deux étapes correspondent à deux appels systèmes différents.

Dans la réalité, le système d'exploitation ne copie pas vraiment le processus, mais utilise des optimisations pour faire comme si tout se passait ainsi.

### 4.1.2. Destruction de processus

Si les processus peuvent être créés, il est aussi possible de les détruire. De façon générale, un processus peut se terminer de deux façons :

— de façon normale (il a terminé son exécution) ;
— de façon anormale (le processus a commis une « faute », par exemple en tentant d'accéder à une zone mémoire protégée ou en réalisant une division par zéro, ou s'est trouvé en présence d'un problème ou d'une erreur l'empêchant de poursuivre).

Dans tous les cas, le processus père est informé de la terminaison du processus fils par le système d'exploitation. Le processus fils passe alors dans un mode appelé **mode zombie** : il attend que son père prenne connaissance de sa terminaison. Lorsque le processus se termine, le système récupère tout ce qu'il a attribué au processus (PCB, espaces mémoire, etc.), mais garde une trace du processus dans sa table des processus. C'est seulement lorsque le père prend connaissance de la mort de son fils qu'il supprime l'entrée du fils dans la table des processus.

Et si un processus père est terminé avant son fils, il se passe quoi ?

Bonne question ! Nous avons dit plus haut qu'un processus devait toujours avoir un père. C'est le processus qui est le père de tous qui « adopte » le processus qui a perdu son père et qui se chargera alors de le « tuer ».

## 4.2. Ordonnancement

Pouvoir lancer plusieurs programmes en même temps est quelques chose de commun. Mais tout cela doit pouvoir fonctionner avec des ordinateurs qui ne sont pourvu que d'un seul processeur. Or, un processeur ne peut exécuter qu'un seul programme à la fois (il existe toutefois des exceptions). Pour éviter tout problème, le système d'exploitation utilise une technique pour permettre la multiprogrammation sur les ordinateurs à un seul processeur : **l'ordonnancement**. Cela consiste à constamment *switcher* entre les différents programmes qui doivent être exécutés. Ainsi, en *switchant* assez vite, on peut donner l'illusion que plusieurs processus s'exécutent en même temps.


FIGURE 4.1. – Image de Raphael Javaux, GFDL et CC-BY-SA 3.0, wikicommons

### 4.2.1. États d'un processus

Tous les programmes n'ont pas forcément besoin du processeur, certains attendent par exemple la réponse d'un périphérique et ne peuvent donc rien faire durant cette période. Pour éviter de les exécuter inutilement, l'OS se souvient que ces processus sont bloqués en attente des entrées-sorties. Même chose pour les programmes qui attendent le processeur : l'OS doit savoir quel est le programme en cours d'exécution et quels sont ceux qui attendent. Pour cela, on va donner à chaque processus un état qui permettra de savoir si notre programme veut s'exécuter sur le processeur.

Un processus peut être dans (au moins) trois états :

— **Élu** : un processus en état élu est en train de s'exécuter sur le processeur.
— **Prêt** : celui-ci a besoin de s'exécuter sur le processeur et attend son tour.
— **Bloqué** : celui-ci n'a pas besoin de s'exécuter (par exemple, parce que que celui-ci attend une donnée en provenance d'une entrée-sortie).

Il arrive qu'un programme stoppe son exécution. Il peut y avoir plusieurs raisons à cela :

— il a besoin d'accéder à un périphérique et passe donc à l'état bloqué ;
— il n'a plus rien à faire et a fini (temporairement ou définitivement) son exécution ;
— il est interrompu par le système d'exploitation, histoire de laisser la place à un autre programme.

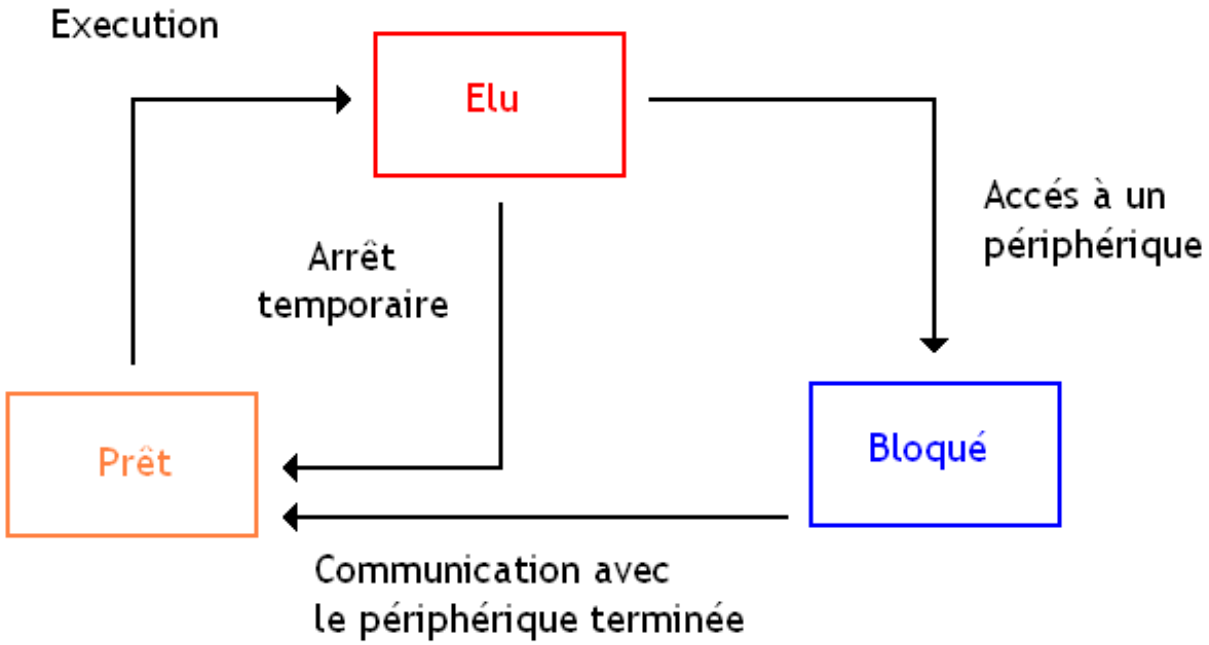


FIGURE 4.2. – Ordonnancement

### 4.2.2. Sélection

Lors d'un changement d'état, le processus en cours d'exécution passe en état bloqué ou prêt. En même temps, un processus va passer de l'état prêt à l'état élu. Reste qu'il faut choisir un programme dans la liste d'attente et c'est le rôle d'un programme système spécialisé :

l'**ordonnanceur**. Pour utiliser notre processeur au mieux, il faut faire en sorte que tous les processus aient leur part du gâteau et éviter qu'un petit nombre de processus monopolisent le processeur. Bien choisir le processus à exécuter est donc une nécessité.

On peut préciser qu'il existe deux grandes formes d'ordonnancements : l'**ordonnancement collaboratif** et l'**ordonnancement préemptif**. Dans le premier cas, c'est le processus lui-même qui décide de passer de l'état élu à l'état bloqué ou prêt. Pour ce faire, le programme en question doit exécuter un appel système bien précis qui rendra la main à l'ordonnanceur. Dans ces conditions, il n'est pas rare qu'un processus « un peu égoïste » décide de ne pas rendre la main et monopolise le processeur. Un bon exemple d'ordonnancement collaboratif n'est autre que les co-routines, des fonctionnalités de certains systèmes d'exploitations ou langages de programmation.

Avec l'ordonnancement préemptif, c'est le système d'exploitation qui stoppe l'exécution d'un processus. L'ordonnancement préemptif se base souvent sur la technique du **quantum de temps** : chaque programme s'exécute durant un temps fixé une bonne fois pour toute. En clair, toutes les « x » millisecondes, le programme en cours d'exécution est interrompu et l'ordonnanceur exécuté. Ainsi, il est presque impossible qu'un processus un peu trop égoïste puisse monopoliser le processeur au dépend des autres processus. La durée du quantum de temps doit être grande devant le temps mis pour changer de programme. En même temps, on doit avoir un temps suffisamment petit pour le cas où un grand nombre de programmes s'exécutent simultanément. Généralement, un quantum de temps de 100 millisecondes donne de bons résultats.

Divers algorithmes permettent de choisir un processus dans la file d'attente. Pour faire simple, on peut classer ces algorithmes en trois classes :

— traitement par lots ;
— traitement interactif ;
— traitement temps réel.

Nous n'aborderons pas le traitement temps réel, vu qu'il s'agit d'un cas assez spécial qui demanderait une petite connaissance de ce qu'est le temps réel et les contraintes associées. Mais nous allons voir le traitement par lots et le traitement interactif.

On peut préciser que certains algorithmes vont privilégier certains programmes sur d'autres. Chaque programme aura alors une priorité, codée par un nombre, plus ou moins grande. Plus celle-ci est grande, plus l'ordonnanceur aura tendance à élire souvent ce processus ou plus ce processus s'exécutera sur le processeur longtemps.

#### 4.2.2.1. Traitement par lots

Tout d'abord, on va commencer par un système d'exploitation qui exécute des programmes les uns après les autres, jusqu'à ce qu'ils finissent leur travail ou décident eux-mêmes de stopper leur exécution, soit pour accéder à un périphérique, soit pour laisser la place à un autre programme. Ce genre de chose est notamment très utilisé dans les banques ou les grosses entreprises, pour sauvegarder les comptes ou mettre à jour de grosses bases de données.

Avec l'algorithme **Premier arrivé, premier servi**, l'ordonnanceur exécute le programme entré dans la file d'attente : les programmes sont exécutés dans l'ordre dans lequel ils sont rentrés dans la file d'attente. Cet algorithme est particulièrement trivial à programmer : une simple liste chainée suffit.

Avec l'algorithme **Dernier arrivé, premier servi**, l'ordonnanceur exécute le programme entré en dernier dans la file d'attente et non le premier. Cet algorithme peut souffrir d'un phénomène assez particulier : dans certains cas, un programme peut très bien mettre énormément de temps avant d'être exécuté. Si vous exécutez beaucoup de programme, ceux-ci seront rentrés dans la file d'attente avant les tout premiers programmes. Ceux-ci doivent alors attendre la fin de l'exécution de tous les programmes rentrés avant eux.

Avec l'algorithme du **job le plus court en premier**, on suppose que les temps d'exécution des différents programmes sont connus à l'avance et sont parfaitement bornés. Cette contrainte peut sembler absurde, mais elle a un sens dans certains cas assez rares dans lesquels on connait à l'avance le temps mis par un programme pour s'exécuter. Dans ce cas, l'algorithme est simple : on exécute le programme qui met le moins de temps à s'exécuter en premier. Une fois celui-ci terminé, on le retire de la file d'attente et on recommence.

L'algorithme ***Shortest Remaining Time Next*** est une variante préemptive de l'algorithme précédent. Le fonctionnement est donc identique, à un point de détail prêt : que se passe-t-il si un nouveau processus est ajouté ? Dans ce cas, on compare le temps d'exécution du processus ajouté avec celui du processus en cours d'exécution. Si le processus ajouté prend moins de temps, alors il est exécuté directement. Néanmoins, cet algorithme a un léger problème : dans certaines conditions, des processus peuvent attendre indéfiniment leur tour. En effet, il suffit pour cela d'ajouter en permanence de nouveaux processus plus prioritaires avant leur exécution, qui passeront systématiquement devant le processus en attente.

#### 4.2.2.2. Traitement interactif

Maintenant, il est temps de passer aux algorithmes que l'on trouve sur nos OS actuels. Il s'agit des algorithmes destinés à des environnements multimédias où l'utilisateur veut une réactivité optimale.

Une première idée consiste à adapter l'algorithme du « job le plus court en premier » sur les systèmes interactifs. Le seul problème est que le temps d'exécution des processus n'est pas connu. La seule solution consiste à effectuer une estimation de ce temps d'exécution à partir des temps d'exécution précédents.

L'**algorithme du tourniquet** utilise la méthode du quantum de temps. Avec cet algorithme, chaque programme est exécuté l'un après l'autre, dans l'ordre. Quand son quantum se termine, le programme doit attendre que tous les autres programmes aient eu droit à leur tour. Pour cela, à la fin de chaque quantum de temps, on place le programme à la fin de la liste d'attente. Cet algorithme est redoutablement efficace et des OS comme Windows ou Linux l'ont utilisé durant longtemps. Mais cet algorithme a un défaut : le choix du quantum de temps doit être calibré au mieux et n'être ni trop court, ni trop long.

L'algorithme des **files d'attentes multiple-niveau** ordonnance aux mieux un ensemble de programmes aux temps d'exécution et aux particularités disparates en donnant la priorité aux programmes rapides et à ceux qui accèdent souvent aux périphériques.

Cet algorithme utilise plusieurs files d'attentes, classées de la plus basse à la plus haute, auxquelles ont donne des quantum de temps différents. Plus une file est haute, plus son quantum est petit. Quand on exécute un programme, celui-ci est placé dans la file d'attente la plus haute. Si un programme utilise la totalité de son quantum de temps, c'est le signe qu'il a besoin d'un quantum plus gros : il descend alors d'un niveau. La seule façon pour lui de remonter d'un niveau est de ne plus utiliser complètement son quantum de temps. Ainsi, un programme va changer de file suivant son temps d'exécution et se stabilisera dans la file dont le quantum de temps est optimal (ou presque).

Dans certaines variantes de cet algorithme, chaque file d'attente peut utiliser un algorithme d'ordonnancement différent ! L'utilité, c'est que chaque file aura un algorithme d'ordonnancement adapté pour des programmes plus ou moins rapides.

L'**ordonnancement par loterie** repose sur un ordonnancement aléatoire. Chaque processus se voit attribuer des tickets de loterie : pour chaque ticket, le processus a droit à un quantum de temps. À chaque fois que le processeur change de processus, il tire un ticket de loterie et le programme qui a ce ticket est alors élu. Évidemment, certains processus peuvent être prioritaires sur les autres : ils disposent alors de plus de tickets de loterie que les autres. Dans le même genre, des processus qui collaborent entre eux peuvent échanger des tickets. Comme quoi, il y a même moyen de tricher avec de l'aléatoire...

### 4.2.3. Changements d'états

Pour que notre programme puisse reprendre où il en était quand l'ordonnanceur lui redonne la main, il faut impérativement qu'un certain nombre de données soient restaurées. Par exemple, les registres du processeur doivent redevenir ce qu'ils étaient quand le programme a été interrompu. Le **contexte d'exécution** d'un processus est l'ensemble de ces informations. Ce contexte d'exécution est sauvegardé en RAM quand l'ordonnanceur interrompt l'exécution du programme. Pour reprendre l'exécution d'un programme là ou il en était, il suffit de remettre en place le contexte sauvegardé.

Une fois un processus élu, il doit être exécuté par notre processeur, c'est le rôle du **dispatcher**. Celui-ci stoppe le programme en cours d'exécution et restaure le contexte du programme élu. Le système d'exploitation doit donc mémoriser les contextes de tous les processus. Cette ensemble de contexte est appelé la **table des processus**. Certains systèmes d'exploitation utilisent deux listes : une pour les processus prêts et une autre pour les processus bloqués.

L'ensemble des processus prêts est ce qu'on appelle la **file d'attente**. Quand un programme démarre, celui-ci est ajouté à cette file d'attente. Les programmes ayant fini leur exécution ou qui passent en état bloqué sont retirés de la liste d'attente. Cette file d'attente est implémentée par ce qu'on appelle une liste chainée. Bien sûr, cette file d'attente n'accepte qu'un nombre limité de programmes : l'ordonnanceur décide toujours si oui ou non un programme peut être ajouté à cette liste et le nombre de programmes dans cette liste a une limite fixée par le système d'exploitation.

## 4.3. Communication entre processus

Dans les chapitres précédents, nous avons fait la confusion entre programmes et processus, notamment pour ce qui est de la protection mémoire. Nous avons dit que deux programmes ne pouvaient pas se marcher l'un sur l'autre, dans le sens où chaque programme n'a accès qu'à une portion de mémoire qui lui est dédiée et pas au reste de la mémoire. Cela vaut toujours pour les processus, qui sont placés dans des espaces mémoire dédiés où seuls eux peuvent écrire, on parle d'**isolation des processus**. Il faut cependant noter que tous les systèmes d'exploitation n'implémentent pas cette isolation des processus, MS-DOS en étant un exemple.

### 4.3.1. Avec isolation des processus

Des processus peuvent avoir besoin de s'échanger des données alors qu'ils s'exécutent sur un même ordinateur, ce qui est très compliqué sur les systèmes avec isolation des processus. Sur ces systèmes, on est obligé d'utiliser des mécanismes de **communication entre processus**. La méthode la plus simple consiste à partager un bout de mémoire entre processus, qui pourront lire ou écrire dedans pour échanger des données (cela peut être un fichier, un morceau de mémoire RAM, etc). Cette méthode pose toutefois des problèmes assez spécifiques et plutôt complexe, aussi, nous ne les aborderons pas ici. Mais il existe des méthodes plus simples, que nous allons voir maintenant.

#### 4.3.1.1. Échange de messages

L'**échange de messages** est un autre moyen pour faire communiquer deux processus. Avec celui-ci, les processus s'échangent des blocs de données de taille fixe, appelés **messages**. Ce message contient des données, qui peuvent être de tout type simple (entier positif ou négatif, flottant positif ou négatif, caractère isolé ou chaine de caractères) ou une combinaison de ces éléments (plusieurs nombres, des nombres et une chaine de caractères, ...). De plus, il contient un type, un entier positif, choisi par le programmeur, qui indique quel est le contenu du message. Il n'y a que trois restrictions sur le contenu du message :

- le type doit être présent ;
- le format des données doit être connu ;
- la taille des données doit être connue pour pouvoir être stockée par le processus qui recevra le message.

Sur certaines implémentations, les messages sont envoyés directement, sans aucune mise en attente. Le processus destinataire du message est alors prévenu qu'on souhaite communiquer avec lui et doit arrêter ce qu'il fait pour échanger avec l'émetteur du message. Dit autrement, les deux processus doivent se synchroniser pour l'échange. On parle alors tout simplement de **passage de message**. Cela fonctionne bien quand les deux processus ne sont pas sur le même ordinateur et communiquent entre eux via un réseau local ou par Internet.

Avec les **tubes** et les **files de messages** (*message queue* en anglais ou MSQ en abrégé), le processus destinataire d'un message et son émetteur n'ont pas besoin de se synchroniser. Il est dès lors possible d'envoyer un message sans avoir à prévenir le processus destinataire. Pour cela, les messages que s'envoient les processus sont mis en attente dans une file d'attente (tant que celle-ci n'est pas remplie, auquel cas le processus émetteur se bloque). Les processus destinataires

peuvent consulter cette file de message quand ils le souhaitent, notamment pour voir si on leur a envoyé des données (si la file d'attente est vide, ils se bloquent en attendant des données qui leur sont destinés).

Dans tous les cas, les données envoyées via la file d'attente doivent de préférence être récupérées dans l'ordre d'envoi. Pour cela, le processus récupère systématiquement le message le plus ancien. Ainsi, les messages sont triés dans leur ordre d'envoi et sont retirés dans cet ordre par le processus destinataire. Notez que j'ai bien dit « retiré » et non pas « lus » : une fois qu'un message est consulté, il est retiré de la file. C'est ce qu'on appelle un fonctionnement « FIFO » pour *First In First Out*, premier entré, premier sorti.

**4.3.1.1.1. Les tubes** Le premier type de file d'attente est celui des tubes. La différence entre les deux tient dans le fait que les tubes permettent à seulement deux processus de communiquer, qui plus est dans un seul sens, là où une file de message ne souffre pas de ces limites.

Il faut donc deux tubes pour que les deux processus s'échangent des informations dans les deux sens : l'un fonctionnant dans un sens et le second dans l'autre.

Il existe deux types de tubes : les tubes anonymes et les tubes nommés. Un **tube anonyme** n'est disponible que pour le processus qui l'a créé et ses fils. De plus, il est détruit sitôt qu'il n'a plus d'utilité. Toute rupture de communication (perte du processus qui lit ou ce celui qui écrit) entraine dès lors la fermeture du tube.

Un **tube nommé** est un tube qui permet la communication entre processus qui n'ont pas de lien de parenté. L'un d'eux écrit dans le fichier, l'autre lit ce fichier. Le tube reste en place une fois la communication terminée. Pour le reste, le fonctionnement est le même que pour un tube anonyme.

**4.3.1.1.2. Les files de messages** Après avoir vu les tubes, il est temps de passer aux files de messages. On peut les voir comme un tube partagé entre plusieurs processus, chacun pouvant émettre à un ou plusieurs autres processus. De plus, chaque processus peut aussi bien lire qu'écrire dans la file de message.

Créer une file de message se fait assez simplement, souvent avec un appel système. Toutefois, il faut pouvoir identifier et sélectionner celle-ci parmi toutes les autres MSQ qui peuvent avoir été créées. Pour cela, le système d'exploitation utilise un système de clé, c'est-à-dire un nombre choisi par le programmeur ou qui est généré automatiquement. Avec cette clé on génère une MSQ qui sera identifiée par un numéro dans le système.

Une fois la MSQ en place, tout processus qui souhaite l'utiliser doit récupérer la clé de la file de messages. À partir de ce moment, les processus rattachés peuvent déposer ou récupérer un message. Pour récupérer un message, le processus se connecte à la MSQ et demande à récupérer les messages. Là encore, le processus récupère systématiquement le message le plus ancien.

Contrairement à ce qui se passe avec le passage de message, la file d'attente est persistante. Elle reste donc en place une fois créé, et doit être supprimée soit par l'un des processus soit par une action des administrateurs système. Cette caractéristique permet de déposer des messages même lorsque le processus receveur est absent (comme dans votre boite mail ou postale : vous

recevez toujours le courrier même si vous n'êtes pas chez vous). En revanche, cela peut mener à la saturation de la table. Il faudra donc toujours veiller à supprimer une file de messages lorsqu'elle n'est plus utilisée. Attention que si une MSQ est supprimée, les messages qu'elle contenait au moment de la suppression sont perdus, sans possibilité de les récupérer.

#### 4.3.1.2. Signaux

Enfin, il existe une dernière méthode de communication inter-processus : les **signaux**. Lorsqu'un processus souhaite signaler un évènement à un autre processus, il lui envoie un numéro appelé un signal. La plupart concernent des erreurs, c'est-à-dire qu'ils sont envoyés à un processus lorsque celui-ci commet une « faute », une opération interdite, mais ce n'est pas systématique. Il en existe plusieurs qui sont définis par le système d'exploitation. Le tableau d'exemple ci-dessous est celui des signaux du noyau Linux.

| Numéro (dépend de l'implémentation ou du processeur) | Nom | Signification |
|---|---|---|
| 1 | SIGHUP | Déconnexion du terminal ou mort du processus de contrôle |
| 2 | SIGINT | Interruption depuis le clavier |
| 3 | SIGQUIT | Quitter (depuis le clavier) |
| 4 | SIGILL | Instruction illégale |
| 5 | SIGTRAP | Un point d'arrêt a été trouvé |
| 6 | SIGABRT | Signal d'arrêt depuis la fonction abort() du langage C |
| 6 | SIGIOT | Synonyme de SIGABRT |
| 7 | SIGEMT | Terminaison |
| 8 | SIGFPE | Erreur mathématique en virgule flottante |
| 9 | SIGKILL | Signal KILL. Permet de terminer un processus en urgence |
| 10 | SIGBUS | Erreur de bus |
| 11 | SIGSEGV | Référence mémoire invalide |
| 11 | SIGSEGV | Référence mémoire invalide |
| 12 | SIGSYS | Mauvais argument de routine |
| 13 | SIGPIPE | Tentative d'écriture dans un tube sans lecteur à l'autre bout |
| 14 | SIGALARM | Envoyé quand un timer atteint la durée décomptée |
| 15 | SIGTERM | Terminaison normale de processus |

| | | |
|---|---|---|
| 16, 10, 30 | SIGUSR1 | Au choix de l'utilisateur |
| 16 | SIGSTKFLT | Erreur de pile sur coprocesseur |
| 16, 23, 21 | SIGURG | Un socket a le flag "urgent d'activé |
| 17, 12, 31 | SIGUSR2 | Au choix de l'utilisateur |
| 17, 20, 18 | SIGCHLD | Processus fils arrêté ou terminé |
| 17, 19, 23 | SIGSTOP | Arrêt du processus |
| 18, 20, 24 | SIGTSTP | Stop déclenché depuis un terminal |
| 18 | SIGTSTP | Idem SIGCHLD |
| 19, 18, 25 | SIGCONT | Continuer si le processus est arrêté |
| 20, 21, 26 | SIGTTIN | Lecture depuis un terminal (en arrière-plan) |
| 20, 21, 26 | SIGTTOU | Écriture depuis un terminal (en arrière-plan) |
| 23, 29, 22 | SIGIO | Entrées-sorties à nouveau possibles |
| - | SIGPOLL | Idem SIGIO |
| 24, 24, 30 | SIGXCPU | Limite de temps CPU dépassée |
| 25, 31 | SIGFSZ | Taille de fichier excessive |
| 26, 28 | SIGVTALRM | Alarme virtuelle |
| 27, 29 | SIGPROP | Profile Alarm Clock |
| 28, 20 | SIGWINCH | Fenêtre redimensionnée |
| 29, 30, 19 | SIGPWR | Chute d'alimentation |
| 29 | SIGINFO | Idem SIGPWR |
| - | SIGLOST | Perte de verrou de fichier |
| 31 | SIGUNUSED | Signal inutilisé |

Un signal n'est pas toujours immédiatement transmis au processus concerné. En effet, celui-ci peut très bien être occupé et ne pas pouvoir le traiter directement. Il existe donc un système pour placer en attente ces signaux et les mettre à disposition du processus. Cette mise en attente prend la forme de drapeaux (*flags*), un par signal reçu. Si un signal a été reçu, on lève le drapeau correspondant à ce signal précis, sinon le drapeau reste baissé. Un signal qui est ainsi en attente et qui n'a pas encore été transmis est appelé un signal *pendant*.

Lorsque le processus est à nouveau disponible, il reçoit les signaux qui étaient en attente (on dit qu'ils sont *délivrés* au processus concerné). Cette transmission a lieu lors d'un retour du processus dans le mode utilisateur. À partir de ce moment, le processus peut :

— ignorer le signal (même si certains signaux **ne peuvent pas** être ignorés, comme SIGKILL et SIGCHLD) ;
— traiter le signal avec l'action définie par le système ;
— traiter le signal de façon spécifique.

Le programmeur du logiciel a la possibilité de traiter les signaux comme il l'entend, que ce soit pour ignorer le signal ou lancer une action spécifique. Dans le second cas, le programmeur peut prévoir une fonction pour gérer le signal reçu appelée un ***handler*** (gestionnaire) de signal.

Dans le cas où le programmeur n'a pas prévu d'action particulière pour un ou plusieurs signaux, le système d'exploitation a prévu une **action par défaut**, les plus courantes étant les suivantes :

— ignorer le signal ;
— la fermeture (abandon) du processus ;
— la fermeture du processus et l'écriture des causes dans un fichier spécifique appelé fichier *core* ;
— s'arrêter (se mettre en pause) ;
— reprendre (fin de la pause, au boulot ! ) .

### 4.3.2. Sans isolation des processus

Par convention, on qualifie de processus tout programme pour lequel il y a isolation des processus, les autres formes de programmes pouvant exister. Avec cette définition, un processus est simplement un fichier exécutable chargé en mémoire et isolé des autres processus/programmes. Mais cette définition ne dit pas que ce processus correspond à un seul programme qui s'exécute d'un seul bloc. Il est en effet possible de rassembler plusieurs programmes dans un seul processus : ces sous-programmes sont appelés des ***threads***.

Les *threads* peuvent s'exécuter sur le processeur tout comme les processus : on peut ordonnancer des *threads* ou les exécuter en parallèle sur des processeurs séparés. Chaque *thread* possède son propre code à exécuter et sa propre pile d'appel. Cela facilite beaucoup la programmation d'applications qui utilisent plusieurs processeurs. Cependant, les *threads* doivent partager leur zone de mémoire avec les autres *threads* du processus : ils peuvent ainsi partager des données sans devoir passer par de lourds systèmes de communication inter-processus.


FIGURE 4.3. – Comparaison entre un et plusieurs threads

Le changement de contexte entre deux *threads* est beaucoup plus rapide que pour les processus. En effet, le fait que les *threads* se partagent la même mémoire évite certaines manipulations, obligatoires avec les processus. Par exemple, on n'est pas obligé de vider le contenu des mémoires caches sur certains processeurs. Généralement, les *threads* sont gérés en utilisant un ordonnancement préemptif, mais certains *threads* utilisent l'ordonnancement collaboratif (on parle alors de ***fibers***).

#### 4.3.2.1. Threads utilisateurs

Les threads utilisateurs sont des *threads* qui ne sont pas liés au système d'exploitation. Ceux-ci sont gérés à l'intérieur d'un processus, par une bibliothèque logicielle. Celle-ci s'occupe de la création et la suppression des *threads*, ainsi que de leur ordonnancement. Le système d'exploitation ne peut pas les ordonnancer et n'a donc pas besoin de mémoriser les informations des *threads*. Par contre, chaque *thread* doit se partager le temps alloué au processus lors de l'ordonnancement : c'est dans un quantum de temps que ces *threads* peuvent s'exécuter.


FIGURE 4.4. – Image adaptée d'une image de Silberschatz, Abraham provenant de wikicommons

#### 4.3.2.2. Threads noyaux

Les *threads* noyaux sont gérés par le système d'exploitation, qui peut les créer, les détruire ou les ordonnancer. L'ordonnancement est donc plus efficace, vu que chaque *thread* est ordonnancé tel quel. Il est donc nécessaire de disposer d'une table des *threads* pour mémoriser les contextes d'exécution et les informations de chaque *thread*.


FIGURE 4.5. – Image adaptée d'une image de Silberschatz, Abraham provenant de wikicommons

---

Je tiens à remercier particulièrement Lucas-84, qui a participé au premier jet de ce tutoriel, pour son travail. Je vous invite d'ailleurs à aller lire son tutoriel sur la programmation système en C sous Unix , qui fera un bon complément à ce tutoriel théorique.

# Liste des abréviations

**FAT** File Allocation Table. 13