Les systèmes d'exploitation

Une version à jour et éditable de ce livre est disponible sur Wikilivres, une bibliothèque de livres pédagogiques, à l'URL : http://fr.wikibooks.org/wiki/Les_syst%C3%A8mes_d%27exploitation

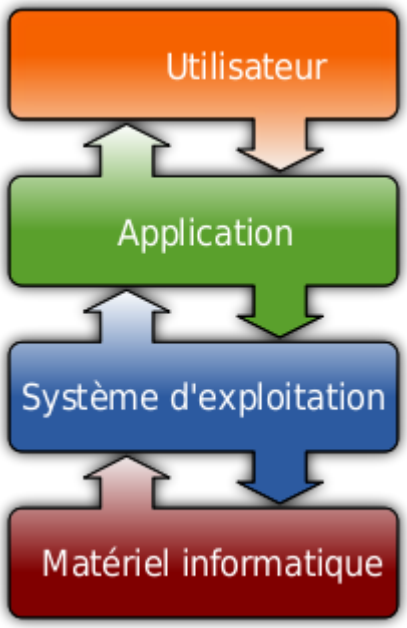
---

Le noyau d'un système d'exploitation

Au tout début de l'informatique, les logiciels n'étaient pas compatibles sur une large gamme de matériel. Avec le temps, les informaticiens ont inventé des techniques d'abstraction matérielle qui permettent à un programme de s'exécuter sur des ordinateurs avec des matériels très différents. Il ont pour cela fabriqué . Rendre les programmes indépendants du matériel a demandé de revoir l'organisation des logiciels, qui ont dû être découpés en plusieurs programmes séparés :

- les **programmes systèmes** gèrent la mémoire et les périphériques ;
- les **programmes applicatifs** ou applications délèguent la gestion de la mémoire et des périphériques aux programmes systèmes.

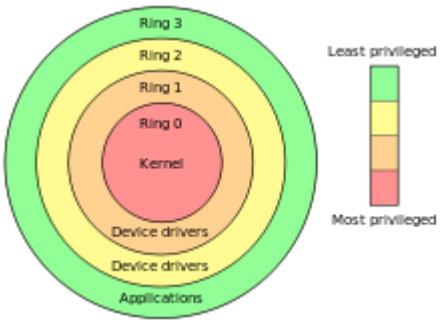
Pour simplifier, l'ensemble des programmes système porte le nom de **système d'exploitation**.



Séparation entre programmes systèmes et applicatifs.

Anneaux mémoire

L'O.S doit garantir que seuls les programmes systèmes ont accès aux périphériques ou aux registres de gestion de la mémoire virtuelle. Pour cela, les processeurs actuels incorporent une technique : les **anneaux mémoires**. Dans les grandes lignes, le processeur gère deux niveaux de privilèges : un mode noyau pour les programmes systèmes, où les instructions d'accès aux périphériques peuvent s'exécuter, et un mode utilisateur pour les applications, où ces opérations sont interdites.



Niveaux de privilèges sur les processeurs x86.

Ces anneaux mémoire/niveaux de privilèges, sont gérés en partie par le processeur. Celui-ci contient un registre qui précise s'il est en espace noyau ou en espace utilisateur. A chaque accès mémoire ou exécution d'instruction, le processeur vérifie si le niveau de privilège permet l'opération demandée. Lorsqu'un programme effectue une instruction interdite en mode utilisateur, une exception matérielle est levée. Généralement, le programme est arrêté sauvagement et un message d'erreur est affiché. Un programme ne peut changer d'anneau mémoire au cours de son exécution, sous certaines conditions relativement drastiques, afin d'éviter que tout programme s'arroge des droits d'accès aux périphérique sans restrictions.

Sur certains processeurs, on trouve des niveaux de privilèges intermédiaires entre l'espace noyau et l'espace utilisateur. Les processeurs de nos PC actuels contiennent 4 niveaux de privilèges. Le système Honeywell 6180 en possédait 8. A l'origine, ceux-ci ont étés inventés pour faciliter la programmation des

pilotes de périphériques. Certains d'entre eux peuvent en effet gagner à avoir des niveaux de privilèges intermédiaires entre celui d'une simple application et celui d'un O.S. Mais force est de constater que ceux-ci ne sont pas vraiment utilisés, seuls les espaces noyau et utilisateur étant pertinents. Il en est de même pour beaucoup de méthodes de protection mémoire. Une des raisons à cet état de fait est tout simplement la compatibilité entre architectures matérielles différentes. Par exemple, les premières versions de Windows NT n'utilisaient que deux anneaux de privilèges sur les processeurs x86, en partie parce que d'autres jeu d'instructions supportés par Windows n'avaient que deux niveaux de privilèges. Ce n'est pas la seule raison, la facilité de programmation de l'O.S devant aussi être prise en compte.

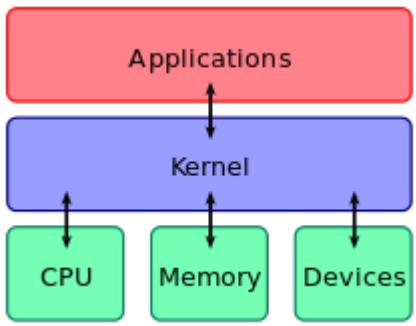
Appels systèmes

Tous les programmes systèmes sont des routines d'interruptions, fournies par l'OS ou les pilotes, qui permettent d'exploiter les périphériques. Sur les PC actuels, où le BIOS fournit des routines de base, l'O.S doit modifier le vecteur d'interruption avec les adresses de ses routines. On dit qu'il détourne détourner l'interruption. Les applications peuvent appeler à la demande ces routines via une interruption logicielle : ils effectuent ce qu'on appelle un **appel système**. Tout O.S fournit un ensemble d'appels systèmes de base, qui servent à manipuler la mémoire, gérer des fichiers, etc. Par exemple, linux fournit les appels systèmes open, read, write et close pour manipuler des fichiers, les appels brk, sbrk, pour allouer et désallouer de la mémoire, etc. Évidemment, ceux-ci ne sont pas les seules : linux fournit environ 380 appels systèmes distincts. Ceux-ci sont souvent encapsulé dans des librairies et peuvent s'utiliser comme de simples fonctions.

L'appel système se charge automatiquement de switcher le processeur dans l'espace noyau. Cette commutation n'est cependant pas gratuite, de même que le l'interruption qui lui est associée. Ainsi, les appels systèmes sont généralement considérés comme lents, très lents. Divers processeurs incorporent des techniques pour rendre ces commutations plus rapides, via des instructions spécialisées (SYSCALL/SYSRET et SYSENTER/SYSEXIT d'AMD et Intel), ou d'autres techniques (call gate de Intel, Supervisor Call instruction des IBM 360, etc.

Le noyau du système d'exploitation

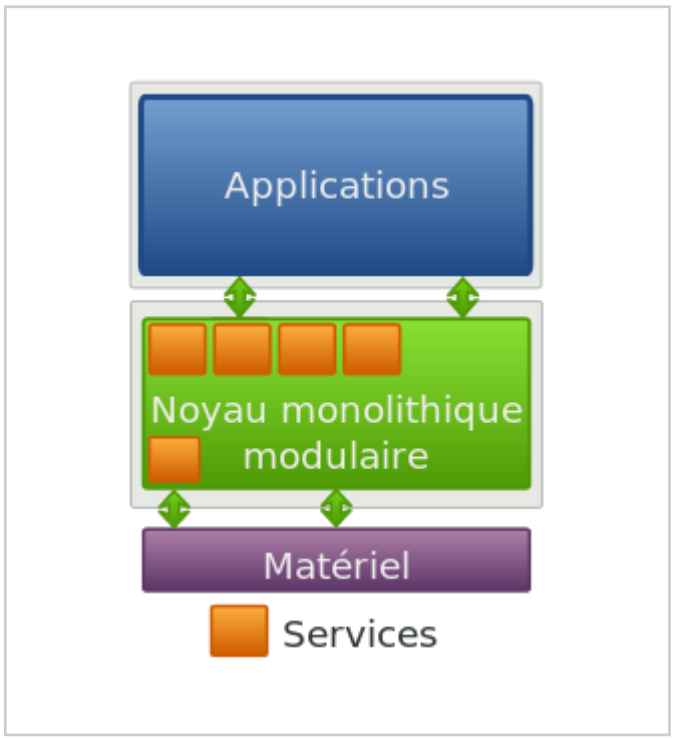
La portion du système d'exploitation placée dans l'espace noyau est ce qu'on appelle le **noyau du système d'exploitation**, le reste de l'O.S étant composé d'applications qui servent à afficher une interface graphique ou une ligne de commande, par exemple. Reste que beaucoup de programmes de l'O.S peuvent être placés indifféremment dans le noyau ou dans l'espace utilisateur. Mais cela a un cout : exécuter un appel système est très lent, changer de niveau de privilège étant assez couteux. Conserver de bonnes performances impose de diminuer la quantité d'appels systèmes, et donc de placer un maximum de choses en espace noyau. Mais cela se fait au prix de la sécurité, toute erreur de programmation dans le noyau entrainant un écran bleu. On peut ainsi classer les noyaux en plusieurs types, selon l'accent mit sur les performances ou la sécurité.



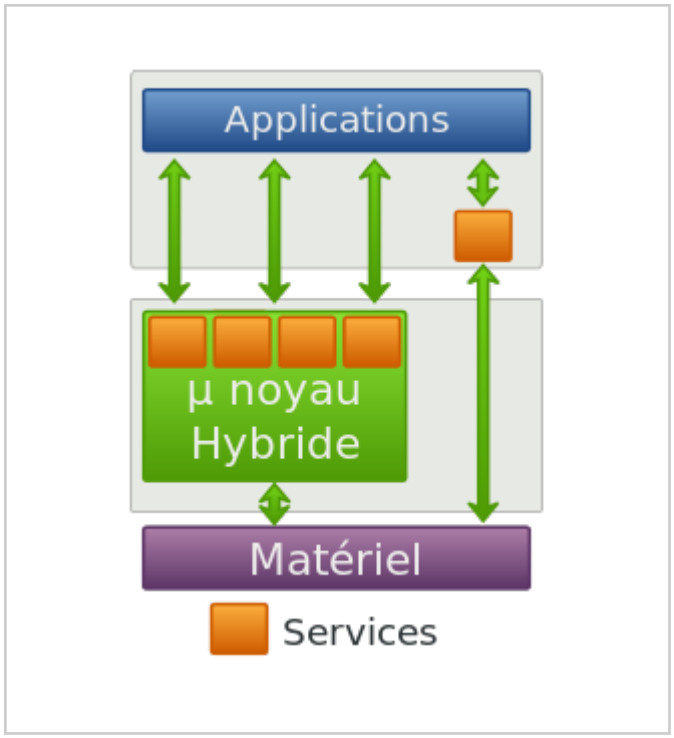
Séparation entre noyau et applications.

- Les **noyaux monolithiques** placent un maximum de programmes systèmes dans l'espace noyau. Leurs performances sont donc excellentes, vu que les appels systèmes à faire sont peu nombreux. Par contre, leur fiabilité est plutôt faible, la majorité des erreurs de programmation de l'O.S se trouvant dans le noyau, source de beaucoup d'écrans bleus.
- Les **micro-noyaux** préfèrent au contraire placer le moins de choses dans l'espace utilisateur. Leur sécurité est excellente, la majorité des erreurs de programmation n'entrainant pas un crash, mais simplement l'affichage d'un message d'erreur. Mais le nombre d'appels système est nettement plus important que pour les noyaux monolithiques, ce qui source de performances relativement faibles.
- Les **noyaux hybrides**, sont un intermédiaire entre les deux précédents.
- Certaines versions relativement extrêmes de noyaux monolithiques, où tout l'O.S est placé dans l'espace noyau, portent le nom de **noyaux mégalithiques**. De même, on trouve des versions extrêmes de micro-noyaux, où tout l'O.S est composé de bibliothèques logicielles exécutées en espace utilisateur, à l'exception du minimum vital. On parle alors d'**exokernel** ou de **nanokernel**.

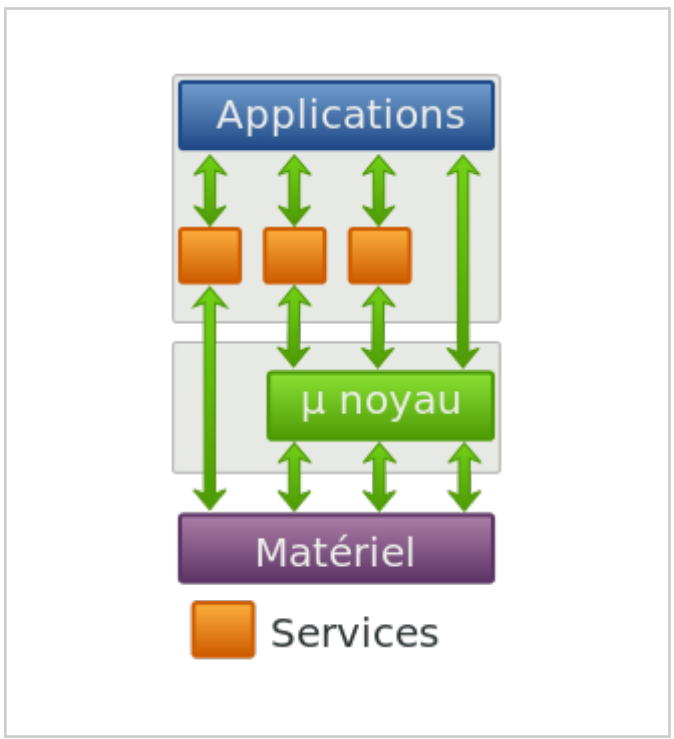
Pour information, les systèmes Unix et Linux sont basés sur des noyaux monolithiques, tandis que les systèmes Windows utilisent des micro-noyaux.



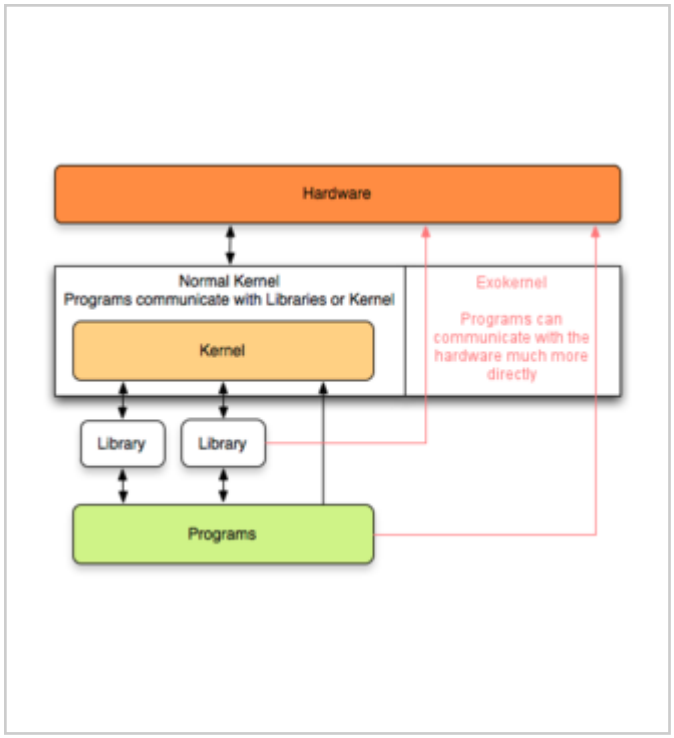
Noyau monolithique modulaire



Micro noyau hybride simplifié



Micro noyau simplifié


Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Exokernel
Programs can communicate with the hardware much more directly
Library
Library
Programs

Exonoyau.

La multiprogrammation

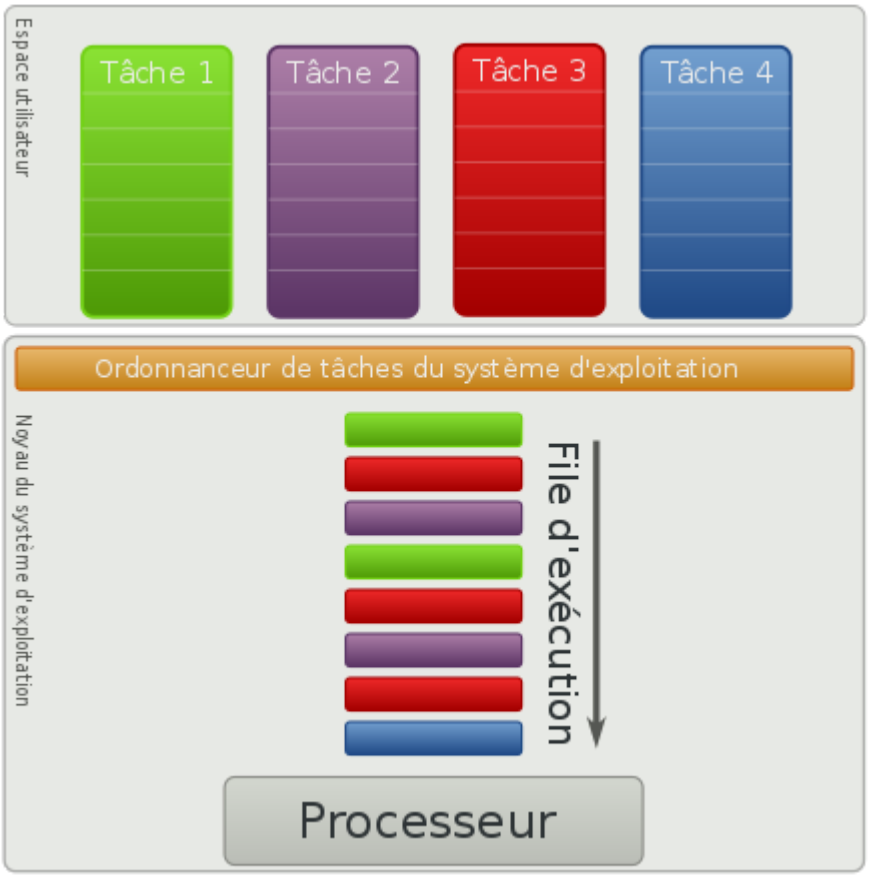
Les tout premiers OS datent des années 1950, et ceux-ci ont commencé à devenir de plus en plus utilisés à partir des années 60-70. Évidemment, ces OS étaient relativement simples. Notamment, ceux-ci ne pouvaient pas exécuter plusieurs programmes en même temps. Il était ainsi impossible de démarrer à la fois un navigateur internet, tout en écoutant de la musique. Dit autrement, ces systèmes d'exploitation ne permettaient de ne démarrer qu'un seul programme à la fois. On appelait ces OS des OS **mono-programmés**.

Mais cette solution avait un problème. Lors de l'accès à un périphérique, le processeur doit attendre que le transfert avec le périphérique aie cessé et est inutilisé durant tout ce temps. Pour certains programmes qui accèdent beaucoup aux périphériques, il est possible que le processeur ne soit utilisé que 20% du temps, voire moins. Une solution à cela est d'exécuter un autre programme pendant que le programme principal accède aux périphériques. Un OS qui permet cela est un OS **multiprogrammé**. Le nombre de programme pouvant être chargés en mémoire simultanément est appelé le taux de multiprogrammation. Plus celui-ci est élevé, plus l'OS utilisera le processeur à ses capacités maximales.

Les OS actuels vont plus loin et permettent d'exécuter plusieurs programmes "simultanément", même si aucun programme n'accède aux périphériques. Les OS de ce genre sont des **OS multi-tâche**. L'apparition de la multiprogrammation a posé de nombreux problèmes, notamment au niveau du partage de la mémoire et du processeur. Dans ce qui va suivre, nous allons voir comment ces OS gèrent la mémoire et les commutations entre programmes (appelés processus sur ces OS). Pour manipuler plusieurs processus, l'OS doit mémoriser des informations sur eux. Ces informations sont stockées dans ce qu'on appelle un Process Control Block, une portion de la mémoire.

Allocation du processeur

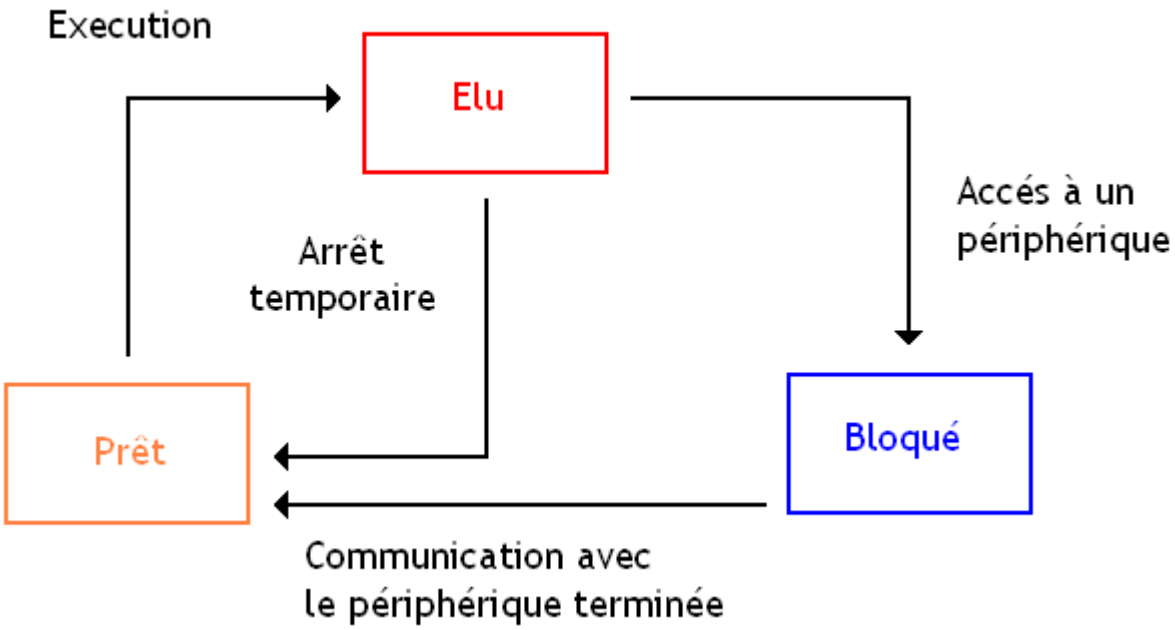
Le système d'exploitation utilise une technique très simple pour permettre la multiprogrammation sur les ordinateurs à un seul processeur : l'ordonnancement. Cela consiste à constamment switcher entre les différents programmes qui doivent être exécutés : en switchant assez vite, on peut donner l'illusion que plusieurs processus s'exécutent en même temps.



Ordonnanceur noyau os

Avec cette technique, chaque 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).



Les processus en état prêt sont placés dans une **file d'attente**, le nombre de programme dans cette liste a une limite fixée par le système d'exploitation. Lorsque l'O.S décide de switcher de processus, il doit choisir un processus dans la file d'attente et le lancer sur le processeur. Pour comprendre comment faire, il faut se souvenir qu'un processus manipule un ensemble de registres processeur, aussi appelé **contexte d'exécution du processus**. Pour qu'un processus puisse reprendre où il en était, il faut que son contexte d'exécution redevienne ce qu'il était. Pour cela, ce contexte d'exécution est sauvegardé quand il est interrompu et restauré lors de l'ordonnancement. On appelle cela une **commutation de contexte**. Avec cette méthode de sauvegarde du contexte, le système d'exploitation doit fatalement mémoriser tous les contextes des processus dans une liste appelée **table des processus**, elle-même très liée à la file d'attente.

Reste que pour utiliser notre processeur au mieux, il faut faire en sorte que tous les processus aient leur part du gâteau. C'est ce qu'on appelle l'**ordonnancement**. 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, par l'intermédiaire d'un appel système. Dans le second, c'est le système d'exploitation qui stoppe l'exécution d'un processus. L'avantage du dernier cas est que les processus ne peuvent plus monopoliser le processeur.

Ordonnancement collaboratif

Tout d'abord, on va commencer par un systéme d'exploitation qui exécute des programmes qui s'exécute les uns après les autres. Ces programmes sont exécutés 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.

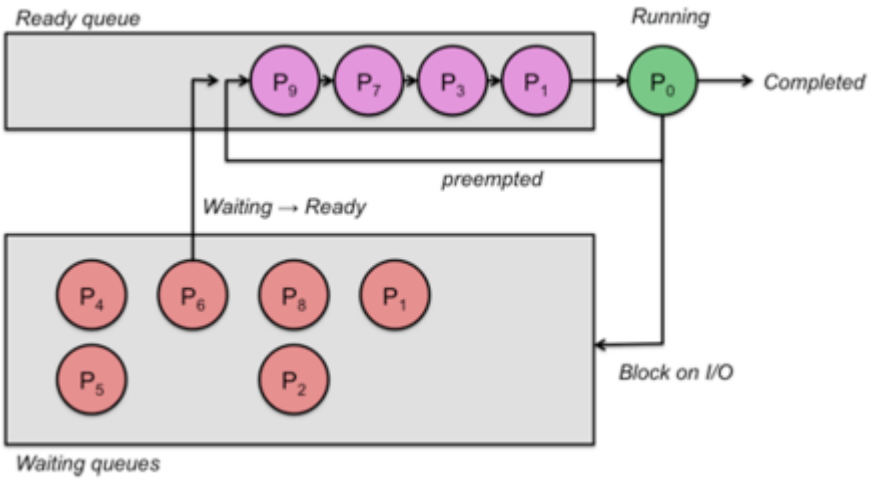
- Avec l'algorithme First Input First Output, les programme à exécuter sont ajoutés dans la file d'attente quand on les démarre. Ceux-ci sont alors stockés dans la file d'attente dans l'ordre dans lesquels on les a fait démarrer. L'ordonnanceur décide alors d'exécuter le programme entré dans la file d'attente avant tous les autres en premier. En clair, les programmes sont exécutés dans l'ordre dans lequel ils sont rentrés dans la file d'attente.
- Avec l'algorithme Last Input First Output, les programmes à exécuter sont ajoutés dans la file d'attente quand on les démarre. Ceux-ci sont alors stockés dans la file d'attente dans l'ordre dans lesquels on les a fait démarrer. Mais contrairement à l'algorithme First Input First Output, l'ordonnanceur exécuter le programme entré en dernier dans la file d'attente, et non le premier. Si vous ajoutez un programme dans une file d'attente contenant déjà des programmes en attente, ce sera lui le prochain à être exécuté, vu qu'il a été ajouté en dernier. Cet algorithme peu 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 exécutés en premier. Ceux-ci doivent alors attendre la fin de l'exécution de tous les programmes rentrés avant eux.
- L'algorithme Shortest Job First est basé sur une logique simple. Les programmes à exécuter sont placés dans la file d'attente et l'ordonnanceur va alors décider d'exécuter ces processus dans l'ordre. Pour appliquer cet algorithme, 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 mit 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 fil

d'attente et on recommence.

Ordonnancement préemptif

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. Voyons maintenant les algorithmes préemptifs qui permettent de choisir quel programme faire passer de l'état prêt à l'état élu. Pour faire simple, nous allons en voir quelque uns, relativement importants et très proches de ceux utilisés dans les OS actuels.

- L'algorithme Shortest Remaining Time Next est une variante préemptive de l'algorithme Shortest Job First vu au dessus. Dans cette version, si un programme est ajouté dans la file d'attente, on regarde le temps que ce nouveau venu mettrait à s'exécuter, et on compare avec le temps qu'il reste au processus en cours d'exécution avant que celui-ci finisse son travail. Si le temps mit par le nouveau programme est plus faible que le temps d'exécution du programme en train de s'exécuter, on change et on exécute le nouveau venu à la place.
- L'**algorithme du tourniquet** exécute chaque programme l'un après l'autre, dans l'ordre. 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.



Algorithme du tourniquet (Round-robin)

- L'**algorithme des files d'attentes multiple-niveau** donne la priorité aux programmes rapides ou qui accèdent souvent aux périphériques. Cet algorithme utilise plusieurs files d'attentes, auxquelles ont donne des quantum de temps différents. Tout programme démarre dans la file d'attente avec le plus petit quantum de temps, et change de file d'attente suivant son utilisation du quantum. Si un programme utilise la totalité de son quantum de temps, c'est le signe qu'il a besoin d'un quantum plus gros et doit donc changer de file. La seule façon pour lui de remonter d'un niveau est de ne plus utiliser complètement son quantum de temps. Ainsi, un programme va se stabiliser dans la file dont le quantum de temps est optimal (ou presque).
- 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...

Création et destruction 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.

Si les processus peuvent être créés, il est aussi possible de les détruire quand ils ont terminé leur travail ou s'ils commettent une erreur/faute lors de leur exécution. 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. Si un processus père est terminé avant son fils, 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 ».

Communication inter-processus

Comme on l'a vu plus haut, les processus sont chargés 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. Faire communiquer des processus entre eux demande d'utiliser des mécanismes de communication inter-processus. La méthode la plus simple consiste à partager un bout de mémoire entre processus, mais il existe d'autres méthodes que nous allons aborder.

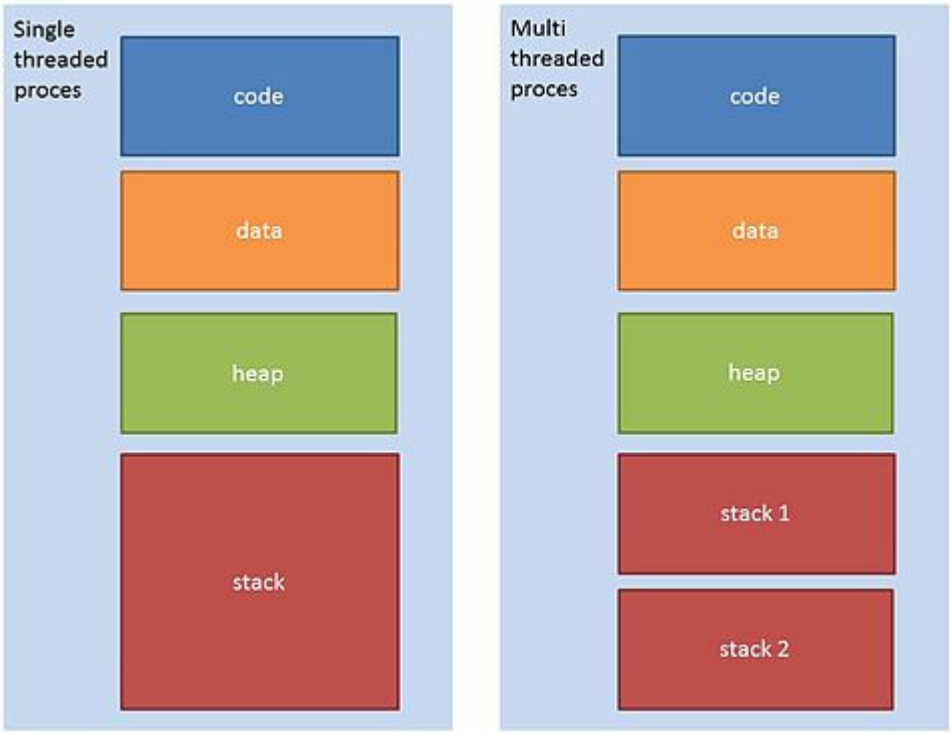
Avec isolation des processus

La première méthode est appelée l'**échange de messages**. Il permet aux processus de s'échanger des **messages**, des blocs de données de taille variable ou fixe selon l'implémentation. Mais le contenu n'est pas standardisé, et chaque processus peut mettre ce qu'il veut dans un message. Le tout est que le processus qui reçoit le message sache l'interpréter. Pour cela, le format des données, le type du message, ainsi que la taille totale du message, est souvent envoyé en même temps que le message.

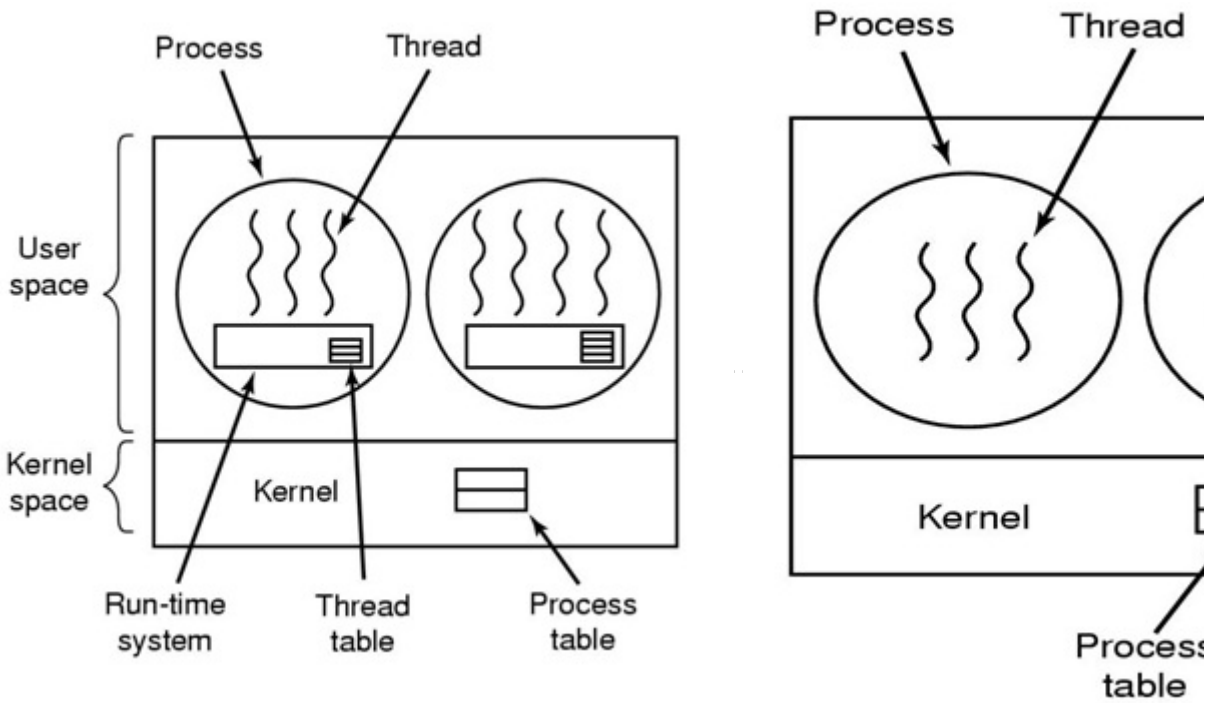
Dans le cas le plus simple, il n'y a pas de mise en attente des messages dans un espace de stockage partagé entre processus. On parle alors de **passage de message**. Cela fonctionne bien quand les deux processus ne sont pas sur le même ordinateur et communiquent entre eux via un réseau local ou par Internet. Mais certains O.S permettent de mettre en attente les messages envoyés d'un processus à un autre, le processus récepteur pouvant consulter les message en attente quand celui-ci est disponible. Les messages sont alors mis en attente dans diverses structures de mémoire partagée. Celles-ci sont appelées, suivant leur fonctionnement, des **tubes** ou des **files de messages**. Ces structures stockent les messages dans l'ordre dans lequel ils ont étés envoyés (fonctionnement en file ou FIFO). La différence entre les deux est que les tubes sont partagés entre deux processus, un pouvant lire et un autre écrire, alors qu'une file de message peut être partagée entre plusieurs processus qui peuvent aussi bien lire qu'écrire.

Sans isolation des processus

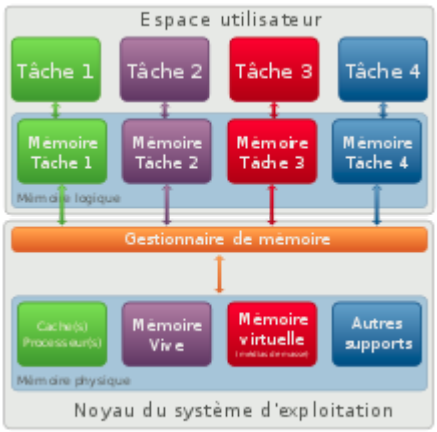
Il est possible de rassembler plusieurs programmes dans un seul processus. Ces programmes n'étant pas isolés, ils se partagent le même morceau de mémoire et peuvent ainsi communiquer entre eux via une mémoire partagée. Les programmes portent alors le nom de **threads**. Ces threads doivent aussi être ordonnancés. On fait ainsi la distinction entre threads proprement dits, gérés par un ordonnancement préemptif, et les fibers, gérés par un ordonnancement collaboratif. Le changement de contexte entre deux threads est beaucoup plus rapide que pour les processus, vu que cela évite de devoir faire 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**). Les threads et fibers partagent le même espace d'adressage. Ils partagent généralement certains segments : ils se partagent le code, le tas et les données statiques. Par contre, chaque thread dispose de son propre pile d'appel.



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. 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.



La gestion de la mémoire

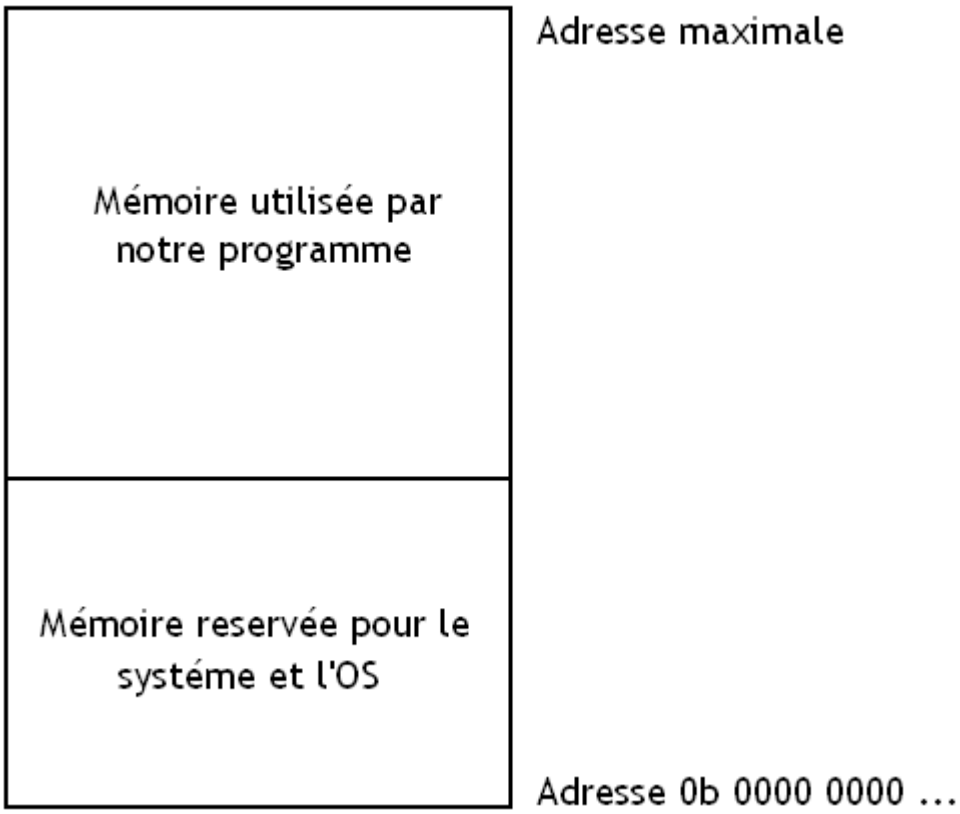


Gestionnaire de mémoire.

Dans le chapitre précédent, on a vu comment plusieurs programmes peuvent se partager le processeur. Mais ceux-ci doivent aussi se partager la mémoire physique : chaque programme doit avoir sa propre mémoire, sans marcher sur la mémoire des autres. Si un programme pouvait modifier les données d'un autre programme, on se retrouverait rapidement avec une situation non prévue par le programmeur, avec des conséquences allant de comiques à catastrophiques. Il faut donc introduire des mécanismes de partage de la mémoire, ce qui implique plusieurs chose : chaque programme doit avoir son ou ses propres blocs de mémoire, et il doit être le seul à pouvoir y accéder. Nous avons vu que ces problèmes peuvent être réglés directement au niveau du matériel par des techniques de mémoire virtuelle. Mais le système d'exploitation a aussi un grand rôle à jouer sur les architectures sans mémoire virtuelle. Dans ce chapitre, nous allons voir comment le système d'exploitation fait pour gérer le partage de la mémoire physique entre programmes, quand la mémoire virtuelle n'est pas utilisée.

Monoprogrammation

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.

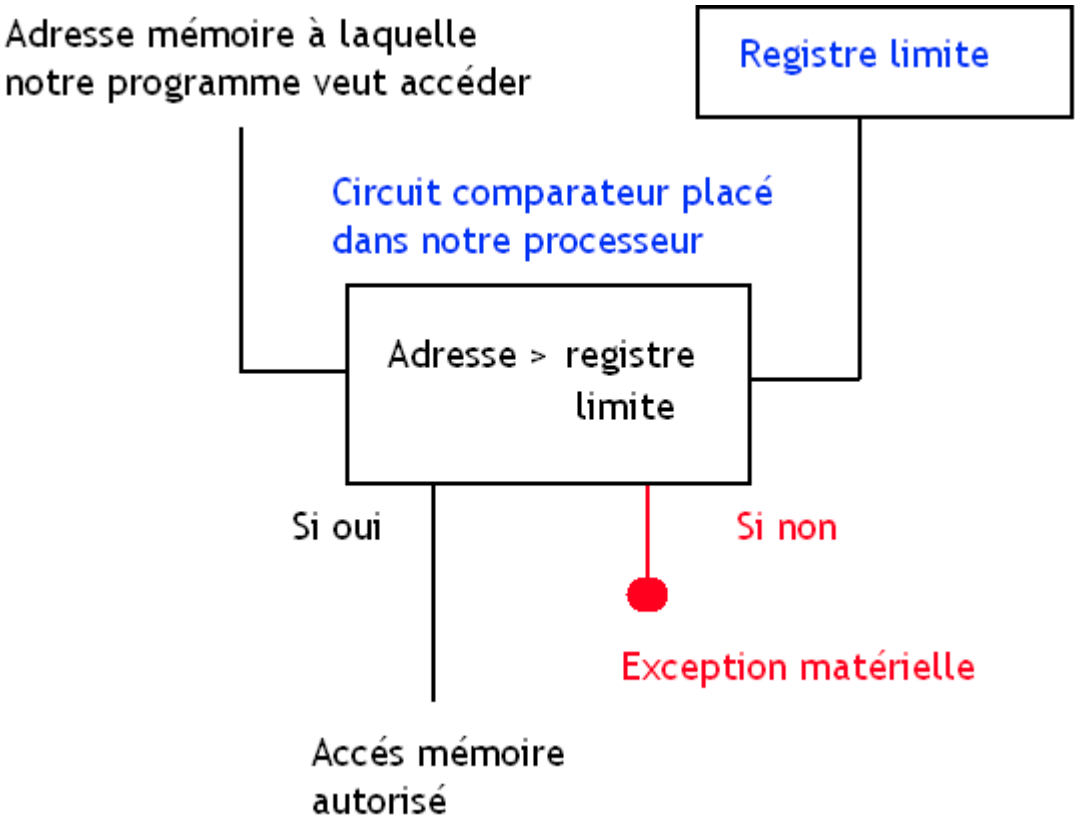


Chargement d'un programme

Le démarrage d'un programme est le rôle d'un morceau du système d'exploitation appelé le **chargeur de programme** (loader en anglais). Celui-ci rapatrie le fichier exécutable en RAM et l'exécute avec un branchement vers la première instruction du programme démarré. 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).

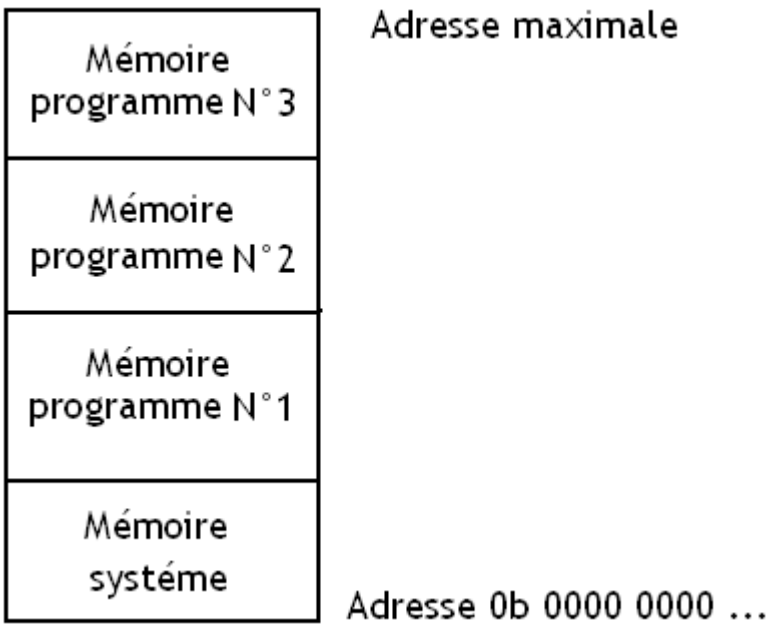
Protection mémoire

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.



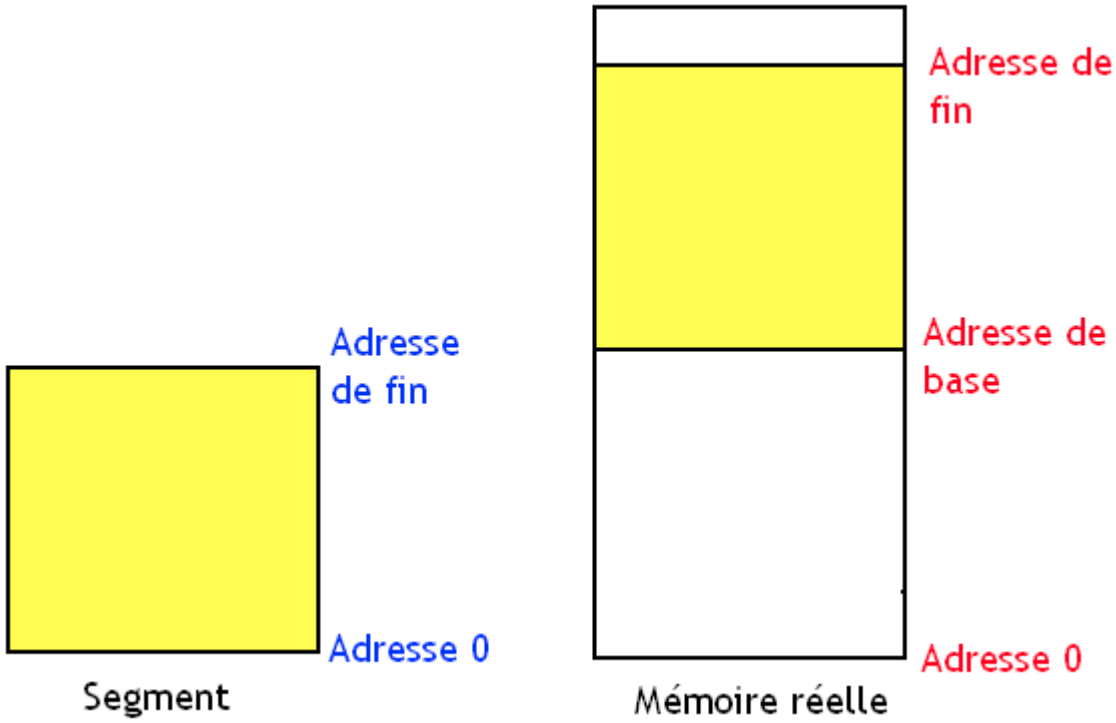
Multiprogrammation

Sur les vieux systèmes d'exploitation, chaque programme recevait un bloc de RAM pour lui tout seul, une **partition mémoire**. Ces partitions n'ont pas une taille fixe, ce qui fait qu'un processus peut demander du rab ou libérer la mémoire qui lui est inutile, l'O.S fournissant des appels système pour.



Relocalisation

Chaque partition/segment peut être placé n'importe où en mémoire physique par l'OS, ce qui fait qu'il n'a pas d'adresse fixée en mémoire physique et peut être déplacé comme bon nous semble. C'est le système d'exploitation qui gère le placement des partitions/segments dans la mémoire physique. Ainsi, les adresses de destination des branchements et les adresses des données ne sont jamais les mêmes. Or, chaque accès à une donnée ou instruction demande de connaitre son adresse, qui varie à chaque exécution. 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 **relocalisation**. Pour résoudre ce problème, le compilateur considère que le programme commence à l'adresse zéro et laisse l'OS corriger les adresses du programme. Cette correction d'adresse demande simplement d'ajouter un décalage, égal à la première adresse de la partition mémoire. Cette correction peut se faire de deux manière : soit l'OS corrige chaque adresse lors du lancement du programme, soit le processeur s'en charge à chaque accès mémoire.



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. Pour cela, cette première adresse est mémorisée dans un registre de base, mis à jour automatiquement lors de chaque changement de programme. 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.

Protection mémoire

L'utilisation de partitions mémoire entraine l'apparition de plusieurs problèmes. La première est qu'un programme ne doit avoir accès qu'à la partition qui lui est dédiée et pas aux autres, sauf dans quelques

rares exceptions. Toute tentative d'accès à une autre partition doit déclencher une exception matérielle, qui entraine souvent l'apparition d'un message d'erreur. Tout cela est pris en charge par le système d'exploitation, par l'intermédiaire de mécanismes de **protection mémoire**, que nous allons maintenant aborder.

Registres de base et limite

La première solution permet au processeur (ou l'OS) de détecter les accès hors-partition, à savoir un programme qui lit/écrit de la mémoire au-delà de la partition qui lui réservée. Pour cela, le processeur utilise les registre limite et de base de chaque partition. Une implémentation naïve de la protection mémoire consiste à vérifier pour chaque accès mémoire ne déborde pas au-delà de la partition qui lui est allouée, ce qui n'arrive que si l'adresse d'accès dépasse la valeur du registre limite.

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.

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.

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 en utilisant 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.

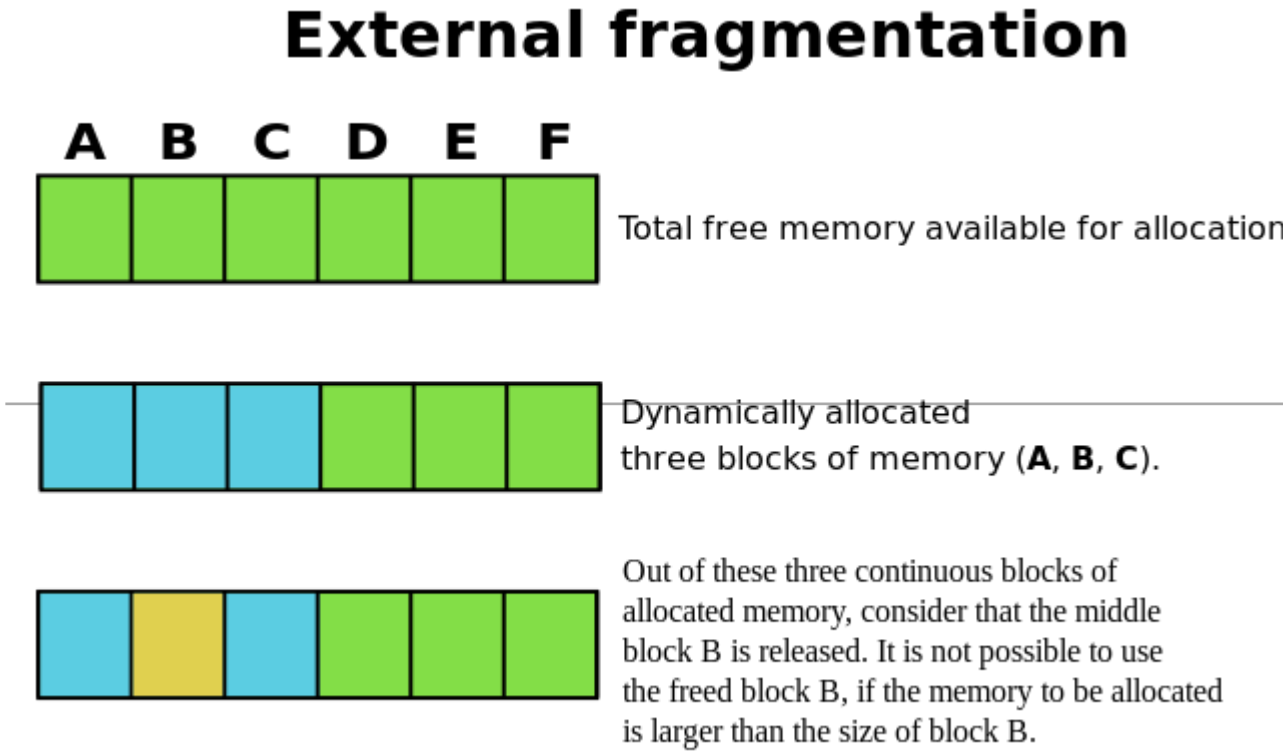
Liste des partitions libres

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.

Choix des partitions libres

A partir de ces informations, l'OS doit trouver une partition vide suffisamment grande pour accueillir le programme démarré. Si le programme n'utilise pas tout le segment, on découpe la partition de manière à créer une partition pour la zone non-occupée par le programme lancé. 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. Mais si on choisit mal la partition, de multiples allocations entrainent un phénomène de **fragmentation externe** : on dispose de suffisamment de mémoire libre, mais celle-ci est dispersée dans beaucoup de segments qui sont trop petits. Si cela arrive, l'O.S doit compacter les partitions dans la mémoire RAM, ce qui prend beaucoup de temps inutilement.



Algorithme de la première/dernière 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.

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.

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.

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 consécutives, adjacentes. Il arrive alors parfois qu'il n'y aie 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.

Les systèmes de fichiers

Le **système de fichiers** est la portion du système d'exploitation qui s'occupe de la gestion des mémoires de masse. Il prend en charge le stockage des fichiers sur le disque dur, le rangement de ceux-ci dans des répertoires, l'ouverture ou la fermeture de fichiers/répertoires, et bien d'autres choses encore. La gestion des partitions est aussi assez liée au système d'exploitation.

Partitions

Avant d'installer un système d'exploitation sur un disque dur, celui-ci doit être partitionné, du moins sur les PC actuels. Partitionner un disque dur signifie le diviser en plusieurs morceaux, qui seront considérés par le système d'exploitation comme autant de disques durs séparés. Les informations sur les partitions sont mémorisées dans une **table des partitions**, qui stocke la taille et le premier secteur de chaque partition, ainsi que quelques informations complémentaires.

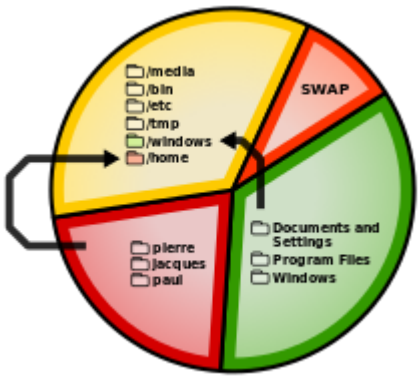


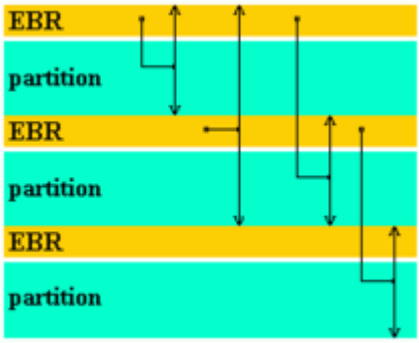
Illustration du partitionnement d'un disque dur

Master Boot Record

Sur les PC actuels, on a vu que cette table est stockée dans le Master Boot Record. Celui-ci peut contenir 4 partitions. Les informations de chaque partition sont codées sur 16 octets, de la manière indiquée ci-dessous. La partition correspond évidemment à un bloc de secteurs consécutifs. Vu que la taille d'une partition est codée sur 32 bits, la taille d'une partition ne peut pas dépasser les 4 Giga-secteurs. Sur les mémoires de masse qui ont des secteurs de 512 Kio, cela correspond à une capacité totale de 2,2 Téraoctets.

Flag qui indique si la partition est bootable ou non	Adresse CHS du début de la partition	Type de la partition	Adresse CHS de la fin de partition	Adresse LBA du début de la partition	Taille de la partition en nombre de secteurs
1 octets	3 octets	1 octets	3 octets	4 octets	4 octets

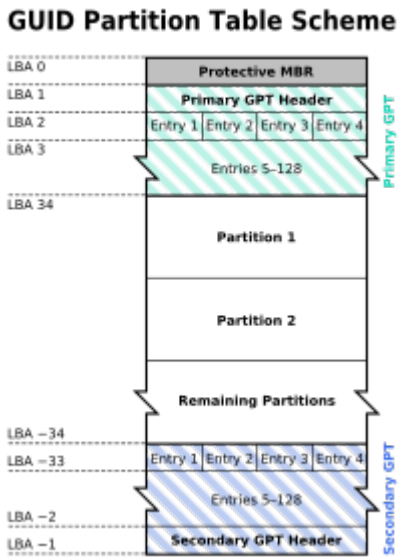
Afin de dépasser la limite de 4 partitions, il est possible de subdiviser une partition en sous-partitions, appelées **partitions logiques**. Pour cela, la partition subdivisée doit être définie comme une partition dite étendue dans le MBR (le type de la partition doit avoir une valeur bien précise). Les informations sur une partition logique sont mémorisées dans un secteur situé au début de la partition logique, dans une structure appelée **Extended Boot Record**. Celui-ci a une structure similaire au MBR, à quelques détails près. Premièrement, seules les deux premières entrées de la table des partitions sont censées être utilisées. La première indique l'adresse de la prochaine partition logique (ce qui fait que les EBR forment une liste chainée), tandis que la seconde sert pour les informations de la partition logique proprement dite. De plus, la place normalement réservée au chargement de système d'exploitation est remplie de zéros.



Extended Boot Record.

Tables GUID

Afin de faire face aux limitations de ce système, le MBR est progressivement remplacé par des **table de partitionnement GUID**. Celle-ci permet de gérer des partitions de plus de 2,2 Téra-octets, sans problèmes. Avec elle, la table des partitions peut contenir 32 partitions différentes. La table des partitions contient donc 32 entrées, une pour stocker les informations sur chaque partition. Ces entrées sont précédées par un en-tête, qui contient des informations générales permettant au système de fonctionner. Cet en-tête est lui-même précédé par un MBR tout ce qu'il y a de plus normal, pour des raisons de compatibilité.



GUID Partition Table Scheme

Les adresses de chaque partition sont des adresses LBA codées sur 8 octets (64 bits), ce qui permet de gérer des partitions plus grandes qu'avec le MBR. Voici à quoi ressemble une entrée de la table des partitions.

Type de la partition	Identifiant GUID	Première adresse LBA	Dernière adresse LBA	Attributs de la partition	Nom de la partition
16 octets	16 octets	8 octets	8 octets	8 octets	72 octets

Fichiers

Les données sont organisées sur le disque dur sous la forme de **fichiers**. Ce sont généralement de simples morceaux d'une mémoire de masse, sur lesquels un programme peut écrire ce qu'il veut. Le système de fichiers attribuent plusieurs caractéristiques à chaque fichier.

- Chaque fichier reçoit un **nom de fichier**, qui permet de l'identifier et de ne pas confondre les fichiers entre eux. 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, et d'autres choses encore. Ces informations en plus sont appelées des **attributs de fichier**.
- Il existe des normes qui décrivent comment les données doivent être rangées dans un fichier, suivant le type de données à stocker : ce sont les **formats de fichiers**. Le format d'un fichier est indiqué à la fin du nom du fichier par une **extension de fichier**. Généralement, cette extension de nom commence par un ".", suivi d'une abréviation. Par exemple, le .JPEG, le .WAV, le .MP3 ou le .TXT sont des formats de fichiers. Ces formats de fichiers ne sont pas gérés par le système d'exploitation. Tout ce que peut faire l'OS, c'est d'attribuer un format de fichier à une application : il sait qu'un .PDF doit s'ouvrir avec un lecteur de fichiers .PDF, par exemple.

Clusters

Pour rappel, toutes les mémoires de masse sont découpées en secteurs de taille fixe, qui possèdent tous une adresse. Les systèmes de fichiers actuels ne travaillent pas toujours sur la base des secteurs, mais utilisent des **clusters**, des groupes de plusieurs secteurs consécutifs. L'adresse d'un cluster est celle de son premier secteur. La taille d'un cluster dépend du système de fichiers utilisé, et peut parfois être configuré. Par exemple, Windows permet d'utiliser des tailles de clusters comprises entre 512 octets et 64 kilooctets.

La taille idéale des clusters dépend du nombre de fichiers à stocker et de leur taille. Pour de petits fichiers, il est préférables d'utiliser une taille de cluster faible, alors que les gros fichiers ne voient pas d'inconvénients à utiliser des clusters de grande taille. En effet, un fichier doit utiliser un nombre entier de clusters. Illustrons la raison par un exemple. Si un fichier fait 500 octets, il utilisera un cluster complet : seuls les 500 octets seront utilisés, les autres étant tout simplement inutilisés tant que le fichier ne grossit pas. Si le cluster a une taille de 512 octets, seuls 2 octets seront gâchés. Mais avec une taille de cluster de 4 kilooctets, la grosse majorité du cluster sera inoccupé. La même chose a lieu quand les fichiers prennent un faible nombre de clusters. Par exemple, un fichier de 9 kilooctets n'utilisera pas d'espace inutile avec des clusters de 512 octets : le fichier utilisera 18 clusters au complet. Mais avec une taille de clusters de 4 Ko, trois secteurs seront utilisés, dont le dernier ne contiendra qu'1 Ko de données utiles, 3 Ko étant gâchés.

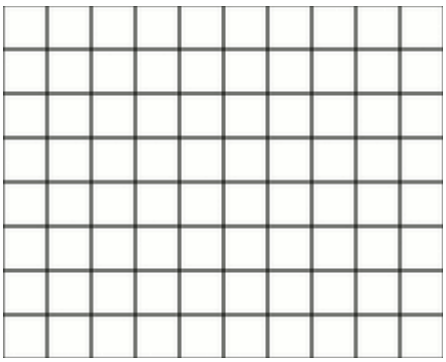
En comparaison, le débit en lecture et écriture sera nettement amélioré avec des clusters suffisamment gros. Pour comprendre pourquoi, il faut savoir que le débit d'un disque dur est le produit de deux facteurs : la taille d'une unité d'allocation (ici, un cluster) multiplié par le nombre d'**opérations disque par secondes**. Ces opérations disque correspondent à une lecture ou écriture de cluster. Chaque opération d'entrée-sortie utilise un peu de processeur, vu qu'il s'agit d'un appel système. Cela peut se résumer par la formule suivante, avec D le débit du HDD, IOPS le nombre d'opérations disque par secondes, et $T_{cluster}$ la taille d'un cluster : $D = IOPS \times T_{cluster}$. Pour un débit constant, augmenter la taille d'un cluster diminue

le nombre d'opérations disque nécessaires. Le débit maximal est donc plus stable, beaucoup de disques durs ayant une limite au nombre d'opérations disque qu'ils peuvent gérer en une seconde. Expérimentalement, on constate que le débit en lecture ou écriture est meilleur avec une grande taille de cluster, du moins pour des accès à de gros fichiers, dont la lecture/écriture demande d'accéder à des secteurs/cluster consécutifs.

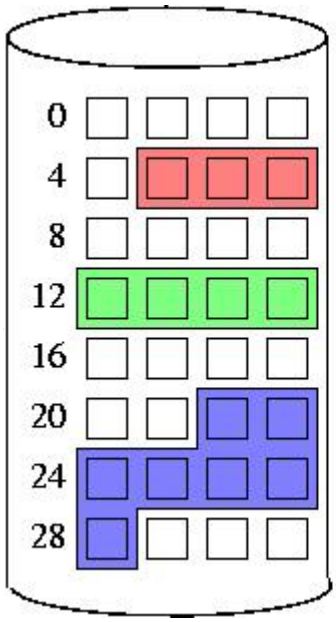
Allocation des secteurs

Le système d'exploitation doit mémoriser pour chaque fichier, quels sont les secteurs du HDD qui correspondent (leur adresse). Cette liste de secteurs est stockée dans une table de correspondance, située au tout début d'une partition : la **Master File Table**. Celle-ci contient, pour chaque fichier, son nom, ses attributs, ainsi que la liste de ses clusters. Avec un système de fichiers de ce type, chaque partition est décomposée en quatre portions :

- le secteur de boot proprement dit, qui contient le chargeur de système d'exploitation ;
- la **Master File Table**, qui contient la liste des fichiers et leurs clusters ;
- et enfin, les fichiers et répertoires contenus dans le répertoire racine.


Opération de défragmentation

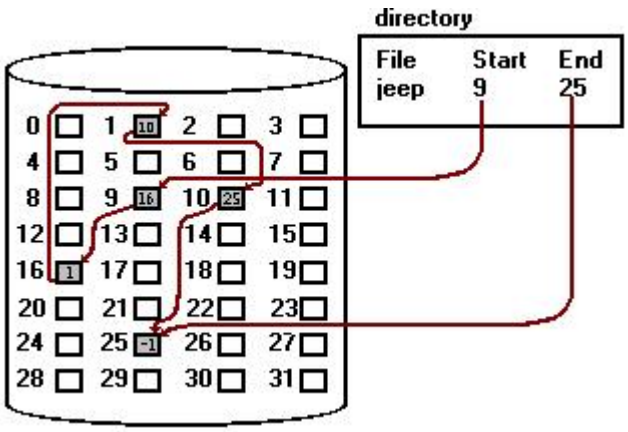
Reste que lors de la création ou de l'agrandissement d'un fichier, l'O.S doit lui allouer un certain nombre de secteurs. Pour cela, il existe diverses méthodes d'allocation. La première méthode est celle de l'**allocation contiguë**. Elle consiste à trouver un bloc de secteurs consécutifs capable d'accepter l'ensemble du fichier. Repérer un fichier sur le disque dur demande simplement 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 l'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, les choses changent. A force de supprimer ou d'ajouter des fichiers de taille différentes, on se retrouve avec des blocs de secteurs vides, 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.



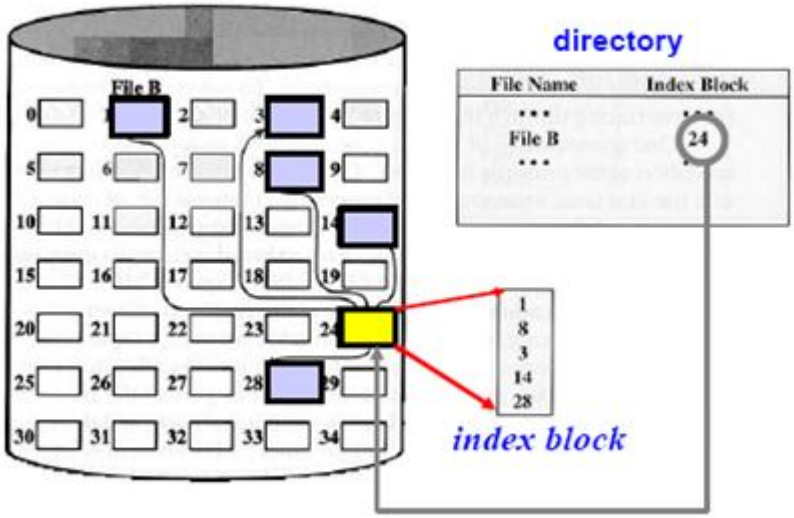
Directory:

file	start	length
moo	5	3
snow	22	7
fall	12	4

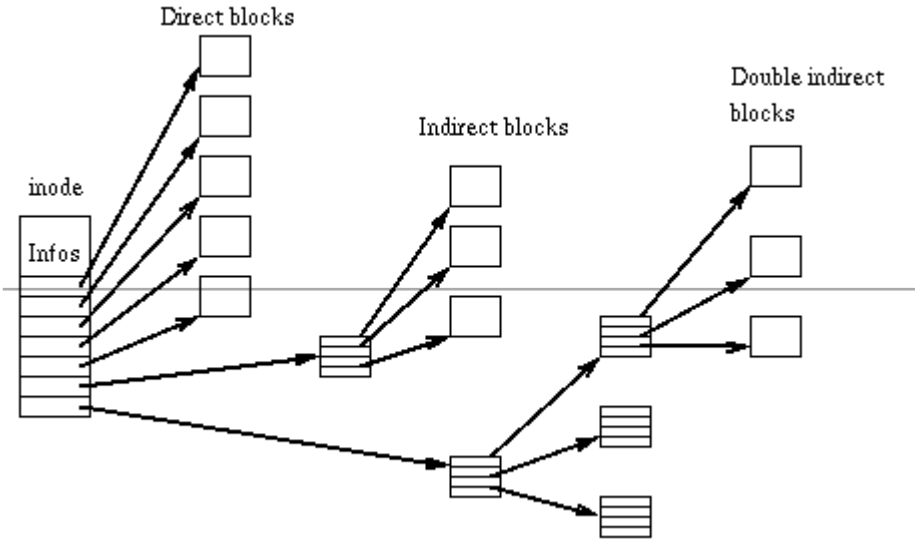
Sur d'autres systèmes de fichiers, les clusters d'un fichier ne sont donc pas forcément consécutifs sur le disque dur. Le système d'exploitation doit garder, pour chaque fichier, la liste des clusters qui correspondent à ce fichier. Cette liste est ce qu'on appelle une liste chainée : chaque cluster 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 défragmentation reste utile sur les HDD, vu que l'accès à des secteurs consécutifs est plus rapide. On parle d'**allocation par liste chainée**.



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 : parcourir cette table en mémoire RAM, est plus rapide 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. On parle d'**allocation indexée**, ou allocation par FAT.



Une autre méthode consiste à ne pas stocker les correspondances (cluster -> cluster suivant), mais simplement les adresses des clusters les unes à la suite des autres : on obtient ainsi un **i-nœud**. C'est cette méthode qui est utilisée dans les systèmes d'exploitation UNIX et leurs dérivés (Linux, notamment). Sous Linux, la table des adresses de cluster a cependant une taille limitée. Pour gérer des fichiers de taille arbitraire, il est possible d'utiliser plusieurs tables d'i-noeuds par fichier. Ces tables sont reliées sous la forme d'une liste chainée, chaque table d'i-noeud indiquant la suivante. Dans certains cas extrêmes, cette organisation en liste chainée disparait au profit d'une organisation partiellement arborescente, certaines tables contenant plusieurs pointeuyrs vers les tables suivantes (tables doublement indirectes).



Répertoires

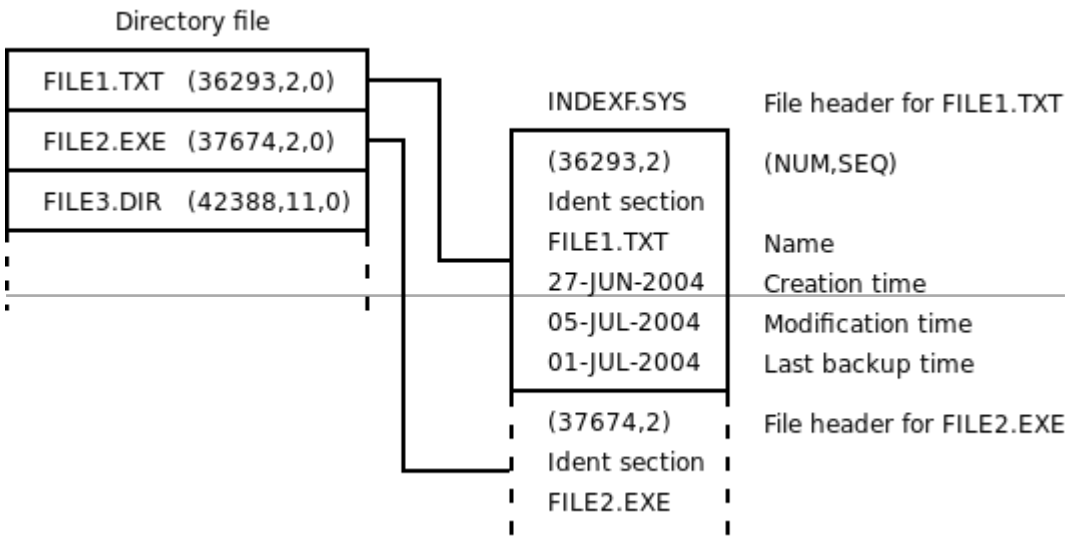
Sur les premiers systèmes d'exploitation, on ne pouvait pas ranger les fichiers : tous les fichiers étaient placés sur le disque dur sans organisation. De nos jours, tous les systèmes d'exploitation gèrent des **répertoires**. Ceux-ci sont représentés sur le disque dur par des fichiers, qui contiennent des liens vers les fichiers contenus dans le répertoire. Le fichier répertoire mémorise aussi toutes les informations concernant les fichiers : leur nom, leur extension, leur taille, leurs attributs, etc. Ces informations sont rarement stockées dans le fichier lui-même. Ainsi, quand vous ouvrez un répertoire ou que vous consultez

les propriétés d'un fichier, ce fichier n'est jamais ouvert : ces informations sont récupérées dans le répertoire.

Codage des répertoires

Ce schéma illustre l'entrée d'un fichier (dans un répertoire) telle qu'elle est codée avec les systèmes de fichier FAT 12 et FAT 16. La FAT-32 utilise un système quasiment identique, sauf pour l'adresse du premier cluster, qui est codée sur 4 octets.

Un répertoire contient une liste de fichier. Pour chaque fichier, il mémorise son nom, ses attributs, ainsi que la liste des clusters qui lui sont attribués. Les attributs sont stockés dans une structure de taille fixe, le nombre des attributs par fichier étant connu à l'avance. Mais la liste des clusters et les noms des fichiers n'ont pas de taille fixée à l'avance. Cela ne pose pas de problème pour la liste des clusters, contrairement à ce qu'on a pour les noms. Généralement, les répertoires mémorisent les noms des fichiers qu'ils contiennent, avec leurs attributs. Ceux-ci sont donc placées dans un bloc de taille fixe, un en-tête.



La gestion des noms de fichiers peut se faire de plusieurs manières. Avec la première méthode, on limite la

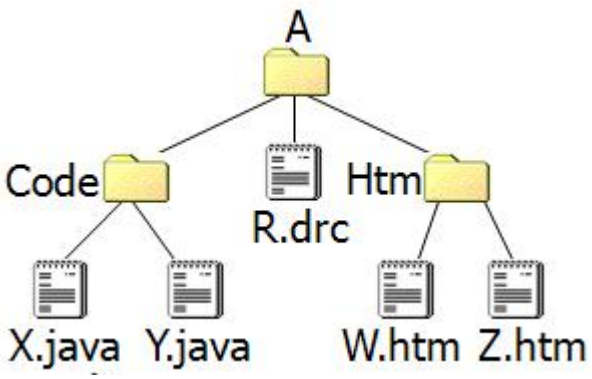
taille des noms de fichiers. On peut alors réserver une taille suffisante dans le fichier répertoire pour stocker ce nom. Par exemple, sur les premiers systèmes de fichiers FAT, les noms de fichiers étaient limités à 8 caractères (plus 3 pour l'extension de fichier). Le répertoire allouait 8 octets pour chaque nom de fichier, plus 3 octets pour l'extension. La taille utilisée était alors limitée. Cette méthode, bien que simple d'utilisation, a tendance à gâcher de la mémoire si de grands noms de fichiers sont utilisés. Et si de petits noms de fichier, l'espace utilisé sera faible, mais les utilisateurs pourront se sentir limités par la taille maximale des noms de fichier.

On peut aussi utiliser des noms de fichiers de taille variable. La solution la plus simple consiste à placer ceux-ci à la suite de l'en-tête (qui contient les attributs). Cette méthode a cependant un désavantage lors de la suppression du fichier : elle génère de la fragmentation assez facilement. Pour éviter cela, on peut placer les noms de fichier de taille variable à la fin du fichier répertoire, chaque en-tête pointant vers le nom de fichier adéquat (Linux).

Hiérarchie de répertoires

Sur les premiers systèmes d'exploitation, tous les fichiers étaient placés dans un seul et unique répertoire. Quand le nombre de fichier était relativement faible, cela ne posait pas trop de problèmes. L'avantage de cette méthode était que le système d'exploitation était plus simple et qu'il n'avait pas à gérer une arborescence de répertoire sur le disque dur. Mais l'inconvénient se faisait sentir quand les fichiers devenaient de plus en plus nombreux. Cette organisation est encore utilisée sur les baladeurs et appareils photo numériques.

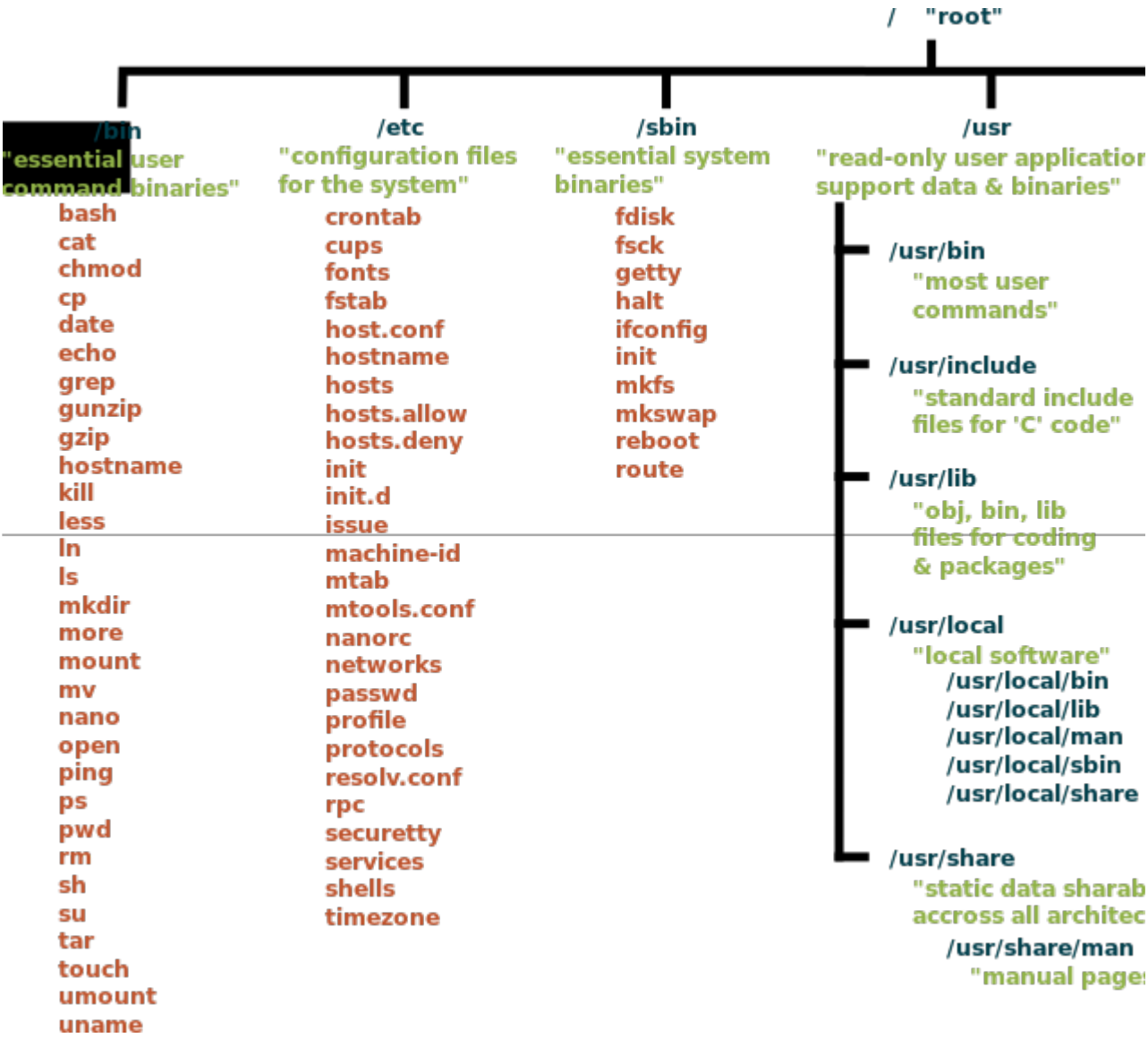
De nos jours, les répertoires sont organisés en une **hiérarchie de répertoire** : un répertoire peut contenir d'autres répertoires, et ainsi de suite. Évidemment, cette hiérarchie commence par un répertoire maître, tout en haut de cette hiérarchie : ce répertoire est appelé la **racine**. Sous Windows, on trouve une seule et unique racine par partition ou disque dur : on en a une sur C : , une autre sur D : , etc. Sous Linux, il n'existe qu'une seule racine, chaque partition étant un répertoire accessible depuis la racine : ces répertoires sont appelés des **points de montage**. Sur Windows, ce répertoire est noté : soit \, soit nom_lecteur : \ (C:\, D:\). Sous Linux, il est noté : / .



Reste que savoir où se trouve un fichier demande de préciser quel est le chemin qu'il faut suivre en partant du répertoire maître : on doit préciser qu'il faut ouvrir tel répertoire, puis tel autre, et ainsi de suite jusqu'au fichier. Ce chemin, qui reprendre le chemin de la racine jusqu'au fichier, est appelé le **chemin d'accès absolu**. Sous Windows, les noms de répertoires sont séparés par un \ : C:\Program Files\Internet Explorer. Sous Linux, les noms de répertoires sont séparés par un / : /usr/ast/courrier. Les

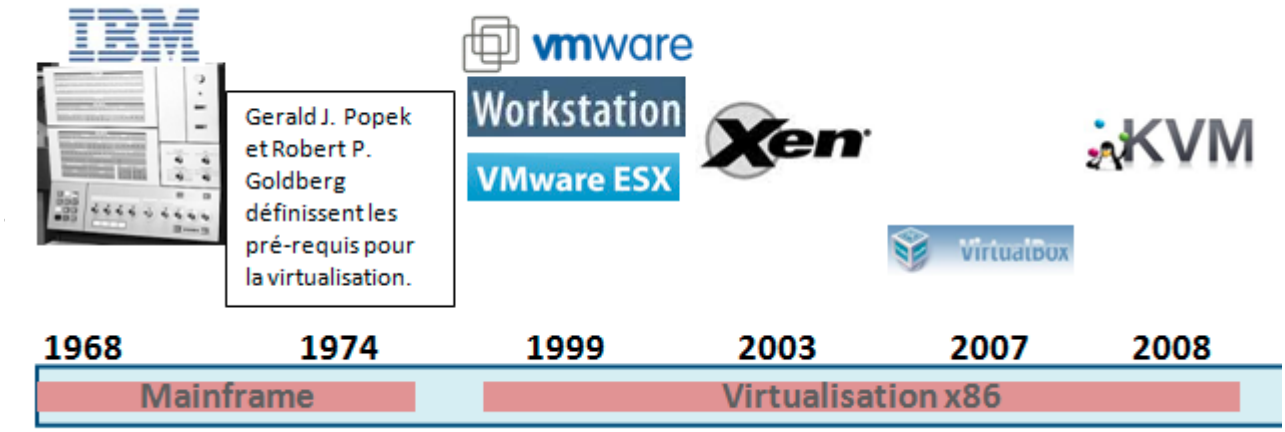
chemins d'accès relatifs sont similaires, sauf qu'ils ne partent pas de la racine : ils partent d'un répertoire choisit par l'utilisateur, nommé répertoire courant. Dans les grandes lignes, l'utilisateur ou un programme choisissent un répertoire et lui attribuent le titre de répertoire courant. Généralement, chaque processus a son propre répertoire de travail, qu'il peut changer à loisir. Pour cela, le programme doit utiliser un appel système, chdir sous Linux.

Linux a une particularité : il normalise la place de certains répertoires bien précis du système d'exploitation. L'arborescence des répertoires est en effet standardisé par le **Filesystem Hierarchy Standard**, la dernière version datant de juin 2015.



Virtualisation et machines virtuelles

Grâce aux différentes technologies de virtualisation, il est possible de lancer plusieurs systèmes d'exploitation en même temps sur le même ordinateur. Il est ainsi possible de passer d'un O.S à un autre relativement rapidement, suivant les besoins, avec un simple raccourci clavier. Vous avez certainement déjà eu l'occasion d'utiliser VMWARE ou un autre logiciel de virtualisation. Ces technologies sont aussi beaucoup utilisées sur les serveurs, entre autres.

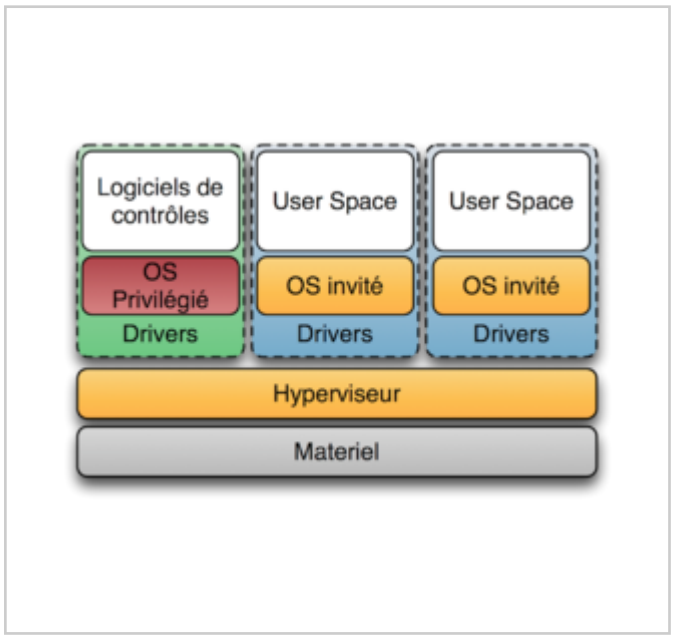


Machine virtuelle

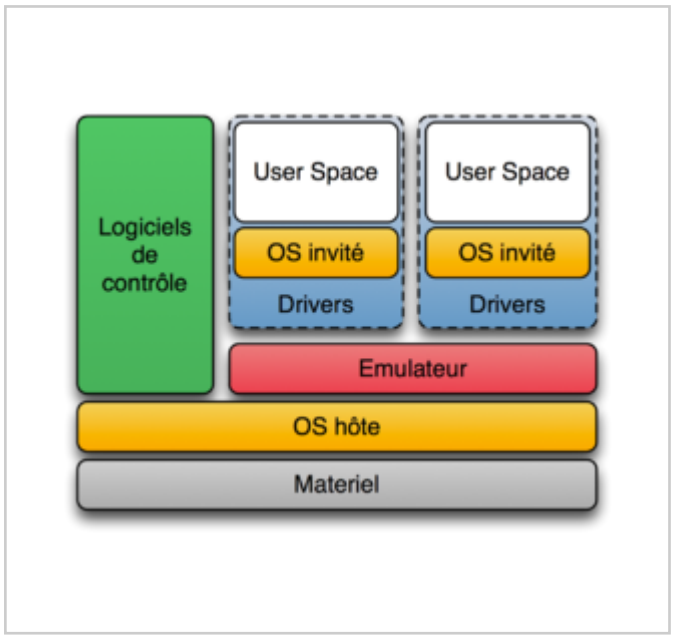
Ces logiciels implémentent ce qu'on appelle des **machines virtuelles**. Il s'agit d'une sorte de faux matériel, garantie par un logiciel. Un logiciel qui s'exécute dans une machine virtuelle aura l'impression de s'exécuter sur un matériel et/ou un O.S différent du matériel sur lequel il est en train de s'exécuter.

Ces machines virtuelles sont utilisés aussi bien pour les techniques de virtualisation que sur ce qu'on appelle les **émulateurs**, des logiciels qui permettent de simuler le fonctionnement d'anciens ordinateurs ou consoles de jeux. Mais les techniques de virtualisation fonctionnent un peu différemment de l'émulation de consoles de jeux. En effet, un émulateur a besoin d'émuler le processeur, qui est généralement totalement différent entre le matériel émulé et le matériel sur lequel s'exécute la V.M (virtual machine, machine virtuelle). Ce n'est pas le cas avec la virtualisation, le jeu d'instruction étant généralement le même.

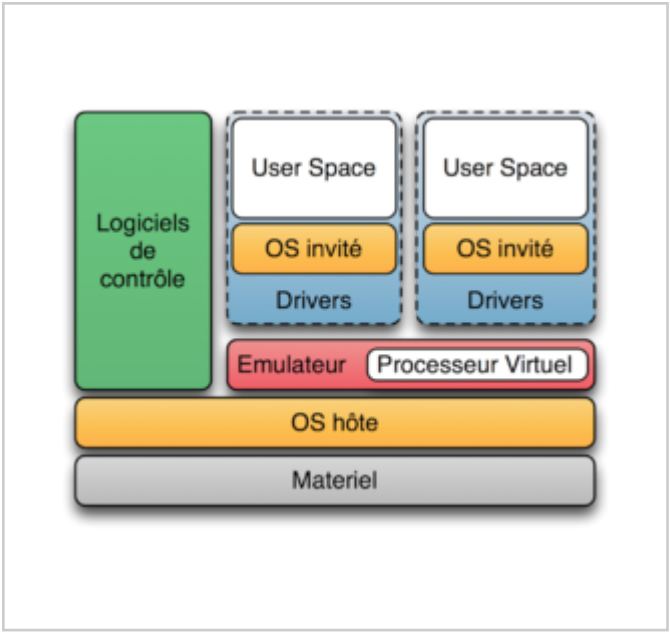
Le logiciel de virtualisation est généralement limité à un logiciel appelé **hyperviseur**. Celui-ci gère plusieurs machines virtuelles, une par O.S lancé sur l'ordinateur. Il existe différents types d'hyperviseurs, qui sont généralement appelées sous les noms techniques de virtualisation de type 1 et 2. Dans le premier cas, l'hyperviseur s'exécute directement, sans avoir besoin d'un O.S sous-jacent. Dans le second cas, l'hyperviseur est un logiciel applicatif comme un autre, qui s'exécute sur un O.S bien précis. Les émulateurs utilisent systématiquement un hyperviseur de type 2 amélioré, capable de simuler le jeu d'instruction du processeur émulé.



Hyperviseur de type 1



Hyperviseur de type 2



Emulateur

Une méthode annexe à la virtualisation de type 2 consiste à modifier l'O.S à virtualiser de manière à ce qu'il puisse communiquer directement avec l'hyperviseur. On parle alors de **paravirtualisation**. Avec cette méthode, les appels systèmes des O.S sont remplacés par des appels systèmes implémentés par l'hyperviseur.

Certains processeurs disposent de technologies matérielles d'accélération de la virtualisation. Sur les processeurs x86, on peut citer l'Intel Vtx ou l'AMD-V.

Fonctionnement/implémentation

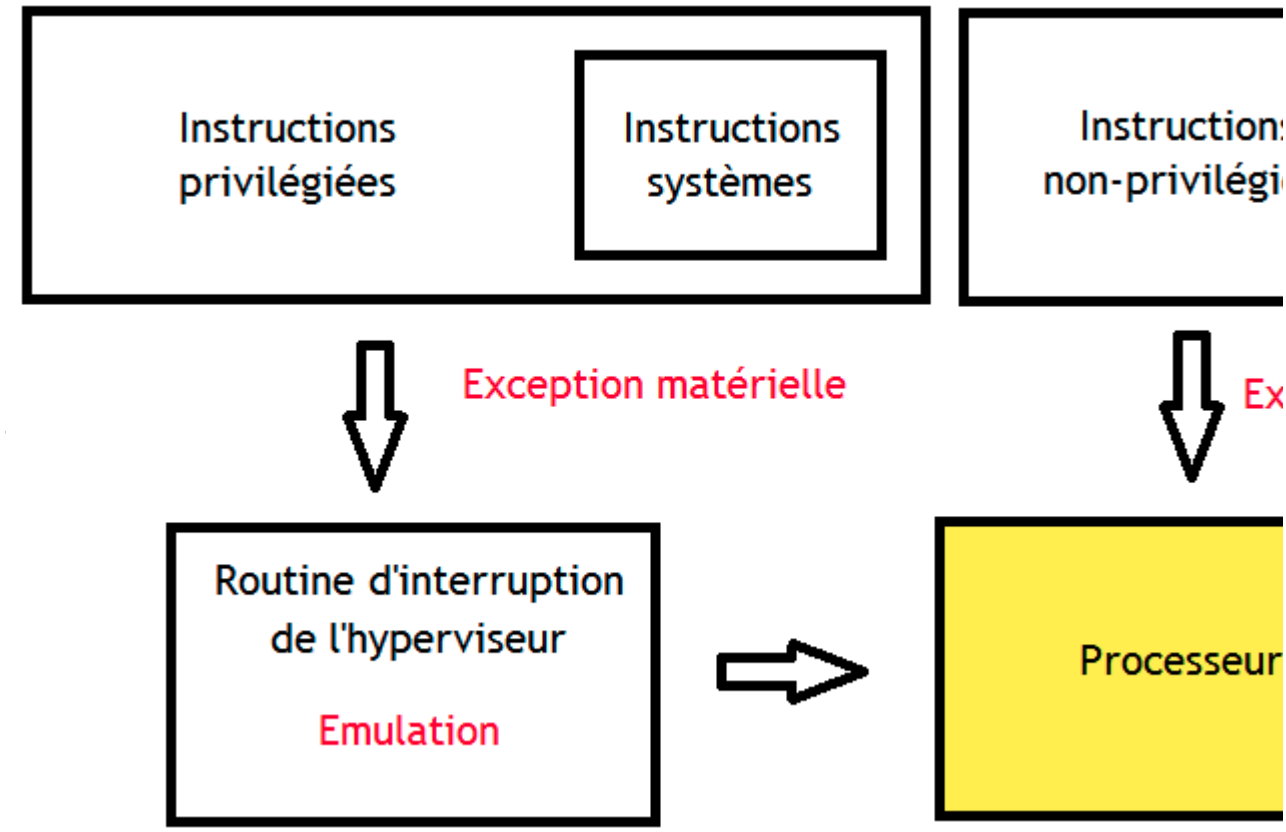
On peut se demander comment les technologies de virtualisation et les émulateurs font pour simuler une machine virtuelle. Pour être considéré comme un logiciel de virtualisation, un logiciel doit remplir trois critères :

- L'équivalence : l'O.S virtualisé et les applications qui s'exécutent doivent se comporter comme s'ils étaient exécutés sur le matériel de base, sans virtualisation.
- L'efficacité : La grande partie des instructions machines doit s'exécuter directement sur le processeur, afin de garder des performances correctes. Ce critère n'est pas respecté par les émulateurs matériels, qui doivent simuler le jeu d'instruction du processeur émulé.
- Le contrôle des ressources : tout accès au matériel par l'O.S virtualisé doit être intercepté par la machine virtuelle et intégralement pris en charge par l'hyperviseur.

Remplir ces trois critères est possible sous certaines conditions, établies par la théorie de la virtualisation de Popek et Goldberg. Ceux-ci ont réussit à établir plusieurs théorèmes sur la possibilité de créer un hyperviseur pour une architecture matérielle quelconque. Ces théorèmes se basent sur la présence de différents types d'instructions machines. Pour faire simple, on peut distinguer les **instructions systèmes**, qui agissent sur le matériel. Il s'agit d'instructions d'accès aux entrées-sorties, aussi appelées instructions sensibles à la configuration, et d'instructions qui reconfigurent le processeur, aussi appelées instructions sensibles au comportement. Certaines instructions sont éxecutables uniquement si le

processeur est en mode noyau. Si ce n'est pas le cas, le processeur considère qu'une erreur a eu lieu et lance une exception matérielle. Ces instructions sont appelées des **instructions privilégiées**. On peut considérer qu'il s'agit d'instructions que seul l'O.S peut utiliser. A côté, on trouve des instructions **non-privilégiées** qui peuvent s'éxecuter aussi bien en mode noyau qu'en mode utilisateur.

La théorie de Popek et Goldberg dit qu'il est possible de virtualiser un O.S à une condition : que les instructions systèmes soient toutes des instructions privilégiées. Si c'est le cas, virtualiser un O.S signifie simplement le démarrer en mode utilisateur. Quand l'O.S éxecutera un accès au matériel, il le fera via une instruction privilégiée. Ce faisant, celle-ci déclenchera une exception matérielle, qui sera traitée par une routine d'interruption spéciale. L'hyperviseur n'est ni plus ni moins qu'un ensemble de routines d'interruptions, chaque routine simulant le fonctionnement du matériel émulé. Par exemple, un accès au disque dur sera émulé par une routine d'interruption, qui utilisera les appels systèmes fournit par l'OS pour accéder au disque dur réellement présent dans l'ordinateur. Cette méthode est souvent appelé la méthode trap-and-emulate.



Mais le critère précédent n'est pas respecté par beaucoup de jeux d'instructions. Par exemple, ce n'est pas le cas sur les processeurs x86, qu'ils soient 32 ou 64 bits. Diverses instructions systèmes ne sont pas des instructions privilégiées. La solution est simplement de traduire à la volée les instructions systèmes (privilégiées ou non-privilégiées) en appels systèmes équivalents. Cette technique est appelée la **réécriture de code**.

Assembleur et édition des liens

La majorité des programmes actuels sont écrits dans des langages de haut niveau, comme le C, le Java, le C++, l'Erlang, le PHP, etc. Mais ces langages ne sont pas compréhensibles par le processeur, qui ne comprend que le langage machine. Les codes sources écrits dans des langages de haut niveau doivent donc être traduit en langage machine par la **chaine de compilation**. A l'heure actuelle, la majorité des compilateurs ne traduit pas directement un langage de haut niveau en langage machine, mais passe par un langage intermédiaire : l'**assembleur**. Cet assembleur est un langage de programmation, utilisé autrefois par les programmeurs, assez proche du langage du langage machine. Il va de soit que cet assembleur doit être traduit en langage machine pour être exécuté par le processeur. Tout ce qui se passe entre la compilation et l'exécution du programme est pris en charge par trois programmes qui forment ce qu'on appelle la **chaîne d'assemblage**. Cette chaine d'assemblage commence par le logiciel d'assemblage qui traduit le code assembleur en code machine. Ce code machine est alors mémorisé dans un fichier objet. Ensuite, l'éditeur de liens, ou linker, qui combine plusieurs fichiers objets en un fichier exécutable. Enfin, le chargeur de programme, aussi appelé loader, charge les programmes en mémoire. L'ensemble regroupant compilateur et chaine d'assemblage, avec éventuellement des interpréteurs, est appelé la chaine de compilation.

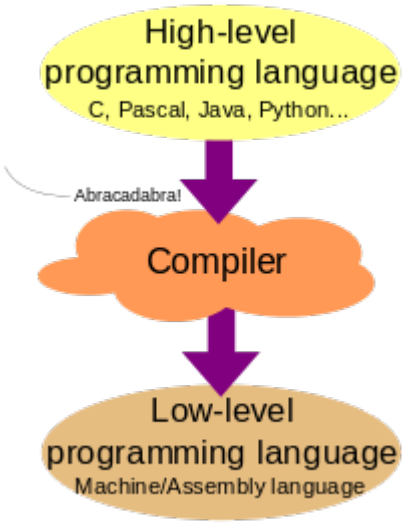


Illustration du processus de compilation.

La chaine de compilation

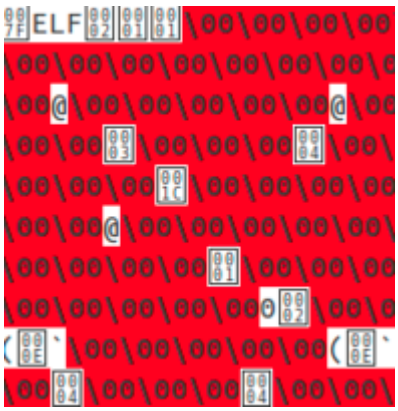
```c
#include <stdio.h>

int main()
{
  printf("hello world");
  return 0;
}
```

Un programme écrit dans un langage de haut niveau (le C, en l'occurrence).

```
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "hello world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

Le code assembleur.



Le programme en langage machine.

Traduire les l'assembleur en instructions-machine est la première tâche du logiciel d'assemblage, mais ce n'est pas la seule. Il s'occupe aussi de la **résolution des symboles**, à savoir qu'il transforme les labels (et autres symboles) en adresses mémoires (l'adresse de l'instruction cible d'un branchement, ou adresse d'une variable). Pour détailler plus, il faut faire la différence entre les symboles locaux, utilisés uniquement dans le module où ils sont définis (ils servent à fabriquer des fonctions, variables statiques et structures de contrôle), des symboles globaux, référencés dans d'autres fichiers (ils servent pour nommer les fonctions accessibles à l'extérieur du module, et les variables globales). De plus, la chaine d'assemblage doit s'occuper en partie de la **relocation**, même si cela dépend fortement du processeur. Les contraintes d'alignement mémoire, la taille variable des instructions, et les modes d'adressages utilisés vont devoir être pris en compte lors du processus de relocation. Évidemment, les logiciels de la chaine d'assemblage peuvent aussi optimiser le code assembleur généré, de manière à le rendre plus rapide. Mais nous passerons cette fonctionnalité sous silence dans ce qui suit.

Le logiciel d'assemblage

L'assembleur permet, comme le langage machine, d'utiliser directement le instructions du jeu d'instruction, mais dans une version nettement plus facile pour les êtres humains. Au lieu d'écrire ces instructions sous la forme de suites de bits, la programmation en assembleur décrit chaque instruction en utilisant du texte. Ainsi, les opérandes peuvent être écrites en décimal, au lieu de devoir être écrites en binaires. De plus, les opcodes sont remplacées par un mot : la **mnémonique**. Pour donner un exemple, l'instruction d'addition aura pou mnémonique ADD, ce qui est plus facile à taper que le code binaire correspondant. Il existe aussi des mnémoniques pour les registres, qui portent le nom de noms de registres. Par exemple, l'assembleur x86 nomme les registres avec les ménmoniques EAX, EBX, ECX, EDX, etc. Les modes d'adressages sont pris en charge via des syntaxes spéciales. Il est aussi possible de nommer une ligne de code (une instruction, donc) en lui attribuant un label, qui est remplacé par l'adresse de l'instruction lors de la traduction en langage machine. Ces labels sont utiles pour remplacer l'adresse de destination des branchements, ce qui permet de ne pas écrire ces adresses en dur.

Certains assembleurs permettent aussi de créer des macros, des ancêtres des fonctions. Ces macros sont simplement des suites d'instructions qui sont nommées. Il est possible d'insérer ces macros n'importe où dans le code, en mettant leur nom à cet endroit en utilisant une syntaxe spéciale. La suite d'instruction sera alors recopiée à l'endroit où le nom de la macro a été inséré, lors de la traduction en langage machine. Cela évite d'avoir à faire des copier-coller.

Le logiciel d'assemblage traduit chaque ligne de code l'une après l'autre. Cela demande de :

- remplacer la mnémonique par l'opcode correspondant ;
- remplacer les opérandes par la suite de bits correspondante ;
- remplacer les labels par l'adresse qui correspond ;
- remplacer les macros par leur contenu ;
- ne rien faire sur les commentaires.

```
MONITOR FOR 6802 1.4          9-16-80  TSC ASSEMBLER  PAGE    2


C000             ORG    ROM+$0000 BEGIN MONITOR
C000 8E 00 7F  START  LDS    #STACK

                     ***********************************
                     * FUNCTION: INITA - Initialize ACIA
                     * INPUT: none
                     * OUTPUT: none
                     * CALLS: none
                     * DESTROYS: acc A

0013             RESETA  EQU    %00010011
0011             CTLREG  EQU    %00010001

C003 86 13       INITA   LDA A  #RESETA   RESET ACIA
C005 B7 80 04            STA A  ACIA
C008 86 11               LDA A  #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04            STA A  ACIA

C00D 7E C0 F1            JMP    SIGNON    GO TO START OF MONITOR

                     ***********************************
                     * FUNCTION: INCH - Input character
                     * INPUT: none
                     * OUTPUT: char in acc A
                     * DESTROYS: acc A
                     * CALLS: none
                     * DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04  INCH   LDA A  ACIA      GET STATUS
C013 47               ASR A            SHIFT RDRF FLAG INTO CARRY
C014 24 FA            BCC    INCH      RECIEVE NOT READY
C016 B6 80 05         LDA A  ACIA+1    GET CHAR
C019 84 7F            AND A  #$7F      MASK PARITY
C01B 7E C0 79         JMP    OUTCH     ECHO & RTS

                     ***********************************
                     * FUNCTION: INHEX - INPUT HEX DIGIT
                     * INPUT: none
                     * OUTPUT: Digit in acc A
                     * CALLS: INCH
                     * DESTROYS: acc A
                     * Returns to monitor if not HEX input

C01E BD F0     INHEX  BSR    INCH      GET A CHAR
C020 81 30            CMP A  #'0       ZERO
C022 2B 11            BMI    HEXERR    NOT HEX
C024 81 39            CMP A  #'9       NINE
C026 2F 0A            BLE    HEXRTS    GOOD HEX
C028 81 41            CMP A  #'A
C02A 2B 09            BMI    HEXERR    NOT HEX
C02C 81 46            CMP A  #'F
C02E 2E 05            BGT    HEXERR
C030 80 07            SUB A  #7        FIX A-F
C032 84 0F     HEXRTS AND A  #$0F      CONVERT ASCII TO DIGIT
C034 39               RTS

C035 7E C0 AF  HEXERR JMP    CTRL      RETURN TO CONTROL LOOP
```

Langage assembleur du processeur Motorola 6800.

Il faut cependant noter que les langages assembleurs ne sont pas forcément rudimentaires. Il existe notamment des assembleurs de haut-niveau, qui gèrent structures de contrôle, fonctions, procédures, types de données composites (structures et unions), voire la programmation orientée objet par classes. Ceux-ci demandent d'ajouter une phase de compilation en assembleur « pur » à un logiciel d'assemblage plus rudimentaire.

Macros

La première opération que va faire le logiciel d'assemblage est le remplacement [appel de macro → corps de la macro]. Ces macros sont sauvegardées dans une table de hachage qui associe un nom de macro à son corps : la **table des macros**. Cependant, les choses deviennent beaucoup plus compliquées quand les macros contiennent des arguments/paramètres, ou quand le logiciel d'assemblage utilise des macros imbriquées ou récursives. Pour effectuer ce remplacement, le logiciel d'assemblage peut procéder en une ou deux étapes, la situation la plus simple à comprendre étant celle à deux étapes.

Un assembleur à deux étapes (on peut aussi dire : à deux passes) procède comme suit : la première étape parcourt le code assembleur pour détecter les macros, alors que la seconde effectue le remplacement. L'algorithme de la première passe lit le fichier source ligne par ligne. Si une ligne est une macro, il crée une entrée dans la table des macros et copie les lignes de code suivantes dans celle-ci. Il continue ainsi jusqu'à tomber sur le mot-clé qui indique la fin du corps de la macro. C'est aussi lors de cette passe que sont gérées les directives. Par la suite, le logiciel d'assemblage va effectuer le remplacement, qui ne peut pas se faire « en place » : le logiciel d'assemblage doit copier le fichier source dans un nouveau fichier.

Certains assembleurs forcent la déclaration des macros en début de fichier, juste avant le code principal. Dans ce cas, les macros sont toutes déclarées avant leur utilisation, ce qui permet de fusionner ces deux passes en une seule : c'est la passe 0.

- Lire une ligne de code dans le fichier source ;
- Si cette ligne est un nom de macro, créer une entrée dans la table des macros, et copier les lignes de code suivantes dans celle-ci jusqu'à tomber sur le mot-clé qui indique la fin du corps de la macro : à ce moment-là, on retourne à l'étape 1 ;
- Si cette ligne est une directive qui peut être gérée lors de la passe 0, l'exécuter, puis retourner à l'étape 1 ;
- Si cette ligne est une instruction-machine, copier celle-ci dans le nouveau fichier source (sauf s'il s'agit de la ligne qui indique la fin du programme) ;
- Sinon, accéder à la table des macros et copier le corps de la macro ligne par ligne dans le nouveau fichier.
- Si cette ligne indique la fin du code source, arrêter là.

Mnémoniques

Pour traduire les mnémoniques en opcode, le logiciel d'assemblage contient une table de hachage mnémonique → opcode : la table des opcodes. Cette table associe à chaque mnémonique l'opcode qui correspond, éventuellement la taille de l'opcode pour les architectures à instructions de taille variable ou d'autres informations. Il se peut qu'une mnémonique corresponde à plusieurs instructions-machine, suivant le mode d'adressage. Dans ce cas, chaque entrée de la table des mnémoniques contient plusieurs opcodes, le choix de l'opcode étant fait selon le mode d'adressage. Identifier le mode d'adressage est relativement simple : une simple expression régulière sur la ligne de code suffit. Certaines mnémoniques sont des cas particuliers d'instructions-machine plus générales : on parle de pseudo-instructions. Par exemple, sur les processeurs assez anciens qui utilisent le jeu d'instruction MIPS, l'instruction MOV r1 → r2 est synthétisée à partir de l'instruction suivante : add r1, r0, r2. Dans ces conditions, la table des mnémoniques ne contient pas l'opcode de l'instruction, mais littéralement une bonne partie de la représentation binaire de l'instruction, avec des opérandes (ici, des noms de registres) déjà spécifiés. Un champ pseudo-instruction est ajouté dans la table des opcodes pour savoir si seul l'opcode doit être remplacé ou non.

Symboles locaux

Pour traduire les symboles locaux, le logiciel d'assemblage utilise une **table des symboles**, qui mémorise les correspondances entre un symbole (un label le plus souvent) et sa valeur/adresse. Pour chaque symbole, cette table mémorise diverses informations, dont le nom du symbole, sa valeur et son type. La table des symboles est sauvegardée dans le fichier objet, vu qu'elle servira par la suite pour la relocation au niveau du linker et du loader. Pour remplir la table des symboles, le logiciel d'assemblage doit savoir quelle est l'adresse de l'instruction qu'il est en train de traduire. Pour cela, il suffit de profiter du fait que le logiciel d'assemblage traduit une ligne de code à la fois : le logiciel d'assemblage utilise un simple compteur, mis à zéro avant le démarrage de la traduction, qui est incrémenté de la taille de l'instruction après chaque traduction.

Il se peut que la valeur d'un symbole soit définie quelques lignes plus bas. Le cas le plus classique est celui des branchements vers une ligne de code située plus loin dans le programme. Toute instruction qui utilise un tel label ne peut être totalement traduite qu'une fois la valeur du label connue. Pour résoudre ce problème, il existe deux méthodes : la première correspond à celle des logiciels d'assemblage à deux passes, et l'autre à celle des logiciels d'assemblage à une passe.

Les **assembleurs à deux passes** sont les plus simples à comprendre : le logiciel d'assemblage parcourt une première fois le code ASM pour remplir la table des symboles et repasse une seconde fois pour traduire les instructions. Ces assembleurs font face à un léger problème sur les processeurs dont les instructions sont de taille variable : lors de la première passe, le logiciel d'assemblage doit déterminer la taille des instructions qu'il ne traduit pas, afin de mettre à jour le compteur d'adresse. Et cela demande d'analyser les opérandes et de déterminer le mode d'adressage de l'instruction.

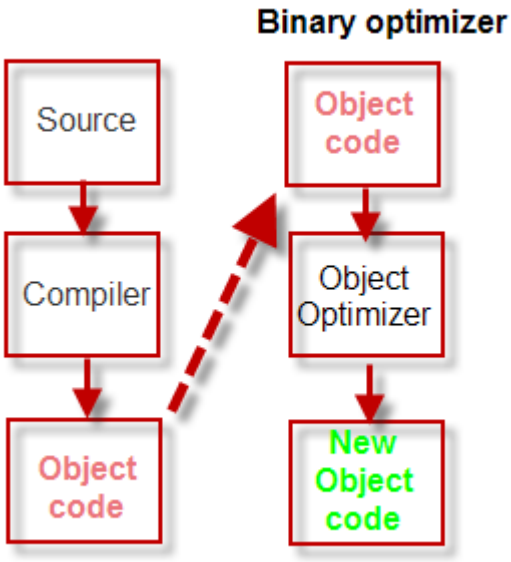
Les **assembleurs à une passe** fonctionnent autrement : ceux-ci vont passer une seule fois sur le code source. La table des symboles de ces assembleurs contient, en plus des correspondances nom-valeur, un bit qui indique si la valeur du symbole est connue et une liste d'attente, qui contient les adresses des instructions déjà rencontrées qui ont besoin de la valeur en question. Quand ils rencontrent une référence dont la valeur n'est pas connue, ils ajoutent le symbole à la table des symboles et l'adresse de l'instruction dans la liste d'attente, et poursuivent le parcours du code source. S'ils rencontrent une instruction qui utilise ce symbole à valeur inconnue, l'adresse de l'instruction est ajoutée à la liste d'attente. Quand la valeur est enfin connue, le bit de la table de symboles est mis à 0 (pour indiquer que la valeur est connue), et le logiciel d'assemblage met à jour les instructions mémorisées dans la liste d'attente. Cette méthode pose cependant quelques problèmes sur les processeurs dont les instructions sont de taille variable.

Noms de registres et autres constantes

On pourrait penser qu'il existe une table de correspondance pour les noms de registres, comme la table de mnémoniques pour les mnémoniques, mais on peut ruser en considérant les noms de registres comme des symboles comme les autres : il suffit de remplir la table des symboles avec les correspondances nom de registre → représentation binaire, et le tour est joué. On peut utiliser le même principe pour gérer les préfixes des instructions, comme sur le jeu d'instruction x86.

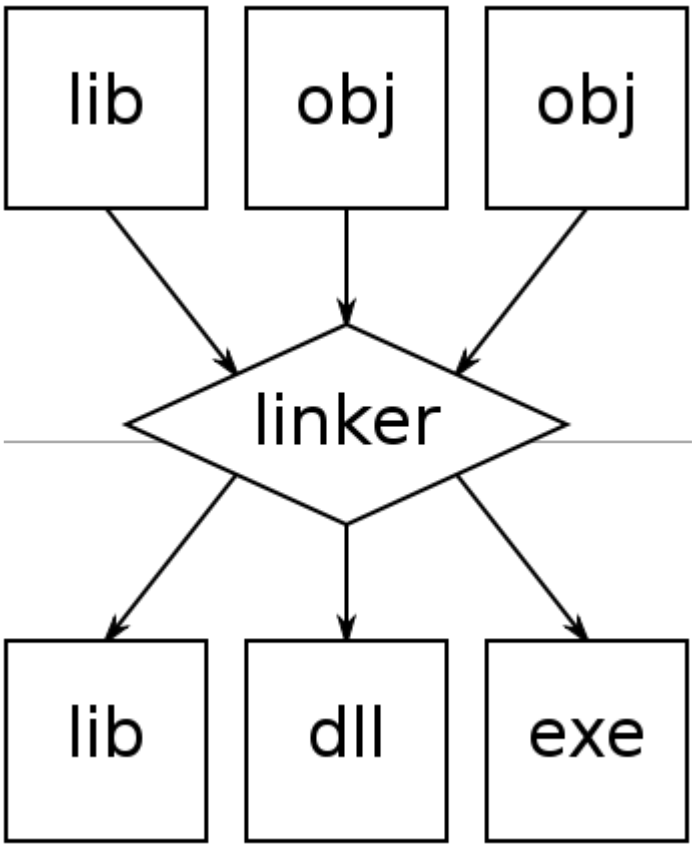
Optimisation

Il n'est pas rare que le code assembleur soit optimisé avant d'être traduit en langage machine. D'autres assembleur optimisent directement le code objet crée. Ces optimisations sont relativement rudimentaires. Il s'agit le plus souvent de remplacement d'instructions lentes par des instructions équivalentes mais plus rapides. Par exemple, il est possible de remplacer des multiplications ou divisions par une puissance de deux par des décalages, plus rapides. Le logiciel d'assemblage peut aussi modifier l'ordre des instructions pour profiter au mieux du pipeline. Dans tous les cas, ces optimisations sont prises en charge à part, soit dans une passe d'optimisation du code assembleur, soit par un logiciel annexe au logiciel d'assemblage, qui optimise directement le code objet.

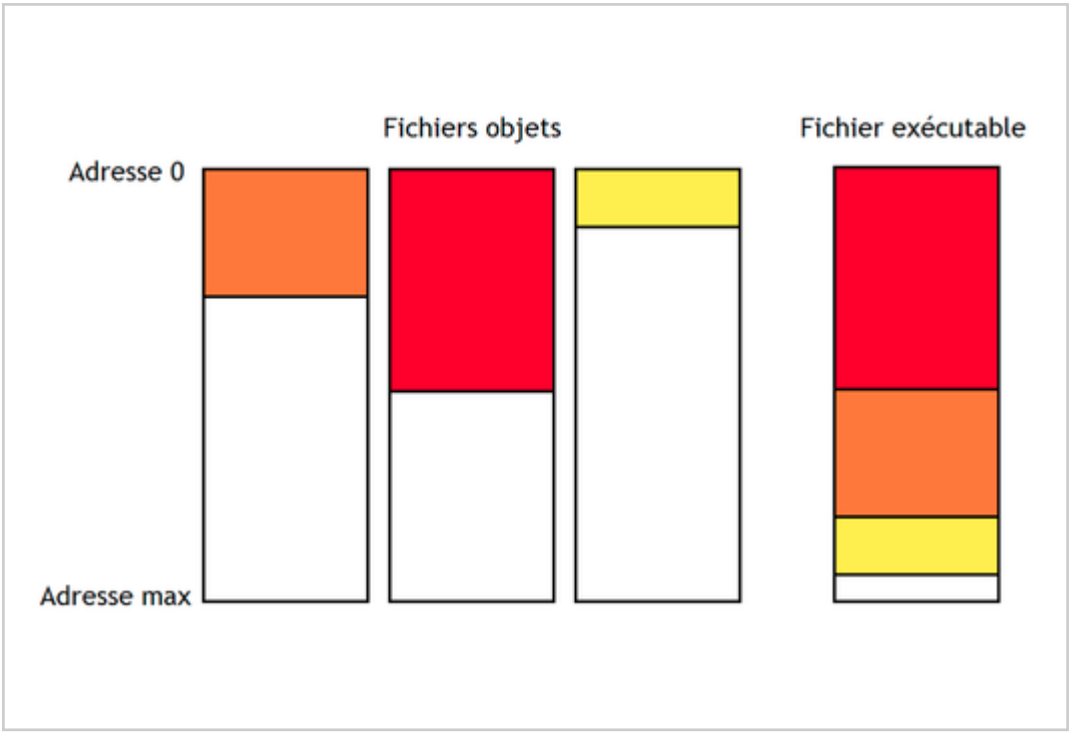


L'édition des liens

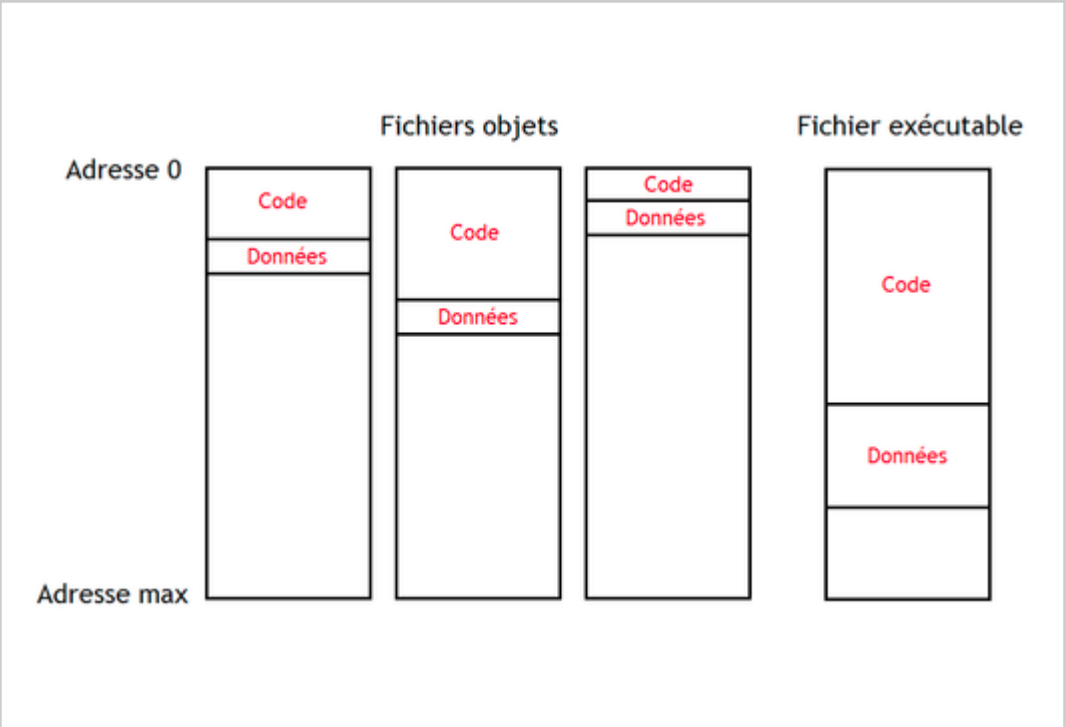
De nos jours, les programmes actuels sont composés de plusieurs modules ou classes, qui sont traduits chacun de leur côté en fichier objet. Les symboles globaux, qui référencent une entité déclarée dans un autre fichier objet, ne sont pas traduits en adresses par le logiciel d'assemblage. Pour obtenir un fichier exécutable, il faut combiner plusieurs fichiers objets en un seul fichier exécutable, et traduire les symboles globaux en adresses : c'est le but de la phase d'édition des liens, effectuée par un **éditeur de liens**, ou linker.



En l'absence de segmentation mémoire (une forme de mémoire virtuelle), le linker concatène les fichiers objets. Avec la segmentation, chaque fichier objet a ses propres segments de code et de données, qui doivent être fusionnés. Pour faire son travail, le linker doit savoir où commence chaque segment dans le fichier objet, son type (code, données, etc.), et sa taille. Ces informations sont mémorisées dans le fichier objet, dans une table de segments, créée par le logiciel d'assemblage. De plus, le linker doit rajouter des vides dans l'exécutable pour des raisons d'alignement au niveau des pages mémoire.



Édition des liens sans segmentation.



Édition des liens avec segmentation.

Relocation

Combiner plusieurs fichiers objets en exécutable va poser un problème similaire à celui obtenu lors du lancement d'un programme en mémoire : les adresses calculées en supposant que le fichier objet commence à l'adresse o sont fausses. Pour corriger celles-ci, le linker doit effectuer une relocation, et considère que le début du fichier exécutable est à l'adresse o. Pour cela, le linker doit connaître l'emplacement des adresses dans le fichier objet, informations qui sont mémorisées dans la table des symboles sauvegardée dans le fichier objet.

Résolution des symboles globaux

De plus, le linker s'occupe de la résolution des symboles globaux. En effet, quand on fait appel à une fonction ou variable localisée dans un autre fichier objet, la position de celle-ci dans le fichier exécutable (et donc son adresse) n'est pas définie avant son placement dans le fichier exécutable : la table des symboles contient des vides. Une fois la fusion des fichiers objets opérée, on connaît les adresses de ces symboles globaux, qui peuvent être traduits : le linker doit donc mettre à jour ces adresses dans le code machine (c'est pour cela qu'on parle d'édition des liens).

Les chargeurs de programme

Au démarrage d'un programme, un morceau du système d'exploitation, le chargeur de programme ou loader, rapatrie le fichier exécutable en RAM et l'exécute. Sur les systèmes sans mémoire virtuelle, le loader vérifie s'il y a suffisamment de mémoire pour charger le programme. Pour cela, il a besoin de savoir quelle est la taille du code machine et de la mémoire statique, ces informations étant mémorisées dans

l'exécutable par le linker.

Loaders absolus

Sur les systèmes d'exploitation mono-programmés, il était impossible d'exécuter plusieurs programmes en même temps. Ces systèmes d'exploitation découpaient la mémoire en deux parties : une portion de taille fixe réservée au système d'exploitation et le reste de la mémoire, réservé au programme à exécuter. Avec cette technique, l'adresse du début du programme est toujours la même : les loaders de ces systèmes d'exploitation n'ont 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).

Quand les programmes consistaient en un seul fichier assembleur de plusieurs milliers de lignes de code, il était possible de les traduire en langage machine et de les lancer directement. Le logiciel d'assemblage était un **assemble-go loader**, un assembleur à une passe qui écrivait le code machine directement en RAM. Il n'y avait alors pas besoin d'édition des liens et le loader était fusionné avec le logiciel d'assemblage.

Loaders à relocation

Avec le temps, les systèmes d'exploitation sont devenus capables de faire résider plusieurs programmes en même temps dans la RAM de l'ordinateur, ce qui demanda au loader de s'occuper de la relocation. Le loader doit donc savoir où sont les adresses dans le fichier exécutable, histoire de les « relocaliser ». Pour cela, il se base sur une table de relocalisation qui indique la localisation des adresses dans le fichier exécutable. Cette table est généralement une copie de la table des symboles, à quelques différences près : pour chaque label, on a ajouté un bit qui indique si l'adresse correspondante doit être relocalisée. Les assembleurs à deux passes peuvent produire une telle table sans aucune modification, ce qui n'est pas le cas des assembleurs habituels à une seule passe.

Avec la multiprogrammation, il est devenu possible de démarrer plusieurs copies d'un même programme simultanément. Dans ce cas, il est possible de partager certains segments du code machine entre les différentes copies du programme. Pour cela, le loader doit savoir où se situent le code machine et les données statiques dans le fichier exécutable, ce qui demande de sauvegarder ces informations dans le fichier exécutable. Ces informations sont mémorisées dans un en-tête, qui mémorise la position et la taille des autres segments dans le fichier (avec souvent d'autres informations).

Sur certains processeurs, la relocation est totalement gérée en matériel, à l'intérieur du processeur. C'est notamment le cas pour les ordinateurs qui utilisent la mémoire virtuelle. Sur ces ordinateurs, les loaders ne font que modifier la table des pages pour faire croire que le fichier exécutable est « swappé » sur le disque dur : le chargement du programme se fera à la demande, quand un chargement d'instruction déclenchera un défaut de page (page miss). Certains processeurs ont tenté d'être encore plus novateurs, en se payant le luxe de supprimer aussi l'étape d'édition des liens. Sur ces processeurs, l'attribution d'une adresse à un objet est déterminée au cours de l'exécution du programme, directement au niveau matériel. Mais ces architectures, dites à arithmétique à zéro adresse (Zero address arithmetic), sont cependant très rares.

Loaders à édition des liens dynamique

Les librairies partagées sont des librairies mises en commun entre plusieurs programmes : il n'existe qu'une seule copie en mémoire, que les programmes peuvent utiliser au besoin. Les librairies statiques ont toujours la même place en mémoire physique : pas besoin de relocation ou de résoudre les labels qui pointent vers cette librairie. Avec les librairies dynamiques, la position de la librairie en mémoire varie à chaque démarrage de la machine : il faut faire la relocation, et résoudre les labels/symboles de la librairie.

Dans certains cas, l'édition des liens s'effectue dès le démarrage du programme, qui doit indiquer les librairies dont il a besoin. Mais dans d'autres cas, le programme charge les librairies à la demande, en appelant le loader via une interruption spéciale : il faut alors passer le nom de la librairie à exécuter en paramètre de l'interruption.

Pour aller plus loin

- Livre Assemblers and loaders, disponible gratuitement via le lien suivant : Assemblers and loaders (http://www.davidsalomon.name/assem.advertis/asl.pdf) ;
- Livre Linkers and Loaders de John R. Levine, publié par by Morgan-Kauffman, (ISBN 1-55860-496-0).