La mémoire

Introduction

La mémoire a toujours été une ressource indispensable d'un système. Elle est partagée entre tous les processus. La mémoire est une suite non structurée d'octets, le système d'exploitation ne connait donc pas la structure des informations en mémoire (qui dépendent de chaque processus).

Importance de la mémoire

Les instructions d'un programme peuvent contenir des adresses mémoires en argument, il est ainsi nécessaire qu'un processus se trouve entièrement en mémoire pour pouvoir s'exécuter. Une gestion efficace de cette mémoire est alors primordiale.

Types de mémoires

Registres

Les registres sont des zones mémoires attachées au processus, leur taille est généralement très réduite (de l'ordre de quelques Ko) mais sont extrêmement rapides, pouvant généralement être accédé en un seul cycle d'horloge du processeur.

Mémoire vive (RAM)

La mémoire vive est la mémoire principale du système. Elle est la ressource importante à gérer et est souvent présente en quantité (entre deux et 64 Go). Elle est standardisée aux normes DDR3 et DDR4. Elle est plus lente par rapport au CPU (il faut parfois plusieurs cycles d'horloge pour récupérer une information), le CPU doit donc attendre ou mettre la donnée en cache. Cependant, cette ressource reste beaucoup plus rapide qu'un disque (SSD ou HDD).

Mémoire cache

La mémoire cache a pour but d'améliorer les performances générales du système en planquant en mémoire plus rapide les données fréquemment demandées.

Il existe trois niveaux de mémoire cache, le premier niveau sert à stocker temporairement des instructions et données. Les deux autres niveaux sont présents entre la mémoire RAM et la mémoire cache de premier niveau.

Mémoire virtuelle

La mémoire virtuelle est une mémoire simulée par le système. Elle utilise le disque dur comme RAM additionnelle. Elle est seulement limitée par la taille du disque dur, mais elle est très lente.

Isolation de la mémoire

Il est essentiel que chaque processus ait une zone mémoire réservée pour éviter des problèmes de sécurité et d'intégrité des données.

Un processus ne peut donc pas écrire dans l'espace mémoire d'un autre. S'il essaye, on aura une "Segmentation Fault". La mémoire n'est de ce fait pas partagée entre les processus sauf lorsque cela est explicitement demandé. (Voir Mémoire partagée pour plus d'informations).

Translation d'adresse

Lorsqu'un programme est chargé, il est placé entièrement en mémoire à une adresse de départ. Son espace d'adressage est défini et les instructions du programme font référence à l'adresse 0.

La translation peut être effectuée à 3 moments :

- À la **compilation** (inexistant aujourd'hui), on met les adresses physiques dans le programme compilé ;

– Au **chargement** (l'adresse de départ est choisie lors du chargement du processus par le système d'exploitation) ;

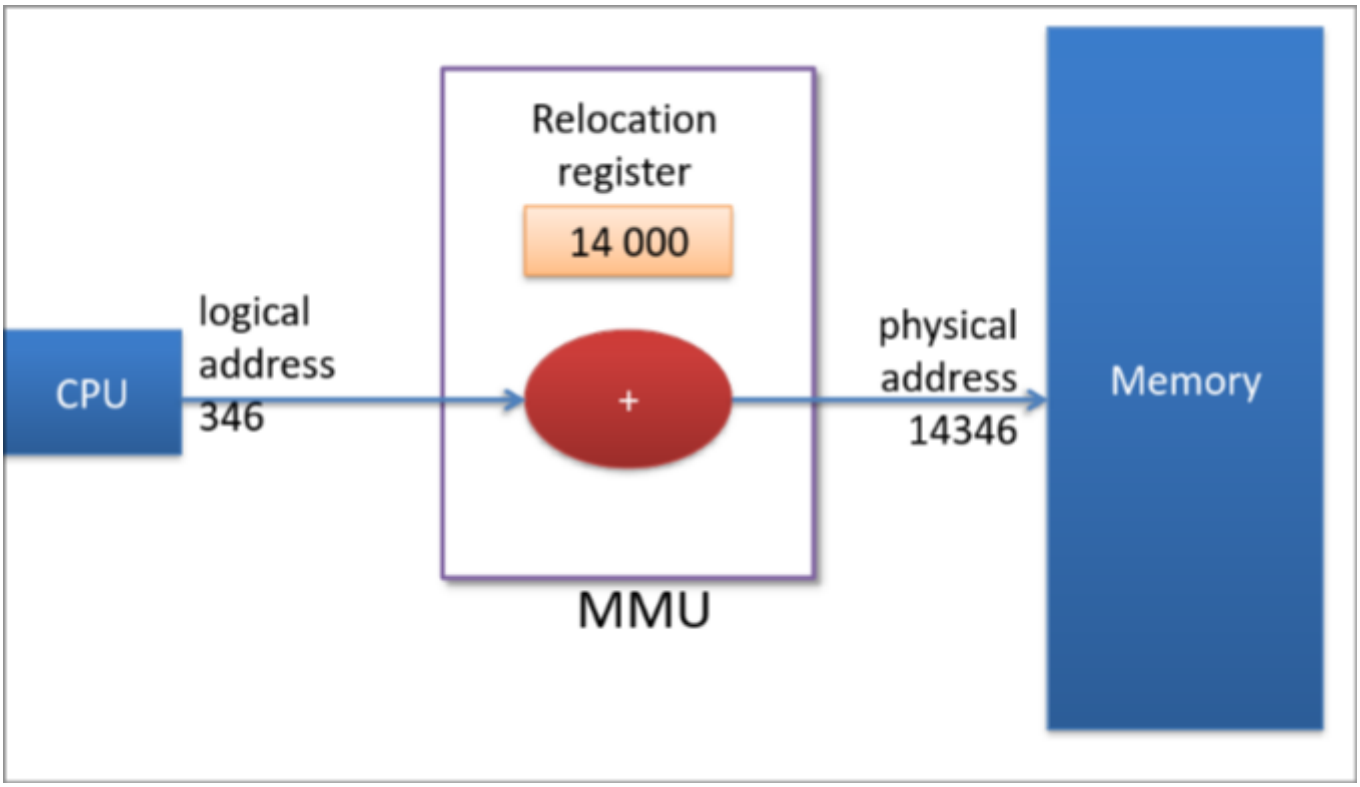
- A l'**exécution**, lors de l'exécution des instructions, cela demande un hardware précis et est utilisé par la segmentation.

Types d'adresses

On parle d'**adresse logique** pour parler de l'adresse présente dans les programmes (qui se réfèrent à l'adresse 0).

On parle en revanche d'**adresse physique** pour parler d'une case mémoire adressable de la mémoire RAM.

La conversion entre l'adresse **logique** et l'adresse **physique** est faite par le **Memory Management Unit (MMU)** qui est un composant matériel spécialisé dans l'opération de translation.


Relocation
register
14 000
CPU
logical
address
346
+
physical
address
14346
Memory
MMU

Allocation

Le système d'exploitation, les processus systèmes et utilisateurs se trouvent dans la mémoire, il faut donc un mécanisme de protection pour isoler les processus. Ce mécanisme, c'est le MMU vu plus tôt.

Mono-programmation

Lorsqu'un seul processus s'exécute à la fois en même temps que le système d'exploitation, c'est le cas sur les systèmes très rudimentaires comme MS-DOS.

Étant donné qu'il n'y a qu'un seul processus à la fois, le système d'exploitation n'a pas beaucoup à faire, car il n'y a pas besoin d'isoler la mémoire (sauf pour la petite partie réservée au système d'exploitation).

Le BIOS réalise une gestion des périphériques.

Multi-programmation

Partitions fixes

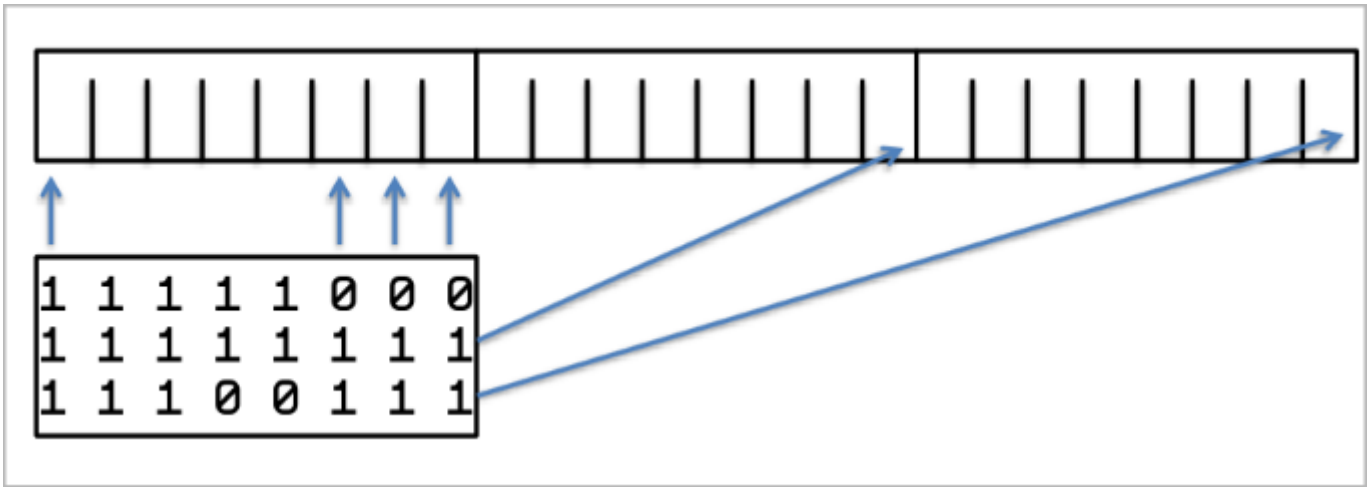
On peut pré-découper la mémoire en morceaux de taille variable (les partitions). On va donc tenter de placer chaque processus dans la plus petite partition qui peut la contenir.

L'un des problèmes de ce système est qu'il va y avoir beaucoup de restes (fragmentation interne) car si un a besoin de quatre unités, mais que la seule partition que l'on peut prendre en contient 8, on a quatre unités non utilisées.

Partitions variables

On alloue l'espace **selon les besoins des processus**, on va créer des partitions en fonction des demandes faites par les processus.

Table de bits



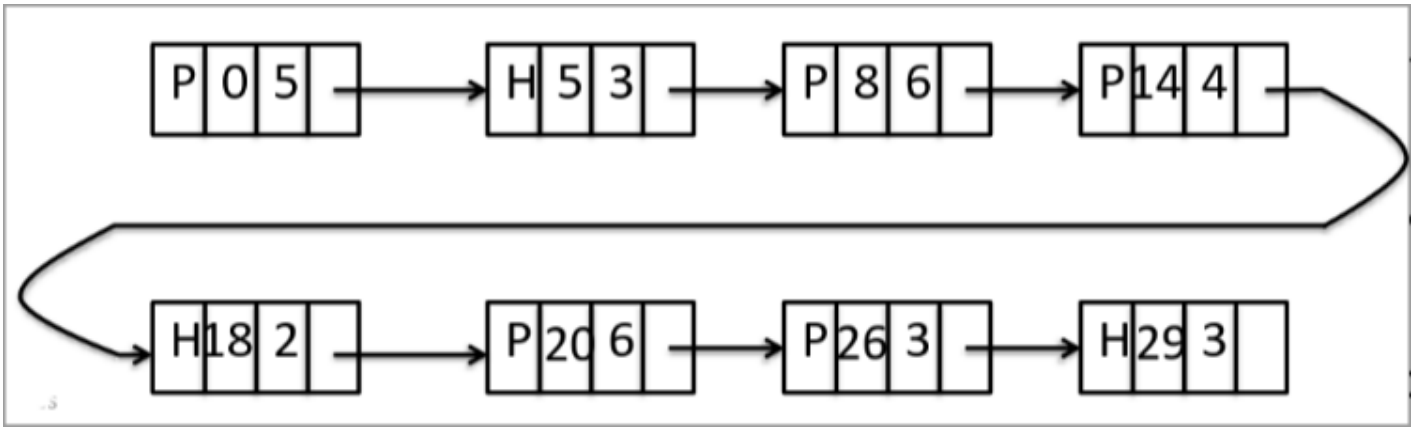
La table de bits correspond à une cartographie de la mémoire découpe en blocs d'allocations de taille fixe. Pour chaque bloc, on va noter un 1 si c'est occupé, ou un 0 si le bloc est libre dans la table de bits.

Ainsi, il suffit de seulement stocker 1 bit par unité d'allocation. Cependant, le problème est que si les unités d'allocations sont petites, alors il la table sera grande, mais ce sera plus précis.

À l'inverse, si les unités d'allocations sont grandes, la table sera plus petite, mais sera moins précise (ce qui implique donc un plus grand gaspillage de mémoire).

Un autre problème est que plus la taille est grande, moins l'allocation sera rapide, car il faudra parcourir toute la table jusqu'à trouver le segment d'unités libre recherché.

Liste chainée



Une autre méthode correspond à conserver une liste chainée des partitions mémoire. Cette liste est généralement triée suivant les adresses des partitions libres, ainsi les partitions qui se suivent dans la mémoire se suivront dans la liste.

Chaque élément de la liste chainée n'a alors qu'à stocker quatre informations : si le bloc est libre ou pas, la position du début de la partition, la longueur de la partition et le lien avec la partition suivante. Par exemple `P86` signifie qu'il y a un processus sur la partition en position huit d'une longueur de six unités.

Ce qui implique également de facilité la fusion de blocs libre, si un programme libère une partition, cette partition pourra être facilement fusionnée avec les éventuelles partitions libres adjacentes pour former une partition plus grande.

L'avantage est de rendre l'allocation beaucoup plus rapide, le désavantage est que cela crée de la fragmentation.

> Pour en savoir plus, vous pouvez regarder sur ce site.

Choix de la partition

Lorsque l'on recherche la partition de mémoire qui pourra accueillir un programme, il y a plusieurs algorithmes possibles.

- L'algorithme **first-fit** qui correspond à prendre la première partition libre trouvée étant suffisamment grande pour contenir le programme. C'est souvent cet algorithme qui est utilisé, car il est très rapide.
- L'algorithme **best-fit** qui correspond à parcourir toute la mémoire à la recherche de la meilleure partition. Cependant, cette méthode est assez peu efficace et crée beaucoup de fragmentation externe en créant des bouts de partitions trop petites pour être utilisées
- L'algorithme **worst-fit** qui correspond à de nouveau parcourir toute la mémoire, mais cette fois-ci à la recherche de la plus grande zone disponible afin d'éviter de créer trop de *fragmentation externe* comme le best-fit. Cet algorithme est particulièrement rapide pour les listes triées par taille.

Fragmentations

- **La fragmentation interne** survient lorsque la mémoire est divisée en unité d'allocation de taille fixe. Elle provient de la mémoire allouée par le SE, mais pas demandée au processus. Le SE alloue donc un surplus de mémoire par rapport à la demande du processus.
- **La fragmentation externe** survient suite aux allocations et libérations successives, la mémoire ressemble alors à un gruyère.

Solutions à la fragmentation

Pour rassembler les zones mémoires et résoudre ce problème de fragmentation, on peut utiliser le **compactage**. Cela consiste à rassembler les zones mémoires occupées et les zones libres ensembles de façon à créer de grands espaces libres.

Cela est cependant uniquement possible dans le cas de la translation à l'exécution et c'est un mécanisme assez couteux en ressource.

Une autre solution est d'utiliser de la **pagination** que nous allons voir en détail dans la section suivante.

Swapping

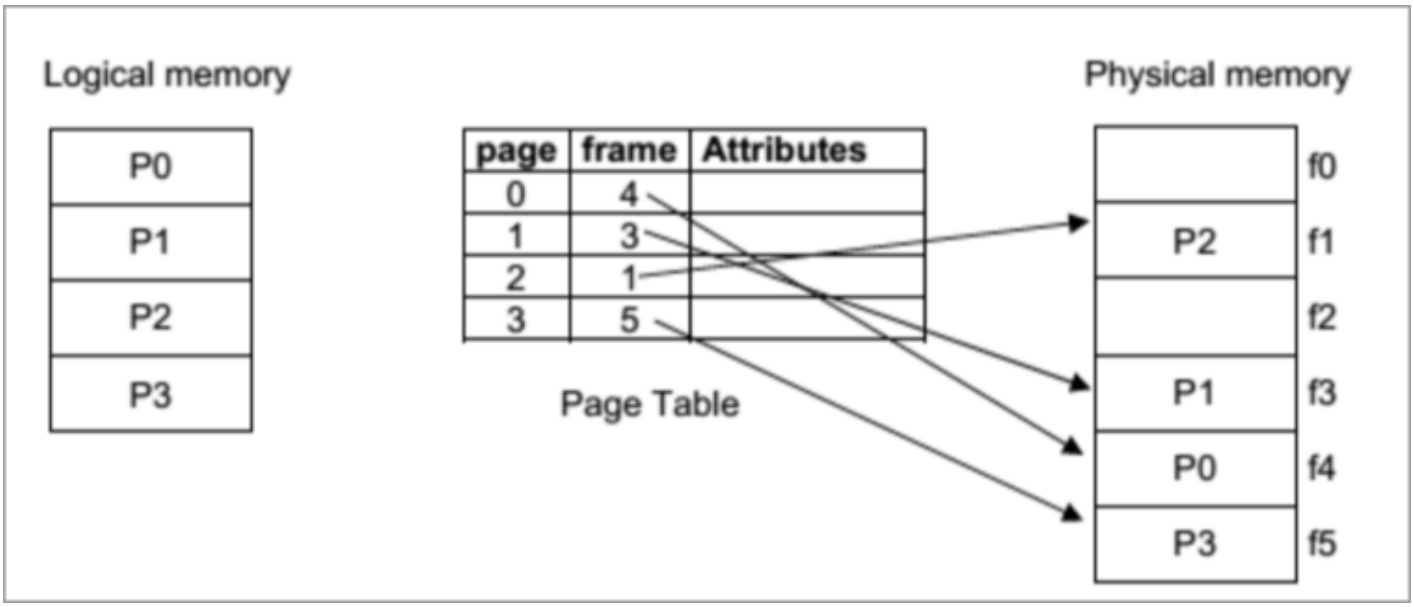
Pour pouvoir s'exécuter un processus doit se trouver entièrement en mémoire, alors lorsqu'il y a beaucoup de processus en mémoire et peu de mémoire disponible, on peut mettre un processus qui ne s'exécute pas (état ready) sur le disque dur pour libérer de la mémoire (via une partition sur le disque dur ou via un fichier "swap file").

Un processus peut être swappé lorsque son quantum de temps a expiré (retour de running à ready), si le quantum est petit il y aura beaucoup de changement de processus et donc beaucoup d'accès au disque en revanche si le quantum est grand, il y aura moins de changements de processus et moins d'accès au disque.

Les performances de ce mécanisme sont de ce fait déterminées par le temps de transfert mémoire ⇆ disque.

On ne peut swap que les processus auquel on n'a pas besoin d'accéder à la mémoire, c'est-à-dire les processus en état **ready** car pour les processus en état **waiting**, le système d'exploitation doit pouvoir accéder à la mémoire du processus pour y faire les entrées-sorties.

La pagination



La pagination permet d'avoir un espace adressage en mémoire physique non contigu. Elle nécessite toute fois une modification du MMU pour intégrer la *table des pages*. Elle va avoir une "table des pages" pour savoir où sont les morceaux. Cette méthode est utilisée par tous les systèmes d'exploitations actuels.

Fonctionnement

On va donc diviser la mémoire logique en blocs de taille fixe (les **pages**), on va faire de même avec la mémoire physique (que l'on va appeler les **frames**). Ainsi une page correspond à une frame. De ce fait, la **table des pages** va lier chaque page à une frame. L'espace de stockage pour le swapping est également découpé en blocs de même taille que les frames.

Lorsqu'un processus démarre, on va donc calculer la taille nécessaire en nombre de pages, ensuite, on va regarder le nombre de frames disponibles. S'il y a suffisamment de frames disponibles le processus peut alors démarrer, sinon le processus ne peut pas démarrer. Par exemple, si on a un processus nécessitant 10 octets et que chaque page fait quatre octets, cela nécessitera 3 pages. Sauf que s'il n'y a que 2 frames disponibles le processus ne pourra pas démarrer.

Les frames sont gérées par le système d'exploitation, ce dernier va garder la liste des frames libres et occupées et le nombre de frames disponibles. Le SE doit aussi mettre en place une protection pour empêcher un processus de dépasser son espace d'adressage.

Effet sur la fragmentation

Ainsi, la fragmentation externe n'existe plus, car bien que l'adressage soit vu comme contigu au niveau du programme, elle ne l'est pas au niveau physique.

La fragmentation interne, elle sera d'en moyenne 1/2 page par processus, comme on ne peut pas assigner des demi-frames, donc si un processus nécessite 10 octets, mais qu'une frame fait 4 octets et que chaque page correspond à une frame (en conséquence chaque page fait 4 octets également), on devra alors assigner 3 pages pour le processus de 10 octets, ce qui laisse ensuite deux octets (soit une demi-page) non utilisée.

La segmentation

Un·e utilisateur·ice voit un programme comme un ensemble d'instructions, de données, de fonctions, etc. Bref, un ensemble de blocs distinct dont les données ne sont pas mélangées avec les autres.

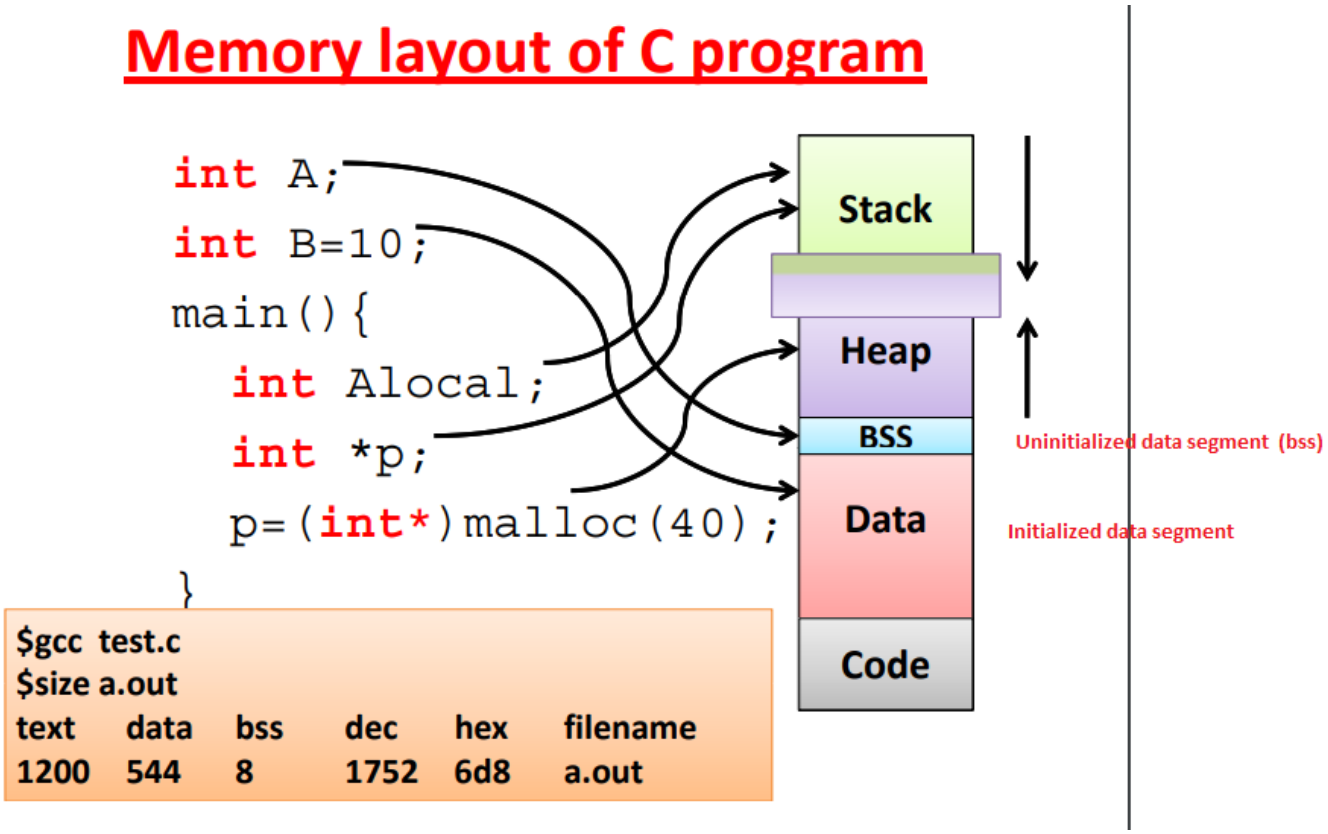
Il convient donc de traduire cette vue à l'intérieur du système d'exploitation. Pour cela, on va diviser l'espace d'adressage du processus en segments. Chaque segment a un nom, un numéro et une taille.

Les différents segments

Ainsi dans une architecture Intel x86, on peut avoir :

- Le **Code Segment (CS)** qui désigne le segment qui contient les instructions du programme (lecture seule)
- Le **Data Segment (DS)** qui contient les données du programme (variables globales, variables `static`)
- Le **Stack Segment (SS)** qui contient la pile du programme (variables locales, appels, etc)
- Le **Extra Segment (ES)** qui est un segment supplémentaire défini par le·a développeur·euse du programme

De manière plus descriptive, on peut définir la mémoire du point de vue d'un programme comme ceci. À noter cependant que, cela est surtout comme ça que fonctionnaient les anciens systèmes, mais comme on va le voir dans la suite ce n'est plus exactement comme ça que cela fonctionne.



Avec la segmentation comme ceci, une adresse logique devient ainsi `<segment-number, offset>`.

Avantages

En utilisant la segmentation, le système offre ainsi une protection adaptée car chaque segment contient des données d'une même nature et ne peuvent donc pas intervenir dans d'autres données. On ne peut pas par exemple accéder aux instructions du programme depuis le *data segment*.

Également, les segments peuvent être partagés entre plusieurs processus (par exemple avec les libraires systèmes) et ainsi faire une économie d'espace.

Désavantages

Cela fait un retour à la fragmentation, étant donné que les segments sont de taille variables, on retrouve de la segmentation externe. C'est pour cela qu'il faut combiner la segmentation avec la pagination.

Segmentation et pagination

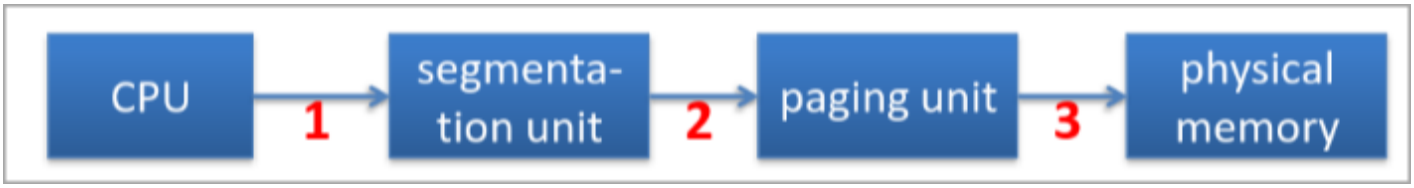
Pour combiner les avantages de la segmentation avec les avantages de la pagination, on va regarder à l'exemple du fonctionnement de l'architecture des processeurs Intel 32 bits.

Ce processeur peut gérer un maximum de 16 384 segments de maximum 4 Go. L'espace d'adressage est découpé en deux parts égales. Une partition pour les segments privés (par exemple, processus) et une autre pour les segments partagés (par exemple, librairie partagée).

Il y a également deux tables, la LDT (Logical Descriptor Table) pour la partition des segments privés, et la GDT (Global Descriptor Table) pour les segments partagés.

Conversion vers adresse physique

Pour passer de l'adresse logique à l'adresse physique, il va falloir passer par deux intermédiaires :



- L'**unité de segmentation** qui transforme l'adresse logique en adresse linéaire
- L'**unité de pagination** qui transforme l'adresse linéaire en adresse physique.

Unité de segmentation

L'adresse logique est composée d'un sélecteur et d'un déplacement. Le sélecteur est lui-même composé de 3 informations :

- Le numéro du segment (noté *s*)
- Si c'est un segment privé ou partagé (LDT ou GDT) (noté *g*)
- Des informations de protection (noté *p*)

À partir de cette adresse logique, l'unité de segmentation construit l'adresse linéaire.

Unité de pagination

L'adresse linéaire est composée de 3 informations,

- Le **directory**, qui indique la table des pages à utiliser parmi le répertoire des tables de pages du processus courant
- La **page**, qui indique la page dans la table des pages
- Le **offset** (déplacement), qui est le même que celui cité dans l'adresse logique.

À partir de cette adresse linéaire, l'unité de pagination construit l'adresse physique comme vu précédemment dans le chapitre sur la Pagination.

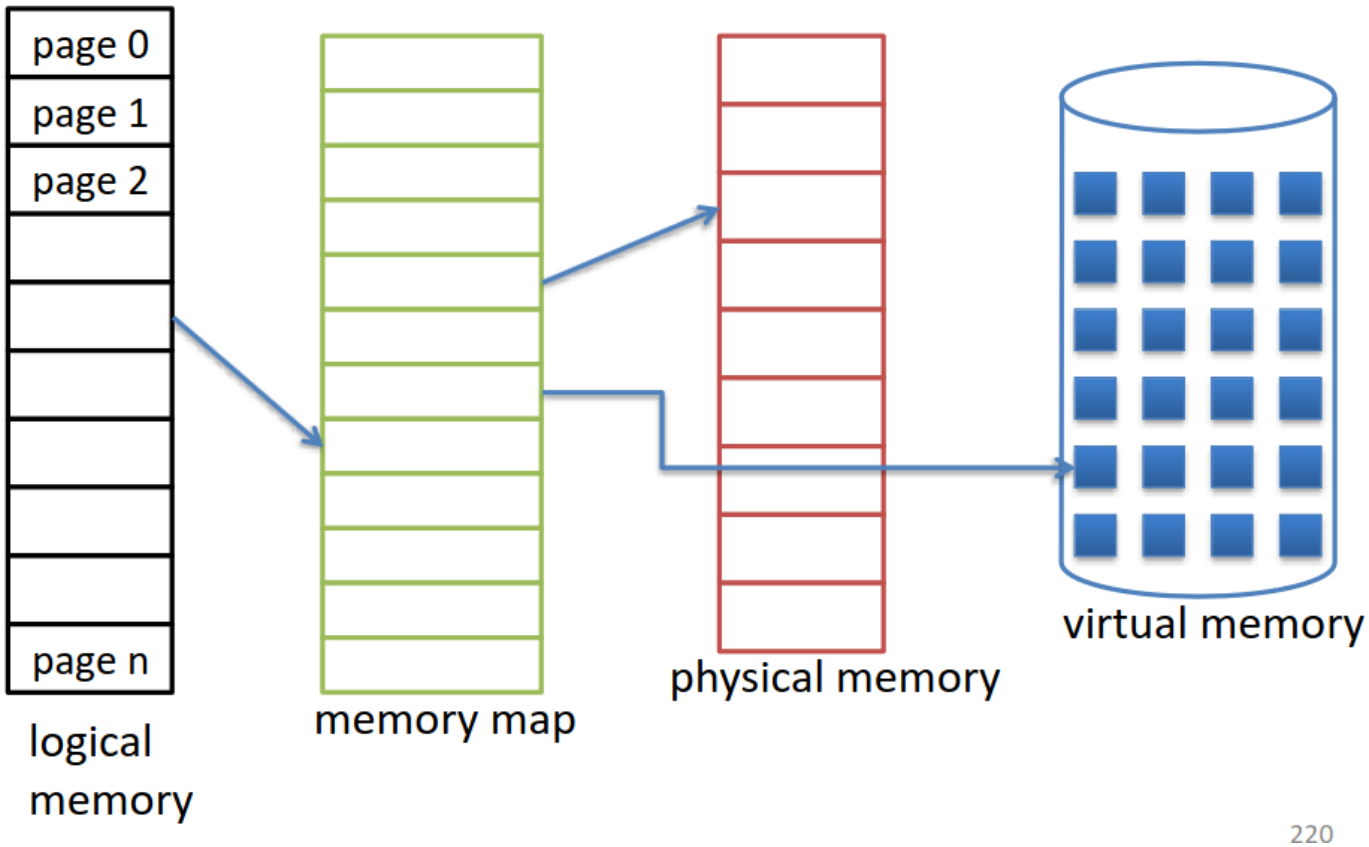
Mémoire virtuelle

Il a été dit précédemment qu'il faut qu'un programme soit entièrement en mémoire pour pouvoir s'exécuter comme processus, alors comment faire lorsque le programme est plus gros que la taille de mémoire vive ?

Ce qui est utilisé par les systèmes aujourd'hui c'est la mémoire virtuelle, il s'agit d'utiliser le disque dur comme mémoire supplémentaire à la RAM.

Cela est bien sûr plus lent, surtout si le stockage est un disque dur (et pas un SSD), mais il permet de faire tourner de très gros programmes (ou juste énormément de processus) avec peu de RAM.

Utilisation de la pagination



220

Cela tombe bien, car la pagination est justement très adaptée à cela, car il suffit de lier de l'espace sur le disque à la table des pages (*memory map* sur le schéma). Ainsi la mémoire virtuelle sur le disque va être divisée en pages.

La table des pages doit bien sûr être adaptée pour pouvoir disposer d'un bit indiquant si la page se trouve sur le disque ou sur la mémoire physique.

Et le stockage/disque doit avoir une partie prévue pour contenir la mémoire virtuelle, celle-ci est appelée "swap device". Il s'agit d'un fichier sous Windows (Pagefile.sys), et d'un fichier ("swapfile") ou d'une partition ("swap partition") sous Linux.

Fonctionnement

Avec la mémoire virtuelle, les pages du processus à exécuter se trouvent sur le disque dur. Lors du chargement du programme, **seules les pages nécessaires** sont placées en mémoire physique.

Le changement entre mémoire virtuelle et mémoire physique est effectué lorsqu'une page non chargée est demandée. Lors du chargement, le système alloue quelques pages et les charge depuis le disque.

Ainsi lorsqu'une page est demandée, on regarde dans la table des pages pour voir si elle est présente en mémoire (bit valide), si oui tout va bien et tout se passe comme d'habitude. En revanche si ce n'est pas le cas (bit invalide) alors, il s'agit d'un **défaut de page**.

Traitement d'un défaut de page

Quand un défaut de page survient, il faut le résoudre le plus vite possible pour ne pas impacter les performances du système :

1. On vérifie si la page demandée est présente dans l'adressage du programme, si ce n'est pas le cas alors c'est un *memory overflow* ce qui provoque un arrêt du processus.
2. Si la page existe, ainsi on trouve une frame libre en mémoire physique et récupère la page depuis le disque
3. On ajuste la table des pages pour indiquer la page comme valide et indiquer sa position dans la mémoire physique
4. On redémarre le processus à l'instruction qui a causé l'erreur

Les défauts de pages faisant des appels d'entrée-sortie au disque, il est important d'en avoir le moins possible afin de ne pas trop impacter les performances du système.

Que faire quand il n'y a plus de frames ?

Lorsqu'un défaut de page survient, mais qu'il n'y a plus de frame disponible sur la mémoire physique, il faut alors que le système en libère une.

Pour cela, il faut que le système choisisse quelle frame remplacer, l'écrivent sur le disque et mettent à jour la table afin de libérer une frame pour résoudre le défaut de page.

Algorithmes de choix des pages victimes

Il est important d'utiliser un bon algorithme pour choisir les pages qui seront mises en mémoire virtuelle, car sinon cela risque d'augmenter le nombre de défauts de page et donc de considérablement ralentir le système.

Voici une comparaison de plusieurs algorithmes :

- **FIFO** (First In, First Out) la page qui est sélectionnée est la page la plus ancienne. Cependant, la performance de cette méthode n'est pas très bonne, car ce n'est pas parce qu'une page est vieille qu'il ne faudra pas y accéder souvent.
- **OPT** (Remplacement Optimal), cela consiste à essayer d'avoir le taux de faut de page minimal en éliminant la page qui ne sera plus nécessaire avant longtemps. Cependant, ce mécanisme est dur à implémenter, car on ne peut pas savoir en avance ce qui sera nécessaire. Ce mécanisme sert uniquement de base théorique aux autres systèmes.
- **LRU** (Least Recently Used) consiste à choisir la page la moins récemment utilisée. Pour implémenter cela, il faut soit y ajouter une timestamp (ce qui rendra la chose moins efficace), ou alors mieux utiliser un système de pile, ainsi dès qu'une page est utilisée, on la met en haut de la pile, comme ceci, on sait que la page la moins utilisée sera forcément la dernière de la pile.
  - **LRU approximé** est une des variantes du LRU. L'idée est de définir un bit de référence pour chaque page. Initialement le bit est à 0 pour tous, dès qu'une page est accédée, le bit est mis à 1. Lorsque tous les bits sont à 1, on met tout à 0 sauf la dernière. Ainsi lorsqu'un défaut de page survient, la page victime sera la première page avec un 0 (vous pouvez trouver plus d'information dans cette vidéo).
  - **LRU avec octet de référence**, l'idée ici est d'utiliser la méthode précédente, mais à intervalle régulier collecter le bit de référence pour le placer dans un octet de référence. Ainsi lorsqu'il faut choisir quel page sera la victime, il suffit de prendre celle qui a l'octet de référence le plus bas. Et dans le cas où il y en a plusieurs avec la même valeur, prendre le premier.
  - **Algorithme de la seconde chance**, l'idée est de stocker 2 bits par page, une pour le bit de référence (voir LRU approximé) et l'autre pour un bit de modification. Le bit de modification est mis à un si la page a été modifiée sur la mémoire physique, mais pas encore sur la mémoire virtuelle. À l'inverse, elle est mise à 0 si elle est égale à la mémoire virtuelle. Ainsi, on peut savoir si une page a été utilisée et si elle nécessite une écriture sur le disque. Le meilleur choix étant de choisir une page qui n'a pas été utilisée récemment ET qui n'a pas été modifiée.
- **LFU** (Least Frequently Used) consiste à compter le nombre de fois qu'une page est utilisée et à choisir la page avec le plus petit compteur. Le problème est que ce n'est pas par ce qu'une page a été très utilisée qu'elle le sera toujours. À l'inverse ce n'est pas par ce qu'une page n'a pas beaucoup été utilisée qu'elle ne va pas le devenir.
- **MFU** (Most Frequently Used) consiste, elle aussi, à compter le nombre de fois qu'une page est utilisée et à choisir la page avec le plus *grand* compteur en réponse à l'observation statistique du LFU. Cependant, ces deux algorithmes sont peu utilisés, car ils sont peu efficaces.

Amélioration des performances

Écriture retardée des pages

Pour améliorer les performances en cas de défaut de page, on peut garder un ensemble de page (appelée **pool**) qui sont toujours immédiatement disponibles. Ainsi, lorsqu'une page victime nécessite d'être sauvegardée (bit de modification à 1 dans l'algorithme de la seconde chance), il suffit de donner une page du pool au processus qui est donc utilisable directement. Ensuite, on sauvegarde la page victime dans la mémoire virtuelle et cette page intègre ensuite le pool.

Allocation des frames

Il faut avoir une allocation raisonnable de la mémoire physique (frames), si trop de frames sont alloués au processus, on risque de tomber vite à court de frame et ainsi faire beaucoup de défauts de pages. Si trop peu de mémoire frames sont allouées, alors on tombe vite dans des défauts de page.

Alors il faut trouver un moyen d'allouer les frames de manière à satisfaire le plus possible tous les processus.

- La première méthode est celle de l'**allocation équitable**, on divise le nombre de frames disponible par le nombre de processus actifs. Cette méthode n'est pas intéressante, car cela créera un gaspillage pour les petits processus et une mauvaise performance pour les plus gros.
- La deuxième méthode est celle de l'**allocation proportionnelle**, les frames sont allouées proportionnellement à la taille du processus. Il faut toute fois tenir compte de la multiprogrammation, si de nouveaux processus sont créé le nombre de frames par processus va diminuer et si des processus se terminent les frames sont de nouveau disponibles. Il faut aussi tenir compte de la priorité des processus, un processus de haute priorité doit s'exécuter vite, plus il a de ressources plus, il va s'exécuter vite et donc plus vite ces frames seront libérées.

Trashing

L'allocation des frames est ainsi un problème compliqué, si on n'alloue pas assez de frame à un processus, le processus va passer beaucoup de temps à transférer des informations. Le **trashing** c'est lorsqu'un processus n'a pas suffisamment de frames allouées pour s'exécuter correctement et passe ainsi plus de temps à récupérer des pages qu'à s'exécuter.

La cause de ce problème est que, comme on a vu avec les processus, pour utiliser au mieux le CPU lorsque ce dernier est peu utilisé, le système va augmenter le degré de multi-programmation, ce qui va diminuer le nombre de frames disponibles et donc augmenter le nombre de défauts de pages. Sauf que pour résoudre le défaut de page les processus concernés devront être mis dans

l'état **waiting** en attente de l'entrée-sortie, ce qui va ainsi de nouveau réduire l'utilisation du CPU. Ainsi le cycle vicieux se répète.

Modèle de la localité

Une manière de résoudre ce problème est d'utiliser le **modèle de la localité**.

Une localité, c'est un ensemble de pages qui sont activement utilisées ensemble. Un programme est ainsi composé de plusieurs localités différentes qui peuvent se chevaucher. Lorsqu'un processus s'exécute, ce dernier va de localité en localité.

Par exemple, lorsqu'une fonction est appelée, cela définit une nouvelle localité et quand l'on quitte cette fonction, on entre dans une autre localité.

Ainsi si on arrive à identifier la localité en cours, on peut alors charger suffisamment de frames pour les accommoder, cela va créer des défauts de pages jusqu'à ce que toutes les pages de la localité jusqu'à ce qu'elle soit toutes en mémoire, ensuite elles ne feront plus aucun défaut de page jusqu'à ce qu'on change de localité.

Si on n'alloue pas suffisamment de frames par rapport à la localité courante le système va faire du trashing.