

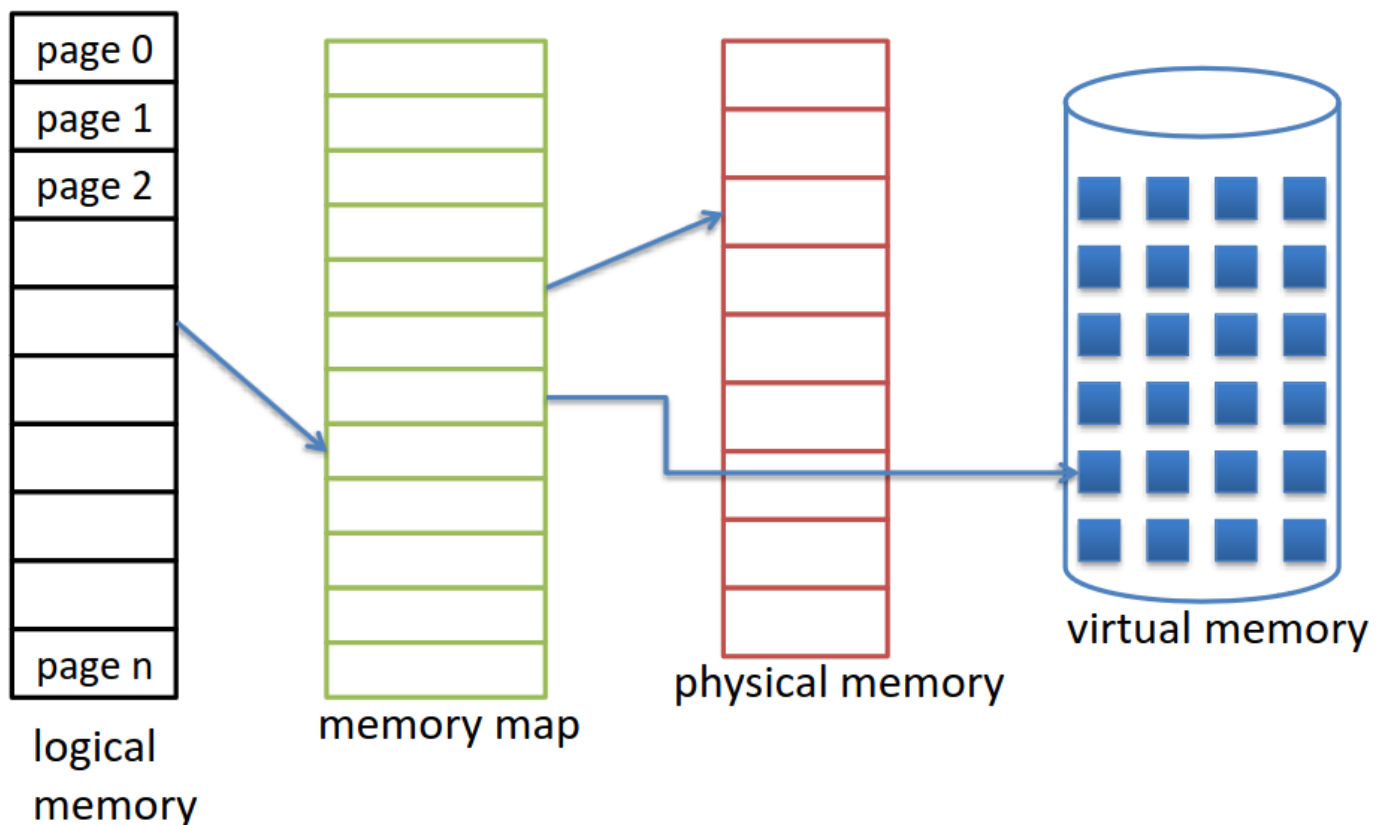
# Mémoire virtuelle

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 défaut 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 parce qu'une page a été très utilisée qu'elle le sera toujours. À l'inverse ce n'est pas parce 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.

---

Revision #2

Created 3 January 2024 17:27:45 by SnowCode

Updated 6 January 2024 19:13:15 by SnowCode