Mémoire virtuelle

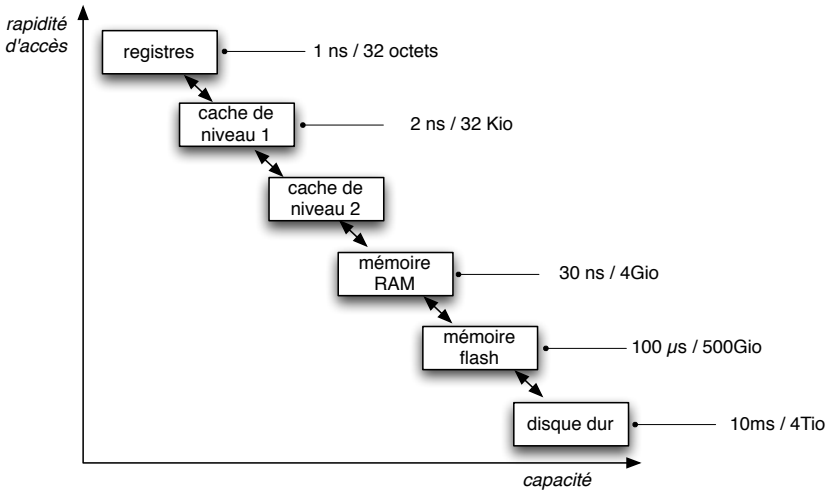
Thèmes traités
- principe de localité et hiérarchie de mémoires
- mécanisme de pagination sous Linux
- couplage de fichiers en mémoire : principe et API Unix (`mmap`...)

1 Principe de localité

Question. *Quel temps fera-t-il demain ?*

1.1 Politiques de choix des pages à remplacer

Diffférents types de mémoire sont disponibles pour conserver les données nécessaires à l'exécution d'un programme. Ces différents types de mémoire se distinguent par leurs caractéristiques de capacité et de temps d'accès qui permettent de les classer selon une **hiérarchie** :



L'idéal serait d'avoir **une mémoire ayant la capacité du disque et la rapidité des registres**. Pour s'approcher de cet idéal, on peut remarquer que seules les données et instructions nécessaires à l'exécution de la prochaine opération doivent être présentes en mémoire rapide. La **hiérarchie de mémoires** va donc être gérée en essayant
- d'amener au niveau supérieur les données et instructions nécessaires aux prochains calculs à effectuer ;
- de renvoyer au niveau inférieur les données et instructions inutiles pour les calculs à venir.

En ce qui concerne la pagination, cette gestion d'une hiérarchie de mémoires est mise en œuvre par le mécanisme de défaut de page (qui déclenche le chargement de pages en RAM depuis le disque) combiné avec les algorithmes de remplacement, qui déterminent les pages RAM à éjecter (et sauvegarder sur disque). Cette section étudie les algorithmes de remplacement, ainsi que l'utilisation du principe de localité comme heuristique pour guider les transferts au sein de la hiérarchie de mémoires.

1.2 Un exemple simple : FIFO

Les pages sont éjectées dans l'ordre chronologique où elles ont été chargées.

Exemple (et Question) : On suppose que l'on dispose d'une mémoire de 4 cases.
Déterminer le nombre de chargements de pages provoqué par la suite de demandes des pages suivantes : 1,2,3,4,1,2,5,1,2,3,4,5

Anomalie de Belady . Intuitivement, on peut s'attendre à ce que, plus l'on dispose de mémoire, moins il est nécessaire d'éjecter des pages. Cette intuition est correcte pour la plupart des politiques de remplacement de pages, mais pas pour la politique FIFO. La suite d'accès précédente fournit un contre-exemple

Question. *Déterminer le nombre de chargements de pages provoqué par cette suite, en supposant que l'on ne dispose que de 3 cases.*

1.3 LRU

1.3.1 Algorithme optimal

L'algorithme de remplacement optimal consisterait à éjecter (au besoin) la page qui sera demandée dans le futur le plus éloigné : on assure ainsi que les prochaines pages utilisées restent présentes en mémoire. L'inconvénient de cet algorithme est bien sûr que l'on connait rarement par avance quelle sera la page qui sera demandée dans le futur le plus éloigné.

La **politique LRU** consiste à éjecter (au besoin) la moins récemment accédée (Least Recently Used)

Question. *La politique LRU est une approximation heuristique de la politique optimale. Pourquoi ?*

1.3.2 LRU n'est qu'une heuristique...

La politique LRU peut être mise en défaut lorqu'une exécution ne respecte pas le principe de localité, c'est-à dire lorsque les pages demandées sont toutes différentes et dépassent la capacité de mémoire disponible. L'exercice suivant illustre l'importance de connaître les limites de cette heuristique, et de connaître la manière dont les données sont représentées en mémoire, dès lors que l'on considère l'**efficacité** des applications que l'on conçoit.

Exercice On suppose que l'on dispose d'un mécanisme de mémoire virtuelle, avec chargement de pages à la demande, dont la politique de remplacement (choix de pages victimes) est LRU.

Le programme suivant initialise à zéro un tableau A, que l'on suppose totalement absent de la mémoire centrale au départ de l'exécution. On suppose qu'un entier est représenté sur 4 octets, que la taille d'une page est 4 Kio, que la taille de la mémoire physique est de 4092 Kio, et que les matrices sont rangées par lignes (c'est-à-dire que A[i][j] et A[i][j+1] sont rangés à des adresses successives)

```
#define DIM 1024
void main(int argc, char* argv[]) {
   int A [DIM][DIM];
   for (int i=0 ; i<DIM; i++) {
    for (int j=0 ; j<DIM; j++) { A[i][j] = 0; }
   }
 }
```

- Combien de chargements de page engendre l'exécution de ce programme ?
- Si l'on remplace, dans ce programme, "`A[i][j]`" par "`A[j][i]`", combien de chargements de page sont alors engendrés ? Commentez.
- Même question, en supposant maintenant que la taille de la mémoire physique est de 4 Mio.

2 Couplage de segments mémoire sous Unix

Le système Linux offre un ensemble de primitives permettant de rendre directement accessible une plage d'adresses virtuelles, en associant à cette plage d'adresses un segment mémoire partageable et/ou un fichier. Dans ce dernier cas, le contenu du fichier peut alors être lu et/ou modifié par simple lecture/écriture mémoire dans le segment correspondant.

L'utilisation de ces primitives nécessite le fichier de définitions `#include <sys/mman.h>`. Le fichier `#include <unistd.h>` peut être utile pour certaines fonctions annexes telles que sysconf.

2.1 API Unix de gestion du couplage

2.1.1 Primitive mmap

Cette primitive rend utilisable (accessible) un segment de mémoire virtuelle avec des droits d'accès spécifiés (lecture, écriture, exécution) en lui couplant un espace mémoire ou un contenu présent dans un fichier.

```
void *mmap(void *addr, size_t len, int prot, int flags,  int fildes, off_t off);
```

Résultat La primitive retourne l'adresse (virtuelle) du début du segment mémoire adressable en cas d'exécution correcte. En cas d'erreur, la valeur usuelle (-1) [1] est renvoyée.

Signification des paramètres

- Le paramètre addr indique l'adresse souhaitée de début du segment. La primitive prend cette information comme une indication. Par défaut, la valeur 0 laisse le choix libre pour la primitive (meilleure solution, sauf cas très spécifique) ;
- Le paramètre len indique la taille du segment en octets. Cette taille ne doit pas être inférieure à celle d'une page de mémoire virtuelle [2].
- Le paramètre prot précise les droits d'accès permis sous la forme de constantes :
  - PROT_READ : autorise la lecture ;
  - PROT_WRITE : autorise l'écriture ;
  - PROT_EXEC : autorise le branchement pour exécution ;
  - PROT_NONE : interdit tout accès.
- Le paramètre flags précise la gestion du contenu du segment couplé. En particulier, le contenu du segment de mémoire virtuelle peut être ou non partagé entre plusieurs processus (père et fils par exemple) ou privé à chaque processus.
  - MAP_SHARED : le contenu du segment de mémoire virtuelle est partageable ;
  - MAP_PRIVATE : le contenu du segment de mémoire virtuelle est privé ;
  - MAP_FIXED : le premier paramètre addr est considéré comme l'adresse obligatoire de début du segment ; Si cette adresse n'est pas compatible avec l'état courant de la mémoire virtuelle du processus, la primitive peut donc échouer (`errno` EINVAL).
  - MAP_NORESERVE : pas de préréservation d'espace de swap ;
  - MAP_ANON : couplage d'un contenu à la demande. Autrement dit, le segment n'est pas couplé à un fichier existant.
  - MAP_ALIGN : le premier paramètre addr doit être une adresse multiple de la taille de page.
- Le paramètre fildes indique le canal (descripteur) d'accès à un fichier ouvert qui représente le contenu du segment couplé. Si le segment n'est pas couplé à un fichier (option MAP_ANON du paramètre précédent), ce paramètre doit avoir la valeur (-1).
- Le paramètre off précise le début de l'espace du fichier couplé au segment. Si la somme off+len dépasse la taille du fichier, les références au-delà de la limite fixée par la taille du fichier auront un effet non défini, mais qui s'avère être très souvent la réception d'un signal SIGBUS ou SIGSEGV...

1. Plus rigoureusement : `MAP_FAILED`, qui est un mnémonique pour `(void *) -1`
2. La fonction sysconf fournit la valeur de diverses constantes et options de configuration système, dont la taille d'une page (`sysconf(_SC_PAGESIZE)`)

Exemples

- Couplage d'un segment ayant la taille d'une page à un fichier existant ouvert en lecture/écriture. Le segment est accessible en lecture et écriture et partageable. Le contenu du segment est le contenu des *pagesize* premiers octets du fichier.
```
long pagesize = sysconf(_SC_PAGESIZE);
int cf = open("contenu",O_RDWR);
char* base = mmap(0, pagesize, PROT_WRITE|PROT_READ, MAP_SHARED, cf ,0);
/* l'espace d'adresses [base,base+pagesize[ est accessible en lecture/écriture */
```
- Couplage d'un segment ayant la taille d'une page sans contenu initial. L'accès à un tel segment doit d'abord être en écriture pour y placer un contenu significatif. Le contenu du segment peut être partagé et accédé en écriture et lecture.
```
long pagesize = sysconf(_SC_PAGESIZE);
char* base = mmap(0, pagesize, PROT_WRITE|PROT_READ, MAP_SHARED|MAP_ANON, -1, 0);
```

2.1.2 Primitive munmap

Cette primitive permet de découpler totalement ou partiellement un segment.

```
int munmap(void *addr, size_t len);
```

Résultat La primitive retourne 0 en cas d'exécution correcte. En cas d'erreur, la valeur (-1) est renvoyée.

Signification des paramètres
- Le paramètre addr fixe l'adresse de début de la zone à découpler. Cette adresse de début doit être sur une frontière de page ;
- Le paramètre len fixe la longueur de la zone à découpler et sera arrondi à un multiple de la taille d'une page par excès.

2.1.3 Primitive mprotect

Cette primitive permet de modifier totalement ou partiellement les droits d'accès à un segment.

```
int mprotect(void *addr, size_t len, int prot);
```

Résultat La primitive retourne 0 en cas d'exécution correcte. En cas d'erreur, la valeur usuelle (-1) est renvoyée. La primitive peut échouer si le processus n'a pas les droits d'accès suffisants sur l'objet couplé au segment (par exemple, fichier ouvert en lecture seule et tentative de donner le droit d'écrire des pages).

Signification des paramètres
- Le paramètre addr fixe l'adresse de début de la zone dont les droits d'accès vont être modifiés. Cette adresse de début doit être sur une frontière de page ;
- Le paramètre len fixe la longueur de la zone concernée et sera arrondi par excès à un multiple de la taille d'une page.
- Le paramètre prot précise les nouveaux droits d'accès demandés avec les même constantes symboliques que pour la primitive mmap.

2.2 Exercices

1. *(Fonctionnement d'un segment partagé, couplage segment/fichier)*
   Ecrire un programme qui exécute l'algorithme suivant :
   – Créer un fichier contenant deux pages de caractères 'a' ;
   – Ouvrir de ce fichier en lecture/écriture ;
   – Coupler un segment de taille 2 pages à ce fichier en mode partageable et lecture/écriture ;
   – Créer un processus fils ;
   – Pour le processus fils, attendre 2 secondes et lister les 10 premiers octets de chaque page du segment puis remplir la première page de caractères 'c' et terminer.
   – Pour le processus père, remplir la deuxième page du segment de caractères 'b' et attendre la fin du fils. Lister alors les 10 premiers octets de la première page et terminer.

   (a) Quel va (vraisemblablement) être l'affichage observé ?

   (b) Quel va être le contenu du fichier ?

2. Ecrire une implémentation de la commande `cp f1 f2` basée sur le couplage mémoire.

3. *(Contrôle des accès aux pages d'un segment)*
   Ecrire un programme qui
   – couple, en mode anonyme, une zone d'adressage de la taille d'une page en interdisant tout accès.
   – puis provoque l'exception de violation mémoire en tentant un accès en écriture dans cette zone.
   Modifier ce programme en associant au signal SIGSEGV un traitant qui change la protection en utilisant la primitive mprotect de façon à permettre l'écriture.

4. On considère le programme suivant :

```
#include <stdio.h>
#include <setjmp.h>
#include <sys/mman.h>
#include <sys/signal.h>

char    *base;      // adresse de base de page
long    pagesize;   // taille d'une page
jmp_buf env;        // zone de sauvegarde de point de reprise

void traitant (int quelsignal) { ... }   // traitant du signal SIGSEGV

int main ( int argc, char *argv[]) {   // programme principal
  mon_action.sa_handler = traitant;
  sigemptyset(&mon_action.sa_mask);
  mon_action.sa_flags = 0;

  pagesize = sysconf(_SC_PAGESIZE);
  sigaction(SIGSEGV, &mon_action, NULL);
  setjmp(env);
  printf("base[0] = %c\n",base[0]);
}
```

   Que devrait faire le traitant du signal pour que finalement l'instruction printf soit exécutable ?

3 Etude de cas : mémoire virtuelle sous Linux

Le système Linux comporte un sous-système de gestion de mémoire virtuelle de type page à la demande. L'espace d'adressage virtuel d'un processus est de 4 Giga-octets, les adresses étant sur 32 bits [3]. À partir d'une image exécutable d'un programme (voir figure 1), en format ELF (Executable and Linking Format), le noyau Linux place les différentes sections de cette image dans des segments de la mémoire virtuelle associée au processus. Un segment a une taille qui est un multiple de pages.

On distingue 3 types de fichiers binaires :
- Les fichiers binaires rééditables qui peuvent être (et doivent être) assemblés avec d'autres pour constituer un fichier binaire exécutable final ;
- Les fichiers binaires exécutables à partir desquels le noyau de gestion de processus construit une image mémoire du programme à exécuter par le processus ;
- Les fichiers objets partagés qui permettent une édition de liens dynamique et utilisés pour les librairies partagées.

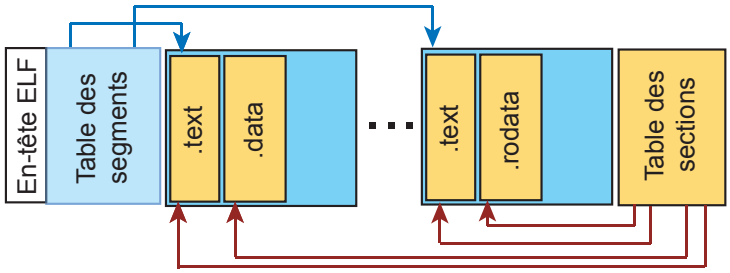


Figure 1 – Le format binaire ELF

Une image exécutable comporte au moins :
- une section de code (appelée text) ;
- une section de données globales statiques (appelée data) ;

auxquelles s'ajoutent des sections utilisées pour l'édition des liens : section de relocation, section de définition de symboles globaux, etc D'autres sections peuvent être aussi présentes comme par exemple des sections couplées à des bibliothèques partagées.

Ces sections seront placées (représentées) dans des segments d'une mémoire virtuelle. Un segment de mémoire virtuelle contient une ou plusieurs sections. Un segment de pile nécessaire pour l'exécution est aussi alloué. Autrement dit, une image exécutable couplée dans une mémoire virtuelle est composée d'un ensemble de segments contenant des sections de code ou de données, ainsi qu'un segment de pile.

3.1 La structuration de la mémoire physique

La mémoire physique est décomposée en 3 types distincts de zones :
- La zone d'entrées/sorties (type ZONE_DMA) : en général les 16 premiers Mo ;
- La zone normale (type ZONE_NORMAL) : de 16 Mo à au plus 896 Mo ;
- La zone haute (type ZONE_HIGHMEM) de 896 Mo à 1Go ;

Pour la gamme de processeurs x86, le découpage en terme d'adressage est le suivant :
- ZONE_DMA : Les 16 premiers Mo
- ZONE_NORMAL : De 16Mo à 896Mo
- ZONE_HIGHMEM : de 896 Mo à la fin

Par ailleurs, la mémoire est structurée en *nœuds* qui regroupent une zone de chaque type, chaque nœud correspondant physiquement à des bancs mémoires distincts pour des machines à architecture mémoire non uniforme (NUMA). Pour une machine à architecture memoire uniforme (UMA), il existe un seul *nœud*.

3. Des extensions existent actuellement sur la famille des processeurs Intel x86 dans leur version à mots de 64 bits.

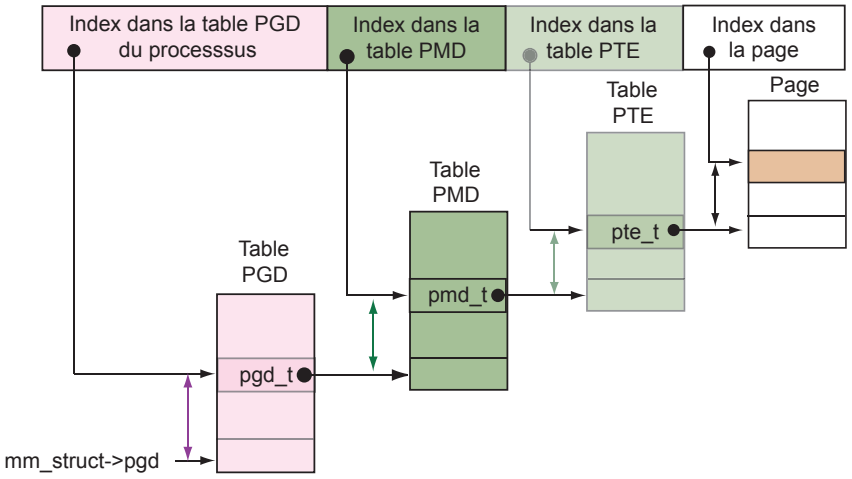


Figure 2 – Structure d'une adresse virtuelle

3.2 La structuration de la mémoire virtuelle

Linux distingue jusqu'à trois niveaux de segmentation-pagination. Une adresse virtuelle est donc structurée en quatre zones « index » (*offset*) comme dans la figure 2, chaque index, excepté la dernière, faisant référence à un index dans une table avec, de gauche à droite :

- La table globale de pages : *Page Global Directory* (PGD) /* assimilable à des super-segments */
- La table intermédiaire de pages : *Page Middle Directory* (PMD /* assimilable au niveau segment */
- La table des pages : *Page Table Entries* (PTE).

La dernière zone (de poids faible) indique classiquement un déplacement dans une page.

3.2.1 Implantation du noyau en mémoire virtuelle

La figure 3 décrit le placement de la zone noyau dans l'espace virtuel. Les trois premiers Giga-octets sont réservés à l'espace virtuel utilisateur. Le dernier Giga-octet constitue l'espace réservé au noyau.

La zone virtuelle noyau est elle-même découpée en régions :

- Une première région accueille le binaire exécutable du noyau (chargeable à partir d'une image disque ISO par exemple). Classiquement, cette zone a une taille de 8Mo. Le code du noyau est en fait édité (au sens édition de liens) pour être exécuté en 0x0C0100000 (3Go+1Mo).
- Une deuxième région contient la table des descripteurs de pages réelles. La taille de cette table dépend évidemment de la taille de la mémoire réelle (du nombre de pages de mémoire réelle).
- Les autres zones sont utilisées par le noyau pour gérer dynamiquement l'allocation de mémoire dont il a besoin, excepté la dernière réservée pour la programmation des interruptions, liées aux entrées/sorties notamment, sur multiprocesseurs et spécifiques aux architectures Intel (APIC : Advanced Programmable Interrupt Controller).

Un gap de 2 pages est systématiquement introduit entre les régions en rouge (ou noir) sur la figure 3.

L'espace noyau est couplé à la mémoire réelle à partir de l'adresse 0x0. Autrement dit, à une adresse virtuelle $@_v$ correspond une adresse réelle obtenue par soustraction de 3Go : $@_v - 0x0C0000000$. La conversion inverse (par addition) est immédiate. On peut ainsi coupler le premier Giga de mémoire réelle à l'espace noyau.

3.2.2 Principes de base du mécanisme de pagination

La structuration de la mémoire virtuelle proposée correspond, dans la terminologie classique, à deux niveaux de segmentation. En fait, pour les processeurs de la famille x86 à 32 bits, le niveau intermédiaire n'existe pas.

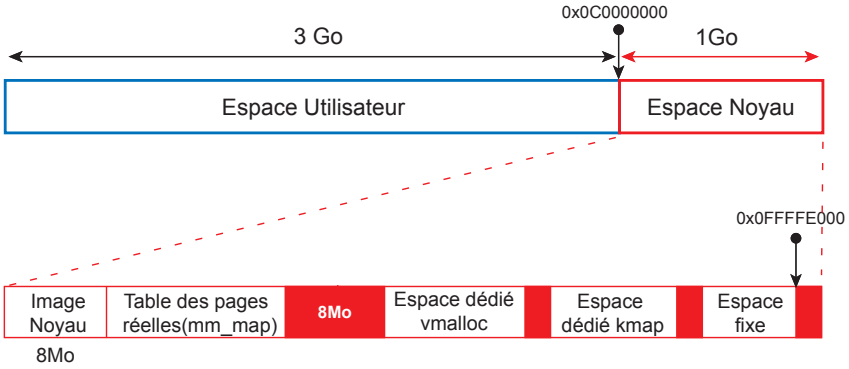


Figure 3 – Implantation du noyau en mémoire virtuelle

3.2.3 Gestion de la pagination à la demande

Un système de gestion de mémoire virtuelle utilisant la stratégie d'allocation des pages à la demande alloue les pages réelles au fur et à mesure des références mémoires faites par tout processus.

Le système de pagination s'appuie sur un processus démon pour préempter des pages virtuelles lorsque le nombre de pages réelles libres devient trop bas [4] . Ce processus appelé kswapd est réveillé lorsque des paramètres de mesure de l'état d'allocation des pages réelles vérifient certains prédicats d'alerte. Trois paramètres pour chaque zone, appelés watermarks, sont opérandes de ces prédicats :

- pages_min : fixé initialement par rapport au nombre de pages libres initial (valeur comprise entre 20 pages et 255 pages) ;
- pages_low : par défaut, fixé au double de page_min ;
- pages_high : nombre minimum de pages devant être libres après l'intervention du démon (détermine l'arrêt du démon).

Principe algorithmique Lorsque pages_low est atteint, l'allocateur réveille le démon. Si le nombre de pages libres continue à baisser et si le nombre de pages libres atteint pages_min, le démon passe en phase d'exécution « urgente » . Le démon se rendort lorsque l'on dispose à nouveau d'au moins pages_high pages libres.

Lorsqu'un processus s'exécute, il référence deux types de pages :

- les pages couplées à un contenu en lecture seule telles que les pages de code ou de librairies partagées. Ces pages ont un contenu qui existe dans un fichier sur disque repéré (comme tout fichier) par une structure appelée vnode (virtual node).
- les pages qui sont créées ou modifiées durant l'exécution du processus. Ces pages sont qualifiées d'*anonymes* et leur contenu est sauvegardé en zone de sauvegarde (swap) si la page réelle qui leur est associée doit être libérée.

4. Si plusieurs nœuds existent, il y a un démon par nœud.