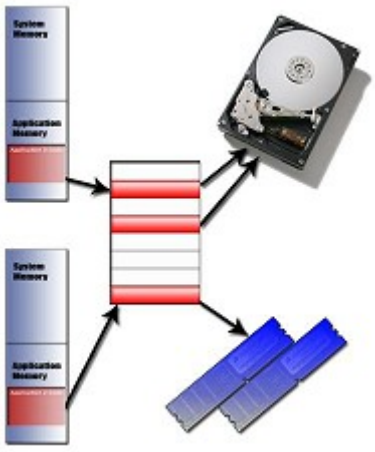
Gestion de la mémoire

Table des matières

1. Introduction....2
  1.1. Monoprogrammation sans va-et-vient ni pagination....2
  1.2. Multiprogrammation avec des partitions fixes....2
2. Swaping....3
  2.1. Gestion de la mémoire par tables de bits....3
  2.2. Gestion de la mémoire par listes chaînées....4
3. La mémoire virtuelle....4
  3.1. Pagination....5
4. La segmentation....6
  4.1. Implantation de segments purs....6
  4.2. Segmentation avec pagination....7
5. Gestion de la mémoire sous GNU/Linux....8
  5.1. Espace d'adressage d'un processus....8
  5.2. Gestion des pages et swap....11
  5.3. Appels systèmes Unix....12
    5.3.1. Allocation et désallocation de mémoire....12
    5.3.2. Protection de pages mémoires....13
    5.3.3. Verrouillage de pages mémoires....13
    5.3.4. Synchronisation de pages mémoire....14

Le gestionnaire de mémoire est un sous-ensemble du système d'exploitation. Son rôle est de partager la mémoire entre l'OS et les diverses applications. Le terme "mémoire" fait surtout référence à la mémoire principale, c'est à dire à la RAM, mais la gestion de celle-ci demande la contribution de la mémoire auxiliaire (mémoire de masse, spacieuse mais lente) et à la mémoire cache (rapide mais de taille restreinte).



1. Introduction

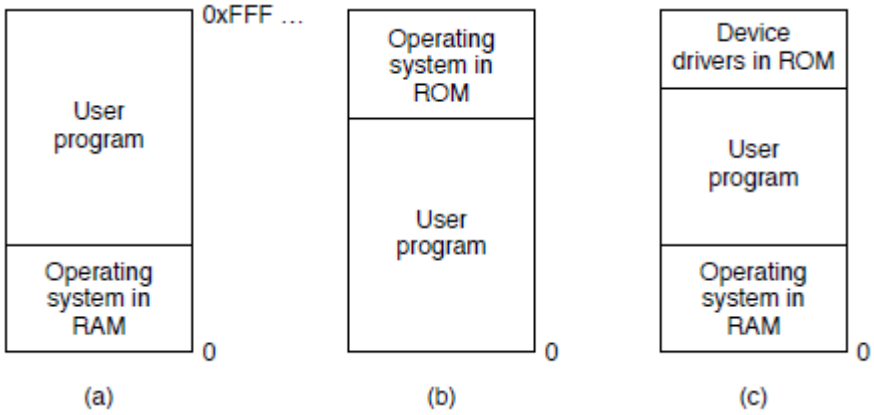
On peut subdiviser les systèmes de gestion de la mémoire en deux catégories :

- les systèmes qui peuvent déplacer les processus en mémoire secondaire pour trouver de l'espace de swap[1],
- les systèmes qui n'utilisent pas la mémoire secondaire.

1.1. Monoprogrammation sans va-et-vient ni pagination

L'approche la plus simple pour gérer la mémoire consiste à n'accepter qu'un seul processus à la fois (monoprogrammation) auquel on permet d'utiliser toute la mémoire disponible en dehors de celle qu'utilise le système.

Cette approche était utilisée, par exemple, dans les premiers micro-ordinateurs IBM PC utilisant MS-DOS comme système d'exploitation (c).

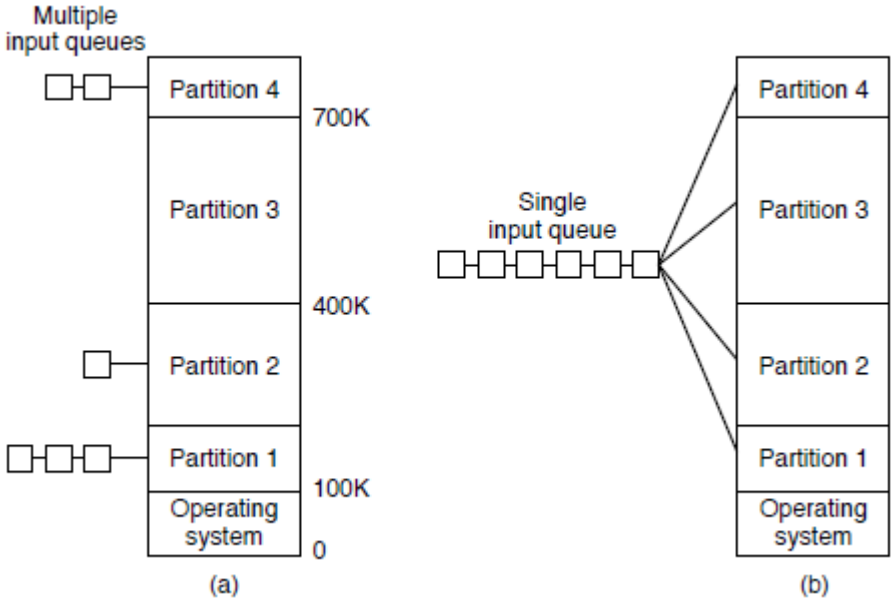


1.2. Multiprogrammation avec des partitions fixes

Dans un système multiprogrammé, il faut gérer la répartition de l'espace entre les différents processus. Cette partition peut être faite une fois pour toute au démarrage du système par l'opérateur de la machine, qui subdivise la mémoire en partitions fixes.

Chaque nouveau processus est placé dans la file d'attente de la plus petite partition qui peut le contenir (a). Cette façon de faire peut conduire à faire attendre un processus dans une file, alors qu'une autre partition pouvant le contenir est libre. L'alternative à cette approche consiste à n'utiliser qu'une seule file d'attente : dès qu'une partition se libère, le système y place le premier processus de la file qui peut y tenir (b).

1 va-et-vient



2. Swaping

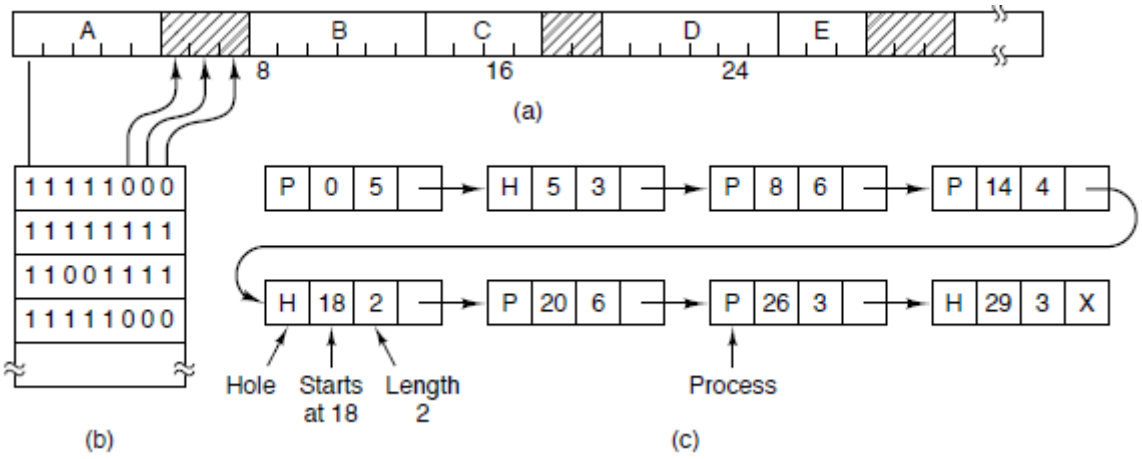
Dans un système qui peut déplacer les processus sur le disque quand il manque d'espace, le mouvement des processus entre la mémoire et le disque (va-et-vient) risque de devenir très fréquent, si on n'exploite pas au mieux l'espace libre en mémoire principale. Sachant que les accès au disque son très lents, les performances du système risquent alors de se détériorer rapidement. Il faut alors exploiter au mieux l'espace libre en mémoire principale, en utilisant un partitionnement dynamique de la mémoire.

Le fait de ne plus fixer le partitionnement de la mémoire rend plus complexe la gestion de l'espace libre. Celui-ci doit cependant permettre de trouver et de choisir rapidement de la mémoire libre pour l'attribuer à un processus. Il est donc nécessaire de mettre en œuvre des structures de données efficaces pour la gestion de l'espace libre.

2.1. Gestion de la mémoire par tables de bits

La mémoire est divisée en unités (quelques mots mémoire à plusieurs kilo-octets), à chaque unité on fait correspondre dans une table de bits :

- la valeur 1 si l'unité mémoire est occupée ;
- la valeur 0 si l'unité mémoire est libre.



(a) Une partie de la mémoire occupée par cinq processus (A,B,C,D et E) et trois zones libres. Les régions hachurées sont libres. (b) La table de bits (0 = libre, 1 = occupée). (c) Liste chainée des zones libres.

Pour allouer k unités contiguës, le gestionnaire de mémoire doit parcourir la table de bits à la recherche de k zéro consécutifs. Cette recherche s'avère donc lente pour une utilisation par le gestionnaire de la mémoire et est rarement utilisée à cet effet.

2.2. Gestion de la mémoire par listes chaînées

Une autre solution consiste à chaîner les segments libres et occupés. La figure ci-dessus montre l'exemple d'un tel chaînage, les segments occupés par un processus sont marqués (P) les libres sont marqués (H). La liste est triée sur les adresses, ce qui facilite la mise à jour.

Lorsqu'on libère la mémoire occupée par un segment, il faut fusionner le segment libre avec le ou les segments adjacents libres s'ils existent :

Ex : Quatre combinaisons de voisins possibles d'un processus X qui termine et libère le segment qu'il occupe

3. La mémoire virtuelle

Le principe de la mémoire virtuelle consiste à considérer un espace d'adressage virtuel supérieur à la taille de la mémoire physique, sachant que dans cette espace d'adressage, et grâce au mécanisme de va-et-viens sur le disque, seule une partie de la mémoire virtuelle est physiquement présente en mémoire principale à un instant donné. Ceci permet de gérer un espace virtuel beaucoup plus grand que l'espace physique sans avoir à gérer les changements d'adresses physiques des processus après un va-et-vient : car même après de multiples va-et-vient un processus garde la même adresse virtuelle. C'est notamment très utile dans les systèmes multiprogrammés dans lesquels les processus

qui ne font rien (la plupart) occupent un espace virtuel qui n'encombre pas nécessairement l'espace physique.

3.1. Pagination

La plupart des architectures actuellement utilisées reposent sur des processeurs permettant de gérer un espace virtuel paginé : l'espace d'adressage virtuel est divisé en pages, chaque page occupée par un processus est soit en mémoire physique soit dans le disque (va-et-vient).

La correspondance entre une page virtuelle et la page physique à laquelle elle peut être associée (si elle est en mémoire physique) est réalisée en utilisant une table de pages :

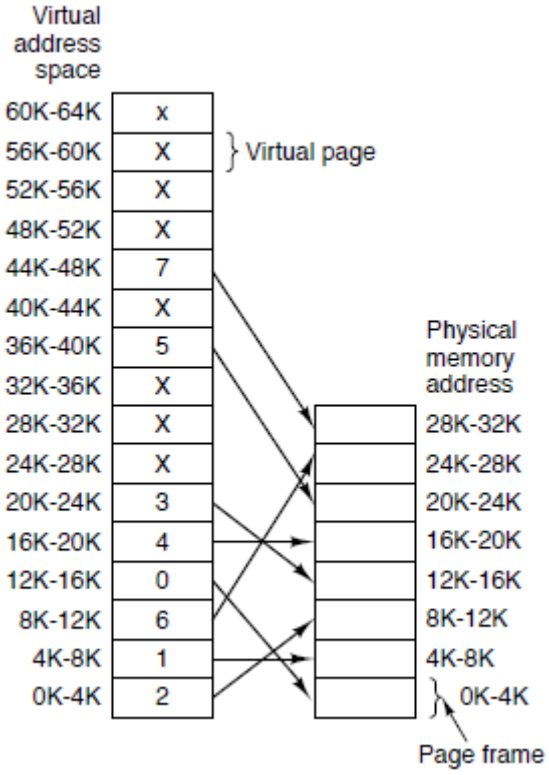


Table de pages

Lorsque le processeur doit exécuter une instruction qui porte sur un mot mémoire donné dont il a l'adresse virtuel, il cherche dans la table des pages l'entrée qui correspond à la page contenant le mot. Si la page est présente en mémoire il lit le mot, sinon il déclenche un déroutement de type défaut de page. A la suite de ce déroutement, le système de gestion de la mémoire est appelé afin de charger la page manquante à partir du disque (va-et-vient).

Chaque entrée de table de pages consiste en un descripteur de page qui contient généralement ces informations (au moins) :

- Numéro de la page en mémoire physique (cf. figure 5.5) si celle-ci est présente en mémoire.
- Un bit qui indique la présence (ou non présence) de la page en mémoire.
- Un bit de modification : qui mémorise le fait qu'une page a été modifiée, ceci permet au gestionnaire de mémoire de voir s'il doit la sauver sur le disque dans le cas ou il veut récupérer l'espace qu'elle occupe pour charger une autre page.
- Un bit de référencement : qui est positionné si on fait référence à un mot mémoire de la

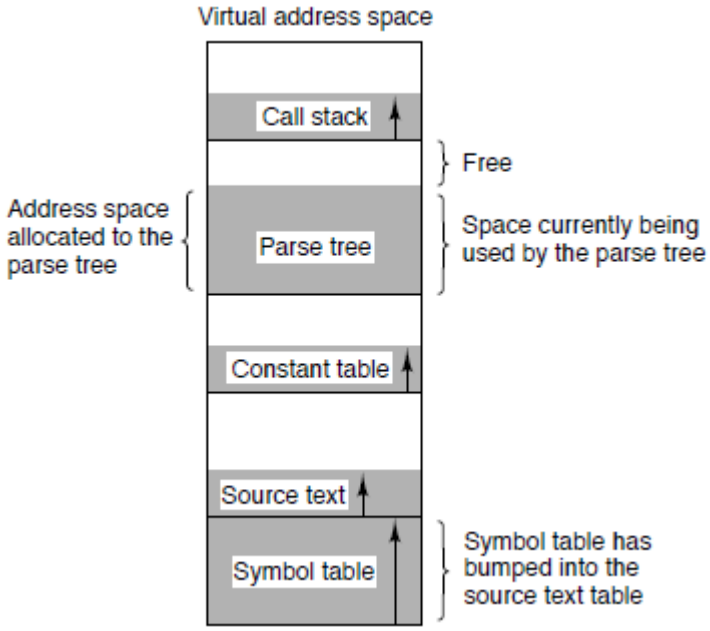
page, ceci permet au système de trouver les pages non référencées qui seront les meilleures candidates pour être retirées de la mémoire physique s'il y a besoin et manque de mémoire physique.

4. La segmentation

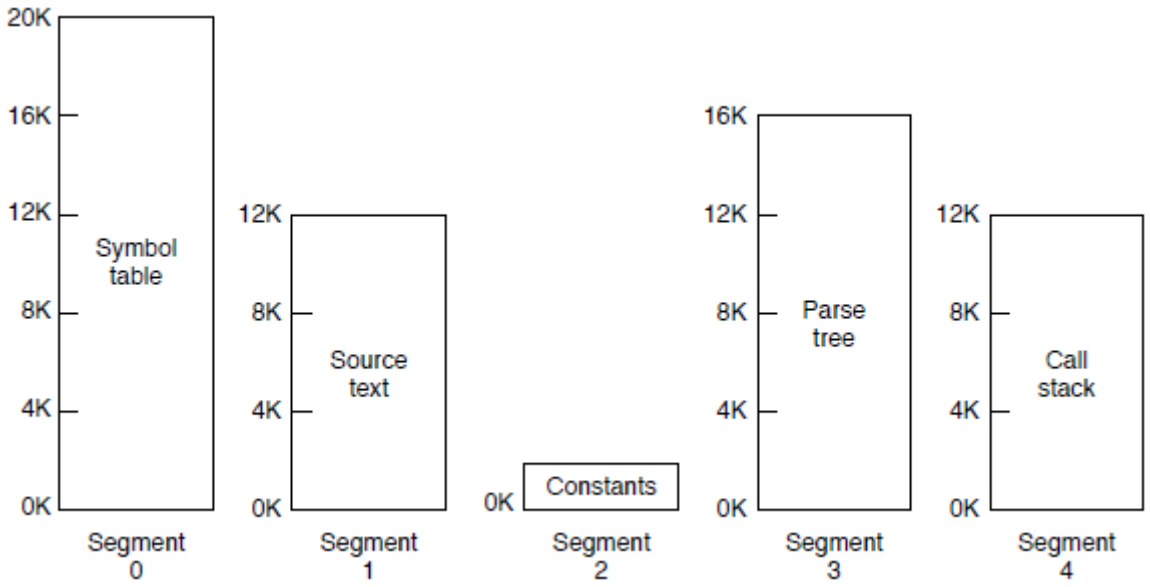
4.1. Implantation de segments purs

La segmentation de la mémoire permet de traiter la mémoire non plus comme un seul espace d'adressage unique, mais plutôt comme un ensemble de segments (portions de la mémoire de taille variable), ayant chacune son propre espace d'adressage. Ceci permet aux processus de simplifier considérablement le gestion de mémoire propre.

Ainsi un processus qui utilise différentes tables : table des symboles, code, table des constantes, etc. doit se préoccuper de la position relative de ses tables et doit gérer les déplacements de tables quand celles-ci croissent et risquent de se recouvrir :



Alors que dans une gestion de la mémoire segmentée, il lui suffit d'allouer un segment pour chaque table, la gestion de la position des segments dans l'espace d'adressage de la mémoire physique devient à la charge du système :



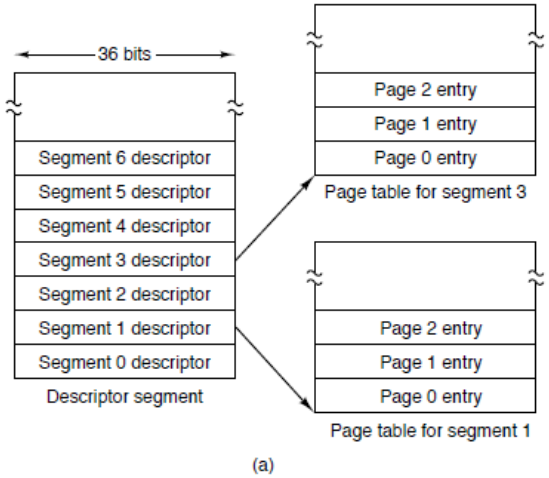
La segmentation permet aussi de faciliter le partage de zones mémoire entre plusieurs processus.

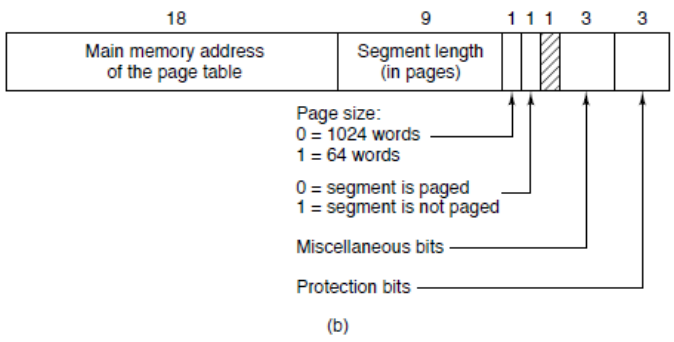
L'implémentation d'un système utilisant la segmentation pure (sans pagination) pose le problème de la fragmentation de la mémoire. La fragmentation est due au fait que les segments (contrairement aux pages) sont de taille non fixe.

4.2. Segmentation avec pagination

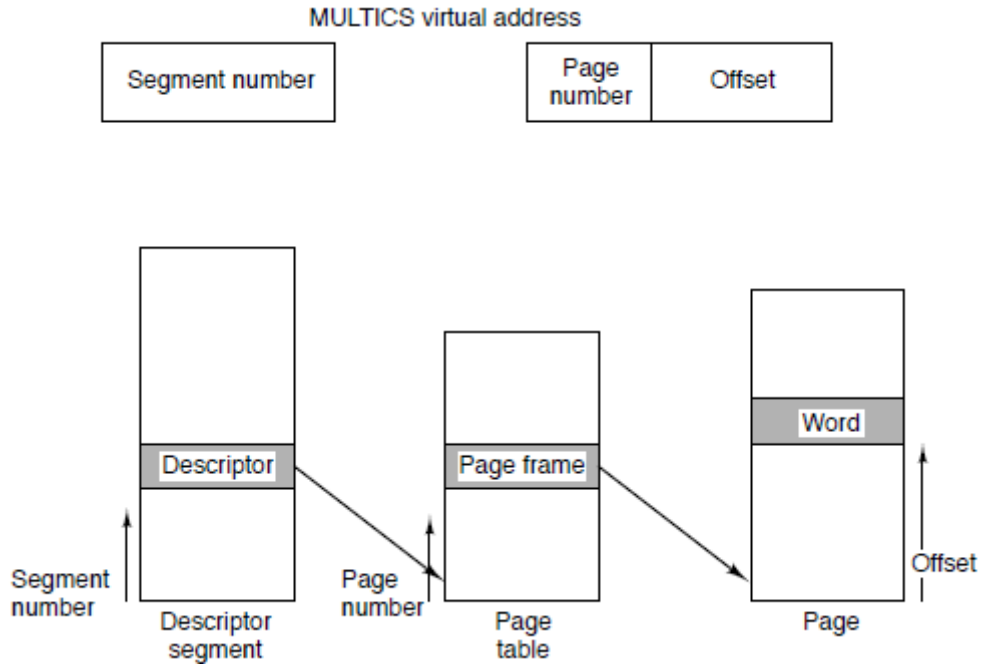
L'apparition du système d'exploitation MULTICS dans les années soixante a permis d'introduire divers concepts novateurs encore utilisés dans les systèmes les plus récents. Parmi ces idées, on trouve le principe de la segmentation avec pagination. En combinant segmentation et pagination MULTICS combine les avantages des deux techniques en s'affranchissant des principaux défauts qu'ils ont : fragmentation de la mémoire pour la segmentation, et espace d'adressage unique pour un système utilisant un adressage virtuel paginé.

Sous MULTICS chaque segment possède son propre espace d'adressage paginé. Ainsi chaque segment possède sa propre table de pages (a). Le descripteur de chaque segment contient, en plus de l'adresse de la table de pages qui lui est associée, la taille du segment, la taille des pages (1024 mots ou 64 mots), un indicateur permettant de paginer ou non l'espace d'adressage du segment, des bits de protection, ainsi que d'autres informations (b).





Ainsi, pour trouver une case mémoire, MULTICS doit consulter d'abord la table des segments, puis la table des pages, et enfin accéder à la page physique si elle est présente en mémoire :



Afin d'accélérer le processus, une mémoire associative permet de conserver les numéros des pages physiques correspondants aux pages les plus récemment référencées à partir de leur numéro de segment et de page :

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

5. Gestion de la mémoire sous GNU/Linux

Comme MULTICS, la gestion de la mémoire sous Linux est basée sur la segmentation avec pagination.

5.1. Espace d'adressage d'un processus

L'espace d'adressage d'un processus comprend plusieurs segments appelés régions (areas) parmi lesquels figurent typiquement :

- un segment de code ;
- deux segments de données, un concernant les données initialisées (à partir du fichier exécutable), et l'autre les données non initialisées ;
- un segment de pile. D'autres segments peuvent se trouver dans l'espace d'adressage du processus, ils concernent généralement les segments des librairies partagées qu'utilise le processus.

On peut trouver les informations concernant l'espace d'adressage d'un processus en visualisant le contenu du fichier maps du répertoire concernant ce processus dans le système de fichier /proc. Dans l'exemple qui suit, on découvre la liste des segments concernant un processus portant le numéro 798 associé à un shell /bin/tcsh :

```
$ cat /proc/798/maps
08048000-08083000 r-xp 00000000 03:03 28828    /bin/tcsh
08083000-08087000 rw-p 0003a000 03:03 28828    /bin/tcsh
08087000-080df000 rwxp 00000000 00:00 0
2aaab000-2aabf000 r-xp 00000000 03:03 34954    /lib/ld-2.1.1.so
2aabf000-2aac0000 rw-p 00013000 03:03 34954    /lib/ld-2.1.1.so
2aac0000-2aac1000 r--p 00000000 03:03 437930   /usr/share/locale/fr_FR/LC_MESSAGES/SY:
2aac1000-2aac4000 r--p 00000000 03:03 435875   /usr/share/locale/fr_FR/LC_CTYPE
2aac4000-2aac5000 r--p 00000000 03:03 435876   /usr/share/locale/fr_FR/LC_MONETARY
2aac5000-2aac6000 r--p 00000000 03:03 435878   /usr/share/locale/fr_FR/LC_TIME
2aac6000-2aac7000 r--p 00000000 03:03 435877   /usr/share/locale/fr_FR/LC_NUMERIC
2aac7000-2aac8000 r-xp 00000000 03:03 324948   /usr/lib/gconv/1508859-1.so
2aac8000-2aac9000 rw-p 00000000 03:03 324948   /usr/lib/gconv/1508859-1.so
2aaca000-2aadd000 r-xp 00000000 03:03 34974    /lib/libnsl-2.1.1.so
2aadd000-2aade000 rw-p 00012000 03:03 34974    /lib/libnsl-2.1.1.so
```

Le fichier maps permet ainsi de voir les principaux attributs d'un segment :

- L'adresse de début et de fin du segment dans l'espace d'adressage virtuel. On voit dans cette implémentation sur processeur x86 que les adresses virtuelles sont sur 32 bits et permettent donc d'adresser 4 Go.
- Les droits d'accès au segment, le caractère p indique que le segment peut être partagé. A partir des droits on peut en déduire que le premier segment associé à /bin/tcsh est son segment de code, le second son segment de données (initialisé).
- Le déplacement du début de segment dans l'objet qui lui est associé (le fichier exécutable dans le cas de /bin/tcsh).
- Le numéro de périphérique contenant l'objet.
- Le numéro d'i-node de l'objet associé au segment (nul s'il n'existe pas).
- Le chemin de l'objet.

Le descripteur associé à un segment est défini dans le fichier <linux/mm. h> :

```
struct vm_area_struct {
struct mm_struct * vm_mm; /* VM area parameters */
unsigned long vm_start;
```

```
unsigned long vm_end;

/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm next;

pgprot_t vm_page_prot;
unsigned short vm_flags;

/* AVL tree of VM areas per task, sorted by address */
short vm_avl_height;
struct vm_area_struct * vm_avl_left;
struct vm_area_struct * vm_avl_right;

/* For areas with inode, the list inode->i_mmap, for shm areas,
the list of attaches, otherwise unused. */
struct vm_area_struct *vm_next_share;
struct vm_area_struct **vm_pprev_share;

struct vm_operations_struct * vm_ops;
unsigned long vm offset;
struct file * vm_file;
unsigned long vm pte; /* shared mem */
};
```

On y retrouve notamment des champs correspondant à des attributs vus précédemment :

- adresse de début et de fin du segment (vm_start et vm_end) ;
- les droits d'accès (drapeau vm_f lags) ;
- le déplacement du début de segment dans le fichier associé (vm_of f set) ;
- le fichier associé (vm f ile).

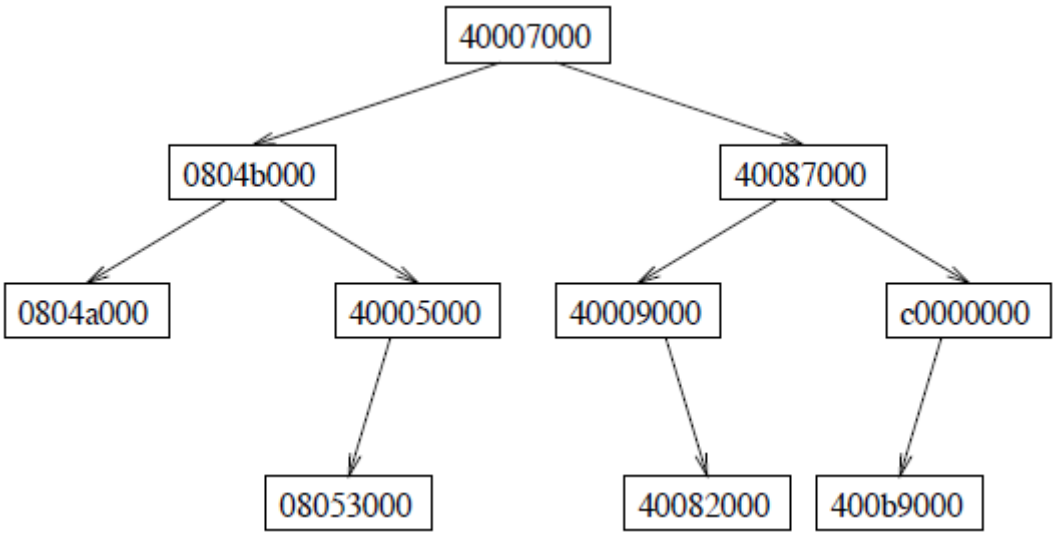
On y trouve aussi plusieurs pointeurs sur des descripteurs de segment correspondant aux structures de données suivantes :

- une liste des segments associés au processus (pointeur vm_next) ;
- une liste des segments partagés (pointeurs vm_next_share et vm pprev_share);
- un arbre AVL[2] des segments du processus (pointeurs vm avl_left et vm_avl_right). Organisation des descripteurs de segment

Ainsi les descripteurs de segment associés à un même processus sont référencés par l'espace d'adressage de deux façons différentes : par une liste chaînée d'une part et par un arbre AVL d'autre part.

Quand le noyau recherche le descripteur d'un segment particulier il n'utilise pas la liste mais l'arbre AVL, ce qui permet de réduire la complexité de la recherche de 0(n) à 0(log n), n étant le nombre de segments adressés par le processus.

2 Adelson-Velskii et Landis



Un arbre AVL est un arbre de recherche équilibré. Dans cet arbre chaque nœud associé à un descripteur de segment a pour clé l'adresse de fin du segment dans l'espace d'adressage virtuel. Comme pour tout arbre de recherche les clés du sous arbre gauche d'un nœud x valent au plus la clé de x, et celles du sous-arbre droit valent au moins la clé de x. Le fait que l'arbre soit équilibré lui assure une profondeur qui croît en O(logn). La figure 5.14 montre l'arbre AVL correspondant à un processus dans l'espace d'adressage. Ce processus contient les segments suivants :

```
08048000-0804a000 r-xp 00000000 03:02 7914
0804a000-0804b000 rw-p 00001000 03:02 7914
0804b000-08053000 rwxp 00000000 00:00 0
40000000-40005000 r-xp 00000000 03:02 18336
40006000-40007000 rw-p 00000000 00:00 0
40007000-40009000 r--p 00000000 03:02 18255
40009000-40082000 r-xp 00000000 03:02 18060
40082000-40087000 rw-p 00078000 03:02 18060
40087000-400b9000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp ffff0000 00:00 0
```

5.2. Gestion des pages et swap

Les tables de pages gérées par Linux sont organisées en trois niveaux :

- la table globale dont chaque entrée pointe sur une table intermédiaire ;
- les tables intermédiaires dont chaque entrée pointe sur une table de page ;
- les tables de pages dont les entrées contiennent les adresses de pages physiques.

Selon la plate forme sur laquelle Linux est implanté, le nombre de niveaux effectifs peut être plus réduit. Sur un processeur de type x86, par exemple, étant donné que les pages sont organisées en deux niveaux seulement, le noyau Linux considère que la table intermédiaire ne contient qu'une seule entrée.

Le descripteur associée à une page est définie dans le fichier <linux/mm. h> :

```
typedef struct page {
struct page *next;
struct page *prev;
struct inode *inode;
unsigned long offset;
struct page *next_hash;
atomic_t count;
unsigned long flags;
struct wait_queue *wait;
struct page **pprev_hash;
struct buffer_head * buffers;
} mem_map_t;
```

Lorsqu'une page est libre elle est chaînée avec les autres pages libres (pointeurs next et prev de la structure page).

Lorsque Linux manque de mémoire, il évince des pages mémoire. Pour ce faire le processus kswapd (endormi la plupart du temps) est réveillé. Ce processus explore la liste des processus et essaye d'évincer des pages. Si une page est réservée ou verrouillée, ou si elle a été récemment accédée, elle n'est pas évincée. Dans le cas contraire, son état est testé. Si la page a été modifiée, elle doit être sauvegardée sur disque avant d'être libérée (retirée de la mémoire donc du cache).

Ce mécanisme est utilisé également par Linux (depuis la vesion 2.0) pour les lectures de fichiers. Lors d'un appel à read la lecture est effectuée en chargeant en mémoire les pages correspondantes de l'i-node. Ceci permet d'utiliser le cache de pages pour optimiser l'accès aux fichiers. Ceci n'est réalisé que pour les lectures, les écritures sur fichiers. Les manipulations de répertoires s'effectuent normalement dans le cache du système de fichiers.

5.3. Appels systèmes Unix

Changement de taille du segment de données

```
#include <unistd.h>

int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

brk positionne la fin du segment de données (le premier mot mémoire hors de la zone accessible) à l'adresse spécifiée par end_data_segment. end_data_segment doit être supérieur à la fin du segment de texte (code), et doit être 16 Ko avant la fin de la pile.

sbrk incrémente l'espace de données du programme de increment octets.

5.3.1. Allocation et désallocation de mémoire

Les fonctions de la bibliothèque standard du langage C :

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

utilisent de manière interne les fonctions brk et sbrk pour allouer et désallouer des zones mémoire.

5.3.2. Protection de pages mémoires

La fonction :

```
#include <sys/man .h>

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

permet de contrôler la manière d'accéder à une portion de la mémoire.

prot est un OU binaire entre les valeurs suivantes :

- PROT_NONE : On ne peut pas accéder du tout à la zone de mémoire.
- PROT_READ : On peut lire la zone de mémoire.
- PROT_WRITE : On peut écrire dans la zone de mémoire.
- PROT_EXEC : La zone de mémoire peut contenir du code exécutable.

La nouvelle protection remplace toute autre protection précédente.

5.3.3. Verrouillage de pages mémoires

```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

- mlock désactive la pagination pour la portion de mémoire débutant à l'adresse addr avec une longueur de len octets. Toutes les pages contenant une partie de la zone mémoire spécifiée résident en mémoire principale et y resteront jusqu'à un déverrouillage par la fonction munlock ou munlockall, ou encore jusqu'à ce que le processus se termine ou se recouvre avec exec.
- munlock revalide la pagination pour la zone de mémoire commençant à l'adresse addr et s'étendant sur len octets.
- mlockall désactive la pagination pour toutes les pages représentées dans l'espace d'adressage du processus appelant. Ceci inclut les pages de code, de données, et le segment de pile, tout autant que les bibliothèques partagées, l'espace utilisateur dans le noyau, la mémoire partagée et les fichiers projetés en mémoire. Le paramètre f Tags détermine si les pages dont il faut désactiver la pagination sont celles qui sont actuellement présentes en mémoire physique (MCL_CURRENT), celles qui le seront dans le futur (MCL_FUTURE), ou les deux à la fois.
- munlockall revalide la pagination pour toutes les pages de l'espace d'adressage du processus en cours.

Il y a deux domaines principaux d'application au verrouillage de pages : les algorithmes en temps réel, et le traitement de données confidentielles. Les applications temps réel réclament un comportement temporel déterministe, et la pagination est, avec l'ordonnancement, une cause majeure de délais imprévus.

Les logiciels de cryptographie manipulent souvent quelques octets hautement confidentiels, comme des mots de passe ou des clés privées. A cause de la pagination ces données secrètes risquent d'être transférées sur un support physique où elles pourraient être lues longtemps après que le logiciel se

soit terminé.

5.3.4. Synchronisation de pages mémoire

La fonction :

```
#include <unistd.h>
#include <sys/mman.h>

int msync(const void *start, size_t length, int flags);
```

permet d'écrire sur le disque les modifications qui ont été effectuées sur la copie d'un fichier qui est projeté en mémoire par mmap. La portion du fichier correspondant à la zone mémoire commençant en start et ayant une longueur de length est mise a jour.

L'argument flags comprend les bits suivants :

- MS_SYNC : indique que l'appel à msync est bloquant : attend qu'elle se termine avant de revenir.
- MS ASYNC : indique que l'appel n'est pas bloquant.