Chapitre 9

La mémoire virtuelle

Les méthodes de gestion mémoire que nous venons de voir ont toutes un défaut majeur qui est de garder l'ensemble du processus en mémoire, ce qui donne :

- un coût en swap important
- Impossibilité de créer de très gros processus.

Les méthodes de mémoire virtuelle permettent d'exécuter un programme qui ne tient pas entièrement en mémoire centrale !

Nous avons commencé par présenter des algorithmes de gestion de la mémoire qui utilisent le concept de base suivant :
l'ensemble de l'espace logique adressable d'un processus doit être en mémoire pour pouvoir exécuter le processus.

Cette restriction semble à la fois raisonnable et nécessaire, mais aussi très dommageable car cela limite la taille des processus à la taille de la mémoire physique.

Or si l'on regarde des programmes très standards, on voit que :

- il y des portions de code qui gèrent des cas très inhabituels qui ont lieu très rarement (si ils ont lieu)
- les tableaux, les listes et autres tables sont en général initialisés à des tailles beaucoup plus grandes que ce qui est réellement utile
- Certaines options d'application sont très rarement utilisées

Même dans le cas où le programme en entier doit résider en mémoire, tout n'est peut-être pas absolument nécessaire en même temps.

Avec la mémoire virtuelle, la mémoire logique devient beaucoup plus grande que la mémoire physique.

De nombreux avantages :
Comme les utilisateurs consomment individuellement moins de mémoire, plus d'utilisateurs peuvent travailler en même temps. Avec l'augmentation de l'utilisation du CPU et de débit que cela implique (mais pas d'augmentation de la vitesse).

Moins d'entrées-sorties sont effectuées pour l'exécution d'un processus, ce qui fait que le processus s'exécute (temps réel) plus rapidement.

9.0.5 Les overlays

Une des premières versions d'exécutables partiellement en mémoire est celle des "overlay" qui est l'idée de charger successivement des portions disjointes et différentes de code en mémoire, exécutées l'une après l'autre.

Les différentes passes d'un compilateur sont souvent réalisées en utilisant un overlay (préprocesseurs, pass1, pass2, pour les compilateurs C).

Les overlay nécessitent quelques adaptations de l'éditeur de liens et des mécanismes de relocation.

9.0.6 Le chargement dynamique

Un autre système couramment utilisé dans les logiciels du marché des micros est le chargement dynamique. Avec le chargement dynamique, une fonction n'est chargée en mémoire qu'au moment de son appel. Le chargement dynamique demande que toutes les fonctions soient repositionnables en mémoire de façon indépendante.

A chaque appel de fonction on regarde si la fonction est en mémoire sinon un éditeur de liens dynamique est appelé pour la charger.

Dans les deux cas (overlay et chargement dynamique), le système joue un rôle très restreint, il suffit en effet d'avoir un bon système de gestion de fichiers.

Malheureusement, *le travail* que doit réaliser le programmeur pour choisir les overlays et/ou installer un mécanisme de chargement dynamique efficace *est non trivial* et requiert que le programmeur ait une *parfaite connaissance* du programme.

Ceci nous amène aux *techniques automatiques*.

9.1 Demand Paging

La méthode de **Demand Paging** est la plus répandue des implémentations de mémoire virtuelle, elle demande de nombreuse capacités matérielles.

Nous partons d'un système de swap où la mémoire est découpée en pages. Comme pour le swap, quand un programme doit être exécuté nous le chargeons en mémoire (swap in) mais au lieu de faire un swap complet, on utilise un "swappeur paresseux" (lazy swapper).
Un swappeur paresseux charge une page **uniquement si** elle est nécessaire.

Que ce passe-t-il quand le programme essaie d'accéder à une page qui est hors mémoire ?
- le matériel va traduire l'adresse logique en une adresse physique grâce à la table des pages.
- tant que les pages demandées sont en mémoire, le programme tourne normalement, sinon si la page est contenue dans l'espace des adresses logiques mais n'est pas chargée, il y a une **page fault**.

En général, une erreur d'adresse est dûe à une tentative d'accès à une adresse extérieure (invalide). Dans ce cas, le programme doit être interrompu, c'est le comportement normal d'un système de swap.

Mais il est possible avec un swappeur paresseux que la page existe mais ne soit pas en mémoire centrale, d'où les étapes suivantes dans ce cas :

On peut faire démarrer un processus sans aucune page en mémoire. La première **Page Fault** aurait lieu à la lecture de la première instruction (l'instruction n'étant pas en mémoire).

Il faut réaliser une forme spéciale de sauvegarde de contexte, il faut garder une image de l'état du processus qui vient d'effectuer une **Page Fault** mais de plus il faudra redémarrer (réexécuter) l'instruction qui a placé le processus dans cet état, en effet il est possible que l'instruction ne se soit pas terminé par manque de données.

Le système d'exploitation a ici un rôle important, c'est lui qui va réaliser le chargement de la page manquante puis relancer le processus et l'instruction.

Les circuits nécessaires à la méthode de Demande Paging sont les mêmes que ceux que l'on utilise pour un système de swap paginé, c'est-à-dire une mémoire secondaire et un gestionnaire de pages (table des pages).

Par contre, la partie logicielle est beaucoup plus importante.

Enfin il faut que les **instructions soient interruptibles**, ce qui n'est pas toujours le cas sur tous les processeurs et ce qui est fondamental, comme nous allons le voir sur des exemples :

add A,B in C

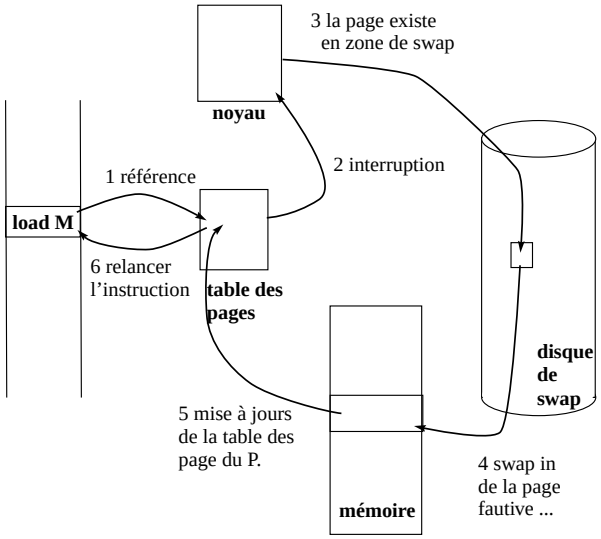

FIG. 9.1 – Etapes de la gestion d'une erreur de page

1. chercher et décoder l'instruction add
2. charger le contenu de l'adresse A
3. charger le contenu de l'adresse B
4. sommer et sauvegarder dans C

Si l'erreur de page a lieu dans le 4ième accès à la mémoire (C), il faudra de nouveau recommencer les 3 accès mémoire de l'instruction, c'est-à-dire lire l'instruction, etc.

Un autre type de problème vient d'instructions comme la suivante que l'on trouve sur PDP-11 :

MOV (R2)++,–(R3)

cette instruction déplace l'objet pointé par le registre R2 dans l'adresse pointé par R3, R2 est incrémenté après le transfert et R3 avant.

Que se passe-t-il si l'on a une erreur de page en cherchant à accéder à la page pointé par R3 ?

9.1.1 Efficacité

Efficacité des performances de Demand Paging :

Soit ma = 500 nanosecondes, le temps moyen d'accès a une mémoire.
le temps effectif d'accès avec le Demand Paging est

temps effectif = (1-p)*ma + p * "temps de gestion de l'erreur de page"
où p est la probabilité d'occurrence d'une erreur de page (page fault).

Une erreur de page nécessite de réaliser les opérations suivantes

1. lever une interruption pour le système
2. sauvegarder le contexte du processus
3. déterminer que l'interruption est une erreur de page
4. vérifier que la page en question est une page légale de l'espace logique, déterminer où se trouve la page dans la mémoire secondaire.
5. exécuter une lecture de la page sur une page mémoire libre (libérer éventuellement une page cf. algorithme de remplacement de page)
   – attendre que le périphérique soit libre

   - temps de latence du périphérique
   - commencer le transfert
6. allouer pendant ce temps-là le cpu à un autre utilisateur
7. interruption du périphérique
8. sauvegarde du contexte du processus courant
9. déterminer que l'interruption était la bonne interruption (venant du périphérique)
10. mise à jour de la table des pages et d'autres pages pour indiquer que la page demandée est en mémoire maintenant.
11. attendre que le processus soit sélectionné de nouveau pour utiliser l'unité centrale (cpu)
12. charger le contexte du processus !

Toutes ces instructions ne sont pas toujours réalisées (on peut en particulier supposer que l'on ne peut pas préempter l'unité centrale, mais alors quelle perte de temps pour l'ensemble du système).

Dans tous les cas, nous devons au moins réaliser les 3 actions suivantes :

- gérer l'interruption
- swapper la page demandée
- relancer le processus

Ce qui coûte le plus cher est la recherche de la page sur le disque et son transfert en mémoire, ce qui prend de l'ordre de 1 à 10 millisecondes.

Ce qui nous donne en prenant une vitesse d'accès mémoire de 1 microseconde et un temps de gestion de page de 5 millisecondes un

$$\text{temps effectif} = (1-p) + p \times 5000 \ \text{ microsecondes}$$

Une erreur de page toutes les mille pages nous donne un temps effectif onze fois plus long que l'accès standard.

Il faut réduire à moins d'une erreur de page tout les 100000 accès pour obtenir une dégradation inférieure à 10

On comprend bien que les choix à faire sur des pages qu'il faut placer en mémoire sont donc très importants.

Ces choix deviennent encore plus importants quand l'on a de nombreux utilisateurs et qu'il y a sur-allocation de la mémoire, exécution concurrente de 6 processus de la taille supérieure ou égale à la mémoire physique !

Si l'on suppose de plus que nos 6 programmes utilisent dans une petite séquence d'instructions toutes les pages de leur mémoire logique, nous nous trouvons alors dans une situation de pénurie de pages libres.

Le système d'exploitation peut avoir recoure à plusieurs solution dans ce cas-là

1. tuer le processus fautif ...
2. utiliser un algorithme de remplacement de page

Cet algorithme de remplacement est introduit dans notre séquence de gestion d'erreur de page là où l'on s'attribuait une page libre de la mémoire centrale.

Maintenant il nous faut sélectionner une victime, c'est-à-dire, une des pages occupées de la mémoire centrale qui sera swappée sur disque et remplacée par la page demandée.

Remarquons que dans ce cas-là notre temps de transfert est doublé, comme il faut à la fois lire une page et sauvegarder une page sur disque (le temps de transfert disque est ce qui est le plus coûteux dans la gestion d'une erreur de page).

Il est possible de réaliser des systèmes de **demand segments**, mais le lecteur avisé remarquera rapidement les problèmes posés par la taille variable des segments.

9.2 Les algorithmes de remplacement de page

Un algorithme de remplacement de page doit minimiser le nombre de Page Faults.

On recherche l'algorithme qui réduit au mieux la probabilité d'occurrence d'une erreur de page. Un algorithme est évalué en prenant une chaîne de numéros de page et en comptant le nombre de fautes de page qui ont lieu au cours de cette suite d'accès, et cela en fonction du nombre de pages de mémoire centrale dont il dispose.

Pour illustrer les algorithmes de remplacement, nous utiliserons la suite de pages suivante :
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
et 3 pages en mémoire centrale.

9.2.1 Le remplacement optimal

Utiliser comme victime la page qui ne sera pas utilisée pendant le plus longtemps.
Soit pour notre suite :

```
7xx 70x 701 201 - 203 - 243 - -203 - - 201 - - - 701 - -
```

soit seulement 9 fautes de page.
Mais cet "algorithme" n'est valable que dans un cas où l'on connaît à l'avance les besoins, ce qui n'est généralement pas le cas.

9.2.2 Le remplacement peps (FIFO)

L'algorithme le plus simple est Premier Entré Premier Sorti (First-In-First-Out).

Quand une victime doit être sélectionnée c'est la page la plus ancienne qui est sélectionnée.

Soit pour la liste

```
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
```

et trois page de mémoire centrale :

```
7XX/70X/701/201-201/231/230/430/420/423/
023-023-023/013/012-012-012/712/702/701
```

soit **Quinze** Page Faults.
Ce mécanisme rapide et simple à programmer n'est malheureusement pas très efficace. Il existe des suites de pages pour lesquelles cet algorithme fait plus de page faults avec quatre pages mémoire qu'avec trois ! (par exemple : 1,2,3,4,1,2,5,1,2,3,4,5).

9.2.3 Moins récemment utilisée LRU.

LRU (Least Recently Used page).
Nous utilisons ici le vieillissement d'une page et non plus l'ordre de création de la page. On fait le pari que les pages qui ont été récemment utilisées le seront dans un proche avenir, alors que les pages qui n'ont pas été utilisées depuis longtemps ne sont plus utiles.

Soit pour notre suite :

```
7xx 70x 701 201 - 203 - 403 402 432 032 - - 132 - 102 - 107 -
```

soit **Douze** Page Faults.

L'algorithme LRU est un bon algorithme mais il pose de nombreux problèmes d'implémentation et peut demander de substantiels outils matériels.

Des solutions logicielles :

Des compteurs à chaque entrée de la table des pages, on ajoute un compteur de temps qui est mis à jour à chaque accès à la page. Il faut rechercher sur l'ensemble de la table la victime. De plus, ces temps doivent être mis à jour quand on change de table de page (celle d'un autre processus ...). On ne peut utiliser le temps réel ...

Une pile à chaque fois que l'on accède à une page, la page est placée en sommet de pile. Le dessus est toujours la page la plus récemment utilisée et le fond de la pile la moins récemment utilisée.

Des masques On utilise un octet associé à chaque page. Le système positionne à 1 le bit de poids fort à chaque accès à la page. Toutes les N millisecondes (click d'horloge, cf clock, N = 100 sur fillmore) le système fait un décalage à droite de l'octet associé à chaque page. On obtient ainsi un historique de l'utilisation de la page. L'octet à 00000000 indique que la page n'a pas été utilisée depuis 8 cycles, 11111111 indique que la page a été utilisée pendant les 8 cycles. La page de masque 11000100 à été utilisée plus récemment que 01110111. Si l'on interprète ces octets comme des entiers non-signés, c'est la page ayant le plus petit octet qui a été utilisée le moins récemment (l'unicité des numéros n'étant pas assurée, la sélection entre numéros identiques se fait avec l'ordre FIFO).

9.2.4 L'algorithme de la deuxième chance

Un bit associé à chaque page est positionné à 1 à chaque fois qu'une page est utilisée par un processus. Avant de retirer une page de la mémoire, on va essayer de lui donner une deuxième chance. On utilise un algorithme FIFO plus la deuxième chance :
Si le bit d'utilisation est à 0, la page est swappée hors mémoire (elle n'a pas été utilisée depuis la dernière demande de page).
Si le bit est à 1, il est positionné a zéro et l'on cherche une autre victime. Ainsi cette page ne sera swappée hors mémoire que si toutes les autres pages ont été utilisées, et utilisent aussi leur deuxième chance.

On peut voir ceci comme une queue circulaire, où l'on avance sur les pages qui ont le bit à 1 (en le positionnant à zéro) jusqu'à ce que l'on trouve une page avec le bit d'utilisation à zéro.

9.2.5 Plus fréquemment utilisé MFU

Plus fréquemment Utilisée :
Comme son nom l'indique, c'est la fréquence d'utilisation qui joue au lieu de l'ancienneté, mais c'est le même mécanisme que LRU. Ces deux algorithmes de LRU et MFU sont rarement utilisés car trop gourmands en temps de calcul et difficiles à implémenter, mais ils sont assez efficaces.

9.2.6 Le bit de saleté (Dirty Bit)

Remarquons que si il existe une copie identique sur disque (zone de swap) d'une page de mémoire, il n'est pas nécessaire dans le cas d'un swapout de sauvegarder la page sur disque, il suffit de la libérer.
Le bit de saleté permet d'indiquer qu'une page est (ou n'est plus) conforme à la page en zone de swap.

Ce bit de propreté est utilisé dans les autres algorithmes, on choisit entre deux victimes possibles la plus propre, c'est-à-dire celle qui ne nécessite pas de swapout.

9.3 Allocation de pages aux processus

Comment répartir les pages sur les différents processus et le système ?

remplacement local le processus se voit affecté un certain nombre de pages qu'il va utiliser de façon autonome, son temps d'exécution ne dépend que de son propre comportement.

remplacement global le comportement d'allocation de pages aux processus dépend de la charge du système et du comportement des différents processus.

Le remplacement local demande que l'on réalise un partage entre les différents processus.

Le partage "équitable" : m pages de mémoire physique, n processus, m/n pages par processus ! On retrouve ici un problème proche de la fragmentation interne, un grand nombre de pages est donné à un processus qui en utilise effectivement peu.

On fait un peu mieux en utilisant : $S = \Sigma\ s_i$ où s_i est le nombre de pages de la mémoire logique du Processus i. Chaque processus se voit attribué $(s_i/S)m$ pages. On améliore en faisant varier ce rapport en fonction de la priorité de chaque processus.

Problèmes d'écroulement Si le nombre de pages allouées à un processus non-prioritaire tombe en dessous de son minimum vital, ce processus est constamment en erreur de page : il passe tout son temps à réaliser des demandes de pages. Ce processus doit être alors éjecté entièrement en zone de swap et reviendra plus prioritaire quand il y aura de la place.
Un exemple de bonne et mauvaise utilisation des pages (rappel les compilateurs c allouent les tableaux sur des plages d'adresse croissante contigües int m[A][B] est un tableau de A tableaux de B entiers) :

```
/* bonne initialisation */
int m[2048][2048];
main()
{int i,j;
for(i=0;i<2048;i++)
        for(j=0;j<2048;j++)
        m[i][j] = 1;
}
```

ce processus accède a une nouvelle page toute les 2048 affectation.

```
/* mauvaise initialisation */
int m[2048][2048];
main()
{int i,j;
for(i=0;i<2048;i++)
        for(j=0;j<2048;j++)
        m[j][i] = 1;
}
```

ce processus accède a une nouvelle page toute les affectations !
Attention : En `fortran` l'allocation des tableaux se fait dans l'autre sens par colones ...

Si la mémoire est libre et assez grande, les deux processus sont grossièrement aussi rapides, par contre si on lance dix exemplaires du premier, le temps d'attente est juste multiplié par 10. Pour le deuxième, le temps d'attente est au moins multiplié par 100 (je n'ai pas attendu la fin de l'exécution).

9.4 L'appel fork et la mémoire virtuelle

Nous avons vu que la primitive `fork()` réalise une copie de l'image mémoire du processus père pour créer le processus fils. Cette copie n'est pas intégrale car les deux processus peuvent partager des pages marquées en lecture seule, en particulier le segment du code est partagé par les deux processus (réentrance standard des processus unix).

Mais avec le système de demand-paging, on peut introduire une nouvelle notion qui est la "copie sur écriture" (copy on write). On ajoute à la structure de page de la table des pages des indicateurs de "copie sur écriture". L'idée est de réaliser la copie de la page uniquement dans le cas où l'un des processus qui peuvent y accèder réalise une écriture. Dans ce cas-là, la page est recopiée avant l'écriture et le processus écrivain possède alors sa propre page.

L'intérêt de ce mécanisme est surtout visible dans le cas très fréquent où le `fork` est immédiatement suivi par un `exec`. En effet, ce dernier va réaliser une libération de toutes les pages, il est donc inutile de les recopier juste avant cette libération.

Le système BSD a introduit la première version de cette idée en partant de l'appel système `vfork()` qui lui permet le partage totale de toutes les pages entre le processus père et le processus fils sans aucune copie. L'intérêt est de pouvoir réaliser rapidement un `execve` sans avoir à recopier l'espace d'adressage du processus père.

9.5 Projection de fichiers en mémoire

La fonction `mmap` permet la projection de fichiers en mémoire. Le segment du fichier indiqué est placé en mémoire à partir de l'adresse indiquée. Le segment de fichier peut ainsi être parcouru par des accès par adresse sans utiliser de commande de lecture ou d'écriture.

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

void *mmap(void  *adr, int len,
           int   prot, int options,
           int   desc, int offset);

int munmap(void *adr, int len);
```

L'adresse `adr` indique où doit être placé le fichier, cette adresse doit être une adresse de début de page (un multiple de `sysconf(_SC_PAGE_SIZE)`), si le paramètre est NULL alors le système sélectionne l'adresse de placement qui est retournée par la fonction. L'intervalle de position

`[offset, offset+len]`

du fichier `desc` est placé en mémoire.
`prot` indique les protections d'accès sous HP-UX les protections suivantes sont disponible :

```
---     PROT_NONE
r--     PROT_READ
r-x     PROT_READ|PROT_EXECUTE
rw      PROT_READ|PROT_WRITE
rwx     PROT_READ|PROT_WRITE|PROT_EXECUTE
```

`options` indique si l'on veut que les écritures réalisées dans les pages contenant la projection soient partagées (MAP_SHARED), ou au contraire qu'une copie sur écriture soit réalisée (MAP_PRIVATE).

La fonction `munmap` permet de libérer la zone mémoire d'adresse `adr` et de longueur `len`.

Pour une autre forme de mémoire partagée, voir le chapitre sur les IPC (sur le web).

Un exemple d'utilisation de `mmap` pour copier un fichier :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(int argc,char *argv[])
{
  int fdin,fdout ;
  struct stat statbuf ;
  char *src,*dst ;
  if (argc != 3)
    {
      fprintf(stderr,"usage : %s source destination ",argv[0]) ;
      exit(-1) ;
    }
  if ((fdin = open(argv[1], O_RDONLY)) < 0)
    {
      fprintf(stderr,"impossible d\'ouvrir : %s en lecture ",argv[1]) ;
      exit(-2) ;
    }
  if ((fdout = open(argv[2], O_RDWR|O_CREAT|O_TRUNC,0666)) < 0)
    {
      fprintf(stderr,"impossible d\'ouvrir : %s en ecriture ",argv[2]) ;
      exit(-3) ;
    }
  if (fstat(fdin,&statbuf) < 0 )
    {
      fprintf(stderr,"impossible de faire stat sur  %s ",argv[1]) ;
      exit(-4) ;
    }
  if (lseek(fdout, statbuf.st_size -1 , SEEK_SET) == -1 )
    {
      fprintf(stderr,"impossible de lseek  %s ",argv[2]) ;
      exit(-5) ;
    }
  if (write(fdout,"",1) != 1)
    {
      fprintf(stderr,"impossible d\'ecrire sur   %s ",argv[2]) ;
      exit(-6) ;
    }
```

```
  if ((src = mmap (0,statbuf.st_size, PROT_READ,
                   MAP_FILE | MAP_SHARED, fdin,0)) == (caddr_t) -1 )
    {
      fprintf(stderr,"impossible de mapper   %s ",argv[1]);
      exit(-7);
    }
  if ((dst = mmap (0,statbuf.st_size, PROT_READ | PROT_WRITE,
                   MAP_FILE | MAP_SHARED, fdout,0)) == (caddr_t) -1 )
    {
      fprintf(stderr,"impossible de mapper   %s ",argv[2]);
      exit(-8);
    }
  memcpy(dst,src,statbuf.st_size); /* copie */

  exit(0);

}
```

Attention, quand vous utilisez `mmap` c'est de la mémoire, mais c'est a vous de gérer l'allignement. Exemple :

```
char *p = map(...);
int *q = p+1; // warning
 *q = 1 ; // problème d'allignement
```

9.6 Les conseils et politiques de chargement des zones mmappées

Une foit que l'on décider de faire des projection en mémoire avec `mmap` il peut être oportin de faire appel à la fonction `madvise` qui permet de donner un conseil au système en le prévenant par avance de la façon dont vous aller utiliser le segement de mémoire. En particulier allez vous lire le fichier sequentiellement ou de façon alléatoire. Avez vous encore besoin du fichier après lecture etc. Biensur la fonction `madvise` ne se limite pas aux pages mappés mais c'est sur celle ci qu'il est le plus facile de prendre des décisions, les autres pages étant gérer dans la pile le tas et le code zone plus délicates et moins bien cartographiées en générale (sic).

```
#include <sys/mman.h>
int madvise(void *start, size_t length, int advice);
```

La valeur du conseil `advice` :

MADV_NORMAL Comportement par défaut.

MADV_RANDOM prevoit des acces aux pages dans un ordre aleatoire.

MADV_SEQUENTIAL prevoit des acces aux pages dans un ordre sequentiel.

MADV_WILLNEED prevoit un acces dans un futur proche.

MADV_DONTNEED Ne prevoit pas d'acces dans un futur proche. Biensur si c'est une mmap vous pouvez aussi utiliser la commande `munmap`.

Biensur ce ne sont que des conseils le système les utilisera si il en a la possibilité, soit parce qu'il y a du temps idle (sans activité) soit parce qu'il profitera des lectures groupées sur disque en réalisant des lectures en avance cas séquentiel. Il peut aussi profiter de l'indication DONTNEED pour prendre des décissions dans le code de remplacement de page.

9.7 Chargement dynamique

Indépendement de l'existance de la mémoire virtuel il est possible de gérer "à la main" le code accessible en utilisant le chargement direct (non automatique) de librairies.

Pour construire une librairie de fichier lib.c :

```
#include <stdio.h>

/* fonction optionelle qui permet d'initialiser la libraire */
void _init()
{
  fprintf(stderr," intialisation \n");
}
/* fonction optionelle qui est appellée avant le déchargement */
void _fini()
{
fprintf(stderr," déchargement exécution de _fini \n");
}
/* des fonctions spécifiques a votre libraire */
int doit(int u)
{
  fprintf(stderr," doit to u= %d " , u );
  return u*u;
}
```

Compilation de la libraire, l'option nostartfile pour que le compilateur ne construise pas d'exécutable.

```
 gcc -shared -nostartfiles -o ./malib lib.c
```

Le fichier main qui va charger la libraire puis appeler une fonction de cette libraire.

```
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
  void *handle;
  int (*doit)(int);
  char *error;

  handle = dlopen ("./malib", RTLD_LAZY);
  if (!handle) {
    fputs (dlerror(), stderr);
    exit(1);
  }

  doit = dlsym(handle, "doit");
  if ((error = dlerror()) != NULL)  {
    fprintf (stderr, "%s\n", error);
    exit(1);
  }

  printf ("%d\n", (*doit)(23));
  handle = dlopen ("./malib", RTLD_LAZY); // nouveau référencement avce le même handle
 dlclose(handle); // décrementation du conteur de référence
```

```
 fprintf(stderr," avant le deuxieme dlclose \n");
  dlclose(handle);
}
```