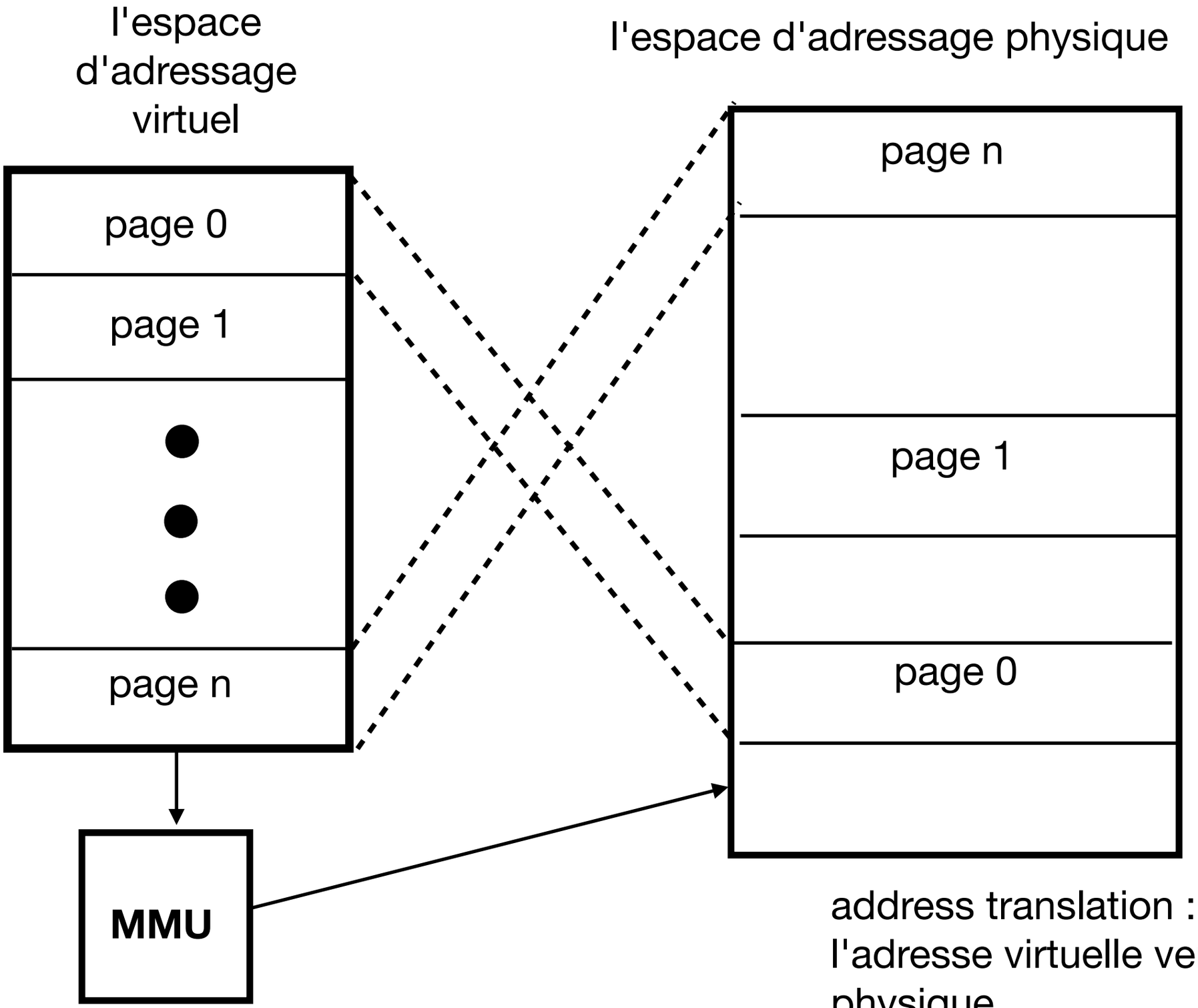
# Programmation système avancée

Wieslaw Zielonka
zielonka@irif.fr

# Rappel : mémoire virtuelle et mémoire physique




address translation : traduction de l'adresse virtuelle vers l'adresse physique

MMU - Memory Management Unit

# mémoire

Chaque page virtuelle marquée soit comme "*resident" ou "nonresident" (présente ou absente).*

Si la page physique absente est demandée : défaut de page (page fault).

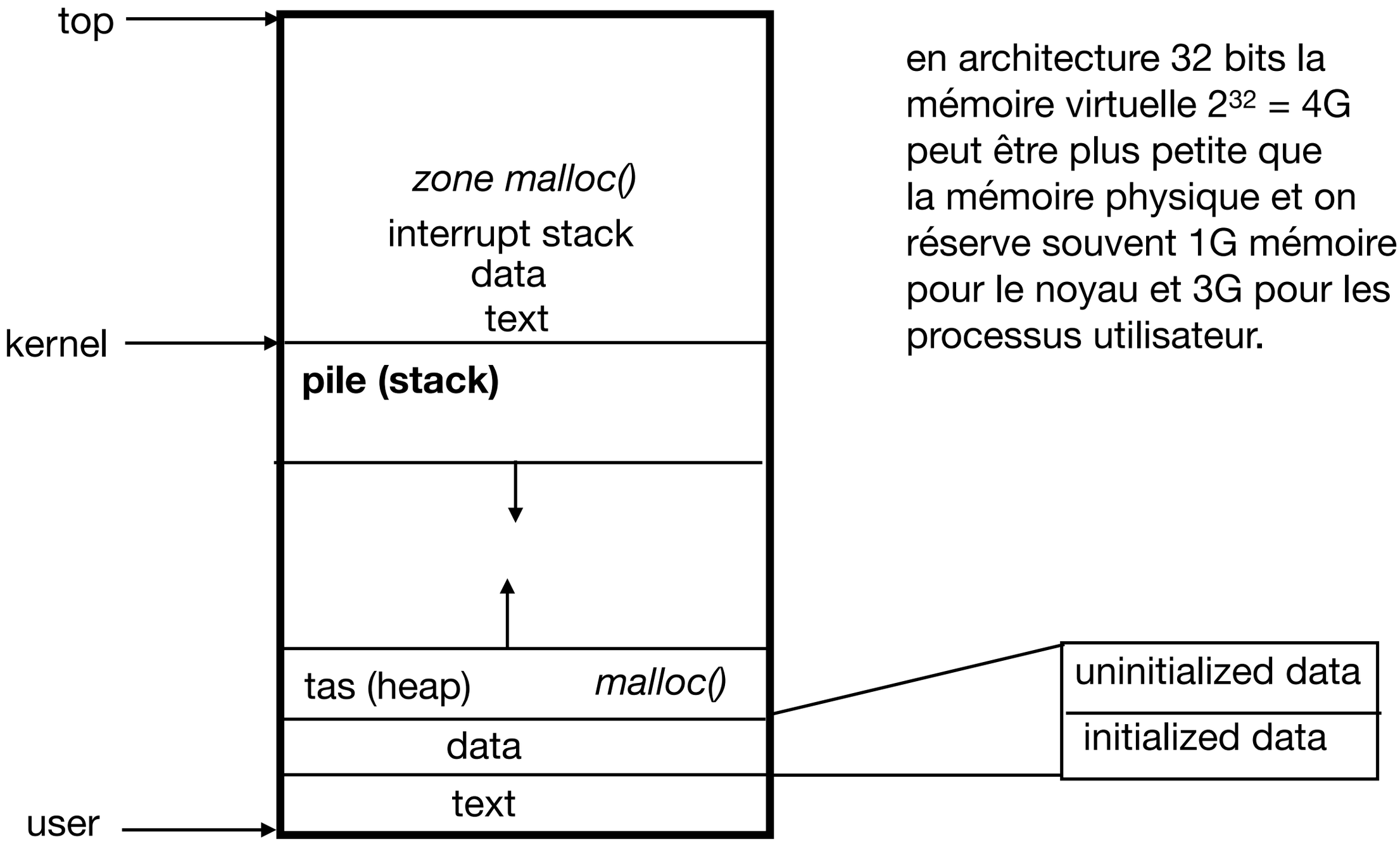
MMU - le dispositif, matériel dans les systèmes modernes, qui fait la translation entre l'adresse virtuelle et l'adresse physique.

Les pages sont marquées comme

- soit disponibles en lecture. Ces pages peuvent être partagées par plusieurs processus (par exemple les pages du segment *text* qui contient le code du programme)

- soit  disponibles en lecture et écriture (comme les page de données, pile, et tas).  Les pages disponibles en lecture et écriture peuvent être partagées entre plusieurs processus (par exemple après fork())  mais le processus qui essaie d'écrire sur une telle page provoque la création d'une copie de la page et l'écriture est effectué sur cette copie (*copy-on-write*).

# mémoire




en architecture 32 bits la mémoire virtuelle $2^{32}$ = 4G peut être plus petite que la mémoire physique et on réserve souvent 1G mémoire pour le noyau et 3G pour les processus utilisateur.

# mémoire

La page à l'adresse 0 est protégée pour détecter les références avec le pointeur NULL (dans FreeBSD, toute la page à l'adresse virtuelle 0 est hors l'espace de processus).

En pratique le partage en zone bien délimitées est moins clair, par exemple les bibliothèques partagées peuvent placer le texte et data de manière arbitraire (souvent juste au-dessus de la valeur limite maximale de tas). Et il faut aussi placer dans l'espace d'adresses virtuelles la mémoire partagée entre les processus (plus loin dans ce cours).

les moyens de communication traditionnel entre les processus :

- pipes
- sockets
- fichiers
- les files de messages

isolent chaque processus. Si un processus détruit son espace d'adressage cela n'a aucune influence sur d'autres processus. Mais cela a un coût : deux appels système, un pour le processus qui envoie des données et un pour le processus qui reçoit les données.  S'il y a beaucoup d'échange de données entre les processus le coût d'appels système peut devenir plus grand que le coût de communication.

En plus chaque communication implique l'opération de copie de données.

Mémoire partagée entre le processus réduit drastiquement le coût d'appels système, pas besoin d'appels système : un processus écrit dans la mémoire partagée  et l'autre lit directement dans cette mémoire.

Toutefois besoin de mécanisme de synchronisation entre les processus, et il est préférable que ce mécanisme lui-même n'utilise pas les appels système.

Inconvénient et danger : un processus peut détruire de structures de données utilisées par l'autre (structures dans la mémoire partagée, il n'y a plus d'isolation entre les processus).

Quand deux processus veulent de créer une mémoire partagée ils ont besoin de

- nommer cette mémoire,
- décider qu'elle est sa taille,
- décider comment elle est initialisée.

On utilise pour cela les fichiers qui fournissent le "nom" de la mémoire partagé et servent à initialiser le contenu.

# Projection de fichier en mémoire

# projection de fichier en mémoire (memory-mapped file)

Projection de fichier en mémoire consiste à charger un fichier (ou un segment de fichier) directement dans la mémoire sans utilisation de `read()`.

Cette mémoire peut être partagée avec d'autres processus c'est-à-dire  plusieurs processus partagent les pages de la mémoire physique utilisées pour la projection.

Deux moyens pour partager la mémoire :

- deux processus effectuent la projection "partagée" du même segment d'un fichier
- le processus enfant créé par `fork()` hérite la référence vers la même zone mémoire partagée allouée par le processus père.

# projection de fichier en mémoire

De types de projection :

- Projection privée `MAP_PRIVATE`

`Le segment de fichier est copié dans la mémoire mais les` modifications dans la mémoire ne sont pas visibles par d'autres processus et ne sont pas répercutés dans le fichier. Le noyau utilise la technique *copy-on-write.* Initialement la mémoire est partagée avec d'autres processus qui font la projection en mémoire du même segment de fichier. Mais quand un processus avec projection MAP_PRIVATE modifie la mémoire le noyau crée une copie privée de pages modifiées.

- projection partagée `MAP_SHARED`

Modifications de la mémoire sont visibles par d'autres processus qui projettent le même fichier dans la mémoire en mode MAP_SHARED. Les modifications dans la mémoire sont (peut-être pas immédiatement) réécrits dans fichier correspondant .

# projection de fichier en mémoire

Projections en mémoire (partagées et privées) sont préservées par `fork()` mais perdues par `exec()`.

Les informations sur les projections d'un processus PID sont disponibles sous Linux dans le fichier

`/proc/PID/maps`

# projection de fichier en mémoire

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

void *mmap(void *adr, size_t len, int prot,

           int flags, int fd, off_t offset)
```

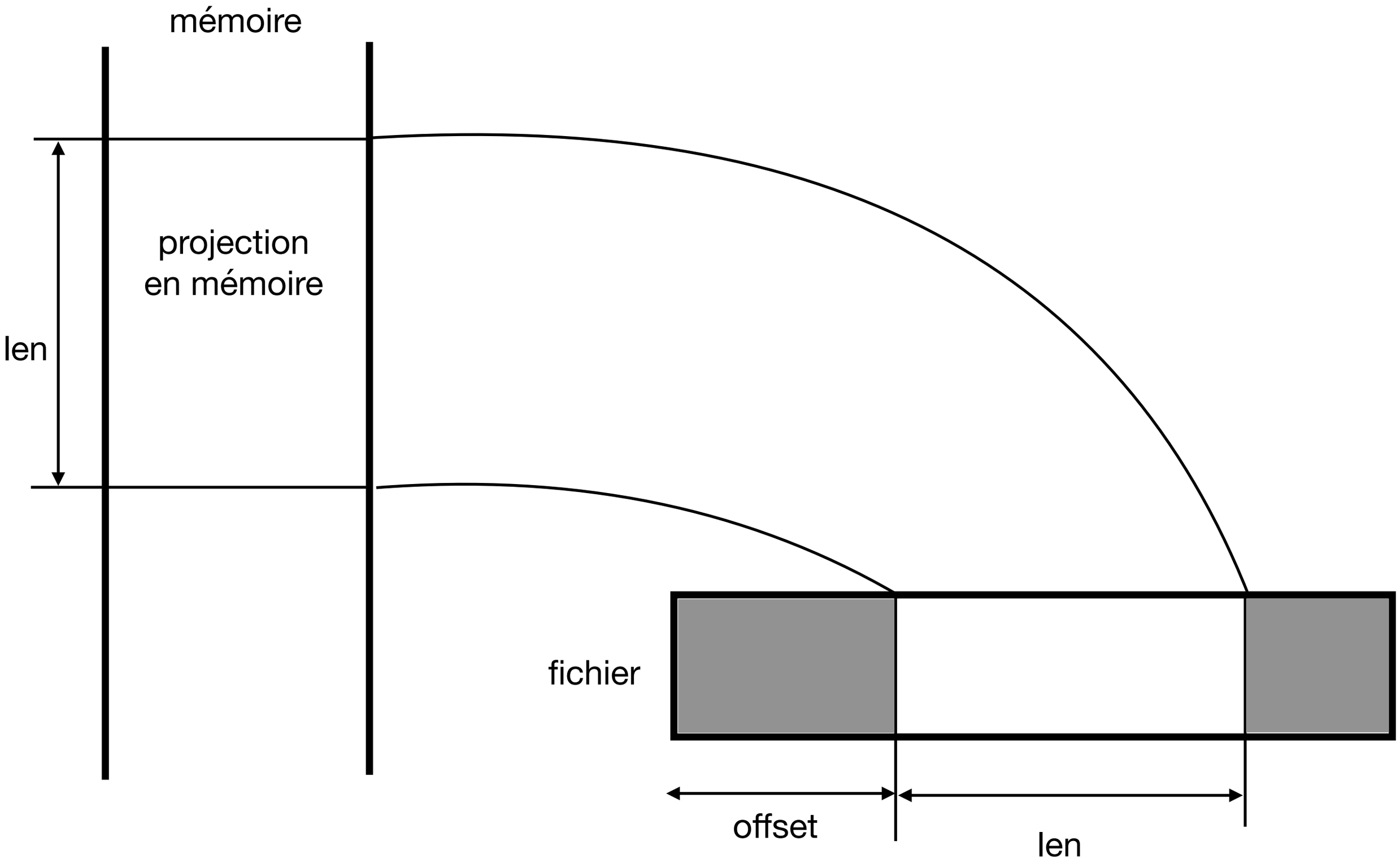
retourne l'adresse de mémoire en cas de succès et `MAP_FAILED` en cas d'erreur

**`fd`** - un descripteur de fichier valable

**`offset`** - la position du début du segment de fichier projeté en mémoire

**`len`** - la longueur du segment

# projection de fichier en mémoire



# paramètre pro

**prot** - spécifie la protection de la mémoire. Les valeurs possibles :

- PROT_NONE – la région de la mémoire n'est pas accessible ou

- combinaison OR bit à bit de n'importe quel ensemble parmi les trois valeurs possibles :

  - PROT_READ - le contenu peut être lu
  - PROT_WRITE - le contenu peut être modifié
  - PROT_EXEC - le contenu peut être exécuté.

## paramètre flags

**`flags`** - un de deux drapeaux :

- MAP_SHARED  création de mémoire partagée. Modifications de la mémoire sont visibles par d'autres processus qui partagent  la même région avec MAP_SHARED. Le modification de la mémoire sont répercutées dans le fichier mais peut-être pas immédiatement. Voir *`msync()`*.

- `MAP_PRIVATE`  modification du contenu de la mémoire n'est pas visible par d'autre processus et n'est pas répercutée sur le contenu du fichier sur le disque.

Si un processus essaie d'accéder à la mémoire de manière inconsistante avec la valeur de prot, par exemple écrire quand la protection est établie avec `prot==PROT_READ`

alors le processus recevra le signal `SIGSEGV` (ou `SIGBUS` dans certaines implementations).

A quoi sert les pages de projections avec

prot == PROT_NONE ?

Utilisées comme de page de garde au début et/ou à la fin de la zone allouée. Si le processus  dépasse par inadvertance la mémoire utile et entre dans la mémoire protégée par PROT_NONE  il recevra le signal approprié.

Les différents processus peuvent marquer la même page avec des droits différents, un avec PROT_READ et l'autre avec PROT_WRITE.

*`mprotect()`* peut changer le niveau de protection prot.

`PROT_READ et PROT_EXEC` exigent que le fichier projeté soit ouvert en `O_RDONLY ou O_RDWR.`

`PROT_WRITE` exige que le fichier soit ouvert en `O_WRONLY ou O_RDWR.`

Mais parfois il y a de contraintes imposées par l'architecture, par exemple dans certains systèmes `PROT_WRITE` implique automatiquement `PROT_READ` donc le fichier doit être ouvert en lecture même si on veut juste écrire.

Si MAP_PRIVATE nous pouvons spécifier n'importe quelle protection parce qu'on opère sur la copie qui n'est jamais réécrit dans le fichier.

## paramètre adr

**`adr`** - l'adresse de la mémoire utilisée pour la projection.

Valeur conseillée de adr 0 : le système alloue lui même la mémoire pour la projection.

## taille réelle de la mémoire

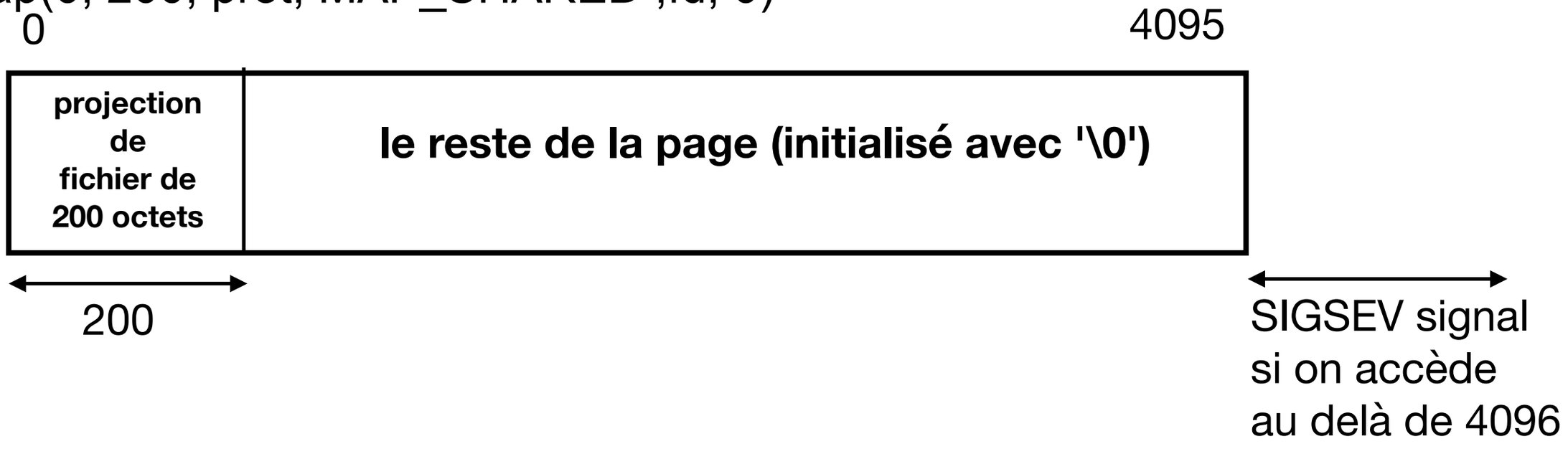
La taille de la mémoire allouée réellement pour la projection peut être supérieure à la taille demandée len, la mémoire effectivement allouée est toujours de taille multiple de la taille d'une page mémoire. La mémoire supplémentaire est initialisée avec '\0'.

Pour trouver la taille d'une page mémoire sur votre système faire appel à :

sysconf(_SC_PAGESIZE)

qui retourne la taille d'une page (valeur de retour long). (Sur mon MacBook 4096).

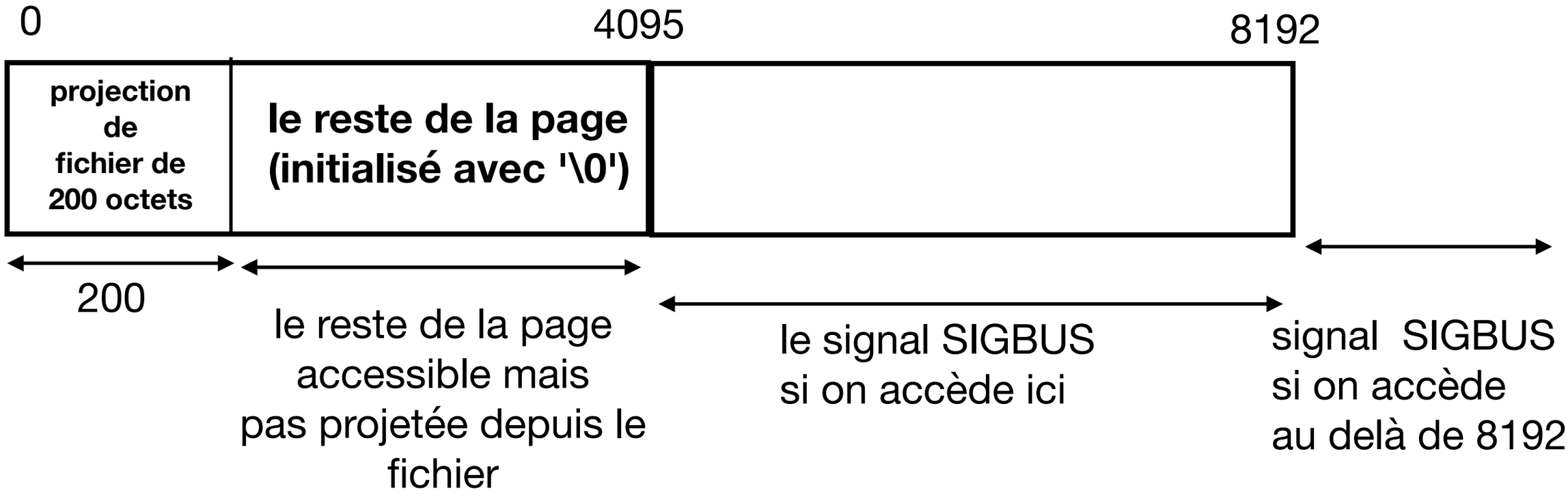
mmap(0, 200, prot, MAP_SHARED ,fd, 0)



# taille réelle de la mémoire

mmap(0, 8192, prot, MAP_SHARED, ,fd, 0)

projection d'un fichier de 200 octets




Le fichier de 200 octets et on demande une projection de taille 8K = 8192.
Seulement 4096 octets de mémoire  sera alloués par mmap().

la mémoire allouée = taille de segment de fichier  arrondie vers le haut au multiple de la taille de page.

## synchronizer la projection MAP_SHARED avec le fichier

Le noyau synchronise automatiquement la mémoire partagée MAP_SHARED avec le fichier après les modifications de la mémoire, mais aucune garantie quand la synchronisation sera effectuée.

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

int msync(void *adr, size_t len, int flag)
```

`msync()` demande explicite de synchronisation de fichier avec la mémoire (écriture dans le fichier)

**`adr`** `et` **`len`** pour spécifier la région de mémoire à synchroniser.

**`adr`** doit être aligné avec la page mémoire et **len** sera arrondi au multiple de la taille de page**.**

## synchronizer la projection MAP_SHARED avec le fichier

`Les flags dans msync:`

- **`MS_SYNC`** – le processus bloqué tant que la synchronisation n'est pas terminée

- **`MS_ASYNC`** – l'écriture dans le fichier asynchrone, peut-être plus tard quand ce sera possible. Pas de bloquage de processus.

- **`MS_INVALIDATE`** – les pages dans la mémoire inconsistantes avec le fichier sont invalidées. Quand le processus accède à une page invalidée le contenu de a page est lu depuis le fichier. De cette façon le processus peut "apprendre" ce qu'un autre processus a écrit dans le fichier.

msync() n'a pas d'effet sur les projections anonymes

# supprimer la projection mémoire

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

int munmap(vois *adr, size_t len)
```

La projection mémoire d'un processus est automatiquement supprimée quand le processus termine. unmap() permet de supprimer explicitement la projection mémoire.

Projection MAP_PRIVATE est simplement écartée.

Le contenu de la mémoire MAP_SHARED sera écrit dans le fichier (si la mémoire a été modifiée) mais le moment de l'écriture n'est pas déterminé.

# exemple

On suppose que le fichier a 100 octets.

```
size_t len = 1200;

char *mem = mmap(0, len, PROT_READ| PROT_WRITE,
                 MAP_SHARED, fd, 0);

memset( mem, 'x', len);  /* 1200 caractères 'x' dans mem*/

msync(mem,  len, MS_SYNC);
```

Est-ce len==1200 caractères 'x' sont écrits dans le fichier?

**Non, on ne peut pas d'allonger (ni raccourcir) la taille de fichier avec mmap().**

msync() remplacera 100 caractères présents dans le fichier par 'x' mais n'écrit rien à la place de caractères qui n'existait pas dans le fichier.

Il faut procéder dans l'ordre suivant :

- d'abord agrandir le fichier à l'aide de **truncate() ou ftruncate()**
- ensuite construire l'image mémoire avec mmap().

# changer la taille de fichier

Le changement de taille de fichier doit être effectuer avant l'appel à mmap.

mmap ne change jamais de taille de fichier.

# changer la taille de fichier

```
#include <unistd.h>

int
ftruncate(int descriptor, off_t longueur)

int
truncate(const char *path, off_t longueur)
```

les deux fonctions modifient la longueur de fichier. Les fonction retournent 0 si succès et -1 si échec.

# changement de niveau de protection

```
int mprotect(void *adr, size_t len,
             int prot)   (0 si OK, -1 sinon)
```

change le niveau de protection à prot sur les pages qui contiennent  l'intervalle qui commence à l'adresse adr et qui est de longueur len.

Les valeurs possibles de prot :

PROT_NONE

ou | (ou bit à bit) de constantes : PROT_READ PROT_WRITE  PROT_EXEC

.