Programmation systèmes avancée
Projection mémoire

Table des matières

1 Introduction

`mmap` crée une projection en mémoire (*mapping*) dans l'espace virtuel d'un processus.
Il y a deux types de projections :

- *projection de fichier* : Une région de fichier est directement copiée dans la mémoire de processus. Une fois la projection effectuée on pourra accéder au contenu du fichier en examinant le contenu de la mémoire.
- *projection anonyme* : Il n'y a aucun fichier correspondant, c'est juste une allocation de mémoire dont les octets sont initialisés à 0.

La mémoire d'une projection peut être partagée par plusieurs processus ou elle peut être privée :

- *mapping privé* `MAP_PRIVATE` - les modifications de la projection effectuées par un processus ne sont pas visibles par d'autres processus et, dans le cas d'une projection de fichier, les modifications dans la mémoire ne sont pas répercutées dans le fichier (le fichier sert juste pour initialiser la projection).
- *mapping partagé* `MAP_SHARED` – les modifications dans la mémoire de projection sont visibles par d'autres processus et, pour la projection de fichier, les modifications dans la projection sont répercutées dans le contenu du fichier.

Quand un processus fait `fork()` le processus enfant hérite la référence vers la mémoire de projection. Si la projection est de type `MAP_SHARED` le père et l'enfant partagent la mémoire de projection.
Si la projection est de type `MAP_PRIVATE` c'est le mécanisme *copy-on-write* qui est mis en oeuvre, tant que ni père ni enfant ne modifient pas la mémoire les deux partagent la projection mais si un des deux modifie la mémoire de projection le système lui fournit une copie de la mémoire. Le résultat final est que les modifications de la mémoire par le père et l'enfant sont privées et invisibles pour l'autre.
La projection privée ou partagée est toujours perdue à la suite de l'`exec`. Sous Linux les informations concernant les projections sont visibles dans le fichier `/proc/PID/maps` où PID et le pid du processus.

2 Création de projection

```
#include <sys/mman.h>
void *mmap(void *adr, size_t len, int prot, int flags, int fd, off_t offset)
retourne l'adresse de mapping si OK, la constante  MAP_FAILED si échec
```

`adr` l'adresse virtuelle de la mémoire de la projection. Sauf dans certains cas non traités dans ce cours il faut spécifier `NULL` et laisser le système effectuer l'allocation de la mémoire pour la projection.
`len` est la longueur de la projection mesurée en octets. La longueur peut être quelconque mais le système alloue la mémoire par tranche de taille multiple de la taille de la page mémoire. Pour trouver la taille de la page il suffit de récupérer la valeur retournée par `sysconf(_SC_PAGESIZE)`.
`prot` prend soit la valeur `PROT_NONE` soit « OR », bit à bit de constantes `PROT_READ`, `PROT_WRITE` ou `PROT_EXEC`.

`PROT_NONE`	la mémoire est inaccessible
`PROT_READ`	mémoire accessible en lecture
`PROT_WRITE`	mémoire accessible en écriture
`PROT_EXEC`	mémoire accessible en exécution

Dans beaucoup de systèmes, si on spécifie `PROT_WRITE` il faut aussi spécifier `PROT_READ` (la mémoire accessible en écriture doit être accessible en lecture).
`fd` : pour la projection en mémoire d'un fichier, `fd` est un descripteur de fichier ouvert en lecture, et si `PROT_WRITE` est spécifié alors `fd` doit être aussi ouvert en écriture.
Pour une projection anonyme on posera `fd==-1`.
`flags` : soit `MAP_PRIVATE` soit `MAP_SHARED`
`offset` correspond pour la projection d'un fichier au point de depart de la projection dans le fichier. Dans certaines implémentations `offset` doit être multiple de la taille de page, nous allons utiliser toujours `offset==0`.
Pour faire une projection de tout le fichier prendre `offset == 0` et `len` la longueur de fichier :

```
int fd = open(....);
struct stat statBuf;
fstat(fd, &statBuf);
void *adr = mmap(NULL, statBuf.st_size, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE, fd, 0);
if( adr == MAP_FAILED ){
   /* traiter erreur de projection */
}
```

Une fois la projection fichier effectuée nous pouvons fermer le descripteur `fd`.

2.1 Protection mémoire de fichier en détail

Supposons que le fichier `toto.txt` a 5000 octets et que la taille de la page mémoire est $4K$ (4096 octets).

```
int fd = open("toto.txt",O_RDWR);
void *adr = mmap(NULL, 15000, PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, 0);
```

Pour mettre en mémoire les 5000 octets du fichier il suffit deux pages ($2 * 4096 = 8192$ octets). Donc le système alloue deux pages, 5000 premiers octets sont copiés depuis le fichier, le 3192 octets suivants sont initialisés à 0.

Le programmes peut modifier tous les 8192 octets de la mémoire à l'adresse `adr` mais si on effectue `msync` pour réécrire les modifications dans le fichier seulement 5000 premiers octets seront écrits.
La projection de fichier **ne permet ni d'augmenter ni de diminuer** la taille d'un fichier.
La tentative d'accéder aux octets au delà de 8192 provoque l'envoi de signal `SIGSEGV` (dans certaines implémentation `SIGBUS`).

Comme un autre exemple prenons une projection de fichier de longueur 2000 octets à l'aide de

```
1 void *m=mmap(0, 8192, prot, MAP_SHARED, fd, 0);
```

Supposons que les pages de mémoires sont de longueur $4k = 4096$ octets, donc on demande 2 pages mémoire et pour la projection de ce fichier une page mémoire suffit. Dans ce cas `mmap` va allouer juste une page de 4096 octets dont les premiers 2000 octets contiennent la copie du fichier, le reste de la page est initialisé à 0. La tentative d'accès aux octets entre 4096 et 8191 provoque l'envoie de signal `SIGBUS`. La tentative d'accès aux octets à partir de l'octet 8192 provoque l'envoie de signal `SIGSEGV`.

3 Supprimer la projection

```
#include <sys/mman.h>
int munmap(void *adr, size_t len)
retourne 0 si OK,  -1 sinon
```

`munmap` supprime de l'espce de mémoire virtuelle la mémoire obtenu par `mmap`. `adr` est l'adresse initiale de la mémoire à supprimer. Le plus souvent c'est l'adresse renvoyer auparavant par `mmap`. Mais il est possible de libérer une partie de la mémoire et donner l'adresse à l'intérieur de la mémoire pourvu qu'elle soit multiple de la taille de page.
`len` le nombre d'octets de la mémoire à supprimer, qui sera arrondi par le système vers un multiple de la taille de page.
Intuitivement, aussi bien allocation par `mmap` que la suppression avec `munmap` se font par les pages entières.

4 Synchroniser la projection mémoire avec le fichier

Pour les projections de fichier `MAP_SHARED` le noyau de système automatiquement répercute les modifications de la mémoire dans le fichier [1]. Mais il n'y a aucune garantie quand les modifications de la mémoire sera réécrite dans le fichier.

```
#include <sys/mman.h>
int msync(void *adr, size_t len, int flags)
retourne 0 si OK,  -1 sinon
```

`adr` et `len` spécifient l'adresse et la longueur de mémoire à synchroniser avec le fichier. `adr` doit être multiple de la taille de page (l'adresse retournée par `mmap` satisfait cette condition).
`len` est arrondi vers un multiple de la taille de page.
`flags` prend une des trois valeurs suivantes :

- `MS_SYNC` l'appel à `msync` est bloqué jusqu'à ce que le contenu de la mémoire soit écrit dans le fichier

1. Rappelons que la projection de fichier `MAP_PRIVATE` n'est jamais réécrite dans le fichier correspondant.

- `MS_ASYNC` écriture asynchrone non-bloquante. Le contenu de la mémoire sera écrit à un moment indéfini plus tard. L'appel n'est pas bloquant.
- `MS_INVALIDATE` les pages dans la mémoire inconsistantes avec le fichier sont marquées comme invalides. Quand le processus essaie d'accéder à ces pages le contenu de la mémoire partagé sera recopié depuis le fichier. Donc ici l'opération a lieu dans le sens inverse par rapport aux deux cas précédents, au lieu de copier la mémoire vers le fichier, on (ré)copie le fichier vers la mémoire.

4.1 Exemple : copier un fichier à l'aide de `mmap`

Pour copier un fichier à l'aide de `mmap` il faut :

1. ouvrir le fichier source en lecture,
2. trouver la longueur `len` de fichier source à l'aide de `fstat` (ou `stat`),
3. faire la projection de fichier source en mémoire,
4. ouvrir le fichier de destination en lecture et écriture : `O_RDWR|O_CREAT`,
5. ramener la taille de fichier destination à `len` avec `ftruncate`,
6. faire la projection de fichier destination en mémoire en mode `MAP_SHARED`,
7. copier la mémoire de projection source vers la mémoire projection destination avec `memmove`,
8. synchroniser mémoire destination avec le fichier destination avec `msync`.

5 Projections anonymes

La projection anonyme n'est pas vraiment une projection, c'est une allocation de mémoire qui peut être soit privée soit partagée. Cette mémoire n'est pas lié à un fichier. On obtient une projection anonymes en ajoutant la constante `MAP_ANONYMOUS` (ou `MAP_ANON`) dans les flags de `mmap`. Les descripteur `fd` est ignoré mais certains systèmes exigent la valeur -1. La projection anonyme n'est pas définie dans Single UNIX Specification v4 mais est implémenté dans la plupart de systèmes UNIX.
La projection anonyme `MAP_SHARED` effectuée par un processus permet d'accéder à cette mémoire par le processus père et le processus enfant (tant que l'enfant n'ait pas fait exec).

```
void *adr = mmap(NULL, len, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS, -1, 0)
if( adr == MAP_FAILED ){
   perror("mmap"); exit(1);
}
```

Si à la place de `MAP_SHARED` on met `MAP_PRIVATE` on obtient une allocation de mémoire qui n'est pas partagée par d'autres processus. L'implémentation de `malloc` dans *glibc* utilise la projection anonyme privée pour allouer de tranches de mémoire de longueur supérieure à une certaine valeur.
Dans la majorité de systèmes UNIX une fois la projection faite (anonyme ou non, partagée ou privée) il est impossible de modifier sa longueur (mais c'est possible sous Linux).