Programmation Système IV
Memory mapping

Juliusz Chroboczek

27 février 2016

Le système de mémoire virtuelle (voir poly 1) permet de maintenir une vision de la mémoire assez flexible, en particulier de construire des zones de mémoire qui sont partagées entre processus et de donner l'illusion qu'un fichier est dans la mémoire principale. On appelle cette technique le *memory mapping*[1].

1 Rappel sur la mémoire virtuelle

Le système maintient l'illusion d'une mémoire toujours disponible à l'aide des bits valide et protégé en écriture utilisés par le MMU. Lorsque l'intervention du système est nécessaire, il consulte des structures de données dont la nature exacte dépend du système, mais dont l'effet est d'associer à chaque page de mémoire virtuelle les attributs suivants :

- une page peut être *file-backed*, signifiant qu'elle sera initialisée à partir d'un fichier, ou *anonyme*, signifiant qu'elle sera initialisée avec la valeur 0 et sauvegardée en *swap* ;
- une page peut être *partagée*, signifiant qu'elle sera partagée à travers *fork* et, si elle est *file-backed*, écrite dans le fichier sous-jacent *(backing file)*, ou alors *privée*, signifiant qu'elle sera marquée pour un *copy-on-write* en cas de *fork* ;
- une page a des permissions et si elles sont violées le processus coupable sera tué à l'aide d'un signal `SIGSEGV`.

2 *Memory mapping*

L'appel système qui permet à un programme utilisateur de manipuler la mémoire virtuelle s'appelle `mmap` :

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

1. Je ne connais pas de terme français. Dominique utilise *projection de mémoire*, ce qui est une bonne traduction mais pas à ma connaissance un terme standard.

Le paramètre `addr` vaudra toujours NULL, ce qui demande au système de choisir une adresse libre pour le *mapping*. Le paramètre `length` indique la taille du *mapping* (le système peut choisir une taille plus grande, il arrondit à un nombre entier de pages). Le paramètre `prot` indique les permissions. Les paramètres `fd` et `offset` indiquent le fichier utilisé par un *mapping* qui est *file-backed* ; ils sont ignorés pour un *mapping* anonyme, mais il est conventionnel dans ce cas de les positionner à -1 et 0 respectivement[2]. Cet appel système retourne `MAP_FAILED` en cas d'échec (pas `NULL` !), et alors `errno` est positionné.

Une zone de mémoire peut être marquée invalide à l'aide de `munmap` :

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

La mémoire physique correspondante est libérée et le contenu est perdu.

2.1 *Mapping* anonyme privé

`mmap` peut servir à allouer de la mémoire anonyme privée :

```
p = mmap(NULL, 16 * 4096,
         PROT_READ | PROT_WRITE,
         MAP_PRIVATE | MAP_ANONYMOUS,
         -1, 0);
if(p == MAP_FAILED)
    ...
```

Cette technique est normalement utilisée par `malloc` pour obtenir de la mémoire, mais rien n'empêche un programme utilisateur de contourner `malloc` par exemple s'il implémente un allocateur de mémoire spécialisé.

2.2 *Mapping* anonyme partagé

`mmap` peut servir à allouer de la mémoire anonyme partagée, ce qui permet à des processus de communiquer sans effectuer de copies à travers le tampon d'un tube. Attention cependant aux *race conditions* et aux problèmes de cohérence de la mémoire — voir poly suivant.

```
p = mmap(NULL, 16 * 4096,
         PROT_READ | PROT_WRITE,
         MAP_SHARED | MAP_ANONYMOUS,
         -1, 0);
if(p == MAP_FAILED)
    ...
```

Une autre technique pour obtenir de la mémoire partagée est d'utiliser les *threads*, ce que nous verrons dans la deuxième partie de ce cours.

2. Parce que Sun OS 4.

2.3 *Mapping* de fichiers privé

`mmap` permet de donner l'illusion qu'un fichier se trouve en mémoire principale :

```
fd = open(... O_RDONLY ...);
if(fd < 0)
    ...
rc = fstat(fd, &st);
if(rc < 0)
    ...
p = mmap(NULL, st.st_size,
         PROT_READ,
         MAP_PRIVATE,
         fd, 0);
if(p == MAP_FAILED)
    ...
```

Une lecture de la zone de mémoire retourne la partie correspondante du fichier sous-jacent (mais attention aux problèmes de cohérence, voir poly suivant). Une écriture (si les permissions le permettent) cause la copie d'une page — elle n'est pas réflétée dans le fichier sous-jacent.

2.4 *Mapping* de fichiers partagé

Enfin, `mmap` peut donner l'illusion qu'un fichier se trouve en mémoire principale et permettre de le modifier :

```
fd = open(... O_RDWR ...);
if(fd < 0)
    ...
rc = fstat(fd, &st);
if(rc < 0)
    ...
p = mmap(NULL, st.st_size,
         PROT_READ | PROT_WRITE,
         MAP_SHARED,
         fd, 0);
if(p == MAP_FAILED)
    ...
```

Une écriture dans la zone de mémoire correspondante est en principe réflétée dans le fichier sous-jacent et dans les autres *mappings* du même fichier — attention cependant aux problèmes de cohérence, cette écriture peut s'effectuer arbitrairement tard.