Programmation systèmes avancée
Objets mémoire POSIX (Shared Memory Objects)

Table des matières

1 Création et ouverture de l'objet mémoire **1**
1.1 Fixer la taille de l'objet mémoire 2
1.2 fstat sur le descripteur objet mémoire 2

2 Lire le contenu/modifier l'objet mémoire **2**
2.1 avec read/write 2
2.2 avec mmap 3

3 Durée de vie et suppression de l'objet mémoire **3**

4 Suppression l'objet mémoire **3**

5 Compilation **3**

6 Coordination d'accès aux ressources partagées **4**

7 Remarques **4**

1 Création et ouverture de l'objet mémoire

L'objet mémoire POSIX peut être vu comme une sorte de fichier. La différence entre un objet mémoire et un fichier est que le fichier est stocké sur la mémoire externe, par exemple un disque, clé USB, tandis que l'objet mémoire réside uniquement dans la mémoire vive (RAM) de l'ordinateur.
L'ouverture et, éventuellement la création, de l'objet mémoire s'effectue à l'aide de

```
#include <fcntl.h>    /* pour les constantes O_ */
#include <sys/stat.h> /* pour les constantes droits d'acces */
#include <sys/mman.h> /* pour shm_open */
int  shm_open(cont char *name, int oflag, mode_t mode);
retourne un descripteur fichier si OK,  -1 sinon
```

`shm_open` retourne un descripteur sur l'objet mémoire.
Dans Linux les objets mémoire sont implémentés comme des fichiers virtuels (le système de fichiers *tmpfs* monté dans le répertoire `/dev/shm` ou `/run/shm`). Mais ce n'est pas forcement le cas pour tous les systèmes POSIX (par exemple ce n'est pas le cas dans MacOS).
L'argument `name` est le nom de l'objet mémoire. Le nom doit commencer par le caractère `/` suivi par des caractères differents de `/`, par exemple `/monobjet`. Linux et MacOS suivent cette règle concernant les noms d'objet mémoire.
Le paramètre `oflag` est un « or » bit-à-bit de drapeaux `O_CREAT`, `O_EXCL`, `O_RDONLY`, `O_RDWR`, `O_TRUNC`[1].

1. MacOS ne supporte pas `O_TRUNC`

La sémantique de ces constantes est identique à celle quand les constantes sont utilisées pour ouvrir un fichier avec `open`.
Le paramètre `mode` indique les permission d'accès à l'objet. Les valeurs possibles sont les mêmes que pour les troisième paramètre de `open`. Le troisième paramètre est ignoré si on ouvre un objet mémoire existant.

```
int fd = shm_open("/monobjet", O_CREAT | O_RDWR | O_EXCL,
                  S_IRUSR | S_IWUSR );
```

ouvre objet mémoire dont le nom est `/monobjet`, l'objet est créé s'il n'existe pas (`O_CREAT`), et il sera ouvert en lecture et écriture (`O_RDWR`). Si l'objet existe déjà alors `shm_open` échoue (`O_EXCL`). Quand le nouveau objet est créé le propriétaire aura les permissions de lecture et écriture (mais les permissions sont modifiées par le masque `umask`).

Les objets mémoire sont « persistants », il faut les supprimer explicitement (comme il faut le faire pour les fichiers). Et tant que l'objet mémoire n'est pas supprimé il garde son contenu. Donc un processus PA peut stocker des octets dans l'objet mémoire et terminer mais le contenu de l'objet mémoire sera préservé après la terminaison de PA. Si, après un certain temps, un autre processus PB ouvre l'objet mémoire il retrouvera le contenu écrit par PA. C'est comportement est le même comme pour le fichier.
Il y a cependant une différence par rapport aux fichiers ordinaires. Les objets mémoire ne résident pas dans la mémoire externe (disque dur ou carte SSD) mais ils sont mémorisés dans la mémoire vive RAM. Quand le système s'arrête ou reboote les objets mémoire ne survivent pas à cette opération et disparaissent.
Cela explique l'utilité de `O_TRUNC`. Si l'objet existe ce flag permet d'effacer son contenu (mais pas sur MacOS).
Le flag `FD_CLOEXEC` est positionné automatiquement sur le descripteur retourné par `shm_open`, ce qui implique qu'en effectuant `exec` le processus perd les descripteurs ouverts sur les objets mémoires. (Mais le flag `FD_CLOEXEC` peut être annulé avec `fcntl`).

1.1 Fixer la taille de l'objet mémoire

Au moment de la création la taille l'objet mémoire la taille de l'objet mémoire créé est 0. Donc après la création de l'objet il convient d'appeler immédiatement `ftruncate` pour lui attribuer une taille > 0 et cela avant qu'on fasse l'appel à `mmap` pour projeter l'objet en mémoire.

1.2 fstat sur le descripteur objet mémoire

Quand on appelle `fstat` sur le descripteur de l'objet mémoire, les champs `st_size`, `st_mode`, `st_uid` et `st_gid` de `struct stat` acquièrent les informations habituelles (la longueur de l'objet, les droits d'accès, le propriétaire et groupe propriétaire).

2 Lire le contenu/modifier l'objet mémoire

2.1 avec read/write

Sur Linux une fois objet mémoire ouvert on peut utiliser les opérations `read` et `write` sur son descripteur comme pour les fichiers. Mais cela n'est pas un comportement préconisé par POSIX. Par exemple sur MacOS il est impossible d'effectuer `read/write` sur les objets mémoire même si MacOS est conforme à Single Unix Specification version 4.

2.2 avec mmap

La méthode conforme à Single Unix Specification pour accéder à un objet mémoire passe par une projection en mémoire avec `mmap` (comme pour les fichiers ordinaires).
En particulier cette méthode qui marche aussi bien sous Linux que MacOS.
(Cela semble bien étrange. Après tout, une fois créé l'objet mémoire réside déjà dans la mémoire vive, donc pourquoi le faire projeter ? Intuitivement, l'objet mémoire réside dans la mémoire du noyau et la projection de l'objet crée son image dans la mémoire virtuelle du processus.)
Dans le code suivant on ouvre l'objet mémoire existant et on fait sa projection en mémoire avec `mmap` :

```
int fd = shm_open("/monobjet", O_RDWR, 0 );
struct stat bufStat;
fstat(fd, &bufStat);
void *adr = mmap(NULL, bufStat.st_size, PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, 0) ;
```

N'oubliez pas que, comme explique la section 1.1, si `shm_open` utilise `O_CREAT` il faut un appel à `ftruncate` après `shm_open` et mais avant `mmap`.
Après la projection nous pouvons accéder à la projection en utilisant l'adresse `adr`.
Comme pour la projection de fichiers, puisque la projection dans l'exemple est `MAP_SHARED` de temps en temps le noyau écrit le contenu de la projection vers l'objet mémoire. Pour forcer l'écriture de la mémoire partagée vers l'objet mémoire on utilise `msync`.

3 Durée de vie et suppression de l'objet mémoire

Les objets mémoire ne résident pas dans la mémoire externe (disque dur ou mémoire SSD) mais dans la mémoire du noyau. Donc ces objets possèdent la persistence noyau, une fois créés ils existent en gardant le contenu tant que le système ne soit pas rebooté.
Nous pouvons comparer cela avec le tubes nommés (fifo). Quand il n'y a plus de processus connecté à fifo le fifo perd son contenu. Par contre le fifo n'est pas détruit quand on reboote le système.

4 Suppression l'objet mémoire

On peut explicitement supprimer l'objet mémoire à l'aide de

```
#include <sys/mman.h> /* pour shm_open */
int  shm_unlink(cont char *name);
retourne 0 si OK,  -1 sinon
```

L'appel à `shm_unlink` n'affecte pas les projections mémoire déjà existantes, ni les descripteurs déjà ouverts, mais il rend impossible la réouverture de l'objet avec `shm_open`. Une fois toutes les projections supprimées avec `munmap` l'objet mémoire sera effectivement supprimé et son contenu perdu.

5 Compilation

Sous Linux, à l'étape d'edition de liens, les programmes utilisant les objets mémoire doivent spécifier `-lrt` pour ajouter la bibliothèque real-time *librt* pour avoir accès à `shm_open, shm_unlink` (dans Makefile on ajoutera la ligne `LDLIBS=-lrt`).
Cette remarque ne concerne pas MacOS.

6 Coordination d'accès aux ressources partagées

Si plusieurs processus accèdent à la mémoire partage (ou une autre ressource partagée) il faut utiliser un méchanisme qui permet de coordonner les accès. Quand un processus modifie la mémoire partagée les autres ne doivent pas y accéder. Quand un processus accède au contenu de la mémoire partagée sans modification (une « lecture » de la mémoire partagée) d'autres processus ne doivent pas modifier cette mémoire.
Pour coordonner les accès de plusieurs processus à la mémoire partagée on utilisait les verrous, soit les verrous POSIX (verrous fcntl) soit les verrous BSD flock.
Pour cordonner les accès à la mémoire partagée il y a essentiellement trois méthodes possible :

verrous fcntl : La première méthode consiste à créer un fichier vide qui sera associé à la mémoire partagée.

Au lieu de poser une sorte de verrou sur la mémoire partagée on pose le verrous sur le premier octet du fichier associé à l'aide de `fcntl`.

Le processus qui possède le verrou fait l'opération sur la mémoire partagée.

Quand le processus termine l'utilisation de la mémoire partagée il lève le verrou sur le fichier associé.

Si nous avons à implémenter les accès partagés sur plusieurs régions de la mémoire partagée il suffit de faire correspondre à chaque région de la mémoire un octet de fichier vide. Avant d'accéder à la région de la mémoire partagée le processus posera le verrous sur l'octet correspondant du fichier.

sémaphores : On peut coordonner les accès à la mémoire partagée à l'aide de sémaphores,

mutex : Et finalement on peut utiliser les variables mutex partagés entre les processus.

7 Remarques

Sous Linux les objets mémoire créés avec `shm_open` sont créés dans le repertoire `/dev/shm` qui est un système de fichiers virtuel et nous pouvons supprimer les objets mémoire avec la commande `rm` comme les fichiers ordinaires.
Sous MacOS :

- le flag `O_TRUNC` ne s'applique pas,
- l'appel à `ftruncate` échoue si on le fait sur l'objet mémoire qui a déjà une taille supérieure à 0. En pratique cela veut dire qu'il est impossible de changer la taille d'un tel objet, sauf la première fois quand `ftruncate` est appliqué à l'objet de taille 0.
  Donc sur MacOs pour changer la taille d'un objet mémoire if faut le détruire avec `shm_unlink` et le reconstruire en appelant `shm_open` suivi de `ftruncate`.