Programmation Système avancée

Wieslaw Zielonka
zielonka@irif.fr

Résumé de cours précédent

Avec les méthodes de cours précédent pour obtenir une mémoire partagée entre des processus il faut :

- soit disposer d'un fichier qui nous sert pour initialiser la mémoire (et qui par le mécanisme de synchronisation permet de sauvegarder le contenu de la mémoire partagée)

- soit utiliser la mémoire partagée anonyme. Mais cela permet uniquement le partage de mémoire entre un processus et ces descendants (à condition que les descendants n'exécutent jamais exec()). Cette méthode ne fait pas partie de Single Unix Specification.

- Ces deux problèmes résolues grâce aux objets mémoire POSIX

Et il nous manque toujours un mécanisme de synchronisation entre les processus qui partagent la mémoire.

objets mémoire POSIX

Objets mémoire

Objets mémoire peuvent peuvent être vu comme de fichiers et dans Linux ils sont implémentés comme fichiers dans un système de ficher virtuel /dev/shm .

```
int shm_open(char *name, int oflag, mode_t mode)
```

crée et ouvre un objet mémoire et retourne le descripteur vers cet objet ou -1 en cas d'erreur.

`name` - le nom de l'objet. Ce nom doit commencer par '/' et ne doit pas contenir d'autres caractères '/'

`oflag` - bitmask de O_CREATE, O_EXCL, O_RDONLY, O_RDWR, O_TRUNC avec la même signification que pour open sur les fichiers ordinaires. Si O_CREATE n'est pas spécifié alors le fichier doit déjà exister.

`mode` - les permissions, les mêmes valeurs que pour les fichiers ordinaires.

Objets mémoire

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

ouverture de l'objet mémoire "/monobjet" avec la création si l'objet n'existe pas.

L'objet sera ouvert en lecture et écriture.

Si l'objet n'existe pas le nouveau objet aura les permission read+write pour le propriétaire.

Objets mémoire

`fstat()` peut être appliqué sur un objet mémoire ouvert et le champ `st_size` de `struct stat` donne la taille de l'objet mémoire en octet :

```
int fd= shm_open("/monobjet", O_RDWR, 0);

struct stat bufStat;
fstat( fd, &bufStat);
printf( "longueur objet %d\n",
        (int) bufStat.st_size);
```

Objets mémoire

La taille d'objet mémoire juste après la création est `0`.

Donc la première chose à faire après la création c'est appliquer `ftruncate()` pour fixer la taille de l'objet mémoire

```
int fd = shm_open("/monobjet",
                  O_CREAT | O_RDWR ,
                  S_IRUSR | S_IWUSR);
if( fg == -1 ){ perror("shm_open"); exit(1);}

off_t len = 1024;

if( ftruncate( fd, len) == -1)
{perror("ftruncate"); exit(1);}
```

modifier/lire objet mémoire

Sous Linux on peut appliquer read()/write() pour lire et écrire dans un objet mémoire, comme pour les fichiers ordinaires. Mais cette possibilité vient de l'implémentation d'objets mémoire sous linux, sous linux les objets mémoire sont implémentés par les fichiers virtuels.

Single Unix Specification permet l'implémentation des objets mémoires pas des fichiers mais permet aussi d'autres implémentations.

Par exemple, **il n'est pas possible** d'utiliser read/wrire sur les objets mémoire sous MacOS où les objets mémoires sont implémentés mais pas comme les fichiers.

Comme Linux implémente les objets mémoire comme les fichiers virtuels, sous Linux il est possible d'ouvrir un objet existant et changer sa taille à l'aide de ftruncate().

Sous MacOS, on peut (et il faut) appliquer ftruncate seulement après la création de l'objet mémoire quand sa taille est 0. La tentative de modifier la taille de l'objet qui a déjà une taille différente de 0 échoue.

modifier/lire objet mémoire

La méthode applicable sur tous les systèmes : faire la projection de l'objet mémoire dans l'espace d'adressage du processus avec `mmap()` :

```
int fd = shm_open(”/monobjet” , ORDWR, 0 );
struct stat bufStat ;
fstat(fd, &bufStat);
void *adr = mmap(NULL, bufStat->st_size ,
                 PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, 0) ;
```

Mais n'oubliez pas `ftruncate()` si `O_CREAT` est spécifié dans `shm_open()`, `ftruncate()` qui doit être appliqué **avant** `mmap()`

durée de vie d'objet mémoire

L'objet mémoire est supprimé avec

```
int  shm_unlink(cont char *name);
```

L'objet mémoire sera effectivement supprimé quand il n'y a plus de processus qui possèdent une référence vers cet objet.

Sous Linux on peu retrouver les objets mémoire dans le répertoire `/dev/shm` et le supprimer avec la commande `rm`.

Si l'objet mémoire n'est pas supprimé alors **il garde son contenu jusqu'au redémarrage (reboot)** du système (comparer avec les tubes nommés `fifo`).

compilation sous Linux

Les objets mémoire sont implémentés dans la bibliothèque real-time donc dans l'étape de l'édition de liens ajouter

```
-lrt
```

ou encore mieux dans Makefile

```
LDLIBS = -lrt
```

(Cette remarque ne concerne pas MacOS).

protéger l'accès à la mémoire partagée avec les variables mutex et les conditions

synchronisation de processus qui utilisent la mémoire

Deux possibilités pour synchroniser les accès des processus à la mémoire partagée :

- Les sémaphores
- les variables mutex et conditions

les variables mutex

Les variables mutex et conditions utilisées surtout pour synchroniser l'accès à la mémoire entre les threads du même processus.

Mais on peut utiliser les variables mutex et condition pour synchroniser l'accès à la mémoire partagée entre les processus .

Pour que les processus puissent accéder aux variables mutex et condition il faut que ces variables elles-mêmes résident dans la mémoire partagées obtenue grâce à mmap() de type MAP_SHARED.

compilation et traitement d'erreurs

`#include <pthread.h>` //dans les fichiers source

`LDLIBS = -pthread` #dans le Makefile

les fonctions de pthread n'utilisent pas `errno`, elles retournent une valeur int :

0 - si l'appel réussie; numéro d'erreur sinon.

```
int n=pthread_fonction( ...);
if( n != 0){ //traiter erreur
     char *s= strerror( n ); //recuperer le message d'erreur
     fprintf(strerr, "%s\n", s);
    exit ?  return ? ou autre action
}
```

le schéma d'appel
de fonctions la bibliothèque pthread

la variable mutex

les variables mutex et condition doivent être initialisées avant l'utilisation

Nous allons utiliser les fonctions suivantes :

```
int initialiser_mutex(pthread_mutex_t *pmutex)
int initialiser_cond(pthread_cond_t *pcond)
```

pour initialiser ces variables.

initialisation de mutex

```
// la fonction pour initialiser une variable mutex

int initialiser_mutex(pthread_mutex_t *pmutex){
  pthread_mutexattr_t mutexattr;
  int code;
  if( ( code = pthread_mutexattr_init(&mutexattr) ) != 0)
    return code;

  if( ( code = pthread_mutexattr_setpshared(&mutexattr,
                    PTHREAD_PROCESS_SHARED) ) != 0)
    return code;
  code = pthread_mutex_init(pmutex, &mutexattr)  ;
  return code;
}
```

initialisation de condition

```
//initialisation de variable condition


int initialiser_cond(pthread_cond_t *pcond){
  pthread_condattr_t condattr;
  int code;

  if( ( code = pthread_condattr_init(&condattr) ) != 0 )
    return code;
  if( ( code = pthread_condattr_setpshared(&condattr,
                                           PTHREAD_PROCESS_SHARED) ) != 0 )
    return code;

  code = pthread_cond_init(pcond, &condattr) ;
  return code;
}
```

la variable mutex

Les variables mutex et conditions doivent être accessibles par chaque processus qui les utilise donc elle forcement doivent résider elles-mêmes dans la mémoire partagée obtenues par `mmap()` de type `MAP_SHARED`.

La variable mutex est toujours dans un de deux états : verrouillée (lock) ou déverrouillée (unlock). Juste après initialisation elle est déverrouillée.

opérations sur mutex

int
pthread_mutex_lock(pthread_mutex_t *mutex)

si mutex est dans l'état verrouillé alors le processus bloque sur l'opération `pthread_mutex_lock`.

si mutex dans l'état déverrouillé alors `pthread_mutex_lock` met le mutex dans l'état verrouillé et le processus continue l'exécution

int
pthread_mutex_unlock(pthread_mutex_t *mutex)
met le mutex dans l'état déverrouillé.

protéger une section critique avec mutex

```
int r;
r = pthread_mutex_lock( pmutex );

if( r!= 0){ //traiter erreur }
```

section critique :

opérations sur la mémoire partagée

```
r = pthread_mutex_unlock( pmutex );
if( r!= 0){ //traiter erreur }

pmutex – pointeur vers un  mutex
```

règles d'utilisation de mutex

- C'est **le même processus** qui a verrouillé le mutex avec `pthread_mutex_lock()` qui doit le déverrouiller avec `pthread_mutex_unlock()`.

- Le processus qui détient le verrou sur un mutex ne doit jamais exécuter `pthread_mutex_lock()` sur le même mutex (mais il existe des mutex récursifs).
(Avant de faire un nouveau appel à `pthread_mutex_lock()` doit d'abord déverrouiller le mutex).

opérations lock - variantes

int pthread_mutex_trylock(pthread_mutex_t *_mutex_);

fait la même chose que pthread_mutex_lock() si le processus peut verrouiller le mutex.
Quand le mutex est déjà verrouillé pthread_mutex_trylock() retourne tout de suite la valeur EBUSY. La tentative de verrouillage échoue.

int pthread_mutex_timedlock(pthread_mutex_t *restrict _mutex_,
const struct timespec *restrict _abstime_);

attente sur un mutex limité en temps. La fonction retourne EAGAIN si l'obtention de mutex impossible parce qu'un autre processus détient le mutex.

conditions et mutex

```
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex)
```

cond – pointeur vers une variable de type condition (`pthread_mutex_cond`)

1. le thread qui exécute `pthread_cond_wait()` doit détenir le verrou sur le mutex,
2. il sera suspendu en attente d'un signal (il ne s'agit pas de signal POSIX) et au même temps il déverrouille le `mutex` (il est suspendu et tant qu'il est suspendu il ne va pas utiliser les données, donc le verrous est déverrouillé).

conditions et mutex

```
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex)
```

La fonction crée une association temporaire entre les variables **cond** et **mutex.**
Intuitivement, le processus appelant est suspendu sur la variable **cond** où plutôt sur le couple**(cond, mutex).**

Pendant ce temps un autre processus **ne doit pas** appeler **pthread_cond_wait()** avec la même variable condition **cond** et un autre **mutex.** Le plus commode est d'utiliser toujours la variable **cond** avec le même mutex.

le schéma d'utilisation de mutex et condition

Supposons que **`condition`** est une condition à vérifier à l'entrée de la section critique.

- **`condition`** utilise les variables partagées donc pour vérifier si **`cond`** est vrai il faut déjà obtenir le verrou mutex
- le processus suspendu sur un appel à **`pthread_cond_wait()`** peut-être réveillé même si la condition **cond** est toujours fausse, donc **il faut toujours revérifier la condition.**

conditions et mutex

`int pthread_cond_signal( pthread_cond_t *cond)`

provoque l'envoie de signal vers des processus suspendus par **`pthread_cond_wait()`** et réveille un des processus suspendu sur **`cond.`**

Le processus réveillé réacquiert le mutex associé.

La norme permet le comportement où la réception de signal réveille plusieurs processus en attente.

broadcast ou signal

```
int pthread_cond_broadcast( pthread_cond_t *cond )
```

réveille tous les processus qui attendent sur **cond**

le schéma d'utilisation de mutex et condition

```
int code;

/*  mutex_lock */
if( (code = pthread_mutex_lock( &mutex ) ) != 0 )
      thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );


/* attendre la condition, tjrs dans la boucle */
while( !cond )
   if( ( code = pthread_cond_wait( &cond, &mutex) ) > 0 )
         thread_error( __FILE__, __LINE__, code, "pthread_cond_wait" );

/* section critique : faire les opérations sur la mémoire partagée */

/* mutex unlock */
if( ( code = pthread_mutex_unlock(&mutex)) != 0)
       thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );

/* signaler la sortie de la section critique */
if( ( code = pthread_cond_signal( &wcond ) ) > 0 )
      thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );

Souvent la condition exigée à l'entrée n'est pas la mêmz que la
condition signalé à la sortie.
```

traitement d'erreur

```
/*  file : nom de fichier source
 *  line : le numero de ligne de code
 *  code : le code d'erreur
 *  txt  : le message à afficher
 */

void thread_error(const char *file, int line, int code, char *txt){
  if( txt != NULL )
    fprintf( stderr,  "[%s] in file %s in line %d :  %s\n",
       txt, file , line, strerror( code ) );
  else
    fprintf( stderr,  "in file %s in line %d :  %s\n",
       file , line, strerror( code ) );
  exit(1);
}
```

Exemple : producteur consommateur

exemple d'utilisation

la structure stockée dans la mémoire partagée :

```
struct mess{
  pthread_mutex_t mutex;
  pthread_cond_t read_cond;
  pthread_cond_t write_cond;
  unsigned char libre; /* 1 si data libre, 0 sinon */
  pid_t pid; /* le pid de producteur */
  int data;
}
```

- le processus processus producteur écrit la valeur **`data`** `et son` **`pid.`**
- `Le consommateur lit` **`data.`**
- Chaque lecture `consomme` data, `libre == 1` signifie que data est consommé et libre == 0 signifie qu'il y a data à lire.
- Le producteur peut écrire data uniquement quand `libre == 1`
- Le consommateur peut lire data quand `libre == 0`

Producteur et consommateur écrivent et lisent à tour de rôle.