Programmation Système avancée

Wieslaw Zielonka
zielonka@irif.fr

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

mémoire partagée

Trois possibilités pour obtenir une mémoire partagée par plusieurs processus :

1. les processus effectue une projection mémoire mmap du même fichier

2. un processus crée une mémoire avec mmap anonyme de type MAP_SHARED et il fork(). Le parent et l'enfant partage la mémoire. (Mais si l'enfant ou parent fait exec il perd cette mémoire partagée.)

3. Si un processus fait une projection mémoire de type MAP_SHARED d'un shared memory object, il partage cette mémoire avec d'autres processus qui font la même projection.

synchronisation de processus qui utilisent la mémoire partagée

Deux types de primitives pour synchroniser les accès des processus à la mémoire partagée obtenue par mmap de type MAP_SHARED :

1.Les sémaphores

2. les variables mutex et conditions

Principe général : les primitives de synchronisation doivent être accessibles à tout processus, cela implique qu'il y a deux possibilités :

1. les primitives de synchronisation possèdent un nom et sont identifiés et accessible grâce à ce nom. C'est le cas de sémaphores nommés.

2. les primitives de synchronisation n'ont pas de nom et sont accessibles parce qu'ils résident eux-mêmes dans la mémoire partagée : sémaphores anonymes, les variables mutex et condition.

les variables mutex

Les variables mutex et conditions peuvent être utilisées pour synchroniser l'accès à la mémoire des threads du même processus.

Mais dans ce cours nous allons utiliser les variables mutex et condition pour synchroniser l'accès à la mémoire partagée par des 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

Les fonctions de gestion de mutex et condition se trouvent dans la bibliothèque pthreads.

faire inclusion :

```
#include <pthread.h>
```

et dans le Makefile ajouter la bibliothèque :

```
LDLIBS = –pthread
```

Les fonctions de la bibliothèque pthread n'utilisent pas de variable `errno`, elles retournent une valeur int :

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

compilation et traitement d'erreurs

Le schema d'appel d'une fonction de la bibliothèque pthread :

```
int n = pthread_fonction( ...);

if( n != 0){ //traiter erreur
/* récupérer et afficher le message d'erreur */
    char *s= strerror( n );
    fprintf(stderr, "%s\n", s);
    exit ?  return ? abort ? ou autre action
}
```

la variable mutex

Avant l'utilisation les variables mutex et condition doivent être initialisées.

Pour les initialiser j'utiliserai les fonctions suivantes :

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

initialisation de mutex

```
/* la fonction pour initialiser une variable mutex
 * utilisée pour synchroniser des processus */

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

  code = pthread_mutexattr_setpshared(&mutexattr,
                  PTHREAD_PROCESS_SHARED) ;
  if( code != 0 )  return code;

  code = pthread_mutex_init(pmutex, &mutexattr)  ;
  return code;
}
```

initialisation de variable condition

```
/* la fonction à utiliser pour initialiser une
 * variable condition utilisée pour la
 * synchronisation  des processus */

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

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

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

  return pthread_cond_init( pcond, &condattr ) ;

}
```

Remarques

Important : un seul processus doit initialiser les variables mutex et condition.

Sur MacOS les mutex et variables conditions sont mal supportés pour faire la synchronisation de processus.
La fonction
pthread_condattr_setpshared() n'est pas documenté sur MacOS.

Conclusion : si vous synchronisez les accès à la mémoire partagée par plusieurs processus vous devez develloper vos programmes sur Linux.

variable mutex

La variable mutex est toujours dans un de deux états : verrouillée (locked) ou déverrouillée (unlocked).

Juste après l'initialisation la variable mutex est déverrouillée (unlocked).

opérations sur mutex

int
pthread_mutex_lock(pthread_mutex_t *mutex)

- si le mutex est dans l'état verrouillé alors le processus appelant `pthread_mutex_lock` bloque.

- si le 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( &mutex );

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

section critique :

opérations sur le contenu de la mémoire partagée


r = pthread_mutex_unlock( &mutex );
if( r!= 0){ //traiter erreur }

&mutex – l'adresse d'un mutex
```

règles d'utilisation de mutex

- C'est **le même processus** qui a qui a posé le verrou sur un mutex avec `pthread_mutex_lock()` doit lever le verrou avec `pthread_mutex_unlock()`.

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

opérations lock nonbloquante

`int pthread_mutex_trylock(pthread_mutex_t *mutex);`

Si le mutex n'est pas verrouillée `pthread_mutex_trylock()` fait la même chose que `pthread_mutex_lock()` c'est-à-dire verrouille le mutex.

Quand le mutex est déjà verrouillé par un autre processus `pthread_mutex_trylock()` retourne tout de suite la valeur EBUSY. La tentative de poser le verrou échoue.

opérations lock - variantes

```
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
        const struct timespec *restrict abstime);
```

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

exemple

Le processus écrivain écrit un message dans la mémoire partagée, le processus lecteur lit et affiche le message.

Protection d'accès à la mémoire partagée assurée par un mutex.

Création d'un shared memory object projeté en mémoire assurée par l'écrivain.

exemple : structure de données dans la mémoire partagée

```
/*   data.h    */
#ifndef DATA_H
#define DATA_H
#include <pthread.h>
#include <stdbool.h>
#define MESSAGE_LEN 2048

typedef struct{
  pthread_mutex_t mutex;
  bool libre; /* libre == true si on peut écrire
               * un nouveau message, false sinon */
  size_t compteur; /* compteur de messages */
  pid_t  pid;  /* pid du dernier processus qui
                * a modifié la mémoire */
  size_t len;  /* longueur de message */
  char msg[MESSAGE_LEN]; /* message */
} memory;
#endif
```

traitement d'erreur pour les fonctions de la famille pthread_...()

```
void thread_error_exit(const char *file, /* nom fichier *
                       int line,         /* ligne de code */
                       int code,         /* code d'erreur */
                       char *txt){       /* message suppl.*/
  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);
}
```

ECRIVAIN

écrivain - construire un shared memory object

```
 int fd = shm_open(mem_name, O_CREAT| O_RDWR,
           S_IWUSR | S_IRUSR);

  if( fd < 0 ) PANIC_EXIT( "shm_open" );

#ifdef __linux__
  if( ftruncate( fd, sizeof( memory ) ) < 0 )
    PANIC_EXIT("ftruncate");
#endif

#ifdef   APPLE
  fprintf(stderr, "mutex inutilisable sur MacOS\n");
  exit(EXIT_FAILURE°;
#endif
```

écrivain - initialiser la mémoire partagée

```
memory *mem = mmap(NULL, sizeof(memory),
                   PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0);
if( (void *)mem == MAP_FAILED )
  PANIC_EXIT("mmap");

/*  initialisation de mémoire partagée */
int code =  initialiser_mutex( &mem->mutex );
if( code > 0 )
  thread_error_exit(__FILE__, __LINE__, code,
                    "initialiser_mutex");

mem->libre = true;
mem->compteur = 0;

printf("vous pouvez lancer le lecteur\n");
/* fin d'initialisation de la mémoire partagée */
```

écrivain - boucle d'écriture

```
while( true ){

 /*  mutex_lock */
 code = pthread_mutex_lock( &mem->mutex) ;
 if( code > 0 )
   thread_error_exit(__FILE__, __LINE__, code,
        "mutex_lock");

 /* section critique */
 if( mem->libre ){  /* vérifier la condition d'écriture */
   mem->libre = false;
   mem->compteur++;
   mem->pid = getpid();

   /* fabriquer un message */
   char *message = "Hello!";
   int n = snprintf(mem->msg, sizeof(mem->msg),
            "%d %s\n", (int)mem->compteur,
            message );
   mem->len = n+1;
 }/* fin section critique */

 /* mutex_unlock */
 code = pthread_mutex_unlock( &mem->mutex );
 if( code > 0 )
   thread_error_exit(__FILE__, __LINE__, code,
        "mutex_unlock");
}
```

LECTEUR

lecteur - ouvrir le shared memory object et projeter en

```
int fd = shm_open(mem_name,  O_RDWR, S_IWUSR | S_IRUSR);
if( fd == -1)
  PANIC_EXIT("shm_open");

memory *mem = mmap(NULL, sizeof(memory),
          PROT_READ|PROT_WRITE, MAP_SHARED,
          fd, 0);
if( mem == MAP_FAILED)
  PANIC_EXIT("mmap");
```

```
/* trouver le nombre de lectures dans la variable
 * enviromement TOTAL*/
char *tot = getenv("TOTAL");

int total = 1000;  /* total = le nombre de lectures */
if( tot != NULL ){
  total = atoi( tot );
}


int tentatives = 0; /* compteur de tentatives de lecture */
```

lecteur - boucle de lecture

```
for( int i = 0; i < total ;  ){
   int code ;

   code = pthread_mutex_lock( &mem->mutex)  ;
   if( code > 0 )
     thread_error_exit(__FILE__, __LINE__, code,
          "mutex_lock");

   if( !mem->libre ){ /*lire uniquement si !mem->libre */
     i++;
     mem->pid = getpid();
     printf("message numero %d : %s\n", (int)mem->compteur,
              mem->msg);
     mem->libre = true;
   }
   tentatives ++;

    /* mutex unlock */
   code = pthread_mutex_unlock( &mem->mutex ) ;
   if( code > 0 )
     thread_error_exit(__FILE__, __LINE__, code,
          "mutex_unlock");

}
printf("#tentatives = %d, #réussites = %d\n", tentatives, total);
```

Les variables mutex permettent d'implémenter l'exclusion mutuelle de l'accès à la mémoire partagée.

Mais elles ne fournissent pas de mécanisme de synchronisation, le lecteur et l'écrivain sont dans l'attente active pour pouvoir écrire/lire un message.

synchroniser les processus grâce aux variables condition

pthread_cond_wait

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

cond – pointeur sur une variable de type condition (`pthread_cond_t`)

1. `avant` d'exécuter `pthread_cond_wait()` le processus doit déjà détenir le verrou sur le mutex grâce au `pthread_mutex_lock()`,
2. l'appel à `pthread_cond_wait()` met le processus appelant en attente.
   Le processus appelant reste suspendu jusqu'à la réception d'un signal (il ne s'agit pas de signal POSIX même si la terminologie est similaire).
   (Justification : le processus est suspendu et tant qu'il est suspendu il ne n'utilise pas des données partagées, donc le mutex peut être déverrouillé).
3. A la réception de signal, ou plus généralement, au réveil, le processus suspendu sur `pthread_cond_wait()` obtient automatiquement le verrous sur le mutex.

pthread_cond_wait

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

La fonction pthread_cond_wait() 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.

pthread_cond_signal

`int pthread_cond_signal( pthread_cond_t *cond )`

provoque l'envoie de signal vers les processus suspendus sur la variable **`cond`** et réveille un des ces processus.

Le processus réveillé réacquiert le mutex associé.

La norme POSIX autorise l'implémentaion le comportement où la réception de signal réveille plusieurs processus en attente (mais un seul obtient le mutex).

le schéma d'utilisation de mutex et condition

A chaque variable de type `pthread_cond_t` on associe un prédicat `cndt` (une condition) sur les variables dans la mémoire partagée.

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

- **`cndt`** utilise les variables partagées donc pour vérifier si **`cndt`** est vrai le processus doit déjà posséder le verrou sur le mutex associé
- le processus suspendu sur un appel à **`pthread_cond_wait()`** peut-être réveillé mêmes quand la condition **cndt** est toujours fausse, donc **il faut toujours revérifier si cndt et vrai après le retour de pthread_cond_wait() .**
- le processus qui sort de la session critique doit signaler à chaque processus en attente à l'entrée de la section critique le changement de condition en utilisant pthread_cond_signal()

schéma simplifié d'utilisation de mutex et condition (sans traitement d'erreurs)

```
pthread_mutex_lock( &mutex ) ;

/* attendre la condition, tjrs dans la boucle */

while( ! condition_entrée_de_la_section_critique  ){
   pthread_cond_wait( &rcond, &mutex) ) ;
}

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

pthread_mutex_unlock(&mutex);

/* signaler la sortie de la section critique
 * à tous les processus  suspendu sur wcond */

pthread_cond_signal( &wcond );
```

on peut échanger l'ordre
de pthread_mutex_unlock()
pthread_cond_signal()

Intuitivement la variable rcond est associée à la condition_entrée_de_la_section_critique

broadcast ou signal

```
int pthread_cond_broadcast( pthread_cond_t *c )
```

réveille tous les processus qui attendent sur la variable condition **c**

Exemple : producteur - consommateur

à tour de rôle :
1) le producteur écrit dans la mémoire partagée
2) le consommateur lit l'information écrite par le producteur

Le producteur doit être suspendu tant que le message précédent n'a pas été consommé.

Le consommateur doit être suspendu tant qu'il n'y a pas de nouveau message à consommer.

la structure dans la mémoire partagée

```
/*  memory.h     */
#ifndef MEMORY_H
#define MEMORY_H
#include <pthread.h>
#include <stdbool.h>
typedef struct{
  pthread_mutex_t mutex;
  pthread_cond_t  rcond; /*variable condition pour la lecture */
  pthread_cond_t  wcond; /* variable condition pour l'écriture */
  pid_t  pid;            /* pid du dernier processus qui accède
                          * à la memoire partagée  */
  bool libre;            /* true si le message a déjà été lu*/
  int data;              /* le message */
} memory;
#endif
```

producteur : initialiser la mémoire partagée

```
#ifdef __linux__
#define _XOPEN_SOURCE 500
#endif
/* les includes etc. */

void *init_memory(const char *mem_objet_name){


  /* supprimer l'objet mémoire avant de le récréer */
  shm_unlink(mem_object_name);

  int fd = shm_open(mem_object_name, O_CREAT| O_RDWR , S_IWUSR | S_IRUSR);
  if( fd < 0 ) PANIC_EXIT("shm_open");

  if( ftruncate( fd, sizeof( memory ) ) < 0 )  PANIC_EXIT("ftruncate");
```

```
  /* projection de shared memory object dans  la mémoire    */
  memory *mem = mmap(NULL, sizeof(memory), PROT_READ|PROT_WRITE,
              MAP_SHARED, fd, 0)
  if( (void *)mem == MAP_FAILED ) PANIC_EXIT("mmap");

  //initialiser la mémoire
  int code;
  mem->libre = true;  /* memoire est libre initialement*/
  mem->data = -1;

  code = initialiser_mutex( &mem->mutex );
  if( code > 0 ) thread_error(__FILE__, __LINE__, code, "init_mutex");

  code = initialiser_cond( &mem->rcond );
  if( code > 0 ) thread_error(__FILE__, __LINE__, code, "init_rcond");

  code = initialiser_cond( &mem->wcond );
  if( code >  0 ) thread_error(__FILE__, __LINE__, code, "init_wcond");

  return mem;
}
```

producteur : boucle principale (simplifié sans traitement d'erreurs)

```
memory *mem = init_memory( argv[1] );
while( 1 ){

    int code = pthread_mutex_lock(&mem->mutex);

    /* attendre jusqu'à ce que la mémoire soit libre */
    while( ! mem->libre ){
      code = pthread_cond_wait( &mem->wcond,
                                &mem->mutex);
    }

    /* section critique */
    mem->pid = getpid();  mem->data = value++;
    mem->libre = false;

    code = pthread_mutex_unlock(&mem->mutex);


    /* signaler le lecteur */
    code = pthread_cond_signal( &mem->rcond );
}
```

producteur : boucle principale (complet)

```
memory *mem = init_memory( argv[1] );
while( 1 ){

    int code = pthread_mutex_lock(&mem->mutex);
    if( code > 0 )
      thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );

    /* attendre jusqu'à ce que la mémoire soit libre */
    while( ! mem->libre ){
      code = pthread_cond_wait( &mem->wcond, &mem->mutex);
      if( code > 0 )
         thread_error( __FILE__, __LINE__, code,
                       "pthread_cond_wait" );
    }

    /* section critique */
    mem->pid = getpid();  mem->data = value++; mem->libre = false;

    code = pthread_mutex_unlock(&mem->mutex);
    if( code != 0)
      thread_error( __FILE__ , __LINE__ , code, "mutex_unlock" );

    /* signaler le lecteur */
    code = pthread_cond_signal( &mem->rcond );
    if( code > 0 )
      thread_error( __FILE__ , __LINE__ , code, "cond_signal" );
}
```

consommateur : avant la boucle principale

```
int fd = shm_open(shm_name, O_RDWR, S_IRUSR|S_IWUSR );

if( fd < 0 )
  PANIC_EXIT("shm_open");

//projection en mémoire
memory  *mem =
     mmap(NULL, sizeof(memory), PROT_READ|PROT_WRITE,
     MAP_SHARED, fd, 0);
if( (void *)mem == MAP_FAILED)
  PANIC_EXIT("mmap");

int i = 0; /* le nombre de tentatives */
int j = 0; /* le nombre de lectures effectives */
```

consommateur : la boucle principale (version simplifiée sans traitement d'erreur)

```
while( 1 ){

  /*  mutex_lock */
  int code = pthread_mutex_lock(&mem->mutex);

  /* attendre tant que la mémoire ne contient pas de message */
  while( mem->libre ){
    i++;
    code = pthread_cond_wait( &mem->rcond, &mem->mutex);
  }

  j++;   /* incrémenter le compteur de visites dans la section critique */
  mem->libre = 1;/* les données consommées*/

  code = pthread_mutex_unlock(&mem->mutex);

  /* signaler le producer */
  code = pthread_cond_signal( &mem->wcond );

  if( j >= total )  break;
}
```

consommateur : la boucle principale (version complète)

```
while( 1 ){

  /*  mutex_lock */
  int code = pthread_mutex_lock(&mem->mutex);
  if( code > 0 )
    thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );



  /* attendre tant que la mémoire ne contient pas de message */
  while( mem->libre ){
    i++;
    code = pthread_cond_wait( &mem->rcond, &mem->mutex);
    if( code > 0 )
         thread_error( __FILE__, __LINE__, code, "pthread_cond_wait" );
  }

  j++;  /* incrémenter le compteur de visites dans la section critique */
  mem->libre = 1;/* les données consommées*/


  code = pthread_mutex_unlock(&mem->mutex);
  if( code > 0)
    thread_error( __FILE__ , __LINE__ , code, "mutex_unlock" );

  /* signaler le producer */
  code = pthread_cond_signal( &mem->wcond );
  if( code > 0 )
    thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );

  if( j >= total )
    break;
}

printf("[pid %d] total =%d  #of cond_wait = %d\n", (int) getpid(), j, i);
```