Programmation systèmes avancée
Mutex et conditions

1 Introduction

Les objets `pthread_mutex_t` (les mutex) sont utilisés pour protéger les accès parallèles en mémoire entre les threads mais, après une initialisation appropriée, ils peuvent servir comme moyen de synchronisation entre les processus. Le mutex peut être dans un des deux états suivants : soit *verrouillé (locked)*, soit *déverrouillé (unlocked)*. Immédiatement après l'initialisation, le mutex est dans l'état déverrouillé.
Mais les mutex tous seuls ne suffisent pas (sauf dans les cas triviaux) et pour coordonner les processus et éviter une attente active à l'entrée d'une section critique il faut aussi un autre type d'objet : `pthread_cond_t`.
Comme pour les sémaphores anonymes, les mutex et les objets `pthread_cond_t` doivent être accessibles par tous les processus qui les utilisent. Mais Contrairement aux sémaphores il n'y a pas de mutex nommés, tous les mutex sont « anonymes ». Cela implique que les mutex et les objets `pthread_cond_t` partagés par plusieurs processus doivent être placés

- soit dans la mémoire partagée anonyme obtenue par `mmap` avec `MAP_ANONYMPOUS` et `MAP_SHARED`
- soit dans la mémoire obtenue par `shm_open` suivi de `mmap` de type `MAP_SHARED`.

2 Gestion d'erreur

D'habitude les fonctions C renvoient en cas d'erreur une valeur spéciale (souvent -1 ou `NULL`) et mettent dans `errno` le numéro d'erreur. Ce comportement n'est pas valable pour les fonctions qui opèrent sur les mutex/condition. La raison est que les mutex/condition servent surtout pour synchroniser les threads d'un même processus mais tous ces threads partagent la variable `errno`. Donc la valeur d'erreur d'un thread effacerait la valeur d'erreur d'un autre thread, sans moyen de savoir quel thread a provoqué une erreur.
Pour les fonctions qui utilisent les mutex/condition, c'est la valeur de retour de la fonction qui indique à la fois s'il y a une erreur et, le cas échéant, le code d'erreur :

- la valeur de retour 0 indique l'appel réussi,
- par contre si l'appel se termine avec erreur la fonction retourne le code d'erreur qui est toujours > 0.

Notez que ceci implique que la fonction `perror()` est inutilisable avec les mutex/condition.

3 Utilisation des mutex

Une variable mutex fraîchement initialisée est dans l'état déverrouillé (unlocked).
Pour changer l'état de la variable, on utilise deux fonctions :

```
1 #include <pthread.h>
2 int pthread_mutex_lock( pthread_mutex_t *pmutex )
```

```
int phtread_mutex_unlock( pthread_mutex_t *pmutex)
```

qui permettent respectivement de verrouiller et déverrouiller le mutex.
Les deux fonctions prennent comme paramètre un pointeur vers un mutex.
Nous allons dire qu'un processus *possède un mutex* s'il a réussi l'opération `pthread_mutex_lock` et s'il n'a pas encore libéré le mutex avec `pthread_mutex_unlock`. Si un processus possède un mutex, alors tout autre processus qui fait appel à `pthread_mutex_lock` sur le même mutex sera bloqué jusqu'à ce que le mutex soit libéré (déverrouillé).
Quand le processus qui libère le mutex exécute `pthread_mutex_unlock`, un des processus bloqués sur le mutex est réveillé et verrouille le mutex à son tour.
Il y a plusieurs règles qui gèrent l'utilisation des fonctions lock et unlock :

- seul le processus qui a fait `pthread_mutex_lock` peut ensuite exécuter `pthread_mutex_unlock` pour déverrouiller le mutex [1],
- le processus qui possède le mutex ne doit pas essayer d'exécuter à nouveau `pthread_mutex_lock` sur ce mutex,
- si un mutex est verrouillé par un processus A et qu'un autre processus B exécute `pthread_mutex_lock` sur le même mutex, alors le processus B sera suspendu jusqu'à ce que le processus A lève le verrou,
- si plusieurs processus sont suspendus sur l'opération `pthread_mutex_lock`, et que le processus qui détient le mutex exécute `pthread_mutex_unlock`, un des processus suspendu sur le mutex pourra acquérir le verrou sur le mutex. Il est impossible de spécifier quel processus en attente réussira à acquérir le mutex libéré.

Pour protéger une section critique, un processus doit exécuter `pthread_mutex_lock` à l'entrée et
`pthread_mutex_unlock` à la sortie de la section :

```
pthread_mutex_t *pmutex;

/* initialiser pmutex avec une adresse dans la mémoire partagée */
.......
int code = pthread_mutex_lock( pmutex );

/* pour récuperer le massage correspondant à l'erreur *
 * il faut passer par strerror                        */
if( code != 0 ){
   char *msg = strerror( code );
   fprintf(stderr, "%s\n", msg);
   exit(1);
}

//  faire des opérations sur la mémoire partagée

/* libérer le mutex                                   */
code = pthread_mutex_unlock( pmutex );
if( code != 0 ){
   char *msg = strerror( code );
   fprintf(stderr, "%s\n", msg);
```

1. Notez la différence avec les sémaphores, il est tout à fait légitime et presque toujours nécessaire qu'un processus applique `sem_wait` et un autre `sem_post`.

```
22    exit(1);
23 }
```

Comme montre l'exemple ci-dessus le mutex permet de protéger les accès à la mémoire partagée. Mais dans la plupart de cas c'est loin d'être suffisant. Supposons que la mémoire partagée sert à transférer des données entre le processus producteur et le processus consommateur.
Le producteur, avant de mettre une nouvelle donnée dans la mémoire partagée, doit vérifier si la donnée précédente à été consommée (lue) par le consommateur. A l'inverse, le consommateur doit vérifier si la mémoire contient une nouvelle donnée.
Donc il nous faut un drapeaux `libre` qui, par exemple, est à 0 si le producteur peut écrire une nouvelle donnée et il est à 1 quand le consommateur peut consommer la donnée.
Mais le drapeau lui même doit être accessible par les deux processus, donc doit résider dans la mémoire partagée avec les accès protégés par un mutex.
Si nous avons juste les mutex la seule solution consiste à mettre en place une attente active, par exemple le consommateur devrait exécuter l'algorithme suivant pour accéder aux données :

(1) le consommateur verrouille le mutex protégeant le drapeau `libre`,

(2) il examine le drapeaux `libre` pour voir s'il y a des nouvelles données, et s'il y en a pas alors il déverrouille le mutex et revient à (1).

Au lieu de tourner dans une boucle il est préférable que le consommateur qui trouve le drapeau `libre` indiquant qu'il n'y a pas de données à lire soit suspendu et soit réveillé quand la valeur de `libre` change. Ce mécanisme est implémenté grâce aux objets `pthread_cond_t`.

4 Les objets `pthread_cond_t`

Les objets `pthread_cond_t` permettent une attente passive qui suspend l'exécution d'un processus qui attend qu'une condition soit satisfaite [2].
Les objets `pthread_cond_t` doivent résider dans la mémoire partagée [3] et eux-mêmes doivent être protégés par des mutex.
Trois fonctions permettent de manipuler les objets `pthread_cond_t` :

```
1 #include <pthread.h>
2 int pthread_cond_wait(pthread_cond_t *restrict pcond,
3                       pthread_mutex_t *restrict pmutex)
4 int pthread_cond_signal(pthread_cond_t *pcond)
5 int pthread_cond_broadcast(pthread_cond_t *pcond)
```

`pthread_cond_wait` prend comme arguments un pointeur vers un mutex et un pointeur vers un objet `pthread_cond_t`.
Le processus qui exécute `pthread_cond_wait` doit auparavant acquérir le mutex correspondant. L'appel à `pthread_cond_wait()` s'effectue de manière atomique et a un double effet :

2. une condition dans le sens de conditions qu'on trouve dans `if` ou `while`
3. c'est nécessaire si `pthread_cond_t` sont utilisés par des processus différents, pour synchroniser les threads les `pthread_cond_t` sont souvent déclarées comme des variables globales de programme

(a) le processus appelant suspend son exécution ; il est commode de le considérer comme suspendu sur le couple (`pthread_cond_t`, mutex),

(b) en même temps, il libère le verrou sur le mutex.

`pthread_cond_signal` réveille un processus suspendu sur le couple (`pthread_cond_t`,mutex) tandis que `pthread_cond_broadcast` réveille tous les processus suspendus sur (`pthread_cond_t` , mutex).
Dans les deux cas un des processus réveillés obtient le mutex.
Comment utilise-t-on les mutex avec les objets `pthread_cond_t` ?
Intuitivement, à chaque objet `pthread_cond_t` on associe une condition [4] sur les variables partagées [5]. Nous allons appeler « section critique » la partie de code qui manipule les variables partagées. Par exemple, si on revient à l'exemple discuté à la fin de la section 3, la condition peut être `libre==0`.
Soit `condition` une condition sur les variables partagées, `pcond` un pointeur vers l'objet `pthread_cond_t` associée, et `pmutex` un pointeur vers un `pthread_mutex_t`.
En général l'algorithme pour accéder à une section critique est composé de plusieurs étapes :

(1) le processus demande le verrou sur `pmutex` avec `pthread_mutex_lock`,

(2) le processus vérifie si la `condition` d'entrée dans la section critique est satisfaite, et si ce n'est pas le cas alors il doit attendre : `pthread_cond_wait(pcond, pmutex)`,

(3) dans la section critique le processus modifie les variables partagées,

(4) à la sortie de la section critique, le processus réveille, si nécessaire, le(s) processu(s) en attente pour entrer dans la section critique avec `pthread_cond_signal` ou `pthread_cond_broadcast`.

(5) le processus lève le verrou : `pthread_mutex_unlock`.

Le code très simplifié de cet algorithme [6] :

```
1  //pmutex : pointeur vers un mutex
2  //pcond  : pointeur vers pthread_cont_t
3
4  pthread_mutex_lock( pmutex ); /* obtenir le mutex qui protege
5                                 * les  donnes*/
6
7  while( ! condition ) //attendre tant que la condition non statisfaite
8      pthread_cond_wait( pcond, pmutex );
9
10 <section critique>
11
12 /* signaler éventuellement d'autres processus */
13 if(cond_autre)
14     pthread_signal( pcond_autre );
15
16 pthread_mutex_unlock( pmutex );
```

4. condition similaire à celle qu'on trouve dans `if` ou `while`
5. ce sont les variables qui résident dans la mémoire partagée
6. très simplifié parce que dans le code réel il faut assurer le traitement d'erreur, mais ce qui nous intéresse pour l'instant est la logique du programme

4.1 Attendre sur une variable de condition : `while` ou `if` ?

Notez que l'attente aux lignes 7 et 8 du code ci-dessus est faite par une boucle :

```
1 while( ! condition )
2     pthread_cond_wait( pcond, pmutex );
```

Pourquoi pas un `if` simple :

```
1 if( ! condition )
2     pthread_cond_wait( pcond, pmutex );
```

qui suspend le processus quand la condition n'est pas satisfaite ? La différence entre `while` et `if` se situe au réveil du processus [7] : avec `while`, il re-vérifie la condition, alors qu'avec un simple `if`, la condition n'est pas re-vérifiée après le réveil.
Mais pourquoi re-vérifier la condition ? Après tout, si un autre processus nous réveille, c'est parce qu'il a découvert que la condition est satisfaite, donc le re-vérifier semble inutile.
Il y a plusieurs raisons. D'abord, parfois l'autre processus vérifie une condition plus large donc il peut nous réveiller inutilement.
Ensuite, entre le moment où le signal de réveil est envoyé et le moment du réveil la condition peut changer sa valeur.
Et finalement, et le plus surprenant, il est tout à fait conforme à la norme qu'un processus suspendu sur un `pthread_cond_wait` se réveille spontanément sans être signalé. Dans Single Unix Specification on appelle cela « spurious wakeup ».

4.2 Attendre sur une condition

Quand on attend sur un objet `pthread_cond_t`, il faut spécifier aussi un mutex. Le choix de mutex n'est pas complètement libre. Voilà ce qu'écrit Butenhof dans *Programming with POSIX threads* [8].
« Each condition variable must be associated with a specific mutex, and with a predicate condition. When a thread waits on a condition variable it must always have the associated mutex locked. Remember that the condition variable wait operation will *unlock* the mutex for you before blocking the thread, and it will *relock* the mutex before returning to your code ».
« All threads that wait on any one condition variable concurrently (at the same time) must specify the *same* associated mutex. Pthreads does not allow thread 1, for exemple, to wait on condition variable A specifying mutex A while thread 2 waits on condition variable A specifying mutex B. It is, however, perfectly reasonable for thread 1 to wait on condition variable A specifying mutex A while thread 2 waits on condition variable B specifying mutex A. That is, each condition variable ... must be associated with only one mutex — but a mutex may have any number of condition variables associated with it ».

7. il est réveillé parce qu'un autre processus a exécuté `pthread_cond_signal( pcond )`
8. il parle de threads mais la même remarque s'applique aux processus, il suffit de remplacer "thread" par "process". Et "condition variables" ce sont les objets `pthread_cond_t`.

5 Exemple

Dans l'exemple suivant, les processus producteurs écrivent les données `data` et les processus consommateurs lisent les données. Chaque écriture par un producteur nous autorise une seule lecture par un seul consommateur, mais nous pouvons avoir plusieurs processus producteurs et plusieurs consommateurs. Donc le comportement désiré est une suite d'opérations : écriture, lecture, écriture, lecture , écriture, lecture, ...
On regroupe tout ce qui se retrouve dans la mémoire partagée : les données, les variables mutex, les objets `pthread_cond_t` dans une structure `memory` :

```
typedef struct{
  pthread_mutex_t mutex;
  pthread_cond_t rcond;
  pthread_cond_t wcond;
  pid_t pid;            /* le pid de processus producteur */
  unsigned char libre; /* 1 si pas de données ,
                         * 0 si données prêtes pour la lecture */
  int data;  /* les données*/
} memory;
```

Dans notre exemple on utilise deux objets `pthread_cond_t` et un seul mutex. Il y a deux champs pour les données, `data` pour la valeur `int` écrite par le producteur, `pid` pour enregistrer le pid du producteur. Et il y a un champ de contrôle, `libre` qui vaut 1 si le champ `data` contient une nouvelle valeur.
On suppose qu'avant de lancer les processus producteur/consommateur, on exécutera un programme `init_memory` qui va créer le shared memory object et initialiser tous les champs de `memory`. L'initialisation de mutex et des objets `pthread_cond_t` est discutée dans la section 6.
Donc on assume que quand on lance les processus producteurs et consommateurs le shared memory object existe déjà, il a la taille `sizeof(memory)` et tous les champs de `memory` sont correctement initialisés.
Chaque producteur exécute le code suivant [9] :

```
int fd = shm_open(mem_name,  O_RDWR, 0 );
if( fd < 0 ) PANIC_EXIT("shm_open");

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

while( 1 ){
   int code;
   if( ( code = pthread_mutex_lock(&mem->mutex) ) != 0 )
       thread_error( __FILE__ , __LINE__ , code, "mutex_lock" );

   /* Attendre  que la mémoire soit disponible pour écrire les données.
```

9. En cas d'erreur d'une fonction `pthread...` on appelle la fonction `thread_error` qui affiche le message d'erreur et termine le processus avec `exit`. Regardez l'exemple complet pour le code de cette fonction.

```
     * Toujours la boucle while.                                    */
    while(! mem->libre ){
       if( (code = pthread_cond_wait(&mem->wcond, &mem->mutex )) != 0)
            thread_error( __FILE__ , __LINE__ , code, "cond_wait");
    }

    mem->data = ... ; /* mettre les nouvelles données */
    mem->libre = 0;   /* indiquer qu'il y a des données à lire */

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

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

Notez que l'appel à `pthread_cond_signal` figure après `pthred_mutex_unlock`. Il est possible d'inverser l'ordre de ces deux opérations mais cela peut donner un code moins efficace. Le code du consommateur est symétrique :

```
int fd = shm_open(mem_name,  O_RDWR );
if( fd < 0 ) PANIC_EXIT("shm_open");

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

while( 1 ){
   int code;

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

   /* Attente que les données soient sur place.
    * Toujours dans la boucle while.
    * Tant que mem->libre == 1 les données ne sont pas disponible */
   while( mem->libre ){
      if( (code = pthread_cond_wait( &mem->rcond, &mem->mutex )) != 0)
           thread_error( __FILE__ , __LINE__ , code, "cond" );
   }
   consommer( mem->data ); //consommer

   //indiquer que les données sont consommées
   mem->libre = 1;

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

```

```
30    /* signaler un producteur */
31    if ( ( code = pthread_cond_signal( &mem->wcond) ) != 0)
32        thread_error( __FILE__ , __LINE__ , code, "cond_signal" );
33 }
```

Pour compléter l'exemple, il nous faut initialiser la structure `memory` placée dans un *shared memory object*. Pour cela j'utilise un programme séparé qui doit être lancé **avant** d'avoir lancé les processus `producteur` et `consommateur`. Ce programme

1. crée un *shared memory object*,
2. fixe la taille de l'objet par un appel à `ftruncate`,
3. fait sa projection en mémoire avec `mmap` en mode `MAP_SHARED`,
4. initialise le champ `libre` de la structure `memory`,
5. initialise les champs `pthread_mutex_t` et `pthread_cond_t` de la structure `memory`.

Pour les détails voir le code dans le fichier `init_memory.c` sur moodle.
Pour tester notre exemple (après avoir compilé avec `make`)

- on lance `./init_memory sh_name` pour créer et initialiser l'objet *shared memory* dont le nom est `sh_name`.
- sur un terminal on lance plusieurs producteurs :
  `./producer sh_name & ./producer sh_name`
- et sur un autre terminal on lance plusieurs consommateurs :
  `./consumer sh_name & ./consumer sh_name`

6 Initialiser des `pthread_mutex_t` et `pthread_cond_t`

Dans cette section, nous décrivons uniquement l'initialisation de variables mutex/condition partagées par les processus. C'est une étape un peu fastidieuse, à chaque fois il faut préparer les attributs et indiquer que la variable sera utilisée pour synchroniser des processus et non pas des threads.
Il est commode d'utiliser les fonctions suivantes que vous pouvez simplement recopier dans votre programme. Les deux fonctions retournent 0 si l'initialisation est réussie ou le code d'erreur sinon. Vous ne pouvez pas utiliser de variables mutex/condition non-initialisées. En particulier les deux fonctions sont utilisées dans notre exemple par le programme dans le fichier `init_memory.c` qui initialise les champs dans *shared memory object*.

```
1  int initialiser_mutex(pthread_mutex_t *pmutex){
2      pthread_mutexattr_t mutexattr;
3    int code;
4    if( ( code = pthread_mutexattr_init(&mutexattr) ) != 0)
5      return code;
6
7    if( ( code = pthread_mutexattr_setpshared(&mutexattr,
8                PTHREAD_PROCESS_SHARED) ) != 0)
9      return code;
10   code = pthread_mutex_init(pmutex, &mutexattr)  ;
11   return code;
12 }
13
```

```c
int initialiser_cond(pthread_cond_t *pcond){
  pthread_condattr_t condattr;
  int code;
  pthread_condattr_init(&condattr);
  pthread_condattr_setpshared(&condattr,PTHREAD_PROCESS_SHARED);
  if( ( code = pthread_cond_init(pcond, &condattr) ) != 0)
    return code;
  code = pthread_cond_init(pcond, &condattr)  ;
  return code;
}
```

7 Compilation

Sous Linux un programme utilisant les mutex et les variables de condition doit être linké avec la librairie pthread, et comme les mutex et les conditions doivent être dans la mémoire partagée il faut aussi la librairie librt.
Dans le Makefile bien fait il suffit d'ajouter `LDLIBS= -lrt -pthread`.
Dans la ligne de commande (fortement déconseillé, j'espère que personne ne compile sans Makefile), les librairies doivent apparaître **après** le(s) fichier(s) sources :

```
gcc -Wall source.c -lrt -pthread -o source
```

Sous MacOS les variables mutex ne peuvent pas être utilisées pour synchroniser les processus et la fonction `pthread_mutexattr_setpshared` nécessaire pour initialiser les mutex utilisés pour synchroniser les processus n'est même pas documentée.