Communication Inter-processus

Sémaphores (suite)

Section critique

- Portion de code qui ne doit pas être exécutée par plus d'un thread à la fois ;
- Exemples : accès aux ressources partagées entre les threads :
  - Thread 1 lit la variable v = 1 ;
  - Thread 2 lit la variable v ;
  - Thread 1 ajoute 1 à v et stocke le résultat dans v ;
  - Thread 2 ajoute 1 à v et stocke le résultat dans v ;
  - v ne vaut alors pas 3 mais 2.
- Nécessité d'un mécanisme pour s'assurer qu'un seul thread effectue un ensemble d'action sur une ressource.

Opération atomique

- Opération dont l'exécution est considérée comme « instanée »;
- S'appuie sur des instructions du processeur :
  - Test-and-set : défini une variable à une valeur et retourne la valeur précédente ;
  - Fetch-and-add : lit une valeur, lui ajoute un, stocke le résultat;
  - Compare-and-swap : compare deux valeurs, si elles sont égales alors la première est remplacée par une nouvelle valeur.
- Fondamentalement limité par les instruction implémentées dans le processeur.

Sémaphores

- Mutex étendu pour permettre plusieurs accès simultanés : le nombre d'accès autorisé est stocké dans un compteur ;
- Deux implémentations :
  - SystemV : sem* ;
  - POSIX : sem_*.
- Si possible, privilégier l'implémentation POSIX.

Sémaphores anonymes

- Initialisation avec

```
int sem_init(sem_t *sem, int pshared,
unsigned int value):
```

peut être local à une application ou partagé entre plusieurs processus (mémoire partagée) ;

- `int sem_wait(sem_t *sem)` : prendre une « place » (diminue le compteur de 1) ou attendre qu'une place se libère (si le compteur est à 0) ;
- `int sem_post(sem_t *sem)` : libérer le sémaphore (augmente le compteur de 1) ;
- `int sem_getvalue(sem_t *sem, int *sval)` : obtenir la valeur actuelle du compteur.

Sémaphores nommés

- Ouverture avec

```
sem_t *sem_open(const char *name, int oflag...):
```

crée un nouveau sémaphore nommé ou en ouvre un existant ;

- Le nom doit être de la forme /nomdusemaphore ;
- Ne pas confondre l'attribut partagé d'un sémaphore anonyme et les sémaphores nommés
- Le fonctionnement est similaire à un sémaphore anonyme ;
- Liste des semaphores POSIX : `ls /dev/shm/sem.* ;`
- Liste des semaphores System V : `ipcs -s`.

Exclusion mutuelle

Les sémaphores permettent de résoudre de nombreux problèmes classiques. Le premier est celui de l'exclusion mutuelle. Lorsqu'il est initialisé à 1, un sémaphore peut être utilisé de la même façon qu'un mutex.

```
#include <semaphore.h>

//...

sem_t semaphore;

sem_init(&semaphore, 0, 1);

sem_wait(&semaphore);
// section critique
sem_post(&semaphore);

sem_destroy(&semaphore);
```

Exclusion mutuelle

Les sémaphores peuvent être utilisés pour d'autres types de synchronisation. Par exemple, considérons une application découpée en threads dans laquelle la fonction `after` ne peut jamais être exécutée avant la fin de l'exécution de la fonction `before`

```
#define NTHREADS 2
sem_t semaphore;

void *before(void * param) {
  // do something
  for(int j=0;j<1000000;j++) {
  }
  sem_post(&semaphore);
  return(NULL);
}
```

```
void *after(void * param) {
  sem_wait(&semaphore);
  // do something
  for(int j=0;j<1000000;j++) {
  }
  return(NULL);
}
```

```c
int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  void * (* func[])(void *)={before, after};
  int err;

  err=sem_init(&semaphore, 0,0);
  if(err!=0) {
      error(err,"sem_init");
  }
  for(int i=0;i<NTHREADS;i++) {
    err=pthread_create(&(thread[i]),NULL,func[i],NULL);
    if(err!=0) {
      error(err,"pthread_create");
    }
  }

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(thread[i],NULL);
    if(err!=0) {
      error(err,"pthread_join");
    }
  }
  sem_destroy(&semaphore);
  if(err!=0) {
      error(err,"sem_destroy");
  }
  return(EXIT_SUCCESS);
}
```

Mutex

- Primitive de synchronisation pouvant prendre deux états: disponible ou verrouillé
- Permets de définir des sections critiques arbitrairement
- Initialisation avec

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

(attributs par défaut)

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const   pthread_mutexattr_t *attr)
```

(attributs particuliers)

- Destruction avec

```
pthread_mutex_destroy(pthread_mutex_t *mutex).
```

Mutex

- Verrouillage avec

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

  - Si le mutex est libre : verrouillage, et les appels à `pthread_mutex_lock()` seront alors bloquant pour les autres threads ;
  - Si le `mutex` est déjà verrouillé : le thread sera lui aussi bloqué ;

- Déverrouillage avec

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- `int pthread_mutex_trylock(pthread_mutex_t *mutex)` ne bloque pas le thread si le mutex est verrouillé.

Mutex

- Définitions d'attributs avec

  `int pthread_mutexattr_set*(...)` :

  - `prioceiling` : priorité maximale à laquelle sera exécutée une section critique
  - `protocol` : changement de priorité suivant différents cas ;
  - `pshared` : pour autoriser le partage d'un mutex entre plusieurs processus (cf section sur la mémoire partagée) ;
  - `robust` : comportement en cas de terminaison du thread ayant verrouillé le mutex ;
  - `type` : comportement du mutex lors du (dé)verrouillage.

Problème du rendez-vous

Considérons une application découpée en N threads. Chacun de ces threads travaille en deux phases.

- Durant la première phase, tous les threads sont indépendants et peuvent s'exécuter simultanément.
- Cependant, un thread ne peut démarrer sa seconde phase que si tous les N threads ont terminé leur première phase.
- L'organisation de chaque thread est donc:

```
premiere_phase();
// rendez-vous
seconde_phase();
```

Problème du rendez-vous

Chaque thread doit pouvoir être bloqué à la fin de la première phase en attendant que tous les autres threads aient fini d'exécuter leur première phase. Cela peut s'implémenter en utilisant un `mutex` et un `sémaphore`.

```
sem_t rendezvous;
pthread_mutex_t mutex;
int count=0;

sem_init(&rendezvous,0,0);
```

La variable `count` permet de compter le nombre de threads qui ont atteint le point de rendez-vous.

Le `mutex` protège les accès à la variable `count`.

Le sémaphore `rendezvous` est initialisé à la valeur 0.

Problème du rendez-vous

```
premiere_phase();

 // section critique
 pthread_mutex_lock(&mutex);
 count++;
  if(count==N) {
     // tous les threads sont arrivés
     sem_post(&rendezvous);
     }
 pthread_mutex_unlock(&mutex);
// attente à la barrière
sem_wait(&rendezvous);
// libération d'un autre thread en attente
sem_post(&rendezvous);

seconde_phase();
```

Problème des producteurs-consommateurs

- les producteurs : Ce sont des threads qui produisent des données et placent le résultat de leurs calculs dans une zone mémoire accessible aux consommateurs.
- les consommateurs : Ce sont des threads qui utilisent les valeurs calculées par les producteurs.
- Ces deux types de threads communiquent en utilisant un buffer qui a une capacité limitée.
- La difficulté du problème est de trouver une solution qui permet aux producteurs et aux consommateurs d'avancer à leur rythme sans que les producteurs ne bloquent inutilement les consommateurs et inversement

Problème des producteurs-consommateurs

```
// Initialisation
#define N 10 // slots du buffer
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

pthread_mutex_init(&mutex, NULL);
sem_init(&empty, 0 , N);  // buffer vide
sem_init(&full, 0 , 0);   // buffer vide
```

```
// Producteur
void producer(void)
{
  int item;
  while(true)
  {
    item=produce(item);
    sem_wait(&empty); // attente d'un slot libre
    pthread_mutex_lock(&mutex);
     // section critique
     insert_item();
    pthread_mutex_unlock(&mutex);
    sem_post(&full); // il y a un slot rempli en plus
  }
}
```

Problème des producteurs-consommateurs

```
// Consommateur
void consumer(void)
{
 int item;
 while(true)
 {
   sem_wait(&full); // attente d'un slot rempli
   pthread_mutex_lock(&mutex);
    // section critique
    item=remove(item);
   pthread_mutex_unlock(&mutex);
   sem_post(&empty); // il y a un slot libre en plus
 }
}
```

Problème des *readers-writers*

- Le problème des readers-writers est un peu différent du précédent. Il permet de modéliser un problème qui survient lorsque des threads doivent accéder à une base de données. Les threads sont généralement de deux types.
  - les lecteurs (*readers*) sont des threads qui lisent une structure de données (ou une base de données) mais ne la modifient pas. Comme ces threads se contentent de lire de l'information en mémoire, rien ne s'oppose à ce que plusieurs *readers* s'exécutent simultanément.
  - les écrivains (writers). Ce sont des threads qui modifient une structure de données (ou une base de données). Pendant qu'un *writer* manipule la structure de données, il ne peut y avoir aucun autre *writer* ni de *reader* qui accède à cette structure de données. Sinon, la concurrence des opérations de lecture et d'écriture donnerait un résultat incorrect.

Problème des *readers-writers*

- Une première solution à ce problème est d'utiliser un mutex et un sémaphore.

```
pthread_mutex_t mutex;
sem_t db;  // accès à la db
int readcount=0; // nombre de readers

sem_init(&db, NULL, 1).
```

- La solution utilise une variable partagée : `readcount`. L'accès à cette variable est protégé par `mutex`. Le sémaphore `db` sert à réguler l'accès des *writers* à la base de données. Le *mutex* est initialisé comme d'habitude par la fonction `pthread_mutex_init()`. Le sémaphore `db` est initialisé à la valeur 1.

Problème des *readers-writers*

```
void writer(void)
{
 while(true)
 {
   prepare_data();
   sem_wait(&db);
     // section critique, un seul writer à la fois
     write_database();
   sem_post(&db);
   }
 }
}
```

```
void reader(void)
{
 while(true)
 {
   pthread_mutex_lock(&mutex);
     // section critique
     readcount++;
     if (readcount==1)
     { // arrivée du premier reader
       sem_wait(&db);
     }
   pthread_mutex_unlock(&mutex);
   read_database();
   pthread_mutex_lock(&mutex);
     // section critique
     readcount--;
     if(readcount==0)
     { // départ du dernier reader
       sem_post(&db);
     }
   pthread_mutex_unlock(&mutex);
   process_data();
 }
}
```

Problème des *readers-writers*

- Cette solution fonctionne et garantit qu'il n'y aura jamais qu'un seul *writer* qui accède à la base de données.
- Malheureusement, elle souffre d'un inconvénient majeur lorsqu'il y a de nombreux *readers*. Dans ce cas, il est tout à fait possible qu'il y ait en permanence des *readers* qui accèdent à la base de données et que les *writers* soient toujours empêchés d'y accéder.
- Dès que le premier *reader* a effectué `sem_wait(&db)`, aucun autre *reader* ne devra exécuter cette opération tant qu'il restera au moins un *reader* accédant à la base de données.
- Les *writers* par contre resteront bloqués sur l'exécution de `sem_wait(&db)`.

Problème des *readers-writers*

```
/* Initialisation */
pthread_mutex_t mutex_readcount; // protège readcount
pthread_mutex_t mutex_writecount; // protège writecount
pthread_mutex_t z; // un seul reader en attente
sem_t wsem;        // accès exclusif à la db
sem_t rsem;        // pour bloquer des readers
int readcount=0;
int writecount=0;

sem_init(&wsem, 0, 1);
sem_init(&rsem, 0, 1);
```

```
/* Writer */
while(true)
{
  think_up_data();

  pthread_mutex_lock(&mutex_writecount);
    // section critique - writecount
    writecount=writecount+1;
    if(writecount==1) {
      // premier writer arrive
      sem_wait(&rsem);
    }
  pthread_mutex_unlock(&mutex_writecount);

  sem_wait(&wsem);
    // section critique, un seul writer à la fois
    write_database();
  sem_post(&wsem);

  pthread_mutex_lock(&mutex_writecount);
     // section critique - writecount
     writecount=writecount-1;
     if(writecount==0) {
       // départ du dernier writer
       sem_post(&rsem);
     }
  pthread_mutex_unlock(&mutex_writecount);
 }
```

```
/* Reader */
while(true)
{
  pthread_mutex_lock(&z);
    // exclusion mutuelle, un seul reader en attente sur rsem
    sem_wait(&rsem);

    pthread_mutex_lock(&mutex_readcount);
       // exclusion mutuelle, readercount
       readcount=readcount+1;
       if (readcount==1) {
          // arrivée du premier reader
          sem_wait(&wsem);
       }
  pthread_mutex_unlock(&mutex_readcount);
  sem_post(&rsem);  // libération du prochain reader
  pthread_mutex_unlock(&z);

  read_database();

  pthread_mutex_lock(&mutex_readcount);
     // exclusion mutuelle, readcount
     readcount=readcount-1;
     if(readcount==0) {
          // départ du dernier reader
          sem_post(&wsem);
     }
  pthread_mutex_unlock(&mutex_readcount);
  use_data_read();
}
```