

Synchronisation à l'aide des Sémaphores et Moniteurs Posix

Samia Bouzefrane

Maître de Conférences

CEDRIC –CNAM

samia.bouzefrane@cnam.fr
<http://cedric.cnam.fr/~bouzefra>

Sommaire

Synchronisation en C/Posix

- Les sémaphores Posix
- Les variables condition Posix
- Exemples d'utilisation

Mécanismes étudiés

- ♦ **Mutex Posix (sémaphore binaire)**
- ♦ **Sémaphores Posix**
- ♦ **Moniteurs Posix : variables conditionnelles associées aux Mutex**
- ♦ **Nous n'abordons pas**
 - l'utilisation de tubes ("pipe")
 - l'utilisation de « sockets » (API TCP)
 - l'utilisation des sémaphores à la Unix (tableau de sémaphores)
(voir Livre de Systèmes d'exploitation- exercices en C/Posix et Java,
de Samia Bouzefrane, édition Dunod 2003)

Gestion des Mutex

- Un « **Mutex** » est un **sémaphore binaire** pouvant prendre un des deux états
- "**lock**" (verrouillé) ou "**unlock**" (déverrouillé): valeur de sémaphore 1 ou 0
- Un « **Mutex** » ne peut être partagé que par des threads d'un même processus
- Un « **Mutex** » ne peut être verrouillé que par une seule thread à la fois.
- Une thread qui tente de verrouiller un « **Mutex** » déjà verrouillé est suspendu jusqu'à ce que le « **Mutex** » soit déverrouillé.

Déclaration et initialisation d'un Mutex

- Un mutex est une variable de type « `thread_mutex_t` »
- Il existe une constante `PTHREAD_MUTEX_INITIALIZER` de ce type permettant une déclaration avec initialisation statique du mutex (avec les valeurs de comportement par défaut)

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

- Un mutex peut également être initialisé par un appel de la primitive

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutexattr_t *mutexattr);
```

avec une initialisation par défaut lorsque `mutexattr` vaut `NULL`

```
ex: pthread_mutex_init(&monMutex, NULL);
```

Prise (verrouillage) d'un mutex

- Un mutex peut être **verrouillé** par la primitive

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

- Si le mutex est **déverrouillé** il devient **verrouillé**
- Si le mutex est déjà verrouillé par une autre thread la tentative de verrouillage **suspend** l'appelant jusqu'à ce que le mutex soit déverrouillé.

Relâchement (déverrouillage) d'un mutex

- Un mutex peut être déverrouillé par la primitive

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

- Si le mutex est déjà déverrouillé, cet appel n'a aucun effet (comportement par défaut)
- Si le mutex est verrouillé, une des threads en attente obtient le mutex (qui reprend alors l'état verrouillé) et cette thread redevient active (elle n'est plus bloquée)
- L'opération est toujours *non bloquante pour l'appelant*

Exemple d'utilisation de mutex

```
#define N      10  /* Nb de cases du tampon */  
  
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;  
int tampon[N];
```

```
void Deposer(int m){  
pthread_mutex_lock(&mutex);  
    // déposer m dans tampon;  
    .....
```

```
pthread_mutex_unlock(&mutex);  
}
```

.....

On verrouille le Mutex : accès exclusif

On déverrouille le Mutex

Gestion des sémaphores Posix

- Un sémaphore Posix est un sémaphore à compte pouvant être partagé par plusieurs threads de plusieurs processus
- Le **compteur** associé à un sémaphore peut donc prendre des valeurs plus grande que 1 (contrairement à un mutex)
- La prise d'un sémaphore dont le compteur est négatif ou nul **bloque** l'appelant.
- Il ne faut pas confondre les sémaphores Posix avec les sémaphores Unix qui sont en fait des tableaux de sémaphores (non étudiés ici).

Rappel du concept de sémaphore

- **Un sémaphore s :**
{**Val(s)**: valeur qui doit toujours être initialisée,
File(s): file d'attente qui va contenir les processus bloqués sur ce sémaphore }
- **La valeur initiale d'un sémaphore ne doit jamais être négative.**

Primitive P(s):

Debut

$Val(s) = Val(s) - 1;$

Si $Val(s) < 0$ Alors Mettre le processus actif dans la file File(s);

Fin

Primitive V(s):

Debut

$Val(s) = Val(s) + 1;$

Si $Val(s) \leq 0$ Alors /* il y a au moins un processus bloqué dans File(s) */
Sortir un processus de la file File(s);

Fin

Création / Initialisation d'un sémaphore Posix

- Les prototypes des fonctions et les types sont définis dans « **semaphore.h** »
- Un sémaphore est une variable de type « **sem_t** »
- Il est initialisé par un appel à la primitive :

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

- si "**pshared**" vaut 0 le sémaphore ne peut pas être partagé entre threads de différents processus (partage uniquement au sein d'un même processus)
- **valeur** définit la valeur initiale de ce sémaphore (positive ou nulle)

Prise / Relâchement d'un sémaphore Posix

- Les deux opérations P et V sont implémentées par

```
P : int sem_wait(sem_t *sem);  
V : int sem_post(sem_t *sem);
```

avec les mêmes comportements que les primitives génériques P et V

- Il existe également une version non bloquante de la primitive P :

```
int sem_trywait(sem_t *sem);
```

qui retourne 0 lorsque la prise est possible (et non bloquante) et qui retourne **EAGAIN** sinon (dans le cas où l'appel normal serait bloquant)

Lecture de la valeur d'un sémaphore Posix

- La primitive

```
int sem_getvalue(sem_t *sem, int *sval);
```

qui retourne dans `sval` la valeur courante du sémaphore `sem`

- La primitive

```
int sem_destroy(sem_t *sem);
```

qui permet de libérer les ressources associées au sémaphore `sem`

Sémaphores vs Mutex

- Les Mutex constituent un cas particulier de sémaphores Posix.
- Les Mutex correspondent à des sémaphores binaires.
 - **Avantages de Mutex**
 - Les primitives de manipulation de Mutex servent uniquement à implémenter une exclusion mutuelle.
 - Les primitives de manipulation de Mutex sont faciles à utiliser.
 - **Avantages sémaphores Posix**
 - Les sémaphores Posix implémentent n'importe quel type de synchronisation entre les threads d'un même processus (en plus de l'exclusion mutuelle).

Exemple du Prod/Cons avec les Sémaphores Posix

```
/* prodconsThread.c avec des threads*/  
#include <stdio.h>  
#include <unistd.h>  
#include <semaphore.h>  
#include <fcntl.h>  
  
#define Ncases 10 /* nbr de cases du tampon */  
  
int Tampon[Ncases]; /* Tampon a N cases*/  
sem_t Snvide, Snplein; /* les sémaphores */
```

Exemple du Prod/Cons /Suite

```
void *Producteur(void) {
    int i, queue=0, MessProd;

    srand(pthread_self());

    for(i=0; i<20; i++){
        sleep(rand()%3); /* fabrique le message */
        MessProd = rand() % 10000;
        printf("Product %d\n",MessProd);
        sem_wait(&Snvide);
        Tampon[queue]=MessProd;
        sem_post(&Snplein);
        queue=(queue+1)%Ncases;
    }
    pthread_exit (0);
}
```

Exemple du Prod/Cons /Suite

```
void *Consommateur(void) {
    int tete=0, MessCons, i;

    srand(pthread_self());
    for(i=0; i<20; i++){
        sem_wait(&Snplein);
        MessCons = Tampon[tete];
        sem_post(&Snvide);
        tete=(tete+1)%Ncases;
        printf("\t\tConsomm  %d \n",MessCons);
        sleep(rand()%3); /* traite le message */
    }

    pthread_exit (0);
}
```

Exemple du Prod/Cons (Fin)

```
int    main(void) {
        pthread_t th1, th2;

        /* creation et initialisation des semaphores */
        sem_init(&Snvide, 0, Ncases);
        sem_init(&Snplein, 0, 0);

        /* creation des threads */
        pthread_create (&th1, 0, Producteur, NULL);
        pthread_create (&th2, 0, Consommateur, NULL);

        /* attente de terminaison */

        pthread_join (th1, NULL);
        pthread_join (th2, NULL);

        /* suppression des semaphores */
        sem_destroy(&Snplein);
        sem_destroy(&Snvide);
    return (0);
}
```

Exécution du Prod/Cons

```
$gcc prodconsThread.c -o prodcons -lpthread
$./prodcons
Product 2100
Product 3250

        Consomm  2100
Product 2540
        Consomm  3250
        Consomm  2540

...
```

Rappel du concept de moniteur

- **Concept** proposé par **Hoare** en 1974 pour résoudre le problème de synchronisation.

Type m = moniteur

Début

Déclaration des variables locales;

Déclaration et corps des procédures du moniteur; // **accessibles en exclusion mutuelle**

Initialisation;

Fin

- **Les procédures du moniteur se synchronisent à l'aide de deux primitives :**

Wait()

Signal()

qui permettent de bloquer ou de réveiller un processus sur une condition.

- **Une condition est une variable qui n'a pas de valeur mais qui est implémentée à l'aide d'une file d'attente.**

- **Syntaxe des primitives :**

Cond.Wait() : bloque toujours le processus appelant

Cond.Signal() : réveille un processus bloqué dans la file d'attente associée à **Cond.**

Moniteurs Posix

- **Un moniteur Posix est l'association**

- **d'un mutex** (type `pthread_mutex_t`) qui sert à protéger la partie de code où l'on teste les conditions de progression
- **et d'une variable condition** (type `pthread_cond_t`) qui sert de point de signalisation :

- on se met en attente sur cette variable par la primitive :

```
pthread_cond_wait(&laVariableCondition,&leMutex);
```

- on est réveillé sur cette variable avec la primitive :

```
pthread_cond_signal(&laVariableCondition);
```

Schéma d'utilisation

- Soit la condition de progression C,
- Le schéma d'utilisation des moniteurs Posix est le suivant :

```
pthread_mutex_lock (&leMutex);  
évaluer C;  
while ( ! C ) {  
    pthread_cond_wait(&laVariableCondition,&leMutex);  
    ré-évaluer C si nécessaire  
}  
  
Faire le travail;  
pthread_mutex_unlock(&leMutex);
```

Exemple du Prod/Cons avec les moniteurs Posix

```
#include <pthread.h>

/* définition du tampon */
#define N      10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */
int NbPleins=0;
int tete=0, queue=0;
int tampon[N];

/* définition des conditions et du mutex */
pthread_cond_t vide;
pthread_cond_t plein;
pthread_mutex_t mutex;

pthread_t tid[2];
```

Exemple (suite)

```
void Deposer(int m){
pthread_mutex_lock(&mutex);
    if(NbPleins == N) pthread_cond_wait(&plein, &mutex);
    tampon[queue]=m;
    queue=(queue+1)%N;
    NbPleins++;
    pthread_cond_signal(&vide);
pthread_mutex_unlock(&mutex);
}

int Prelever(void){
int m;
pthread_mutex_lock(&mutex);
    if(NbPleins ==0) pthread_cond_wait(&vide, &mutex);
    m=tampon[tete];
    tete=(tete+1)%N;
    NbPleins--;
    pthread_cond_signal(&plein);
pthread_mutex_unlock(&mutex);
return m;
}
24}
```

Exemple (suite)

```
void * Prod(void * k)                /****** PRODUCTEUR */
{
int i;
int mess;
 srand(pthread_self());
for(i=0;i<=NbMess; i++){
    usleep(rand()%10000); /* fabrication du message */
    mess=rand()%1000;
     Deposer(mess);
    printf("Mess depose: %d\n",mess);
}
}
void * Cons(void * k)                /****** CONSOMMATEUR */
{
int i;
int mess;
 srand(pthread_self());
for(i=0;i<=NbMess; i++){
     mess=Prelever();
    printf("\tMess preleve: %d\n",mess);
    usleep(rand()%1000000); /* traitement du message */
}
}
}
25 }
```

Exemple (fin)

```
void main()                                /* M A I N */
{
  int i, num;
  pthread_mutex_init(&mutex,0);
  pthread_cond_init(&vide,0);
  pthread_cond_init(&plein,0);

  /* creation des threads */
  pthread_create(tid, 0, (void * (*)()) Prod, NULL);
  pthread_create(tid+1, 0, (void * (*)()) Cons, NULL);

  // attente de la fin des threads
  pthread_join(tid[0],NULL);
  pthread_join(tid[1],NULL);

  // libération des ressources
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&vide);
  pthread_cond_destroy(&plein);

  exit(0);
}
```

Conclusion

- **Posix propose plusieurs mécanismes**
 - **Mutex** : pour les sections critiques
 - **Sémaphores** : pour la synchronisation entre threads en général
 - **Variables condition** : pour l'utilisation de moniteur

Références

Jean-François Peyre, supports de cours sur l'informatique industrielle-systèmes temps réel, CNAM(Paris).

Samia Bouzefrane, LES SYSTEMES D'EXPLOITATION: COURS ET EXERCICES CORRIGES UNIX, LINUX et WINDOWS XP avec C et JAVA (566 pages), Dunod Editeur, Octobre 2003, ISBN : 2 10 007 189 0.