Programmation Système sous Linux

Pr. Karima AISSAOUI

Plan de la séance

- Rappel
- Synchronisation des threads
- Synchronisation par Mutex
- Conditions

Rappel

- la programmation concurrente permet à un programme d'effectuer plusieurs tâches simultanément au lieu de devoir attendre la fin d'une opération pour commencer la suivante.
- Il existe trois façons d'implémenter la concurrence dans nos programmes :
  - les processus
  - les threads
  - le multiplexage

Synchronisation

- Des mécanismes sont fournis pour permettre la synchronisation des différents threads au sein de la même tâche:
  - La primitive « pthread_join() » (synchronisation sur terminaison): permet à un thread d'attendre la terminaison d'un autre.
  - Synchronisation avec une attente active (solution algorithmique).
  - Les « mutex » (sémaphores d’exclusion mutuelle): C’est un mécanisme qui permet de résoudre le problème de l'exclusion mutuelle des threads.
  - Les conditions (évènements).

Rappel

- Les threads partagent leur mémoire
  - Un des grands atouts des threads est qu'ils partagent tous la mémoire de leur processus. Chaque thread possède sa propre stack, oui, mais les autres threads peuvent très facilement y accéder à l'aide d'un simple pointeur.
- Le partage de mémoire est généralement voulu et avantageux :
  - cela évite de gaspiller de la mémoire
  - c'est un mécanisme de communication inter-thread (et inter-processus) très rapide
- L'important est de bien savoir gérer l'accès concurrent à la mémoire

Exemple

- Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :
  - y compris au milieu d'une ligne
  - y compris au milieu d'une instruction basique en C (ex : i++)
- Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistant.
  - Comportement imprévisible ! (bug, plantage, exploitation, mauvaise note)

Attente active et passive

- L'attente avant d'entrer dans une section critique peut être :
  - active (spinlock) : le thread continue de tourner jusqu'à ce qu'il a le droit d'entrer dans la section critique
  - passive : le thread est mis en pause jusqu'à ce qu'il a la possibilité de rentrer dans la section critique. Dans ce cas, il faut interférer avec l'ordonnanceur
  - Avantage : on laisse le temps CPU aux autres threads qui peuvent faire des choses plus constructives

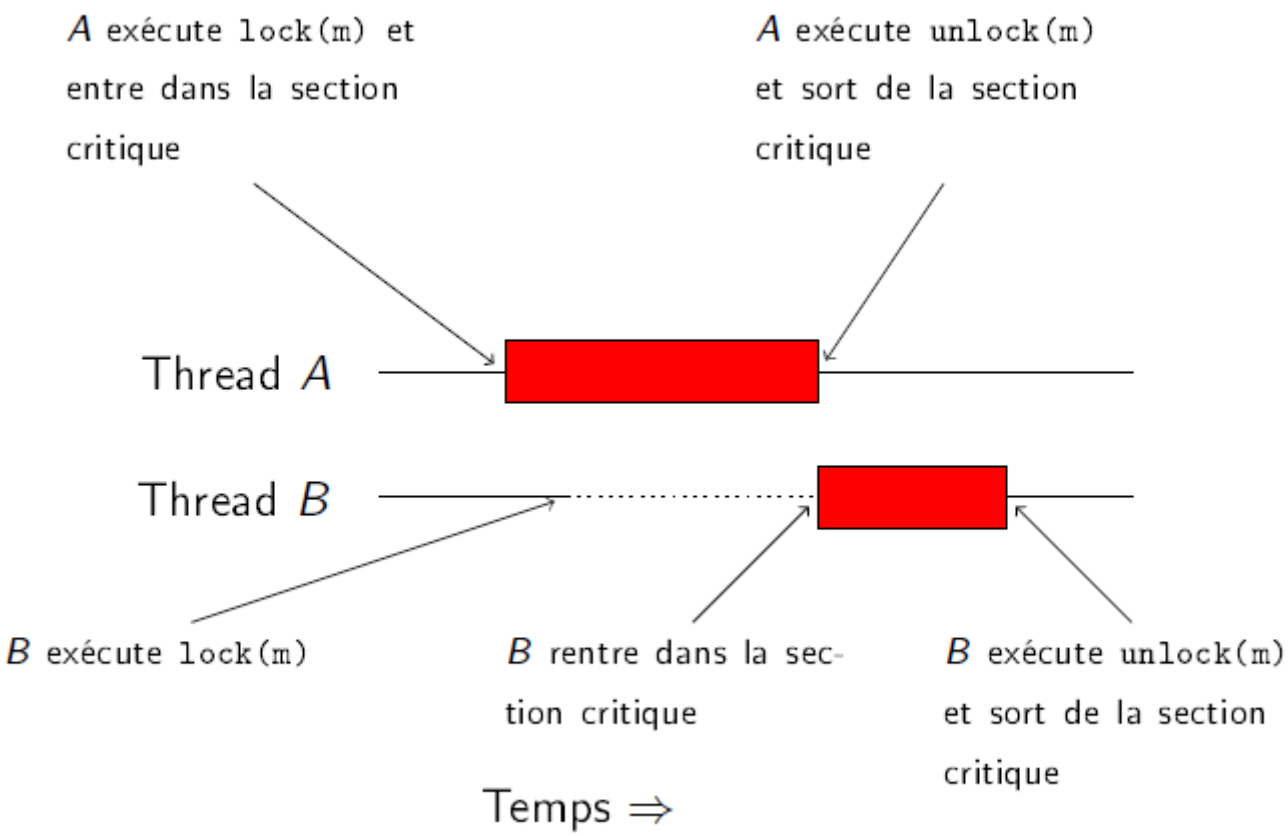
Les primitives

- En général, on ne reprogramme pas soi même les tests d'entrée dans une section critique.
  - C'est fastidieux
  - le risque d'erreur est très important
  - on ne tire pas parti des possibilités de l'OS.
- On passe par des primitives (du langage, de librairies et/ou du système) : des **verrous** .
  - Alors, peut-on empêcher un thread de lire une valeur quand un autre thread la modifie ? Oui, à l'aide de mutex !

Mutex

- Un mutex (abrégé de « mutual exclusion » en anglais, c'est à dire « exclusion mutuelle »)
- c'est une primitive de synchronisation. C'est un verrou qui permet de réguler l'accès aux données et empêcher que les ressources partagées soient utilisées en même temps.
- sont des objets de type «pthread_mutex_t » qui permettent de mettre en œuvre un mécanisme de synchronisation qui résout le problème de l'exclusion mutuelle.
- Permettent d'éviter que des ressources partagées d'un système ne soient utilisées en même temps par plus qu'un thread.

Mutex


A exécute lock(m) et entre dans la section critique
A exécute unlock(m) et sort de la section critique
Thread A
Thread B
B exécute lock(m)
B rentre dans la section critique
B exécute unlock(m) et sort de la section critique
Temps ⇒

Mutex

- Il existe deux états pour un mutex (disponible ou verrouillé), et essentiellement deux fonctions de manipulation des mutex (une fonction de verrouillage et une fonction de libération).
  - Lorsqu'un mutex est verrouillé par un thread, on dit que ce thread tient le mutex. Tant que le thread tient le mutex, aucun autre thread ne peut accéder à la ressource critique.
    - Un mutex ne peut être tenu que par un seul thread à la fois.
  - Lorsqu'un thread demande à verrouiller un mutex déjà maintenu par un autre thread, le thread demandeur est bloqué jusqu'à ce que le mutex soit libéré.
- Le principe des mutex est basé sur l'algorithme de Dijkstra (l'algorithme de Dijkstra sert à résoudre le problème du plus court chemin)
  - Opération P (verrouiller l'accès) .
  - Accès à la ressource critique (la variable globale) .
  - Opération V (libérer l'accès) .

Fonctions de gestion des Mutex

- pthread_mutex_init ( ): permet de créer le mutex (le verrou) et le mettre à l'état "unlock" (ouvert ou disponible).
- pthread_mutex_destroy( ): permet de détruire le mutex.
- pthread_mutex_lock ( ): tentative d'avoir le mutex. S'il est déjà pris, le thread est bloqué
- pthread_mutex_trylock( ): appel non bloquant. Si le mutexest déjà pris, le thread n'est pas bloqué
- pthread_mutex_unlock( ) : rend le mutex et libère un thread

Création de mutex

- La création de mutex (verrou) consiste à définir un objet de type «pthread_mutex_t » et de l'initialiser de deux manières.
  - Initialisation statique à l'aide de la constante « PTHREAD_MUTEX_INITIALIZER »
  - Initialisation par appel de la fonction « pthread_mutex_init() » déclarée dans <pthread.h>, qui permet de créer le verrou(lemutex) et le mettre en état "unlock".
- Syntaxe

int pthread_mutex_init(pthread_mutex_t *mutex_pt,pthread_mutexattr_t *attr) ;

  - « mutex_pt » : pointe sur une zone réservée pour contenir le mutex créé.
  - attr : ensemble d'attributs à affecter au mutex. On le met à NULL
- Code retour de la fonction:
  - 0 : en cas de succès
  - !=0 : en cas d'échec

Verrouiller et déverrouiller un mutex

- pour verrouiller et déverrouiller notre mutex, il nous faudra deux autres fonctions qui ont les prototypes suivants :

```
int pthread_mutex_lock(pthread_mutex_t *mutex)); // Verrouillage
int pthread_mutex_unlock(pthread_mutex_t *mutex); // Déverrouillage
```

L’opération P pour un mutex: Appel bloquant

- La fonction « pthread_mutex_lock() » permet, à un thread de réaliser de façon atomique, une opération P (verrouiller le mutex) par un thread. Si le mutex est déjà verrouillé (tenu par un autre thread), alors le thread qui appelle la fonction «pthread_lock() » reste bloqué jusqu’à la réalisation de l’opération V (libérer le mutex) par un autre thread.
- Syntaxe:
- int pthread_mutex_lock(pthread_mutex_t *mutex_pt);
  - - mutex_pt : pointe sur le mutex à réserver (à verouiller)
- Code retour de la fonction :
  - 0 : en cas de succès
  - !=0 : en cas d'erreur

L’opération P pour un mutex: Appel non bloquant

- La fonction «pthread_mutex_trylock » permet, de façon atomique, de réserver un mutex ou de renvoyer une valeur particulière si le mutex est réservé par un autre thread, le thread n'est pas bloqué.
- Syntaxe:
- int pthread_mutex_trylock(pthread_mutex_t *mutex_pt);
  - « mutex_pt » : pointe sur le mutex à réserver
- Code retour de la fonction:
  - 1: en cas de réservation
  - 0 : en cas d'échec de la réservation
  - !=0 : en cas d'erreur

L’opération V pour un mutex

- L’appel de la fonction « pthread_mutex_unlock() » permet de libérer un mutex et de débloquer les threads en attente sur ce mutex.
- Syntaxe:
- int pthread_mutex_unlock(pthread_mutex_t *mutex_pt);
  - mutex_pt : pointe sur le mutex à libérer
- Code retour de la fonction :
  - 0 : en cas de succès
  - !=0 : en cas d'erreur

Exemple

- Reprendre l’exemple des tableaux.
- Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher son contenu en utilisant les sémaphores d’exclusion mutuelle (les mutex)

Solution avec les mutex

```
#include <stdio.h>
#include <pthread.h>
#define N 1000
pthread_t              th1, th2;
pthread_mutex_t        mutex;
int                    tab[N];
void *ecriture_tab (void * arg);
void *lecture_tab (void * arg);
main ( ) {
  pthread_mutex_init (&mutex, NULL);
  pthread_mutex_lock (&mutex);
  pthread_create (&th1, NULL, ecriture_tab, NULL) ;
  pthread_create (&th2, NULL, lecture_tab, NULL);
  pthread_join (th1, NULL);
  // pthread_mutex_unlock (&mutex);
 pthread_join(th2,NULL);
}
```

Solution avec les mutex - suite

```
void *ecriture_tab (void * arg) {
    int i;
    for (i = 0 ; i <N ; i++) {
        tab[i] = 2 * i;
        printf ("écriture, tab[%d] vaut %d\n", i, tab[i]);
    }
    pthread_mutex_unlock (&mutex);
    pthread_exit (NULL);
}

void *lecture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    for (i = 0 ; i < N ; i++)
        printf ("lecture, tab[%d] vaut %d\n", i, tab[i]);
    pthread_mutex_unlock (&mutex);
}
```

Détruire un mutex

- Quand on n'a plus besoin d'un mutex, il nous faut le détruire avec la fonction pthread_mutex_destroy suivante :

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

- mutex : pointe sur le mutex à détruire
- Code retour de la fonction:
  - 0 : en cas de succès
  - !=0 : en cas d'erreur

Les conditions

- Une condition est un mécanisme permettant de synchroniser plusieurs threads à l'intérieur d'une section critique.
- C’est une autre technique de synchronisation qui utilise les variables de « conditions » représentées par le type « pthread_cond_t ».

Principe des conditions

- Le principe est simple. Lorsqu'un thread doit attendre une condition pour exécuter sa tâche, nous pouvons le mettre en attente.
- Un autre thread qui possède le verrou réveille alors le thread dormant lorsque la condition est remplie.
- Tous les threads ne sont pas des boucles infinies, certains sont créés juste pour faire une action précise puis sont automatiquement détruits, mais pour d'autres, les conditions sont un atout dans la consommation des ressources, car l'attente d'une condition met systématiquement le thread en pause !

Principe des conditions

- Le principe consiste à bloquer un thread (une activité) sur une attente d'évènement (le thread se met en attente d'une condition).
- Lorsque la condition est réalisée par un autre thread, ce dernier l'en avertit directement.
- Pour cela on utilise essentiellement deux fonctions de manipulation des conditions:
  - l'une est l'attente de la condition, le thread appelant reste bloqué jusqu'à ce qu'elle soit réalisée;
  - et l'autre sert à signaler que la condition est remplie.

Principe des conditions

- La variable condition est considérée comme une variable booléenne un peu spéciale par la bibliothèque «pthreads».
- Elle est toujours associée à un « mutex », afin d'éviter les problèmes de concurrences.
  - La variable de condition sert à la transmission des changements d'état.
  - Le mutex assure un accès protégé à la variable.

Le thread qui doit attendre une condition

- On initialise la variable condition et le mutex qui lui est associé.
  - Le thread bloque le mutex. Ensuite, il invoque une routine d'attente (par exemple la routine « pthread_cond_wait() ») qui attend que la condition soit réalisée.
  - Le thread libère le mutex.

Le thread qui réalise la condition

- Le thread travaille jusqu'à avoir réalisé la condition attendue
- Il bloque le mutex associé à la condition
- Le thread appelle la fonction « pthread_cond_signal() » pour montrer que la condition est remplie.
- Le thread débloque le mutex.

Création (Initialisation) des variables de condition

- Une condition est de type « pthread_cond_t » peut être initialisée:
  - de manière statique: **pthread_cond_t condition=PTHREAD_COND_INITIALIZER**;
  - par appel de la fonction « pthread_cond_init() » qui permet de créer une nouvelle condition.
- Syntaxe:

int pthread_cond_init(pthread_cond_t * cond_pt, pthread_condattr_t *attr);

  - « cond_pt » : pointeur sur la zone réservée pour recevoir la condition (pointe sur la nouvelle condition).
  - « attr » : attributs à donner à la condition lors de sa création. Il est mis à NULL.
- Code retour de la fonction:
  - 0 : en cas de succès
  - !=0 : en cas d'erreur.

pthread_cond_wait

- La fonction « pthread_cond_wait() » permet l'attente d'une condition envoyée par exemple par la fonction « pthread_cond_signal() ».
- Syntaxe:
- int pthread_cond_wait(pthread_cond_t *cond_pt, pthread_mutex_t *mutex_pt);
  - « cond_pt » : pointeur sur la condition à attendre
  - « mutex_pt » : pointeur sur le mutex à libérer pendant l'attente.
- Code retour de la fonction:
  - 0 : en cas de succès
  - !=0 : en cas d'erreur.
- Principe de fonctionnement:
- Le thread appelant doit avoir verrouiller au préalable le mutex «mutex_pt».
- L'appel a pour effet:
  - libérer le mutex « mutex_pt »
  - attendre un signal sur la condition désignée et de reprendre son exécution,en verrouillant à nouveau le mutex.

pthread_cond_signal

- La fonction « pthread_cond_signal() » permet de réveiller un des threads attendant la condition désignée.
  - Si aucun thread ne l'attendait, le signal est perdu.
  - Si plusieurs threads attendent sur la même condition, un seul d'entre eux est réveillé mais on ne peut pas prédire lequel.
- Syntaxe:
- int pthread_cond_signal(pthread_cond_t *cond_pt);
  - « cond_pt » : pointeur sur la condition à signaler
- Code retour de la fonction:
  - 0 : en cas de succès.
  - !=0 : en cas d'erreur.

pthread_cond_broadcast

- La fonction «pthread_cond_broadcast() » permet de réveiller tous les threads en attente sur une condition.
- Syntaxe

#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond_pt);

  - cond_pt : pointeur sur la condition à signaler
- Code retour de la fonction:
  - 0 : en cas de succès
  - !=0 : en cas d'erreur

pthread_cond_destroy()

- Une condition non utilisée peut être libérée par appel de la fonction« pthread_cond_destroy() ».
- Aucun autre thread ne doit être en attente sur la condition, sinon la libération échoue sur l'erreur EBUSY.
- Syntaxe:

int pthread_cond_destroy(pthread_cond_t *cond_pt);

- Permet de détruire les ressources associées à une condition
  - cond_pt : pointeur sur la condition à détruire.
- Code retour de la fonction :
  - 0 : en cas de succès.
  - !=0 : en cas d'erreur.

Schéma de mise en oeuvre des conditions

Thread attendant la condition	Thread signalant la condition
Appel de pthread_mutex_lock(): blocage du mutex associé à la condition	
Appel de pthread_cond_wait(): déblocage du mutex	
…. Attente ….	
	Appel de pthread_mutex_lock() sur le mutex
	Appel de pthread_cond_signal(), qui réveille l'autre thread
Dans pthread_cond_wait(), tentative de récupérer le mutex. Blocage	
	Appel de pthread_mutex_unlock(). Le mutex étant libéré, l'autre thread se débloque
Fin de pthread_cond_wait()	
Appel de pthread_mutex_unlock() pour revenir à l'état initial.	

Exemple

- Refaire l’exemple de la lecture et l’écriture d’un tableau global en utilisant les variables de conditions.
- Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher son contenu.

```
#include <stdio.h>
#include <pthread.h>
# define N   100
int tab[N];
pthread_t  thread1, thread2;
pthread_mutex_t  mutex;
pthread_cond_t  condition;
void *ecriture_tab (void * arg);
void *lecture_tab (void * arg);
main ( ) {
   pthread_mutex_init (&mutex,NULL);
   pthread_cond_init (&condition, NULL);
   pthread_create (&th1, NULL, ecriture_tab, NULL) ;
   pthread_create (&th2, NULL, lecture_tab, NULL);
   pthread_join(th1,NULL);
   pthread_join(th2,NULL);
}
```

```
void *ecriture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    for (i = 0 ; i <N ; i++)
        tab[i] = 2 * i;
    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
}
void *lecture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    pthread_cond_wait(&condition,&mutex);
     for (i = 0 ; i < N ; i++)
        printf ("lecture, tab[%d] vaut %d\n", i, tab[i]);
    pthread_mutex_unlock (&mutex);
}
```

Problème

- On risque de perdre le signal. Au moment du réveil l'autre thread n'est pas encore dans l'état d'attente.
- Une solution exacte peut être élaborée comme suit:

```
#include <stdio.h>
#include <pthread.h>
# define N   100
int tab[N];
pthread_t thread1, thread2;
pthread_mutex_t mutex;
pthread_cond_t condition;
int test =0;
void *ecriture_tab (void * arg);
void *lecture_tab (void * arg);
main ( ) {
   pthread_mutex_init(&mutex,NULL);
   pthread_cond_init(&condition, NULL);
    pthread_create (&th1, NULL, ecriture_tab, NULL) ;
    pthread_create (&th2, NULL,lecture_tab, NULL);
    pthread_join(th1,NULL);
   pthread_join(th2,NULL);
}
```

```
void *ecriture_tab (void * arg) {
      int i;
      pthread_mutex_lock (&mutex);
      for (i = 0 ; i <N ; i++)
            tab[i] = 2 * i;
      test=1;
      pthread_cond_signal(&condition);
      pthread_mutex_unlock(&mutex);
}
void *lecture_tab (void * arg) {
      int i;
      pthread_mutex_lock (&mutex);
      while (test==0)
            pthread_cond_wait(&condition, &mutex);
      for (i = 0 ; i < N ; i++)
            printf ("lecture, tab[%d] vaut %d\n", i, tab[i]);
      pthread_mutex_unlock (&mutex);
}
```