UNIVERSITÉ PARIS SUD
IUT D'ORSAY

DUT Informatique
M3101 Système S3
2015 / 2016

Travaux Dirigés n° 5 : Variables de condition

1 Variables de condition

Les mutex protègent les threads concurrents contre les erreurs d'interférence en donnant à un thread l'accès exclusif à des ressources partagées. Il peut pourtant arriver que le thread ne peut pas effectuer immédiatement l'opération désirée et doit se mettre en attente jusqu'à ce que l'état de la ressource change de manière qu'il puisse avancer. Par exemple, un thread qui essaie de débiter un compte bancaire ne peut le faire que quand le compte contient une somme suffisante. Si le thread garde le mutex pendant l'attente, il va empêcher aux autres threads d'accéder à la ressource partagée et de changer son état (dans notre exemple, créditer le compte). Si le thread libère le mutex, il est obligé de le reprendre plus tard pour un nouveau essai. Le faire immédiatement taxe lourdement le système, car le thread tourne en permanence dans une boucle « verrouillage-test-déverrouillage » (on appelle ce genre d'exécution une *attente active* ou *busy waiting*). En même temps, ajouter un délai avant une nouvelle tentative risque de ralentir inutilement le programme.

Une *variable de condition* (qu'on appelle aussi tout simplement une *condition*) est un outil de synchronisation qui permet à un thread de céder temporairement un mutex et de se mettre en attente d'une *notification* qui signalerait le changement de la ressource partagée. À la réception d'une telle notification, le thread reprend le mutex et ressaie d'effectuer l'opération.

On manipule les variables de condition à l'aide des cinq primitives suivantes qui retournent toutes 0 en cas de succès et un code d'erreur sinon :

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, NULL);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

La primitive `pthread_cond_init()` initialise une variable de condition dont on passe l'adresse en premier argument. Le second argument indique les attributs de la condtition, on indiquera `NULL` pour avoir les attributs par défaut.

La primitive `pthread_cond_signal()` envoie une notification à un thread qui est en attente sur la variable de condition au moment de l'appel. Si aucun thread n'est en train d'attendre sur la condition, la primitive n'a aucun effet. Si plusieurs threads sont en train d'attendre sur la condition, un seul parmi eux est notifié, mais on ne peut pas savoir lequel.

La primitive `pthread_cond_broadcast()` envoie une notification à tous les threads qui sont en attente sur la variable de condition au moment de l'appel. Si aucun thread n'est en train d'attendre sur la condition, la primitive n'a aucun effet.

La primitive `pthread_cond_wait(cond,mutex)` libère `mutex` (qui doit être verrouillé par le thread appelant au moment de l'appel) et suspend l'appelant jusqu'à l'arrivée d'une notification sur `cond` (suite à un appel de `pthread_cond_signal(cond)` ou `pthread_cond_broadcast(cond)` dans un autre thread). À la réception d'une notification, la primitive reprend le mutex (de la même façon que `pthread_mutex_lock()`) et retourne. Puisque le déverrouillage et la mise en attente se font de manière atomique (c'est-à-dire, inséparablement), il ne peut pas arriver qu'un autre thread verrouille le mutex libéré, change l'état de la ressource partagée et signale le changement avant que

`pthread_cond_wait()` puisse recevoir la notification. C'est une erreur d'appeler cette primitive sur la même condition avec les mutex différents.

La primitive `pthread_cond_destroy()` détruit la variable de condition passé en argument. Aucun thread ne doit être en attente sur cette condition au moment de l'appel. Il est possible de réinitialiser une condition détruite avec un nouvel appel à `pthread_cond_init()`.

2 Exercices

Exercice 1 : Un compte à rebours. Un compte à rebours peut être implémenté avec les trois opérations suivantes :

```
void car_init(int c);   // initialiser le compteur
void car_count_down();  // décrémenter le compteur
void car_wait();        // attendre la fin du compte
```

Tout thread qui appelle la fonction `car_wait()` doit être mis en attente jusqu'à ce que le compte atteigne 0.

Proposez une implémentation du compte à rebours.

Exercice 2 : Verrou de lecture-écriture. Assez souvent la plupart des accès à une ressource partagée ont pour but d'obtenir l'information sur son état sans le changer. Comme il n'y a aucun danger d'interférence entre plusieurs accès en lecture concurrents, il n'est pas nécessaire d'assurer l'exclusion mutuelle pour eux. Par contre, tout accès en écriture ne peut être fait qu'en absence de lecteurs et d'autres écrivains.

Un verrou de lecture-écriture est essentiellement une paire de verrous avec les quatre opérations suivantes :

```
void reader_lock();     // prise du verrou de lecture
void reader_unlock();   // libération du verrou de lecture
void writer_lock();     // prise du verrou d'écriture
void writer_unlock();   // libération du verrou d'écriture
```

Ces opérations assurent que

— si le verrou d'écriture est pris par un thread, alors tout appel de `reader_lock()` ou de `writer_lock()` met l'appelant en attente jusqu'à ce que le verrou d'écriture soit libéré par le thread écrivain ;

— si le verrou de lecture est pris par un ou plusieurs threads, alors tout appel de `writer_lock()` met l'appelant en attente jusqu'à ce que le verrou de lecture soit libéré par tous les threads lecteurs.

Proposez une implémentation des quatre fonctions ci-dessus en utilisant deux conditions.

Est-il vraiment nécessaire de retester la condition après le retour de `pthread_cond_wait()` ? Décrivez un scénario où l'absence de test après l'attente provoquera une violation des règles d'accès.

En cas d'un influx ininterrompu d'accès en lecture, un écrivain risque de se trouver dans une attente infinie, ce qu'on appelle une situation de *famine*. Une façon de contourner ce problème est de suspendre toute demande du verrou de lecture tant qu'il y a un processus bloqué en attente du verrou d'écriture.

Modifiez votre implémentation afin d'éviter la famine pour les threads écrivains.