TD3: Les conditions et clés

Marc Pérache

27 mars 2009

1 Introduction

1.1 Définitions

Définition 1: ***Processus*** **–** Un processus est une "coquille" dans laquelle le système exécute chaque programme (commande). Cette commande est un espace sûr ; en particulier chaque processus possède sa propre mémoire grâce au mécanisme de mémoire virtuelle. Il possède un numéro qui est unique sur le système, son pid, mais également un certain nombre de tables qui lui sont propres, comme la table des descripteurs ou celle de gestion des signaux. Chaque processus appartient à un utilisateur et un groupe et a les droits qui leur sont associés.

Définition 2: ***Thread*** **–** Un thread est une suite logique d'actions résultat de l'exécution d'un programme. Le contexte d'un thread comprend :
- une pile d'exécution,
- les contenus des registres matériels.

On peut donc dire qu'un processus contient un seul thread.

Définition 3: ***Section critique*** **–** Région du programme où l'on souhaite limiter (généralement à un seul) le nombre de flots d'exécution. Ces régions correspondent généralement à des zones où des invariants sur des données peuvent ne pas être respectés.

Définition 4: ***Attente active*** **–** Méthode de synchronisation où l'entité, attendant de passer cette synchronisation, garde le contrôle du processeur.

Définition 5: ***Réentrance*** **–** Fait, pour une fonction, de pouvoir être exécutée pendant son exécution. Une fonction sans effet de bord et travaillant uniquement avec des variables locales est de fait automatiquement réentrante.

Définition 6: ***Mutex*** **–** Un mutex permet l'exclusion mutuelle et la synchronisation entre threads.

Définition 7: ***Sémaphore*** **–** Les sémaphores sont purement et simplement des compteurs pour des ressources partagées par plusieurs threads.

Définition 8: ***Condition*** **–** Les conditions variables (condvar) permettent de réveiller un thread endormi en fonction de la valeur d'une variable.

2 Conditions

2.1 Sémantique POSIX des conditions

```
PTHREAD_COND(3)

NAME
       pthread_cond_init,      pthread_cond_destroy,     pthread_cond_signal,
       pthread_cond_broadcast,  pthread_cond_wait,  pthread_cond_timedwait  -
       operations on conditions

SYNOPSIS
       #include <pthread.h>

       pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

       int    pthread_cond_init(pthread_cond_t    *cond,   pthread_condattr_t
       *cond_attr);

       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_timedwait(pthread_cond_t   *cond,   pthread_mutex_t
       *mutex, const struct timespec *abstime);

       int pthread_cond_destroy(pthread_cond_t *cond);


DESCRIPTION
       A  condition  (short  for ''condition variable'') is a synchronization
       device that allows threads to suspend  execution  and  relinquish  the
       processors until some predicate on shared data is satisfied. The basic
       operations on conditions are: signal the condition (when the predicate
       becomes  true), and wait for the condition, suspending the thread exe-
       cution until another thread signals the condition.

       A condition variable must always be associated with a mutex, to  avoid
       the  race  condition  where  a  thread prepares to wait on a condition
       variable and another thread signals  the  condition  just  before  the
       first thread actually waits on it.

       pthread_cond_init  initializes  the condition variable cond, using the
       condition attributes specified in cond_attr, or default attributes  if
       cond_attr  is  NULL.   The  LinuxThreads implementation supports no
       attributes for conditions, hence the cond_attr parameter  is  actually
       ignored.

       Variables  of  type pthread_cond_t can also be initialized statically,
```

using the constant PTHREAD_COND_INITIALIZER.

pthread_cond_signal restarts one of the threads that are waiting on the condition variable cond. If no threads are waiting on cond, nothing happens. If several threads are waiting on cond, exactly one is restarted, but it is not specified which.

pthread_cond_broadcast restarts all the threads that are waiting on the condition variable cond. Nothing happens if no threads are waiting on cond.

pthread_cond_wait atomically unlocks the mutex (as per pthread_unlock_mutex) and waits for the condition variable cond to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to pthread_cond_wait. Before returning to the calling thread, pthread_cond_wait re-acquires mutex (as per pthread_lock_mutex).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable.

pthread_cond_timedwait atomically unlocks mutex and waits on cond, as pthread_cond_wait does, but it also bounds the duration of the wait. If cond has not been signaled within the amount of time specified by abstime, the mutex mutex is re-acquired and pthread_cond_timedwait returns the error ETIMEDOUT. The abstime parameter specifies an absolute time, with the same origin as time(2) and gettimeofday(2): an abstime of 0 corresponds to 00:00:00 GMT, January 1, 1970.

pthread_cond_destroy destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to pthread_cond_destroy. In the LinuxThreads implementation, no resources are associated with condition variables, thus pthread_cond_destroy actually does nothing except checking that the condition has no waiting threads.

RETURN VALUE

All condition variable functions return 0 on success and a non-zero error code on error.

ERRORS

pthread_cond_init, pthread_cond_signal, pthread_cond_broadcast, and pthread_cond_wait never return an error code.

The pthread_cond_timedwait function returns the following error codes on error:

ETIMEDOUT
the condition variable was not signaled until the time-out specified by abstime

EINTR pthread_cond_timedwait was interrupted by a signal

The pthread_cond_destroy function returns the following error code on error:

EBUSY some threads are currently waiting on cond.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_condattr_init(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), gettimeofday(2), nanosleep(2).

EXAMPLE

Consider two shared variables x and y, protected by the mutex mut, and a condition variable cond that is to be signaled whenever x becomes greater than y.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Waiting until x is greater than y is performed as follows:

```
pthread_mutex_lock(&mut);
while (x <= y) {
        pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
pthread_mutex_unlock(&mut);
```

Modifications on x and y that may cause x to become greater than y should signal the condition if needed:

```
pthread_mutex_lock(&mut);
/* modify x and y */
```

```
        if (x > y) pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mut);
```

If it can be proved that at most one waiting thread needs to be waken up (for instance, if there are only two threads communicating through x and y), pthread_cond_signal can be used as a slightly more efficient alternative to pthread_cond_broadcast. In doubt, use pthread_cond_broadcast.

To wait for x to becomes greater than y with a timeout of 5 seconds, do:

```
        struct timeval now;
        struct timespec timeout;
        int retcode;

        pthread_mutex_lock(&mut);
        gettimeofday(&now);
        timeout.tv_sec = now.tv_sec + 5;
        timeout.tv_nsec = now.tv_usec * 1000;
        retcode = 0;
        while (x <= y && retcode != ETIMEDOUT) {
                retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
        }
        if (retcode == ETIMEDOUT) {
                /* timeout occurred */
        } else {
                /* operate on x and y */
        }
        pthread_mutex_unlock(&mut);
```

LinuxThreads PTHREAD_COND(3)

2.2 Mise en place des conditions dans mthread

Le code ajouté dans mthread doit être abondamment commenté!!!

Question 2.1: Mettre en place la fonction mthread_cond_init.

Question 2.2: Mettre en place la fonction mthread_cond_wait.

Question 2.3: Mettre en place la fonction mthread_cond_signal.

Question 2.4: Mettre en place la fonction mthread_cond_broadcast.

Question 2.5: Mettre en place la fonction mthread_cond_destroy.

2.3 Démonstration

Question 2.6: Pour chacune des fonctions précédentes, construire un progamme d'exemple qui teste leur bon fonctionnement.

2.4 Bonus les clés posix

Question 2.7: Mettre en place la fonction mthread_key_create.

Question 2.8: Mettre en place la fonction mthread_key_delete.

Question 2.9: Mettre en place la fonction mthread_setspecific.

Question 2.10: Mettre en place la fonction mthread_getspecific.

Question 2.11: Pour chacune des quatres fonctions précédentes, construire un progamme d'exemple qui teste leur bon fonctionnement.