TD2: Les mutex

Marc Pérache

13 mars 2009

A rendre pour le 20 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 Découverte du code de la bibliothèque MThread

Question 2.1: Localiser l'ordonnanceur dans le code de mthread.

Question 2.2: Décrire le fonctionnement de l'ordonnanceur.

Question 2.3: Localiser les fonctions de manipulation des listes dans le code de mthread.

Question 2.4: Décrire comment insérer un thread dans la liste des threads prêts de l'ordonnanceur.

Question 2.5: Décrire comment bloquer un thread.

3 Mutex

3.1 Exemple

```
#include <stdio.h>
#include <pthread.h>
int i = 0;

pthread_mutex_t mutex;

void* run(void* arg){
  pthread_mutex_lock(&mutex);
  i++;
  pthread_mutex_unlock(&mutex);
  return NULL;
}

int main(int argc, char** argv){
  int j;
  pthread_t pids[10];

  for(j = 0; j < 10; j++){
    pthread_create(&(pids[j]),NULL,run,NULL);
  }

  for(j = 0; j < 10; j++){
    pthread_join(pids[j],NULL);
  }

  printf("%d\n",i);
}
```

Question 3.1: Décrire à l'aide d'un schéma l'utilité du mutex dans l'exemple précédent.

3.2 Sémantique POSIX du mutex

```
PTHREAD_MUTEX(3)                                                PTHREAD_MUTEX(3)


NAME
       pthread_mutex_init,     pthread_mutex_lock,     pthread_mutex_trylock,
       pthread_mutex_unlock, pthread_mutex_destroy - operations on mutexes


SYNOPSIS
       #include <pthread.h>

       pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

       int  pthread_mutex_init(pthread_mutex_t  *mutex,  const pthread_mutex-
       attr_t *mutexattr);

       int pthread_mutex_lock(pthread_mutex_t *mutex);

       int pthread_mutex_trylock(pthread_mutex_t *mutex);

       int pthread_mutex_unlock(pthread_mutex_t *mutex);

       int pthread_mutex_destroy(pthread_mutex_t *mutex);


DESCRIPTION
       A mutex is a MUTual EXclusion device, and  is  useful  for  protecting
       shared data structures from concurrent modifications, and implementing
       critical sections and monitors.

       A mutex has two possible states: unlocked (not owned by  any  thread),
       and  locked  (owned  by one thread). A mutex can never be owned by two
       different threads simultaneously. A thread attempting to lock a  mutex
       that is already locked by another thread is suspended until the owning
       thread unlocks the mutex first.

       pthread_mutex_init initializes the mutex object pointed  to  by  mutex
       according  to  the mutex attributes specified in mutexattr.  If mutex-
       attr is NULL, default attributes are used instead.

       Variables  of type pthread_mutex_t can also be initialized statically,
       using the  constants  PTHREAD_MUTEX_INITIALIZER.

       pthread_mutex_lock  locks  the  given mutex. If the mutex is currently
       unlocked, it becomes locked and  owned  by  the  calling  thread,  and
       pthread_mutex_lock returns immediately. If the mutex is already locked
       by another thread,  pthread_mutex_lock  suspends  the  calling  thread
       until the mutex is unlocked.
```

pthread_mutex_trylock behaves identically to pthread_mutex_lock, except that it does not block the calling thread if the mutex is already locked by another thread (or by the calling thread in the case of a ''fast'' mutex). Instead, pthread_mutex_trylock returns immediately with the error code EBUSY.

pthread_mutex_unlock unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to pthread_mutex_unlock.

pthread_mutex_destroy destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance. In the LinuxThreads implementation, no resources are associated with mutex objects, thus pthread_mutex_destroy actually does nothing except checking that the mutex is unlocked.

RETURN VALUE

pthread_mutex_init always returns 0. The other mutex functions return 0 on success and a non-zero error code on error.

ERRORS

The pthread_mutex_lock function returns the following error code on error:

EINVAL the mutex has not been properly initialized.

The pthread_mutex_trylock function returns the following error codes on error:

EBUSY the mutex could not be acquired because it was currently locked.

EINVAL the mutex has not been properly initialized.

The pthread_mutex_unlock function returns the following error code on error:

EINVAL the mutex has not been properly initialized.

EPERM the calling thread does not own the mutex (''error checking'' mutexes only).

The pthread_mutex_destroy function returns the following error code on error:

EBUSY the mutex is currently locked.

```
EXAMPLE
       A shared global variable x can be protected by a mutex as follows:

              int x;
              pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

       All accesses and modifications to x should be bracketed  by  calls  to
       pthread_mutex_lock and pthread_mutex_unlock as follows:

              pthread_mutex_lock(&mut);
              /* operate on x */
              pthread_mutex_unlock(&mut);

                                    LinuxThreads                PTHREAD_MUTEX(3)
```

3.3 Mise en place des mutex dans mthread

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

Question 3.2: Mettre en place la fonction mthread_mutex_init.

Question 3.3: Mettre en place la fonction mthread_mutex_lock.

Question 3.4: Mettre en place la fonction mthread_mutex_unlock.

Question 3.5: Mettre en place la fonction mthread_mutex_destroy.

3.4 Démonstration

Question 3.6: Pour chacune des fonctions précédentes, construire un progamme d'exemple qui teste leur bon fonctionnement.

3.5 Bonus

Question 3.7: Mettre en place la fonction mthread_mutex_trylock.

Question 3.8: Mettre en place PTHREAD_MUTEX_INITIALIZER.

Question 3.9: Pour chacune des deux fonctions précédentes, construire un progamme d'exemple qui teste leur bon fonctionnement.