TD3: Les sémaphores

Marc Pérache

20 mars 2009

A rendre pour le 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 Sémaphore

2.1 Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* affichage (void* name) {
    int i, j;
    for(i = 0; i < 20; i++) {
        sem_wait(&mutex); /* prologue */
        for(j=0; j<5; j++) printf("%s ",(char*)name);
        sched_yield(); /* pour etre sur d'avoir des problemes */
        for(j=0; j<5; j++) printf("%s ",(char*)name);
        printf("\n ");
        sem_post(&mutex); /* epilogue */
    }
    return NULL;
    }

int main (void) {
    pthread_t filsA, filsB;

    sem_init(&mutex, 0, 1);

    if (pthread_create(&filsA, NULL, affichage, "AA")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_create(&filsB, NULL, affichage, "BB")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }

    if (pthread_join(filsA, NULL))
        perror("pthread_join");

    if (pthread_join(filsB, NULL))
        perror("pthread_join");

    printf("Fin du pere\n") ;
```

```
    return (EXIT_SUCCESS);
    }
```

Question 2.1: Décrire à l'aide d'un schéma l'utilité du semaphore dans l'exemple précédent.

2.2 Sémantique POSIX du sémaphore

```
SEMAPHORES(3)                                                SEMAPHORES(3)

NAME
       sem_init, sem_wait, sem_trywait, sem_post, sem_getvalue, sem_destroy -
       operations on semaphores


SYNOPSIS
       #include <semaphore.h>

       int sem_init(sem_t *sem, int pshared, unsigned int value);

       int sem_wait(sem_t * sem);

       int sem_trywait(sem_t * sem);

       int sem_post(sem_t * sem);

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

       int sem_destroy(sem_t * sem);


DESCRIPTION
       Semaphores are counters for resources shared between threads. The
       basic operations on semaphores are: increment the counter atomically,
       and wait until the counter is non-null and decrement it atomically.

       sem_init initializes the semaphore object pointed to by sem. The
       count associated with the semaphore is set initially to value. The
       pshared argument indicates whether the semaphore is local to the cur-
       rent process ( pshared is zero) or is to be shared between several
       processes ( pshared is not zero). LinuxThreads currently does not sup-
       port process-shared semaphores, thus sem_init always returns with
       error ENOSYS if pshared is not zero.

       sem_wait suspends the calling thread until the semaphore pointed to by
       sem has non-zero count. It then atomically decreases the semaphore
       count.

       sem_trywait is a non-blocking variant of sem_wait. If the semaphore
       pointed to by sem has non-zero count, the count is atomically
       decreased and sem_trywait immediately returns 0. If the semaphore
       count is zero, sem_trywait immediately returns with error EAGAIN.
```

sem_post atomically increases the count of the semaphore pointed to by sem. This function never blocks and can safely be used in asynchronous signal handlers.

sem_getvalue stores in the location pointed to by sval the current count of the semaphore sem.

sem_destroy destroys a semaphore object, freeing the resources it might hold. No threads should be waiting on the semaphore at the time sem_destroy is called. In the LinuxThreads implementation, no resources are associated with semaphore objects, thus sem_destroy actually does nothing except checking that no thread is waiting on the semaphore.

RETURN VALUE

The sem_wait and sem_getvalue functions always return 0. All other semaphore functions return 0 on success and -1 on error, in addition to writing an error code in errno.

ERRORS

The sem_init function sets errno to the following codes on error:

EINVAL value exceeds the maximal counter value SEM_VALUE_MAX

ENOSYS pshared is not zero

The sem_trywait function sets errno to the following error code on error:

EAGAIN the semaphore count is currently 0

The sem_post function sets errno to the following error code on error:

ERANGE after incrementation, the semaphore value would exceed SEM_VALUE_MAX (the semaphore count is left unchanged in this case)

The sem_destroy function sets errno to the following error code on error:

EBUSY some threads are currently blocked waiting on the semaphore.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

```
pthread_mutex_init(3),     pthread_cond_init(3),    pthread_cancel(3),
ipc(5).

                                LinuxThreads                    SEMAPHORES(3)
```

2.3 Mise en place des sémaphores dans mthread

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

Question 2.2: Mettre en place la fonction mthread_sem_init.

Question 2.3: Mettre en place la fonction mthread_sem_wait.

Question 2.4: Mettre en place la fonction mthread_sem_post.

Question 2.5: Mettre en place la fonction mthread_sem_destroy.

2.4 Démonstration

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

2.5 Bonus

Question 2.7: Mettre en place la fonction mthread_sem_trywait.

Question 2.8: Mettre en place la fonction mthread_sem_getvalue.

Question 2.9: Pour chacune des deux fonctions précédentes, construire un progamme d'exemple qui teste leur bon fonctionnement.