Multithreading sous Unix : les threads POSIX

TP de Systèmes d'Exploitation

Licence/Master Informatique

22 Novembre 2004

L'objectif[1] de cette séance est de vous familiariser avec l'utilisation des *threads* (en français, *processus légers*) sous Linux au travers de petits programmes d'exemple. L'accent sera mis sur l'interface standard *POSIX Threads* définissant un ensemble de fonctionnnalités supportées par de nombreux systèmes Unix contemporains, et en l'occurence par GNU/Linux. Il est rappelé que les *threads* de GNU/Linux sont implantés au niveau du noyau.

Partie I. L'incontournable *Hello world !*

La figure 1 présente un programme (`simple.c`) créant deux *threads* concurrents affichant chacun un certain nombre de fois une ligne sur le terminal.

Récupérez tous les fichiers sous `~namyst/etudiants/threads/` et compilez le programme simple :

```
make simple
```

(Remarquez l'utilisation de l'option de compilation `-D_REENTRANT`.)

Action I.1. *Lancez le programme et vérifiez qu'il a bien le comportement attendu. Remontez l'instruction* `sleep (1)` *de façon à la placer avant* `pthread_sem_V()`*. Qu'observez-vous à présent ? Expliquez.*

Action I.2. *Que se passe-t-il si vous supprimez l'instruction* `sleep()` *? Comment l'expliquez-vous ? Vérifiez en remplaçant feu l'appel à* `sleep()` *par une boucle d'attente active (compte tenu de la puissance du processeur, comptez plusieurs millions d'itérations !). Que constatez-vous ?*

Action I.3. *Passez le* `int i` *de la fonction* `thread_func` *en* `static int i`*. Que constatez-vous ? Expliquez !*

Partie II. *Threads* et synchronisation

Action II.1. *Mettez les instructions* `pthread_join` *en commentaire dans la fonction* `main`*. Que se passe-t-il ? Qu'en concluez-vous sur le rôle de cette primitive ? Vérifiez avec* `man pthread_join`*.*

Action II.2. *Insérez la ligne :*

```
pthread_detach (pthread_self ());
```

au début de la fonction `thread_func`*. Les threads créés deviennent ainsi "indépendants" et la primitive* `pthread_join` *n'est plus utilisable.*

En remplaçement, utilisez un sémaphore global `sem_end` *pour forcer le* `main` *à attendre la fin des deux threads.*

[1] Un second objectif caché est de faire une petite pause par rapport aux devoirs Nachos...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "pthread_sem.h"

#define NB_ITERATIONS 5

pthread_sem_t mutex;


void * thread_func (void * arg) {
  char * name;
  int i;

  name = (char*) arg;
  for (i = 0; i < NB_ITERATIONS; i++) {
    pthread_sem_P (&mutex);
    printf ("hello world ! (%s)\n", name);
    pthread_sem_V (&mutex);
    sleep (i);
  }
  return NULL;
}


int main (void) {
  pthread_t pid1, pid2;

  pthread_sem_init (&mutex, 1);

  pthread_create (&pid1, NULL, thread_func, "thread 1");
  pthread_create (&pid2, NULL, thread_func, "thread 2");

  pthread_join (pid1, NULL);
  pthread_join (pid2, NULL);

  return EXIT_FAILURE;
}
```

FIG. 1 – Un premier programme : simple.c

Partie III. Le problême des producteurs/consommateurs

Action III.1. *Examinez soigneusement le programme principal* `prod_cons.c` *et le module* `buffer.c`*. En l'état, le programme* `prod_cons` *a-t-il une chance de fonctionner correctement ? Essayez pour voir si vous avez de la chance aujourd'hui...*

Action III.2. *Dans* `buffer.c`*, ajoutez la synchronisation nécessaire dans les fonctions* `put_to_buffer` *et* `get_from_buffer` *afin de bloquer le thread appelant respectivement lorsque le tampon est plein et lorsque le tampon est vide (cf cours).*

Vérifiez le bon focntionnement de `prod_cons`*.*

Action III.3. *Modifiez* `prod_cons.c` *de façon à créer un second thread producteur. Observez l'entrelacement des écritures résultantes. Corrigez* `buffer.c` *de façon à empêcher l'entrelacement des caractères. provenant de différents threads lors de l'appel à* `put_string_to_buffer`*.*

Vous pouvez maintenant comparer le comportement de votre implémentation Nachos des threads et celui de l'implémentation standard POSIX avec des exemples simples !