

Théorie et pratique de la concurrence – Master 1 Informatique

TP 3 : Programmation concurrente en C

Un *thread* est un processus léger, c'est-à-dire un processus qui a ses propres compteurs de programme et pile d'exécution, mais aucun des données "lourdes" associées en général avec un processus (par exemple les tables des pages mémoire).

L'interface de la bibliothèque `Pthreads` a été proposée dans les années 1990 comme un standard POSIX (d'où le préfixe P) pour les routines de programmation multi-*thread* en C. Différentes implémentations de ce standard sont actuellement disponibles sur les systèmes de type UNIX. Cette bibliothèque contient une douzaine de fonctions pour le contrôle des *threads* et pour la synchronisation.

Création et terminaison de *threads*

Pour utiliser la bibliothèque `Pthreads` dans un programme C il faut, premièrement, inclure son fichier de déclarations standard :

```
#include <pthread.h>
```

Deuxièmement, il faut déclarer un ou plusieurs descripteurs de *threads* comme suit :

```
pthread_t pid, cid; /* descripteurs de thread */
```

Finalement, on crée les *threads* comme suit :

```
pthread_create(&pid, NULL, start_func, arg);
```

Le premier argument est l'adresse d'un descripteur de *thread*, le deuxième argument (que nous mettrons toujours à `NULL` dans ces TP) est un attribut de thread de type `pthread_attr_t *` (ces attributs servent par exemple à définir la taille de la pile ou encore à imposer une politique d'ordonnement), le troisième argument est le nom de la fonction à exécuter au lancement du thread, il doit être de type `void *startfunc(void *arg)` et le quatrième argument correspond à l'argument passé à la fonction `startfunc` au moment de l'appel, il doit être de type `void *` (cet argument peut aussi être `NULL`). Si la création est effectuée correctement, la fonction renvoie 0, sinon un code d'erreur. Au moment de l'appel à la fonction `pthread_create`, le *thread* correspondant est lancé et peut exécuter la fonction `start_func`.

Un *thread* se termine en appelant :

```
pthread_exit(value);
```

où l'argument est de type `void *` et peut être égal à `NULL`.

Un processus parent peut attendre la terminaison d'un fils en effectuant :

```
pthread_join(pid, value_ptr);
```

où le deuxième argument est de type `void **` et correspond à une adresse pour le stockage de la valeur de retour du *thread* (avec `pthread_exit`).

Sémaphores

Les *threads* peuvent communiquer entre eux par les variables globales. Ils peuvent se synchroniser par attente active, sémaphores, verrous et variables conditionnelles. On considère ici que les sémaphores.

Pour utiliser les sémaphores il faut, premièrement, inclure le fichier de déclarations standard :

```
#include <semaphore.h>
```

Deuxièmement, déclarer un descripteur de sémaphore global aux fonctions exécutées par les *threads* qui l'utiliseront :

```
sem_t sema;
```

Troisièmement, il faut initialiser les descripteurs comme suit :

```
sem_init(&sema, 0, 1);
```

où le troisième argument correspond à la valeur initiale du compteur du sémaphore (le deuxième argument mis à 0 signifie que le sémaphore peut-être partagé par tous les threads du processus).

Finalement, les opérations P et V sont implémentées par les fonctions `sem_wait` respectivement `sem_post`. Donc, une section critique peut être implémentée comme suit :

```
sem_wait(&sema);          /* P(sema) */
/* section critique */
sem_post(&sema);         /* V(sema) */
```

D'autres fonctions sont disponibles pour les sémaphores, consultez le **man**.

IMPORTANT : Pour compiler vos programmes, n'oubliez pas de mettre l'option `-lpthread`

Exercices

Exercice 1: Programmer en C le problème du producteur-consommateur avec un tampon unitaire.

- Le producteur produit une suite d'entiers terminée par 0.
- Le consommateur calcule et renvoie au parent la somme des entiers consommés.
- Le processus parent attend la terminaison de ses fils et affiche la somme calculée.

Pour cet exercice, le producteur et le consommateur communiqueront à l'aide d'un tampon unitaire dont l'accès se fera en exclusion mutuelle.

Exercice 2:

Diffusion atomique

Soit un système avec un processus producteur et n processus consommateurs qui communiquent via un tampon ayant b cases. Le producteur dépose des messages dans ce tampon et les consommateurs les prennent. Chaque message déposé par le producteur doit être pris par *tous les* n consommateurs. En plus, chaque consommateur doit prendre les messages dans l'ordre dans lequel ils ont été déposés. Toutefois, des consommateurs différents peuvent prendre les messages à des moments différents. Par exemple, un consommateur peut prendre jusqu'à b messages avant un autre, si ce dernier est lent.

Programmez en C une solution à ce problème en utilisant les sémaphores.

Exercice 3:

Salle de bain unisexe

Supposez qu'on dispose d'une seule salle de bain qui peut être utilisée par des hommes ou des femmes mais pas au même moment. Programmez une solution à ce problème. Il faut permettre un nombre quelconque d'hommes OU de femmes être dans la salle de bain au même moment. La solution doit assurer l'exclusion mutuelle demandée et l'absence d'inter-blocage. Modifiez votre solution pour permettre au plus 4 personnes (du même sexe) en même temps.