M2 FEADéP – Systèmes et réseaux
TP1 Sémaphores

Nicolas Chappe
École Normale Supérieure de Lyon

1 Introduction

Les sémaphores POSIX sont définis dans l'en-tête `<semaphore.h>`. Ce mécanisme de synchronisation permet de compter en quelle quantité une ressource partagée est disponible. En pratique, un sémaphore est un entier positif qui propose deux opérations :

— Incrémenter le sémaphore avec `int sem_post(sem_t *sem)`

— Attendre que le sémaphore ait une valeur non nulle puis le décrémenter à l'aide de `int sem_wait(sem_t *sem)`. Le thread est bloqué jusqu'à ce que la décrémentation soit possible. Les variantes non bloquantes `sem_trywait` et `sem_timedwait` peuvent aussi être utiles mais elles ne sont pas au programme.

Pour créer un sémaphore POSIX, on utilise la fonction `int sem_init(sem_t *sem, int pshared, unsigned int value)`, dont le second argument ne nous intéresse pas et le troisième argument est la valeur initiale du sémaphore. Pour détruire un sémaphore, on utilise la fonction `int sem_destroy(sem_t *sem)`.

Exercice 1 (à l'oral)

Comment implanter un mutex à partir d'un sémaphore ?

2 Mini-serveur

Le fichier à trou `miniserveur.c` contient le code d'un mini-serveur qui attend des requêtes (ici factices) et les traite séquentiellement. Mais il a un gros défaut : en l'absence de concurrence il ne peut traiter qu'une requête à la fois.

Exercice 2

Modifier le code pour qu'un nouveau thread soit créé via `pthread_create` pour traiter chaque requête. Il n'est pas nécessaire de modifier le code des fonctions `attendreClient` et `executerRequete`.

Note : on ne se préoccupera pas de la destruction des threads.

Exercice 3

On souhaite maintenant limiter le nombre de connexions simultanées au serveur, afin d'éviter de le surcharger si beaucoup de requêtes arrivent en même temps. Utiliser un sémaphore pour limiter le nombre de requêtes simultanées à 4. Encore une fois, il n'est pas nécessaire de modifier le code des fonctions `attendreClient` et `executerRequete`.

Exercice 4

Selon le système d'exploitation, la création/destruction d'un thread à chaque requête peut être coûteuse. Notre nouvel objectif est donc de créer nos 4 threads au tout début du programme, puis de communiquer les requêtes entrantes au premier thread qui n'a pas déjà une requête en cours de traitement. À cet effet, vous pourrez définir les variables globales suivantes :

```
sem_t semRequete; // sémaphore binaire qui permet aux threads d'attendre que le thread
    principal leur envoie une requête
sem_t semTraitement; // sémaphore binaire qui permet au thread principal d'attendre qu'un
    thread de travail ait pris en compte la dernière requête en attente
requete *reqAttente; // requête en attente de traitement par un des threads de travail
```

Exercice 5 (bonus)

Quand nos 4 threads sont occupés, le thread principal est bloqué et les éventuelles nouvelles requêtes sont ignorées. Pour corriger ce problème, il est possible de maintenir une file de requêtes, qui peut prendre la forme d'une liste simplement chaînée ou d'un tableau, au lieu de conserver une unique requête en attente. Le thread principal ajoute les requêtes entrantes en fin de file, et les threads de travail consomment les requêtes en début de file. C'est le principe du problème *producteur-consommateur*.

Implanter cette nouvelle version.