Systèmes d'exploitation

TP 6 - Sémaphores

2023-05-25

Ce TP est adapté d'un exercice d'Operating Systems: Three Easy Pieces, lui même tiré de l'ouvrage The Little Book of Semaphores, par Allen B. Downey.

Dans ce devoir, nous utiliserons les sémaphores pour résoudre quelques problèmes de concurrence bien connus. Chacune des questions suivantes fournit un squelette de code ; votre tâche consiste à compléter le code pour le faire fonctionner avec des sémaphores. Il est recommandé d'**utiliser Linux** pour réaliser ce TP. Sur Mac (où il n'y a pas de support pour les sémaphores), vous devrez d'abord construire une implémentation de sémaphore (en utilisant des verrous et des *condition variables*, comme décrit dans le cours).

Documentation des sémaphores

Référez-vous aux *man pages* pour savoir comment utiliser un sémaphore.

1. Le premier problème consiste simplement à mettre en œuvre et à tester une solution au problème de **join** telle que décrite dans le cours. Bien que cette solution soit décrite dans le cours, le fait de l'implémenter et de la tester soi-même vous permettra de vérifier que vous l'avez bien comprise.

   Pour rappel, le problème que l'on souhaite résoudre est le suivant : un thread parent crée un thread enfant, il souhaite il souhaite ensuite que le thread enfant ait terminé son exécution pour poursuivre. Lorsque ce programme s'exécute, on souhaiterait avoir l'affichage suivant sur l'écran :

   ```
   parent: begin
   child
   parent: end
   ```

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

sem_t s;

void *child(void *arg) {
```

```
    printf("child\n");
    // use semaphore here
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init semaphore here
    pthread_create(&p, NULL, child, NULL);
    // use semaphore here
    printf("parent: end\n");
    return EXIT_SUCCESS;
}
```

2. Généralisons un peu ce principe en examinant le problème du **rendez-vous**. Le problème est le suivant : vous avez deux threads, chacun d'entre eux étant sur le point d'entrer au point de rendez-vous dans le code. Aucun ne doit quitter cette partie du code avant que l'autre n'y entre.

   Si cela est fait correctement, chaque enfant devrait imprimer son message "before" avant que l'autre n'affiche son message "after". Testez en ajoutant des appels `sleep(3)` à différents endroits.

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

sem_t s1, s2;

void *child_1(void *arg) {
    printf("child 1: before\n");
    // what goes here?
    printf("child 1: after\n");
    return NULL;
}

void *child_2(void *arg) {
    printf("child 2: before\n");
    // what goes here?
    printf("child 2: after\n");
    return NULL;
}

int main(int argc, char *argv[]) {
```

```
    pthread_t p1, p2;
    printf("parent: begin\n");
    // init semaphores here
    pthread_create(&p1, NULL, child_1, NULL);
    pthread_create(&p2, NULL, child_2, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("parent: end\n");
    return EXIT_SUCCESS;
}
```

3. Allons maintenant plus loin en mettant en œuvre une solution générale pour la synchronisation avec des **barrières**. Supposons qu'il y ait deux points dans un morceau de code séquentiel, appelés `P1` et `P2`. La mise en place d'une barrière entre `P1` et `P2` garantit que tous les threads exécuteront `P1` avant que l'un d'entre eux n'exécute `P2`. Votre tâche : écrire le code pour implémenter une fonction `barrier()` qui peut être utilisée de cette manière. On peut supposer que vous connaissez `N` (le nombre total de threads dans le programme en cours d'exécution) et que tous les `N` threads essaieront d'entrer dans la barrière. Encore une fois, vous devriez probablement utiliser deux sémaphores pour réaliser la solution, et d'autres nombres entiers pour compter les choses.

   Si cela est fait correctement, chaque enfant devrait imprimer son message "before" avant que l'autre n'affiche son message "after". Testez en ajoutant des appels `sleep(3)` à différents endroits.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct __barrier_t {
    // add semaphores and other information here
} barrier_t;


// the single barrier we are using for this program
barrier_t b;

void barrier_init(barrier_t *b, int num_threads) {
    // initialization code goes here
}

void barrier(barrier_t *b) {
    // barrier code goes here
}
```

```
//
// XXX: don't change below here (just run it!)
//
typedef struct __tinfo_t {
    int thread_id;
} tinfo_t;

void *child(void *arg) {
    tinfo_t *t = (tinfo_t *) arg;
    printf("child %d: before\n", t->thread_id);
    barrier(&b);
    printf("child %d: after\n", t->thread_id);
    return NULL;
}

// run with a single argument indicating the number of
// threads you wish to create (1 or more)
int main(int argc, char *argv[]) {
    assert(argc == 2);
    int num_threads = atoi(argv[1]);
    assert(num_threads > 0);

    pthread_t p[num_threads];
    tinfo_t t[num_threads];

    printf("parent: begin\n");
    barrier_init(&b, num_threads);

    int i;
    for (i = 0; i < num_threads; i++) {
        t[i].thread_id = i;
        pthread_create(&p[i], NULL, child, &t[i]);
    }

    for (i = 0; i < num_threads; i++)
        pthread_join(p[i], NULL);

    printf("parent: end\n");
    return EXIT_SUCCESS;
}
```