

TP - révisions structures de données et gestion de la concurrence

Florian Bourse

Le but de ce TP est d'implémenter l'expérience suivante :

- on initialise un compteur à 0 ;
- on tire au hasard des nombres entiers ;
- s'ils sont pairs, on les ajoute au compteur, sinon on les retire du compteur ;
- ?
- Profit

4 fichiers de code sont fournis :

- counter.h
- counter.c
- main.c
- queue.h

1 Compteur partagé

- 1 – Compiler et exécuter le code contenu dans les fichiers fournis. Quelle est la valeur du compteur à la fin de l'exécution ? Est-ce la valeur attendue ?
- 2 – Quelles sont les différentes valeurs qu'il est possible d'obtenir en exécutant ce code ? Donner une trace d'exécution pour chaque valeur possible.
- 3 – Alice propose d'ajouter un mutex pour éviter la course critique : elle propose la modification suivante du code de la fonction f, m étant déclaré comme variable globale et initialisé dans la fonction main.

```
void *f(void *args){
    pthread_mutex_lock(&m);
    for (int i = 0; i < 100000; i = i + 1)
        counter_add(&ctr, 1);
    pthread_mutex_unlock(&m);

    return NULL;
}
```

Discuter de la pertinence de cette solution.

□ 4 – On souhaite plutôt gérer les courses critiques pour l'utilisation simultanée des compteurs en associant un mutex à chaque compteur, ainsi si deux compteurs sont utilisés, l'utilisation de l'un n'aura aucun impact sur l'utilisation de l'autre. Modifier les fichiers `counter.h` et `counter.c` afin que l'exécution du code de `main.c` affiche toujours 200000.

□ 5 – Modifier le code de la fonction `f` pour implémenter l'expérience citée en entrée.

On rappelle que la fonction `rand() % n` renvoie un nombre aléatoire entre 0 inclu et n exclu.

2 Implémentation de file dans un tableau circulaire

Nous souhaitons modifier notre implémentation pour spécialiser les fils d'exécution en suivant le modèle des producteurs et consommateurs. Les producteurs sont des fils d'exécution qui génère des nombres aléatoires. Les consommateurs sont des fils d'exécution qui les ajoute au compteur.

Nous commençons avec un producteur et un consommateur qui communiquent via une file.

Pour implémenter une file d'entiers dans un tableau circulaire, nous avons besoin de retenir, en plus du tableau contenant les données :

- la taille du tableau, pour éviter un dépassement ;
- l'indice de la tête de la file, pour savoir quel est le prochain élément à défiler ;
- la taille de la file, pour savoir si on peut défiler ou enfiler.

Afin d'autoriser une implémentation des files par tableau ou par maillons chaînés, nous ne manipulerons les files que via des pointeurs. Une description de l'interface attendue est décrite dans le fichier `queue.h`.

□ 6 – Connaissant la taille du tableau, la taille de la file, et l'indice de la tête de la file, quel est l'indice de la queue de la file ?

□ 7 – Créer un fichier `queue.c` contenant une implémentation de la structure de file **struct** `queue` via des tableaux circulaires, de taille fixée à 1024. On ajoutera dans l'entête du fichier **#include** `"queue.h"`.

□ 8 – Compléter les fonctions nécessaires pour pouvoir compiler le fichier `queue.c` avec la commande

```
gcc -c queue.c
```

On utilisera des assertions (disponibles après avoir inclu `<assert.h>`) pour vérifier que l'on essaye pas de défiler une file vide ou d'enfiler dans une file pleine (contenant 1024 éléments).

□ 9 – Pour tester les files, nous utiliserons l'expérience suivante : un producteur enfle les nombres de 0 à 99999, alors qu'un consommateur défile les nombres et les affiche. Que peut-on observer ?

Lors d'une utilisation concurrente par deux fils d'exécution A et B , l'un enfilant et l'autre défilant des éléments, il est possible que le fil d'exécution A essaye de défiler un élément d'une file vide alors que l'autre fil d'exécution B va enfiler un élément. On souhaite alors que le fil A attende que B ait enfilé un élément plutôt que d'arrêter l'exécution du programme.

De même, si B veut enfiler un élément dans une file pleine, il peut attendre que le fil A ait défilé un élément plutôt que d'arrêter l'exécution du programme.

Ce problème peut être résolu en utilisant des sémaphores.

□ 10 – Ajouter deux sémaphores et un mutex à votre structure de pile, ainsi que le code correspondant à leur initialisation dans la fonction `queue_init`.

□ 11 – Modifier les fonctions `queue_push` et `queue_pop` pour remplacer les assertions par l'utilisation des sémaphores permettant d'attendre respectivement que la file ne soit pas pleine et que la file ne soit pas vide.

Vérifier votre implémentation avec le test précédent.

□ 12 – Nous pouvons à présent lancer notre expérience :

Nous utiliserons 1 compteur partagé, et 2 fils : une pour les nombres pairs, et une pour les nombres impairs.

2 producteurs génèrent des nombres aléatoires et les ajoutent à la file correspondant à leur parité.

1 consommateur défile les nombres pairs et les ajoute au compteur, 1 autre consommateur défile les nombres impairs et les soustrait du compteur.

Le fil principal d'exécution affiche régulièrement la valeur du compteur.

On pourra utiliser la fonction `sleep` disponible dans `<unistd.h>`.

3 Implémentation de file avec des maillons chaînés

Nous souhaitons à présent remplacer notre implémentation des files, utilisant des tableaux circulaires, par une implémentation utilisant des maillons chaînés.

□ 13 – Créer un autre fichier qui implémente l'interface de `queue.h`, mais en utilisant une structure de maillons chaînés. On utilisera toujours un mutex, mais cette fois-ci uniquement un seul sémaphore, comptant le nombre d'éléments dans la file, la taille de la file n'était pas bornée.

□ 14 – Si les producteurs sont plus efficaces que les consommateurs, la taille de la file peut se retrouver trop grande. Proposer en implémenter une solution pour ce problème.

□ 15 – Nous souhaitons à présent gérer les courses critiques au niveau du maillon plutôt qu'au niveau de la file. Proposer une solution de gestion de la concurrence utilisant un mutex par maillon plutôt qu'un mutex pour la file entière. Discuter de la pertinence d'une telle solution.

A Rappels de programmation en C

A.1 Mutex

Exemple d'utilisation de mutex C

```
#include <pthread.h>

/* corps de fonction */
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
pthread_mutex_lock(&m);
/* Section critique */
pthread_mutex_unlock(&m);
```

A.2 Sémaphores

Exemple d'utilisation de sémaphores C

```
#include <semaphore.h>

/* Variables globales */
sem_t sA;
sem_t sB;

/* Code de A */
sem_wait(&sA);
sem_post(&sB);

/* Code de B */
sem_wait(&sB);
sem_post(&sA);

/* Code de main */
sem_init(&sA, 0, 1);
sem_init(&sB, 0, 0);
```