TP 09 : Programmation

1 Mandelbrot

Soit c un nombre complexe. On considère la série $z_0 := 0$ et $z_{n+1} := z_n^2 + c$ pour $n \geq 0$. L'*ensemble de Mandelbrot* est défini comme l'ensemble des valeurs c telles que la série des z_n est bornée. On sait que cela est le cas si z_n ne sort jamais d'un cercle de radius 2 autour de 0. Si jamais la série sort de ce cercle, soit $m(c)$ le plus petit indice n tel que c'est le cas.

Une application populaire pour $m(c)$ est de créer de jolies images ; on associe l'écran avec un rectangle et chaque pixel avec la valeur c qui y correspond ; le pixel est ensuite peint avec une couleur correspondant à $m(c)$.

La page web du cours contient un tel programme. Votre tâche consiste à l'accélérer en lançant plusieurs threads en parallèle. Les machines de la salle 411 sont équipées de plusieurs cœurs (utilisez `cat /proc/cpuinfo` pour trouver plus d'information). Chaque thread va donc travailler sur une partie différente de l'image.

2 Exclusion mutuelle

Dans cet exercice nous illustrerons les problèmes d'accès concurent à une variable et nous nous initierons a la programmation des threads.

Cette fois, pas de code source de départ, le sujet vous aidera un peu mais vous écrirez votre code "from scratch" (ceci est une indication subliminale).

1. Créez un code source composé :
   — d'une constante x supérieure à 10 000 ;
   — d'une fonction `process0` incrémentant x fois une variable globale `cpt` ;
   — d'un corps de programme instanciant deux threads, attendant le résultat de ces executions et l'affichant sur la sortie standard.

   Avant de commencer à écrire votre code, quel est le résultat attendu ? Est-ce le résultat obtenu ? Pourquoi ?

2. Algorithme de Peterson.

   Nous proposons de solutionner le problème précédent en implémentant l'algorithme d'exclusion mutuelle de Peterson décrit sur Wikipedia. Le principe consiste à utiliser 3 variables `f0`, `f1` et `turn` pour gérer l'entrée dans la «section critique» du code. La «section critique» est la section contenant le code à protéger des accès concurents.

   Prouvez que algorithme assure l'exclusion mutuelle.

3. En vous basant sur la page Wikipedia de cet algorithme :
   — modifiez la fonction `process0` et créez la fonction `process1` implémentant l'algorithme de Gary L. Peterson ;
   Note : `process0` et `process1` impléméntent la même fonctionnalité mais pour les threads 0 et 1.
   — Testez votre code et commentez.

4. En vous basant sur le code de `process0` et `process1` écrire la fonction `process` qui permettra l'invocation générique d'un thread.
5. Toujours en vous basant sur la page Wikipedia de l'algorithme de Peterson, modifiez votre code pour 4 processus.
6. Mêmes questions que précédemment pour l'algorithme de Dekker.

3 Sémaphores

Les *sémaphores* représentent une solution pour résoudre le problème des accès concurrents aux variables. Un sémaphore est une strucutre de données utilisée pour la synchronisation et pour assurer un accès coordonné aux ressources partagés. En C, les sémaphores sont définis par le fichier `semaphore.h`. Les fonctions les plus importantes pour nos objectifs seront les suivantes :

— `int sem_init(sem_t *sem, int pshared, unsigned int value);`
Initialise `sem` avec la valeur donnée ce qui donne le nombre de créneaux disponibles pour accéder à une ressource au même temps. Pour nos objectifs, `pshared` devrait être `NULL`.

— `int sem_wait(sem_t *sem);`
Si le sémaphore a la valeur 0 cette fonction attend qu'un créneau devienne disponible. Sinon, elle décroît sa valeur immédiatement.

— `int sem_post(sem_t *sem);`
Augmente la valeur de la sémaphore (un créneau devient disponible).

étudiez les pages `man` (ou rechercher sur google) pour en savoir plus.

(a) Utilisez les sémaphores pour remplacer l'algorithme de Peterson et réparer le programme `counter.c`.

Enfin, considérons le programme suivant (`order.c`) :

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

void* first(void* data)  { printf("First\n");  }
void* second(void* data) { printf("Second\n"); }
void* third(void* data)  { printf("Third\n");  }

int main () {
  pthread_t t1, t2, t3;

  pthread_create(&t3, NULL, third, NULL);
  pthread_create(&t2, NULL, second, NULL);
  pthread_create(&t1, NULL, first, NULL);

  // wait for all threads
  pthread_join(t1, NULL);  pthread_join(t2, NULL); pthread_join(t3, NULL);
}
```

Modifiez le programme pour assurer que le programme donne toujours la sortie suivant :

```
First
Second
Third
```

(b) Écrivez un programme en C avec deux threads. Le premier thread affiche les entiers pairs jusquà 100, le deuxième les impairs. Utilisez des sémaphores pour assurer le bon ordre des entiers.

4 Producteur/consommateur

Un problème typique dans la programmation concurrente est celui du *producteur et consommateur* : un processus produit des données que l'autre consomme. Dans un tel modèle, le producteur génère des caractères dans un fichier et les envoie vers un espace partagé (un buffer) avec le processus consommateur qui les affiche sur l'écran. Écrire un programme qui implémente se modèle et tel que le consommateur affiche correctement les données (cela demande de faire de la synchronization).

5 Éléction d'un leader

Source pour l'exercice disponible à `www.lsv.fr/~{}fthire/teaching/archisys/TP/9/ring.tar.gz`.

Un problème important dans la programmation concurrente est de choisir un *leader*, p.ex. pour l'organisation et la coordination d'un réseau. On part avec un nombre n d'ordinateurs ou de processus (on parlera des *nœuds*) à priori identiques, à l'exception d'un identifiant unique. En suivant un même protocole, tous les nœuds vont s'accorder sur le choix d'un leader.

Il existe une grande variété de protocoles menant à ce résultat. Nous allons dans ce TP en étudier et en réaliser un : le protocole Dolev, Klawe, and Rodeh qui se distingue par le faible nombre de messages échangé pour l'élection : $\mathcal{O}(n \log n)$. Par comparaison, les approches simples ont tendance à utiliser $\mathcal{O}(n^2)$ messages. Le protocole a été presenté en cours et les diapos explicatives sont aussi disponibles dans le répertoire du TD.

Le protocole part du principe que les nœuds sont organiseés en anneau, chaque nœud envoie des messages à son voisin de droite. Le répertoire du TP contient un squelette qui créera n processus (les nœuds) pour n donné ; ces processus seront déjà équipés de pipes pour communiquer. Votre tâche est de compléter le programme en réalisant le protocole :

- Au départ, tous les nœuds sont *actifs* et possèdent un identifiant unique.
- Un nœud actif attend une petite période aléatoire (fonction `delay()`), puis envoie son identifiant à son voisin de droite. Ensuite, il attend les identifiants de ses *deux* plus proches voisins actifs à sa gauche. Il décidera ensuite s'il reste actif, devient passif, ou se déclare leader. Les conditions seront précisées dans la présentation. S'il reste actif, il répète le comportement présenté ci-dessus.
- Un nœud *passif* transmet simplement tous les messages reçus de gauche vers son voisin de droite. En plus, si le message déclare un leader, il affiche un message correspondant à l'écran.
- Il y a trois types de messages échangés par les nœuds, tous dans le format (*type*,*identifiant*).
  - "voisin" (v) : pour envoyer son identifiant vers le voisin de droite.
  - "prochain" (p) : un nœud qui a reçu l'identifiant de son voisin de gauche l'envoie vers son voisin de droite.
  - "gagnant" (g) : un nœud se déclare leader.

1. Complétez la fonction «protocole» du fichier «ring-pipe.c» qui implémente le protocole sus-décrit.
2. Testez votre code avec plusieurs valeurs de n. Observez si votre programme termine toujours et s'il y a toujours un seul leader.
3. La version réseau du programme, «ring-net.c» instancie un seul nœud par exécution. Vous pouvez recopier votre fonction «protocole» depuis «ring-pipe.c». Le programme prend 3 arguments : le port d'écoute du nœud, le nom du nœud voisin (nom de machine), le port de connexion au nœud voisin. Le programme crée un serveur dans un thread et attend la connexion d'un voisin. En parallèle, dans un autre thread, il tente une connexion sur son voisin (nom de machine et port passés en argument). Écrivez un script mettant en œuvre l'exécution de votre code sur 5 machines du département. Commencez par tester l'exécution de votre code localement (machine localhost en utilisant différents ports). Étendre progressivement la taille de l'anneau.
4. Créez un anneau avec vos voisins et l'étendre à toute la salle de TP.

6 Révision examen

Voir la page du cours de Stefan.