TP - LIFPCA Programmation Concurrente et Administration Système

—

Synchro !

Sylvain Brandel, Yves Caniou, Guillaume Damiand, Meriem Ghali, Laurent Lefèvre, Thibaut Modrzyk, Grégoire Pichon, Alec Sadler, Florence Zara, Jerry Lacmou Zeutouo

Printemps 2024

Avec l'aide du travail que vous avez fourni jusqu'à maintenant, nous allons ici nous intéresser à la mise en place de synchronicité dans l'exécution de tâches.

I Synchronisation simple

Le but ici est d'écrire un programme qui effectue les tâches A_1 et A_2 en parallèle, puis la tâche B. Les tâches A_1 et A_2 exécutent du code arbitraire, par exemple Fibonacci() sur un nombre tiré aléatoirement (ou tout simplement `sleep`;D). La tâche B attend que les tâches A_1 et A_2 soient terminées avant de s'exécuter. Attention, les trois tâches doivent s'exécuter en parallèle : vous ne pouvez pas simplement lancer A_1 et A_2 en parallèle, puis faire un `join` avant de lancer B ! Voici le squelette du `main` de votre programme :

```
1  int main() {
2    for(...) {
3      std::thread A_1(...);
4      std::thread A_2(...);
5      std::thread B(...);
6      A_1.join(); A_2.join(); B.join();
7      }
8    return 0;
9  }
```

— Proposez une solution et discutez-en avec votre encadrant.

— La solution est-elle facilement généralisable au cas n threads ?

II Barrière de synchronisation

— À quel type de problème correspondait l'énoncé précédent ?

Ceux qui feront l'UE Parallélisme l'année prochaine verront des couplages de codes MPI/OpenMP : il existe des primitives qui peuvent être utilisées pour simplifier la programmation

de programmes multi-processus (la gestion des communications à travers les sockets se voit comme "Envoyer cette structure de données au processus X" alors que le processus X fait un "Réception d'une structure de données" : pas besoin des `listen()` ou `fork()` vus en Système d'Exploitation en L2 !) et multi-thread (on laisse le compilateur gérer nos demandes de création de threads exprimées par des `#pragma` ! Par contre, il faut bien comprendre les opérations de réduction [1]).

II.1 Travail à réaliser : la barrière

Nous vous proposons ici de coder une barrière de synchronisation afin de faire fonctionner un code jouet. Le code principal, dans votre `main`, doit demander à l'utilisateur le nombre de threads qui s'exécuteront, ainsi qu'une valeur `nbtours` qui représente le nombre d'itérations de la boucle `for`. (Vous pouvez demander la valeur de `nbtours` interactivement via `cin` ou `scanf`, ou en CLI si vous souhaitez utiliser les paramètres `argc` et `argv` de `main`). Le code principal lance ensuite les threads. Chaque thread exécute le pseudo-algorithme suivant :

```
1  Exécute nbtours fois la séquence
2    Tâche()
3    Barrière()
```

Le pseudo-algorithme de $T\hat{a}che()$ est le suivant :

```
Affiche l'heure
Tire un nombre X aléatoirement dans U[10,20]
Calcule Fibonacci(X)
Calcule la durée passée pendant le calcul et l'affiche
```

Votre travail consiste surtout à designer/coder l'opération `Barrière()`. Vous êtes libre de procéder comme vous le souhaitez, à ceci près que vous n'avez pas le droit d'utiliser la structure `pthread_barrier_t`, ni les classes `std::latch` et `std::barrier` de C++20 (qui sont les classes que nous cherchons à ré-écrire). Vous pouvez si vous le souhaitez arrêter la lecture du sujet ici, et coder la solution sans aide supplémentaire (sachez tout de même qu'une difficulté est qu'il peut y avoir plusieurs appels à Barrière sur le même objet du fait de la boucle autour de cet appel. Il est conseillé de démarrer par une version simplifiée qui ne gère qu'un seul appel). La suite du sujet donne des instructions pas à pas si vous ne savez pas par où commencer (c'est sur la page d'après pour limiter la tentation ;-)).

1. Voir en bas de page de `https://graal.ens-lyon.fr/~ycaniou/Teaching/1415/L3/index.html`

II.2 Barrière non-réutilisable

Dans un premier temps, nous supposerons que chaque thread n'appelle `wait()` qu'une seule fois, et qu'il y a N threads au total (N est une constante à définir dans le code). Pour information, c'est ce que fait la classe `std::latch` de C++20, nous appelerons donc notre classe `latch` par analogie.

Nous utiliserons un compteur initialisé à 0 et qui compte le nombre de threads ayant atteint la barrière (c'est-à-dire ayant démarré l'appel à `wait()`). Les threads sont débloqués quand ce compteur arrive à N.

Q.II.1) - Écrivez un squelette de classe `latch`, qui contient une méthode `wait`. On laissera le corps de la fonction vide pour l'instant.

Q.II.2) - Écrivez une fonction principale (`main()`) qui instancie N threads qui chacun 1) affiche la chaîne `"avant\n"`, 2) accède à la barrière, 3) affiche la chaîne `"apres\n"`, puis termine son exécution.

Q.II.3) - Que devrait afficher l'exécution de la fonction `main()` écrite ci-dessus (avec $N = 3$) ? Vous pourrez vérifier votre prédiction juste après la question suivante.

Q.II.4) - Complétez l'implémentation de la classe `latch` : les champs privés, le corps de la fonction `wait()`, et du constructeur de `latch`.

II.3 Barrière réutilisable

Nous allons maintenant écrire une barrière plus générique, où la fonction `wait()` peut être appelée plusieurs fois d'affilée par chaque thread. Par exemple, chaque thread peut exécuter le code suivant :

```
1  void barrier_infinite(barrier &b) {
2          while (true) {
3                  b.barrier();
4          }
5  }
```

La difficulté est qu'en débloquant les threads (quand N threads ont appelé `wait()`), on peut arriver dans une situation où certains threads démarrent leur 2ème appel à `wait()` alors que d'autres n'ont pas terminé leur 1er appel.

La solution que nous allons utiliser ici est d'avoir un compteur de génération :

- Initialement, le compteur de génération vaut 0.
- Quand un thread fait un appel à `wait()` pendant la génération i, il est bloqué jusqu'à ce que le compteur de génération devienne différent de i.
- Quand un thread fait le Nième appel à `wait()` de la génération i, il incrémente i pour débloquer tous les threads (et lui-même n'est pas bloqué).

On suppose qu'il n'y a jamais d'overflow sur le compteur.

Q.II.5) - Proposez une nouvelle implémentation de la classe barrière (qui cette fois-ci correspond à la classe `std::barrier` de C++20, donc nous pourrons l'appeller `barrier`) en suivant ce principe.

On considère un programme principal qui instancie N threads exécutant chacun le code suivant :

```
1  void barrier_ngen(barrier &b) {
2          for (int i = 0; i < 5; ++i) {
3                  sleep(1); // Rappel : sleep(1) fait une attente d'une seconde.
4                  b.wait();
5          }
6  }
```

On néglige les temps de calcul et on considère que seule la fonction `sleep(1)` prend du temps. Attention, `sleep()` fait une attente passive : d'autres threads peuvent s'exécuter pendant son exécution.

Q.II.6) - Combien de temps mettra le programme à s'exécuter avec $N = 10$?

Q.II.7) - Que se passe-t-il si on exécute le programme sur un système mono-cœur ?

Dans chaque cas, faites d'abord une prédiction, par exemple en dessinant un chronogramme qui montre l'exécution du programme, puis vérifiez votre prédiction expérimentalement.

III Lecteur/rédacteur (si temps)

— Rappelez les conditions de présence d'un problème de type Lecteur/rédacteur.

— Codez un moniteur capable de gérer un tel problème (attention à la propreté du code).