TP 3 – Mécanisme de synchronisation

Note

Finir les TP1 et TP2 avant d'attaquer le TP3. Il faut avoir compris

- l'utilisation des threads avec les mutex ;
- la gestion de l'exclusion mutuelle pour les sections critiques ;
- la mise en place d'un mécanisme de type producteur-consommateur ;
- l'implémentation d'un moniteur de Hoare.

Dans ce TP, nous allons voir la mise en place de synchronicité dans l'exécution de tâches.

1 Synchronisation simple

Le but ici est d'écrire un programme qui effectue les tâches A_1 et A_2 en parallèle, et après la terminaison de ces deux taches, 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 :

```
int main() {
  for(...) {
    std::thread A_1(...);
    std::thread A_2(...);
    std::thread B(...);
    A_1.join(); A_2.join(); B.join();
    }
  return 0;
}
```

Q1. Proposez une solution et discutez-en avec votre encadrant. Votre solution est-elle facilement généralisable au cas n threads ?

Q2. À quel type de problème cela correspond-il ?

2 Barrière de synchronisation

Nous vous proposons ici de coder une **barrière de synchronisation** afin de faire fonctionner un code d'exemple. Le code principal de 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 d'une boucle `for`. Vous pouvez demander la valeur de `nbtours` interactivement via `cin` ou `scanf`, ou en ligne de commande en utilisant les paramètres `argc` et `argv` du `main()`.

Le code principal lance ensuite les threads qui exécuteront le pseudo-algorithme suivant :

```
// Chaque thread execute nbtours fois la sequence :
    Tache()
    Barriere() // objectif du TP
```

Le pseudo-algorithme de $Tache()$ est le suivant :

```
Affiche l'heure
Tire un nombre X aleatoirement dans U[10,20]
Calcul de Fibonacci(X)
Calcul de la duree passee pendant le calcul et l'affiche
```

Votre travail consiste surtout à coder l'opération `Barriere()`. 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).

Note

Si vous le souhaitez, vous pouvez arrêter la lecture du sujet ici afin de coder votre solution sans aide supplémentaire (sachez tout de même qu'une difficulté est qu'il peut y avoir plusieurs appels à Barriere 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).

2.1 Barrière non-réutilisable

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

Pour réaliser ce code, nous utiliserons un compteur initialisé à 0 comptant le nombre de threads ayant atteint la barrière (i.e. ayant démarré l'appel à `wait()`). Les threads seront tous débloqués quand ce compteur arrive à N.

Q1. Écrivez le squelette de classe `latch` comprenant une méthode `wait()` dont le corps restera vide.

Q2. Écrivez le programme principal `main()` instanciant N threads. Chacun des threads :
1) affiche la chaîne `"avant\n"`
2) accède à la barrière
3) affiche la chaîne `"apres\n"`, puis termine son exécution.

Q3. Que devrait afficher l'exécution de la fonction `main()` écrite pour $N = 3$? Vous pourrez vérifier votre prédiction juste après la question suivante.

Q4. Complétez l'implémentation de la classe `latch` en ajoutant les champs privés, le corps de la fonction `wait()`, et celui du constructeur de la classe `latch`.

2.2 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.wait();
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 second appel à `wait()` alors que d'autres n'ont pas terminé leur premier 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).

Q1. Proposez une nouvelle implémentation de la classe barrière que nous nommerons `barrier` puisqu'elle correspond désormais à la classe `std::barrier` de C++20.

Q2. Votre programme principal instanciera N threads exécutants chacun le code suivant :

```
1 void barrier_ngen(barrier &b) {
2         for (int i = 0; i < 5; ++i) {
3                 sleep(1); // 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, sachant que `sleep()` fait une attente passive, c'est-à-dire que d'autres threads peuvent s'exécuter pendant son exécution.

Q3. Combien de temps mettra le programme à s'exécuter avec $N = 10$?

Q4. Que se passe-t-il si on exécute le programme sur un système mono-cœur ?

Pour ces deux questions, faites d'abord une prédiction, par exemple en dessinant un chronogramme qui montre l'exécution du programme, puis vérifiez votre prédiction expérimentalement.