TP - ASR7 Programmation Concurrente

—

Synchro !

Sylvain Brandel, Guillaume Damiand, Laurent Lefèvre, Matthieu Moy, Grégoire Pichon

Printemps 2022

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 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 ?

Beaucoup auront certainement commencé leur code avec une variable protégée par un mutex, initialisée à 2 et décrémentée à la fin de A_1 et de A_2 : B doit alors tester que la valeur soit à 0 pour commencer. Espérons que la plupart se soit rendu compte que cela engendre une attente active sur la valeur 0, et qu'il faut donc une variable de condition afin d'être réveillé quand toutes les tâches A ont terminé.

La solution avec un sémaphore est clairement plus simple : il n'y a rien à faire sauf instancier le sémaphore et appeler `P()` et `V()` (2 fois). Par contre, "difficilement" généralisable pour n threads, surtout si on considère une gestion de petites barrières entre groupes de tâches (déploiement de la solution et maintenance). [1]

Une implémentation simple de la solution avec un mutex, une variable de condition et une variable compteur, consiste à placer ces trois entités dans le *namespace* global. Ce n'est cependant ni une solution propre, ni franchement généralisable (maintenance du code !), en particulier dans le cas où différents groupes de tâches doivent se synchroniser (solution non réutilisable !).
Mieux, l'utilisation d'un sémaphore comme vu en CM et TD.
Une solution plus élégante peut être de coder un sémaphore inversé, qui expose deux méthodes, `release` et `acquire`. Tout comme un sémaphore "classique", le sémaphore inversé possède un compteur dont la valeur initiale est choisie au moment de sa création. `acquire` bloque tant que la valeur du sémaphore n'est pas 0, et `release` décrémente le compteur du sémaphore de 1.

II Barrière de synchronisation

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

On pouvait le voir comme un Producteur-Consommateur : il "suffisait" finalement de reprendre le compteur du cours et de l'aménager un peu. On voit que réfléchir à la solution et aux outils déjà à disposition est une aide pour faire bien, proprement, et plus rapidement ;)

Comme expliqué pendant le cours, une barrière de synchronisation est un objet utilisé dans de nombreux codes, souvent itératifs, où il y a des dépendances de données résultantes des travaux réalisés en parallèle de notre propre calcul (du point du vue du thread). Par exemple, le calcul de l'itération $n+1$ requiert les données calculées d'autres tâches de l'itération n, en plus des nôtres. [2]

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 [3]).

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 :

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

```
Exécute nbtours fois la séquence
  Tâche()
  Barrière()
```

Le pseudo-algorithme de *Tâche*() 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 la classe `std::barrier`. À vous de trouver comment fonctionne une barrière !

Les différents threads d'exécution qui participent à une barrière doivent se partager une même instance de la barrière en question. Une barrière pourrait être utilisée pour mettre en place la synchronisation demandée dans le premier exercice.

```
class barrier {
public:
    barrier(unsigned int nb_threads)  : _nb_threads(nb_threads) { }
    void wait() {
        std::unique_lock<std::mutex> lck(_m);
        ++_waiting;
        while (_waiting != _nb_threads) {
            _cv.wait(lck);
        }

        if (_waiting == _nb_threads) {
            _cv.notify_all();
        }
    }

private:
    std::mutex _m;
    std::condition_variable _cv;
    unsigned int _nb_threads;
    unsigned int _waiting = 0;
};
```

Cette solution présente toutefois la particularité de ne pas être réentrante : il n'est pas possible de se synchroniser plusieurs fois sur la barrière. En effet, la variable `_waiting` n'est pas remise à zéro lorsque l'on quitte la barrière. Réinitialiser cette variable présente toutefois un certain nombre de problèmes. En effet, la condition d'attente est `while(_waiting != _nb_threads)`. Si la valeur de `_waiting` repasse à zéro avant que tous les threads ne soient sortis de la boucle, alors certains vont à nouveau attendre sur la variable de condition `_cv`.

Il faut donc parvenir à réinitialiser la valeur de `_waiting` une fois que tous les threads sont sortis de la barrière. On se retrouve donc dans un cercle vicieux où il nous faudrait une barrière pour attendre que tous les threads soient sortis de la barrière afin de coder la barrière. Un

second problème notable est que l'on ne peut pas permettre à de nouveaux threads d'attendre sur la barrière tant que les threads de la synchronisation précédente ne sont pas sortis de la barrière, autrement la valeur de `_waiting` pourrait augmenter au-delà de `_nb_threads`, ce qui serait faux.

La solution la plus simple consiste à compter combien de threads doivent sortir de la barrière. Lorsque ce nombre atteint zéro, la barrière est réinitialisée. Ce compteur de threads va également nous servir de garde à l'entrée de la fonction de synchronisation.

```
1  class barrier {
2  public:
3    barrier(unsigned int nb_threads)  : _nb_threads(nb_threads) { }
4
5    void wait() {
6      std::unique_lock<std::mutex> lck(_m);
7
8      /* Bloque les threads qui arrivent ici après avoir été débloqués
9       * dans la synchro précédente. La synchro précédente n'est pas
10      * résolue tant que _to_free != 0.
11      */
12     while (_to_free != 0) {
13       _reset.wait(lck);
14     }
15
16     ++_waiting;
17
18     // Distinction
19     if (_waiting != _nb_threads) {
20       /* Bloque pour une synchronisation */
21       while (_waiting != _nb_threads) {
22         _cv.wait(lck);
23       }
24
25       _to_free--;
26       /* Le dernier thread qui sort notify les threads qui ont
27        * pu refaire un wait() avant que la synchro ne soit complètement
28        * résolue.
29        */
30       if (_to_free == 0) {
31         reset.notify_all();
32       }
33     } else {
34       /* Normalement faire un notify ne reschedule pas les threads, donc
35        * inverser les deux lignes suivantes ne change rien. Dans la
36        * pratique, il est préférable de faire les modifs sensibles avant
37        * le notify par acquis de conscience.
38        *
39        * _to_free devient _nb_threads - 1 pour bloquer les threads qui
40        * pourraient revenir dans cette fonction avant que la synchro
41        * courante ne soit résolue.
42        */
43       _waiting = 0;
```

```
44          _to_free = _nb_threads - 1;
45          _cv.notify_all();
46        }
47      }
48
49    private:
50      std::mutex _m;
51      std::condition_variable _cv;
52      std::condition_variable _reset;
53      unsigned int _nb_threads;
54      unsigned int _waiting;
55      unsigned int _to_free = 0;
56    };
```

Notez qu'il est important qu'un seul thread effectue les opérations `_waiting = 0` et `_cv.notify_all()`. Autrement, un thread pourrait à nouveau tenter de se synchroniser sur la barrière, et pourrait être réveillé trop tôt.

Plusieurs instances différentes d'un objet Barrière peuvent être utilisées par différents groupes de tâches qui ont besoin de mettre en place une barrière de synchronisation – par exemple ceux présentés en cours, dans le schéma d'un code itératif où une tâche dépend de 3 autres de l'itération précédente.
On en revient ici à l'intérêt des structures de données et classes : fournir un ensemble d'opérations sur un ensemble de données qui participent à former un tout cohérent.

II.2 Travail à réaliser : récapitulatif et scripting

On aimerait pouvoir faire un tableau récapitulatif donnant pour chaque thread (colonne) l'heure de début, la valeur de X tirée, et la durée passée à faire le calcul (ligne). Il y aurait donc une entrée pour chaque itération. Le résultat serait présenté sous une forme similaire à celle donnée dans la table 1 (modulo le contenu des cases intérieures).

Thread	Iteration 1	Iteration 2	Iteration 3...
Thread **1**	(12 :00 :01, 12, 15)	(12 :00 :20, 45, 250)	(12 :10 :14, 1, 0.02)
Thread **2**	(12 :00 :01, 9, 7)	(12 :00 :19, 5, 4)	(12 :10 :20, 2, 0.04)

Table 1 – Exemple de sortie du script

— Proposer à votre encadrant la solution que vous pensez mettre en place.

Il y a surtout 2 options ici : on peut modifier le code C++, pour mettre en place une matrice $nbthreads * nbtours$ de structures contenant les informations que chaque thread donne. Parmi les questions qu'on doit alors se poser, y a t-il des problèmes de synchro qui émergent ? (Normalement, ici, non ! Mais...)

L'autre solution est la solution par défaut : on n'a généralement pas accès au code, mais on suppose qu'il est suffisamment bien écrit, et que ses logs sont suffisamment bien formatés pour être traités. – dans notre cas, les lignes doivent donc être complètes

(printf, ou...) et comportant "Thread I : duration xxx" par exemple : les codes donnés dans les TP précédents sont bien écrits ! – On peut gérer avec un petit script.

Notez que formater votre sortie dans un format standard de représentation de données tel que `JSON` ou `XML` peut permettre un traitement efficace de vos données dans de nombreux langages de programmation. `libxml`, par exemple, est très répandue et permet de traiter du `XML` de façon simple dans de nombreux langages. Formatter vos données de façon standard est également un excellent moyen de les traiter avec des frameworks de traitement de données, tels que le langage R pour du calcul statistique, ou les bibliothèques `pandas` (encore pour le calcul statistique) et `seaborn` (pour la représentation graphique) dans le langage `Python`.

— Écrire un script qui `grep` sur chaque thread et en tire les informations voulues afin d'afficher une belle synthèse : les threads passent-ils le même temps à calculer le même calcul ? Plus important, la barrière de synchronisation fonctionne t-elle comme prévu ?

Une barrière de synchronisation permet de fixer un point de Rendez-Vous entre plusieurs threads, et de permettre que la suite du calcul soit déclenchée en même temps : il est normal de trouver des différences de temps, puisque c'est l'ordonnanceur qui donne la main.

En plus de voir l'importance d'un code propre en soi, l'idée est de comprendre qu'il peut y avoir besoin de l'étude des logs ! L'idée sera creusée dans la partie Administration de l'UE où on verra quelques détails sur certains fichiers de `/var/log/`

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).