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 ?

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

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

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 Tâ$che()$ 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 ;-)).

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.

```
avant
avant
avant
apres
apres
apres
```

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

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.

```
1   class latch {
2   public:
3       latch(unsigned int nb_threads)  : _nb_threads(nb_threads) { }
4       void wait() {
5           std::unique_lock<std::mutex> lck(_m);
6           ++_waiting;
7           while (_waiting != _nb_threads) {
8               _cv.wait(lck);
9           }
10
11          if (_waiting == _nb_threads) {
12              _cv.notify_all();
13          }
14      }
15
16  private:
17      std::mutex _m;
```

```
18          std::condition_variable _cv;
19          unsigned int _nb_threads;
20          unsigned int _waiting = 0;
21      };
```

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   }
```

Une version précédente du corrigé parlait de « réentrante », mais ce n'est pas la définition de ré-entrance. En première approximation, ré-entrant ≈ thread-safe sans mutex (la fonction doit pouvoir être suspendue, et pendant qu'elle est suspendue être rappelée une deuxième fois, cf. `https://en.wikipedia.org/wiki/Reentrancy_(computing)`), donc à partir du moment où on a des mutex, on n'est pas ré-entrant.

Je (Matthieu) n'ai pas trouvé de vocabulaire officiel pour distinguer les deux cas, mais C++20 dit « Unlike std : :latch, barriers are reusable », dont on *réutilise* le vocabulaire « ré-utilisable ».

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.

cf. barrier.cpp, ci-dessous.

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.

$N = 10$ **:** tous les threads font `sleep(1)` en parallèle, puis appellent `wait()` en même temps (donc temps de blocage négligeable), ce qui prend 1 seconde. Répété 5 fois, le total est 5 secondes.

Mono-cœur : Rien ne change : `sleep()` ne monopolise pas de cœur donc plusieurs `sleep()` en parallèle s'exécutent déjà en parallèle sur une machine mono-cœur. C'est difficile à vérifier expérimentalement de nos jours par contre.

```
1  #include <thread>
2  #include <mutex>
3  #include <condition_variable>
4  #include <iostream>
5  #include <vector>
6  #include <sstream>
7  #include <unistd.h> // pour le sleep
8
9  using namespace std;
10
11 static const int NB_THREADS = 10;
12 static const int NB_ITERATIONS = 5;
13
14 class barrier
15 {
16 public:
17   // constructeur
18   barrier(int n)
19   {
20     nb_threads = n;         // nb total de threads initialisé à n
21     nb_waiting = 0;         // nb de threads qui attendent initialisé à
   0
22     generation_number = 0; // compteur de génération mis à 0
23   }
```

```
24
25      void wait()
26      {
27        // instantiation du verrou qui est locké
28        std::unique_lock<std::mutex> l(m);
29
30        // incrémentation du nb de threads qui attendent
31        nb_waiting++;
32
33        // génération courante
34        int current_generation = generation_number;
35
36        // Si tous les threads attendent
37        if (nb_waiting == nb_threads)
38          {
39            // incrémente du nb de générations pour le dernier thread qui fait le wait
40            generation_number++;
41
42            // on remet à zéro le nb de threads qui attendent
43            nb_waiting = 0;
44
45            // notification de la condition
46            all_present.notify_all();
47          }
48        else // sinon
49          {
50            // attente passive, tant que la génération courante ne correspond pas au n
51            // c'est-à-dire, tant que tous les threads ne sont pas en attente
52            while (generation_number == current_generation)
53              {
54                std::cout << "waiting ...\n";
55                all_present.wait(l);
56                std::cout << "waiting ... done\n";
57              }
58          }
59      }
60
61    private:
62      int nb_threads;                       // nb de threads total
63      int nb_waiting;                       // nb de threads qui attendent
64      int generation_number;                // compteur de génération
65      std::condition_variable all_present; // variable de condition
66      std::mutex m;                         // mutex
67    };
68
69    // Travail d'un thread
70    // void barrier_infinite(barrier &b)
71    void barrier_ngen(barrier &b)
72    {
73      // while (true)
74      for (int i = 0; i < NB_ITERATIONS; i++)
```

```
75      {
76        std::stringstream s;
77
78        s << "AVANT - iteration i=" << i << endl;
79        cout << s.str();
80        // Attente de 1 seconde
81        sleep(1);
82
83        // Accede à la barrière
84        b.wait();
85
86        // Vider le stream ;)
87        s.str("");
88        s << "APRES - iteration i=" << i << endl;
89        cout << s.str();
90      }
91  }
92
93  int main()
94  {
95
96    // cout << "combien de threads vous souhaitez ?" << endl;
97    // cin >> NB_THREADS;
98
99    // cout << "Combien de tours ?" << endl;
100   // cin >> NB_ITERATIONS;
101
102   // Tableau de threads
103   vector<thread> vector_threads;
104
105   // Instance de barrier pour NB_THREADS threads
106   barrier b(NB_THREADS);
107
108   // Instantiation des threads
109   for (int i = 0; i < NB_THREADS; i++)
110     {
111       vector_threads.push_back(thread(barrier_ngen, ref(b)));
112     }
113
114   // Attente de la terminaison des threads
115   for (int i = 0; i < NB_THREADS; i++)
116     {
117       vector_threads[i].join();
118     }
119 }
```

Une solution alternative est fournie ici :

```
1  #include <thread>
2  #include <mutex>
3  #include <condition_variable>
4  #include <iostream>
```

```
5   #include <vector>
6   using namespace std;
7   #include "barrier.h"
8
9
10  static const int NB_THREADS = 10;
11  static const int NB_ITERATIONS = 10;
12
13  void work(barrier &b) {
14    int n = 0;
15    for (int i = 0; i <= NB_ITERATIONS; i++)
16      {
17        b.wait();
18        cout << this_thread::get_id() << ": Done iteration " << n << endl;
19      }
20  }
21
22  int main() {
23    // Tableau de threads
24    vector<thread> vector_threads;
25
26    // Instance de barrier pour NB_THREADS threads
27    barrier b(NB_THREADS);
28
29    // Instantiation des threads
30    for (int i = 0; i < NB_THREADS; i++)
31      {
32        vector_threads.push_back(thread(work, ref(b)));
33      }
34
35    // Attente de la terminaison des threads
36    for (int i = 0; i < NB_THREADS; i++)
37      {
38        vector_threads[i].join();
39      }
40  }
```

L'idée de cette version est de bloquer les threads qui essayeraient de « prendre de l'avance » en appelant `wait()` trop vite. 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  {
3  public:
4    // constructeur
5    barrier(unsigned int nb_threads) : _waiting(0) _nb_threads(nb_threads) {}
6
7    void wait()
8    {
9      // mutex locké
10     std::unique_lock<std::mutex> lck(_m);
11
12     /* Bloque les threads qui arrivent ici après avoir été débloqués
13      * dans la synchro précédente. La synchro précédente n'est pas
14      * résolue tant que nb_thread_a_liberer != 0.
15      */
16
17     // On attend que tous les threads aient été libérés
18     // pour pouvoir aller sur la barriere
19     while (nb_thread_a_liberer != 0)
20     {
21       cout << this_thread::get_id() << ": waiting for reset. to_free=" << nb_threa
22
23       // attente que tous les threads soient libérés
24       all_libere.wait(lck);
25     }
26
27     // incrémente le nb de threads qui attendent
28     ++nb_waiting;
29
30     // Tous les threads n'attendent pas
31     if (nb_waiting != nb_threads)
32     {
33       /* Bloque pour une synchronisation. On vient d'incrémenter nb_waiting, donc
34         nb_waiting==0 ne peut venir que d'un débloquage. */
35
36       // on attend que le nb de waiting repasse à zéro
37       // et il repasse à zéro quand le dernier thread a atteint la barrière
38       while (nb_waiting != 0)
39       {
40         cout << this_thread::get_id() << ": waiting for barrier. waiting=" << nb_w
41
42         // Attente que tous les threads soient sur la barriere
43         all_present.wait(lck);
44       }
45
46       // Thread à libérer,
47       // donc on décrémente le nb de thread à libérer
```

```
48        nb_thread_a_liberer--;
49
50        /* Le dernier thread qui sort notify les threads qui ont
51         * pu refaire un wait() avant que la synchro ne soit complètement
52         * résolue.
53         */
54        // cas où tous les threads ont été libérés
55        if (nb_thread_a_liberer == 0)
56        {
57          // notification que tous les threads ont été libérés
58          all_libere.notify_all();
59        }
60      }
61      else // sinon - tous les threads attendent - un seul thread passe ici
62      {
63        /* Normalement faire un notify ne reschedule pas les threads, donc
64         * inverser les deux lignes suivantes ne change rien. Dans la
65         * pratique, il est préférable de faire les modifs sensibles avant
66         * le notify par acquis de conscience.
67         *
68         * nb_thread_a_liberer devient nb_threads - 1 pour bloquer les threads qui
69         * pourraient revenir dans cette fonction avant que la synchro
70         * courante ne soit résolue.
71         */
72
73        // ce thread remet à zéro le nb de threads qui attendent
74        nb_waiting = 0;
75
76        // ce thread met le nb de thread à libérer à nb_threads - 1
77        nb_thread_a_liberer = nb_threads - 1;
78
79        // notification de la condition que tout le monde attend sur la barriere
80        // (par 1 seul thread, le dernier qui a atteint la barriere)
81        all_present.notify_all();
82      }
83    }
84
85  private:
86    std::mutex _m;                // mutex
87    std::condition_variable _cv; // variable de condition
88    std::condition_variable _reset;
89    unsigned int _nb_threads; // nb de threads total
90    unsigned int _waiting;    // nb de threads qui attendent
91    unsigned int _to_free = 0;
92  };
```

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.

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).