Université de Nice-Sophia Antipolis
Polytech ELEC4

Lundi 27 avril 2020
Durée: 2h

Processus légers
Travaux Dirigés – Séance n. 14

---

1 Introduction

Un processus léger, appelé aussi *thread*, est une entité exécutable qui s'exécute au sein d'un processus. Contrairement aux processus qui possèdent chacun leur propre espace de mémoire pour leur exécution, les processus légers partagent la mémoire du processus au sein duquel ils s'exécutent.

Un processus peuvent contenir plusieurs threads qui s'exécutent en (quasi-)*parallèle*, on dit aussi de façon *concurrente*. La fonction `main` est exécutée dans un thread particulier, de sorte qu'un processus contient toujours au moins un thread.

Si les threads s'exécutent de façon indépendante, leur exécution est dite *asynchrone*. En revanche, s'ils doivent collaborer et se coordonner, en particulier pour accéder à une ou plusieurs ressources partagées, leur exécution devra être synchronisée. On parle alors d'exécution *synchrone*.

La programmation concurrente est bien plus difficile que la programmation séquentielle, en particulier à cause des mécanismes de synchronisation qu'il faut mettre en place. Dans ce TD, nous aborderons quelques exemples simples qui présentent les notions de base de la programmation concurrente.

2 Déclaration

La classe `std::thread` de la bibliothèque standard permet de déclarer des processus légers. Elle possède plusieurs constructeurs, et en particulier un constructeur dont :

— le premier paramètre est une fonction dont le code sera exécuté par le thread ;
— les paramètres qui suivent (éventuellement aucun) sont les paramètres de la fonction.

```
#include <thread>


void f1() { /* ... */ }
void f2(int n, char c) { /* ... */}

class C {
public:
  void f3() { /* ... */ }
};

int main() {
  // création de 4 threads
  std::thread t1(f1);
  std::thread t2(f2, 1, 'b');
  std::thread t3(&C::f3, C());
  int x;
  std::thread t4([](int &n) { /* ... */ }, std::ref(x));
  ...
}
```


Notez qu'il faut indiquer à l'aide du *wrapper* `std::ref` que le paramètre `x` sera transmis par référence lors de l'appel de la fonction anonyme au moment de l'exécution du thread.

3 Un producteur de nombres aléatoires

On veut écrire un processus léger qui produit indéfiniment des nombres aléatoires sur l'intervalle $[0; \max - 1]$ et les écrit sur la sortie standard séparés par un espace. Entre deux nombres produits, le thread marquera une pause de quelques secondes. Avec la méthode `std::chrono::seconds` qui permet de fournir une durée en secondes, et la méthode `std::this_thread::sleep_for` qui permet de suspendre l'exécution du thread courant pour une durée donnée, on peut écrire une fonction `prodAlea` et un thread associé comme suit :

```
#include <thread>
#include <chrono>
#include <ctime>
#include <iostream>
#include <cstdlib>

void prodAlea() {
  const int pause = 1;  // seconde
  const int max = 300;

  std::srand(std::time(0));
  while (true) {
    std::this_thread::sleep_for(std::chrono::seconds(pause));
    std::cout <<  std::rand()%max << " " << std::flush;
  }
}

int main()
{
  std::thread t0(prodAlea);
  t0.join();

  return EXIT_SUCCESS;
}
```

exercice 1) Écrivez et testez ce programme. Notez que vous avez besoin :
— de l'option `-pthread` (ou `-lpthread`) pour compiler le programme ;
— de `std::flush` pour vide le tampon de sortie et écrire le nombre produit immédiatement ;
— de la méthode `join()` pour que le thread principal `main` attende la fin du thread `t0`.

exercice 2) Modifiez ce programme de telle sorte que la fonction `prodAlea` soit paramétrée sur le temps de pause.

exercice 3) Créez 3 threads qui exécutent `prodAlea` avec des temps de pause différents, par exemple, 1, 2 et 3 secondes. Testez votre programme.

On souhaite maintenant identifier le thread qui produit tel ou tel nombre.

exercice 4) Modifiez la fonction `prodAlea` pour qu'elle écrive sur la sortie standard chaque nombre n suivi de l'identification du thread *id_t* qui le produit, au format : *[n - id_t]*. La fonction `std::this_thread::get_id()` renvoie l'identifiant, de type `std::thread::id`, du thread courant. Testez votre programme.

Afin de faciliter la programmation des processus légers, vous allez utiliser la classe `Thread` similaire à celle de l'API Java, donnée ci-dessous. Cette classe est abstraite, et est à implémenter

par la classe qui représente le processus léger qu'on veut définir. Elle contient la méthode abstraite `run` à implémenter par la classe héritière. Le démarrage du thread sera fait par la méthode `start`.

```
#pragma once
/*
 * Classe abstraite pour réprésenter un thread défini par la
 * la classe héritière qui devra implémenter la méthode run.
 *
 * Note : Patron de méthode
 *
 * @author: Vincent GranetVincent.Granet@univ-cotedazur.fr
 */
#include <thread>

class Thread
{
private:
  std::thread p;

protected:
  virtual void run()=0;
  virtual ~Thread() {}

public:
  // Rôle : crée et démarre l'exécution du thread courant
  void start() {
    this->p = std::thread(&Thread::run, this);
  }

  void join() { this->p.join(); }
};
```

exercice 5) Écrivez une classe `Aleatoire` qui hérite de la classe `Thread`, et refaites l'exercice 4.

---

exercice 6) Donnez 1 seconde comme valeur de pause à vos trois threads. Compilez et exécutez votre programme. Que constatez-vous ?

Dans votre programme les 3 threads s'exécutent en parallèle, en fait dans un *pseudo-parallelisme*. Chaque processus léger s'exécute à tour de rôle selon un ordre défini par l'*ordonnanceur* qui peut préempter à tout moment l'exécution d'un thread. D'une façon générale, on **ne peut pas** faire de supposition sur les vitesses relatives des threads.

Dans le programme précédent, si on veut que la fonction `prodAlea` fasse une écriture complète *[n - id_t]*, il faut garantir que le thread ne soit pas interrompu avant qu'il ait achevé la totalité de son écriture écriture.

4 Section critique

Le problème précédent est le problème général d'accès par n processus ou threads à une *zone protégée* alors seuls m ($m < n$) processus ou threads sont admis dans la zone. Lorsque $m = 1$, on parle de section *critique*, et donc, au plus un seul processus ou thread peut accéder à la section critique. Les processus ou threads sont alors en *exclusion mutuelle*. Les sections critiques sont typiquement des données qui ne peuvent être manipulées que par un seul thread (*e.g.* solde d'un compte bancaire).

Il existe plusieurs mécanismes pour assurer l'*exclusion mutuelle*. La bibliothèque C++ met en œuvre les *mutex* pour garantir l'exclusion mutuelle entre les threads.

Un mutex est un verrou dont le thread *prend possession* et le ferme après être entré dans la section critique, bloquant ainsi tous les autres threads qui cherchent à entrer. Lorsque le thread sort de la zone critique, il ouvre et libère le verrou, ce qui réveille un thread en attente (s'il y en a un, bien sûr) qui pourra à son tour prendre possession du verrou et accéder à la zone critique.

Le verrou est un objet de type `std::mutex` qui possède les fonctions `lock` et `unlock` pour le fermer et l'ouvrir. Notez qu'un thread ne doit pas posséder le mutex avant d'appeler la méthode `lock`. D'autre part, un mutex ne doit pas être détruit par un thread alors qu'un autre le possède.

```
#include <mutex>
...
std::mutex m;
...
m.lock();
// section critique
m.unlock();
...
```

La classe `stdd::lock_gard` permet de poser un verrou, et de le libérer automatiquement à la sortie du bloc dans lequel il est déclaré :

```
#include <mutex>
...
std::mutex m;
...
{
   std::lock_guard<std::mutex> verrou(m);
   // section critique
   ...
}
// le verrou a été libéré
...
```

---

exercice 7) Modifiez votre classe `Aleatoire` afin que l'écriture du numéro d'identification du thread et du nombre aléatoire ne soient pas interrompu.

---

exercice 8) On veut faire une course de chevaux qui parcourent une distance fixe l. Chaque cheval sera identifié par un nom et sera représenté par un thread. Pendant la course, les chevaux marquent une pause aléatoire entre chacune de leurs foulées. Tous les chevaux partent en même temps de la fonction main, et à la fin de la course le programme indiquera l'ordre d'arrivée des chevaux.

Vous ferez une classe `Cheval` pour représenter chaque cheval, et une classe `Course` qui gère la course des chevaux. Quelle est la ressource critique qu'il faut protéger ? Pensez à faire une classe `Arrivee` pour la représenter.

Votre programme principal pourra avoir la forme suivante :

```
#include <vector>
#include <cstdlib>
#include "Course.hpp"

int main() {
  std::vector <std::string> chevaux = {
    "Easy Rider", "Joli Coeur", "Étoile Filante", "Belle de nuit"
  };
  // créer une course
  Course c(chevaux);
  // lancer la course
```

```
  c.run();
  // afficher les résultats
  std::cout << c.donnerLesResultats();

  return EXIT_SUCCESS;
}
```

5 Modèle Producteurs/Consommateurs

Le modèle « Producteurs/Consommateur » est un exemple classique de programmation concurrente où plusieurs processus/threads, *les producteurs*, produisent de l'information dans une ressource partagée (*e.g.* une file d'attente), et d'autres processus/threads, *les consommateurs*, consomment l'information produite. Dans ce modèle, il est nécessaire que les producteurs et les consommateurs se *synchronisent*.

Tout d'abord, la ressource partagée est *critique*, et doit être protégée de telle sorte qu'à tout moment un seul processus/thread a accès à la ressource. Ensuite, cette ressource partagée peut être d'une taille finie et fixe. Si tel est le cas, lorsqu'elle est pleine, les producteurs devront suspendre leur activité et attendre que des consommateurs libèrent de la place. De façon symétrique, si elle est vide, les consommateurs devront suspendre leur activité et attendre que producteurs produisent dans la ressource partagée.

Pour résoudre ce problème, il existe plusieurs solutions à l'aide des sémaphores de Dijkstra, ou des mécanismes IPC (Inter-process communication).

Nous alors voir comment la mettre en œuvre en C++ à l'aide de *mutex* et de *variable de condition*. Les producteurs et les consommateurs sont représentés par des threads.

Le cœur de l'application est une classe (générique) `Tampon` qui représente la ressource partagée. Dans cette classe, les valeurs produites et consommées par les threads seront conservées dans une file d'attente générique. La file d'attente conservera les valeurs dans un tableau de **taille fixe**.

exercice 9) Complétez la classe générique `File` suivante :

```
/*
 * Cette classe implémente le Type Abstrait File d'attente
 * Les éléments sont conservés dans un tableau de taille fixe
 */
template <typename T>
class File {
protected:
  T *tampon;
  int nbElem, tete, queue, size;
public:
  // Rôle : construit une File de taille n
  File(int n) ....
  }
  // Rôle : renvoie le premier élément de File courante
  T premier() {
    ....
  }
  // Rôle : ajoute x en queue de File
  void enfiler(T x) {
    .....
  }
  // Rôle : retire le premier élément de la File courante
  void defiler() {
   ....
```



```
  }
  bool estVide()  { .... }
  bool estPlein() { .... }
};
```

À l'aide de la classe `File`, on va pouvoir écrire la classe générique `Tampon`. Cette classe possédera deux méthodes :

- `mettre`, utilisée par les producteurs, pour ajouter une valeur dans le tampon ;
- `prendre`, utilisée par les consommateurs, pour retirer une valeur du tampon.

La classe générique `Tampon` a la forme suivante :

```
/*
 * Cette classe représente le tampon, ressource partagée par des
 * threads dans un modèle producteurs/consommateurs
 */
template <typename T>
class Tampon {
private:
  static const int DEFAULT_SIZE = 10;
  std::condition_variable tampon_plein;
  std::condition_variable tampon_vide;
  std::mutex m_em; // mutex pour protégé l'accès à la file
  std::mutex m; // mutex pour les variables de conditions
  File<T> tampon;  // une file d'attente
public:
  Tampon(int n = DEFAULT_SIZE) : tampon(File<T>(n)) {}
  /*
   * Rôle : ajoute la valeur e dans le tampon courant
   */
  void mettre(T e) {
     ....
     tampon.enfiler(e);
     ....
  }
  /*
   *  Rôle : retire du tampon courant une valeur et la renvoie
   */
  T prendre() {
     ....
     T x = tampon.premier(); tampon.defiler();
     ....
  }
};
```

Une *variable de condition* de type `std::condition_variable` assure la communication et la synchronisation de plusieurs threads sur une condition. Une variable de condition est toujours associée à un mutex de type `std::unique_lock<std::mutex>`, qu'il faudra créer au préalable. Ce dernier est construit à partir d'une variable de type `std::mutex`. La déclaration :

```
std::unique_lock<std::mutex> lock(m);
```

construit un mutex `unique_lock` à partir du mutex `m`, et verrouille le mutex par l'appel `m.lock()`.

La classe `std::condition_variable` possède différentes méthodes. Vous utiliserez en particulier :

- la méthode `wait` qui permet de suspendre le thread courant tant qu'une condition passée en 2ème paramètre n'est pas vérifiée. Le premier paramètre est le mutex qui est dans ce cas libéré.
- la méthode `notify_one` réveille **un** thread endormi et le prévient que la condition est vérifiée. Le thread réveillé *récupère* le mutex et poursuit alors son exécution. La méthode `notify_all`

est similaire à `notify_one`, sauf qu'elle réveille **tous** les threads endormis.

Pour synchroniser les threads lorsque le tampon est vide, ou lorsqu'il est plein, vous utiliserez deux variables de condition. Les producteurs, *resp.* les consommateurs, devront être endormis lorsque le tampon est plein, *resp.* vide. Un producteur, *resp.* un consommateur, après avoir produit, *resp.* consommé, une valeur dans le tampon devra notifier le/les consommateurs, *resp.* producteurs, endormis (s'ils existent).

Lisez la description de la classe `std::condition_variable` sur la page `http://en.cppreference.com/w/cpp/thread/condition_variable`.

exercice 10) Complétez la classe `Tampon` en remplaçant les points de suspension par des instructions appropriées.

---

exercice 11) Écrivez les classes génériques `Producteur` et `Consommateur` qui produisent, *resp.* consomment, à intervalles de temps réguliers, des valeurs dans le tampon. Les valeurs sont produites aléatoirement à l'aide d'un générateur générique. Le temps de pause sera passé en paramètre aux constructeurs des deux classes, ainsi que le tampon.

exercice 12) La fonction `main` du programme principal pourra avoir la forme ci-dessous. Faites varier les temps de pauses, et affichez des traces pour mettre en évidence l'exécution des threads.

```
#include <cstdlib>
#include "Producteur.hpp"
#include "Consommateur.hpp"
#include "Generateur.hpp"
int main(int argc, char * argv[]) {
  // créer le tampon (la ressource partagée) d'entiers
  Tampon<int> *tp = new Tampon<int>();
  // créer un générateur d'entiers
  GenerateurInt *rand = new GenerateurInt();
  // créer des producteurs d'entiers
  Producteur<int> p0(rand, tp,4);
  Producteur<int> p1(rand, tp,4);
  Producteur<int> p2(rand, tp,4);
  // créer des consommateurs d'entiers
  Consommateur<int> c0(tp,1);
  Consommateur<int> c1(tp,1);
  // lancer l'exécution des producteurs et des consommateurs
  p0.start(); p1.start(); p2.start();
  c0.start(); c1.start();
  // attendre la fin des threads
  p0.join(); p1.join(); p2.join();
  c0.join();  c1.join();
  return EXIT_SUCCESS;
}
```

---

exercice 13) À l'aide de *gtkmm*, programmez une interface graphique qui affiche l'évolution du tampon au fur et à mesure des productions et des consommations des threads.