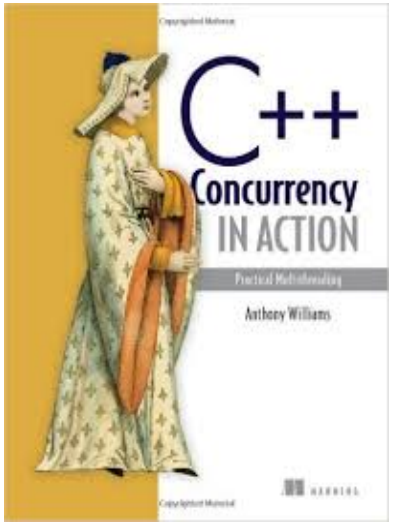
HLIN603 : Concurrence en C++

Hinde Bouziane (bouziane@lirmm.fr)

UM - LIRMM

Avant propos

- Ce cours est une sensibilisation à la notion de concurrence
  - Bonne base mais vous ne serez pas experts à la fin de ce cours
- Illustrations en C++.
  - Standard C++11
  - Livre
    "C++ Concurrency in Action : Practical Multi-threading"
  - Notions non évidentes au premier abord : accrochez vous !



- La majorité des notions présentées existent en Java
  - Travail personnel et/ou exercice pendant le dernier TP.

Concurrence

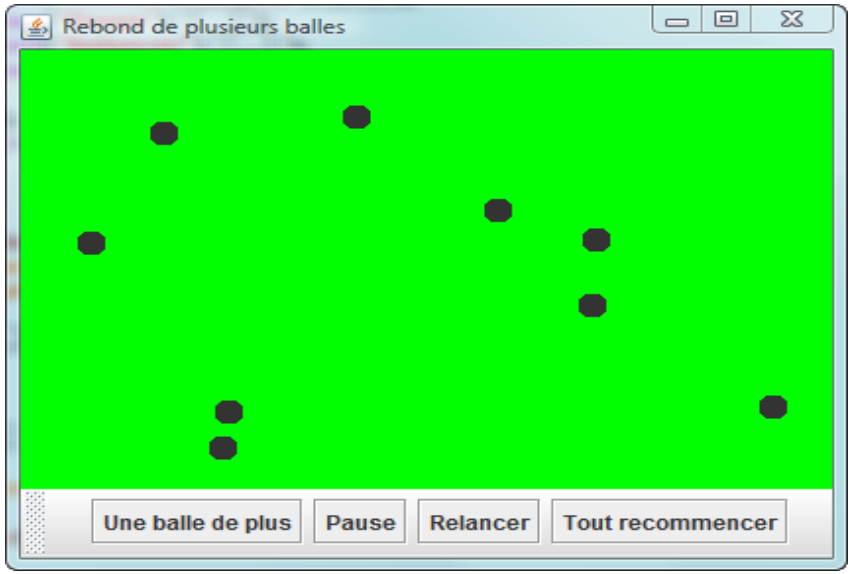
- A votre avis, de quoi s'agit-il ?

Problème introductif 1

- Retour au cours ”Réseau (HLIN611)”
- Si un serveur reçoit des requêtes de plusieurs clients et que les clients réalisent des entrées (saisies au clavier) pour envoyer des requêtes
- Comment fait le serveur pour gérer les différents clients ?
- Et si le temps d’exécution d’une requête est long ?

Problème introductif 2

- Dans une interface graphique, des objects se déplacent.



- Pendant ce déplacement, l'IHM doit rester fonctionnelle.
- Mais le déplacement des objets prend du temps.
- Comment faire pour éviter que toute action sur les boutons soit sans effet pendant le déplacement ?

Concurrence - définition

- Exécuter plusieurs codes en même temps (en parallèle) sur une même machine
  - Ca peut être le même code
  - Ne pas confondre avec application distribuée, qui implique plusieurs machines (exemple d'une application client serveur)
- Exécution en parallèle de plusieurs processus
  - processus indépendants (différentes applications)
  - fork()
- Exécution en parallèle de plusieurs codes dans le même processus
  - *threads*
- Le reste du cours se concentre sur la notion de *thread*

Notion de *thread*

Thread = fil en anglais. Un thread est un fil d'exécution

- Traductions : activité, tâche, fil d'exécution ou processus léger (par opposition au processus lourd créé par fork())
- Les threads permettent de dérouler plusieurs suites d'instructions, en parallèle, à l'intérieur du même processus
- Un thread est une partie d'un processus ou un chemin d'exécution à l'intérieur d'un processus
- Concrètement, un thread exécute une fonction
  - chaque thread a sa propre pile et des variables locales
  - un thread peut partager des données en mémoire avec d'autres threads. Ainsi, la communication entre threads se fait via le partage de variables.
  - un thread s'exécute de manière asynchrone.

Pourquoi ?

- Pouvoir exécuter du code pendant l'attente de données en entrée/sortie (I/O)
- Pouvoir exécuter en parallèle des calculs indépendants
  - améliorer le temps d'exécution global d'un programme
  - utiliser plusieurs cores d'une machine.

Apperçu de la suite

Rappel : le tout en C++11.

Utilisation d'un thread :

- Créer un thread et attendre la fin de son exécution
- Passage de paramètres
- Récupérer un résultat (*futures*)

Notions liées aux threads

- Protection des variables partagées (*atomics*, *mutex*)
- Synchronisation (exclusion mutuelle (*mutex*), *conditions*)

A ne pas oublier : Compilation avec les options "-Wall -std=c++11 "

Création d'un thread : la classe **std : :thread**

```cpp
#include <thread>
#include <iostream>

void my_thread_func(){

  std::cout<<"hello"<<std::endl;
}


int main(){

  // crée un objet thread qui lance l'exécution de my_thread_func
  std::thread t(my_thread_func);

  // atten de la fin de l'exécution du thread (donc de my_thread_func)
  t.join();
  return 0;

}
```

Création d'un thread, suite 1

```cpp
#include <thread>
#include <iostream>

class Affiche class{

public:
  void operator()() const{

    std::cout<<"hello"<<std::endl;
  }
};

int main(){

  Affiche class t_obj;
  // crée un objet thread en utilisant operator
  std::thread t(t_obj);

  // attend de la fin de l'exécution du thread
  t.join();
  return 0;

}
```

Création d'un thread, suite 2

```cpp
#include <thread>
#include <iostream>

class AfficheClass{

public:
  void affiche(){

    std::cout<<"hello"<<std::endl;
  }
};

int main(){

  AfficheClass t_obj;
  // crée un objet thread qui lance l'exécution de affiche
  std::thread t(&AfficheClass::affiche, &t_obj);

  // attend de la fin de l'exécution du thread (donc de affiche)
  t.join();
  return 0;

}
```

Création d'un thread, suite 3

```cpp
#include <thread>
#include <iostream>

int main(){

  // crée un objet thread qui lance l'exécution d'une fonction lambda
  std::thread t([&](){

      std::cout<<"hello"<<std::endl;

    });

  // attend de la fin de l'exécution du thread
  t.join();
  return 0;

}
```

Création d'un thread : pour résumer

- Plusieurs façons de créer des threads
- L'essentiel est d'utiliser des éléments dits "callable"
  - Fonctions, fonctions membres, fonctions Lambda, objets avec "operator()", etc.

Autres opérations de la classe std : :thread

- *void join()* : attend la fin de l'exécution du thread courant
- *id get_id() const noexcept* : retourne l'identifiant de l'objet thread courant
- *void yield()* : permet d'avertir le système d'exploitation que le thread n'a pas de travail à faire dans l'immédiat. Le système d'exploitation est donc libre de passer le contrôle à un autre processus (léger ou non).
- ...

Un autre exemple

```
#include <thread>
#include <iostream>

int main(){

  std::thread t1([&](){

      std::cout<<"+";
    });

  std::thread t2([&](){

      std::cout<<"!";
    });

  // attendre la fin des threads
  t1.join();
  t2.join();

  std::cout<<std::endl;
  return 0;
}
```

- Combien de threads sont exécutés par ce programme ?
- Quel est le résultat attendu ?
- $\rightarrow$ démonstration
- Proposez une modification pour mieux mettre en évidence le parallélisme
- $\rightarrow$ démonstration

Arguments d'un thread

```
#include <thread>
#include <iostream>

void increment(int i, int j){
  ++i;
  ++j;
}

int main(){

  int x=42;
  int y=10;
  std::thread t1(increment, x, y);

  t1.join();
  std::cout<<"x="<<x<<"y="<<y<<std::endl;

}
```

Qu'affiche ce programme ?

Arguments d'un thread, suite

```cpp
#include <thread>
#include <iostream>
#include <functional>

void increment(int& i, int j){
  ++i;
  ++j;
}

int main(){

  int x=42;
  int y=10;

  std::thread t1(increment, std::ref(x), y);

  t1.join();
  std::cout<<"x="<<x<<" y="<<y<<std::endl;

}
```

Qu'affiche ce programme ?

Arguments d'un thread : pour résumer

- Tout mode de passage de paramètre est possible.
- Attention au passage de paramètre par référence ou par adresse. Pourquoi ?

Valeur de retour d'un thread

- Il est possible de récupérer les résultats de calcul d'un thread à l'aide du passage de paramètres par référence ou adresse, mais aussi à l'aide des *futures* et *promises*
- Une promise s'appuie sur la notion de **future**. Ce cours présente uniquement cette dernière.
- *Future* est un moyen de récupérer un résultat qui sera ultérieurement disponible.
- Utilisation via la classe *std : :future*
- En utilisant une future, la fonction *std : :async* est utilisée à la place de la classe std : :thread pour le lancement d'un thread.

Future : exemple

```cpp
#include <future>
#include <iostream>

int testfuture(){

  int res;
  std::cout << "entrez un entier"<<std::endl;
  std::cin>>res;
  return res;
}

int main(){

  std::future<int> resultatThread = std::async(testfuture);

  /* début code s'exécutant en parallèle avec delui du thread*/
  std::cout<<"Attente du resultat du thread"<<std::endl;
  int res = resultatThread.get();
  /* exécution de thread terminée */
  std::cout<<"Resultat du thread = "<<res<<std::endl;
  return 0;
}
```

- Avons nous besoin d'utiliser la fonction *join()* ?

Future : *std : :future* et *std : :async*

L'essentiel pour débuter :

- *T get()* : demande la valeur de retour du thread auquel l'objet future courant est associé. Cette fonction est bloquante : elle attend tant que cette valeur n'est pas disponible.

La définition utilisée de la fonction *std : :async*

```
template< class Function, class... Args>
std::future<typename std::result_of<Function(Args...)>::type>
    async( Function&& f, Args&&... args );
```

- elle est responsable de la creation du thread lançant *f*

Voir la documentation pour d'autres détails

Accès concurrent à une variable (1/2)

- Analisez le code suivant et dites ce qu'il est sensé afficher

```
#include <thread>
#include <iostream>


int main(){

  int counter = 0;

  std::thread t1([&](){

      for(int i=0; i < 1500; i++)
        ++counter;
    });

  std::thread t2([&](){

      for(int i=0; i < 3000; i++)
        ++counter;
    });

  t1.join();
  t2.join();
  std::cout<<" Total = "<<counter<<std::endl;

  return 0;

}
```

Accès concurrent à une variable (2/2)

D'où vient le problème ?

- La commutation de contexte entre les threads peut se faire à n'importe quel moment
  - Sans oublier qu'une simple instruction implique plusieurs instructions en langage assembleur.
- Possibilité de lire une variable dans un état "incohérent"
  - la variable "counter" peut être lu par t1 puis par t2, mais avant que chacun n'effectue une incrémentation (résultat +1 au lieu de +2)
- Origine du problème : l'instruction "++counter ;" est interruptible et counter est accessible par les threads en lecture et écriture
  - Le problème ne se pose pas dans le cas de lectures seules

Protection de l'accès aux variables : Atomics (1/2)

Solution : utiliser des outils de synchronisation

- Pour l'instruction "++counter ;", il suffirait qu'elle soit **atomique**
- Utiliser le type **scalaire atomique**
  - Les opérations effectuées sur une variable de ce type est obligatoirement atomique
  - Support pour les données de type intégral (bool, char, int, long, long long signé ou non), trivialement copiable ou de type pointeur
    - Les opérations arithmétiques sur un entier atomique seront atomiques.
- Exemple d'utilisation pour un entier :
  #include <atomic>
  std : :atomic<int> i ;
- Remarque : avant d'utiliser le type "atomic" pour une variable, vérifier sa faisabilité, sinon utiliser des primitives de synchronisation (suite du cours)

Protection de l'accès aux variables : Atomics (2/2)

- Retour à l'exemple :

```
#include <thread>
#include <iostream>
#include <atomic>

int main(){

  std::atomic<int> counter(0);

  std::thread t1([&](){

      for(int i=0; i < 1500; i++)
        ++counter;
    });

  std::thread t2([&](){

      for(int i=0; i < 3000; i++)
        ++counter;
    });

  t1.join();
  t2.join();
  std::cout<<" Total = "<<counter<<std::endl;

  return 0;

}
```

Protection de l'accès aux variables : Mutex (1/5)

Les atomics ne sont pas suffisants (pensez aux types non intégral)

- Un *mutex* (std : :mutex) est un sémaphore ayant deux états : libre ou occupé (verrouillé).
- Lorsqu'un mutex est libre et qu'un thread demande de le verrouiller, le mutex passe à l'état occupé.
- Un seul thread peut obtenir le verrouillage à la fois.
- Si un thread a obtenu le verrouillage, un autre thread qui demande le verrouillage du même mutex sera bloqué (ou échouera dans un contexte non bloquant) jusqu'à ce que le mutex soit libéré et obtenu.
- Le verrouillage (*lock()*) et déverrouillage (*unlock()*) sont des opérations atomiques.

Protection de l'accès aux variables : Mutex (2/5)

- Analysez et dites ce qu'affiche ce code ?

```
#include <thread>
#include <iostream>

#include <list>

int main(){


  std::list<int> listOfInt;

  std::thread t1([&](){

      listOfInt.push_back(15); listOfInt.push_back(20);

    });

  std::thread t2([&](){

     listOfInt.push_back(25);

    });

  t1.join(); t2.join();
  int elem1 = listOfInt.front(); listOfInt.pop_front();
  int elem2 = listOfInt.front();
  std::cout<<"deux premiers elements de la liste : " << elem1 <<", "<< elem2 << std::endl;
  return 0;
}
```

Protection de l'accès aux variables : Mutex (3/5)

- Protection de la variable listOfInt et plus que ça ...

```
#include <thread>
#include <iostream>
#include <mutex>
#include <list>

int main(){

  std::mutex verrou;
  std::list<int> listOfInt;

  std::thread t1([&](){
      verrou.lock();
      listOfInt.push_back(15); listOfInt.push_back(20);
      verrou.unlock();
    });

  std::thread t2([&](){
     verrou.lock();
     listOfInt.push_back(25);
     verrou.unlock();
    });

  t1.join(); t2.join();
  int elem1 = listOfInt.front(); listOfInt.pop_front();
  int elem2 = listOfInt.front();
  std::cout<<"deux premiers elements de la liste : " << elem1 <<", "<< elem2 << std::endl;
  return 0;
}
```

Protection de l'accès aux variables : Mutex (4/5)

- Une autre façon d'utiliser un mutex, mais prudence !

```
#include <thread>
#include <iostream>
#include <mutex>
#include <list>

int main(){

  std::mutex verrou;
  std::list<int> listOfInt;

  std::thread t1([&](){
      verrou.lock();
      listOfInt.push_back(15); listOfInt.push_back(20);
      verrou.unlock();
    });

  std::thread t2([&](){
      std::lock_guard<std::mutex> locker(verrou);
     listOfInt.push_back(25);
    });

  t1.join(); t2.join();
  int elem1 = listOfInt.front(); listOfInt.pop_front();
  int elem2 = listOfInt.front();
  std::cout<<"deux premiers elements de la liste : " << elem1 <<", "<< elem2 << std::endl;
  return 0;
}
```

Protection de l'accès aux variables : Mutex (5/5)

- Un seul mutex peut protéger plusieurs variables, mais pas l'inverse.
- Les opérations *lock()* et *unlock()* sont atomiques, mais pas la portion de code qui se trouve entre les deux.
- Cette portion de code est appelée **section critique**
- Si un thread est dans une section critique, il doit être garanti qu'aucun autre thread n'y soit simultanément
  - Nous parlons d'**exclusion mutuelle**
- C'est le cas de l'exemple précédent.
- Pour utiliser plus d'opérations avec les mutex, voir la documentation.
  - verrouillage non bloquant, mutex temporels, etc.

Attente d'un évènement : introduction

```
class Participant{

public:
  void operator()(int id,  std::atomic<bool> & ready) const{

    while (!ready) {}

    for (int i=0; i<50000; ++i) {}

    std::cout <<" "<< id << " ";
  }
};

int main (void){

  std::atomic<bool> ready (false);
  Participant participants[5]; std::thread threads[5];
  std::cout << "course impliquant 5 threads comptant jusqu'a 50000\n";

  for (int i=0; i<5; ++i)
    threads[i] = std::thread(participants[i], i, std::ref(ready));

  int x; std::cin>>x;

  ready = true; // top depart !

  for (auto& th : threads)  th.join();

  std::cout << '\n';
  return 0;
}
```

Attente d'un évènement : deux solutions

- Attente active (exemple précédent). Correcte, simple mais à éviter.
- Utilisation des variables conditionnelles.

Attente d'un évènement : variable conditionnelle

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>

class Participant{

public:

  void operator()(int id, bool & ready, std::mutex & verrou, std::condition_variable & cond) const{

    {std::unique_lock<std::mutex> ul(verrou);

      if(!ready)
        cond.wait(ul);
    }
    for (int i=0; i<50000; ++i) {}

    std::cout <<" "<< id << " ";
  }
};
```

- Pourquoi des accolades autour de l'attente ?

Réveil d'un thread en attente d'un évènement

```
int main (void){
  bool ready = false;
  Participant participants[5]; std::thread threads[5];
  std::mutex verrou;
  std::condition_variable cond;
  std::cout << "course impliquant 5 threads comptant jusqu'a 50000\n";

  for (int i=0; i<5; ++i)
    threads[i]= std::thread(participants[i], i, std::ref(ready), std::ref(verrou), std::ref(cond));

  int x; std::cin>>x;

  {std::lock_guard<std::mutex> guard(verrou);
    ready = true;
    cond.notify_all(); // top depart !
  }

  for (auto& th : threads)  th.join();

  std::cout << '\n'; return 0;
}
```

- Que se passe-t-il en supprimant les accolades autour du réveil ?

Variables conditionnelles : suite

- *wait(...)* libère le mutex passé en argument et attend que la variable conditionnelle courante annonce un évènement (par un autre thread).
  - Le mutex doit être obtenu avant l'appel de *wait(...)*.
  - Au réveil, le mutex est à nouveau obtenu.
  - Le verrou sert en particulier à protéger l'accès aux variables partagées impliquées dans la représentation de l'évènement (variable "ready" dans l'exemple)
  - Retester l'évènement attendu après le réveil peut être nécessaire. Pourquoi ?
- Le réveil peut se faire de deux façons
  - Réveiller un seul thread *notify_one(...)*
  - Réveiller tous les threads en attente sur la même variable conditionnelle *notify_all(...)*. Remarque : les threads obtiendront le mutex un par un.

A retenir

- La majorité des concepts présentés ne peuvent pas être copiés (mutex, atomics, etc) .
- La gestion des threads n'est pas difficile, mais la communication (partage de données et synchronisation) peut l'être.
- Plus il y a de threads, plus il y a des chances que les sources d'erreurs augmentent :
  - Décomposer les problèmes à traiter et implémenter de manière progressive.
- La programmation concurrente est une compétence à acquérir
  - Traitement d'images, jeux/vidéos, applications client-serveur, Big-Data, simulations numériques, etc.
  - Et si vous n'en faites pas, vous aurez la capacité d'expliquer le comportement de plusieurs applications.
- Pour les intéressés, la concurrence (en C et pas que les threads) est aussi vu en M1, UE "Réseaux et communication".