Chapitre 11 — Multithreading en C++

1. Introduction au multithreading

Le multithreading permet d'exécuter plusieurs parties d'un programme en parallèle**,** pour :

- Profiter des processeurs multi-cœurs
- Paralléliser les tâches lourdes
- Améliorer réactivité (IHM, serveurs)
- Gérer des tâches asynchrones

Depuis **C++11**, la bibliothèque standard <thread> offre des outils complets :

- std::thread
- std::mutex, std::lock_guard
- std::condition_variable
- std::future, std::promise
- std::atomic
- thread_local

2. Attribut thread_local

thread_local permet qu'une variable ait **une instance par thread**.

Chaque thread possède **sa copie**, indépendante de celle des autres.

Exemple :

```
#include <iostream>
#include <thread>
using namespace std;

thread_local int compteur = 0;

void f() {
   compteur++;
   cout << "Compteur dans ce thread = " << compteur << endl;
}

int main() {
   thread t1(f);
   thread t2(f);
```

```
    t1.join();
    t2.join();
}
```

✓ Chacun affiche 1 car les variables sont séparées**.**

! Très utile pour la performance (évite les mutex).

3. La classe std::thread

Création d'un thread :

```
#include <thread>
void travail() {
    cout << "Thread en cours\n";
}

int main() {
    thread t(travail);
    t.join();
}
```

join() vs detach()

Méthode	Signification
join()	attendre la fin du thread
detach()	thread indépendant → « en arrière-plan »

Exemple detach() :

```
thread t(fonction);
t.detach();
```

⚠ Le programme continue même si le thread n'a pas terminé → danger si ressources locales.

4. Synchronisation : mutex

Un **mutex** (*mutual exclusion*) limite l'accès à une ressource partagée :

```
mutex m;

void f() {
    lock_guard<mutex> lock(m);
    // section critique
}
```

Outil essentiel : lock_guard

- verrouille au début
- libère automatiquement à la fin (RAII)

5. Sémaphore vs Mutex

Mécanisme	Rôle
Mutex	protection d'une ressource (exclusion mutuelle)
Sémaphore	limite le nombre d'accès à une ressource (compteur)

Mutex

- 1 seul accès → binaire (locked/unlocked)

Sémaphore (C++20 : std::counting_semaphore)

- compte courant = nombre de threads autorisés

6. Variables de condition (condition_variable)

Permet à un thread d'attendre un événement**,** par exemple une ressource disponible.

Exemple classique : producteur / consommateur

```
#include <condition_variable>
#include <queue>
mutex m;
condition_variable cv;
queue<int> q;
bool disponible = false;

void producteur() {
   unique_lock<mutex> lock(m);
   q.push(10);
   disponible = true;
   cv.notify_one();     // réveiller un consommateur
}

void consommateur() {
   unique_lock<mutex> lock(m);
   cv.wait(lock, []{ return disponible; });

   cout << "Consommation : " << q.front();
   q.pop();
}
```

7. Futures, promises, async

- std::promise produit une valeur
- std::future reçoit la valeur

Exemple :

```
promise<int> p;
future<int> f = p.get_future();

thread t([&p] { p.set_value(42); });

cout << f.get();   // attend la valeur
t.join();
```

8. async : thread simplifié

std::async démarre automatiquement un thread pour exécuter une fonction.

```
#include <future>

int travail() {
   return 7 * 6;
}

int main() {
   future<int> f = async(travail);
   cout << f.get();
}
```

✓ Gère thread + future automatiquement

✓ Pas besoin de join()

9. Les opérations et types atomiques

std::atomic<T> permet des opérations **indivisibles** sans mutex.

```
#include <atomic>

atomic<int> compteur{0};

void f() {
   compteur++;    // opération atomique
}
```

Types atomiques :

- atomic<bool>

- atomic<int>
- atomic<long>
- atomic_flag
- atomic<uint32_t> etc.

Pourquoi utiliser les atomiques ?

- Très rapides
- Évitent les mutex dans de nombreux cas
- Fondamentaux pour le lock-free programming

10. Verrous intelligents

lock_guard

lock_guard<mutex> lock(m);

unique_lock

Plus flexible (unlock/relock possible) :

unique_lock<mutex> lk(m);

lk.unlock();

lk.lock();

scoped_lock (C++17)

Verrouille plusieurs mutex :

scoped_lock lock(m1, m2);

11. Exemple complet : compteur partagé sécurisé

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
int compteur = 0;
mutex m;
void incrementer() {
   for (int i = 0; i < 10000; ++i) {
      lock_guard<mutex> lock(m);
      compteur++;
   }
}

int main() {
   thread t1(incrementer);
```

```
    thread t2(incrementer);

    t1.join();
    t2.join();

    cout << compteur;
}
```

12. Bonnes pratiques en multithreading

- Utiliser RAII (lock_guard, unique_lock)
- Éviter detach() sauf dans des cas particuliers
- Préférer std::async pour tâches parallèles
- Minimaliser les sections critiques
- Préférer les atomiques en cas d'accès simple
- Ne jamais partager des objets non thread-safe (ex : std::cout) sans mutex