Formation C++

PAR ALAIN CARIOU, MAI 2024

I – Le multithreading

Les applications multitâches

- En programmation, les **applications multitâches** sont des applications pouvant exécuter plusieurs tâches en **simultanée**.

- Ces tâches peuvent être divisées soit entre plusieurs **processus**, soit entre plusieurs **fils d'exécutions** (**thread**). Chaque processus ayant sa propre mémoire physique et étant constitué d'un ou plusieurs threads.

- Cela permet d'exécuter des tâches beaucoup plus **rapidement** qu'une application n'utilisant qu'un seul processus et qu'un seul fil d'exécution.

Présentation des threads

- En C++, le multithreading dispose d'un support intégré depuis **C++11** et peut désormais s'effectuer à partir de la classe ***std::thread*** utilisable à partir du header ***<thread>***.

- Les **threads** vont chacun représenter une partie du programme dans lequel du code va être **exécuté en parallèle**. En général, un programme dispose d'un thread principal et d'un ou plusieurs threads secondaires.

- Tous les threads partagent la **même zone mémoire** mais ils disposent chacun **d'une pile d'appels différente**.

Utiliser un thread

- Le constructeur d'un **thread** s'appelle avec ***std::thread*** et prend en paramètre **une fonction** (pouvant être une lambda) qui va s'exécuter dans le thread et **les paramètres** de cette fonction.

- L'appel à la méthode ***join()*** d'un thread permet de s'assurer que celui-ci se termine avant la fin du programme. Sinon cela pourrait occasionner une erreur.

- La méthode ***joinable()*** permet de s'avoir si le thread est actif ou s'il a fini d'exécuter sa tâche et peut ainsi être « joint ». Cette méthode renvoie un booléen.

Exemple d'utilisation de thread - 1

```cpp
int main()
{
    std::vector<std::string> array = {"Hello", "World", "chaussette", "lorem", "lion"};

    std::thread t1([]() {
        for (int i = 0; i < 10; ++i)
        {
            std::cout << "i : " << (i * 2) << " ";
        }
    });
    std::thread t2(add_sequence, 10, 20);
    std::thread t3(display_vector, array);
    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

Exemple d’utilisation de thread - 2

```cpp
int add_sequence(int start, int end)
{
    int result = 0;

    for (int i = start; i < end; i++)
    {
        result += i;
        std::cout << "add_sequence : " << result << " ";
    }
    std::cout << std::endl;
    return result;
}

void display_vector(std::vector<std::string> array)
{
    for (int i = 0; i < array.size(); i++)
    {
        std::cout << "tableau[" << i << "] : " << array[i] << " ";
    }
    std::cout << std::endl;
}
```

La méthode swap

- La méthode **swap** de ***<thread>*** surcharge l'algorithme ***std::swap()*** et permet d'échanger l'état de deux threads. Exemple :

```
std::thread t1();
std::thread t2();

std::cout << "Avant thread 1 id : " << t1.get_id() << std::endl;
std::cout << "thread 2 id = " << t2.get_id() << std::endl;

t1.swap(t2);

std::cout << "Apres thread 1 id : " << t1.get_id() << std::endl;
std::cout << "Apres thread 2 id : " << t2.get_id() << std::endl;

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

Points essentiels à retenir

- Un programme peut exécuter différentes tâches en parallèle grâce aux **threads**.
- Les threads accèdent tous à la **même zone mémoire** mais ont des piles d'accès différents.
- La déclaration d'un objet **std::thread** prend en paramètre une fonction et les arguments de cette fonction.
- Afin d'éviter des erreurs, on utilise la méthode **join()** pour attendre que chaque thread ait fini de s'exécuter et ait rejoint le fil d'exécution principale.
- L'utilisation des processus et des threads peut être **différent selon le système d'exploitation** utilisé.

II – La gestion des zones de risques

Les zones de risques

- Il arrive parfois que plusieurs threads aient besoin d'accéder **au même objet ou à la même zone mémoire**. Dans ce cas, cela peut créer des erreurs et des bugs dans le programme.

- On appel ces éléments des **zones de risques**, **ressources critiques** ou encore **critical section**.

- Afin d'éviter les conflits, on peut utiliser un **mécanisme de synchronisation** à travers un objet : les **mutex**. Ces derniers vont protéger les objets partagés en cas d'accès simultané par plusieurs threads.

Les différents types de mutex

- Il existe plusieurs types de **mutex** :
  - **std::mutex** est le mutex classique.
  - **std::recursive_mutex** est un mutex pouvant être verrouillé plusieurs fois sur un même fil d'exécution.
  - **std::timed_mutex** est un mutex pouvant être déverrouillé après un certain temps.
  - **std::recursive_timed_mutex** est un mutex pouvant être verrouillé plusieurs fois et pouvant être déverrouillé après un certain temps.

Utiliser un mutex

- Un **mutex** peut être verrouillé par un thread grâce à sa méthode ***lock()*** afin de protéger l'accès à une ressource. A partir de ce moment, il appartient à ce thread.

- Il pourra ensuite être déverrouillé grâce à sa méthode ***unlock()*** pour libérer l'accès à la ressource.

- Néanmoins cela ne peut se faire que par le thread qui le possède ! Une fois libérer, il peut être acquis par n'importe quel autre thread.

Le verrouillage des ressources

- Afin d'éviter qu'un **mutex** ne verrouille une ressource déjà verrouillée par un autre mutex, il est possible d'utiliser la méthode ***try_lock()*** :

  - Dans ce cas, si le mutex n'est pas verrouillé alors le thread appelant le verrouille.
  - Si le mutex est verrouillé par un autre thread, la fonction renvoie *false* et le thread appelant poursuit son exécution.
  - Si le mutex est verrouillé par le même thread qui appelle cette fonction alors un **deadlock** survient.

Exemple d'utilisation de mutex

```cpp
std::mutex mtx;

void display(std::vector<std::string> array)
{
    mtx.lock();
    for (int i = 0; i < array.size(); i++)
    {
        std::cout << array[i] << " ";
    }
    mtx.unlock();
    std::cout << std::endl;
}

int main()
{
    std::vector<std::string> fruits = {"fraise", "banane", "papaye", "mandarine", "citron"};
    std::vector<std::string> animal = { "chat", "chien", "lion", "poisson", "lapin" };

    std::thread t1(display, fruits);
    std::thread t2(display, animal);
    t1.join();
    t2.join();
    std::cout << "Fin du programme" << std::endl;
    return 0;
}
```

L’objet lock_guard

- Une manière de gérer les **mutex** aisément peut se faire avec l’objet **lock_guard**.
- **Lock_guard** va essayer de garder le mutex toujours verrouillé.
- Lors de sa construction, le mutex sera verrouillé par le thread appelant, et lors de la destruction, le mutex sera déverrouillé.
- Ainsi, il garantit au mutex d’être **déverrouillé en cas d'exception**.

Exemple d’utilisation de lock_guard

```cpp
int g_i = 0;
std::mutex mtx;

void safe_increment()
{
    std::lock_guard<std::mutex> lock(mtx);
    g_i++;
    std::cout << "Thread g_i: " << g_i << " - " << std::this_thread::get_id() << std::endl;
}

int main()
{
    std::cout << "Main g_i : " << g_i << std::endl;
    std::thread t1(safe_increment);
    std::thread t2(safe_increment);
    t1.join();
    t2.join();
    std::cout << "Main g_i : " << g_i << std::endl;
}
```

Gérer les deadlocks

- Un **deadlock** survient lorsque un mutex essaye de **verrouiller un mutex déjà verrouillé**, que ce soit par lui-même ou un autre thread. Dans ce cas, aucun des deux threads ne peuvent continuer leur exécution.

- Ainsi il faut faire attention à l'ordre de verrouillage des mutex. Essayez de toujours les verrouiller dans le même ordre dans votre programme !

- Limitez au maximum le nombre de variables pouvant être partagées ! Cela simplifiera le multithreading.

Exemple de deadlock - 1

- Ici la ressource partagée est un entier *i*.
- Un deadlock peut se produire dans les fonctions *shared_cout_thread_even* et *shared_cout_thread_odd*.
- Deux mutex sont utilisés dans ces fonctions.

```
std::mutex mainMtx, mtx1, mtx2;

void count(int n)
{
    for (int i = 10 * (n - 1); i < 10 * n; i++) {
        if (n % 2 == 0)
            shared_cout_thread_even(i);
        else
            shared_cout_thread_odd(i);
    }
}

int main()
{
    std::thread t1(count, 1);  // 0-9
    std::thread t2(count, 2);  // 10-19
    std::thread t3(count, 3);  // 20-29
    std::thread t4(count, 4);  // 30-39
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}
```

Exemple de deadlock - 2

- Ainsi, ci-dessous on passe d'une potentielle situation de deadlock à une situation sécurisée.

```
void shared_cout_thread_odd(int i)
{
    std::lock_guard<std::mutex> g2(mtx2);
    std::lock_guard<std::mutex> g1(mtx1);
    std::cout << " " << i << " ";
}

void shared_cout_thread_even(int i)
{
    std::lock_guard<std::mutex> g1(mtx1);
    std::lock_guard<std::mutex> g2(mtx2);
    std::cout << " " << i << " ";
}
```

```
void shared_cout_thread_odd(int i)
{
    std::lock_guard<std::mutex> g1(mtx1);
    std::lock_guard<std::mutex> g2(mtx2);
    std::cout << " " << i << " ";
}

void shared_cout_thread_even(int i)
{
    std::lock_guard<std::mutex> g1(mtx1);
    std::lock_guard<std::mutex> g2(mtx2);
    std::cout << " " << i << " ";
}
```

Exécutez une tâche unique

- Une autre manière d'exécuter des instructions dans un thread passe par l'utilisation de **std::async**.

- Cette méthode prend une fonction en paramètre et l'exécute dans un thread immédiatement ou de manière différée en fonction du système. On peut aussi spécifier un de ces comportements avec **std::launch::async** ou **std::launch::deferred**.

- Il est ensuite possible de récupérer son résultat à travers un objet de type **std::future**.

Exemple d’utilisation de std::async

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

int calc(int a, int b) {
    std::this_thread::sleep_for(std::chrono::seconds(3));
    return a * b;
}

int main() {
    std::future<int> result = std::async(std::launch::async, calc, 6, 7);
    std::cout << "Resultat : 6 x 7 = " << result.get() << std::endl;
    return 0;
}
```

Points essentiels à retenir

- Les **mutex** servent à gérer les **zones critiques**.
- Ils peuvent **verrouiller et déverrouiller** des ressources grâce à leurs méthodes ***lock()*** et ***unlock()***.
- Mais attention lorsqu'une ressource est déjà verrouillé par un mutex, cela peut créer un **blocage** dans le programme : un **deadlock**.
- **lock_guard** permet de faciliter l'utilisation des mutex.
- Essayez de toujours verrouiller et déverrouiller les mutex dans le même ordre.
- Il est possible d'exécuter une tâche en arrière-plan dans un thread avec **std::async**.