Pcynlitx Platform has been developed by Erkam Murat Bozkurt.

A short introduction to C++ threads

In this document, it is assumed that the reader knows C++ programming rules and can use GNU compiler tool chain (gcc / g++). If you are a windows operating system user, you can use the minimal implementation of the gcc compiler "MinGV". The main purpose of this document is to give a basic introduction to the C++ threads (std::thread). In the next section, at first, the term of multi-threading will be explained briefly. If you already know what is multi-threading and why it is necessary, you can skip this section.

What is the definition of the thread ?

For each program executed by the computer, a memory area and some hardware resources are allocated by the operating systems. In computer science, the collection of these resources is defined as process. Each process have at least one independent flow of execution (or the program flow) which is named as thread. For a multi-core processor, multiple thread can be executed on real-time and they can share process resources. This situation led to optimal usage of the hardware resources of the computer. A simple illustration about the structure of multi-thread process has been given in the below figures.

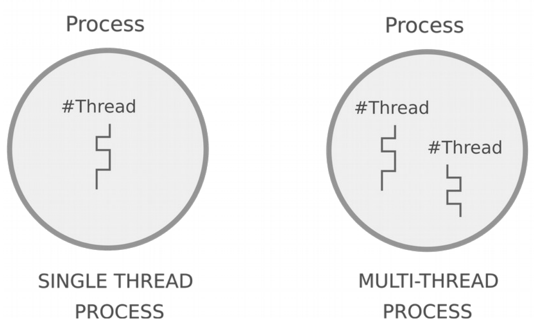


Figure: A sample illustration for multi-thread process

Why multi-threading is necessary ?

The problems of heat dissipation and energy consumption now limit the ability of hardware manufacturers to speed up chips by increasing their clock rate. This phenomenon has led to a major shift in computer architecture, where single-core CPUs have been replaced by CPUs consisting of a number of processing cores. Therefore, not surprisingly, in order to exploit the full potential of multi-core hardware, there is an unprecedented interest in parallelizing the computing tasks that were previously conducted in series.

This trend forces parallel programming upon the average programmer. Even though there are many way to achieve concurrency in software systems, in general-purpose software engineering practice, we have reached a point where one approach to concurrent programming gains popularity, namely, threads.

Introduction to std::thread

In order to learn C++ multi-threading, perhaps, the best starting point is POSIX Threads (pthreads). However, if you don't have too much time to learn multi-threading, this document gives you a short way and with the help of the examples given in below, the reader of the document can learn the usage C++ threads (std::thread) step by step. Before that, let we try to learn how std::thread codes are compiled and let we build and test the code given in below.

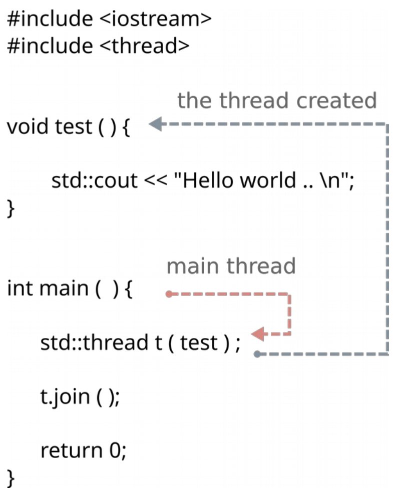
```
#include <iostream>
#include <thread>

void test(){
        std::cout << "Hello world .. \n";
}

int main(){
     std::thread t(test);
     t.join();
     return 0;
}
```

This is a simple multi-threaded hello world example. Now, let we assume that the name of the file in which the code is written is “sample.cpp”. You can easily compile the code with the following command.

```
~$ g++ -o sample sample.cpp -lpthread
```

Just as Pcynlitx, std::thread library is an implementation of the POSIX threads (pthread) library. Therefore, a link to the pthread library is performed with the linker option (-lpthread) on the command. Now, we can investigate the example given in above. In this sample code, the thread function is named as “test”. Actually, the thread function is a term which is used for the function that is executed by the threads created. The thread function only prints a sample word to the screen (“hello world ..”) In addition, the type of the object that creates the threads is “std::thread”. The name of the object which is an instance of the type std::thread is “t”. On the construction, this object takes the name of the function to be executed as its only parameter.

As in the case of single threaded applications, in this sample code, the flow of execution is started from the main thread (*the thread executes the main function*). When the execution of the main thread reaches the code line in which the thread object is constructed, a new thread has been created by the thread object. This situation has been illustrated in below figure.


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

void test ( ) {

      std::cout << "Hello world .. \n";
}

int main (  ) {

    std::thread t ( test ) ;

    t.join ( );

    return 0;
}
```

In the sample code, there is also a member function which is shown as “join”In practice, there are two types of thread. In the first type, the thread automaticly destroy itself when its execution is ended. This type is defined as detached thread. The creation of the detached threads differs from the main thread. The other type is joinnable. The joinable threads must be joined to the main thread and its existence is ended by the main thread. Therefore, the main thread has to wait the completion of the joinnable threads. The creation of both the detached thread and joinnable thread are shown in the following code lines.

```
std::thread  t(test);  → Joinnable thread creation ( It is used in the sample code )

std::thread  t(test).detach(); →  Detachable thread creation
```

If the main thread completes its execution before a joinnable thread, the thread created never ends and continues its existence as a zombie thread. Therefore, In order to wait the completion of the thread's execution, we have to use "join( )" member function of the thread object. In the code example, join member function is used just before the "return" command. Thus, the main thread waits the created thread. The command line is given in below.

```
t.join( ); → this code line joins the thread "t" to the main thread.
```

In this example, the new thread prints a simple string "Hello world" to the screen. Now, let we try to examine more complex problems of the multi-threading. Different from multi-process computing, although the threads have their own stack, they share the process heap memory. For instance, for a moment, let we assume that we have to create more than one thread and they have to process same variable staying on the heap memory. In this case, most probably, when the one of the threads tries to read the value of the variable, the other thread tries to set a new value to the the variable.

In other words, multiple threads attempt to modify the same variable at the same time, or one thread tries to read the value of a variable while another thread is modifying it. In the literature, this problem is defined as data races. Fortunately, the mutual exclusion offers a solution to this problem.

Mutual exclusion in std::thread

For the mutual exclusion, std::mutex class is used and the header of the mutex class is declerated as "mutex". Now, let we look at the following code example.

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

double sum = 0.0;

void increase_1(){

     for(int i=0;i<500;i++){

         sum = sum + 0.01;
     }
}

void increase_2(){

     for(int i=0;i<500;i++){

         sum = sum + 0.01;
     }
}

int main( ){

    std::thread t_increase_1(increase_1);

    std::thread t_increase_2(increase_2);

    t_increase_1.join();

    t_increase_2.join();

    std::cout << "\n sum :" << sum << "\n";

    return 0;
}
```

In this example, without using a mutex, the outcome can vary. The true result of the computations is of course “10”. But, you reads many different results such as “6.13”, “6.14”, “5.23” if you don’t use any mutex object in order to prevent data races. Because, the threads writes the values simultaneously and corrupts the computations of each others. If we use a mutex object, the code example can be written as :

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

std::mutex mt;
double sum = 0.0;

void increase_1(){

     for(int i=0;i<500;i++){
         mt.lock();
         sum = sum + 0.01;
         mt.unlock();
     }
}

void increase_2(){

     for(int i=0;i<500;i++){
         mt.lock();
         sum = sum + 0.01;
         mt.unlock();
     }
}

int main(){

    std::thread t_increase_1(increase_1);
    std::thread t_increase_2(increase_2);
    t_increase_1.join();
    t_increase_2.join();
    std::cout << "\n sum :" << sum << "\n";
    return 0;
}
```

In that case, the result of the computations is always “10”. The threads are never executes simultaneously the code segments which are between the “`mt.lock()`” and “`mt.unlock()`” member functions. However, it is always possible that something goes wrong in the process and an exception can be occur when the mutex is locked. In that case, the mutex never closed. In C++ threads, you can prefer more robust solutions for mutual exclusion. The first solution is lock guard.

Lock Guard object:

Lock guard is an object that manages a mutex object by keeping it always locked. The main idea of the lock guard object can be summarized as in the following.

The mutex object is locked by the calling thread upon the construction. The mutex object must be constructed on the scope of the thread function. In this way, the mutex is unlocked upon destruction and guarantees the mutex is properly unlocked in case an exception is thrown. On the other hand, you will need allocate yet another class instance for the guard every time you need to lock the mutex, as std::lock_guard has no member functions (*it may cause some big problems such as memory starvation* !). Therefore, lock guard is not suitable for repetitive lock-unlock operations such as in the case of this example. After these information, we can give a simple example for the usage of the lock_guard object.

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

std::mutex mu;

void test(){
     std::lock_guard < std::mutex > guard (mu);
     std::cout << "This is a test \n";
}

int main( ){
    std::thread t1 (test);
    std::thread t2 (test);
    t1.join( );
    t2.join( );
    return 0;
}
```

Unique Lock :

Just like the lock guard object, the unique lock object guarantees an unlocked status on destruction. However, unique lock class has member functions lock and unlock and they can be called arbitrarily. For instance, we can performed the mutex operations that are given in the previous example with unique lock object.

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

std::mutex mu;
double sum = 0.0;

void increase_1(){
     std::unique_lock<std::mutex> ul(mu);
     ul.unlock();
     for(int i=0;i<500;i++){
          ul.lock();
          sum = sum + 0.01;
          ul.unlock();
     }
}

void increase_2(){
     std::unique_lock<std::mutex> ul(mu);
     ul.unlock();
     for(int i=0;i<500;i++){
         ul.lock();
         sum = sum + 0.01;
         ul.unlock();
     }
}
```

```
int main(){

    std::thread t_increase_1(increase_1);

    std::thread t_increase_2(increase_2);

    t_increase_1.join();

    t_increase_2.join();

    std::cout << "\n sum :" << sum << "\n";

    return 0;
}
```

In this example, there is a point that needs more attention. This is the usage of the lock/unlock member function. In the example, the construction of the unique_lock object is shown as in the below code line.

```
std::unique_lock<std::mutex> ul(mu);
```

When the unique_lock object is constructed, it locks the mutex. If you try to luck again, a system error is occurr. Therefore, at first, you can unlock the mutex.

The usage of the condition variables in C++ threads:

The condition variables block the threads until a certain condition is occurred. In other words, Condition variables allow us to synchronize threads via notifications. Simultaneous access to the condition variables may leads a suspicious wake up. In order to prevent from simultaneous access, to the condition variables are always used with a mutex. When the condition variable is called, the mutex automatically unlocked. A sample usage of condition variables is given in the following code lines.

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id (int id) {

  std::unique_lock<std::mutex> lck(mtx);

  while (!ready){

      cv.wait(lck);
  }

  // ...
  std::cout << "thread " << id << '\n';
}

void go() {

   std::unique_lock<std::mutex> lck(mtx);

   ready = true;

   cv.notify_all();
}
```

```
int main ()
{
  std::thread threads[10];
  // spawn 10 threads:

  for (int i=0; i<10; ++i) {

       threads[i] = std::thread(print_id,i);
  }

  std::cout << "10 threads ready to race...\n";

  go();                       // go!

  for (int i=0;i<10;i++) {

       threads[i].join();
  }

  return 0;
}
```

In this example, each thread waits on the function call that is indicated as “cv.wait(lck);” until the main thread completes the execution of the for loop that spawns all of the threads and calls the function which is named as “go”.