The Code Examples Comparing C++ std::future-promise facility with Pcynlitx

The sample multithreading applications which are given in below serialize the execution of the critical sections and the critical sections are executed by the threads in an exactly same order in each running of the software. However, the coding sample for C++ std::threads requires experienced programmer and the complexity of the code increases exponentially depending of the parameters such as the number of the critical sections, the number of the functions which are executed and the thread number. The functions and the variables that are used on the C++ std::future-promise implementation are well known. Therefore, only the functions which are used on pcynlitx application have been introduced.

C++ STD THREAD IMPLEMENTATION

```
#include <iostream>
#include <vector>
#include <thread>
#include <future>
#include <barrier>
#include <condition_variable>

std::mutex mtx_1;
std::mutex mtx_2;

std::barrier sync_point_1(4);
std::barrier sync_point_2(4);

std::condition_variable cv_tp1;
std::condition_variable cv_tp2;

int thread_num = 8;

int thread_counter_1 = 0;

int thread_counter_2 = 4;


void function_1(int thread_id, std::shared_future<void> * futObj,

     std::promise<void> * promObj){

     std::unique_lock<std::mutex> mtx1(mtx_1);

     mtx1.unlock();


     mtx1.lock();

     if(thread_id == 0){

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

     mtx1.unlock();


     do{

         mtx1.lock();

         if(thread_id != thread_counter_1){

            cv_tp1.wait(mtx1);

            mtx1.unlock();
         }
         else{
                mtx1.unlock();

                break;
         }
}while(thread_counter_1<4);

 mtx1.lock();

 if(thread_id == thread_counter_1){

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

    std::cout << "\n CRITICAL SECTION -1";
 }

 thread_counter_1++;

 cv_tp1.notify_all();

 mtx1.unlock();


 mtx1.lock();

 if(thread_id == 3){

    thread_counter_1 = 0;
 }

 mtx1.unlock();


 sync_point_1.arrive_and_wait();

 if(thread_id == 0){

    promObj[0].set_value();

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

 futObj[1].wait();


 do{

     mtx1.lock();

     if(thread_id != thread_counter_1){

         cv_tp1.wait(mtx1);

         mtx1.unlock();
     }
     else{
           mtx1.unlock();

           break;
     }

 }while(thread_counter_1<4);


 mtx1.lock();

 if(thread_id == thread_counter_1){

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

    std::cout << "\n CRITICAL SECTION -3";
 }

 thread_counter_1++;

 cv_tp1.notify_all();

 mtx1.unlock();
```

```
     mtx1.lock();
     if(thread_id == 3){
        thread_counter_1 = 0;
     }
     mtx1.unlock();

     if(thread_id == 3){
         std::cout << "\n";
         promObj[2].set_value();
     }
}

void function_2(int thread_id, std::shared_future<void> * futObj,
     std::promise<void> * promObj){
     std::unique_lock<std::mutex> mtx2(mtx_2);
     mtx2.unlock();
     futObj[0].wait();
     do{
        mtx2.lock();
        if(thread_id != thread_counter_2){
          cv_tp2.wait(mtx2);
          mtx2.unlock();
        }
        else{
              mtx2.unlock();
              break;
        }
     }while(thread_counter_2<8);

     mtx2.lock();
     if(thread_id == thread_counter_2){
        std::cout << "\n thread_id:" << thread_id;
        std::cout << "\n CRITICAL SECTION -2";
     }
     thread_counter_2++;
     cv_tp2.notify_all();
     mtx2.unlock();

     mtx2.lock();
     if(thread_id == 7){
        thread_counter_2 = 4;
     }
     mtx2.unlock();
```

```cpp
      sync_point_2.arrive_and_wait();


      if(thread_id == 7){

         std::cout << "\n";

         promObj[1].set_value();
      }

      futObj[2].wait();


      do{

          mtx2.lock();

          if(thread_id != thread_counter_2){

             cv_tp2.wait(mtx2);

             mtx2.unlock();
          }
          else{
                 mtx2.unlock();

                 break;
          }

     }while(thread_counter_2<8);


      mtx2.lock();

      if(thread_id == thread_counter_2){

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

         std::cout << "\n CRITICAL SECTION -4";
      }

      thread_counter_2++;

      cv_tp2.notify_all();

      mtx2.unlock();


      mtx2.lock();

      if(thread_id == 7){

         std::cout << "n";
      }

      mtx2.unlock();

}


int main()
{

    std::promise<void> promiseObj[3];
      // The promise object created.

    std::shared_future<void> futureObj[3];

    futureObj[0] = promiseObj[0].get_future();

    futureObj[1] = promiseObj[1].get_future();

    futureObj[2] = promiseObj[2].get_future();
```

```
   // The value which will be obtained on the
   // future is assigned to the future object

   std::thread threads[8];

   for(int i=0;i<thread_num/2;i++){

       threads[i] = std::thread(function_1,i,futureObj,promiseObj);
   }

   for(int i=thread_num/2;i<thread_num;i++){

       threads[i] = std::thread(function_2,i,futureObj,promiseObj);
   }

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

       threads[i].join();
   }

   return 0;
 }
```

PCYNLITX IMPLEMENTATION

```
 #include "MT_Library_Headers.h"

 int main(int argc, char ** argv){

     pcynlitx::Thread_Server Server;

     for(int i=0;i<4;i++){

        Server.Activate(i,function_1);  // The Member function which crates the threads
     }

     for(int i=4;i<8;i++){

        Server.Activate(i,function_2);
     }

     for(int i=0;i<8;i++){

         Server.Join(i);                // The Member function joining the threads
     };

     std::cout << "\n\n The end \n\n";

     return 0;
 }

 void function_1(pcynlitx::thds * thread_data){

      pcynlitx::TM_Client Manager(thread_data,"function_1");

      // The manager object is responsible from the management of the thread flows


      Manager.lock();

      int thread_id = Manager.Get_Thread_Number();

      // The member function which gets the number of the thread executing the function

      // The number is set by the programmer to the threads on the thread creation

      Manager.unlock();
```

```
Manager.lock();

if(thread_id == 0){

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

Manager.unlock();


Manager.start_serial(0,4,thread_id);   // This function (start_serial) serialize the
                                       threads having the numbers varies between 0 to 3

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

std::cout << "\n CRITICAL SECTION -1";

Manager.end_serial(0,4,thread_id);     // This function (end_serial) ends serial execution
                                       of threads having the numbers varies between 0 to 3



Manager.lock();

if(thread_id == 3){

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

Manager.unlock();




Manager.function_switch("function_1","function_2");

 // This function (function_switch) stops every threads executing the "function_1" and
 transfers the program executions to the threads executing the "function_2". In other
 words, it switch execution flows from "function_1" to "function_2". The execution of the
 program flows is transfers to the point in which the same member function call is
 performed on the "function_2". If the threads executing the function_2 are currently
 blocked, they are rescued from blockage. More specifically, the cybernetic management
 system checks the status of the threads which are performed a call to the same member
 function on the function_2 from its database. If they are blocked by the time a call to
 the member function is performed, the cybernetic thread management system releases the
 threads executing "function_2.



Manager.lock();

if(thread_id == 0){

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

Manager.unlock();




Manager.start_serial(0,4,thread_id);

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

std::cout << "\n CRITICAL SECTION -3";

Manager.end_serial(0,4,thread_id);





Manager.lock();
```

```
     if(thread_id == 3){

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

     Manager.unlock();


     Manager.function_switch("function_1","function_2");


     Manager.Exit();
}

void function_2(pcynlitx::thds * thread_data){

     pcynlitx::TM_Client Manager(thread_data,"function_2");


     Manager.lock();

     int thread_id = Manager.Get_Thread_Number();

     Manager.unlock();


     Manager.function_switch("function_2","function_1");

     Manager.lock();

     if(thread_id == 4){

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

     Manager.unlock();


     Manager.start_serial(4,8,thread_id);

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

     std::cout << "\n CRITICAL SECTION -2";

     Manager.end_serial(4,8,thread_id);


     Manager.lock();

     if(thread_id == 8){

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

     Manager.unlock();


     Manager.function_switch("function_2","function_1");


     Manager.start_serial(4,8,thread_id);

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

     std::cout << "\n CRITICAL SECTION -4";

     Manager.end_serial(4,8,thread_id);



     Manager.lock();
```

```
      if(thread_id == 8){

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

      Manager.unlock();


      Manager.reset_function_switch("function_1","function_2");


      Manager.Exit();
 }
```

THE OUTPUTS OF THE PROGRAMS (THE OUTPUTS ARE SAME FOR BOTH APPLICATION)

```
thread_id:0
CRITICAL SECTION -1
thread_id:1
CRITICAL SECTION -1
thread_id:2
CRITICAL SECTION -1
thread_id:3
CRITICAL SECTION -1


thread_id:4
CRITICAL SECTION -2
thread_id:5
CRITICAL SECTION -2
thread_id:6
CRITICAL SECTION -2
thread_id:7
CRITICAL SECTION -2

thread_id:0
CRITICAL SECTION -3
thread_id:1
CRITICAL SECTION -3
thread_id:2
CRITICAL SECTION -3
thread_id:3
CRITICAL SECTION -3

thread_id:4
CRITICAL SECTION -4
thread_id:5
CRITICAL SECTION -4
thread_id:6
CRITICAL SECTION -4
thread_id:7
CRITICAL SECTION -4

The end
```