PCYNLITX **Pcynlitx is an innovative open-source multi-thread computing platform**

Pcynlitx Platform has been developed by Erkam Murat Bozkurt.

Advantages of Pcynlitx

In this document, it is assumed that the reader is known the basics of the C++ multi-threading and the usage of the Pcynlitx tools.

The big picture of the pcynlitx:

First of all, a new programming paradigm is used on pcynlitx and we can easily say that the pcynlitx is a particular application of these new paradigm for multi-thread computing. The main purpose of the pcynlitx project is to develop a kind of software which interacts its user and the whole programming process is carried out based on this interaction. In this case, as a meta-programming mechanism, the pcynlitx software behaves like a separate intelligent actor that helps the software development process and reduces the complexity of the software development process significantly.

In essence, the pcynlitx collect information about the context of the software to be developed and build some intelligent agents (class library). These intelligent control mechanism provides many advantages to the software developers. For instance, the control tools that are used by the programmer can receive information from the user and steers the threads based on this information.

1. Absolute control over the threads (scheduling the threads) :

This section of the document is shown on the front page of the web site. If you read this section before, you can skip this section and you can pass to the next topic on this document (*2. Synchronization without mutex)*

The execution order and the priority of the threads reflect the term of scheduling. More specifically, it is a decision about which thread will run and which thread will be suspended in a particular time interval. In practice, the operating system schedules the threads. In other words, the programmer can not control the scheduling of the threads on the applications in which a standard multi-threading library such as C++ threads is used. Lets take a look the figure that is given in below.

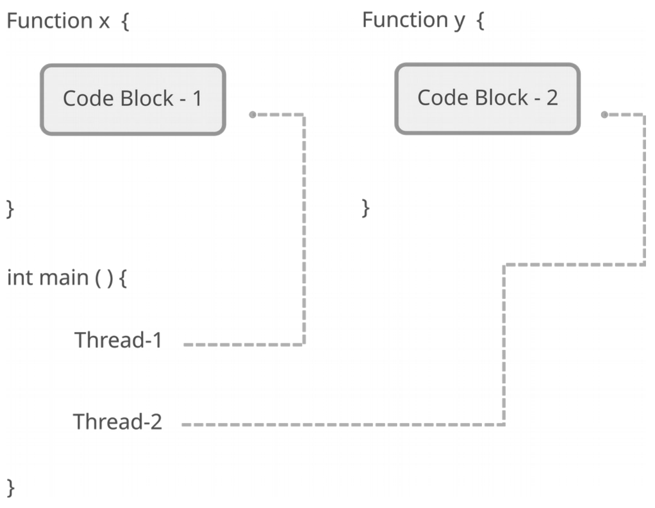

Figure: A sample illustration for the execution order of the threads

Let we look at the figure.

Which code block is executed first ? Can the programmer determine this?

For the standard threading libraries, the answer is no !

On the standard threading libraries, the programmer can not determine exactly which code block will be executed in which order. However, in pcynlitx, the programmer can determine how the threads interleave. This power of the pcynlitx comes from the "*application specific multi-threading library*" construction process. The term "application specific multi-threading library" refers such a library that has written for a specific application. In other words, on the library, there are some special tools which are generated for a certain class or application and the library is constructed based on some application specific information such as how many thread will be created in run-time and what will the name of the thread functions be.

With Pcynlitx, you will have a threading library that is specialized according to your needs. The control tools that are build by the Pcynlitx has a memory and can receive information from the process. In other words, the control tools of the Pcynlitx communicate the process after each operation that is performed on the process. This communication can be considered as a feedback mechanism which provides situation awareness. Therefore, you can give a unique number for each thread and you can control the threads by means of these unique numbers. More specifically, you can stop and rescue each threads with its unique numbers. This property provides more control over the threads. For instance, let we look at the illustrative program depicted in the below figure.

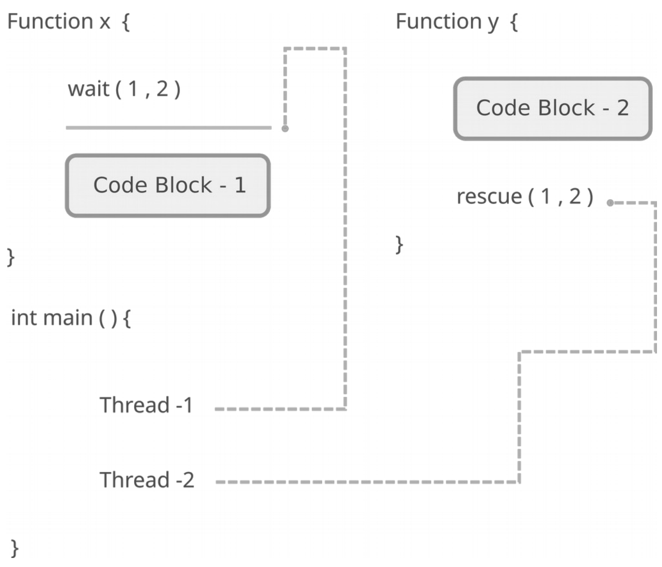

Figure: How Pcynlitx controls the threads

In the figure, the function call depicted as "wait (1, 2)" stops "thread-1." The same function call also records that the "thread-1" will be rescued by the "thread-2" later on. This means that the pcynlitx also defines the relation between the threads and reduces non-determinism of the multi-threading. In the program, "thread-1" always waits on the code line which wait(1,2) function call is performed until its corresponding rescue function is called by "thread-2" . Therefore, the "Code Block -2" is always executed before the "Code Block -1". More specifically, the programmer can determine exactly which thread will be blocked by which thread and how and in which place the threads will be rescued later on. Standard threading libraries don't have such a control over the threads and thanks to the autonomous thread management system, pcynlitx has many other advance properties which the other threading libraries doesn't have. In order to learn how pcynlitx produces an application specific library and how provides such a control over the threads, please look the section which is named as documents in web site of pcynlitx.

2. Synchronization without using any mutex (Lock free synchronization) :

The condition variables is used in critical sections. Therefore, the condition variables must be used with mutexes. On the contrary, in pcynlitx, the synchronization primitives receives the number of the threads calling and act more consciously. Lets do it more clear. the wait member function of condition variables does not average which thread is calling and stop each thread that calls. Therefore, in order to prevent synchronization errors, the threads must be call wait member function of the condition variables one by one. This is the reason why mutexes are used with condition variables. For instance, a simple code can be used in order to demonstrate the usage of condition variables.

```
std::mutex mu;

std::condition_variable cond;

void function ( ) {

     mu.lock( );

     while (!condition( )) {

            cond.wait(mu);
     }

     mu.unlock( );
}
```

In pcynlitx, you can stop each thread with its particular number. This property brings many advantages. The one of them is logical separation of the threads. The thread which is not specified in the parameters of the member function can not call the member functions of the synchronization tools. In that case, there is no need for mutual execution. In pcynlitx, the code given in above can be written such as in below.

```
void function(thds * arg) {

     TM_Client Manager(arg,”function”);

     int Thread_Number = Manager.Get_Thread_Number();

     if(Thread_Number % 2 == 0){

        while(!condition( )){

           Manager.wait(Thread_Number);
        }
     }
}
```

In that case, if the number of the thread is odd, thread is blocked. Otherwise, the thread is not effected from the wait member function and continues its normal execution.

3. There is no any drawback coming from the usage of the condition variables (spurious wake ups):

There is another problem in usage of condition variables. A thread might be awoken from its waiting state even though no thread signaled the condition variable. This condition is named as spurious wake ups. On the other hand, in C++, it is also possible to make condition wake up completely predictable. Spurious wake ups may sound strange, but on some multiprocessor systems, making condition wake up completely predictable might substantially slow all condition variable operations.

4. If you wish, you can use std::mutex class with Pcynlitx

Pcynlitx has its own mutex functions. However, if you wish, you can use std::mutex class as well. Especially, “Lock_guard” and “Unique_Lock” template classes provides exceptions safety. In below, some examples has been given.

```
#include <iostream>
#include <MT_Library_Headers.h>
#include <mutex>

std::mutex mu;

void test(thds * arg){

     TM_Client Manager(arg,”test”);

     std::lock_guard < std::mutex > guard (mu);

     std::cout << “This is a test \n”;

     Manager.Exit();
}

int main( ){

    Thread_Server Server;

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

        Server.Activate(i,test);
    }

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

        Server.Join(i);
    }

    return 0;
}
```

In the following example, the usage of unique_lock template class has been shown.

```
#include <iostream>
#include <MT_Library_Headers.h>
#include <mutex>

std::mutex mu;
double sum = 0.0;

void increase_1(thds * arg){

     TM_Client Manager(arg,”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(thds * arg){

     TM_Client Manager(arg,”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( ){

    Thread_Server Server;

    Server.Activate(0,test);

    Server.Activate(1,test);

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

        Server.join(i);
    }

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

    return 0;
}
```

5. The threads can be controlled according to the name of their function routines

Pcynlitx has some advanced properties which are not seen on the other threading tools. For instance, pcynlitx has some control function that use the name of the function routine which is executed. More specifically, you can stop each thread that executes the same thread function and you can determine how the threads will be rescued later on. Some typical illustration is given in below. There are also completed code examples are given in the documents section of the web site.

```
void thread_function(thds * arg){

     TM_Client Manager(arg,”thread_function”);

     // Code lines

     Manager.wait(“thread_function”);   -> Blocks each thread executing the “thread_function”

     Manager.Exit( );
}
```

In this illustration, each thread that executes the function named as “thread_function” is blocked until all of the threads that execute the “thread_function performs the same function call “Manager.wait(“thread_function”). After all of the threads executing the “thread_function” performs a call to the member function indicating as Manager.wait(“thread_function”), all of the threads will be rescued.

There is a second member function type that stops all of the threads that executes the same thread function. The threads that executes the same thread function can be also rescued by another thread which executes some other thread function. In the below code lines, the usage of this member function has been depicted.

```
void thread_function_1(thds * arg ){

     TM_Client Manager(arg,”thread_function_1”);

     // Code lines

     Manager.wait(“thread_function_1”,3); -> Stops all of the threads executing “thread_function_1”

     /* At the same time, manager object records that the thread which is numbered as “3” will

        rescue the threads waitg some other place. */

     Manager.Exit();
}

void thread_function_2(thds * arg){

     TM_Client Manager(server);

     // Code lines

     Manager.rescue(“thread_function_1”,3); -> Rescues the thread waiting on the above wait point.

     Manager.Exit();
}


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

    Thread_Server Server;

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

        Server.Activate(i,thread_function_1); -> Thread  [0] and Thread [1] activated.
    }

    for(int i=2;i<4;i++){

        Server.Activate(i,thread_function_2); -> Thread  [2] and Thread [3] activated.
    }

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

        Server.Join(i);
    }

    return 0;
}
```

6. Deadlock protection :

Most of the time, the deadlocks are caused by nondeterministic scheduling of the threads. In contrast, in pcynlitx, you can directly program the execution order of the threads. In most cases, the programmer must determine how the threads interleave and how the threads will be rescued later on when the threads are stopped. For instance, when you use a member function such as “wait(1,3)”, in fact you are declared that the thread [1] stops in here and the thread[3] rescued the thread [1] on some other place.