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

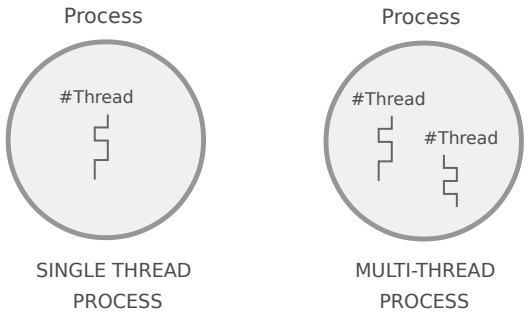
Pcynlitx Platform has been developed by Erkam Murat Bozkurt.

A brief Introduction to POSIX Threads (pthreads):

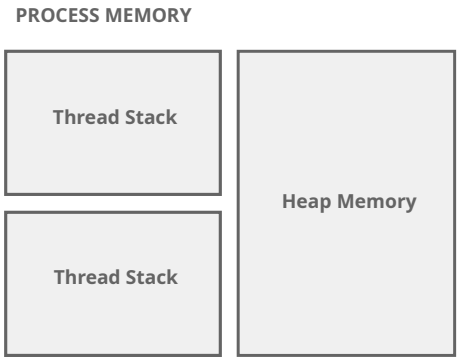
POSIX simply is an abbreviation of the term “Portable Operating System Interface for UNIX like operating systems”. To be able to emphasize what is POSIX thread programming, at first we need to understand what is multi-threading.

Introduction to multi-threading:

It is a well known fact that a particular resources is allocated for each program which is executed by the operating systems. These resources can be classified as memory areas or the accessibility of some hardware resources such as hard disks and CPUs. The collection of these resources is called as process. In fact, the processes are simply known as running programs by the regular computer users. Beside to this, Each running program has at least one independent flow of execution which is called as thread. The computer programs may have more than one thread. and these kind of programs are named as multi-threaded programs.



In multi-threaded programs, the threads share the process memory area. In thread creation, a particular memory area is allocated for each thread created and this memory area is used by the thread until its termination. In other words, each thread has its own memory area and this area is named as thread's stack.



POSIX Thread Programming

Each thread has a unique ID number given by the operating system. In pthread programming, the type of variable which holds the ID numbers of the threads created is “pthread_t”. In pthread programming, like the other libraries, the threads are created by the main thread. In addition, each thread which created by the main thread executes a function routine (*thread functions*) and the trace of these function routines must be defined in a standard form. The only parameter of a function routines which are used in pthread programming must be a void pointer.

Beside to this, the return value of the function routines must be a void pointer as well. Therefore, the trace of the function routines that are used in the pthread programming is in the form given in below.

```
void * function(void * arg){

}
```

Figure -1: Thread Function deceleration on pthread programming.

In C/C++ programming, void pointers are special type pointers that hold the address of the variables. You can pass the address of any variable to the threads created by means of the arguments of the thread functions. In pthread programming, the arguments are passed by reference to the thread functions.

The reason of these restrictions is to be able to provide optimal usage of the process memory. From the declaration of the thread function on the Figure-1, it can be easily realized that “pass by reference” method is used in order to pass variables to the thread functions. In fact, theoretically, in C++ programming, there are two possible methods which can be used in order to pass a variable to the newly created thread’s function. These methods are named as -“pass by value” and “pass by reference”. However, even though “pass by reference” is more difficult than “pass by value”, “pass by reference” method is preferred when the arguments are passed into the thread functions. For a moment, let we consider the opposite of this and let we suppose that the typical “pass by value” concept is used in order to pass the arguments into the parameter of the thread’s start routine. In that case, on each thread creation, a copy of the argument is passed to the new thread. Thus, even if the parameter’s value is changed on the inside of the thread function, it does not effect the original argument. In a first look, this concept can be seen as a good idea for the isolation of thread executions, but in general, this is not really a good practice and this choose can cause some devastating consequences such as memory starvation. This drawback of the “pass by value” method comes from the architectural limitations of multi-threaded programs.

In a multi-threaded process, each thread has its own stack and on each thread’s creation, a new memory area is allocated to the new thread from the process's heap memory. Beside to this, the variables which are passed to a new thread’s stack is valid until the thread’s termination. Hence, if the typical “pass by value” concept is used in order to pass the arguments to the new threads, there may be many copies of the same variables on the process memory. In this case, if a large data structure must be passed to the many number of thread simultaneously, the same data structure will be copied to the each thread’s stack and occupy significantly large process memory area unnecessarily. Therefore, the typical pass by value mechanism is not a memory efficient way especially when the number of thread is respectively bigger. Hence, on POSIX-Thread programming, rather than pass by value, pass by reference mechanism is used in order to pass variables to the new threads.

Thread creation on pthread programming:

In pthread programming, pthread_create( ) function is used for the thread creation. In below, the definition of the pthread_create( ) function is given. In the first look, the definition of the pthread_create( ) function may look confusing. However, its practical usage is much more easy. The trace of the pthread_create( ) function is shown in below.

```
int pthread_create( pthread_t * thread, const pthread_attr_t *attr,
                      void * (*start_routine) (void *), void * arg );
```

The input parameter of this function has been listed in below:

`pthread_t *`	:	A pointer to a pthread_t variable, in which the thread ID of the new thread is stored.
`const pthread_attr_t *`	:	A pointer to a thread attribute object. For a standard thread, it is NULL.
`void * (*) (void *)`	:	A pointer to the thread function. This is an ordinary function pointer. In practice, the name of the thread function is a function pointer that holds the address of the thread function and it is passed to this parameter.
`void *`	:	A thread argument value of type void* whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

In order to examine the usage of the pthread_create function, let we look at the code example given in below.

```
#include <iostream>
#include <pthread.h>

void * printCharacter (void * args){

       std::cout << "\n\n THREAD HAS BEEN CREATED AND STARTED TO RUN! ...\n";

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

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

       std::cout << "\n\n THREAD FINISHED ITS JOB....!";

       std::cout << "\n\n PRESS ANY KEY TO CONTINUE";

       std::cin.get();
}

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

    pthread_t thread;  /* thread is the variable that holds the ID number of the thread created */

    pthread_create(&thread,NULL,printCharacter,NULL);

    std::cin.get();

    return 0;
}
```

Let we consider that this sample code is recorded as “pthread_sample.cpp” and we learn how we can compile this code. This code can be compile with the GCC compiler command given in below.

```
~$ g++ -o pthread_sample.exe pthread_sample.cpp -lpthread
```

g++	:	GCC compiler directive for C++ codes
-o	:	Specifies that the name of the executable file is entered explicitly and if the compiler receives that command, it receives the following word as the name of the executable file.
pthread_sample.exe		The name of the executable file
pthread_sample.cpp		The name of the source file
-lpthread		Linker command for the pthread library

In the example code, as indicated before, the first parameter of the pthread_create( ) function is the address of the variable that holds the ID number of the thread to be created. The thread to be created is a standard thread and therefore the second parameter is NULL. The third parameter of the function is the name of the thread function to be executed and the last parameter is the variable that will be passed to the stack of thread to be created. In this example, the thread only prints an information to the screen and thus, this parameter is also NULL.

Passing data to threads:

In POSIX thread programming, data is passed to the threads by means of the last parameter of the pthread_create( ) function. The type of the parameter is void address and therefore, you can pass the address of any variable over the last parameter of the pthread_create( ) function to the threads. However, if more value or variable are passed to the threads, a structure is defined and the address of the structure is passed to the threads. Now, let we look at the simple example given in below.

```
#include <iostream>
#include <string>
#include <pthread.h>


struct Datas {

   std::string name;

   int age;

   float loan;
};


void * print(void * args){

       Datas * threadDatas;

       threadDatas = (Datas *)args;

       std::cout << "\n NAME:" << threadDatas->name;

       std::cout << "\n AGE: " << threadDatas->age;

       std::cout << "\n LOAN:" << threadDatas->loan;

       std::cin.get();
}


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

    pthread_t thread;

    Datas personDatas = {"Erkam",33,100.25};

    pthread_create(&thread,NULL,print,&personDatas);

    std::cin.get();

    return 0;
}
```

Joinning the threads:

If there is no any protection system, the main thread may finish its job and the process may end when the other threads continue their executions. In these case, the other threads can not be ended and continue their existence as zombie threads. This is also means that the memory areas which are allocated for the threads are hold by the thread as well. In order to prevent this situation, the main thread must wait the other threads.
Let we consider that the sample program given in below.

```
#include <iostream>
#include <pthread.h>

void * printInfo(void * args){

       std::cout << “\n Thread has been created successfuly”;
}
```

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

      pthread_t thread;

      pthread_create(threadPointer,NULL,functionPointer,NULL);

      return 0;
}
```

In this program, the main thread may complete its execution before the thread created. If this is the case, the thread created continues its existence as zombie thread.

In pthread programming, pthread_join( ) function is used for joining the threads. The pthread_join( ) member function takes two parameters. The first one is the name of the thread to be waited. The second parameter is the return value of the threads to be waited. If the threads does not have any return value, the second parameter must be NULL. If pthread_join( ) function is used, the main thread waits on the point in which the pthread_join( ) function is called until the thread which is specified completes its execution.

The trace of the pthread_jon( ) function is shown in below.

```
void pthread_join( pthread_t arg, void ** arg );
```

The input parameter of this function has been listed in below.

`pthread_t` : The variable that holds the ID number of the thread created.

`void **` : The address of the thread's return value (it will be explained soon)

If we use the pthread_join( ) function such as in below, the main thread waits until the thread created completes its execution.

```
#include <iostream>
#include <pthread.h>

void * printInfo(void * args){

       std::cout << "\n Thread has been created successfuly";
}

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

      pthread_t thread;

      pthread_create(threadPointer,NULL,functionPointer,NULL);

      pthread_join(thread,NULL);

      return 0;
}
```

In the sample code, the first parameter of the pthread_join( ) function is the variable that holds the ID number of the thread. The thread just prints a string to the screen. Therefore, there is no any return value of the thread and the second parameter is NULL.

Thread Return Values:

A thread can exit explicitly by calling pthread_exit function. This function may be called from within the thread function or from some other function called directly or indirectly by the thread function. The argument to pthread_exit( ) is the thread's return value. In both cases, if the second argument which you pass to pthread_join( ) is non-null and you want to return a value from a thread, the thread's return value will be placed in the location pointed

to by that argument (*the second argument of the* pthread_join( ) *function*). The thread return value, like the thread argument, is of type " void * ". For instance, if you want to pass back a single int or other small number, you can do this easily by casting the value to " void * " and then casting back to the appropriate type after calling pthread_join.

```
void pthread_exit(void * retval);
```

The value pointed to by "retval" should not be located on the calling thread's stack, since the contents of that stack (*the thread's stack*) are undefined after the thread terminates. For instance, the address of the variable which is passed to the thread is defined on the main thread's stack and a data structure that is defined on the main thread can be used for inter-thread communication. A simple example makes everything more clear.

```
#include <iostream>
#include <pthread.h>

struct Thread_Data {

       double x;

       double y;

       double return_value;
};


void * compute_add(void * arg) {

       Thread_Data * Data_Pointer = ( Thread_Data * )arg;

       Data_Pointer->return_value = Data_Pointer->x + Data_Pointer->y;

       pthread_exit(arg);
}

int main ( ) {

    pthread_t thread;

    Thread_Data myData;

    myData.x = 2.0;

    myData.y = 3.0;

    pthread_create(&thread,NULL,compute_add,( void * )&myData);

    pthread_join(thread,(void ** )&myData);

    std::cout << "\n return_value : " << myData.return_value << "\n";

    return 0;
}
```

The output of the program is in below.

```
Return value :  5
```

However, you can return the values from a thread with a more simple way. In fact, in below code example, you don't have to return any value from the thread. Because, the data structure which is used defined on the main thread and you can use the last member of the data structure on the main thread directly. The same code example can be written as in below and gives exactly the same result.

```
#include <iostream>
#include <pthread.h>

struct Thread_Data {

	double x;

	double y;

	double return_value;
};

void * compute_add(void * arg) {

	Thread_Data * Data_Pointer = ( Thread_Data * )arg;

	Data_Pointer->return_value = Data_Pointer->x + Data_Pointer->y;

	pthread_exit(0);
}

int main ( ){

    pthread_t thread;

    Thread_Data myData;

    myData.x = 2.0;

    myData.y = 3.0;

    myData.return_value = 0.0;

    pthread_create(&thread,NULL,compute_add,(void * )&myData);

    pthread_join(thread,NULL);

    std::cout << "\n return_value : " << myData.return_value << "\n";

    return 0;
}
```

Therefore, after a data structure has been defined on the main thread, you can use any element of that data structure as a communication mechanism between the threads. In the last example, the pthread_exit( ) function is used in order to end the thread's execution and set the exit status of the thread.

Determining the ID number of the threads:

The pthread_self( ) function returns the ID of the calling thread. With the help of this function, the programmer can receive an information from the operating system and obtain which thread performs a particular job or which code segment is executed by which thread. By this way, in order to synchronize the threads, the programmer can distinguish the threads from each others. Actually, thread synchronization is a decision about which thread will be blocked and which thread will continue its execution in a particular code segment. A simple example makes everything more clear. Let we consider that two thread executes the same thread function. However, one of them reads the file and the other thread waits until the other threads reads the file.

```
#include <iostream>
#include <pthread.h>
#include <fstream>           // For file operations
#include <string>
#include <unistd.h>          // Unix standard header for Sleep function
```

```
pthread_t read_thread;       // this variable hold the ID number of the thread which reads the file.

pthread_t write_thread;      // this variable hold the ID number of the thread which writes in to the file.

bool read_status = false;

std::fstream DataFile;
std::string read_data;
std::string file_data =  "";
std::string write_data = "Hello world";

void * FileOperations( void * arg){

       int * thread_data = (int * )(arg);

       if(*thread_data == 1){

           read_thread = pthread_self( );   // The ID number of the thread which reads the file.
       }
       else {

           write_thread = pthread_self( );  // The ID number of the thread which writes into the file.
       }

       if(pthread_self( ) == write_thread ){

          while(!read_status){

                sleep(5);
          }

          std::cout << "\n Data readed. Please press enter";

          std::cin.get();
       }

       if(pthread_self( ) == read_thread ){

          DataFile.open("test",std::ios::in);

          while (!DataFile.eof()) {

                 DataFile >> read_data;

                 file_data = file_data + read_data;
          }

          std::cout << "\n Readed data:" << file_data;

          DataFile.close();

          read_status = true;
       }

       if(pthread_self( ) == write_thread ){

          DataFile.open("test",std::ios::out);

          DataFile <<  write_data;

          DataFile.close();
       }

       pthread_exit(0);
};
```

```
int main( ){

    pthread_t read;

    pthread_t write;

    int read_determiner = 1, write_determiner = 0;

    pthread_create(&read,NULL,FileOperations,(void *)&read_determiner);

    pthread_create(&write,NULL,FileOperations,(void *)&write_determiner);

    pthread_join(read,NULL);

    pthread_join(write,NULL);

    std::cout << "\n\n The end of the program .. \n\n";

    return 0;
}
```

Shared memory programming and mutual exclusion:

In most cases, the threads have to share a variable or memory location. If the threads only read the data location, there is no problem. However, if the threads try to manipulate the same data simultaneously, this situation may lead a race condition. In other words, when a thread reads the data, the other thread tries to write a new value at the same memory location and the data which is shared between the threads is corrupted. In order to prevent data races, a mutual exclusion (mutex) is used in multi-thread programming. The term mutex is a short for “mutual exclusion locks” and a mutual exclusion is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

To create a mutex, you simply declare a variable that represents your mutex and initializes it with a special symbolic constant. The mutex is of type “pthread_mutex_t” and is demonstrated as follows:

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
```

Now that you have a mutex, you can lock and unlock it to create your critical section. This is done with the “pthread_mutex_lock” and “pthread_mutex_unlock” API functions. Finally, you can destroy an existing mutex using pthread_mutex_destroy . The prototypes of these functions are given as follows:

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

A simple code example that shows the usage of the pthread_mutex function is given in below. In this example, the threads computes the sum of the numbers. Therefore, a global variable which is named as “sum” has been defined. On each computation, the threads update the sum of the numbers. However, if a mutex does not use, the threads corrupt each others data and data race is occurred.

```
#include <iostream>
#include <pthread.h>

pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;

double sum = 0.0;
```

```
void increase_1(void * arg){

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

         pthread_mutex_lock(&myMutex);

         sum = sum + 0.01;

         pthread_mutex_unlock(&myMutex);
     }

     pthread_exit(0);
}


void increase_2(void * arg){

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

         pthread_mutex_lock(&myMutex);

         sum = sum + 0.01;

         pthread_mutex_unlock(&myMutex);
     }

     pthread_exit(0);
}

int main(){

    pthread_t threads[2];

    pthread_create(&threads[0],NULL,increase_1,NULL);

    pthread_create(&threads[1],NULL,increase_1,NULL);

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

        pthread_join(threads[i],NULL);
    }

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

    return 0;
}

THE OUTPUT OF THE PROGRAM :

sum : 10


THE OUTPUT OF VARIES ON EACH EXECUTION OF THE PROGRAM IF A MUTEX DOES NOT USE:

sum : 5.92

sum : 6.99

sum : 5.06
```

The codes which is between the pthread_mutex_lock and pthread_mutex_unlock functions is called as the critical section. In shared memory programming, the threads must reach the critical sections exclusively. In other words, the critical sections must be executed by only one thread in a particular time interval.

Condition variables:

The condition variables block the threads until a certain condition is occurred. More specifically, the condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread modifies a shared variable (the *condition*), and notifies the condition_variable.

In other words, condition variables allow us to synchronize threads via notifications. However, simultaneous access to the condition variables may leads a suspicious wake up. In such case, threads wakes up without any notification and this situation may lead significant synchronization errors. 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.

The pthreads library provides a number of functions supporting condition variables. These functions provide condition variable creation, waiting, signaling, and destruction. The condition variable library functions are presented as follows:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime );
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

The condition variables can be initialized with two different way. The first way is static initialization.

```
pthread_cond_t count = PTHREAD_COND_INITIALIZER;
```

The other way dynamic initialization. In dynamic initialization, condition variable is created dynamically and it is valid in a particular code segment. The initialization is performed with pthread_condition_init( ) function and the prototype of the pthread_condition_init( ) is given in below.

```
int pthread_cond_init(pthread_cond_t * cv, const pthread_condattr_t *cattr);
```

A sample usage of condition variables is given in the following code lines. In the example, the condition variable has been initialized statically.

```
#include <iostream>
#include <pthread.h>

pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

bool ready = false;

void * print_id(void * arg) {

       pthread_mutex_lock(&myMutex);

       while (!ready){

              pthread_cond_wait(&cond,&myMutex);
       }

       std::cout << "thread " <<  pthread_self( ) << '\n';

       pthread_mutex_unlock(&myMutex);

       pthread_exit(0);
}
```

```
void go() {

   pthread_mutex_lock(&myMutex);

   ready = true;

   pthread_cond_broadcast(&cond);

   pthread_mutex_unlock(&myMutex);
}

int main ()
{
  pthread_t threads[10];
  // spawn 10 threads:

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

       pthread_create(&threads[i],NULL,print_id,NULL);
  }

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

  go();                       // go!

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

       pthread_join(threads[i],NULL);
  }

  return 0;
}
```

The output of the program is given in below.

```
10 threads ready to race...
thread 140585122285312
thread 140585105499904
thread 140585097107200
thread 140585130678016
thread 140585088714496
thread 140585080321792
thread 140585113892608
thread 140585071929088
thread 140585063536384
thread 140585055143680
```

In this example, each thread waits on the function call that is indicated as "`pthread_cond_wait(&cond,&myMutex);`" 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".

THREAD ATTRIBUTES

In order to synchronize the threads, you must also determine how the execution of the threads will be ended. In pthread programming, a thread may have two different attributes. A thread may be created as a joinable thread (*the default*) or as a detached thread. A thread is automatically joinable when using the default attributes of pthread_create . If the attribute for the thread is defined as detached, then the thread can't be joined (because it's detached from the creating thread). A detached thread ends its own execution when its procedures completed. In the previous sections, it has been clearly shown that how the pthread_join( ) function is used.

In this section, we will examine how a detached thread is programmed. Recall that pthread_create accepts an argument that is a pointer to a thread attribute object. If you pass a null pointer, the default thread attributes are used to configure the new thread. However, you may create and customize a thread attribute object to specify other values for the attributes.

To specify customized thread attributes, you must follow these steps:

1. Create a **pthread_attr_t** object.The easiest way is simply to declare an automatic variable of this type.
2. Call **pthread_attr_init** , passing a pointer to this object.This initializes the attributes to their default values.
3. Modify the attribute object to contain the desired attribute values.
4. Pass a pointer to the attribute object when calling pthread_create.
5. Call **pthread_attr_destroy** to release the attribute object.The **pthread_attr_t** variable itself is not deallocated; it may be reinitialized with **pthread_attr_init**.

A code example is given in below.

```
#include <pthread.h>

void* thread_function (void* thread_arg)
{
    std::cout << "\n Hello world";
}


int main ()
{
    pthread_attr_t attr;

    pthread_t thread;

    pthread_attr_init(&attr);

    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    pthread_create(&thread, &attr, &thread_function, NULL);

    pthread_attr_destroy(&attr);

    std::cout << "\n Press enter .. \n";

    std::cin.get();

    /* No need to join the second thread. */

    return 0;

}
```

In this example, if std::cin.get( ) member function is not used, the main thread may complete its execution before the thread created and, the string "hello world" may not be printed to the screen. Therefore, in order to synchronize the threads, a function which stops the execution of the main thread must be used.