# Practical Guide To Pthread Programming in C++

*By Swaminathan Bhaskar*
*10/22/2008*

## Terminology:

* ***Process*:** A process is an instance of a program executable (ex: vi) that is identified as a unique entity in the operating environment (ex: linux). It has its own address space and system state information (ex: handles to system resources). For two processes to communicate with one another, they need to use some form of Inter Process Communication (IPC) mechanism (ex: pipes, sockets, shared memory).

* ***Thread*:** A thread is a path of execution that is identified by a unique entity within a process. A process can have multiple threads of execution and they all share the same process address space and system state information. Since all the threads within a process share the same address space, they can communicate with each other directly using program objects.

* ***Mutex*:** A mutex (short form for mutual exclusion lock) is a programming construct that is used to control access to a shared resource by multiple threads within a process.

* ***Condition Variable*:** A condition variable is a notification mechanism by which one or more threads may be alerted of a condition they are interested in.

## Prerequisites:

* Linux 2.6.x kernel
* GNU C++ compiler - g++

## Pthread Fundamentals:

To use Posix thread (pthread), we need to include the following header:

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

To successfully compile the C++ code with Posix thread (pthread), we need to link in the Posix thread (pthread) library by specifying the following library option to g++:

```
-lpthread
```

To create a Posix thread (pthread), we use the following function:

```
int pthread_create(
      pthread_t *id,
      const pthread_attr_t *attr,
      void *(*exec)(void*),
      void *arg
);
```

The arguments of ***pthread_create*** are as follows:

| | |
|---|---|
| ***id*** | *Input pointer that will contain the thread identifier on successful return from the function* |
| ***attr*** | *Input pointer to a structure that provides additional parameters for creating a custom thread. The default is an unbound, non-detached thread with a stack size of 8M and default priority* |
| ***exec*** | *Input pointer to a global function that is to be executed in the thread of execution* |
| ***arg*** | *Input pointer to the argument that is to be passed to the function to be executed by the thread* |

The default non-detached thread is allocated storage by the system that needs to be released on termination. To wait for a non-detached thread to terminate and reclaim the allocated storage and get the termination status, we use the following function:

```
int pthread_join(
      pthread_t id,
      void **status
);
```

The arguments of ***pthread_join*** are as follows:

| | |
|---|---|
| ***id*** | *Input that specifies the Thread Id* |
| ***status*** | *Input pointer to a pointer that on successful return will contain the termination status of the specified thread* |

To identify threads within a process, each thread is assigned a unique identifier. This is also know as the Thread Id. To get the identifier of a thread, use the following function:

```
pthread_t pthread_self(void);
```

Having seen the basic functions in the pthread library, now we are ready to see some action. The following is a simple Hello World style example:

```
/* Hello.cpp */

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

using namespace std;

// Code path we want the thread to execute
void *welcome(void *arg) {
      cout << "Id: " << pthread_self() << endl;
      cout << "Welcome to Pthreads Programming" << endl;
      return (void *)0;
}

// ----- Main -----
```

```
int main() {
	int ret;
	int *stat;
	pthread_t tid;

	// Create a thread within the process to execute welcome
	if ((ret = pthread_create(&tid, NULL, welcome, NULL)) != 0) {
		cout << "Error creating thread: " << strerror(ret) << endl;
		exit(1);
	}

	cout << "Created Thread " << tid << endl;

	pthread_join(tid, (void **)&stat);

	cout << "Thread " << tid << " terminated, Status = " << stat << endl;

	exit(0);
}
```

We will save the above C++ code in a file called Hello.cpp. To compile the above C++ program, execute the following command in the shell:

```
$ g++ -o Hello Hello.cpp -lpthread
```

Running the executable ***Hello*** will create a thread that executes the function welcome.

Along with the included source, is a basic make file Makefile, which will allow us to compile the provided code. For example, to compile all the code, execute the following command in the shell:

```
$ make
```

To compile just ***Hello.cpp***, execute the following command in the shell:

```
$ make Hello
```

Posix thread (pthread) is a C library. In order to use it in an object-oriented language like C++, we will create an object wrapper that will abstract and hide away the intricacies of Posix thread (pthread) and present a simple interface for the user. This abstraction will be encapsulated in the class named ***Thread*** in all the following examples. The following is the definition of the ***Thread*** class:

```
class Thread {
	private:
		pthread_t _id;

		// Prevent copying or assignment
		Thread(const Thread& arg);
		Thread& operator=(const Thread& rhs);

	protected:
		bool started;
		void *arg;
```

```
            static void *exec(void *thr);

      public:
            Thread();
            virtual ~Thread();
            unsigned int tid() const;
            void start(void *arg = NULL);
            void join();
            virtual void run() = 0;
};
```

As indicated earlier, Posix thread is a C library and the function that creates a thread, ***pthread_create***, needs a pointer to a global function. In the ***Thread*** class, the static member function ***exec*** will serve as the global function. The ***Thread*** class is an abstract class because of the pure virtual function ***run***. The member function ***run*** is internally called by the static member function **exec**.

The following is a simple example that uses the abstract class ***Thread***. This example also shows, how to pass arguments to a thread function:

```
/* Thread.cpp */

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

using namespace std;

// ----- Class : Thread -----

class Thread {
      private:
            pthread_t _id;

            // Prevent copying or assignment
            Thread(const Thread& arg);
            Thread& operator=(const Thread& rhs);

      protected:
            void *arg;

            static void *exec(void *thr);

      public:
            Thread();
            virtual ~Thread();
            void start(void *arg);
            void join();
            virtual void run() = 0;
};

Thread::Thread() {
      cout << "Thread::Thread()" << endl;
}

Thread::~Thread() {
      cout << "Thread::~Thread()" << endl;
```

```
}

void Thread::start(void *arg) {
      int ret;
      this->arg = arg;

      /*
       * Since pthread_create is a C library function, the 3rd argument is
       * a global function that will be executed by the thread. In C++, we
       * emulate the global function using the static member function that
       * is called exec. The 4th argument is the actual argument passed to
       * the function exec. Here we use this pointer, which is an instance
       * of the Thread class.
       */
      if ((ret = pthread_create(&_id, NULL, &Thread::exec, this)) != 0) {
            cout << strerror(ret) << endl;
            throw "Error";
      }
}

void Thread::join() {
      // Allow the thread to wait for the termination status
      pthread_join(_id, NULL);
}

// Function that is to be executed by the thread
void *Thread::exec(void *thr) {
      reinterpret_cast<Thread *> (thr)->run();
}

class MyThread : public Thread {
      public:
            /*
             * This method will be executed by the Thread::exec method,
             * which is executed in the thread of execution
             */
            void run() {
                  cout << "Welcome to Pthreads Programming in C++ with arg "
<< *((int *)arg) << endl;
            }
};

// ----- Main -----

int main() {
      int x = 10;
      MyThread thr;
      thr.start(&x);
      thr.join();
}
```

To create a non-default thread (ex: detached thread), one has to provide the appropriate thread attributes at the thread creation time. Thread attributes are encapsulated in a structure ***pthread_attr_t***.

The following example demonstrates the use of thread attributes to create a detached thread with a stack size of ***PTHREAD_STACK_MIN*** which is the minimum size to start a thread:

```
/* Attributes.cpp */

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

using namespace std;

// ----- Class : Thread -----

class Thread {
      private:
            pthread_t _id;
            pthread_attr_t _attr;

            // Prevent copying or assignment
            Thread(const Thread& arg);
            Thread& operator=(const Thread& rhs);

      protected:
            void *arg;

            static void *exec(void *thr);

      public:
            Thread(bool detach, int size);
            virtual ~Thread();
            void start(void *arg);
            void join();
            virtual void run() = 0;
};

/*
 * The 1st argument specifies whether we want to create a detached thread.
 * The 2nd argument specifies the stack size of the thread
 */
Thread::Thread(bool detach, int size) {
      cout << "Thread::Thread()" << endl;
      int ret;
      if ((ret = pthread_attr_init(&_attr)) != 0) {
            cout << strerror(ret) << endl;
            throw "Error";
      }
      if (detach) {
            if ((ret = pthread_attr_setdetachstate(&_attr,
PTHREAD_CREATE_DETACHED)) != 0) {
                  cout << strerror(ret) << endl;
                  throw "Error";
            }
      }
      if (size >= PTHREAD_STACK_MIN) {
            if ((ret = pthread_attr_setstacksize(&_attr, size)) != 0) {
                  cout << strerror(ret) << endl;
                  throw "Error";
            }
      }
}

Thread::~Thread() {
      cout << "Thread::~Thread()" << endl;
```

```
        int ret;
        if ((ret = pthread_attr_destroy(&_attr)) != 0) {
              cout << strerror(ret) << endl;
              throw "Error";
        }
}

void Thread::start(void *arg) {
        int ret;
        this->arg = arg;

        /*
         * Since pthread_create is a C library function, the 3rd argument is
         * a global function that will be executed by the thread. In C++, we
         * emulate the global function using the static member function that
         * is called exec. The 4th argument is the actual argument passed to
         * the function exec. Here we use this pointer, which is an instance
         * of the Thread class.
         */
        if ((ret = pthread_create(&_id, &_attr, &Thread::exec, this)) != 0) {
              cout << strerror(ret) << endl;
              throw "Error";
        }
}

void Thread::join() {
        // Allow the thread to wait for the termination status
        pthread_join(_id, NULL);
}

// Function that is to be executed by the thread
void *Thread::exec(void *thr) {
        reinterpret_cast<Thread *> (thr)->run();
}

class MyThread : public Thread {
        public:
              MyThread(bool detach, int size) : Thread(detach, size) {}

              /*
               * This method will be executed by the Thread::exec method,
               * which is executed in the thread of execution
               */
              void run() {
                    cout << "Welcome to Pthreads With Attributes in C++ with
arg " << *((int *)arg) << endl;
              }
};

// ----- Main -----

int main() {
        int x = 10;
        // Create a detached thread with minimum stack size
        MyThread thr(true, PTHREAD_STACK_MIN);
        thr.start(&x);
}
```

## Pthread Synchronization:

### Mutex:

When multiple threads access and manipulate a shared resource (ex: a variable for instance), the access to the shared resource needs to be controlled through a lock mechanism so that only one thread is allowed access to the shared resource at any point of time while the other threads are waiting to gain access the shared resource. In Posix thread (pthread) this is enforced by using a synchronization primitive called mutex lock.

A mutex lock object is encapsulated in a structure ***pthread_mutex_t***. Prior to use, an instance of the mutex lock object needs to be initialized, which allocates storage for the internal attributes on the object. This is done using the following function:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mattr
);
```

The arguments of ***pthread_mutex_init*** are as follows:

| | |
|---|---|
| ***mutex*** | *Input pointer to an instance of mutex lock object* |
| ***mattr*** | *Input pointer to a structure that provides additional parameters for creating a custom mutex. This argument is usually specified as NULL in most cases* |

Before one can discard an initialized instance of mutex lock object, the storage space allocated for the internal attributes needs to be deallocated. This is done using the following function:

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

To lock a mutex, use the following function:

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

To unlock a mutex, use the following function:

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

For thread synchronization using mutex, we will create an object wrapper that will abstract and hide away the intricacies of mutex locking mechanism and present a simple interface for the user. This abstraction will be encapsulated in the class named ***Lock*** in all the following examples. The following is the definition of the ***Lock*** class:

```
class Lock {
      protected:
            pthread_mutex_t mutex;

            // Prevent copying or assignment
            Lock(const Lock& arg);
```

```
            Lock& operator=(const Lock& rhs);

      public:
            Lock();
            virtual ~Lock();
            void lock();
            void unlock();
};
```

The ***Lock*** class is simple enough and does not need explanation.

The following is a simple example that uses the abstract classes ***Thread*** and ***Lock***. This example demonstrates the use of mutex lock to guard access to a shared resource (class Shared) by two threads:

```
/* Mutex.cpp */

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

using namespace std;

// ----- Class : Thread -----

class Thread {
      private:
            pthread_t _id;

            // Prevent copying or assignment
            Thread(const Thread& arg);
            Thread& operator=(const Thread& rhs);

      protected:
            bool started;
            void *arg;

            static void *exec(void *thr);

      public:
            Thread();
            virtual ~Thread();
            unsigned int tid() const;
            void start(void *arg = NULL);
            void join();
            virtual void run() = 0;
};

Thread::Thread() : started(false) {
      cout << "Thread::Thread()" << endl;
}

Thread::~Thread() {
      cout << "Thread::~Thread()" << endl;
}

unsigned int Thread::tid() const {
      return _id;
}
```

```
// Uses default argument: arg = NULL
void Thread::start(void *arg) {
      int ret;
      if (!started) {
            started = true;
            this->arg = arg;

            /*
             * Since pthread_create is a C library function, the 3rd
             * argument is a global function that will be executed by
             * the thread. In C++, we emulate the global function using
             * the static member function that is called exec. The 4th
             * argument is the actual argument passed to the function
             * exec. Here we use this pointer, which is an instance of
             * the Thread class.
             */
            if ((ret = pthread_create(&_id, NULL, &Thread::exec, this)) !=
0) {
                  cout << strerror(ret) << endl;
                  throw "Error";
            }
      }
}

void Thread::join() {
      // Allow the thread to wait for the termination status
      pthread_join(_id, NULL);
}

// Function that is to be executed by the thread
void *Thread::exec(void *thr) {
      reinterpret_cast<Thread *> (thr)->run();
}

// ----- Class : Lock -----

class Lock {
      protected:
            pthread_mutex_t mutex;

            // Prevent copying or assignment
            Lock(const Lock& arg);
            Lock& operator=(const Lock& rhs);

      public:
            Lock();
            virtual ~Lock();
            void lock();
            void unlock();
};

Lock::Lock() {
      pthread_mutex_init(&mutex, NULL);
}

Lock::~Lock() {
      pthread_mutex_destroy(&mutex);
}
```

```
void Lock::lock() {
      pthread_mutex_lock(&mutex);
}

void Lock::unlock() {
      pthread_mutex_unlock(&mutex);
}

// ----- Class : Shared -----

class Shared {
      private:
            int _cnt;

      public:
            Shared() : _cnt(0) {};
            int getCnt();
            void incCnt();
};

int Shared::getCnt() {
      return _cnt;
}

void Shared::incCnt() {
      _cnt++;
}

// ----- Class : MyThread -----

class MyThread : public Thread {
      private:
            Lock *_lck;

      public:
            MyThread() {
                  _lck = new Lock();
            }

            ~MyThread() {
                  delete _lck;
            }

            /*
             * This method will be executed by the Thread::exec method,
             * which is executed in the thread of execution
             */
            void run() {
                  Shared *d = reinterpret_cast<Shared *> (arg);
                  for (;;) {
                        _lck->lock();
                        d->incCnt();
                        cout << "Thread<" << tid() << ">: " << d->getCnt()
<< endl;
                        _lck->unlock();
                        sleep(1);
                  }
            }
```

```
};

// ----- Main -----

int main() {
      Shared data;
      MyThread thr1, thr2;
      thr1.start(&data);
      thr2.start(&data);
      thr1.join();
      thr2.join();
}
```

## Condition Variables:

Now we will explore condition variables. A condition variable is another synchronization mechanism where one or more threads will relinquish a lock and wait for an event of interest to occur. When an event of interest happens, another thread will notify the waiting threads of the occurrence of the interested event. A Condition variable is always used in conjunction with a mutex lock.

Producer-Consumer problem is a classic example that use condition variables.

A condition variable object is encapsulated in a structure ***pthread_cond_t***. Prior to use, an instance of the condition variable needs to be initialized, which allocates storage for the internal attributes on the object. This is done using the following function:

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

The arguments of ***pthread_cond_init*** are as follows:

| | |
|---|---|
| ***cond*** | *Input pointer to an instance of condition variable object* |
| ***cattr*** | *Input pointer to a structure that provides additional parameters for creating a custom condition variable. This argument is usually specified as NULL in most cases* |

Before one can discard an initialized instance of condition variable object, the storage space allocated for the internal attributes needs to be deallocated. This is done using the following function:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

To wait on a condition or event, use the following function:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex
);
```

Notice that the above function needs a mutex lock to be used in conjunction with the condition variable. Call to this function causes the thread to release the mutex lock and block on the condition until the condition or event occurs.

To notify a condition or event, use the following function:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

For using condition variables, we will create an object wrapper that will abstract and hide away the intricacies of the mechanism and present a simple interface for the user. This abstraction will be encapsulated in the class named ***Condition*** in all the following example. The following is the definition of the ***Condition*** class:

```
class Condition : public Lock {
      protected:
            pthread_cond_t cond;

            // Prevent copying or assignment
            Condition(const Condition& arg);
            Condition& operator=(const Condition& rhs);

      public:
            Condition();
            virtual ~Condition();
            void wait();
            void notify();
};
```

A condition variable occurs in conjunction with a mutex lock and hence the ***Condition*** class extends the ***Lock*** class. The ***Condition*** class is simple enough and does not need explanation.

The following is a simple example that uses the abstract classes ***Thread***, ***Lock***, and ***Condition***. This example demonstrates the classic Producer-Consumer problem using three threads:

```
/* Condition */

#include <iostream>
#include <list>
#include <stdlib.h>
#include <pthread.h>

using namespace std;

// ----- Class : Thread -----

class Thread {
      private:
            pthread_t _id;

            // Prevent copying or assignment
            Thread(const Thread& arg);
            Thread& operator=(const Thread& rhs);

      protected:
            bool started;
            void *arg;
```

```
            static void *exec(void *thr);

      public:
            Thread();
            virtual ~Thread();
            unsigned int tid() const;
            void start(void *arg = NULL);
            void join();
            virtual void run() = 0;
};

Thread::Thread() : started(false) {
      cout << "Thread::Thread()" << endl;
}

Thread::~Thread() {
      cout << "Thread::~Thread()" << endl;
}

unsigned int Thread::tid() const {
      return _id;
}

// Uses default argument: arg = NULL
void Thread::start(void *arg) {
      int ret;
      if (!started) {
            started = true;
            this->arg = arg;

            /*
             * Since pthread_create is a C library function, the 3rd
             * argument is a global function that will be executed by
             * the thread. In C++, we emulate the global function using
             * the static member function that is called exec. The 4th
             * argument is the actual argument passed to the function
             * exec. Here we use this pointer, which is an instance of
             * the Thread class.
             */
            if ((ret = pthread_create(&_id, NULL, &Thread::exec, this)) !=
0) {
                  cout << strerror(ret) << endl;
                  throw "Error";
            }
      }
}

void Thread::join() {
      // Allow the thread to wait for the termination status
      pthread_join(_id, NULL);
}

// Function that is to be executed by the thread
void *Thread::exec(void *thr) {
      reinterpret_cast<Thread *> (thr)->run();
}

// ----- Class : Lock -----
```

```
class Lock {
      protected:
            pthread_mutex_t mutex;

            // Prevent copying or assignment
            Lock(const Lock& arg);
            Lock& operator=(const Lock& rhs);

      public:
            Lock();
            virtual ~Lock();
            void lock();
            void unlock();
};

Lock::Lock() {
      pthread_mutex_init(&mutex, NULL);
}

Lock::~Lock() {
      pthread_mutex_destroy(&mutex);
}

void Lock::lock() {
      pthread_mutex_lock(&mutex);
}

void Lock::unlock() {
      pthread_mutex_unlock(&mutex);
}

// ----- Class : Condition -----

class Condition : public Lock {
      protected:
            pthread_cond_t cond;

            // Prevent copying or assignment
            Condition(const Condition& arg);
            Condition& operator=(const Condition& rhs);

      public:
            Condition();
            virtual ~Condition();
            void wait();
            void notify();
};

Condition::Condition() {
      pthread_cond_init(&cond, NULL);
}

Condition::~Condition() {
      pthread_cond_destroy(&cond);
}

void Condition::wait() {
      pthread_cond_wait(&cond, &mutex);
}
```

```
void Condition::notify() {
      pthread_cond_signal(&cond);
}

// ----- Class : Job -----

// An interface that represents work to be done
class Job {
      public:
            virtual void work(const Thread& arg) = 0;
};

typedef Job * JobPtr;

// ----- Class : PrintJob -----

// A concrete implementation of the Job interface that represents
// print work to be done
class PrintJob : public Job {
      public:
            void work(const Thread& arg);
};

void PrintJob::work(const Thread& arg) {
      int tm = rand() % 5;
      cout << "PrintJob::work(), Thread = " << arg.tid() << ", Wait = " <<
tm << endl;
      sleep(tm);
}

// ----- Class : JobQueue -----

/*
 * A queue data structure which represnts a collection of Jobs to be handled
 * by consumers
 */
class JobQueue {
      private:
            list<JobPtr> _queue;
            Condition *_cnd;

      public:
            JobQueue();
            virtual ~JobQueue();
            void enqueue(const Thread& arg, JobPtr jp);
            JobPtr dequeue(const Thread& arg);
};

JobQueue::JobQueue() {
      _cnd = new Condition();
}

JobQueue::~JobQueue() {
      delete _cnd;
}

void JobQueue::enqueue(const Thread& arg, JobPtr jp) {
      _cnd->lock();
```

```
        _queue.push_back(jp);
        cout << "Thread = " << arg.tid() << " added job" << endl;
        _cnd->notify();
        _cnd->unlock();
    }

    JobPtr JobQueue::dequeue(const Thread& arg) {
        JobPtr jp = NULL;

        _cnd->lock();
        while (! jp) {
            if (_queue.empty()) {
                cout << "Thread = " << arg.tid() << " to wait" << endl;
                _cnd->wait();
            }
            jp = _queue.front();
            if (jp) {
                _queue.pop_front();
                break;
            }
        }
        _cnd->unlock();

        return jp;
    }

    // ----- Class : ProducerThread -----

    // Instance of this Thread produces Jobs
    class ProducerThread : public Thread {
        public:
            /*
             * This method will be executed by the Thread::exec method,
             * which is executed in the thread of execution
             */
            void run() {
                JobQueue *q = reinterpret_cast<JobQueue *> (arg);

                for (;;) {
                    q->enqueue(*this, new PrintJob());
                    sleep(rand() % 5);
                }
            }
    };

    // ----- Class : ConsumerThread -----

    // Instance of this Thread works on Jobs
    class ConsumerThread : public Thread {
        public:
            /*
             * This method will be executed by the Thread::exec method,
             * which is executed in the thread of execution
             */
            void run() {
                JobQueue *q = reinterpret_cast<JobQueue *> (arg);

                for (;;) {
                    JobPtr jp = q->dequeue(*this);
```

```
                        jp->work(*this);
                        delete jp;
                    }
                }
    };

    // ----- Main -----

    int main() {
        JobQueue jq;
        ProducerThread pth;
        ConsumerThread cth1, cth2;
        pth.start(&jq);
        cth1.start(&jq);
        cth2.start(&jq);
        pth.join();
        cth1.join();
        cth2.join();
    }
```

With this, we come to the end of this guide and hope it serves as a useful and practical reference.