**Almost like Java threads**

By **[rtybase](http://www.codeproject.com/script/Membership/View.aspx?mid=1540412)** | 23 Sep 2007

A Java like threading framework.

**Sponsored Links**

窗体顶端

* [Download source files - 49.3 KB](http://www.codeproject.com/KB/threads/LikeJavaThreads/LikeJavaThreads_src.zip)

**Introduction**

In my previous article, [Sound recording and encoding in MP3 format](http://www.codeproject.com/audio/RecordingAndMP3Encoding.asp), I promised to continue describing some aspects of MP3 streaming. Problem is, they are too many, and they don't fit all in one article. Well, they do, but, I am afraid, you will lose the focus. At the end, by collecting the code split among articles, we will build an MP3 streaming server application.

In this article, I will describe (better to say, offer) a "like **Java** threading" framework. Obviously, a good server application needs a well thought threading engine. Why like **Java**? I implemented it so (inspired from **Java**) because threading implementations in **Java** are simple and elegant (and this is quite a strong argument, I think). Unfortunately, this framework doesn't support all of **Java**'s threading functionalities, that's why they are just "**almost** like **Java** **threads**".

**What we know about Java threads?**

Let's review a few things we know about **Java** **threads**. We know that **Java** **threads** are based on the Thread class and the Runnable interface.

1. When extending from the Thread class, it is important to implement the run() method which will be executed by the Thread:

class MyThread extends Thread {

public void run() {

...

}

}

...

MyThread th = new MyThread();

th.start(); *// will execute run() within the thread*

1. When implementing Runnable, it is important to implement the run() method and pass the Runnable to a Thread:

class MyRunnable implements Runnable {

public void run() {

...

}

}

...

Thread th = new Thread(new MyRunnable());

th.start(); *// will execute run() method of the passed instance of the MyRunnable*

But that isn't all, the stop() method of the Thread class is declared deprecated (see JDK, e.g., **Java** 1.5), and another mechanism for stopping **Javathreads** is offered. Without going deep inside the details (you can check the JDK for more details), there are two methods that are offered:

* interrupt() to set a thread's interrupted status.
* isInterrupted() to verify if the current thread has been interrupted.

So, the previous code snippets will look like:

class MyThread extends Thread {

public void run() {

while (!isInterrupted()) {

...

}

}

}

...

MyThread th = new MyThread();

th.start();

...

th.interrupt();

class MyRunnable implements Runnable {

public void run() {

while (!Thread.currentThread().isInterrupted()) {

...

}

}

}

...

Thread th = new Thread(new MyRunnable());

th.start();

...

th.interrupt();

**What we know about Win32 SDK threads**

Just a few functions like CreateThread, ResumeThread, SuspendThread, and TerminateThread. We also know that the CreateThreadfunction accepts a few parameters, most important being:

* a pointer to a function to be executed within the thread.
* a pointer to a variable to be passed to the thread, which actually will be passed to the thread's function (see the previous parameter).

**Almost like Java threads, implementation 1**

The only function (a few actually, but we will see them later) we need from the Win32 SDK is CreateThread. We are not interested in the ResumeThread and SuspendThread functions since their equivalents in **Java** are also declared deprecated (they may bring the application to deadlocks if a thread owns critical locks and is suspended). We are also not interested in TerminateThread which is declared not safe (check the Win32 SDK), and **Java**'s stop() equivalent is deprecated as well. We will need a general interface which looks like:

class IRunnable {

protected:

virtual void run() = 0;

};

and, similar to **Java** threading, will play the role of the core in this framework. One may see that, comparing to **Java**, I declared run() as protected(in **Java** it's public). Not a big problem, except it is useless (and almost every time it is) to call the run() method not within a thread.

We will also create a class like (just skeleton, without including implementation, if interested in implementation - check the sources):

class CThread: public IRunnable {

private:

...

volatile bool m\_bIsInterrupted;

IRunnable \*m\_RunObj;

...

*// See ::CreateThread(...) within the start() method. This is*

*// the thread's API function to be executed. Method executes*

*// run() method of the CThread instance passed as parameter.*

static DWORD WINAPI Start**ThreadS**T(LPVOID PARAM) {...};

protected:

*// It is not possible to instantiate CThread objects directly.*

*// Possible only by specifying a IRunnable object to execute*

*// its run() method.*

CThread(int nPriority = THREAD\_PRIORITY\_NORMAL): {...};

*// this implementation of the run() will execute the passed IRunnable*

*// object (if not null). Inheriting class is responsible for*

*// using this method or overriding it with a different one.*

virtual void run() {

if (this->m\_RunObj != NULL) this->m\_RunObj->run();

};

public:

CThread(IRunnable \*RunTask, int nPriority = THREAD\_PRIORITY\_NORMAL) {

...

if (this != RunTask) this->m\_RunObj = RunTask;

else throw "Self referencing not allowed.";

};

virtual ~CThread() {

this->interrupt();

*// wait until thread ends*

this->join();

};

*// Method returns the instance of a CThread responsible*

*// for the context of the current thread.*

static CThread& currentThread() {...}

*// Method signals thread to stop execution.*

void interrupt() { this->m\_bIsInterrupted = true; }

*// Check if thread was signaled to stop.*

bool isInterrupted() { return this->m\_bIsInterrupted; }

*// Method will wait for thread's termination.*

void join() {...};

*// Method starts the Thread. If thread is already started/running, method*

*// will simply return.*

void start() {...};

};

Everything looks pretty much simple, except two (maybe) things:

* **Q1**. How run() is actually being executed within the thread?
* **A1**. The key is this method:

static DWORD WINAPI Start**ThreadS**T(LPVOID PARAM)

It is static, and doesn't point to any instance of the CThread. We can implement the start() method like:

void start() {

HANDLE hThread; *// 线程句柄*

LPTHREAD\_START\_ROUTINE pStartRoutine = &CThread::Start**ThreadS**T;

...

hThread = ::CreateThread(NULL, 0, pStartRoutine, (PVOID) this, 0, NULL);

if (hThread == NULL) {

...

throw "Failed to call CreateThread(). Thread not started.";

}

...

}

So, pStartRoutine points to the Start**ThreadS**T method (which is static!!!), and we pass it, as a pointer, to the CreateThread function. We also pass this, the pointer to the current instance of the CThread object, to the CreateThread.

From the other point of view, Start**ThreadS**T should look like:

static DWORD WINAPI Start**ThreadS**T(LPVOID PARAM) {

CThread \*\_this = (CThread \*) PARAM;

if (\_this != NULL) {

...

\_this->run();

...

}

return 0;

}

That's the trick, nothing complicated and, to be honest, a very well-known one.

* **Q2**. OK, this class can be inherited and objects of the inherited classes will execute the proper run() method (of the inherited classes). In this case, we don't have problems with:

void run() {

while (!isInterrupted()) {

...

}

}

But CThread can be instantiated itself since it has a public constructor accepting a pointer to an object of type IRunnable. But, that is a separate object with its own run(), so how do we handle such cases?

class MyRunnable: public IRunnable {

...

void run() {

while (!CThread::currentThread().isInterrupted()) {

...

}

}

};

...

MyRunnable \*rn = new MyRunnable();

CThread \*th = new Thread(rn);

th->start();

...

th->interrupt();

* **A2**. The key point here is the TLS API (Thread Local Storage). See MSDN for the TlsAlloc(), TlsFree(), TlsSetValue(), andTlsGetValue() functions. Going back to the Start**ThreadS**T method, we adjust it like:

static DWORD WINAPI Start**ThreadS**T(LPVOID PARAM) {

CThread \*\_this = (CThread \*) PARAM;

if (\_this != NULL) {

...

TlsSetValue(CThread::m\_TLSDesc.descriptor, (LPVOID) \_this);

\_this->run();

...

}

return 0;

}

and the currentThread() method will look like:

static CThread& currentThread() {

CThread \*thr = (CThread \*) TlsGetValue(CThread::m\_TLSDesc.descriptor);

if (thr == NULL) throw "Call is not within a CThread context.";

return \*thr;

}

This way, anything "running within" an instance of the CThread (only!!!) will have access to that instance, otherwise (if not "within") - an exception will be raised.

**Bad thing(s)**

Unfortunately, this implementation of the CThread has one issue; in the end, this is C++, not **Java**. See the destructor's implementation:

virtual ~CThread() {

this->interrupt();

*// wait until thread ends*

this->join();

};

Yes, when an object of type (including derivates) CThread is being destroyed, an interrupt request is sent (in case someone forgets to do that) andjoin() is called, which waits until the thread stops execution (if running). So, things like:

while (!isInterrupted()) {

...

}

within the run() method are more than mandatory, except if you implement your own stopping mechanism (which is redundant with this implementation, and in **Java**'s one as well). This isn't quite bad. In the end, when you develop code, you must be aware of your own code (and what happens inside your code). Plus, **Java** applications don't exit if there is at least one non-daemon thread running (even if the main() method finishes execution), and **Java** doesn't guarantee that an interrupt() will stop the thread (interrupt() isn't meant to do that at all, it's just a signal to stop). So, be ready for your code to be stuck for a while (or more than a while) when destroying **threads**.

Also, consider the following code:

class MyThread: public CThread{

private:

Object o;

void use(Object &obj) {...}

public:

MyThread(): o(), CThread() {...}

protected:

void run() {

while(!isSuspended()) {

use(o); *// intensive use of the "o"*

}

}

};

Everything looks fine with this code at the first look. But there is a problem inside. Let's follow the execution of:

MyThread \*mth = new MyThread();

mth->start();

...

delete mth;

So, when mth is created, the CThread() constructor is invoked, an object o is created, and the MyThread() constructor is called (the C++ standard says so). The thread pointed by mth is started, and after a while is being destroyed. The destruction will be performed in a way opposite to construction; ~MyThread() is invoked (even if not present), the object o destroyed, and ~CThread() called at the end. Mind that between the~MyThread() and ~CThread() calls, the object o is being destroyed, but the thread is still running and executing run() which uses the object o(as said, intensively). Oops, and I would say a very big one. So, how do we handle this?

**Solution 1**. Easy way, but we need to remember it every time when inheriting from CThread:

~MyThread() {

interrupt();

join();

}

**Solution 2**. Instead of statically initialized members, or even dynamically initialized ones, use smart pointers with a reference counter, e.g.,shared\_ptr ([see this link](http://www.boost.org/libs/smart_ptr/shared_ptr.htm)). The code will look like this (just a pseudo-code):

class MyThread: public CThread {

private:

shared\_ptr<Object> o;

void use(shared\_ptr<Object> &obj) {...}

public:

MyThread(): CThread() {

this->o = shared\_ptr<Object>(new Object());

...

}

protected:

void run() {

shared\_ptr<Object> o1 = this->o;

while(!isSuspended()) {

use(o1);

}

}

};

This is even more similar to **Java**, because the object, to what shared\_ptr<Object> points to, is destroyed automatically when the reference counter is equal to zero.

**Solution 3**. Playing around with templates (making CThread a template, rather than a class), something like:

template <class T> class CThread: public T {

public:

CThread(): T() {...}

...

~CThread() {

suspend();

wait();

}

};

One problem here will be the method currentThread() which yet must be static. Another problem, T must have a constructor with no parameters. And, where is IRunnable?

I couldn't find a very nice, 100% working solution. So, I am going on with "Solution 1" in mind. However, don't hesitate to contact me with good ideas!

**Is that all?**

Not exactly, otherwise, I wouldn't ask that question. With JDK 1.5, a new package was included: **java**.util.concurrent. This package brings a set of very nice functionalities (if I can call them so) related to concurrent processing and multi-threading. A few of them being:

* Executors factory class and
* ExecutorService interface

The Executors (factory) class has a few static methods, each one returning objects of type ExecutorService. Depending on the method being called, the relevant object of type ExecutorService is returned with the relevant thread pooling engine implementation. Each of the returnedExecutorService contains a task queue and a few overloaded submit() methods to submit tasks to the pool in order to execute them. And guess what; one of the submit() methods accepts objects of type Runnable as tasks.

**Almost like Java threads, implementation 2**

Let's try to do something similar, but rather than being very generic, I will focus only on the Executors.newFixedThreadPool(...). So, we will build a thread pool with a fixed number of **threads** (again skeleton only; for more details, see sources):

class CSimpleThreadPool: public IRunnable {

private:

vector<CThread\*> m\_arrThreadTasks;

mpriority\_queue<CPriorityTask> m\_PQueue;

*// Method will return a task from the queue,*

*// if there are no tasks in the queue, method will return NULL.*

IRunnable \*get() {...}

public:

*// How many* ***threads*** *are in the collection.*

int **threads**() const { m\_arrThreadTasks.size(); };

*// Method starts pool's* ***threads****.*

void startAll() {...};

*// Constructor creates the thread pool and sets capacity for the task queue.*

CSimpleThreadPool(unsigned int n**Threads**Count, unsigned int nQueueCapacity = 16) {

int i;

CThread \*thTask = NULL;

...

*// Set initial capacity of the tasks Queue.*

m\_PQueue.reserve(nQueueCapacity);

*// Initialize thread pool.*

for (i = 0; i < n**Threads**Count; i++) {

thTask = new CThread(this);

if (thTask != NULL) m\_arrThreadTasks.push\_back(thTask);

}

};

*// Submit a new task to the pool*

void submit(IRunnable \*pRunObj, int nPriority = 0) {...}

*// Method will execute task's run() method within its CThread context.*

virtual void run() {

IRunnable \*task;

while (!CThread::currentThread().isInterrupted()) {

*// Get a task from the queue.*

task = get();

*// Execute the task.*

if (task != NULL) task->run();

::Sleep(2);

}

}

virtual ~CSimpleThreadPool() {...};

*// Method signals* ***threads*** *to stop and waits for termination*

void shutdown() {

for (int i = 0; i < m\_arrThreadTasks.size(); i++) {

m\_arrThreadTasks[i]->interrupt();

m\_arrThreadTasks[i]->join();

}

}

};

The only difference from the **Java**'s FixedThreadPool is, CSimpleThreadPool uses a priority queue, and it has only one submit method.mpriority\_queue is the same STL priority\_queue, except it allows pre-setting the capacity as well as obtaining the current capacity of the container.

Now, we can simply do the following:

#define NUMLIM 50

class A: public IRunnable {

private:

int m\_num;

protected:

void run() {

Log::LogMessage(L"Executing %d thread\n", m\_num);

}

public:

A(int num) { m\_num = num; }

};

class B: public IRunnable {

protected:

void run() {

int i = 0;

while (!CThread::currentThread().isInterrupted()) {

Log::LogMessage(L"Executing B cycle %d\n", i);

::Sleep(100);

i++;

}

}

};

int main(int argc, char\* argv[])

{

CSimpleThreadPool tp(2, NUMLIM + 1);

A \*a[NUMLIM];

B \*b = new B();

int i = 0;

for (i = 0; i < NUMLIM; i++) a[i] = new A(i);

tp.startAll();

tp.submit(b);

for (i = 0; i < 20\*NUMLIM; i++) {

tp.submit(a[i % NUMLIM], i % NUMLIM);

::Sleep(1);

}

::Sleep(10000);

tp.shutdown();

delete b;

for (i = 0; i < NUMLIM; i++) delete a[i];

return 0;

}

窗体底端