MD Thread Guide

Drew Dolgert
http://www.tc.cornell.edu/Research

May 3, 2000

1 Introduction

This should help you get started with the MD thread classes. We will cover

- how to use the thread class;
- how the class structure enables us to make it cross-platform;
- comments on implementation of the thread class.

You will hopefully know enough from this to be able to write code which can execute using this thread class.

By *thread* I mean a thread of execution. Every process has at least one thread of execution when it starts its main. This class is meant to help you run your simulation in a separate thread.

The "MD" thread classes are used in our molecular dynamics code. They have an interface limited to the needs of running and controlling a numerical simulation. The design goals are therefore ease of use and speed while running.

2 Use

Assume we start with two bodies of code. The gritty numerics of the simulation are of primary interest and act as a server to some client set of code which controls and displays that simulation's data. Right now the Windows GUI is the client and we call the simulation a Unix main().

2.1 Client Interface

The simulation thread class has two interfaces, one for the client and one for the threaded server. Let's first discuss the client because it is simple. Here is the class inteface:

```
class Thread {
public:
  int Start();
```

```
  int Stop();
  int Pause();
  int Resume();
  int Toggle(); // if paused, resume.  if running, pause
};
```

Start() and Stop() create and destroy the thread. Pause() and Resume() block the thread. If the underlying simulation cooperates (by implementing a Threadable interface), then the thread exits cleanly when it stops and pauses and resumes at an acceptable checkpoint in the calculation.

The most important property to note about this thread class is that it is synchronous. That is, when you tell the simulation to Start(), it will have started by the time the method returns. If you ask it to Pause(), it will have reached its checkpoint and paused before the method returns. This makes it very clear how to deal with the thread.

There are issues with this implementation. For instance, the running graphical interface should not wait on the thread to complete because it could lock up for seconds. That would flaunt user-interface conventions. In addition, if we are thinking about running the simulation on a distant machine and connecting to it with the user interface, we would hardly want to freeze the window until we hear a reply that the thread stopped. Were we to use an asynchronous thread, we would not be tempted to rely on immediate responses from commands to the thread.

For now we use the synchronous thread because we do, in fact, rely on immediate responses from the simulation thread (in DoModal while changing dialog information). Making an asynchronous thread is a simple and fun task from here. Just try not to rely on immediate replies when you code your own applications.

There is an additional part of the client interface used when creating the thread.

```
class Thread {
public:
  SetThread(Runnable *thread);
  SetThread(Threadable *thread);
};
```

During initialization, one uses these commands to tell the thread class what body of code it is to run. How does that work?

2.2 Server Interface

2.2.1 Runnable

The body of code that wants to be run inside a thread has to conform to an interface. The simplest looks like a Java model.

```
class Runnable {
```

```
public:
  int Run(void) = 0;
};
```

Inheriting from this abstract class is a guarantee that your simulation has a Run() method. When a Thread class is initialized with a Runnable simulation, it knows to start the thread and call its Run() method for the body of the thread. The simulation could have other methods if it liked.

```
class MDThread {
public:
  int Init();
  int Run(void);
  int Cleanup();
};
```

The only one that gets called during the execution of the thread is the Run() method.

Just having a Run() method doesn't do a lot for a threaded object because it allows very limited interaction between the Thread class and the Runnable class. It is easy to start a class by calling its Run() method, but how does one stop, pause, or resume the thread? You could pause it with a system signal which would freeze it mid-computation, but that is system-dependent and hardly useful. You could stop it only by terminating the thread, a dangerous prospect on Win32. The only way a Runnable class is useful is if you expect it to run to completion every time without interference from a client. Then it starts and ends cleanly.

2.2.2 Threadable

There is more functionality in the Threadable class. This class allows the simulation to take a breather when you ask it to. It is a cooperative multitasking in the sense that the simulation promises to choose some times when it will call a function where it is at a good point to stop in the code.

There are two functions in the Threadable class. The first is called by the simulation in its Run() method, presumably every loop or every ten loops through. The second allows the thread to register itself with the simulation as the person to talk with when it comes time to pause.

```
class Threadable : public Runnable {
public:
  bool IsRunning() {
    if (this->pThread) return pThread->IsRunning();
    else return true;
  }
  void SetThread(Thread *const thread) {
    this->pThread = thread;
  }
```

```
private:
  Thread *pThread;
};
```

When the Thread gets hold of the Threadable simulation, it registers itself with SetThread() so that the simulation can call back to the Thread's IsRunning routine.

Here's the trick: it may take a long time to return from IsRunning() if the client requested a pause. It is inside IsRunning() that the simulation learns we want it to quit its inner loop.

We expect the MDThread, therefore, to have a Run() method like the following.

```
int MDThread::Run() {
  while (this->IsRunning) {
    do this stuff
    do that stuff
  }
  return 0;
}
```

You can, of course, initialize in the run loop if you want, but anything initialize on the stack of the run loop disappears when the thread ends. Anything initialized in the MDThread class is practically global as far as threads are concerned. It will live before and after the thread runs so that the client need not be concerned about its lifetime.

3 Class Structure for Cross-Platform

When I described the class structure to Chris Myers, he just nodded his head and said, "Sure." I get the impression this is a standard mixture of the patterns from *Design Patterns*. The goal is to be able to use the same Thread class on Win32, Linux, and whatever else even though the implementation is highly system-dependent. We can mix and match a few patterns to make the process transparent.

The first pattern is an abstract factory as shown in Fig. 1. It is a Singleton instance, so your code can always retrieve the same reference to ProcessFactory by calling the static member function ProcessFactory::Instance(). The only trick to system dependence is that, before you use the factory, your code must somewhere instantiate the version appropriate to your machine with a call to UnixProcessFactory::Create() or WinProcessFactory::Create().

Now that the call to ProcessFactory::Instance() will now return the appropriate Unix or Windows version, you may use the member functions to create process objects like a read-write lock or a simulation thread. They have similar class structures to the process factory in that there is an abstract base

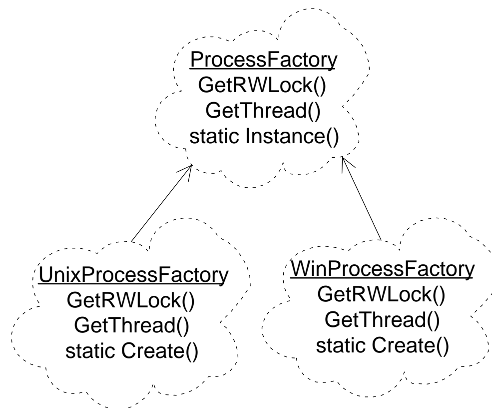

Figure 1: The system-specific ProcessFactories are concrete instances of the abstract ProcessFactory class. You must create only one of them, and do it only once, by calling its Create() method. After that, your code can query the ProcessFactory::Instance() without concern for which system you are on.

class RWLock and a system-dependent implementation, WinRWLock or UnixRWLock. The trick is that your code need not know which you need because the concrete ProcessFactory decides which to give you.

References

[1] Programming Applications with Win32 by Richter. This is my favorite place to learn about threads. It is mature and covers the basics.

[2] I have seen a recommendation for the nutshell book on Win32 Thread Programming, but it looked lightweight in the bookstore.

[3] Compound Win32 Synchronization Objects by Ruediger R. Asche `http://msdn.microsoft.com/library/techart/msdn_locktest.htm` Implements read/write locks and group locks with timings.

[4] Synchronization on the Fly by Ruediger Asche `http://msdn.microsoft.com/library/techart/msdn_onthefly.htm` Gives a demonstration of concurrency analysis.

[5] Detecting Deadlocks in Multithreaded Win32 Applications by Ruediger R. Asche `http://msdn.microsoft.com/library/techart/msdn_deadlock.htm` Describes deadlock detection using Petri net formalism.

[6] The Implementation of DLDETECT.EXE by Ruediger R. Asche `http://msdn.microsoft.com/library/techart/msdn_dldetect.htm` Part II of the series.

[7] Putting DLDETECT to Work by Ruediger R. Asche http://msdn.microsoft.com/library/techart/msdn_dldwork.htm Shows how to analyze multithreading with Petri nets.

[8] Bugslayer, MSJ, October 1998 by John Robbins http://www.microsoft.com/msj/1098/bugslayer/bugslayer1098.htm About an included utility to detect deadlocks in your code.

[9] Win32 Q&A January 1997 by Richter http://msdn.microsoft.com/library/periodic/period97/win320197.htm Implements WaitForMultipleExpressions.

[10] Multithreading for Rookies by Ruediger R. Asche http://msdn.microsoft.com/library/techart/msdn_threads.htm A wide introduction to Win32 threads, but it does cover APCs.