More threading with templates

Introduction and Linux diversion

Even before my article in Overload 31 had gone to print I'd suggested to Einar that there was probably enough material for a second article. This falls into two broad categories, firstly improving the presented template and re-enforcing the IOU design pattern; secondly, porting it to a UNIX environment.

I'd always hoped to port the material to UNIX but had just never got round to it. As I don't use UNIX at work this meant I needed UNIX at home and as it happened by hard disc contained a Slackware Linux installation. However, I'd never really finished the install so, after two years I was still booting from floppy, and couldn't get the modem to work. Just before Overload 31 arrived fate struck and my NT set up locked me out. After re-installing NT on a new hard disc I decided to start again with Linux and installed Red Hat Linux 5.2.

The point of telling you all this is so I can say one thing: Red Hat Linux 5.2 is easier to install than Windows NT 4.0. Although the X-server worked fine first time my connection to Demon Internet was more difficult and involved tracking down a few Demon specific how-to's. So with that said back to the main thread of this article - sorry for the pun!

All the code presented here has been tested on NT 4 and Windows 98 (from here on collectively referred to as Win32) with Visual C++ 5.0 and on Red Hat Linux 5.2 with egcs 1.0.3 and the Posix threads library. My main source of reference for the Posix threads was Programming with POSIX threads by David R. Butenhof[1] which I found to be both authoritative and approachable. However, I'm not as familiar with Posix threads as I am with Win32 so if you know a better way of doing any of this I'd like to hear from you. While I've tested the UNIX code presented here I've not had a chance to use it in a production environment, but I have learnt a lot from writing this.

Posix

Posix is the thread library standard which is now supported by most UNIX platforms. Before the Posix threads (pthreads for short) standard different vendors had different libraries and different standards. Pthreads is just another library on the kernel. The library I have used was produced by Xavier Leroy and is described as "Linux threads, a posix thread implementation." This is supplied as standard in the Red Hat Linux distribution and several others.

As pthreads is just a library it should be possible to implement it on top of the NT threading model. However, I suspect this may prove more difficult in practice then theory.

Strategy

Writing platform independent code places additional requirements on the developer. Good design and implementation practice is the best ally here. Abstracting and hiding all low-level dependencies behind fixed interfaces both helps platform independence and layers software.

The strategy I present here emphasizes separation of interface - which should be platform independent in the header files - from the implementation - which is necessarily platform specific and which I try to confine to the source CXX[2] files. John Lakos[3] gives a tour-de-force in describing how to layer programs and separate interface and implementation, highly recommended.

Hence one of my objectives when writing any code, and especially when writing cross platform code is to minimise the number of macro's and the general use of anything using a #.

Critical section

As the critical section is easier to port and sets the pattern for what is to follow I'm starting this time by porting the critical section to Linux. First it is necessary to remove any OS specific information from the header file. This means we can't in-line any of our functions. Listing critical.h shows the revised header file. Listing critical.cxx shows the implementation.

I started out with the intention of putting all code in the Accu namespace. I'm quiet taken with namespaces at the moment and this would allow my code to compile with the code from the last article. However, when I came to compile under egcs I got a "sorry, not implemented: namespace" message. What surprised me was that the standard library all seems to be in the std namespace. I assume some work around has been used here. As this article is about threads and not namespaces I decided to kludge it with a NAME_SPACES macro!

Under Win32 threads are automatically recursive while Posix threads are not. That is to say, if a Win32 thread enters a critical section and then attempts to enter it again it is allowed to. (As far as I know there is no other way for a critical section to behave under Win32.) Because critical sections are recursive threads cannot block against themselves.

However, under Posix things are different. There are four types of critical section[4]. For simplicity I have used the recursive type, which has the same behaviour as Win32 critical sections. However, should you require one of the other types it should be possible to write your own implementation under Win32 and hide it behind the same interface provided here.

An NT critical section is an optimised mutex which can only be used among threads of the same process. If two threads in different processes need to guard a section of code a full mutex must be used. Pthreads does not implement the critical section optimisation, hence it is necessary to use a mutex. Posix allows you to set various attributes on critical sections (and threads) while Microsoft limit the choices or implement the functionality differently. For simplicity I have restricted myself to the basic functionality.

Beyond this the CriticalSection and CriticalToken classes work exactly the same as the ones provided in my first article; so to create a critical section you simply declare a CriticalSection object, to enter

the section declare a CriticalToken object using the critical section object as a parameter to the constructor. To leave the critical section use scope rules to destroy the critical token.

Changing the thread template

Before porting the thread template I wish to modify it. IouThrd1.h shows what I'm calling the IouThread. While editing my original article Einar suggested the Redeem() function and the addition of the return type parameter. This enhances the original IOU design pattern of the template but at the expenses of removing some of the versatility. I think there is a role for both versions of the template. I make most use of the original template not as an IOU pattern but as a quick and easy thread launcher.

In this version when Redeem() is called the function attempts to return a type V which is supplied as a template parameter. If the thread has terminated the return is immediate; if the thread has yet to completed the call blocks until there is a value to return, this makes it easier to create producer-consumer chains.

The value type is required for Redeem() to ensure type safety in returning a value. However, simply adding the Redeem() function to original template would causes several problems which all contributed to my decision to create a new template:

? Why change code that works? Why add functionality which by my own admission is not always needed or used?

? The original template allows you to orphan threads or kill them. This is not compatible with a Redeem() function. Redeem() has no use if the thread is orphaned; while what is the effect of Redeem() on a thread that may be killed?

? In compilation terms we hit a problem, the original template defaulted the second parameter thus making it difficult to add a third; existing code would be broken.

There have been some other changes which are more subtle. Firstly the constructor takes an instance of the worker object and copies it. Here I'm following STL style in using value semantics. One of the main problems I have with the original template is managing the life times of the worker objects and templates - this approach removes the issue altogether. However, I'm not completely happy with this. It raises two more issues; firstly, are we sure our worker objects can be copied, indeed do we want to copy them if the copy is an expensive operation? Secondly, and to my mind more significant, I often make the thread template a member of some high level controller class, mentioning the worker object directly means we must include it's header file in the controller. This introduces a dependency that may not logically exists between the controller and worker. If the template accepts a pointer or reference I can use a forward declaration in the controller class and break the dependency.

There are two sides to this argument, and I'm not completely sure which is best. In truth the same dependency problem has lead me to use pointers with the standard containers before now. If the dependency problem causes you problems it is fairly trivial to make the IouThread accept a pointer and access the worker through that. I would suggest that you *adopt* the pointer in the template and delete it in the destructor. This would allow you to write:

```
IouThread<Worker, int> thread(new Worker(…));
```

I would advise against passing a reference as you must not only manage the life time of the worker but also raise the possibility of someone attempting direct access to the worker.

(A quick aside here: the terms *adopt* and *orphan* are used in preference to *get* and *set* because they imply a transfer of ownership. This should help when tracking down resource leaks.)

One option I rejected in the rework was to run the thread as soon as the constructor was done, this would remove the need for the RunThread method. Again, this is because I typically embed several thread templates in a controller class and I want full control of when to start them. If the constructor started the thread I would be forced to control the life time of the controller; using this method the controller can control the life time of the threads.

One more modification is evident to NT programmers. The CreateThread which I used but warned against has been replaced by _beginthreadex. Richard Howells mail me to say that _beginthread actually returned the thread handle which I wanted. While this involves some extra casting (due to Microsoft's function signature) it is not dangerous and actually fits in with the next item on the agenda. However, _beginthread conveniently closes the handle for you when it thread reaches the end, as I use this handle in Complete() I don't want it closed if the thread has finished, I get an "invalid handle" error. Hence I've used _beginthreadex which leaves the handle open for me to close – but I must remember to close it.

Finally, I have removed some code. The comments where removed to tighten up the code up and make this article shorter; and I've also had to remove the Suspend, Resume and IsTerminated functions. In part these where removed because they have less of a role in a strict IOU pattern; but also, I found it had to replicate their functions under Posix without adding more management code (that will make a nice exercise for the reader!)

Example program

If you take a look at main.cxx you will find a simple model of the British economy. The main function creates economies for England, Scotland and Wales, runs them as threads and redeems the result to predicts the final GDP.

Under Windows you can either use IouThrd1.h which shows the changes made to the template in the simplest way or IouThrd2.h. Linux/Posix users can only use IouThrd2.h. Apart from one include line main.cxx is unchanged for any platform.

IouThrd2.h exposes the same interface as IouThrd.h but does not contain OS specific calls. These are the subject of the next section.

To show the multi-threaded nature of the application at run time I decided to add a short sleep to the worker between calculations. Once again, Win32 is at odds with the Unix world. Win32 uses a Sleep(number of milliseconds) function, while Unix uses a sleep(number of seconds) function. I am sure this is not the only example we will find so I've added the auxiliary functions in auxfuncs.h and

auxfuncs.cxx. These follow the same considerations as the critical section for splitting platform independent interface from platform specific implementation.

Differences in threading models

NT and pthreads have a number of differences when it comes to threads. Under pthreads the overall process will terminate when the main thread (the one which started with main(…)) terminates regardless of how many other threads are still running – they will be terminated[5]. NT however, will terminate the process only when the last thread has finished executing[6].

The thread templates ignore this rather substantial difference. If this causes you a problem then pthreads can be made to behave like NT by calling pthread_exit at the end of the main thread. This causes the process to wait for all threads to terminate – at the expense of allowing a return value from the main thread. AuxFuncs contains the function WaitForThreads, this can be called on either platform, on Win32 it does nothing, on Posix it calls pthread_exit.

NT implements a *detach* paradigm when creating new processes but not for threads. Pthreads implements an *attached* and *detached* paradigm for threads. Simply, if thread A is attached to thread B then any resources used by thread B will not be reclaimed until A and B have been terminated. If B is detached then resources are released as soon as B terminates.

When a detached pthread terminates the same thing happens. However, a thread must either be created detached or be detached (using pthread_detach) after creation. The catch is, once the thread is detached the creating thread has no simple means of communicating with it.

Alternatively, and the approach used here, is leave the new thread (B) attached to the creating thread (A). When Redeem() is called the thread is *joined* – using pthread_join. This causes the callee (A) to pause until the called (B) completed. The exit code from B is returned to A via the second parameter pthread_join. Had B been detached from A it would not be possible to call pthread_join. The final action of pthread_join before returning to A is to detach B allowing the resources to be reclaimed but prevents any subsequent call to pthread_join.

(In NT the exit code of a thread is only available through the GetExitCodeThread function.)

Porting the Thread Template to Linux

The first problem we face here is that my previous technique won't work. Because we have template here I can't separate the implementation code from the interface code in the same way I did with the critical section; a modified approach is required.

There are four options here:

? Write another version of the template with the same signatures but use POSIX calls.
  While this is undoubtedly the simplest it mingles high level application code with the low level system dependent code. Personally, I like to think of the OS facing functions at the bottom of the dependency tree, I'd place the IouTemplate in the next level. This would result in platform independent threads only where this template is used.

? Add a third parameter to the template to indicate the platform and produce partial specialisations of the template for each platform.
Although I would need to provide a macro which would expand to the current OS type and probably defaulting the template to this, developers could still re-introducing platform dependence by writing: IouTemplate<MyWorker, double, Posix>. Additionally, not all compilers have partial template specialisation support yet.

? Provide the template signatures in the header file and export the actual code using the template export keyword : unfortunately Microsoft don't support this yet.
? Move the template to an abstract set of function calls and implement these differently on NT and Linux.

You may guess that I chose the third option. The first option doesn't fit with the strategy I outlined at the start. The second option would lead to some rocket science C++, so it's omission is probably a relief to maintenance programmers everywhere!

thdfuncs.h gives the prototypes for platform independent thread functions. These represent a sub-set of the calls available on both systems. Before describing the implementation of these in thdfuncs.cxx we need to look at the typedefs in more detail, again we need to jump through a couple of hoops to keep OS specific #if's out of the header files.

Both Win32 and Posix start thread execution using a pointer to a function. Under Posix this has the signiture:

```
Void* Start (void*)
```

But Win32 asks for:

```
unsigned int _stdcall Start(void *param)
```

To ensure that the thread template is not troubled by this significant different the platform independent thread functions ask for a:

```
int Start(void*)
```

Under Win32 the OS specific thread functions provide a Win32 compatible start function and the Posix functions provide a suitable Posix start function. From here we can call the user supplied, platform neutral function. However, this means we must pass the address of the platform neutral function to the specific function. Both Win32 and Posix only allow one void* parameter to be passed to the start function and this is already used for an object pointer.

The solution is to bundle both the object pointer and platform neutral function pointer into another structure (StartData) and pass this to the starter function.

Hence the following chain of events occurs: 1) user calls RunThread on the thread template, this calls the low level CreateThread function in thrdfuncs.cxx with a pointer to it's own platform independent StartRun function and a pointer to the worker object; 2) CreateThread bundles both of these pointers into a new StartData object and calls the platform specific start function with a pointer to this structure; 3) the OS creates the thread and starts execution at the platform specific StartRunFunc function; 4) this function now disassembles the StartData strucuture to get a pointer to the static

tempate member and a pointer to the object, it can now call the static function with the worker object; 5) the static template member can recover the type of the worker and run it's own, non-static Run() method; 6) when the worker completes it returns to StartFunc where the StartData object is destroyed. Although this may sound long winded once you've stepped though it with a debugger it makes a lot more sense.

An additional problem occurs when we hit the WaitForThread call. Although Win32 will keep a thread handle open even after a thread has terminated (see the discussion above concerning *beginthreadex)* Posix does not. Calling pthread_join with the id of a complete thread results in an error. Fortunately, while Win32 creates a structured exception which may bring all our code down, Posix simple returns and error code which we can use to imply the thread has already terminated.

Compiling

Under Win32 you must remember to use the mutli-threaded standard libraries, while under Linux you need to include the pthread library.

Return to Critical Section

If you look closely at the Economy::Run method you will see that the CriticalSection and output is in a scope of it's own, which is not strictly necessary if the sleep where not present. My experiments appear to indicate that the egcs compiler performs some kind of optimisation here, either with the class construction and destruction or with the loop. Removing the sleep, results in the English economy thread executing, or locking i/o at the exclusion the Scottish and Welsh threads, after which the Scottish thread runs exclusively followed by the Welsh. My suspicion is that the compiler optimises away the constructor and destructor calls.

Conclusion

Although I dreamt up the thread template some time ago and used it very successfully when reviewed by others some very good point emerged. The desire incorporate these and port it to Posix set off a chain of extensive changes but the original interface remained largely unchanged. Once again, the worth of separating interface from implementation has been proved.

At the September ACCU conference in Oxford Steve Clamage, in an aside suggested that one day there should be a standard C++ threading model. The difference shown here between just NT and UNIX threads shows a clear need. However, producing a model that keeps both NT and UNIX programmers happy (let alone Mac, BeOS, VMS…..) may be hard work!

And finally….

As I stated in the references with my last article the IOU pattern is used by Rogue Wave software. While I've used Rogue Wave Tools++ in the past I've never used, or seen, their Threads++ libraries. The April edition of the *C++ Report* contains a review of this library and some aspects sound similar to what I'm describing here. If anyone has a used Threads++ and would like to compare and contrast

I'd be interested to hear - as I'm sure Einar and Jon would be if you would write the piece for Overload.

The June 1999 issue of Dr. Dobbs contain a piece by Bob Krause on platform independent mutli-threading. Bob looks at Win32 (MFC) threads and Macintosh threads. His approach differs in two main ways: firstly, he attempts to avoid the lowest common denominator issue. Although I have adopted that here for the purposes of a short article I think additional functionality could be added and either mapped onto native functions or written afresh – see my discussion on critical secions above. Secondly Bob uses inheritance, this I dislike because it means the worker must know more about it's place in life as a thread, and it can only be used as a thread. By aggregating the worker in a template the worker is more reusable.

I'm sure my technique could be ported to the Mac but will not be rushing to volunteer to write the article as my only experience of Mac programming was under Scheme many years ago. However, from what I understand, the Mac only supports co-operative multi-threading, this should present an additional challenge which would probably require more interaction between the worker and the template.

---

[1] Addison-Wesley 1997

[2] Some Window programmers may be wonder why I use CXX instead of CPP. Historically there was no standard C++ file extension. Some compilers used capital-C, others used CXX (imagine the X rotated through 45 degrees) while Microsoft used CPP. The Microsoft compiler is pretty forgiving on these things. Hopefully CXX should be generally acceptable to all.

[3] Large-scale Software Engineering in C++ 1997 Addison-Wesley

[4] For further information see Butenhof section 10.1.2.

[5] Butenhof section 2.1

[6] See Jeffrey Richter, Advanced Windows for more details on processes and threads under Windows.