Using templates to handle multi-threading

Introduction

Neither C++ nor the standard library include any support for multi-threading or multi-processing. Some other languages provide native support for parallelism but C++ programmers are thrown back on operating system specific support. However, OS support offers little in the way of abstraction for thread creation, termination, critical sections and so on. Abstracting these mechanisms to make the division of processing into discreet units easier, mapping into the object model and providing some degree of platform independence has a clear attraction.

This article deals specifically with multi-threading but the concepts and code can be applied to multi-process situations.

Creating threads

While working for Sema Group, Jason Ross and myself wished to sub-divide a program into multiple threads which could run in parallel. The intention was to allow several threads to perform database access (which is i/o bound), several more to perform compute intensive tasks and a final thread to perform the output to file. Our design and code were object oriented and we were making extensive use of the STL. We wanted to model our multi-threading in the object model.

We first looked at giving each object methods such as start-thread, end-thread, pause, etc. Obviously this was a generic problem, in the style of STL we should be able to use a template. At this point we found our system provided adequate performance with one thread and the idea went no further.

Since then I have developed this idea further and present here the ThreadTemplate which provides a generic method of multi-threading any worker object. The techniques used are similar to the IOU design pattern[1,2].

Worker object

Developers who use Java, Smalltalk and other OO languages will be familiar with the idea of placing the top of the program within an object; e.g. the *public static void main* in Java which provides the start of the program.

I have assumed that any task we wish to perform in a system can be modelled as an object with a Run() method. The Run() method acts as a starting point for the task execution. Other features of the class, e.g. the constructor and destructor, may be used if desired while persistence of the object after the task has finished provides a latch to store data in, hence allowing the IOU design pattern to be modelled.

The Worker class in listing 1 (ttexample.cpp) shows such an object which performs a simple Eratosthene sieve. (The sieve serves as a simple task to demonstrate the principal, therefore I am not concerned with any optimisation of it.)

The life cycle of the worker object can be summarised as:

- A worker object comes into being in the main thread
- Main thread runs the worker task in a new thread
- Another worker, in another thread runs in parallel with main thread until completion and then terminates - this is for example only.
- Worker object still exists. Main thread can collect results when it is ready

As with the IOU pattern the main thread keeps a reference to the worker threads and collects the result (or, to use the language of the IOU pattern “redeem”) when the task is done. While the worker thread is running the

main thread can query, via the template object, the worker thread to see if it has finished, may suspend (and resume) the thread or choose to suspend itself until the worker is complete. Any additional functionality required (e.g. kill the thread before termination) can to added to the template in the same fashion.

A word about the IOU pattern

The IOU pattern (sometime called *future* pattern) allows multiple threads to communicate results typically in a producer-consumer chain. The producer (worker thread) calculates a result which is used by the consumer. Until the value is available the consumer holds a voucher for the value. The consumer may choose to block until the voucher can be redeemed (i.e. the worker thread has a value which can be collected) or it may do some other work.

Multiple threads may hold vouchers for one producer thread. Although the producer may of finished running as a thread the results still remain accessible. The potential for infinite loops and deadlock is reduced because of the technique's simplicity.

The template presented here actually expands the IOU pattern slightly. A true IOU pattern produces a single result, however, the worker objects used with the template here may contain complex data structures of results. The worker object may be derived from an STL template, for example, the Worker class in listing 1 could be derived from list<short> hence removing the fixed size array member m_list.

Implementation

The implementation given here has been created using Visual C++ (5.0 and 6.0) on NT 4.0, however the concepts should travel to UNIX systems with no problems - I hope to produce a UNIX version of the is template in the near future. The NT API calls are for the most part self explanatory, you don't need to know all the parameters concerned to understand what is happening.

The worker object can be written without reference to thread primitives or other low-level considerations - except were shared resources are involved. The ThreadTemplate shown in listing 2 (threadtemplate.h) provides all the necessary thread handling. Listing 1 main() body shows how the two are brought together.

It is necessary to create a worker object and an instance of the ThreadTemplate<T1, T2> which takes T1 the worker class as a type parameter, the order in which these two objects are created is not important. Once both are in existence the RunThread() method is called on the ThreadTemplate<> object with the worker object as a parameter.

ThreadTemplate<>::RunThread(t) issues a CreateThread call. At this point NT creates a new thread and starts execution at the function StartRun and passes one, void*, parameter. Actually the signature of StartRun() is slightly more complex to conform to NT's expectations but this is explained in the code.

The StartRun(void*) function casts the void parameter to type T and calls the Run() method on the object. On first inspection this function may look nasty but the function is simply casting a void parameter to an object of the type specified in the template parameter type.

It is important to note that no parameters can be passed to the Run() function call. This restriction is imposed by the CreateThread function which only allows one data parameter to be passed to the new thread, and this is needed for the object pointer. However, this is not a great problem as any input parameters required can be passed in the constructor call when the worker object is created. Any output parameter, or other results, can be stored in the object and redeemed when the thread has finished execution. This is in the IOU pattern style and is demonstrated by the call to Print() in listing 1; here the worker thread has finished and the results are redeemed by the main thread.

One modification would be to pass the object handle to the constructor rather than RunThread. Carrying on down this avenue, you could have the thread constructor call StartRun itself - dispensing with the need for RumThread. In part this is a matter of style. However, as CreateThread can fail I would rather deal with this failure in a function other than the constructor.

The behaviour of the destructor is determined by the second parameter, T2, passed to ThreadTemplate at instanciation. The ThreadTemplate<> object exists in the scope of the main thread and when it passes out of scope the destructor is called. By default the destructor will block until the worker task has terminated, this will prevent orphaned threads.

However, two other options exists. Firstly, orphan the thread, i.e. let it run independently until it's natural end without redeeming the results; second: kill it. All three options are allowed by the second template parameter. Notice also that the destructor closes the thread handle. This is important in NT as without it the kernel would keep it's own thread object in existence. Dead threads would over time degrade system performance - this is similar to the UNIX problem of zombie processes. Although we could close the handle immediately after the CreateThread call we need the handle to call SuspendThread, ResumeThread, etc..

Provided the threads have no resource dependencies between then (e.g. critical sections, using memory allocated and freed by another thread) the order in which the destructors are called has little importance. However, if one thread is using a resource allocated by another it is necessary to introduce some kind of resource protection.

Finally, those paying attention will notice that I use CreateThread. Microsoft recommend that for threads using the standard C library (which I think is probably the majority of all threads) it is better to use _beginthread. However, unlike CreateThread _beginthread does not return a handle, and _endthread only operates on the current thread. Therefore, the Kill options would not be available for threads. This may lead to a slight memory leak, so if you program cannot tolerate any leaks you may want to change these calls.

Divide and conquer

This approach to multi-threading provides several other advantages:

- ? It becomes easier to break a task down into sections because each is modelled by it's own object which, as mentioned above, maps the multi-threading into the object model.
- ? Worker objects can be developed separately from each other and the main thread because each one is self contained (notwithstanding shared resources).
- ? The abstraction can help with debugging because each thread can be viewed as one object. This doesn't help where shared resources are involved but if a thread is simply performing a calculation the mechanics are hidden. The macro SINGLE_THREAD_DEBUGGING can be defined which will switch off multi-threading. This will suspend the main thread as soon as a worker thread is created until the worker thread has completed. This need not be controlled solely at compile time, the #ifdef could be replace with a real if and the macro replaced with a flag reflecting an environment variable, or .ini file setting.
- ? Assuming all threads are created and launched using the RunThread() method, it is possible limit the number of threads active in the system at any one time. Although not implemented here, it would be possible to add a live-thread counter which may be incremented on each call to RunThread() and decremented on each destructor call.
- ? Reuse : code which is free of low level OS calls becomes easier to reuse. Once a worker object is developed it may be used in a multi-threaded or single-threaded environment.
- ? Porting : this approach can help with porting because all the OS specific calls exist in one place, namely the ThreadTemplate<> rather than being scattered around various parts of the code.

Synchronisation

It is possible to continue mapping parallelism into the object model. Listing 3 (critical.h) shows a critical section class. The same principal may be used to wrap semaphores, mutexs and other synchronisation items. Further wrapping a resource, say, a database connection, inside one of these objects would protect it from competing resource claims. (Writing functions in the header file is not my normal style, I do it here for convenience and to save space.)

Several threads will use one critical section object which is created before they are spun off, a reference to the critical section is supplied to each thread object. When a thread wishes to enter the critical section it simply creates a CriticalToken object which, using the "resource allocation is initialisation" metaphor[3], attempts to enter the critical section. Once entered, the critical section remains in force until the token passes out of scope at which point the CriticalToken destructor leaves the critical section. Listing 4 (csexample.cpp) shows a simple example. (If you comment out the declaration of the CriticalToken object you will see how failure to guard the i/o channels leads to problems.)

As I just said each thread is passed a reference to the critical section. Passing a copy of the critical section would simply result in a second critical section and would not protect anything. For this reason, I have blocked the default copy constructor and assignment operator.

ThreadTemplate<> itself does not provide any protection against threads competing for resources or encountering deadlock. The basic assumption exists that the worker threads need not know about one another. There are various enhancements that could be made to the model to account for this, most centre on serialising calls to a resource within a resource object. For example, suppose several threads wished to write to a single output stream, e.g. cerr. The output stream could be wrapped in a class with its own critical section, each call to << would enter the critical section, call the wrapped <<, then leave the critical section and return, so serialising calls. Worker objects would be passed a reference to the wrapper object via the constructor allowing access.

Conclusion

By modelling the threads with a generic template we enable threads to be viewed from an OO perspective while also achieving a better level of abstraction making and aiding platform independence. The model can be expanded to other multi tasking concepts (semaphores, etc.) and to multiple processes.

Although modelled on the IOU pattern - which helps to simplify documentation - I have actually found the template to be of most us in simply starting threads without needing to code up API calls all the time.

I hope to revisit this subject in the near future by porting the template to use Posix threads and explore further, how generic classes and templates can be used to hide the complexities of multi-threading.

Thanks

Thanks to Jason Ross for helping with the original idea and drafts of this article.

Listings

Listing 1 - worker object and example of use

Listing 2 - thread template

Listing 3 - critical section

Listing 4 - critical section and threads

---

[1] Allan Vermeulen, Dr Dobbs Journal, June 1996

[2] Patrick Thompson, C++ Report, July/August 1998

[3] Bjarne Stroustrup, C++ Programming Language, 1997, section 14.4.1