<a href="http://www.omniture.com" title="Web Analytics"><img src="http://cmpglobalvista.112.2O7.net/b/ss/cmpglobalvista/1/H.16--NS/0" height="1" width="1" border="0" alt="" /></a>

Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Prefer Using Active Objects Instead of Naked Threads

How to automate best practices for threads and raise the semantic level of our code

June 14, 2010
URL:http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095

As described in Use Threads Correctly = Isolation + Asynchronous Messages[1]), to use threads well we want to follow these best practices:

- Keep data isolated, private to a thread where possible. Note that this doesn't mean using a special facility like thread local storage; it just means not sharing the thread's private data by exposing pointers or references to it.
- Communicate among threads via asynchronous messages. Using asynchronous messages lets the threads keep running independently by default unless they really must wait for a result.
- Organize each thread's work around a message pump. Most threads should spend their lifetime responding to incoming messages, so their mainline should consist of a message pump that dispatches those messages to the message handlers.

Using raw threads directly is trouble for a number of reasons, particularly because raw threads let us do anything and offer no help or automation for these best practices. So how can we automate them?

A good answer is to apply and automate the Active Object pattern [2]. Active objects dramatically improve our ability to reason about our thread's code and operation by giving us higher-level abstractions and idioms that raise the semantic level of our program and let us express our intent more directly. As with all good patterns, we also get better vocabulary to talk about our design. Note that active objects aren't a novelty: UML and various libraries have provided support for active classes. Some actor-based languages already have variations of this pattern baked into the language itself; but fortunately, we aren't limited to using only such languages to get the benefits of active objects.

This article will show how to implement the pattern, including a reusable helper to automate the common parts, in any of the popular mainstream languages and threading environments, including C++, C#/.NET, Java, and C/Pthreads. We will go beyond the basic pattern and also tie the active object's private thread lifetime directly to the lifetime of the active object itself, which will let us leverage rich object lifetime semantics already built into our various languages. Most of the sample code will be shown in C++, but can be directly translated into any of the other languages (for example, destructors in C++ would just be expressed as disposers in C# and Java, local object lifetime in C++ would be expressed with the using statement in C# or the "finally dispose" coding idiom in Java).

Active Objects: The Caller's View

First, let's look at the basics of how an active object is designed to work.

An active object owns its own private thread, and runs all of its work on that private thread. Like any object, an active object has private data and methods that can be invoked by callers. The difference is that when a caller invokes a method, the method call just enqueues a message to the active object and returns to the caller immediately; method calls on an active object are always nonblocking asynchronous messages. The active object's private thread mainline is a message pump that dequeues and executes the messages one at a time on the private thread. Because the messages are processed sequentially, they are atomic with respect to each other. And because the object's private data is only accessed from the private thread, we don't need to take a mutex lock or perform other synchronization on the private shared state.

Here is an example of the concurrency semantics we want to achieve for calling code:

```
Active a = new Active();
a.Func();         // call is nonblocking
… more work …  // this code can execute in parallel with a.Func()
```

Next, we want to tie the lifetime of the private thread to the lifetime of the active object: The active object's constructor starts the thread and the message pump. The destructor (or "disposer" in C# and Java) enqueues a sentinel message behind any messages already waiting in the queue, then joins with the private thread to wait for it to drain the queue and end. Note that the destructor/disposer is the only call on the active object that is a blocking call from the viewpoint of the caller.

Expressing thread/task lifetimes as object lifetimes in this way lets us exploit existing rich language semantics to express bounded nested work. For example, in C# or Java, it lets us exploit the usual language support and/or programming idiom for stack-based local lifetimes by writing a using block or the Dispose pattern:

```
// C# example
using( Active a = new Active() ) {    // creates private thread
       …
       a.SomeWork();                  // enqueues work
       …
       a.MoreWork();                   // enqueues work
       …
} // waits for work to complete and joins with private thread
```

As another example, in C++ we can also express that a class data member is held by value rather than by pointer or reference, and the language guarantees that the class member's lifetime is tied to the enclosing object's lifetime: The member is constructed when the enclosing object is constructed, and destroyed when the enclosing object is destroyed. (The same can be done by hand in C# or Java by manually wiring up the outer object's dispose function to call the inner one's.) For example:

```
// C++ example
class Active1 {        // outer object
     Active2 inner;     // embedded member, lifetime implicitly
                           // bound to that of its enclosing object
};

// Calling code
void SomeFunction() {
      Active1 a;    // starts both a's and a.inner's private threads
     …
}                    // waits for both a and a.inner to complete
```

That's what we want to enable. Next, let's consider how to actually write a class with objects that are active and behave in this way.

An Active Helper

The first thing we want to do is to automate the common parts and write them in one reusable place, so that we don't have to write them by hand every time for each new active class. To do that, let's introduce a helper class `Active` that encapsulates a thread behind a messaging interface and provides the basic message-sending and automatic pumping facilities. This section shows a straightforward object-oriented implementation that can be implemented as shown in any of our major languages (even in C, where the class would be written as a struct and the object's methods would be written as plain functions with an explicit opaque "this" pointer). After that, we'll narrow our focus to C++ specifically and look at another version we can write and use even more simply in that language.

The `Active` helper below provides the private thread and a message queue, as well as a sentinel done message that it will use to signal the private thread to finish. It also defines a nested `Message` class that will serve as the base of all messages that can be enqueued:

```
// Example 1: Active helper, the general OO way
//
class Active {
public:

  class Message {        // base of all message types
  public:
    virtual ~Message() { }
    virtual void Execute() { }
  };

private:
  // (suppress copying if in C++)

  // private data
  unique_ptr<Message> done;               // le sentinel
  message_queue< unique_ptr<Message> > mq;    // le queue
  unique_ptr<thread> thd;                // le thread
```

Note that I'll assume `message_queue` is a thread-safe internally synchronized message queue that doesn't need external synchronization by its caller. If you don't have such a queue type handy, you can use an ordinary queue type and protect it with a mutex.

With that much in place, Active can go on to provide the common machinery, starting with the message pump itself:

```
private:
  // The dispatch loop: pump messages until done
  void Run() {
    unique_ptr<Message> msg;
    while( (msg = mq.receive()) != done ) {
      msg->Execute();
    }
  }

public:
  // Start everything up, using Run as the thread mainline
  Active() : done( new Message ) {
    thd = unique_ptr<thread>(
                  new thread( bind(&Active::Run, this) ) );
  }

  // Shut down: send sentinel and wait for queue to drain
  ~Active() {
    Send( done );
    thd->join();
  }

  // Enqueue a message
  void Send( unique_ptr<Message> m ) {
    mq.send( m );
  }
};
```

The Active Cookbook: A Background Worker

Now let's use the helper and actually write an active class. To write a class with objects that are active, we must add an `Active` helper and then for each public method:

- Have the public method capture its parameters as a message and call `Send` to send the call for later execution on the private thread. Note that these functions should not touch any member variables. They should use and store their parameters and locals only.
- Provide a corresponding nonpublic class derived from `Message` with constructor and data members that will capture the parameters and this pointer, and with an `Execute` method that will perform the actual work (and can manipulate member variables; these `Execute` methods should be the only code in the class that does so).

For example, say we want to have an agent that does background work that we want to get off a responsive thread, including the performance of background save and print functions. Here's how we can write it as an active object that can be launched from the responsive thread using asynchronous method calls like `b.Save("filename.txt")` and `b.Print(myData)`:

```
class Backgrounder {
public:
  void Save( string filename ) {
    a.Send( new MSave( this, filename ) );
  }

  void Print( Data& data ) {
    a.Send( new MPrint( this, data ) );
  }

private:
  class MSave : public Active::Message {
    Backgrounder* this_;    string filename;
  public:
    MSave( Backgrounder* b, string f ) : this_(b), filename(f) { }
    void Execute() { … }    // do the actual work
  };

  class MPrint : public Active::Message {
    Backgrounder* this_;    Data& data;
  public:
    MPrint( Backgrounder* b, Data& d ) : this_(b), data(d) { }
    void Execute() { … }    // do the actual work
  };

  // Private state if desired
  PrivateData somePrivateStateAcrossCalls;

  // Helper goes last, for ordered destruction
  Active a;
};
```

Now we can simply enqueue work for the background thread as messages. If the worker's private thread is idle, it can wake up and start processing the message right away. If the worker is already busy, the message waits behind any other waiting messages and all get processed one after the other in FIFO order on the private thread. (Teaser: What if the background thread wants to report progress or output to the caller, or a side effect such as reporting that a `Print` message is done using the shared *data*? We'll consider that next month.)

A C++0x Version

The aforementioned OO-style helper works well in any mainstream language, but in any given language we can often make it even easier by using local language features and idioms. For example, if we were writing it in C++, we could observe that `Message` and its derived classes are simply applying the usual "OO way" of rolling your own function objects (or functors), and `Execute` could as well be spelled `operator()`, the function call operator. The only reason for the `Message` base class is to provide a way to hold and later invoke an arbitrary message, whereas in C++0x we already have `std::function<>` as a handy way to hold and later invoke any suitable callable function or functor.

So let's leverage the convenience of C++ function objects. We'll avoid a lot of the "OO Message hierarchy" boilerplate. Active will be a simpler class. Derived classes will be easier to write. What's not to like?

```
// Example 2: Active helper, in idiomatic C++(0x)
//
class Active {
public:
  typedef function<void()> Message;

private:

  Active( const Active& );           // no copying
  void operator=( const Active& );    // no copying

  bool done;                         // le flag
  message_queue<Message> mq;        // le queue
  unique_ptr<thread> thd;          // le thread
  void Run() {
    while( !done ) {
      Message msg = mq.receive();
      msg();            // execute message
    } // note: last message sets done to true
  }

public:

  Active() : done(false) {
    thd = unique_ptr<thread>(
                  new thread( [=]{ this->Run(); } ) );
  }

  ~Active() {
    Send( [&]{ done = true; } ); ;
    thd->join();
  }

  void Send( Message m ) {
    mq.send( m );
  }
};
```

Next, we can use the lambda functions language feature to make an implementing class like `Backgrounder` even simpler to write, because we don't even have to write the external "launcher" method and the actual body in different places…we can simply write each method the same way we write it naturally, and send the body of the message as a lambda function:

```
class Backgrounder {
public:
  void Save( string filename ) { a.Send( [=] {
    …
  } ); }

  void Print( Data& data ) { a.Send( [=, &data] {
    …
  } ); }

private:
  PrivateData somePrivateStateAcrossCalls;
  Active a;
};
```

This isn't just a C++ trick, by the way. If you're using C#, which also has generalized delegates and lambda functions, you can do likewise. Here's a sketch of how the simplified `Backgrounder` code would look in C#:

```
class Backgrounder : IDisposable {
  public Backgrounder() { /*…*/ a = new Active(); }
  public Dispose() { a.Dispose(); }

  public void Save( String filename ) { a.Send( () => {
    …
  } ); }

  public void Print( Data data ) { a.Send( () => {
    …
  } ); }

private:
  PrivateData somePrivateStateAcrossCalls;
  Active a;
};
```

Summary

Unlike with free threading, which lets us randomly do anything at all, active objects make it easier to express what we mean by automatically organizing the private thread's work around an event-driven message pump, naturally expressing isolated private data as simple member data, and offering strong lifetime guarantees by letting us express thread and task lifetime in terms of object lifetime (and therefore directly leverage the rich support for object lifetime semantics already built into our programming languages). All of this raises the semantic level of our program code, and makes our program easier to write, read, and reason about. Major uses for active objects include the same motivating cases as for any threads: to express long-running services (for example, a physics thread or a GUI thread); to decouple independent work (for example, background save, or pipeline stages); to encapsulate resources (for example, I/O streams or certain shared objects). You may never need or want to write a naked thread again.

Coming Up

So far, we've looked only at two of the basics of active objects, namely their lifetimes and asynchronous method call semantics. Next month, we'll complete the overview by considering important remaining details — how to handle a methods' return values and/or out parameters, and other kinds of communication back to the caller. Stay tuned.

References

[1] H. Sutter. "Use Threads Correctly = Isolation + Asynchronous Messages" (Dr. Dobb's Digest, April 2009: http://www.drdobbs.com/high-performance-computing/215900465).

[2] R. Lavender and D. Schmidt. "Active Object: An Object Behavioral Pattern for Concurrent Programming" (update of paper published in Pattern Languages of Program Design 2, Addison-Wesley, 1996: http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf).

[3] The ADAPTIVE Communication Environment (ACE): http://www.cs.wustl.edu/~schmidt/ACE.html.

Herb Sutter is a bestselling author and consultant on software development topics, and a software architect at Microsoft. He can be contacted at www.gotw.ca.