Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Know When to Use an Active Object Instead of a Mutex

Got state? Hide it! And if it's shared and either popular or slow, make it asynchronous...it's just good encapsulation

September 16, 2010
URL:http://www.drdobbs.com/architecture-and-design/know-when-to-use-an-active-object-instea/227500074

Let's say that your program has a shared log file object. The log file is likely to be a popular object; lots of different threads must be able to write to the file; and to avoid corruption, we need to ensure that only one thread may be writing to the file at any given time.

Quick: How would you serialize access to the log file?

Before reading on, please think about the question and pencil in some pseudocode to vet your design. More importantly, especially if you think this is an easy question with an easy answer, try to think of at least two completely different ways to satisfy the problem requirements, and jot down a bullet list of the advantages and disadvantages they trade off.

Ready? Then let's begin.

Option 1 (Easy): Use a Mutex (or Equivalent)

The most obvious answer is to use a mutex. The simplest code might look like that in Example 1(a):

```
// Example 1(a): Using a mutex (naive implementation)
//

// The log file, and a mutex that protects it
File logFile = …;
mutex_type mLogFile( … );

// Each caller locks the mutex to use the file
lock( mLogFile ) {
  logFile.write( … );
  logFile.write( … );
  logFile.write( … );
} // unlock
```

Example 1(a): Using a mutex (naive implementation)

If you've been paying attention to earlier installments of this column, you may have written it as shown in Example 1(b) instead, which lets us ensure that the caller doesn't accidentally write a race because he forgot to take a lock on the mutex (see [1] for details):

```
// Example 1(b): Using a mutex (improved implementation)
//

// Encapsulate the log file with the mutex that protects it
struct LogFile {
  // Hide the file behind a checked accessor
  // (see [1] for details)
  PROTECTED_WITH( mutex_type );
  PROTECTED_MEMBER( File, f );

  // A convenience method to avoid writing "f()" a lot
  void write( string x ) { f().write( x ); }
};

LogFile logFile;

// Each caller locks the entire thing to use the file
lock( logFile ) {
  logFile.f().write( … ); // we can use the f() accessor
             // explicitly
  logFile.write( … );    // but mostly let's use the
  logFile.write( … );    // convenience method
}
```

Example 1(b): Using a mutex (improved implementation)

Examples 1(a) and 1(b) are functionally equivalent, the latter is just more robust. Ignoring that for now, what are the advantages common to both expressions of our Option 1?

The main advantage of Option 1 is that it's correct and thread-safe. Protecting the log file with a mutex serializes callers to ensure that no two threads will be trying to write to the log file at the same time, so clearly we've solved the immediate basic requirement.

But is this the best solution? Unfortunately, Option 1 has two performance issues, one of them moderate and the other potentially severe.

The moderate performance problem is loss of concurrency among callers. If two calling threads want to write at the same time, one must block to wait for the other's work to complete before it can acquire the mutex to perform its own work, which loses concurrency and therefore performance.

The more serious issue is that using a mutex doesn't scale, and that becomes noticeable quickly for high-contention resources. Sharing is the root of all contention (see [2]), and there's plenty of potential contention here on this global resource. In particular, consider what happens when the log file is pretty popular, with lots of threads intermittently logging things, but the log file's write function is a slow, high-latency operation — it may be disk- or network-bound, unbuffered, or slow for other reasons. Say that a typical caller is calling `logFile.write` regularly, and that the calls to `logFile.write` take about 1/10 of the wall-clock time of the caller's computation. That means that 10% of a typical caller's time spent inside the lock — which means that at most 10 such threads can be active at once before they start piling up behind the lock and throttling each other. It's not really great to see the scalability of the entire program be limited to at most 10 such threads' worth of work.

We can do better. Given that there can be plenty of contention on this resource, the only winning strategy is not to share it…at least, not directly. Let's see how.

Option 2 (Better): Use Buffering, Preferably Asynchronous

One typical strategy for dealing with high-latency operations is to introduce buffering. The most basic kind of buffering is synchronous buffering; for example, we could do all the work synchronously inside the calls to `write`, but have most calls to `write` only add the data to an internal queue, so that `write` only actually writes anything to the file itself every *N*-th time, or if more than a second has elapsed (perhaps using a timer event to trigger occasional extra empty calls to `write` just to ensure flushing occurs), or using some other heuristic.

But this column is about effective concurrency, so let's talk about asynchronous buffering. Besides, it's better in this case because it gets much more of the work off the caller's thread.

A better approach in this case is to use a buffer in the form of a work queue that feeds a dedicated worker thread. The caller writes into the queue, and the worker thread takes items off the queue and actually performs the writing. Example 2 illustrates the technique:

```
// Example 2: Asynchronous buffering
//

// The log file, and a queue and private worker thread that
// protects it
message_queue<string> bufferQueue;

// Private worker thread mainline
File logFile = …;
while( str = bufferQueue.pop() ) {    // receive (async)
  // If the queue is empty, pop blocks until something is available.
  // Now, just do the actual write (now on the private thread).
  logFile.write( str );
}

// Each caller assembles the data they don't want interleaved
// with other output and just puts it into the buffer/queue
string temp = …;
temp.append( … );
temp.append( … );
bufferQueue.push( temp );        // send (async)
```

Example 2: Asynchronous buffering

Note that in this approach the individual calls to `send` on multiple threads are thread-safe, but they can interleave with each other. Therefore, a caller who wants to send several items that should stay together can't just get away with making several individual calls to `send`, but has to assemble them into an indivisible unit and send that all in one go, as shown above. This wasn't a problem in Option 1, because the indivisible unit of work was already explicit in the Example 1(a) and 1(b) calling code — while the lock was held, no other thread could get access to the file and so no other calls could interleave.

Another minor drawback is that we have to manage an extra thread, including that we have to account for its termination; somehow, the private thread has to know when to go away, and Example 2 leaves that part as an exercise for the reader. I call this issue "minor" because the extra complexity isn't much, and termination is easy to deal with in a number of ways (note that Option 1 had a similar termination issue, too, to make sure it destroyed the file object), but I mention it for completeness — if you use a strategy like Example 2, don't forget to join with those background helper threads at the end of the program!

But enough about minor drawbacks, because Option 2 delivers major advantages in the area of performance. Instead of waiting for an entire write operation to complete, possibly incurring high-latency accesses and all the trimmings, now the caller only has to wait for a simple and fast `message_queue.push` operation. By never executing any part of the actual write on the caller's thread, callers will never have to wait for each other for any significant amount of time even if two try to write at the same instant. By thus eliminating throttling, we

eliminate both performance issues we had with Option 1: We get much better concurrency among callers, and we eliminate the scalability problem inherent in the mutex-based design.

Guideline: Prefer to make high-contention and/or high-latency shared state, notably I/O, be asynchronous and therefore inherently buffered. It's just good encapsulation to hide the private state behind a public interface.

Oh, but wait — don't modern languages have something called "classes" to let us express this kind of encapsulation? Indeed they do, which brings us to Option 3…

Option 3 (Best): Use an Active Object to Encapsulate the Resource

In [3] and [4], we covered how to encapsulate a threads within an active object, which gives us a disciplined way to talk to the thread using plain old method calls that happen to be asynchronous, as well as to easily manage the thread and its lifetime just like any ordinary object. It turns out that this is a generally useful pattern, and if you didn't already believe me before, consider how naturally it helps us out with the `logFile` situation.

For Option 3, let's continue to buffer and do the work asynchronously just as we did in Option 2, except now use an active object to express it in code instead of having a distinct visible thread and message queue. Note that the queue buffer is still there, but now it's implicit and automated; we simply use the active object's message queue, and it happens naturally because we just turn `write` into an asynchronous method on the active object so that the actual `logFile.write` call is sent via the internal queue to be executed on the active object's hidden private thread:

```
// Example 3: Private -- expressed as active object (see [3] for details)
//
class AsyncLogFile {
public:
  void write( string str )
    { a.Send( [=] { log.write( str ); } ); }
private:
  File log;    // private file
  Active a;    // private helper (queue+thread)
};
AsyncLogFile logFile;

// Each caller just uses the active object directly
string temp = …;
temp.append( … );
temp.append( … );
logFile.write( temp );
```

This has all the benefits of Option2, including full concurrency and scalability, but expressing it this way is significantly better. Why?

First, it's simpler and better encapsulated. Instead of exposing a raw queue and helper thread, we instead raise the level of abstraction and provide a simpler and more natural object-based interface to calling code. The caller gets to use the original and natural `logFile.write "object.method"` syntax instead of dealing with an exposed message queue.

Second, shutdown is greatly simplified. Now, whenever we're done using the `logFile` and destroy it, it naturally performs its own orderly shutdown of the private thread, including that it correctly drains its remaining messages (if any) before returning from the destructor (see [3] for details). We no longer have to worry about special code to arrange shutdown of the helper thread, because we've expressed the thread as an

object and so we can just deal with its lifetime the same way we do with that of any regular object.

Summary

If you have high-contention and/or high-latency shared state, especially I/O, prefer to make it asynchronous. Doing so makes buffering implicit, because the buffering just happens as a side effect of the asynchrony. It also makes correct serialization implicit, because buffered operations are naturally performed in serial order by the single private thread that services them. Prefer to use the active object pattern as the most convenient and simple way to express an asynchronous object or resource.

In convenience, correctness, and (most especially) performance and scalability, this strategy can beat the pants off using a mutex to protect popular and/or slow shared resources. If you haven't tried this technique already in your code, pick a high-contention or high-latency shared resource that's currently protected with a mutex, test the performance of replacing the mutex with an active object, and see. Try it today.

References

[1] H. Sutter. "Associate Mutexes with Data to Prevent Races" *Dr. Dobb's Digest*, May 2010.

[2] H. Sutter. "Sharing Is the Root of All Contention" *Dr. Dobb's Digest*, March 2009.

[3] H. Sutter. "Prefer Using Active Objects Instead of Naked Threads" *Dr. Dobb's Digest*, June 2010).

[4] H. Sutter. "Prefer Using Futures or Callbacks to Communicate Asynchronous Results" *Dr. Dobb's Digest*, August 2010.

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.