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.

Associate Mutexes with Data to Prevent Races

Come together: Associate mutexes with the data they protect, and you can make your code race-free by construction

By Herb Sutter Dr.Dobb's Journal
May 13, 2010
URL : http://drdobbs.com/go-parallel/article/224701827

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.

Race conditions are one of the worst plagues of concurrent code: They can cause disastrous effects all the way up to undefined behavior and random code execution, yet they're hard to discover reliably during testing, hard to reproduce when they do occur, and the icing on the cake is that we have immature and inadequate race detection and prevention tool support available today.

The holy grail of the Consistency pillar is to make a concurrent program race-free and deadlock-free by construction. [1] "By construction" means to write our code in such a way that we can verify with confidence during development and testing that it does not contain races or deadlocks -- not just to rely on doing enough stress testing to get a probabilistic level of confidence, but to really know that there are no or deadlocks at all, at least for the shared data and locks that we control. I've written about some deadlock prevention techniques in "Use Lock Hierarchies to Avoid Deadlock." [2] This time, let's consider race prevention.

This article shows how to achieve the "race-free by construction" grail via sound engineering, with automatic and deterministic race identification at test time based on code coverage alone. Notes: For convenience, from now on I'm going to talk about just locks, but the issues and techniques apply to any kind of mutual exclusion. And although the example code I'll show will happen to be in C++, the techniques we'll discuss apply in any mainstream language, including C/Pthreads, C#, and Java.

The Problem

Let's say you want to use some shared data protected using mutexes:

```
// use table1 and table2 (both are shared)
table1.erase( x );
table2.insert( x );
```

Quick: What locks should you take to make this code correct?

The answer is usually to look in some documentation or comment, because the connection between a mutex and the data it protects usually exists primarily in the mind of the programmer, and we rely on discipline to remember which lock to take.

Let's say that **table1** and **table2** are protected by two mutexes **mutTable1** and **mutTable2**, respectively. Then we might write something like this:

```
lock mutTable1 and mutTable2

table1.erase( x );
table2.insert( x );

release mutTable1 and mutTable2 appropriately
(e.g., in finally blocks, or using automatic dispose/destructor methods)
```

The trouble is that the programmer has to remember that **mutTable1** and **mutTable2** are the "right" mutexes to take to use **table1** and **table2**. This has two main problems.

The minor problem is productivity: We're creating work for the developer by forcing him to look up which mutex to use, because in general he won't get automatic IDE completion/suggestion support to answer questions like, "Which mutex do I need to take to use **table2**?"

The major problem is correctness: We're relying on programmer discipline to make the code correct, and discipline can fail us because we will sometimes make mistakes. What can go wrong? We can fail to take the correct lock, in one of two ways:

- "Oops, forgot to lock." We can forget to take a lock at all, such as that in this example we might forget to take the lock on **mutTable2**.

- "Oops, took the wrong lock." We can inadvertently take a lock on the wrong mutex, either because of confusion about which mutex is the right one or because of a simple accident like a typo that misspells **mutTable2** as **mutTable3**.

The trouble is, either way we'll end up writing a race condition with a horrible failure mode: The code will silently compile and appear to work during testing, only to fail nondeterministically in production -- typically in the form of intermittent, timing-dependent, hard-to-reproduce bug reports from customer sites.

The Goal

Instead of relying on comments and discipline, what we'd like to do is state in code which mutex protects which data, so that we can build our own guardrails to ensure we did it right. It turns out that this is a powerful approach, because it lets us turn a class of nondeterministic run-time failures into ones we can detect:

- automatically,
- deterministically,
- at test time, and
- based on code coverage alone.

Let's see how.

Solution: Associate Data with Locks

Observation: If we knew which mutex covered which data, we could verify at test time that when we access shared data we're holding the right locks. So let's explicitly group the mutex and the data it protects together in code, in the same struct or class behind an interface that we can check at test time:

```
// Example 1: Illustrating the mutex association pattern
//
struct MyData {
public:
     // provide access to data
     vector<int>&    v()       {  assert( m_.is_held() );   return v_;   }
      Widget*  w()             {  assert( m_.is_held() );   return w_;  }
    // provide access to mutex
    void     lock()          {  m_.lock();  }
    bool     try_lock()      {  return m_.try_lock();  }
    void     unlock()        {  m_.unlock();  }
private:
    // hide
    vector<int>    v_;
    Widget*        w_;
    mutex_type     m_;
};
```

We require that callers always access the shared object through the accessor methods. That's enough to let us provide automated checking on every use of the shared object, and find many latent race conditions deterministically at test time based on code coverage alone.

Let's consider a few examples of how this helps verify that our program is race-free. First, what if we accidentally try to access the data without taking a lock at all? Consider:

```
MyData data1 = …, data2 = …;
// Later, but without having taken any locks:
data1.v().push_back( 10 );    // A: error: good, will assert
data2.w()->ProcessYearEnd();    // B: error: good, will assert
```

In line A, the call to **data1.v()** first performs **assert(data1.m_.is_held())**, and because the lock isn't held the program will stop with a test-time error and point directly to the offending line. Note that we will catch this error automatically and deterministically based on code coverage alone; as long as our testing exercises line A, we will discover the error the first time we try to execute the line. Similarly, line B's the call to **data2.w()** first performs **assert(data2.m_.is_held())**, which will fail deterministically and point directly to the offending line.

What if our testing doesn't do full code coverage, so that we can ship knowing we have no races? For example, what if during testing, we fail to exercise lines A and B at all, or they are callable along multiple paths -- each of which must take the lock -- and we fail to exercise each path at least once? Even with incomplete testing, the worst case is that the error will be consistently diagnosed in production the very first time the offending code path is actually exercised, with a clear diagnostic that points directly to the offending line -- whether or not the potential race condition actually manifests by having another thread performing a conflicting access at the same time. Unlike today, even in the worst case (including no testing at all), the faults are never intermittent, timing-dependent, or hard to reproduce.

Next, what if we try to access the data while having taken a lock, but it's the wrong lock? Consider:

```
{    // acquire lock on data1
     lock_guard<MyData> hold ( data1 );
     data1.v().push_back( 10 );    // C: ok
     data2.w()->ProcessYearEnd();    // D: error: good, will assert
}
```

In line C, the call to **data1.v()** again first performs **assert(data1.m_.is_held())**, and this time because the lock is held the program continues normally, having validated that the access to **data1.v_** is not a race.

In line D, however, the call to **data2.w()** first performs **assert(data2.m_.is_held())**, which fails because although we are holding a lock, we're not holding the one on **data2**. Again, because the lock isn't held the program will stop with a test-time error and point directly to the offending line.

There is only one specific abusive caller pattern that can't be checked automatically using this technique. Callers that remember a direct pointer or reference to the object for later use must be prevented by programmer education and team policy:

```
// The only hole: Require and enforce that programmers don't do this.
//
vector<int>* sneaky = nullptr;
{              // enter critical section
     lock_guard<MyData> hold( data1 );
     sneaky = &data1.v();          // E: compiles, but avoid doing this
     sneaky->push_back( 10 );    // F: ok, but not checked
}
sneaky->push_back( 10 );      // G: error: race, but won't assert
```

In line E, we correctly verify that **data1.m._is_held()** and allow access to the member, but the caller remembers a sneaky pointer or reference to the object. That's unsafe. It's not inherently always wrong: For example, the access in line F is still correct and not a race, but the problem is that we can no longer verify that it's correct. It's unsafe because it opens up the one hole demonstrated in line G, where the call to **sneaky->push_back(10)** is a race, but still compiles and runs and won't be caught as it bypasses the validation.

The mutex association pattern catches both the "Oops, forgot to lock" and "Oops, took the wrong lock" mistakes. As a bonus, it also provides a reasonable migration for existing source code, where the impact on existing calling code can sometimes be made as small as just adding **()**or a similar minor syntactic change.

Finally, and this can't really be overemphasized -- this is a vast improvement over intermittent timing-dependent hard-to-reproduce bug reports from customers.

Automation

The pattern we've presented is sound, but as we apply it to the shared data in our system we'll find that some parts of the pattern are repetitive (e.g., the **lock/try_lock/unlock** code). It would be nice to automate the boring parts. Here's one way.

First, an initial optional step: We might need to make our mutex testable if it isn't already. Many mutex services (including C++0x **std::mutex**) don't provide a way to ask, "Do I already hold a lock on this mutex?" But we need a method like **is_held** to be available. Fortunately, we can easily get it by wrapping together an arbitrary mutex type with a thread ID that stores the ID of the thread that currently holds the mutex. The thread ID initially holds an invalid value that means no threads own the mutex. When we acquire a lock, we set the ID; and when we release a lock, we reset it back to its default nobody-loves-me state:

```
template<typename Mutex>
class TestableMutex {
public:
     void lock()        {  m.lock();  id = this_thread::get_id();  }
     void unlock()    {  id = 0;  m.unlock();  }
     bool try_lock() {  bool b = m.try_lock();
                 if( b ) id = this_thread::get_id();
                 return b;  }
     bool is_held()    { return id == this_thread::get_id(); }
private:
     Mutex m;
     atomic<thread::id> id;
```

```
     // for recursive mutexes, add a count
};
```

Now we can automate the boring parts of the mutex association pattern. For example, given the following convenience macros:

```
#define PROTECTED_WITH(MutType)  \
     public:    void    lock()            {  mut_.lock();  } \
     public:    bool     try_lock()    {  return mut_.try_lock();  } \
     public:    void     unlock()      {  mut_.unlock();  } \
     private:    TestableMutex<MutType> mut_;
#define PROTECTED_MEMBER(Type,name) \
     public:    Type&    name()     { assert(mut_.is_held()); return
name##_; } \
     private:    Type    name##_;
```

Now we associate data with a mutex more easily:

```
// Example 2: Example 1 rewritten using the convenience macros.
//
struct MyData {
     PROTECTED_WITH( some_mutex_type );
     PROTECTED_MEMBER( vector<int>, v );
     PROTECTED_MEMBER( Widget*, w );
};
```

The above is identical to the longer version we wrote out by hand in Example 1. If you don't like macros, or don't have them in your language, you can use other methods to get a similar effect.

Granular Control

But what if we already have struct or data that naturally goes together, but different groups of members are covered by different mutexes? To express that, first just define each group on its own, either by hand as in Example 1 or by using a convenience library as in Example 2. For convenience, I'll use the latter:

```
struct Group1 {
     PROTECTED_WITH( some_mutex_type );
     PROTECTED_MEMBER( vector<int>, v );
     PROTECTED_MEMBER( Widget*, w );
};
struct Group2 {
     PROTECTED_WITH( some_mutex_type );
     PROTECTED_MEMBER( (map<string,list<int>>), map_from_2 );
};
struct Group3 {
     PROTECTED_WITH( some_other_mutex_type );
     PROTECTED_MEMBER( complex<float>, cf );
};
```

Then combine them:

```
struct MyData { Group1 group1; Group2 group2; Group3 group3; };
// Sample use
{
  lock_guard<Group1> lock( data.group1 );
  data.group1.v().push_back( 100 );
}
```

Summary

In code, associate mutexes with the data they control. Leverage this to find latent race conditions automatically and deterministically at test time based on code coverage alone, and ensure before shipping that you didn't write a race by forgetting to take a lock, or taking the wrong lock, in any of the code that your team controls. Then your code will be in the best possible place: Not racy, and not even just accidentally race-free, but race-free by construction.

References

[1] H. Sutter. "The Pillars of Concurrency" (Dr. Dobb's Journal, August 2007; http://www.drdobbs.com/high-performance-computing/200001985).

[2] H. Sutter. "Use Lock Hierarchies to Avoid Deadlock" (Dr. Dobb's Journal, January 2008; http://www.drdobbs.com/high-performance-computing/204801163).

The Pillars of Concurrency

In his inaugural column, Herb makes the case that we must build a consistent mental model before talking about concurrency.

By Herb Sutter
July 02, 2007
URL:http://drdobbs.com/high-performance-computing/200001985

Herb is a software architect at Microsoft and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

Quick: What does "concurrency" mean to you? To your colleagues? To your project?

You probably know the fable of the blind men and the elephant: Several blind men explore an elephant by feel, and each one draws his own conclusion about what the elephant is. The man at the tail concludes that the elephant is like a rope; the one at the tusk asserts that, no, the elephant is like a spear; the one at the leg thinks it's like a tree; the one at the ear that it's like a fan; and so on. The blind men argue, each one knowing he is "right" from his own experience and point of view. In time, several give up hope of ever agreeing on what the creature is, and some resign themselves to accepting that the best they can do is deeply

analyze each disconnected part in isolation as its own separate self. Meanwhile, the poor elephant waits patiently, faintly puzzled, because of course, it is not like any of those things alone and it doesn't see itself as that hard to understand as a whole creature.

Have you ever talked with another developer about concurrency, and felt as though you were somehow speaking completely different languages? If so, you're not alone. You can see the confusion in our vocabulary (and this is not an exhaustive list):

acquire, and-parallel, associative, atomic, background, cancel, consistent, data-driven, dialogue, dismiss, fairness, fine-grained, fork-join, hierarchical, interactive, invariant, isolation, message, nested, overhead, performance, priority, protocol, read, reduction, release, structured, repeatable, responsiveness, scalable, schedule, serializable update, side effect, systolic, timeout, transaction, throughput, virtual, wait, write,...

Some of the words have many meanings (performance, for instance), while others are unrelated (responsiveness and throughput, for example). Much of the confusion we may encounter arises when people inadvertently talk past each other by using incompatible words.

A basic problem is that this shouldn't be a single list. But how can we group these terms sensibly, when experienced parallel programmers know scores or hundreds of different concurrency requirements and techniques that seem to defy grouping?

Callahan's Pillars

My colleague David Callahan leads a team within Visual Studio that is working on programming models for concurrency. He has pointed out that fundamental concurrency requirements and techniques fall into three basic categories, or pillars. They are summarized in Table 1 [1]. Understanding these pillars gives us a framework for reasoning clearly about all aspects of concurrency—from the concurrency requirements and tradeoffs that matter to our current project, to why specific design patterns and implementation techniques are applicable to getting specific results and how they are liable to interact, and even to evaluating how future tools and technologies will fit with our needs.

Let's consider an overview of each pillar in turn, note why techniques in different pillars compose well, and see how this framework helps to clarify our vocabulary.

[Click image to view at full size]

Pillar	1. Responsiveness and Isolation Via Asynchronous Agents	2. Throughput and ScalabilityVia Concurrent Collections
Tagline	Less coupling/blocking	Re-enable the free lunch
Summary	Stay responsive by running tasks independently and tasks asynchronously, communicating via messages	Use more cores to get theanswer faster by runnin operations on groupsof things; exploit parallelismi data and algorithmstructures
Examples	GUIs, web services, background print/compile	Trees, quicksort,compilation
Key metrics	Responsiveness	Throughput, scalability
Requirements	Isolated, independent	Low overhead
Today's Mainstream Tools	Threads, thread pools, message queues, futures	Thread pools, futures, OpenMP
Tomorrow's Tools?	Active objects/services, channels, contracts	Chores, work stealing,parallel STL & LINQ
Sample Vocabulary	background, cancel, dismiss, dialogue, fairness, fork-join, interactive, isolation, message, overhead, performance, priority,protocol, responsiveness, schedule, timeout, wait	and-parallel, associative,data-driven, fine-grained,fork-join, hierarchical, nested,overhead, performance, reduction,repeatable, scalable, schedule,serializab side effect, structured, systolic, throughput

Table 1: The Pillars of Concurrency.

Pillar 1: Responsiveness and Isolation Via Asynchronous Agents

Pillar 1 is all about running separate tasks, or agents, independently and letting them communicate via asynchronous messages. We particularly want to avoid blocking, especially on user-facing and other time-sensitive threads, by running expensive work asynchronously. Also, isolating separable tasks makes them easier to test separately and then deploy into various parallel contexts with confidence. Here we use key terms like "interactive" and "responsive" and "background"; "message" and "dialogue"; and "timeout" and "cancel."

A typical Pillar 1 technique is to move expensive work off an interactive application's main GUI pump thread. We never want to freeze our display for seconds or longer; users should still be able to keep clicking away and interacting with a responsive GUI while the hard work churns away in the background. It's okay for users to experience a change in the application while the work is being performed (for example, some buttons or menu items might be disabled, or an animated icon or progress bar might indicate status of the background work), but they should never experience a "white screen of death"—a GUI thread that stops responding to basic messages like "repaint" for a while because the new messages pile up behind one that's taking a long time to process synchronously. Typically, users keep trying to click on things anyway to wake the application up, and finally either give up and kill the application because they think it crashed (possibly losing or corrupting work, even though the

program would eventually have started responding again!) or else wait it out only to experience a flood where all the clicks they tried to perform finally get processed, usually with unsatisfactory or wrong results. Don't let this happen to your application, even if big companies let it happen to theirs.

So what kind of work do we want to ship out of responsiveness-sensitive threads? It can be work that performs an expensive or high-latency computation (background compilation or print rendering, for instance) or actual blocking (idle waiting for a lock, a database result, or a web service reply). Some of these tasks merely want to return a value; others will interact more to provide intermediate results or accept additional input as they make progress on their work.

Finally, how should the independent tasks communicate? A key is to have the communication itself be asynchronous, preferably using asynchronous messages where possible because messages are nearly always preferable to sharing objects in memory (which is Pillar 3's territory). In the case of a GUI thread, this is an easy fit because GUIs already use message-based event-driven models.

Today, we typically express Pillar 1 by running the background work on its own thread or as a work item on a thread pool; the foreground task that wants to stay responsive is typically long-running and is usually a thread; and communication happens through message queues and message-like abstractions like futures (Java *Future*, .NET *IAsyncResult*). In coming years, we'll get new tools and abstractions in this pillar, where potential candidates include active objects/services (objects that conceptually run on their own thread, and calling a method is an asynchronous message); channels of communication between two or more tasks; and contracts that let us explicitly express, enforce, and validate the expected order of messages.

This pillar is not about keeping hundreds of cores busy; that job belongs to Pillar 2. Pillar 1 is all about responsiveness, asynchrony, and independence; but it may keep some number of cores busy purely as a side effect, because it still expresses work that can be done independently, and therefore, in parallel.

Pillar 2: Throughput and Scalability Via Concurrent Collections

Pillar 2, on the other hand, is about keeping hundreds of cores busy to compute results faster, thereby re-enabling the "free lunch"[2]. We particularly want to target operations performed on collections (any group of things, not just containers) and exploit parallelism in data and algorithm structures. Here, we use key terms like "scalability" and "throughput"; "data-driven" and "fine-grained" and "schedule"; and "side effect" and "reduction."

New hardware no longer delivers the "free lunch" of automatically running single-threaded code faster to the degree it did historically. Instead, it provides increasing capacity to run more tasks concurrently on more CPU cores and hardware threads. How can we write applications that will regain the free lunch, that we can ship today and know they will naturally execute faster on future machines having ever greater parallelism.

The key to scalability is not to divide the computation-intensive work across some fixed number of explicit threads hard-coded into the structure of the application (for instance, when a game might try to divide its computation work among a physics thread, a rendering thread,

and an everything-else thread). As we'll see next month, that path leads to an application that prefers to run on some constant number *K* of cores, which can penalize a system with fewer than *K* cores and doesn't scale on a system with more than *K* cores. That's fine if you're targeting a known fixed hardware configuration, like a particular game console whose architecture isn't going to change until the next console generation, but it isn't scalable to hardware that supports greater parallelism.

Rather, the key to scalability is to express lots of latent concurrency in the program that scales to match its inputs (number of messages, size of data). We do this in two main ways.

The first way is to use libraries and abstractions that let you say what you want to do rather than specifically how to do it. Today, we may use tools like OpenMP to ask to execute a loop's iterations in parallel and let the runtime system decide how finely to subdivide the work to fit the number of cores available. Tomorrow, tools like parallel STL and parallel LINQ [5] will let us express queries like "select the names of all undergraduate students sorted by grade" that can be executed in parallel against an in-memory container as easily as they are routinely executed in parallel by a SQL database server.

The second way, is to explicitly express work that can be done in parallel. Today, we can do this by explicitly running work items on a thread pool (for instance, using Java *ThreadPoolExecutor* or .NET *BackgroundWorker*). Just remember that there is overhead to moving the work over to a pool, so the onus is on us to make sure the work is big enough to make that worthwhile. For example, we might implement a recursive algorithm like quicksort to at each step sort the left and right subranges in parallel if the subranges are large enough, or serially if they are small. Future runtime systems based on work stealing will make this style even easier by letting us simply express all possible parallelism without worrying if it's big enough, and rely on the runtime system to dynamically decide not to actually perform the work in parallel if it isn't worth it on a given user's machine (or with the load of other work in the system at a given time), with an acceptably low cost for the unrealized parallelism (for example, if the system decides to run it serially, we would want the performance penalty compared to if we had just written the recursive call purely serially in the first place to be similar to the overhead of calling an empty function).

Pillar 3: Consistency Via Safely Shared Resources

Pillar 3 is all about dealing with shared resources, especially shared memory state, without either corruption or deadlock. Here we use key terms like acquire and release; read and write; and atomic and consistent and transaction. In these columns, I'll mostly focus on dealing with mutable objects in shared memory.

Today's status quo for synchronizing access to mutable shared objects is locks. Locks are known to be inadequate (see [3] and [4]), but they are nevertheless the best general-purpose tools we have. Some frameworks provide selected lock-free data structures (hash tables) that are internally synchronized using atomic variables so that they can be used safely without taking locks either internally inside the data structure implementation or externally in your calling code; these are useful, but they are not a way to avoid locking in general because they are few and many common data structures have no known lock-free implementations at all.

In the future, we can look forward to improved support for locks (for example, being able to express lock levels/hierarchies in a portable way, and what data is protected by what lock)

and probably transactional memory (where the idea is to automatically version memory, so that the programmer can just write "begin transaction; do work; end transaction" like we do with databases and let the system handle synchronization and contention automatically). Until we have those, though, learn to love locks.

Composability: More Than The Sum of the Parts

Because the pillars address independent issues, they also compose well, so that a given technique or pattern can apply elements from more than one category.

For example, an application can move an expensive tree traversal from the main GUI thread to run in the background to keep the GUI free to pump new messages (responsiveness, Pillar 1), while the tree traversal task itself can internally exploit the parallelism in the tree to traverse it in parallel and compute the result faster (throughput, Pillar 2). The two techniques are independent of each other and target different goals using different patterns and techniques, but can be used effectively together: The user has an application that is responsive no matter how long the computation takes on a less-powerful machine; he also has a scalable application that runs faster on more powerful hardware.

Conversely, you can use this framework as a tool to decompose concurrency tools, requirements, and techniques into their fundamental parts. By better understanding the parts and how they relate, we can get a more accurate understanding of exactly what the whole is trying to achieve and evaluate whether it makes sense, whether it's a good approach, or how it can be improved by changing one of the fundamental pieces while leaving the others intact.

Summary

Have a consistent mental model for reasoning about concurrency—including requirements, tradeoffs, patterns, techniques, and technologies both current and future. Distinguish among the goals of responsiveness (by doing work asynchronously), throughput (by minimizing time to solution), and consistency (by avoiding corruption due to races and deadlocks).

In future columns, I'll dig into various specific aspects of these three pillars. Next month, we'll answer the question, "how much concurrency does your application have or need?" and distinguish between *O(1)*, *O(K)*, and *O(N)* concurrency. Stay tuned.

Notes

[1] The elephant analogy and the pillar segmentation were created by David Callahan (www.microsoft.com/presspass/exec/de/Callahan/ default.mspx) in an unpublished work.

[2] H. Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software" (www.ddj.com/dept/architect/184405990).

[3] H. Sutter. "The Trouble With Locks" (www.ddj.com/dept/cpp/184401930)

[4] H. Sutter and J. Larus. "Software and the Concurrency Revolution" (*ACM Queue*, September 2005). (gotw.ca/publications/concurrency-acm.htm).

[5] J. Duffy, http://www.bluebytesoftware.com/ blog/PermaLink,guid,81ca9c00-b43e-4860-b96b-4fd2bd735c9f.aspx.

Prefer Using Futures or Callbacks to Communicate Asynchronous Results

Active objects offer an important abstraction above raw threads. In a previous article, we saw how active objects let us hide the private thread, deterministically organize the thread's work, isolate its private data, express asynchronous messages as ordinary method calls, and express thread/task lifetimes as object lifetimes so that we can manage both using the same normal language features. [1]

What we didn't cover, however, was how to handle methods' return values and/or "out" parameters, and other kinds of communication back to the caller. This time, we'll answer the following questions:

- How should we express return values and out parameters from an asynchronous function, including an active object method?
- How should we give back multiple partial results, such as partial computations or even just "percent done" progress information?
- Which mechanisms are suited to callers that want to "pull" results, as opposed to having the callee "push" the results back proactively? And how can "pull" be converted to "push" when we need it?

Let's dig in.

Getting Results: Return Values and "Out" Parameters

First, let's recall the active object example we introduced last time.

We have a GUI thread that must stay responsive, and to keep it ready to handle new messages we have to move "save this document," "print this document," and any other significant work off the responsive thread to run asynchronously somewhere else. One way to do that is to have a background worker thread that handles the saving and print rendering work. We feed the work to the background thread by sending it asynchronous messages that contain the work to be performed; the messages are queued up if the worker thread is already busy, and then executed sequentially on the background worker thread.

The following code expresses the background worker using a **Backgrounder** class that follows the active object pattern. The code we'll show uses C++0x syntax, but can be translated directly into other popular threading environments such as those provided by Java, .NET, and Pthreads (see [1] for a discussion of the pattern and translating the code to other environments).

The **Active** helper member encapsulates a private thread and message queue, and each **Backgrounder** method call simply captures its parameters and its body (both conveniently

automated by using lambda function syntax) and Send that as an asynchronous message that's fired off to be enqueued and later executed on the private thread:

```
// Baseline example
class Backgrounder {
public:
  void Save( string filename ) { a.Send( [=] {
    // … do the saving work …
  } ); }

  void Print( Data& data ) { a.Send( [=, &data] {
    do {
      // … do the printing work for another piece of the data …
    } while( /* not done formatting the data */ );
  } ); }

private:
  PrivateData somePrivateStateAcrossCalls;
  Active a;    // encapsulates a private thread, and
};    // pumps method calls as messages
```

The GUI thread instantiates a single **Backgrounder** object (and therefore a single background worker thread), which might be used from the GUI thread as follows:

```
class MyGUI {
public:
  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    backgrounder.Save( filename );
      // this fires off an asynchronous message
  } // and then we return immediately to the caller

  // …

private:
  Backgrounder backgrounder;
};
```

This illustrates the ability to treat asynchronous messages like normal method calls and everything is all very nice, but you might have noticed that the **Save** and **Print** methods (in)conveniently don't have any return value or "out" parameters, don't report their progress or intermediate results, nor communicate back to the caller at all. What should we do if we actually want to see full or partial results?

Option 1: Return a Future (Caller Naturally "Pulls")

First, let's deal with just the return value and output parameters, which should be somehow communicated back to the caller at the end of the asynchronous method. To keep the code

simple, we'll focus on the primary return value; output parameters are conceptually just additional return values and can be handled the same way.

For an asynchronous method call, we want to express its return value as an asynchronous result. The default tool to use for an asynchronous value is a "future" (see [2]). To keep the initial example simple, let's say that **Save** just wants to return whether it succeeded or failed, by returning a bool:

```
// Example 1: Return value, using a future
class Backgrounder {
public:
  future<bool> Save( string filename ) {
    // Make a future (to be waited for by the caller)
    // connected to a promise (to be filled in by the callee)
    auto p = make_shared<promise<bool>>();
    future<bool> ret = p->get_future();
    a.Send( [=] {
      // … do the saving work …
      p->set_value( didItSucceed() ? true : false );
    } );
    return ret;
  }
```

(C++0x-specific note: Why are we holding the promise by reference-counted smart pointer? Because promise is a move-only type and C++ lambdas do not yet support move-capture, only capture-by-value and capture-by-reference. One simple solution is to hold the promise by **shared_ptr**, and copy that.)

Now the caller can wait for the "future":

```
future<bool> result = backgrounder.Save( filename );
…
… this code can run concurrently with Save()
…
Use( result.get() );  // block if necessary until result is available
```

This works, and returning a "future" is generally a useful mechanism.

However, notice that waiting for a "future" is inherently a "pull" operation; that is, the caller has to ask for the result when it's needed. For callers who want to find out if the result is ready without blocking if it isn't, "future" types typically provide a status method like **result.is_ready()** for the caller to check without blocking, which he can do in a loop and then sleep in between calls -- that's still a form of polling loop, but at least it's better than burning CPU cycles with outright busy-waiting.

So, although the caller isn't forced to busy-wait, the onus is still on him to act to "pull" the value. What can we do if instead the caller wants a "push" notification sent to him proactively when the result is available? Let's consider two ways, which we'll call Options 2 and 3.

Option 2: Return a Future (Caller Converts "Pull" Into "Push")

The "pull" model is great for many uses, but the example caller we saw above is a must-stay-responsive GUI thread. That kind of caller certainly doesn't want to wait for the "future" on any GUI thread method, because responsive threads must not block or stop processing new events and messages. There are workarounds, but they're not ideal: For example, it's possible for the GUI thread to remember there's a "future" and check it on each event that gets processed, and to make sure it sees it soon enough it can generate extra timer events to be woken up just so it can check the "future" -- but that seems (and is) a lot harder than it should be.

Given that a responsive thread like a GUI thread is already event-driven, ideally we would like it to be able to receive the result as a new queued event message that it can naturally respond to just like anything else.

Option 2 is to have the caller do the work to convert "pull" to "push." In this model, the callee still returns a "future" as in Option 1, and it's up to the caller turn it into a proactive result. How can the caller do that? One way is to launch a one-off asynchronous operation that just waits for the "future" and then generates a notification from the result. Here's an example:

```
// Example 2(a): Calling code, taking result as a future
// and converting it into an event-driven notification
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    shared_future<bool> result;
    result = backgrounder.Save( filename );
    // queue up a continuation to notify ourselves of the
    // result once it's available
    async( [=] { SendSaveComplete( result.get() ); } );
  }
  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

The statement **SendSaveComplete(result->get());** does two things: First, it executes **result->get()**, which blocks if necessary to wait for the result to be available. Then, and only then, it calls **SendSaveComplete**; in this case, an asynchronous method that when executed ends up calling **OnSaveComplete** and passes the available result. (C++0x-specific note: Like promises, ordinary "futures" are intended to be unique and therefore not copyable, but are move-only, so again we use the **shared_ptr** workaround to enable copying the result into the lambda for later use.)

Option 2 Variant: ContinueWith

As written, the Example 2(a) code has two disadvantages:

- First, it spins up (and ties up) a thread just to keep the asynchronous operation alive, which is potentially unnecessary because the first thing the asynchronous operation does is go idle until something happens.
- Second, it incurs an extra wakeup, because when the "future" result is available, we need to wake up the asynchronous helper operation's thread and continue there.

Some threading platforms offer a "future-like" type that has a **ContinueWith**-style method to avoid this overhead; for example, see **.NET's Task<T>**. [3] The idea is that **ContinueWith** takes a continuation to be executed once the thread that fills the "future" makes the "future" ready, and the continuation can be executed on that same target thread.

Tacking the continuation onto the "future-generating" work itself with **ContinueWith**, rather than having to use yet another fresh async operation as in Example 2(a), lets us avoid both of the problems we just listed: We don't have to tie up an extra thread just to tack on some extra work to be done when the "future" is ready, and we don't have to perform a wakeup and context switch because the continuation can immediately run on the thread that fills the "future." For example:

```
// Example 2(b): Calling code, same as 2(a) except using
// ContinueWith method (if available)
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    shared_future<bool> result;
    result = backgrounder.Save( filename );
    // queue up a continuation to notify ourselves of the
    // result once it's available
    result.ContinueWith(  [=]
      SendSaveComplete( result->get() );
    } );
  }
  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

Prefer to use a **ContinueWith** style if it is available in your "futures library."

Option 3: Accept an Event or Callback (to "Push" to Caller)

Both of the alternatives we've just seen let the callee return a "future," which by default delivers a "pull" notification the caller can wait for. So far, we've left it to the caller to turn that "future" into a "push" notification (event or message) if it wants to be proactively notified when the result is available.

What if we want our callee to always offer proactive "push" notifications? The most general way to do that is to accept a callback to be invoked when the result is available:

```
// Example 3: Return value, using a callback
class Backgrounder {
public:
  void Save(
    string filename,
    function<void(bool)> returnCallback
  ) {
    a.Send( [=] {
      // … do the saving work …
      returnCallback( didItSucceed() ? true : false );
    } ); }
```

This is especially useful if the caller is itself an active object and gives a callback that is one of its own (asynchronous) methods. For example, this might be used from a GUI thread as follows:

```
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    // pass a continuation to be called to give
    // us the result once it's available
    shared_future<bool> result;
    result = backgrounder.Save( filename,
       [=] { SendSaveComplete( result->get() ); } );
  }

  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

Since Example 3 uses a callback, it's worth mentioning a drawback common to all callback styles, namely: The callback runs on the callee's thread. In the aforementioned code that's not a problem because all the callback does is launch an asynchronous message event that gets queued up for the caller. But remember, it's always a good idea to do as little work as possible in the callee, and just firing off an asynchronous method call and immediately returning is a good practice for callbacks.

Getting Multiple or Interim Results

All of the aforementioned options deal well with return values and output parameters. Finally, however, what if we want to get multiple notifications before the final results, such as partial computation results, updated status such as progress updates, and so on?

We have two main options:

- Provide an explicit message queue or channel back to the caller, which can enqueue multiple results.
- Accept a callback to invoke repeatedly to pass multiple results back to the caller.

Example 4 will again use the callback approach. If the caller is itself an active object and the callback it provides is one of its own (asynchronous) methods, we've really combined the two paths and done both bullets at the same time. (Note: Here we're focusing on the interim progress via the **statusCallback;** for the return value, we'll again just use a "future" as in Examples 1 and 2.)

```
// Example 4: Returning partial results/status
class Backgrounder {
public:
  // Print() puts print result into spooler, returns one of:
  //   Error (failed, couldn't process or send to spooler)
  //   Printing (sent to spooler and already actively printing)
  //   Queued (sent to spooler but not yet actively printing)
  future<PrintStatus>
  Print(
    Data& data,
    function<void(PrintInfo)> statusCallback
  ) {
    auto p = make_shared<promise<PrintStatus>>();
    future<PrintStatus> ret = p->get_future();
    a.Send( [=, &data] {
      PrintInfo info;
      while( /* not done formatting the data */ ) {
        info.SetPercentDone( /*…*/ );
        statusCallBack( info );      // interim status
        // … do the printing work for another piece of the data …
      } while( /* not done formatting the data */ );
      p->set_value( /* … */ );        // set final result
      info.SetPercentDone( 100 );
      statusCallBack( info );          // final interim status
    } );
    return ret;
  }
```

This might be used in the GUI thread example as follows:

```
class MyGUI {
public:
  // …

  // When the user clicks [Print]
  void OnPrintClick() {
    // …
    // … turn on printing icon, etc. …
    // …
    // pass a continuation to be called to give
    // us the result once it's available
    shared_future<PrintStatus> result;
    result = backgrounder.Print( theData,
       [=]( PrintInfo pi ) { SendPrintInfo( pi, result ); } );
  }
```

```
void OnPrintInfo(
  PrintInfo pi,
  shared_future<PrintStatus> result
) {
  // … update print progress bar to
  //     pi.GetPercentDone(), etc. …
  // if this is the last notification
  // (100% done, or result is ready)
  if( result.is_ready() ) {
    // … turn off printing icon, etc. …
  }
}
```

Summary

To express return values and "out" parameters from an asynchronous function, including an active object method, either:

- Return a "future" to invoke that the caller can "pull" the result from (Example 1) or convert it to a "push" (Examples 2(a) and 2(b), and prefer to use **ContinueWith** where available); or
- Accept a callback to invoke to "push" the result to the caller when ready (Example 3).

To return multiple partial results, such as partial computations or even just "percent done" progress information, also use a callback (Example 4).

Whenever you provide callbacks, remember that they are running in the callee's context, so we want to keep them as short and noninvasive as possible. One good practice is to have the callback just fire off asynchronous messages or method calls and return immediately.

On Deck

Besides moving work off to a background thread, what else could we use an active object for? We'll consider an example next time, but for now, here's a question for you to ponder: How might you use an active object to replace a mutex on some shared state? Think about ways you might approach that problem, and we'll consider an example in my next column.

References

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

[2] H. Sutter. Prefer Futures to Baked-In 'Async APIs'. Dr. Dobb's Digest, January 2010.

[3] Task.ContinueWith Method (Action<Task>); MSDN.

Prefer Using Active Objects Instead of Naked Threads

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

By Herb Sutter Dr.Dobb's Journal
June 14, 2010
URL : http://drdobbs.com/go-parallel/article/225700095

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.

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.

Prefer Futures to Baked-In "Async APIs"

When designing concurrent APIs, separate "what" from "how"

By Herb Sutter Dr.Dobb's Journal
January 15, 2010
URL : http://drdobbs.com/go-parallel/article/222301165

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.

Let's say you have an existing synchronous API function:

```
// Example 1: Original synchronous API
// that could take a long time to execute
// (to compute, to wait for disk/web, etc.)
//
RetType DoSomething(
  InParameters ins,
  OutParameters outs
);
```

Because **DoSomething** could take a long time to execute (whether it keeps a CPU core busy or not), and might be independent of other work the caller is doing, naturally the caller might want to execute **DoSomething** asynchronously. For example, consider calling code like this:

```
// Example 1, continued:
// Sample calling code
//
void CallerMethod() {
  //

  result = DoSomething( this, that, outTheOther );

  // These could also take a long time
  OtherWork();
  MoreOtherWork();

  //  now use result and outOther
}
```

If **OtherWork** and **MoreOtherWork** don't depend on the value of result or other side effects of **DoSomething**, and if they can correctly run concurrently with **DoSomething** because they don't conflict by using the same data or resources, then it could be useful to execute **DoSomething** concurrently with **OtherWork/MoreOtherWork**.

The question is, how should we enable that? There is a simple and correct answer, but because many interfaces have opted for a more complex answer let's consider that one first.

Option 1: Explicit Begin/End (Poor)

One option is for the API itself to provide an asynchronous version of the function. This is such a popular option that there are patterns for it, each of which has been reinvented with variations a number of times.

One form of pattern that is commonly used in .NET (as well as reinvented in other places, such as in C++ code inside Microsoft Office) is called the Frameworks Async Pattern. The idea in this pattern is to split the original method into a pair of methods, one to begin (launch) the work, the other to end (join with) the work. In a moment we'll see how this can get elaborate, with intermediate structures and explicit callback registrations, but here's the simple version:

```
// Example 2: Applying the basic "Async Pattern"
//
// The "Begin" part takes the input parameters.
//
IAsyncResult BeginDoSomething(
  InParameters ins
);
// The "End" part yields the return value and
// output parameters. Note: The caller must
// explicitly call EndDoSomething.
//
RetType  EndDoSomething(
  IAsyncResult asyncResult,
  OutParameters outs
);
```

Here's how the earlier calling code would look, using this **BeginXxx/EndXxx** pattern to call **DoSomething** asynchronously:

```
// Example 2, continued:
// Sample calling code
//
void CallerMethod() {
  //

  // Launch, passing the input parameters.
  IAsyncResult ar = BeginDoSomething( this, that );
  result = DoSomething( this, that, outTheOther );

  // These could also take a long time
  // but now run concurrently with DoSomething
  OtherWork();
  MoreOtherWork();

  // Join, then call End to get the return
  // value and output parameters.
  ar.AsyncWaitHandle.WaitOne();
  result = EndDoSomething( ar, outtheOther );

  //  now use result and outOther
}
```

This approach works. However, it has some drawbacks.

First, this pattern is intrusive and bloats the API. The API surface area has now tripled for calls that could be asynchronous, because instead of one function we have three that must be kept in sync. There is also the issue of consistency: The pattern may vary in that it depends on manual discipline for each API author to do it the same way each time. Further, the API designer has to know in advance which methods should support asynchronous calling.

We can eliminate this problem by providing a generalized form of the begin/end pattern that doesn't need to be baked into a specific API. For example, the .NET Framework provides a generalized **BeginInvoke/EndInvoke** that can call arbitrary methods asynchronously. [1] Unfortunately, this does not address the other drawbacks below.

Second, this approach becomes complex for callers, who instead of calling one function must now call three functions: one each to begin, to wait, and to end. It also has more potential points of failure where the calling code can go wrong. For example, if you end up deciding you don't need the result after all because your work is being cancelled or you called the function speculatively or for any other reason, do you have to call the **EndXxx** method? Typically, yes (and on .NET specifically, always yes), because the **BeginXxx** call will allocate resources to launch and track the work, and the **EndXxx** call releases those resources. Hence, the .NET Design Guidelines document says: "Do ensure that the **EndMethodName** method is always called even if the operation is known to have already completed. This allows exceptions to be received and any resources used in the async operation to be freed." This is a common source of errors as even expert programmers, and books written by true gurus, have got this wrong by thinking the **EndXxx** call was optional, or just forgetting to write it on all paths.

Third, this approach actually grows more complex because following this path leads to more complex and ornate styles. In particular, if you look at the actual implementations the begin/end pattern actually grows more parts:

```
// Example 3: The fuller "Async Pattern"
//
// The "Begin" part takes the input parameters
// and optionally a callback to be invoked
// at the end of the work, and a cookie to
// identify this invocation.
//
IAsyncResult BeginDoSomething(
  InParameters ins,
 // + caller can supply a callback notification
  AsyncCallback callback,
  // + extra state to identify "this call"
  Object cookie
);
// The "End" is the same as in Example 2.
RetType EndDoSomething(
  IAsyncResult asyncResult,
  OutParameters outs
);
```

The reason that we grow this requirement is so that callers can provide a kind of "finally do X with the result" handler in cases where the caller wants to return without necessarily needing or wanting to wait for the async call to complete. For example:

```
// Example 3, continued:
// Alternative calling code that doesn't wait
//
void CallerMethod() {
  //

  // Launch, passing the input parameters.
  // But don't join, just eventually use the result.
  IAsyncResult ar = BeginDoSomething( this, that,
    (Object myExtraState)=>{
      result = EndDoSomething( ar, outtheOther );
      // do whatever is necessary with result
      // (write to disk, update a GUI text box, )
    },
    new MyExtraState( /**/ )
  );
  result = DoSomething( this, that, outTheOther );

  OtherWork();
  MoreOtherWork();

  //  now return without waiting for DoSomething
}
```

Fourth, and finally, all variations of this begin/end approach hardwire a specific way to perform the asynchronous call. The caller can choose to run the work asynchronously, but cannot choose how to run it -- whether to run the work on a thread pool thread, on a fresh new thread, as a task on an automatically load-balanced work stealing runtime, on a particular processor core, and so on.

We definitely want to decouple the idea of "run this work asynchronously" ("how" to call) from any given API itself ("what" to call). We partly achieved that much with the generalized form of the begin/end pattern. But we want even more: We want to make the calling code simpler and more robust, and give the caller the flexibility of launching the work in arbitrary ways.

The good news is that we can do better.

Option 2: Decouple "What" From "How"

Abstraction to the rescue:

- Use a separate general-purpose task launcher to launch the work. You probably have a number of options already available in your environment, such as being able to write pool.run(/*task*/) to run a task in a Java or .NET thread pool, or async(/*task*/) in C++0x.
- Use futures to manage the asynchronous results. A "future" is an asynchronous value -- think of it as a "ticket redeemable for a value in the future." [2] This abstraction is

available in Java as Future<T>, in the upcoming C++0x standard as future<T>, and also as Task<T> in the next release of .NET.

For example, here is a simple synchronous call to **CallSomeFunc**:

```
// synchronous call (this will block)
int result = CallSomeFunc(x,y,z);

//  code here doesn't run until call completes and result is ready

// use result which is already available
DoSomethingWith( result );
```

Here is a corresponding asynchronous call (a mix of C++0x and C# syntax):

```
// asynchronous call
future<int> result =
  async( ()=<{ return CallSomeFunc(x,y,z); } );

//  code here runs concurrently with CallSomeFunc

// use result when it's ready (this might block)
DoSomethingWith( result.value() );
```

The **future** allows us to decouple the call (launch) from the receiving of the result (join). This allows us full flexibility in how to launch the work without being invasive in either the API which remains synchronous and unaware, or the task handle which is represented as a simple and robust **future**. Also, we do not need to remember to call an explicit **EndXxx** method as the cleanup is encapsulated in the future abstraction (typically, the future object's destructor or disposer method).

Here's how it looks in the context of our original example. Note that there is no change to the original API:

```
// Example 4: Same original synchronous API.
//
RetType DoSomething(
  InParameters ins,
  OutParameters outs
);
// Sample asynchronous calling code
//
void CallerMethod() {
  //

  // launch work asynchronously (in any
  // fashion; for yuks let's use a thread pool)
  // note that the types of "result" and
  // "outTheOther" are now futures.
  result = pool.run( ()=>{
    DoSomething( this, that, outTheOther ) } );
```

```
  // These could also take a long time
  // but now run concurrently with DoSomething
  OtherWork();
  MoreOtherWork();

  //  now use result.wait() (might block) and outOther
}
```

Note: For convenience only, I'm showing the code using C# lambda syntax. If you don't have C++0x or C# lambdas available in your environment, you can still do this: Just replace "()=>" with a separate Runnable object (Java), delegate (C#), or functor (C++). The lambda is just syntactic sugar for writing a runnable or functor.

What if we don't want to wait for the result but just want some final steps to be performed whenever the call completes, as in Example 3? Easy: Just make it part of the async work, no callback required:

```
// Example 5 (compare with Example 3):
// Alternative calling code that doesn't wait
//
void CallerMethod() {
  //

 // Launch, passing the input parameters.
  // But don't join, just eventually use the result.
  async( ()=>{
    DoSomething( this, that, outTheOther );
    // do whatever is necessary with result
    // (write to disk, update a GUI text box, )
  } );

  OtherWork();
  MoreOtherWork();

  //  now return without waiting for DoSomething
}
```

Summary

How should we supply an async version of an API? Often the best answer is to do nothing at all, because a caller can make any call asynchronous externally using features like "async" or task launching together with futures.

If you are providing a framework or library or any other API interface, prefer to keep asynchronous launching ("how") separate from the task to be done ("what"). This path leads to simpler APIs, simpler and more robust calling code, and great flexibility in where and how to execute the work.

Notes

[1] Calling Synchronous Methods Asynchronously

[2] In the past, people have sometimes used the word "future" mean both the asynchronous work and its result, but this is conflating two separate things. Always think of a future as just an asynchronous value or object

Prefer Structured Lifetimes: Local, Nested, Bounded, Deterministic

What's good for the function and the object is also good for the thread, the task, and the lock

By Herb Sutter Dr.Dobb's Journal
November 11, 2009
URL : http://drdobbs.com/go-parallel/article/221601309

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.

There was a time when it was a novel idea that function calls should obey proper nesting, meaning that the lifetime of a called function should be a proper subset of the lifetime of the function that called it:

```
void f() {
  //
  g();  // jump to function g here and then
  // return from function g and continue here!
  //
}
```

"Eureka!" said Edsger Dijkstra. "Function **g**'s execution occurs entirely within that of function **f**. Boy, that sure seems easier to reason about than jumping in and out of random subroutines with unstructured **goto**s. I wonder what to call this idea. There seems to be inherent structure to it. Hmm, I bet I could build a deterministic and efficient model of 'stack local variables' around it too and maybe I should write a letter" (I paraphrase.) [1]

That novel idea begat the discipline of structured programming. This was a huge boon to programming in general, because structured code was naturally localized and bounded so that parts could be reasoned about in isolation, and entire programs became more understandable, predictable, and deterministic. It was also a huge boon to reusability and a direct enabler of reusable software libraries as we know them today, because structured code made it much easier to treat a call tree (here, **f** and **g** and any other functions they might in turn call) as a distinct unit -- because now the call graph really could be relied upon to be a tree, not the previously usual plate of "**goto** spaghetti" that was difficult to isolate and disentangle from its original environment. The structuredness that let any call tree be designed, debugged, and delivered as a unit has worked so well, and made our code so much easier to write and understand, that we still apply it rigorously today: In every major language, we just expect

that "of course" function calls on the same thread should still logically nest by default, and doing anything else is hardly imaginable.

That's great, but what does it have to do with concurrency?

A Tale of Three Kinds of Lifetimes

In addition to the function lifetimes we've just considered, Table 1 shows three more kinds of lifetimes -- of objects, of threads or tasks, and of locks or other exclusive resource access -- and for each one lists some structured examples, unstructured examples, and the costs of the unstructured mode.

	Object lifetimes	Thread/task lifetimes	Lock lifetimes
Structured Examples	Function local vars: • C++ stack scope • C# using blocks • Java dispose pattern By-value parameters By-value nested objects	Recursive/data decomposition: • Subrange per nested stack call • Partitioned sub-ranges, parallel calls Join-before-return, internal parallelism	Scoped locking: • C++ lock_guard • C# lock blocks • Java synchronized blocks Release-before-return
Unstructured Examples	Global objects Heap/lib allocation	Spawn "new thread()" Spawn process	Explicit "lock" call, with no automatic unlock or "finally unlock" call
Unstructured Costs and Drawbacks	Nondeterministic finalization timing Allocation overhead Tracking overhead to not use after delete: • GC • smart pointers Ownership cycles	Nondeterministic execution/join timing Allocation overhead Tracking overhead to not use task after EOT, join task before EOP Ownership/waiting cycles	Nondeterministic lock acq/rel ordering Allocation overhead Tracking overhead to avoid deadlock, priority inversion, ... Waiting/blocking cycles (deadlock)

Table 1

For familiarity, let's start with object lifetimes (left column). I'll dwell on it a little, because the fundamental issues are the same as in the next two columns even though those more directly involve concurrency.

In the mainstream OO languages, a structured object lifetime begins with the constructor, and ends with the destructor (C++) or **dispose** method (C# and Java) being called before returning from the scope or function in which the object was created. The bounded, nested lifetime means that cleanup of a structured object is deterministic, which is great because there's no reason to hold onto a resource longer than we need it. The object's cleanup is also typically much faster, both in itself and in its performance impact on the rest of the system. [2] In all of the popular mainstream languages, programmers directly use structured function-local object lifetimes where possible for code clarity and performance:

- In some languages, we get to express the structured lifetime using a language feature, such as stack-based or by-value nested member objects in C++, and **using** blocks in C#.
- In other languages, we use a programming idiom or convention, such as the **try/finally** dispose pattern in Java, and explicit dispose-chaining (to have our object's

dispose also call **dispose** on other objects exclusively owned by our object, the equivalent of by-value nested member objects) in both C# and Java.

Unstructured, non-local object lifetimes happen with global objects or dynamically allocated objects, which include objects your program may explicitly allocate on the heap and objects that a library you use may allocate on demand on your behalf. Even basic allocation costs more for unstructured, heap-based objects than for structured, stack-based ones. Objects with unstructured lifetimes also require more bookkeeping -- either by you such as by using smart pointers, or by the system such as with garbage collection and finalization. Importantly, note that C# and Java GC-time finalization [3] is not the same as disposing, and you can only do a restricted set of things in a finalizer. For example, in object **A**'s finalizer it's not generally safe to use any other finalizable object **B**, because **B** might already have been finalized and no longer be in a usable state. Lest we be tempted to sneer at finalizers, however, note also that C++'s shutdown rules for global/static objects, while somewhat more deterministic, are intricate bordering on arcane and require great care to use reliably. So having an unstructured lifetime really does have wide-ranging consequences to the robustness and determinism of your program, particularly when it's time to release resources or shut down the whole system.

Speaking of shutdown: Have you ever noticed that program shutdown is inherently a deeply mysterious time? Getting orderly shutdown right requires great care, and the major root cause is unstructured lifetimes: the need to carefully clean up objects whose lifetimes are not deterministically nested and that might depend on each other. For example, if we have an open SQLConnection object, on the one hand we must be sure to **Close()** or **Dispose()** it before the program exits; but on the other hand, we can't do that while any other part of the program might still need to use it. The system usually does the heavy lifting for us for a few well-known global facilities like console I/O, but we have to worry about this ourselves for everything else.

This isn't to say that unstructured lifetimes shouldn't be used; clearly, they're frequently necessary. But unstructured lifetimes shouldn't be the default, and should be replaced by structured lifetimes wherever possible. Managing nondeterministic object lifetimes can be hard enough in sequential code, and is more complex still in concurrent code.

Enter Concurrency

All of the issues and costs associated with unstructured object lifetimes apply in full force to concurrency. And not only do we see "similar" issues and costs arise in the case of unstructured concurrency, but often we see the very same ones.

Threads and tasks have unstructured lifetimes by default on most systems, and therefore by default non-local, non-nested, unbounded, and nondeterministic. That's the root cause of most of threads' major problems. [4] As with unstructured object lifetimes, to manage unstructured thread and task lifetimes we need to perform tracking to make sure we don't try to wait for or communicate with a task after that task has already ended (similar to use-after-delete or use-after-dispose issues with objects); and to join with a thread before the end of the program to avoid risking undefined behavior on nearly all platforms (similar to unstructured "eventual" finalization at shutdown time). Unstructured threads and tasks also incur extra over-heads [5], such as extra blocking when we attempt to join with multiple pieces of work; they are also more likely to have ownership cycles that have to be detected and cleaned up, and similarly for waiting cycles and even deadlock (e.g., when two threads wait for each other's messages).

Structured thread and task lifetimes are certainly possible. You just have to make it happen by applying a dose of discipline when they aren't structured by default: A function that spawns some asynchronous work has to be careful to also join with that work before the function itself returns, so that the lifetime of the spawned work is a subset of the invoking function's lifetime and there are no outstanding tasks, no pending futures, no lazily deferred side effects that the caller will assume are already complete. Otherwise, chaos can result, as we'll see when we analyze an example of this in the next section.

With locks, the stakes are higher still to keep lifetimes bounded -- the shorter the better -- and deterministic. Deadlocks happen often enough when lock lifetimes all are structured, and being structured isn't enough by itself to avoid deadlock [6]; but toying with unstructured locking is just asking for a generous helping of latent deadlocks to be sprinkled throughout your application. Unstructured lock lifetimes also incur the other issues common to all unstructured lifetimes, including tracking overhead to manage the locks, and performance overhead from excessive waiting/blocking that can cause throttling and delay even when there isn't an outright deadlock.

Given the stakes, it's no surprise that every major OO language offers direct language support for bounded lock lifetimes where the lock is automatically released at the end of the scope or the function:

- C++0x has **std::lock_guard**, which is used in a way that just leverages C++'s bounded stack-based variable rules.
- C# has **lock** blocks, and can also leverage its **using** blocks with lock types that implement **IDisposable**.
- Java has **synchronized** blocks, and can leverage the try/**finally** dispose pattern for disposable lock objects.

These tools exist for a reason. Strongly prefer using these structured lifetime mechanisms for scoped locking whenever possible, instead of following the dark path of unstructured locking by explicit standalone calls to Lock functions without at least a finally to Unlock at the end of the scope.

Having all that in mind, we'll turn to an example that illustrates a simple but common mistake that has its root in unstructuredness.

An Example, With a Common Initial Mistake

Let's take a fresh look at the same Quicksort example we considered briefly last month. [7] Here is a simplified traditional sequential implementation of quicksort:

```
// Example 1: Sequential quicksort
void Quicksort( Iter first, Iter last ) {
  // If the range is small, another sort is usually faster.
  if( distance( first, last ) < limit ) {
    OtherSort( first, last );
    return;
  }
  // 1. Pick a pivot position somewhere in the middle.
  // Move all elements smaller than the pivot value to
  // the pivot's left, and all larger elements to its right.
```

```
  Iter pivot = Partition( first, last );
  // 2. Recurse to sort the subranges.
  Quicksort( first, pivot );
  Quicksort( pivot, last );
}
```

How would you parallelize this function? Here's one attempt, in Example 2 below. It gets several details right: For good performance, it correctly reverts to a synchronous sort for smaller ranges, just like a good synchronous implementation reverts to a non-quicksort for smaller ranges; and it keeps the last chunk of work to run synchronously on its own thread to avoid a needless extra context switch and preserve better locality.

```
// Example 2: Parallelized quicksort, take 1 (flawed)
void Quicksort( Iter first, Iter last ) {
  // If the range is small, synchronous sort is usually faster.
  if( distance( first, last ) < limit ) {
    OtherSort( first, last );
    return;
  }
  // 1. Pick a pivot position somewhere in the middle.
  // Move all elements smaller than the pivot value to
  // the pivot's left, and all larger elements to its right.
  Iter pivot = Partition( first, last );
  // 2. Recurse to sort the subranges. Run the first
  // asynchronously, while this thread continues with
  // the second.
  pool.run(  [=]{ Quicksort( first, pivot ); }  );
  Quicksort( pivot, last );
}
```

Note: **pool.run(x)** just means to create task **x** to run in parallel, such as on a thread pool; and **[=]{ f; }** in C++, or **()=>{ f(); }** in C#, is just a convenient lambda syntax for creating an object that can be executed (in Java, just write a runnable object by hand).

But now back to the main question: What's wrong with this code in Example 2? Think about it for a moment before reading on.

What's Wrong, and How To Make It Right

The code correctly executes the subrange sorts in parallel. Unfortunately, what it doesn't do is encapsulate and localize the concurrency, because this version of the code doesn't guarantee that the subrange sorts will be complete before it returns to the caller. Why not? Because the code failed to join with the spawned work. As illustrated in Figure 1, subtasks can still be running while the caller may already be executing beyond its call to Quicksort. The execution is unstructured; that is, unlike normal structured function calls, the execution of subtasks does not nest properly inside the execution of the parent task that launched them, but is unbounded and nondeterministic.

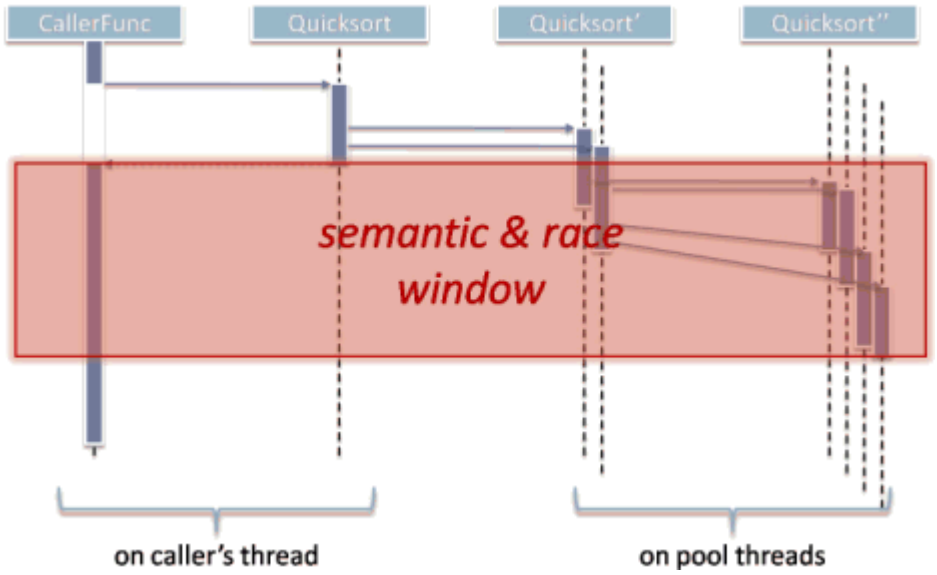

Figure 1

The failure to join actually causes two related but distinct problems:

- **Completion of side effects**: The caller expects that when the function returns the data in the range is sorted.
- **Synchronization**: The caller expects Quicksort won't still be accessing the data in the range, which would race with the caller's subsequent uses of that same data.

Remember, to the caller this appears to be a synchronous method. The caller expects all side effects to be completed, and has no obvious way to wait for them to complete if they're not.

Fortunately, the fix is easy: Just make sure that each invocation of Quicksort not only spawns the subrange sorting in parallel, but also waits for the spawned work to complete. To illustrate the idea, the following corrected example will use a **future** type along the lines of **futures** available in Java and in C++0x, but any kind of task handle that can be waited for will do:

```
// Example 3: Parallelized quicksort, take 2 (fixed)
void Quicksort( Iter first, Iter last ) {
  // If the range is small, synchronous sort is usually faster.
  if( distance( first, last ) < limit ) {
    OtherSort( first, last );
    return;
  }
  // 1. Pick a pivot position somewhere in the middle.
  // Move all elements smaller than the pivot value to
  // the pivot's left, and all larger elements to its right.
  Iter pivot = Partition( first, last );
  // 2. Recurse to sort the subranges. Run the first
  // asynchronously, while this thread continues with
  // the second.
  future fut =  pool.run(  [=]{ Quicksort( first, pivot ); }  );
  Quicksort( pivot, last );
  // 3. Join to ensure we don't return until the work is complete.
  fut.wait();
}
```

As illustrated in Figure 2, just making parent tasks wait for their subtasks makes the execution nicely structured, bounded, and deterministic. We've implemented a barrier where everyone including the caller waits before proceeding, and thus returned the desired amount of synchronicity:

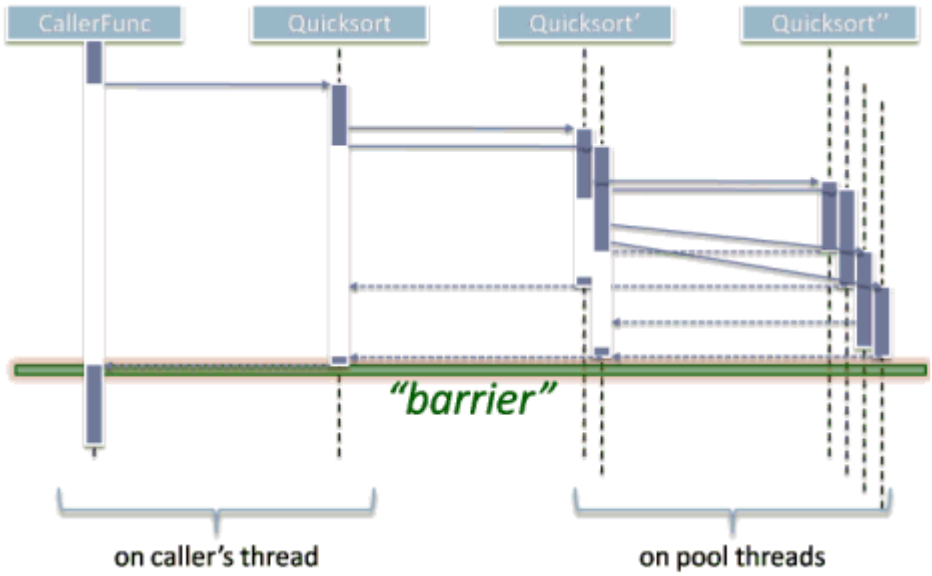

Figure 2

- **Completion of side effects**: We guarantee that when the function returns, it has done its full job and the range is sorted.
- **Synchronization**: After the function returns, some part of its work won't still be accessing the data, and so won't invisibly race with the caller's subsequent uses of the data.

Summary

Table 2 illustrates the differences between structured and unstructured lifetimes in the form of a little artsy photo sequence. In the following descriptions, notice how consistently we use synonyms for our four keywords: local, nested, bounded, and deterministic.

Structured	Unstructured: What You Buy	Unstructured: What You Eat
		

Table 2

- **Structured**: As illustrated on the left side of Table 2, structured lifetimes are like Russian dolls. They nest cleanly, so that each doll fits entirely inside the next larger one. As the program is assembled and executed, each subgroup can be manipulated as a unit. When fully assembled, every individual part is nicely encapsulated and in a well-known place, easy to understand and reason about.
- **Unstructured (initial intent)**: When we first buy into unstructured lifetimes, we think we're buying into the middle picture of raw pasta. We start out with a manageable handful of stiff little independent and parallel lifetimes that don't twist or cross very much. Life is good, at least in the initial PowerPoint design
- **Unstructured (in practice)**: Unfortunately, as the final picture shows, we end up with something quite different. Software under maintenance behaves a lot like pasta: Once it's cooked and combined with other ingredients, the unstructured threads don't stay easy to grasp and easy to separate, but rather get intertwined and easier to break. What we actually wind up with ends up being quite slippery -- in the worst case, "spaghetti code."

Just because some work is done concurrently under the covers, behind a synchronous API call, doesn't mean there isn't any sequential synchronization with the caller. When a synchronous function returns to the caller, even if it did internal work in parallel it must guarantee that as much work as is needed is fully complete, including all side effects and uses of data.

Where possible, prefer structured lifetimes: ones that are local, nested, bounded, and deterministic. This is true no matter what kind of lifetime we're considering, including object lifetimes, thread or task lifetimes, lock lifetimes, or any other kind. Prefer scoped locking, using RAII lock-owning objects (C++, C# via **using**) or scoped language features (C# **lock**, Java **synchronized**). Prefer scoped tasks, wherever possible, particularly for divide-and-conquer and similar strategies where structuredness is natural. Unstructured lifetimes can be perfectly appropriate, of course, but we should be sure we need them because they always incur at least some cost in each of code complexity, code clarity and maintainability, and run-time performance. Where possible, avoid slippery spaghetti code, which becomes all the worse a nightmare to build and maintain when the lifetime issues are amplified by concurrency.

Acknowledgment

Thanks to Scott Meyers for his valuable feedback on drafts of this article.

Notes

[1] E. Dijkstra. "Go To Statement Considered Harmful" (Communications of the ACM, 11(3), March 1968). Available online at search engines everywhere.

[2] Here is one example of why deterministic destruction/dispose matters for performance: In 2003, the microsoft.com website was using .NET code and suffering performance problems. The team's analysis found that too many mid-life objects were leaking into Generation 2 and this caused frequent full garbage collection cycles, which are expensive. In fact, the system was spending 70% of its execution time in garbage collection. The CLR performance team's suggestion? Change the code to **dispose** as many objects as possible before making server-to-

server calls. The result? The system spent approximately 1% of its time in garbage collection after the fix.

[3] C# mistakenly called the finalizer the "destructor," which it is not, and this naming error has caused no end of confusion.

[4] E. Lee. "The Problem with Threads" (University of California at Berkeley, Electrical Engineering and Computer Sciences Technical Report, January 2006). http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf.

[5] The new parallel task runtimes (e.g., Microsoft's Parallel Patterns Library and Task Parallel Library, Intel's Threading Building Blocks) are heavily optimized for structured task lifetimes, and their interfaces encourage structured tasks with implicit joins by default. The resulting performance and determinism benefits are some of the biggest improvements they offer over older abstractions like thread pools and explicit threads. For example, knowing tasks are structured means that all their associated state can be allocated directly on the stack without any heap allocation. That isn't just a nice performance checkmark, but it's an important piece of a key optimization achievement, namely driving down the cost of unrealized concurrency -- the overhead of expressing work as a task (instead of a synchronous function call), in the case when the task is not actually executed on a different thread -- to almost nothing (meaning on the same order as the overhead of an ordinary function call).

[6] The key to avoiding deadlock is to acquire locks in a deterministic and globally recognized order. Although releasing locks in reverse order is not as important for avoiding deadlock, it is usually natural. The key thing for keeping lock lifetimes structured is to release any locks the function took at least by the end of the function, which makes code self-contained and easier to reason about. Even when using a technique like hand-over-hand locking where lock release is not in reverse order of lock acquisition, locking should still be scoped so that all locks a function took are released by the time the function returns. See also H. Sutter. "Use Lock Hierarchies to Avoid Deadlock" (Dr. Dobb's Journal, 33(1), January 2008). Available online at http://www.ddj.com/hpc-high-performance-computing/204801163.

[7] H. Sutter. "Effective Concurrency: Avoid Exposing Concurrency -- Hide It Inside Synchronous Methods" (Dr. Dobb's Digest, October 2009). Available online at http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=220600388 .

Avoid Exposing Concurrency: Hide It Inside Synchronous Methods

You have a mass of existing code and want to add concurrency. Where do you start?

By Herb Sutter Dr.Dobb's Journal
October 12, 2009
URL : http://drdobbs.com/go-parallel/article/220600388

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.

Let's say you need to migrate existing code to take advantage of concurrent execution or scale on parallel hardware. In that case, you'll probably find yourself in one of these two common situations, which are actually more similar than different:

- Migrating an application: You're an application developer, and you want to migrate your existing synchronous application to be able to benefit from concurrency.
- Migrating a library or framework: You're a developer on a team that produces a library or framework used by other teams or external customers, and you want to let the library take advantage of concurrency on behalf of the application without requiring application code rewrites.

You have a mountain of opportunities and obstacles before you. Where do you start?

This column is about the most basic strategy to have in our toolchest. It may seem so simple that it goes without saying, but there are a few important details to keep in mind to do it correctly.

Adding Concurrency: Target It, Localize It, Hide It

Perhaps the most effective way to deliver concurrency is to hide it inside "synchronous" APIs. The key is to inject concurrency in a targeted, localized way inside key functions or methods, and let the application code that calls those methods stay happily sequential and blissfully unaware of the concurrency. We want keep from exposing concurrency throughout the entire application, because the existing code isn't designed with concurrency and nondeterminism in mind. The less code has to be upgraded to know about concurrency, the better.

First, identify the places where we can benefit from concurrency. Here are two major kinds of situations to look for:

- High-latency operations that can benefit from concurrent waiting. Here we want to look for places where our lower-level or library code is doing synchronous work when the caller requests it. If that work could be launched actively sooner (even if sometimes speculatively), then the waiting can overlap with the caller's execution, so that when the caller finally does ask for the result some or all of the work or waiting is already complete.
- Compute-intensive operations that can benefit from parallel execution. Here, we want to look for "hot spot" functions or methods where your application or library code is spending significant parts of computational effort. Focus on the ones that are amenable to parallelization because we can readily break the work into chunks that can be performed in parallel. Typically, we're looking for natural parallelism that we can exploit in the algorithm (e.g., recursive divide-and-conquer where nonoverlapping subranges can be processed in parallel) or in the data structures (e.g., nonoverlapping subtrees or subgraphs or hash table buckets can be processed in parallel). These are our initial targets.

Of course, there's overlap between these two categories, as we'll see in the following examples. Second, once we've found that low-hanging fruit, add concurrency in those

functions internally. Note that word "internally" is essential -- we don't want to expose the concurrency in the API we present to the caller. Ideally, all concurrency is bounded within the scope of the function call, so that the caller calls our function and we launch the concurrency and join with it before returning, and the function just appears to execute faster. Next best, concurrency is not bounded but is managed automatically, so that the caller calls our function and we launch the concurrency and return with something still running under the covers, and the caller doesn't need to know because in some subsequent call into our code (or upon a timeout, or some other automated way) we join with the concurrent work.

With the concurrency thus compartmentalized, the calling code doesn't have to know about the parallel work. As far as the caller is concerned, our library method just happens to execute better or faster than before.

Example: An Internally Parallelized Algorithm

The first example is simple and obvious, but shows an important way to deliver scalable parallelism in a tactical way for an existing application. Consider the following sequential implementation of quicksort, simplified to its essentials:

```
// Example 1(a): Sequential quicksort
void Quicksort( Iter first, Iter last ) {
  // If the range is small, another sort is faster.
  if( distance( first, last ) < limit ) {
   OtherSort( first, last );
   return;
  }
  // 1. Pick a pivot and partition the data.
  Iter pivot = Partition( first, last );
  // 2. Recurse to sort the subranges.
  Quicksort( first, pivot );
  Quicksort( pivot, last );
}
```

This is a traditional divide-and-conquer recursive algorithm, performed sequentially. If we're trying to parallelize our application, this is just the sort (sorry) of function we're looking for: A compute-intensive function that contains natural parallelism, in this case in the algorithm. The subrange sort operations work on nonoverlapping subsets of the data, and their processing is probably fully independent as long as sorting the left subrange doesn't cause other side effects that could affect sorting the right subrange, and vice versa.

This turns out to be a perfectly simple case to illustrate the point of this article, because the natural way to parallelize this code inherently hides the parallelism from the caller (assuming we get the mechanical details right; more about avoiding mistakes and pitfalls in another article):

```
// Example 1(b): Parallelized quicksort
void Quicksort( Iter first, Iter last ) {
  // If the range is small, synchronous sort is faster.
  if( distance( first, last ) < limit ) {
   OtherSort( first, last );
   return;
```

```
  }
  // 1. Pick a pivot and partition the data synchronously.
  Iter pivot = Partition( first, last );
  // 2. Recurse to sort the subranges in parallel.
  future fut = pool.run(  [=]{ Quicksort( first, pivot ); }  );
  Quicksort( pivot, last );
  fut.join();
}
```

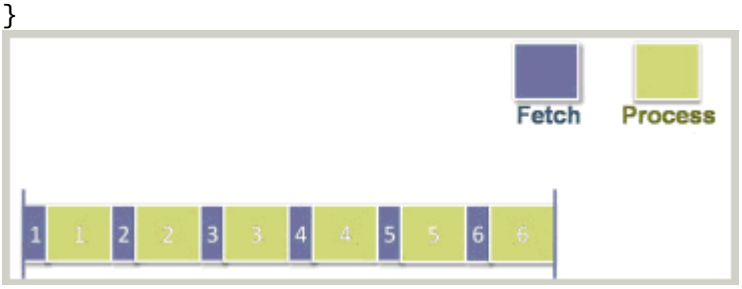

Figure 1: Synchronous fetch-process

Note that the code may look slightly different in your environment, but we're simply saying to execute the two subrange sorts in parallel:

- The syntax **pool.run(x)** denotes creating a task **x** to run in parallel, which is common when using thread pools for example.
- The syntax **[=]{ f(); }** is C++0x lambda function syntax, a convenient syntactic sugar for writing a function object to be executed elsewhere. If you're using C#, the syntax looks like **()=>{ f(); }**. If you're using Java, just create a runnable object by hand.

  This function may spawn a lot of parallel work recursively, but everything is joined before we return to the caller. The function just happens to run faster when run on machine that has more cores. So sequential code that calls this library function also gets to reap the benefits of leveraging whatever hardware parallelism there happens to be on a given end user's machine, at least for this function, without having to know about the parallel work being done on its behalf.

Example: Optimistic Iteration

Now consider the following loop that uses an enumerator/iterator style to visit each item in a collection:

```
// Example 2, common calling code
void f( WidgetCollection wc ) {
  using( i = wc.GetEnumerator() ) {
   while( i.MoveNext() ) {       // fetch next element
      //  some work                        // and process it
      DoSomethingWith( i.Current );
      //  more work
    }
  }
}
```

Imagine the enumerator is implemented something like this:

```
// Example 2(a), synchronous enumerator (sketch)
class WidgetEnumerator {
  //
  // Construct
 WidgetEnumerator() {
    // set up any necessary resources
  }
  // Destroy (C++ destructor, Java/C# disposer)
  void Dispose() {
    // release any resources we got
  }
  bool MoveNext() {
   if( AtEnd() ) {
      return false;
    }
    // note: this may be an expensive operation
   NavigateToNextItem();
   return true;
  }
  Widget Current() {
    lock( myWidgetQueue ) {
        return myWidgetQueue.First;
     }
  }
```

Each time the caller asks for the next element, we go look for it, then return and let the caller process it. So in this sequential version of the code, the fetching and processing work is interleaved, as in Figure 1.

What if **NavigateToNextItem** (and therefore **MoveNext**) is an expensive operation? What if it has to perform an expensive computation, or take another round trip to a disk or database or web service? Then it's quite likely that we can do significantly better even for this simple common iteration pattern by applying concurrency.

Everyone who has implemented or used an enumeration API like this knows that, if the caller is asking for one item, he's likely to soon come back, hat in hand, and ask for the next one too, and then the next one after that. If getting the next element takes nontrivial time for any of a variety of reasons, and we have resources to spare, we may want to be more proactive about generating the results so that we can deliver them faster once the caller actually requests them. And, if we guess wrong, we want a way to throw away the results we didn't need after all. Even if the cost to find and prepare the next item is less than the time for the caller to complete a loop iteration and ask for the next item, as in Figure 1, we can gain a useful speedup by running the enumerator's fetching concurrently with the caller's processing.

So consider performing the iteration concurrently (and speculatively, because if the caller doesn't end up traversing the whole range we may end up throwing some work away):

```
// Example 2(b), concurrent enumerator (sketch)
class WidgetEnumerator {
  //
  // Construct
 WidgetEnumerator() {
```

```
  // set up any necessary resources
  // and launch a traversal worker thread
 done = false;
 pleaseStop = false;
 myWidgetQueue.Append( /* dummy */ );
 myWorker = new thread( () => {
    while( !pleaseStop && !AtEnd() ) {
      // note: this may be an expensive operation
      NavigateToNextItem();
      lock( myWidgetQueue ) {
        myWidgetQueue.Append( /* current item */ );
      }
      newItem.Notify();            // notify consumer
    }
    lock( myWidgetQueue ) {
      myWidgetQueue.Append( /* sentinel */ );
    }
    newItem.Notify();            // notify one last time
  } );
}
// Destroy (C++ destructor, Java/C# disposer)
void Dispose() {
  // stop and join the background worker
 pleaseStop = true;
 myWorker.join();
  // release any resources we got
}
bool MoveNext() {
 if( !done ) {
    lock( myWidgetQueue ) {
     myWidgetQueue.PopFirst();
    }
    newItem.Wait();
  }
 lock( myWidgetQueue ) {
    if( myWidgetQueue.First == /* sentinel */ ) {
      done = true;
      return false;
    }
    return true;
  }
}
 Widget Current() {
 lock( myWidgetQueue ) {
    return myWidgetQueue.First;
  }
}
```

Now we can perform the fetching and processing concurrently, as in Figure 2. We can further improve this code by doing things like bounding the size of the queue, so that in general fewer resources are in use at once and in particular less work is wasted if the caller should happen to throw away the enumerator before reaching the end, but this is enough to sketch the basic strategy.

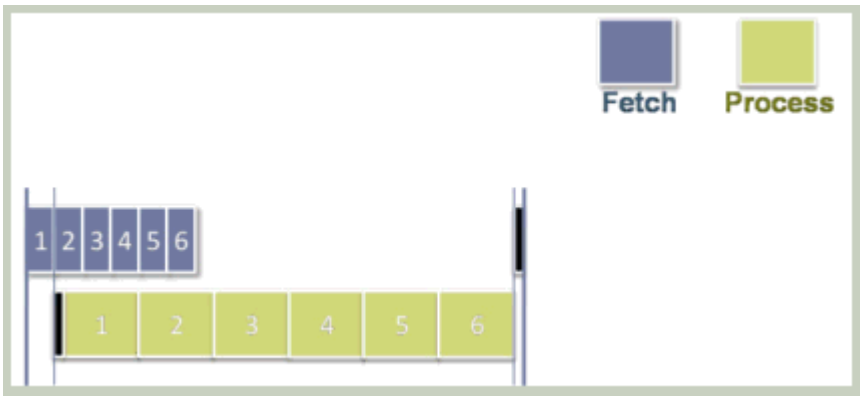

Figure 2: Background fetching

You may have noticed that what we've really done is create a two-stage pipeline, and we'll look at the general strategy of pipelining in more depth in a future article. For now, the key point to notice here is that the caller's code has no idea that it's participating as a pipeline stage, but remains happily oblivious to the concurrency because we've once again managed to hide it under the covers behind a synchronous-looking API.

Summary

The less code has to be upgraded to know about concurrency and nondeterminism, the better. Prefer to keep knowledge of concurrent work compartmentalized: Where possible, hide it behind a "synchronous" API to limit its effect and let callers remain insulated from the concurrency. This is an excellent strategy to reach for first when migrating an existing code base to take advantage of concurrent execution or scale on parallel hardware.

Acknowledgment

Thanks to Terry Crowley for his insightful comments on drafts of this article.

The Power of "In Progress"

Hold no hostages: Know when to design for "partially updated" as the normal state

By Herb Sutter Dr.Dobb's Journal
July 10, 2009
URL : http://drdobbs.com/go-parallel/article/218401447

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.

Don't let a long-running operation take hostages. When some work that takes a long time to complete holds exclusive access to one or more popular shared resources, such as a thread or a mutex that controls access to some popular shared data, it can block other work that needs

the same resource. Forcing that other work to wait its turn hurts both throughput and responsiveness.

To solve this problem, a key general strategy is to "design out" such long-running operations bysplitting them up in to shorter operations that can be run a piece at a time. Last month in Break Up and Interleave Work to Keep Threads Responsive, we considered the special case when the hostage is a thread, and we want to prevent one long operation from commandeering the thread (and its associated data) for a long time all at once. Two techniques that accomplish the strategy of splitting up the work are continuations and reentrancy; both let us perform the long work a chunk at a time, and in between those pieces our thread can interleave other waiting work to stay responsive.

Unfortunately, there's a downside: We also saw that both techniques require some extra care because the interleaved work can have side effects on the state the long operation is using, and we needed to make sure any interleaved work wouldn't cause trouble for the next chunk of the longer operation already in progress. That can be hard to remember, and sometimes downright complicated and messy. Is there another, more general way?

Let It "Partly" Be: Embrace Incremental Change

Let's look at the question from another angle, suggested by my colleague Terry Crowley: Instead of viewing partially-updated state with in-progress work as an 'unfortunate' special case to remember and recover from, what if we simply embrace it as the normal case?

The idea is to treat the state of the shared data as stable, long-lived and valid even while there is some work pending. That is, the "work pending" that results from splitting long operations into smaller pieces isn't tacked on later via a black-box continuation object or encoded in a stack frame on a reentrant call; rather, it's promoted to a full-fledged, first-class, designed-in-up-front part of the data structure itself.

Looking at the problem this way has several benefits. Perhaps the most important one is correctness: We're making it clear that each "chunk" of processing is starting fresh and not relying on system state being unchanged since a previous call. In [1], we had to remember not to make that implicit assumption when we resumed a continuation or returned from a yield; this way, were are explicit about the "data plus work pending" state as the normal and expected state of the system.

This approach also enables several performance and concurrency benefits, including that we have a range of options for when and how to do the pending work. We'll look at those in more detail once we've seen a few examples, but for now note that they include that we can choose to do the pending work:

- with finer granularity, so that we hold exclusion (e.g., locks) for shorter times as we do smaller chunks of the work;
- asynchronously, so that it can be more easily performed by another helper thread; and/or
- lazily, instead of all up front.

Note the "and/or" -- one or more of these may apply in any given situation. Some of these techniques, notably enabling lazy evaluation, are well-known optimizations for ordinary performance, but they also directly help with concurrency and responsiveness.

The rest of this article will consider two mostly orthogonal issues: First, we'll consider different ways to represent the pending work in the data structure, with real-world examples. Then, we'll separately consider what major options we have for actually executing the work, how to use them singly or in combination, and what their tradeoffs are and where each one applies.

Option 1: Data + Pending Changes

Perhaps the simplest option is to represent the data along with a (possibly empty) queue of pending updates or other work to be performed against it, as illustrated in Figure 1. This configuration is the most similar to the continuation style in [1]. However, unlike that style where we explicitly created a continuation object and appended it to a queue that was logically separate from the data, here the pending work queue is an intentional first-class part of the data structure integrated into its design and operation.

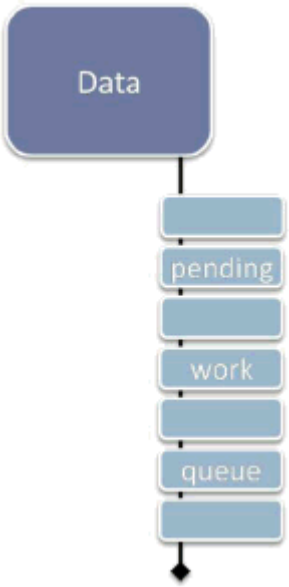

Figure 1: Deferring work via "data + pending changes."

One potential drawback to Option 1 is that it can be less flexible if we may need to interrupt and restart an operation we've split into pieces and enqueued. For example, if recalculations are affected by further user input, and multiple chunks of the recalculation work are already in the queue, we may need to traverse the queue to remove specific work items. One way to minimize this concern is to only enqueue one "next step" at a time for each long-running operation, and have the end of each chunk of work enqueue its own continuation (see sample code in [1]).

Option 2: Data + Cookies

Figure 2a shows a different way to encode pending work: Attach reminders (or "cookies") that go with the data, each of which remembers the current state of a given operation in progress.

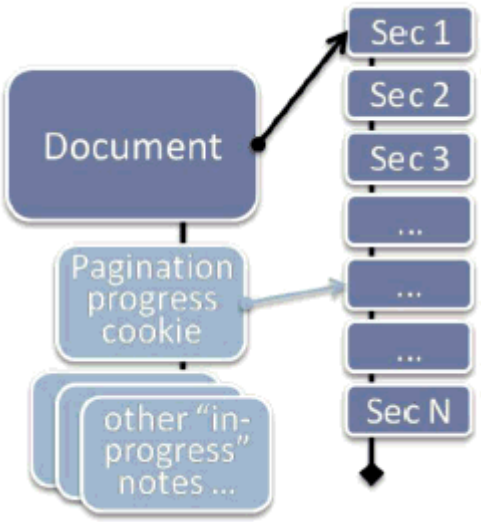

Figure 2a: Deferring work via "data + cookies."

Consider the example of a word processing application, as suggested in Figure 2a: Pagination, or formatting the content of a document as a series of pages for display, is one of dozens of operations that can take far longer than would be acceptable to perform synchronously; there's no way we want to make the user wait that long while typing. In Microsoft Word, the internal data structure that represents a document makes it well-defined to have a document that is only partially paginated, specifically so Word can break the pagination work up into pieces. By executing the operation a piece at a time, it can stay responsive to any pending user messages and provide intermediate feedback before continuing on to the next chunk.

At any given time, we only need to immediately and synchronously perform pagination up to the currently displayed page, and then only if the program is in a mode that displays page breaks. Later pages in the document can be paginated lazily during idle time, on a background thread, or using some other strategy (see "When and How To Do the Work" later in this article). However, if the user jumps ahead to a later page, pagination must be completed to at least that page, and any that is not yet performed must be done immediately.

This approach can be more flexible than Option 1 in the case when we may need to interrupt and restart the operation, such as if pagination is impacted by some further user input such as inserting a paragraph. Instead of walking a work queue, we update the (single) state of pagination for this document; in this case, it's probably sufficient to just resume pagination from a specific earlier point where the new paragraph was inserted.

Figure 2b shows another example of how this technique can be used in a typical modern word processor, in this case OpenOffice Writer.

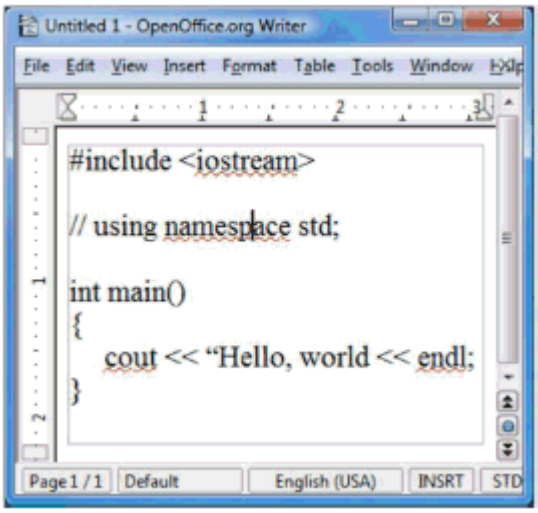

Figure 2b: Red squigglies in a word processor -- without "spell check document" (courtesy OpenOffice.org Writer).

Like many word processors, Writer offers a synchronous "spell check my document now" function that pops up a dialog and lets you step through all the potential spelling mistakes in your document. But it also offers something even more helpful: "red squigglies," or immediate visual feedback as you type to highlight words that may be misspelled. In Figure 2b, the document contains what looks more like C++ code than English words, and so a number of words get the red-squiggly I-can't-find-that-in-my-dictionary treatment. (Aside: Yes, the typo is intentional. See Figure 3b.)

Consider: Would you add red squigglies synchronously as the user types, or would you do it asynchronously? Well, if all the user has done is type a new word, it may be okay to do it synchronously for that word because each dictionary lookup is probably fast. But what about when the user pastes several paragraphs at once? Or loads a large document for the first time? With a lot of work left to do, we may well want to do the spell checking in the background and let the red squigglies appear asynchronously. We can optimize this technique further in a number of ways, such as giving priority to checking first the visible part of the document, but in general we can get better overall responsiveness by doing all spell checking in the background by default to build up a list of spelling errors in the document, so that the information is already available and the user doesn't have to wait as he navigates around the document or enters the dedicated spell-check mode.

Option 3: Data + Lazy Evaluation

Lazy evaluation is a traditional optimization technique that happens to also help concurrency and responsiveness. The basic idea is simple: We want to optimize performance by not doing work until that work is actually needed. Traditional applications include demand-loading persistent objects from a slow database store and creating expensive Singleton objects on demand. In Excel, for example, it is well-defined to have worksheets with pending recalculations to perform, and each cell may be "completely evaluated" or "pending evaluation" at any given time, so that we can immediately evaluate those that are visible on-screen and lazily evaluate those that are off-screen or hidden.

Figure 3a illustrates how we can use lazy evaluation as a natural way to encode pending work. A compiler typically stores the abstract representation of your program code as a tree.

To fully compile your code to generate a standalone executable, clearly the compiler has to process the entire tree so that it can generate all the required code. But do we always need to process the whole tree immediately?

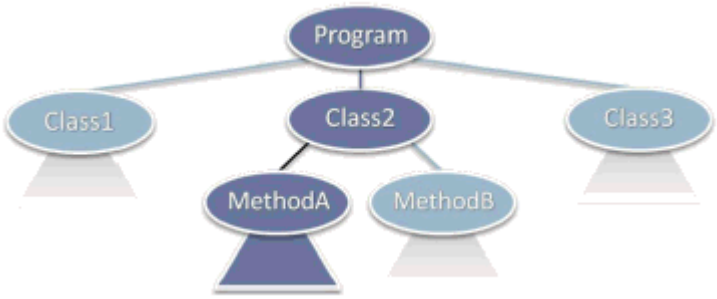

Figure 3a: Deferring work via traditional lazy evaluation.

One common example where we do not is compilation in managed environments like Java and .NET, where it's typical to use a just-in-time (JIT) compiler that compiles classes and methods on demand as they get invoked. In Figure 3a, for example, we may have called and compiled **Class2::MethodA()**, but if we haven't yet called **Class2::MethodB** or anything in **Class1** or **Class3**, those entities can be compiled later on demand (or asynchronously in the background during idle time or some other strategy; again, see "When and How To Do the Work").

But lazy compilation is useful for much more than just JIT. Now consider Figure 3b: Let's say we want to dynamically provide red-squiggly feedback, not on English misspellings, but rather on code warnings and errors. Just as we wanted dynamic spell-checking feedback without going into a special spell-check-everything-now mode (see Figure 2b), we might want dynamic compilation warnings and errors without going into a separate compile-everything-now mode.

In Figure 3b, the IDE is helpfully informing us that the code has several problems, and even provides the helpful message "missing closing quote" in a tooltip balloon as the mouse hovers over the second error -- all dynamically as we write the code, before we try to explicitly recompile anything. Clearly, we have to compile something in order to diagnose those errors, but we don't have to compile the whole program. As with the word processing squigglies, we can prioritize the visible part of the code, and lazily compile just enough of the context to make sense of the code the user sees; in this example, we can avoiding compiling pretty much the entire **<:iostream>** header because nothing in it is relevant to diagnosing these particular errors.

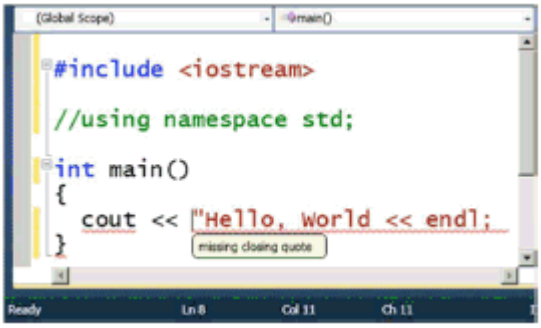

Figure 3b: Red squigglies in a C++ IDE -- without "rebuild" (courtesy Visual Studio 2010).

When and How To Do the Work

This brings us to the key question: So, when and how is the pending work performed?

One option is to do the pending work interleaved with other work, such as in response to timer messages or using explicit continuations as in [1]. This approach is especially useful when the updates must be performed by a single thread, either for historical reasons (e.g., on systems that require a single GUI thread) or to avoid complex locking and synchronization of internal data structures by making the data isolated to a particular thread.

A second approach is to do the work when idle and there is no other work to do. For example, Word normally performs pagination and similar operations in incremental chunks at idle time. This approach is usually used in combination with a fallback to one of the other approaches if we discover that some part of the work needs to happen more immediately, for example if the user jumps to not-yet-paginated part of the document. There are multiple ways to implement "when idle":

- If all of the updates must be performed by a single thread, we can register callbacks that the system will invoke when idle (e.g., using a Windows WM_IDLE message); this is a traditional approach for GUI applications that have legacy requirements to do their work on the GUI thread.
- If the updates can be performed by a different thread, we can perform the work on one or more low-priority background threads, each of which pauses every time it completes a chunk of work. To minimize the potential for priority inversion, we want to avoid being in the middle of processing an item (and holding a lock on the shared data) when the background thread is preempted by the operating system, so each chunk of work should fit into an OS scheduling quantum and the pause between work items should include yielding the remainder of the quantum (e.g., using **Sleep(0)** on Windows).

Third, we can do the work asynchronously and concurrently with other work, such as on one or more normal background worker threads, each of which locks the structure long enough to perform a single piece of pending work and then pauses to let other threads make progress. For example, in Excel 2007 and later, cell recalculation uses a lock-free algorithm that executes in parallel in the background while the user is interacting with the spreadsheet; it may run on several worker threads whose number is scaled to match the amount of hardware parallelism available on the user's machine.

Fourth, in some cases it can be appropriate to do the work lazily on use, where each use of the data structure also performs some pending work to contribute to overall progress; or similarly we may do it on demand specifically in the case of traditional lazy evaluation. With these approaches, note that if the data structure is unused for a time then no progress will be made; that might be desirable, or it might not. Also, if the accesses can come from different threads, it must be safe and appropriate to run different pieces of pending work on whatever threads happen to access the data.

Summary

Embrace change: For high-contention data that may be the target of long-running operations, consider designing for "partially updated" as a normal case by making pending work a first-

class part of the shared data structure. Doing so enables greater concurrency and better responsiveness. It lets us shorten the length of time we need to hold exclusion on a given piece of shared data at any time, while still allowing for operations that take a long time to complete -- but can now run to completion without taking the data hostage the whole time.

We can express the pending work in a number of ways, including as a queue of work, as cookies representing the state of operations still in progress, or using lazy evaluation for its concurrency and responsiveness benefits as well as for its traditional optimization value. Then we can execute the work using one or more strategies that make sense; common ones include executing it interleaved with other work, during idle processing, asynchronously on one or more other threads, on use, or on demand.

It's true that we'll typically incur extra overhead to store and "rediscover" how to resume the longer operation at the appropriate point, but the benefits to overall system maintainability and understandability will often far outweigh the cost. Especially when the interleaved work may need to be canceled or restarted in response to other actions, as in the pagination and recalculation examples, it's easier to write the code correctly when the work still in progress is a well-defined part of the overall state of the system.

Acknowledgments

Thanks to Terry Crowley, director of development for Microsoft Office, for providing the motivation to write about this topic and several of the points and examples. Other examples were drawn from the development of Visual C++ 2010.

Notes

[1] H. Sutter. "Break Up and Interleave Work to Keep Threads Responsive" (Dr. Dobb's Digest, June 2009). Available online at http://www.ddj.com/architect/217801299.

Break Up and Interleave Work to Keep Threads Responsive

Breaking up is hard to do, but interleaving can be even subtler

By Herb Sutter Dr.Dobb's Journal
June 15, 2009
URL : http://drdobbs.com/go-parallel/article/217801299

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.

In a recent article, we covered reasons why threads should strive to make their data private and communicate using asynchronous messages.[1] Each thread that needs to receive input should have an inbound message queue and organize its work around an event-driven message pump mainline that services the queue requests, idealized as follows:

```
// An idealized thread mainline
//
do {
    message = queue.pop() // get the message
                          // (might wait)
    message->run();    // and execute it
} while( !done );         // check for exit
```

But what happens when this thread must remain responsive to new incoming messages that have to be handled quickly, even when we're in the middle of servicing an earlier lower-priority message that may take a long time to process?

If all the messages must be handled on this same thread, then we have a problem. Fortunately, we also have two good solutions, both of which follow the same basic strategy: somehow break apart the large piece of work to allow the thread to perform other work in between, interleaved between the chunks of the large item. Let's consider the two major ways to implement that interleaving, and their respective tradeoffs in the areas of fairness and performance.

Example: Breaking Up a Potentially Long-Running Operation

Consider this potentially expensive message we might be asked to execute:

```
// A simplified message type to accomplish some
// long operation
//
class LongOperation : public Message {
public:
   void run() {
      LongHelper helper = GetHelper();
// issue: what if this loop could take a long time?
   for( int i = 0; i < items.size(); ++i ) {
      helper->render( items[i] );
   }
   helper->print();
 }
}
```

This thread may be a background worker that runs all the work we want to get off the GUI thread (see [1]). Alternatively, in cases where it is impossible to obey the good hygiene of

getting all significant work off the GUI thread (for example, because for some reason the work may need to happen on the GUI thread itself for legacy or OS-specific reasons), this thread may be the GUI itself. Whatever the case, what matters is that to remain responsive to other messages we need to break up **LongOperation.run** into smaller pieces and interleave them with the processing of other messages.

Option 1:Use Continuations

The first way to tackle the problem is to use continuations. A continuation is just a way to talk about "the rest of the work that's left to do." It includes capturing the state of any local or intermediate variables that we're in the middle of using, so that when we resume the computation we can correctly pick up again where we left off.

Example 1(a) shows one way to implement a continuation style:

```
// Example 1(a): Using a continuation style
//
class LongOperation : public Message {
 int start;
   LongHelper helper;
public:
 LongOperation( int start_ = 0,
      LongHelper helper_ = nullptr )
        : start(start_), helper(helper_) { }
   void run() {
 if( helper == nullptr
             // if first time through, get helper
        helper = GetHelper();
      int i = 0;
 // do just another chunk's worth
      for( ; i < ChunkSize && start+i < items.size();
           ++i ) {
        helper->render( items[ start+i ] );
   }
 if( start+i < items.size() )
        // if not done, launch a continuation
     queue.push(LongOperation(start+i, helper));
   else
           // if last time through, finish up
      helper->print();
   }
}
```

The first **LongOperation**object executes only a suitably-sized chunk of its loop. To ensure that the remainder of the work gets done, it creates a new **LongOperation** object (the continuation) that stores the current intermediate state, including the helper local variable and the loop index, and pushes the continuation on the message queue. (For simplicity this code assumes **LongOperation** has direct access to queue; in practice you would provide an indirection such as a callback to decouple the message type from a specific queue.)

A good way to think about this is that we're "saving our stack frame" off in a separate object on the heap. The overhead we're incurring for each continuation is the cost of copying the local variables, one allocation (and subsequent destruction) for the continuation message, and

one extra pair of enqueue/dequeue operations. This approach has the major advantage of fairness. It's fair to the waiting messages, because each continuation gets pushed onto the end of the queue and so all messages currently waiting will be processed before we do another chunk of the longer work. More importantly, it's fair to **LongOperation** itself, because any other new messages that arrive after the continuation is enqueued, during the time while we're draining the queue, will stay in line behind the continuation.

This fairness can also be a disadvantage, however, if we'd actually like to execute most of the messages in order no matter how long they take, and only enable interleaved "early" execution for certain high-priority messages. Some of that can be accomplished using a priority queue as the message queue, but in general this kind of flexibility will be easier to accomplish under Option 2, where we opt for a different set of tradeoffs.

Beware State Changes When Interleaving

Note that there is a subtle but important semantic issue that we have to be aware of that wasn't possible in the noninterleaved version of the code.

The issue is that whenever the code cedes control to allow other messages to be executed, it must be aware that the thread's state can be changed by that other code that executed on this thread in the meantime. When the code resumes, it cannot in general assume that any data that is private to the thread, including thread-local variables, has remained unchanged across the interruption.

In Example 1(a), consider: What happens if another message changes the size or contents of the **items** collection? If our operation is trying to process a consistent snapshot of the collection's state, we may need to save even more off to the side by taking a snapshot of the collection and doing all of our work against the snapshot, so that we maintain the semantics of doing our operation against the collection in the state it had when we started:

```
// Example 1(b): Using a continuation style,
// and adding resilience to collection changes
//
class LongOperation : public Message {
 Collection myItems;
   int start;
   LongHelper helper;
public:
   LongOperation(
     int start_ = 0,
     LongHelper helper_ = nullptr,
 Collection myItems = nullptr
   )
   : start(start_),
     helper(helper_),
     myItems(myItems_)
{ }
void run() {
   if( helper == nullptr ) {// if first time through
      helper = GetHelper(); // get helper
 myItems = items.copy();// and take a snapshot of items
   }
   int i = 0; // do just another chunk's worth
```

```
   for( ; i < ChunkSize && start+i < myItems.size(); ++i ) {
      helper->render( myItems[ start+i ] );
   }
   if( start+i < myItems.size() ) // if not done,
                                  // launch a continuation
      queue.push( LongOperation( start+i, helper, myItems ) );
   else { // if last time through, finish up
      helper->print();
Free( myItems ); // and clean up myItems
   }
 }
}
```

Now the continuation is resilient to the case where other messages may change items, by doing all of its processing using the state the collection had when our operation began.

Note that we have still introduced one other semantic change, in that we're deliberately allowing later messages to run against newer state before this earlier operation on the older state is complete. That's often just fine and dandy, but it's important to be aware that we're buying into those semantics.

All of these considerations apply even more to Option 2. Let's now turn to our second strategy for interleaving work, and see how it offers an alternative set of tradeoffs.

Option 2:Use Reentrant Message Pumping

Option 2 could be subtitled: "Cooperative multitasking to the rescue!" It will be familiar to anyone who's worked on systems based on cooperative multitasking, such as early versions of MacOS and Windows.

Example 2 illustrates the simplest implementation of Option 2, where instead of "saving stack frames on the heap" in the form of continuations, we instead just keep our in-progress state right on the stack where it already is and use nesting (aka reentrancy) to pump additional messages.

```
// Example 2(a): Explicit yield-and-reenter (possibly flawed)
//
class LongOperation : public Message {
public:
   void run() {
      LongHelper helper = GetHelper();
      for( int i = 0; i < items.size(); ++i ) {
 if( i % ChunkSize == 0 ) // from time to time
             Yield();             // explicitly yield control
         helper-> render( items[i] );
      }
      helper->print();
   }
}
```

In a moment we'll consider several options for implementing **Yield**. For now, the key is just that the **Yield** call contains a message pump loop that will process waiting messages, which is what gives them the chance to get unblocked and interleave with this loop.

Option 2 avoids the performance and memory overhead of separate allocation and queueing we saw in Option 1, but it leads to bigger stacks. Stack overflow shouldn't be a problem, however, unless stack frames are large and nesting is pathologically frequent (in which case, there are bigger problems; see next paragraph).

Probably the biggest advantage of this approach is that the code is simpler. We just have to call **Yield** every so often to allow other messages to be processed, and we're golden ... or so we think, because unfortunately it's not actually quite that easy. The code is simpler to write, but requires a little more care to write correctly. Why?

Remember: Beware State Changes When Interleaving, Really

Just as we saw with continuations, so with any other interleaving including **Yield**: Whenever the code **Yields** it must be aware that the thread's state can be changed by the code that executed on this thread during **Yield** operation. It cannot in general assume that any data that is private to the thread, including thread-local variables, remains unchanged across the **Yield** call. With continuations, the issue was a bit easier to remember because that style already requires the programmer to explicitly save the local state off to the side and then return, so that by the time we get to the code where the continuation resumes it's easy to remember that time has passed and the world might have changed.

When using **Yield**-reentrancy instead of continuations, it's easier to forget about this effect, in part because it really is (too) easy to just throw a **Yield** in anywhere. To see how this can cause problems, assume for a moment that semantically we don't care if nested messages change the contents of **items** (which was the case in the discussion around Example 1(b)). That is, assume we don't necessarily need to process a snapshot of the state, but only get through items until we're done. Even with those relaxed requirements, do you notice a subtle bug in Example 2?

Consider: What happens if a nested message changes the size of the **items** collection? If that's possible, then the collection contains at least **i** objects before the **Yield** and the expression **items[i]** is valid, but after the **Yield** the collection may no longer contain **i** objects and so **items[i]** could fail.

In this simple example, we can apply a simple fix. The issue is that we have a window between observing that **i < items.size()** and using **items[i]**, so the simplest fix is to eliminate the problematic window by not yielding in between those points:

```
// Example 2(b): Explicit yield-and-reenter (immediate problem fixed)
//
class LongOperation : public Message {
public:
   void run() {
      LongHelper helper = GetHelper();
      for( int i = 0; i < items.size(); ++i ) {
 helper->render( items[i] );
         if( i % ChunkSize == 0 )    // from time to time
```

```
            Yield();                                        // explicitly
yield control
      }
      helper->print();
   }
}
```

Now the code is resilient to having the **items** collection change during the call.

Of course, as in the discussion around Example 1(b), it's possible that we may not want the collection to change at all, for example to ensure we're processing a consistent snapshot of the collection's state. If so, we may need to save even more off to the side just like we did in Example 1(b), except here it's easier to do because we don't have to mess with a continuation object:

```
// Example 2(c): As resilient as Example 1(b)
//
class LongOperation : public Message {
public:
   void run() {
 Collection myItems = items.copy();
   LongHelper helper = GetHelper();
   for( int i = 0; i < myItems.size(); ++i ) {
      if( i % ChunkSize == 0 )           // from time to time
         Yield();                                                  //
explicitly yield control
      helper->render(myItems[i] );   // ok now do to this here
   }
   helper->print();
 Free( myItems );
  }
}
```

Notice that we can now correctly use **myItems[i]** both before and after **Yield** because we've insulated ourselves against the problematic state change.

What About *Yield*?

The **Yield** call contains a message pump loop that will process some or all waiting messages. Here's the most straightforward way to implement it:

```
// Example 3(a): A simple Yield that pumps all waiting messages
// (note: contains a subtle issue)
//
void Yield() {
   while( queue.size() > 1 ) { // do message pump
      message = queue.pop();
      message->run();
   }
}
```

Quick: Is there anything that's potentially problematic about this implementation? Hint: Consider the context of how it's used in Example 2(c) and the order in which messages will be handled.

The potential issue is this: With this **Yield**, Option 2 has a subtle but significant semantic difference from Option 1. In Option 1, the continuation was pushed on the current end of the queue and so enjoyed the fairness guarantee of being executed right after whatever waiting items were in the queue at the time it relinquished control. If more messages arrive while the continuation waits in line, they will get enqueued behind the continuation and be serviced after it in a pure first-in/first-out (FIFO) queue order—modulo priority order if the queue is a priority queue.

Using the **Yield** in Example 3(a), however, we've traded away these pure FIFO queue semantics because we will also execute any new messages that arrive after we call **Yield** but before we completely drain the queue. This might seem innocuous at first; after all, it's usually just "as if" we had called **Yield** slightly later. But notice what happens in the worst case: If messages arrive quickly enough so that the queue never gets completely empty, the operation that yielded might never get control again; it could starve. Starvation is usually unlikely, because normally we don't give a thread more work than it can ever catch up with. But it can arise in practice in systems designed to always keep just enough work ready to keep a thread busy and avoid making it have to wait, and so by design the thread always has a little backlog of work ready to do and the queue stays in a small-but-not-empty steady state. In that kind of situation, beware the possibility of starvation.

The simplest way to prevent this potential problem is to remember how many messages are in the queue and then pump only that many messages:

```
// Example 3(b): A Yield that pumps only the messages
// that were already in the queue at the time it began
//
void Yield() {
 int n = queue.size();
while( n-- > 0 ) {
                     // pump 'n' messages
      message = queue.Pop();
      message->run();
   }
}
```

This avoids pumping newer messages that arrive during the **Yield** processing and usually guarantees progress (non-starvation) for the function that called **Yield**, as long as we avoid pathological cases where the nested messages **Yield** again. (Exercise for the reader: Instrument **Yield** to detect starvation due to nesting.)

Incidentally, note that once again we're applying the technique of taking a snapshot of the state of the system as it was when we began the operation, just like we did in Examples 1(b) and 2(c) where we took a copy of the items collection. In this case, thanks to the FIFO nature of the queue, we don't need to physically copy the queue; it's sufficient to remember just the number of items to pump.

Summary

Sometimes a thread has to interleave its work in order to stay responsive, and avoid blocking new time-sensitive requests that may arrive while it's already processing a longer but lower-priority operation.

There are two main ways to interleave: Option 1 is to use continuations, which means saving the operation's intermediate local state in an object on the heap and creating a new message containing "the rest of the work left to do," which gets inserted into the queue so that we can handle other messages and then pick up and resume the original work where we left off. Option 2 is to use cooperative multitasking and reentrancy by yielding to a function that will pump waiting messages; this yields simpler code and avoids some allocation and queueing overhead because the locals can stay on the stack where they already are, but it also allows deeper stack growth.

In both cases, remember: The issues of interleaving other work are really nasty and it's all too easy to get things wrong, especially when **Yield** calls are sprinkled around and make the program very hard to reason about locally. Always be careful to remember that the interleaved work could have side effects, and make sure the longer work is resilient to changes to the data it cares about. If we don't do that correctly and consistently, we'll expose ourselves to a class of subtle and timing-dependent surprises. Next month, we'll consider a way to avoid this class of problems by making sure the state of the system is valid at all times, even with partially done work outstanding. Stay tuned.

Notes

[1] H. Sutter. "Use Threads Correctly = Isolation + Asynchronous Messaging (Dr. Dobb's Digest, March 2009).

Acknowledgments

Thanks to Terry Crowley for his insightful comments on drafts of this article.

By Herb Sutter Dr.Dobb's Journal
May 14, 2009
URL : http://drdobbs.com/go-parallel/article/217500206

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.

In two previous articles I pointed out the performance issue of false sharing (aka cache line ping-ponging), where threads use different objects but those objects happen to be close enough in memory that they fall on the same cache line, and the cache system treats them as a single lump that is effectively protected by a hardware write lock that only one core can hold at a time. [1,2] This causes real but invisible performance contention; whichever thread currently has exclusive ownership so that it can physically perform an update to the cache line will silently throttle other threads that are trying to use different (but, alas, nearby) data that sits on the same line. It's easy to see why the problem arises when multiple cores are writing to different parts of the same cache line, because only one can hold the exclusive hardware lock at a time. In practice, however, it can be even more common to encounter a reader thread using what it thinks is read-only data still getting throttled by a writer thread updating a different but nearby memory location, because the reading thread has to invalidate its copy of the cache line and wait until after the writer has finished to reload it.

A number of readers have asked for more information and examples on where false sharing arises and how to deal with it. I mentioned one concrete example in passing in [3] where Example 4 showed eliminating false sharing as one of the stages of optimizing a queue.

This month, let's consider a concrete example that shows an algorithm in extremis due to false sharing distress, how to use tools to analyze the problem, and the two coding techniques we can use to eliminate false sharing trouble.

The Little Parallel Counter That Couldn't

Consider this sequential code to count the number of odd numbers in a matrix:

```
int odds = 0;
for( int i = 0; i < DIM; ++i )
  for( int j = 0; j < DIM; ++j )
    if( matrix[i*DIM + j] % 2 != 0 )
       ++odds;
```

If our job is to parallelize existing code, this is just what the doctor ordered: An embarrassingly parallel problem where it should be trivial to achieve linear speedups simply by assigning 1/*P*-th of the independent workload to each of *P* parallel workers. Here's a simple way to do it:

```
// Example 1: Simple parallel version (flawed)
//
int result[P];
```

```
// Each of P parallel workers processes 1/P-th
// of the data; the p-th worker records its
// partial count in result[p]
for( int p = 0; p < P; ++p )
  pool.run( [&,p] {
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p];
  } );
// Wait for the parallel work to complete
pool.join();
// Finally, do the sequential "reduction" step
// to combine the results
odds = 0;
for( int p = 0; p < P; ++p )
  odds += result[p];
```

Quick: How well would you expect Example 1 to scale as *P* increases from 1 to the available hardware parallelism on the machine? You already have a hint from the topic of this column -- is there any part of the code that might worry you?

Let's find out. When I ran the code in Example 1 with values of *P* from 1 to 24 on a 24-core machine, I got the results shown in Figure 1.

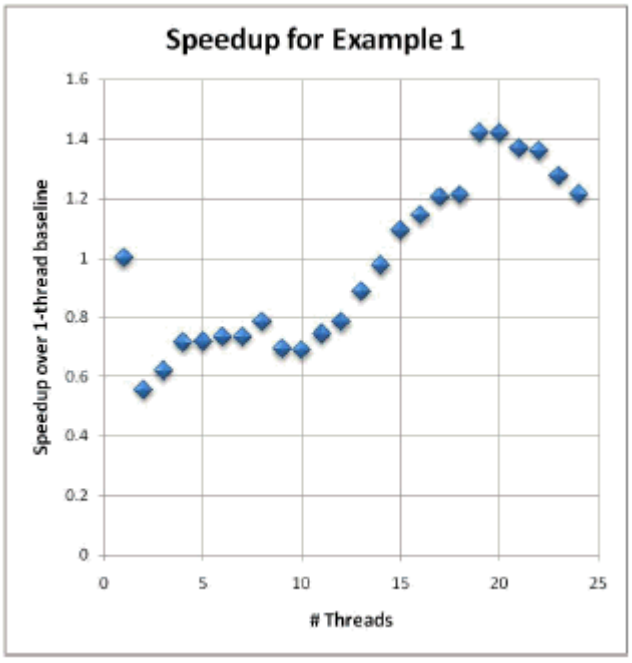

Figure 1: Example 1 seems to be about how to use more cores to get less total work done.

These results aren't just underwhelming; they're staggeringly bad. In most cases, the parallel code ran actually ran slower than the sequential code, and in no case did we get any better than a 42% speedup no matter how many cores we threw at the problem. Yet this is supposed

to be an embarrassingly parallel (i.e., embarrassingly easy to scalably parallelize) algorithm suitable for an introductory concurrency class. What's going on?

Analyzing What Went Wrong

To figure out where the problem lies, perhaps the first thing you might try to do is run this code while watching a CPU monitor to see which system cores are busy and for how long. If we run the example code with *P* set to 1, which makes it run sequentially, then we would expect to see one hardware core light up for the duration of the execution. Figure 2 shows that this is in fact exactly what happened on my test system. [4]

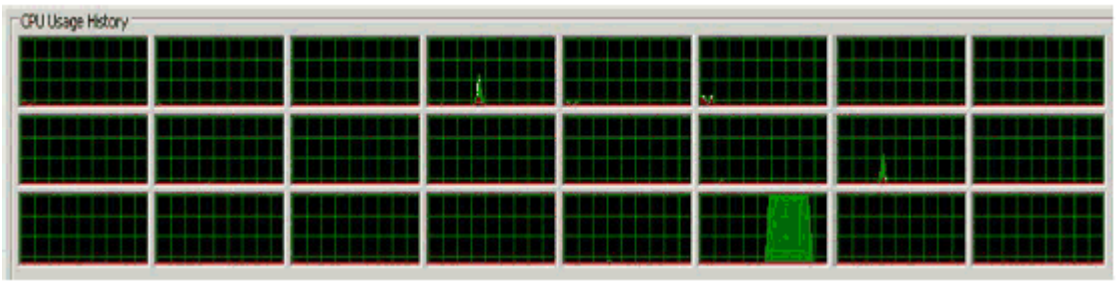

Figure 2: Running Example 1 with P = 1 (sequential baseline case).

Now let's run Example 1 again with *P* = 14, which will use 14 hardware cores if available, and watch the CPU monitor to see which cores are busy and for how long. I picked 14 just because in Figure 1 the execution time for *P* = 14 was about the same as for *P* = 1, so we can compare CPU utilization for about the same execution time as well as the same workload; remember that regardless of the value of *P* every execution is doing exactly the same amount of counting work on the identical data set, just carving the work up *P* different ways. Figure 3 shows the result on my system.

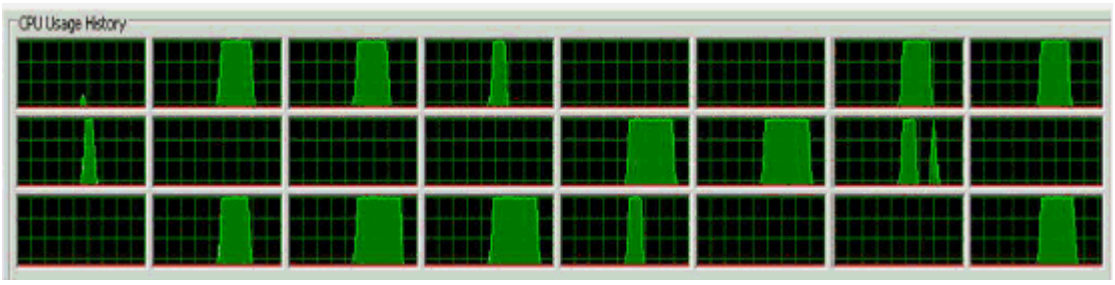

Figure 3: Running Example 1 with P = 14 (ugh, approximately equal execution time as for P = 1; see Figure 1).

What does Figure 3 tell us? First, it confirms that the parallel version is indeed lighting up 14 different cores, so we didn't make a mistake and accidentally run sequentially on a single core. So far, so good. Second, we see that some of the cores stay busy for as long as the single core in Figure 2 did, which is why the total execution time is about the same. Not great, to be sure, but at least it's consistent with our previous data.

Third, if you add up the total CPU busy time in Figure 3, we clearly used much more total CPU horsepower than in Figure 2. Yet both executions performed exactly the same number of traversals and additions; the work was merely divided differently. Now that's just crazy; it

means the CPU monitor must somehow be lying to us, because it claims that the cores are pegged and busily computing when that can't be true because we know full well the total number of matrix cell visits and counter additions that our program is doing is identical as in the execution in Figure 2. Hmm. Well, for now, let's just remember this on our list of mysteries to be solved and press on.

Fourth, we find a new clue: We can see some of the cores didn't stay busy as long as others. Now that's a puzzling thing, because each of the 14 workers was given an identically sized chunk of work to do and so should have taken the same length of time to do it. Yet some workers apparently ran faster (or slower) than others. Hmm; add that to the mystery list, too.

That's about as far as we can go with a CPU monitor. Time for other tools.

If you run this code under a performance analyzer that lets you examine the cycles per instruction (CPI) for each line of source code, you'll probably discover that the CPI for one line in particular is amazingly high: The offending line is **++result[p]**. Now, that ought to strike us as strange, because **++result[p]** isn't some heavyweight function call; it's just computing an offset into an array and then incrementing an integer, and so should have a very low CPI.

Next, if you run the code under a performance analyzer that can measure cache misses, or better still cache line contention, and attribute the cost to particular lines of code, you'll get even more targeted information about the line **++result[p]**: It's a hot spot that's spending its time waiting for memory, specifically for cache line ownership.

Put the CPI and cache miss information together, and there we have it: A classic case of false sharing. Each worker is incrementing its own distinct counter, but the counter values are adjacent in the result array. To increment its counter, a worker must have exclusive ownership of the cache line containing the counter, which means that the other workers trying to update their counters elsewhere in that same cache line must stall and wait until they can in turn get exclusive access to the line containing their chunk of result. The ownership of the cache line ping-pongs madly from core to core, and only one core can be running at a time, silently throttling the life out of the program.

For reasons that are invisible in the code, we ended up writing, not a truly parallel algorithm, but just a complicated sequential algorithm.

From Zero to Hero: The Little Parallel Counter That Could

The simplest way to fix the problem is simply to have each *p*-th worker increment its own local variable, and only at the end write its final tally to **result[p]**. It's an amazingly small change to the code:

```
// Example 2: Simple parallel version
// (de-flawed using a local variable)
//
int result[P];

// Each of P parallel workers processes 1/P-th
// of the data; the p-th worker records its
```

```
// partial count in result[p]
for( int p = 0; p < P; ++p )
   pool.run( [&,p] {
   int count = 0;
   int chunkSize = DIM/P + 1;
   int myStart = p * chunkSize;
   int myEnd = min( myStart+chunkSize, DIM );
   for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;
   result[p] = count;
} );
//  etc. as before
```

Could such a small change really make a big difference in scalability? Let's measure and find out. When I ran the code in Example 2 with values of *P* from 1 to 24 (on a 24-core machine), I got the results shown in Figure 4.

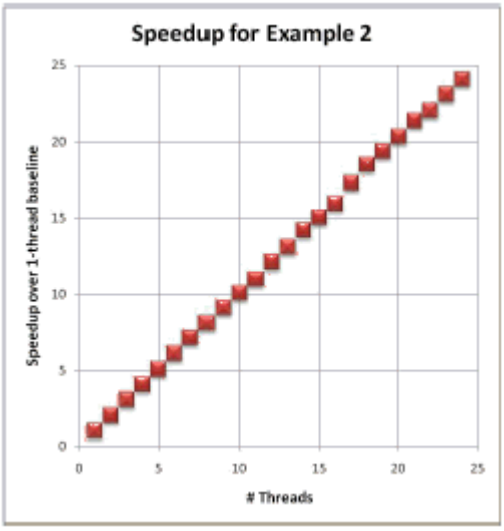

Figure 4: Removing cache line contention on the result array takes us from zero scaling to linear scaling, up to the available hardware parallelism (test run on a 24-core machine).

The amended code's scalability isn't just better -- it's perfect scaling, linear in the number of processors. Figure 5 shows the work per CPU core with *P* = 24; each core's work was complete so fast that it didn't manage to peg the core long enough to fill a whole CPU monitor polling interval.

Figure 5: Running Example 2 with P = 24 — now that's more like it, the kind of workload distribution we want to see.

Now that we've confirmed the culprit in Example 1 was memory contention due to false sharing on the result array, this helps clear up the mystery of why the cores appeared pegged in the CPU monitor (Figure 3) when they were actually not doing as much work: Many CPU monitors, like this one, count the time a core is waiting for cache and memory as part of its "busy" time. After all, the core is executing an instruction; it just happens to be an expensive memory fetch instruction. That explains why a core can appear to be fully utilized when it's actually only doing useful computation work a fraction of the time; it's spending the rest of its time just waiting for memory.

We've also cleared up the mystery of why some workers finished faster than others in Figure 3: The ones that took longer were the ones that were experiencing more contention because their counters happened to be on cache lines containing a greater number of other workers' counters. Workers whose counters happened to be on less-popular cache lines had to wait less and so ran faster.

Our small code change has taken us from zero scaling to perfect scaling: Now that's a zero-to-hero technique worth knowing about.

False Sharing: What To Look For

Note that Example 1 shows only one common case where data can end up being close together in memory, namely the case of elements of contiguous arrays. But the same thing can occur when we have two independently used fields within the same popular object, or objects are close to each other on the heap, or other situations.

The general case to watch out for is when you have two objects or fields that are frequently accessed (either read or written) by different threads, at least one of the threads is doing writes, and the objects are so close in memory that they're on the same cache line because they are:

- objects nearby in the same array, as in Example 1 above;
- fields nearby in the same object, as in Example 4 of [3] where the head and tail pointers into the message queue had to be kept apart;
- objects allocated close together in time (C++, Java) or by the same thread (C#, Java), as in Example 4 of [3] where the underlying list nodes had to be kept apart to eliminate contention when threads used adjacent or head/tail nodes;
- static or global objects that the linker decided to lay out close together in memory;
- objects that become close in memory dynamically, as when during compacting garbage collection two objects can become adjacent in memory because intervening objects became garbage and were collected; or
- objects that for some other reason accidentally end up close together in memory.

What To Do

When two frequently-used objects are sources of false sharing because they're in the same far-too-popular cache line, there are two general ways to remove the false sharing.

First, we can reduce the number of writes to the cache line. For example, writer threads can write intermediate results to a scratch variable most of the time, then update the variable in the popular cache line only occasionally as needed. This is the approach we took in Example 2, where we changed the code to update a local variable frequently and write into the popular result array only once per worker to store its final count.

Second, we can separate the variables so that they aren't on the same cache line. Typically the easiest way to do this is to ensure an object has a cache line to itself that it doesn't share with any other data. To achieve that, you need to do two things:

- Ensure that no other object can precede your data in the same cache line by aligning it o begin at the start of the cache line or adding sufficient padding bytes before the object.
- Ensure that no other object can follow your data in the same cache line by adding sufficient padding bytes after the object to fill up the line.

Here's how you can write this as a reusable library component in C++:

```
// C++ (using C++0x alignment syntax)
template<typename T>
struct cache_line_storage {
   [[ align(CACHE_LINE_SIZE) ]] T data;
   char pad[ CACHE_LINE_SIZE > sizeof(T)
        ? CACHE_LINE_SIZE - sizeof(T)
        : 1 ];
};
```

To get an object of type **MyType** that is stored on its own cache line, we would write **cache_line_storage<MyType>**. Note that this code assumes you've defined CACHE_LINE_SIZE to a suitable value for your target processor, commonly a power of two from 16 to 512. It also uses the standardized C++0x alignment syntax; if you don't have that yet, you can use a compiler-specific extension like Gnu's **__attribute__((aligned(x)))** or Microsoft's **__declspec(align(x))**.

If you're on .NET, you can write something similar but for value types only, which in their unboxed form are always laid out "inline" rather than as a separate heap object:

```
// C#: Note works for value types only
//
[StructLayout(LayoutKind.Explicit, Size=2*CACHE_LINE_SIZE)]
public struct CacheLineStorage<T>
   where T : struct
{
   [FieldOffset(CACHE_LINE_SIZE)] public T data;
}
```

It may seem strange that this code actually allocates enough space for two cache lines' worth of data instead of just one. That's because, on .NET, you can't specify the alignment of data beyond some inherent 4-byte and 8-byte alignment guarantees, which aren't big enough for our purposes. Even if you could specify a starting alignment, the compacting garbage collector is likely to move your object and thus change its alignment dynamically. Without

alignment to guarantee the starting address of the data, the only way to deal with this is to allocate enough space both before and after data to ensure that no other objects can share the cache line.

For Java and .NET full-fledged objects (reference types), the solution is basically the same as for .NET value types, but more intrusive: You need to add the before-and-after padding internally inside the object itself because there is no portable way to add external padding directly adjacent to an object.

Applying this second approach to Example 1, we could change just the definition of the result array to space the array elements far enough apart. For example:

```
// Example 3: Simple parallel version (de-flawed using padding)
//
cache_line_storage<int> result[P];
// etc. as before, just replacing result[p] with result[p].data
```

Running performance tests confirms that this results in the same scalability curve as Example 2.

Finally, Don't Forget This Affects Readers, Too

In Example 1 we've been considering the case where all workers are writers, but readers are affected too. Consider the following variant of Example 1 where we arbitrarily force half the workers to only perform reads from their **result[p]**, and measure the program's execution to see what happens:

```
// Example 3: Simple parallel version with half the accesses as
// reads (still flawed)
//
int result[P];
// Each of P parallel workers processes 1/P-th of the data;
// the p-th worker records its partial count in result[p]
for( int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int local = 0;
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          if( p % 2 != 0 ) // abitrarily have every second
                           // worker
            local += result[p]; // only read from its
                                // unshared result[p]
       else
         ++result[p];
  } );
//  etc. as before
```

How's the performance? To paraphrase Family Feud, in Figure 6 our "survey saaaays"

Alas, Example 3 is roughly just as bad as Example 1, and with more erratic performance behavior to boot. Even with half the workers performing only reads of their **result[p]**, even at *P* = 23 we get about the same performance as when *P* = 1.

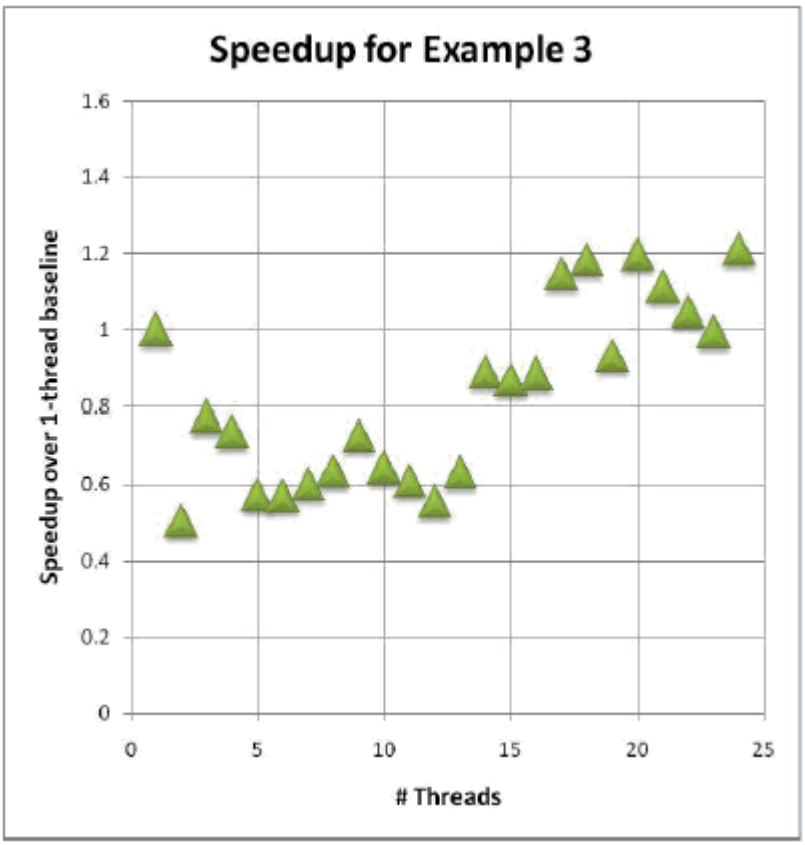

Figure 6: Example 3, with half the threads doing reads, is just as awful as Example 1.

Figure 7 shows the CPU monitor screen shots for *P* = 1 and *P* = 23, and confirms that for *P* = 23 we're lighting up 23 cores without any useful effect on total execution time.

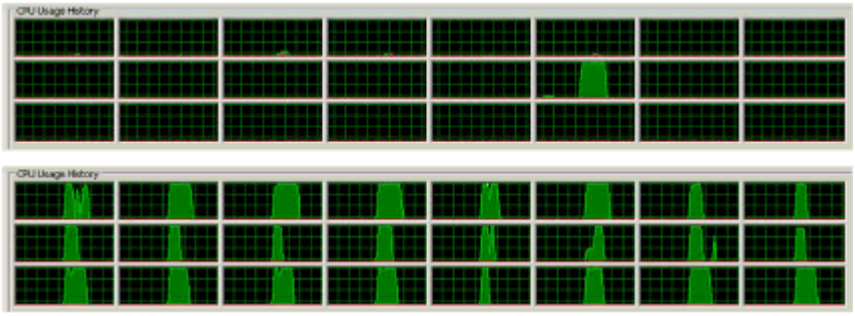

Figure 7: Running Example 3 with P = 1 and P = 23 (ouch, approximately equal execution time to perform the same work, as we saw in Figure 6).

Finally, Figure 8 summarizes the relative scalability of Examples 1 to 3 side by side, where ideal linear scalability would be a horizontalline at scaling = 1. As we've already seen,

Example 2 scales perfectly while Examples 1 and 3 don't scale at all -- and for no other reason than false sharing, even though in Example 3 half the workers are merely performing reads.

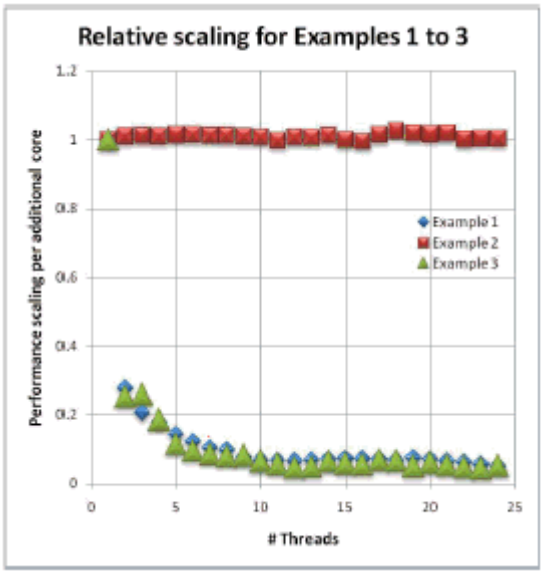

Figure 8: Examples 1 to 3 side by side (1 = perfect scaling).

Summary

Watch out for false sharing; it's an invisible scalability buster. The general case to watch out for is when you have two objects or fields that are frequently accessed (either read or written) by different threads, at least one of the threads is doing writes, and the objects are so close in memory that they're on the same cache line.

Detecting the problem isn't always easy. Typical CPU monitors completely mask memory waiting by counting it as busy time, which doesn't help us here, although the irregular lengths of the individual cores' busy times gives us a clue. Look for code performance analysis tools that let you measure, for each line of your source code, the cycles per instruction (CPI) and/or cache miss rates those source statements actually experience at execution time, so that you can find out which innocuous statements are taking extremely disproportionate amounts of cycles to run and/or spending a lot of time waiting for memory. You should never see high cache miss rates on a variable being updated by one thread in a tight inner loop, because it should just be loaded into cache once and then stay hot; lots of misses mean lots of contention on that variable or on a nearby one.

Resolve false sharing by reducing the frequency of updates to the falsely shared variables, such as by updating local data instead most of the time. Alternatively, you can ensure a variable is completely unshared by by using padding, and alignment if available, to ensure that no other data precedes or follows a key object in the same cache line.

Acknowledgments

Thanks to Joe Duffy and Tim Harris for their observations and comments on this article.

Notes

[1] H. Sutter. "Sharing Is the Root of All Contention" (Dr. Dobb's Digest, March 2009). http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=214100002.

[2] H. Sutter. "Maximize Locality, Minimize Contention." (Dr. Dobb's Journal, 33(6), June 2008.) http://www.ddj.com/architect/208200273.

[3] H. Sutter "Measuring Parallel Performance: Optimizing a Concurrent Queue" (Dr. Dobb's Journal, 34(1), January 2009). http://www.ddj.com/cpp/211800538.

[4] If you run this test yourself, you might instead see one core light up for a while, then go dark and a different core light up for a while, and so on. That's just an artifact of operating system thread scheduling, as the OS moves the thread to a different core for its own reasons, for example to keep heat distributed more evenly across the chip. The result is still logically equivalent to that in Figure 2 because only one core is running at a time, and the total execution time should not be materially affected by just occasional migration to a different core.

Dr.Dobb's

go parallel

Translating Multicore Power into Application Performance

Use Thread Pools Correctly: Keep Tasks Short and Nonblocking

How many threads must a thread pool pool? For a thread pool must pool threads.

By Herb Sutter Dr.Dobb's Journal
April 13, 2009
URL : http://drdobbs.com/go-parallel/article/216500409

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

---

What are thread pools for, and how can you use them effectively? As shown in Figure 1, thread pools are about letting the programmer express lots of pieces of independent work as a "sea" of tasks to be executed, and then running that work on a set of threads whose number is

automatically chosen to spread the work across the hardware parallelism available on the machine (typically, the number of hardware cores [1]). Conceptually, this lets us execute the tasks correctly one at a time on a single-core machine, execute them faster by running four at a time on a four-core machine, and so on.

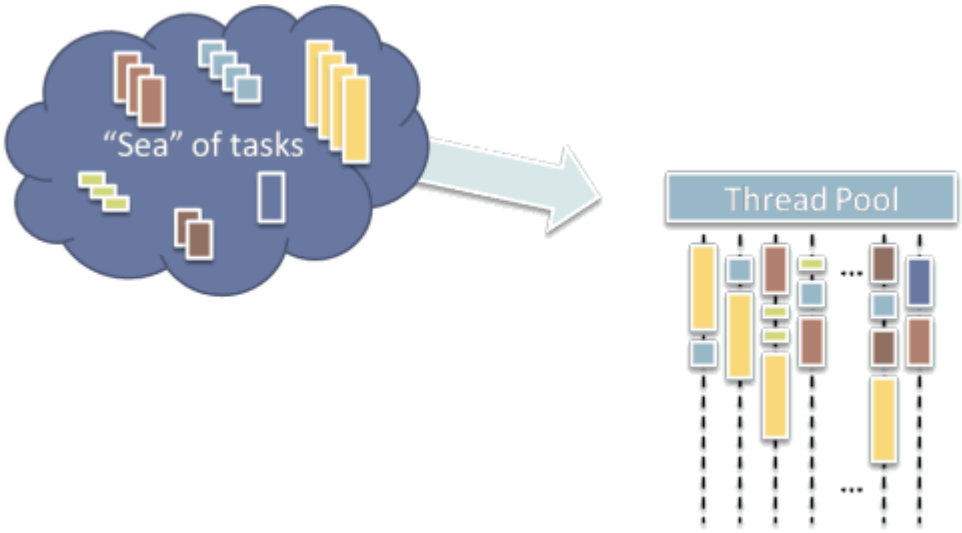

Figure 1: Thread pools are about taking a sea of work items and spreading them across available parallel hardware.

Besides scalable tasks, one other good candidate of work to run on a thread pool is the small "one-shot" thread. This is work that we might ordinarily express as a separate thread, but that is so short that the overhead of creating a thread is comparable to the work itself. Instead of creating a brand new thread and quickly throwing it away again, we can avoid the thread creation overhead by running the work on a thread pool, in effect playing "rent-a-thread" to reuse an existing pool thread instead. (See [2] for more about using threads correctly, including running small threads as pool work items.)

But the thread pool is a leaky abstraction. That is, the pool hides a lot of details from us, but to use it effectively we do need to be aware of some things a pool does under the covers so that we can avoid inadvertently hitting performance and correctness pitfalls. Here's the summary up front:

- Tasks should be small, but not too small, otherwise performance overheads will dominate.
- Tasks should avoid blocking (waiting idly for other events, including inbound messages or contested locks), otherwise the pool won't consistently utilize the hardware well -- and, in the extreme worst case, the pool could even deadlock.

Let's see why.

Tasks Should Be Small, but Not Too Small

Thread pool tasks should be as small as possible, but no smaller.

One reason to prefer making tasks short is because short tasks can spread more evenly and thus use hardware resources well. In Figure 1, notice that we keep the full machine busy until

we start to run out of work, and then we have a ragged ending as some threads complete their work sooner and sit idle while others continue working for a time. The larger the tasks, the more unwieldy the pool's workload is, and the harder it will be to spread the work evenly across the machine all the time.

On the other hand, tasks shouldn't be too short because there is a real cost to executing work as a thread pool task. Consider this code:

```
// Example 1: Running work on a thread pool.
pool.run( [=] { SomeWork(); } );
```

By definition, **SomeWork** must be queued up in the pool and then run on a different thread than the original thread. This means we necessarily incur queuing overhead plus a context switch just to move the work to the pool. If we need to communicate an answer back to the original thread, such as through a message or Future or similar, we will incur another context switch for that. Clearly, we aren't going to want to ship **int result = int1 + int2;** over to a thread pool as a distinct task, even if it could run independently of other work. It's just like the sign you see in a theme park at the entrance to the roller coaster: "You must be at least this big to go on this ride."

So although we like to keep thread pool tasks small, a task should still be big enough to be worth the overhead of executing it on the pool. Measure the overhead of shipping an empty task on your particular thread pool implementation, and as a rule of thumb, aim to make the work you actually ship an order of magnitude larger.

Tasks Should Avoid Waiting

Next, let's consider a key implementation question: How many threads should a thread pool contain? The general idea is that we want to "rightsize" the thread pool to perfectly fit the hardware, so that we provide exactly as much work as the hardware can actually physically execute simultaneously at any given time. If we provide too little work, the machine is "undersubscribed" because we're not fully utilizing the hardware; cores are going to waste. If we provide too much work, however, the machine becomes "oversubscribed" and we end up incurring contention for CPU and memory resources, for example in the form of needless extra context switching and cache eviction. [3]

So our first answer might be:

Answer 1 (flawed): "A thread pool should have one pool thread for each hardware core."

That's a good first cut, and it's on the right track. Unfortunately, it doesn't take into account that not all threads are always ready to run. At any given time, some threads may be temporarily blocked because they are waiting for I/O, locks, messages, events, or other things to happen, and so they don't contribute to the computational workload.

Figure 2 illustrates why tasks that block don't play nice with thread pools, because they idle their pool thread. While the task is blocked, the pool thread has nothing to do and probably won't even be scheduled by the operating system. The result is that, during the time the task is blocked, we are actually providing less parallel work than the hardware could run; the

machine is undersubscribed. Once the task resumes, the full load is restored, but we've wasted the opportunity to get more work done on otherwise-idle hardware.

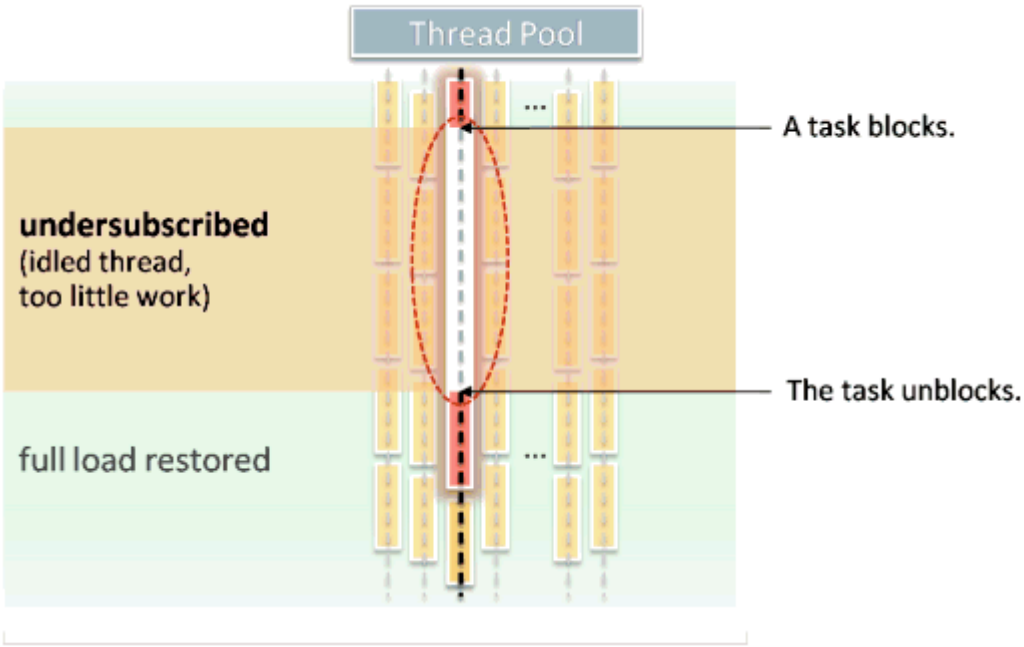

Figure 2: Blocking inside a task idles a pool thread.

At this point, someone is bound to ask the natural question: "But couldn't the pool just reuse the idled thread that's just sitting blocked anyway, and use it to run another task in the meantime?" The answer is "no," that's unsafe in general. A pool thread must run only one task at a time, and must run it to completion. Consider: The thread pool cannot reclaim a blocked thread and use it to run another task, thus interleaving tasks, because it cannot know whether the two tasks will interact badly. For example, the pool cannot know whether the blocked task was using any thread-specific state, such as using thread-local storage or currently holding a lock. It could be disastrous if the interleaved task suddenly found itself running under a lock held by the original task; that would be a fancy way to inject potential deadlocks by creating new lock acquisition orders the programmer never knew about. Similarly, the original task could easily break if it returned from a blocking call only to discover unexpected changes were made to its thread-local state made by an interloping interleaver. Some languages and platforms do provide special coroutine-like facilities (e.g., POSIX **swapcontext**, Windows **fibers**) that can let the programmer write tasks that intentionally interleave on the same thread, but these should be thought of as "manual scheduling" and "having multiple stacks per thread" rather than just one stack per thread, and the programmer has to agree up front to participate and has to opt into a restrictive programming model to do it safely. [4,5] A general-purpose pool can't safely just interleave arbitrary tasks on the same thread without the programmer's active participation and consent.

So we actually want to match, not just any threads, but specifically threads that are ready to run, with the hardware on this machine that can run them. Therefore a better answer is:

Answer 2 (better): "A thread pool should have one ready pool thread for each hardware core."

Figure 3 illustrates how a thread pool can deal with tasks that block and idle their pool thread. First, a thread pool has to be able to detect when one of its threads goes idle even though it still has a task assigned; this can require integration with the operating system. Second, once idling has been detected, the pool has to create or activate an additional pool thread to take the place of the blocked one, and start assigning work to it. From the time the original task blocks and idles its thread to the time the new thread is active and begins executing work, the system has more cores than work that is ready to run, and is undersubscribed.

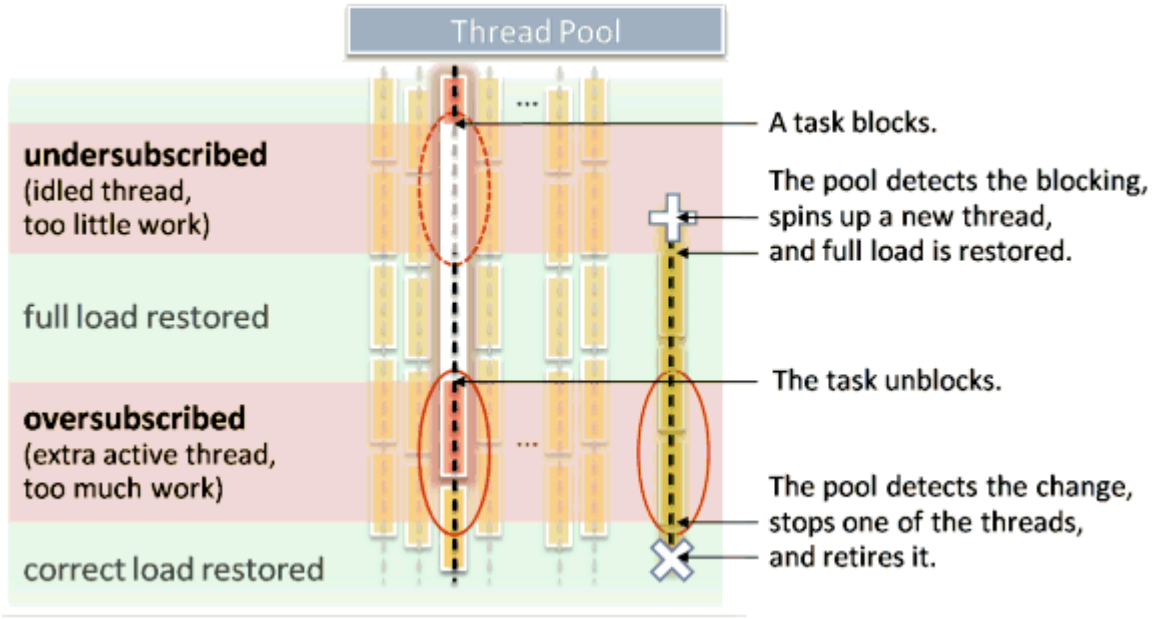

Figure 3: Thread pools can adapt by adding threads.

But what happens when the original task unblocks and resumes? Then we enter a period where the system is oversubscribed, with more work than there are cores to run the work. This is undesirable because the operating system will schedule some tasks on the same core, and those tasks will contend against each other for CPU and cache. Now we do the dance in reverse: The thread pool has to be able to detect when it has more ready threads than cores, and retire one of the threads as soon as there's an opportunity. Once that has been done, the pool is "rightsized" again to match the hardware.

The more often tasks block, the more they interfere with the thread pool's ability to match ready work with available hardware that can execute it.

Tasks Should Avoid Waiting For Each Other

The worst kind of blocking is when tasks block to wait on other tasks in the pool. To see why, consider the following code:

```
// Example 2: Launching pathologically interdependent tasks
//
// First, launch N tasks into the thread pool. Each task
// performs some work while blocking twice in the middle.
for( i = 0; i < N; ++i ) {
  pool.run( [=] {
```

```
    //  do some work
    phase1[i].wait();        // A: wait point #1
    //  and more work
    phase1[i+1].signal();        // B
    //  and more here
    phase2[i].wait();        // C: wait point #2
    //  and still more
    phase2[i+1].signal();        // D
  } );
}
// back on the parent thread
phase1[0].signal();    // release phase 1
phase1[N].wait();    // wait for phase 1 to complete
// E: what is the state of the system at this point?
phase2[0].signal();    // release phase 2
phase2[N].wait();    // wait for phase 2 to complete
```

This code launches **N** tasks into the pool. Each task performs work, much of which can execute in parallel. For example, there is no reason why the first "do some work" section of all **N** tasks couldn't run at the same time because those pieces of work are independent and not subject to any mutual synchronization.

There is some mutual synchronization, though. Each task has two points where it waits for the previous task—in this toy example it's just a simple wait; but in real code, this can happen when the task needs to wait for an intermediate result, for a synchronization barrier between processing stages, or for some other purpose. Here, there are two phases of processing. In each phase, each **i**th task waits to be signaled, does more work, then signals the following **i+1**th task.

Execution proceeds as follows: After launching the tasks, the parent thread kicks them past the first wait simply by signaling the first task. The first task can then proceed to do more work, and signal the next task, and so on until the last task signals the parent that phase 1 is now complete. The parent then signals the first task to wake it up from its second wait, and the phase 2 wakeups proceed similarly. Finally, all tasks are complete and the parent is notified of the "join" and continues onward.

Now stop for a moment and consider these questions:

- What is the state of the system when we reach line E?
- How many threads must the thread pool have to execute this program correctly? What happens if it doesn't have enough threads?

Let's consider the answers.

First, the state of the system at line **E** is that we have **N** tasks each of which is partway through its execution. All **N** tasks must have started, because **phase1[N]** has been signaled by the last task, which can only happen if the previous task signaled **phase 1[N-1]**, and so on. Therefore, each of the **N** tasks must have performed its line **B**, and is either still performing the processing between **B** and **C**, or else is waiting at line **C**. (No task can be past **C** because the parent thread hasn't kicked off the phase 2 signal cascade yet.)

So, then, how many threads must the thread pool have to execute this program? The answer is that it must have at least **N** threads, because we know there is a point (line **E**) at which every one of the **N** tasks must have started running and therefore each must be assigned to its own pool thread. And there's the rub: If the thread pool has fewer than N threads and cannot create any more, typically because a maximum size has been reached, the program will deadlock because it cannot make progress unless all tasks are running.

Just make **N** sufficiently large, and you can deadlock on most production thread pools. For example, .NET's **ThreadPool.GetMaxThreads** returns the maximum number of threads available; the current default is 250 threads per core, so to inject deadlock on a default-configured pool, let **N** be **250 × #cores + 1**. Similarly, Java's **ThreadPoolExecutor** lets you set the maximum number of pool threads via a constructor parameter or **setMaximumPoolSize;** the pool may also fail to grow if the active **ThreadFactory** fails to create a new thread by returning null from **newThread**.

At this point you may legitimately wonder: "But isn't Example 2 pathological in the first place? It even says so in the comment!" Actually, Example 2 is fairly typical of many kinds of concurrent algorithms that can proceed in parallel much of the time but occasionally need synchronization points as barriers to exchange intermediate results or enter a new stage of processing. The only thing that's pathological about Example 2 is that the program is trying to run each work item as a thread pool task; instead, each work item should run on its own thread, and everything would be fine.

Summary

Thread pool tasks should be as small as possible, but no smaller. The shorter the tasks, the more evenly they will spread across the pool and the machine, but the more the per-task overhead will start to dominate. Thread pool tasks should still be big enough to be worth the round-trip overhead of shipping them off to a pool thread to execute and shipping the result back, without having the overhead dominate the work itself. Thread pool tasks should also avoid blocking, or waiting for other things to happen, because blocking interferes with the ability of the pool to keep the right number of ready threads available to match the number of cores. Tasks should especially avoid blocking to wait for other tasks in the pool, because when that happens it can lead to worse performance penalties -- and, in the extreme worst case, having a large number of tasks that block waiting for each other can deadlock the entire pool, and with it your application.

Notes

[1] I'll use the term "cores" for convenience to denote the hardware parallelism. Some CPUs have more than one hardware thread per core, and so the **actual amount of hardware parallelism available = #processors × #cores/processor × #hardware threads/core**. But "total number of hardware threads" is a mouthful, not to mention potentially confusing with software threads, so for now I'll stick with talking about the "total number of cores."

[2] H. Sutter. "Use Threads Correctly = Isolation + Asynchronous Messages" (*Dr. Dobb's Digest*, April 2009).

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

[4] Single UNIX Specification (IEEE Standard 1003.1, 2004).

[5] Windows Fibers (MSDN).

A Base Class for Intrusively Reference-Counted Objects in C++

If you want to use Boost's intrusive_ptr, but have no reference counter at hand, here is a starting point

By Peter Weinert Dr.Dobb's Journal
February 21, 2011
URL : http://drdobbs.com/go-parallel/article/229218807

Peter Weinert is a scientific staff member of the Leibniz Supercomputing Centre in Munich, Germany.

This article presents a reference-counting base class template that is efficient and easy to apply for the smart pointer **intrusive_ptr**. Smart pointers encapsulate pointers to dynamically allocated resources such as heap memory or handles. They relieve you of explicit resource deallocation, therefore simplifying resource management, exception-safe programming, and the "Resource Acquisition Is Initialization" (RAII) idiom. Your tool box provides different smart pointer templates: **auto_ptr, unique_ptr, shared_ptr, weak_ptr**, or **intrusive_ptr**. This section sketches these template classes.

An Introduction to Common Smart Pointers

The **auto_ptr** class template provides a semantics of strict ownership of objects obtained via new ([C++03 §20.4.5]). Strict ownership means that no more than one **auto_ptr** object shall own the same object. The owned object is disposed using delete when the **auto_ptr** object itself is destroyed. Even though strict ownership precludes the **CopyConstructible** and the CopyAssignable requirements ([C++0x §20.2.1]), the copy constructor and the copy

assignment operators are accessible. If an **auto_ptr** object is copied, ownership is transferred, thereby resetting the source. The copy and the source are not equivalent. This limits usability and may lead to incorrect and dangerous use. C++0x therefore deprecates **auto_ptr** and offers **unique_ptr** as an improved class template. The **unique_ptr** class template provides a semantics of strict ownership of objects ([C++0x §20.9.9]). Contrary to **auto_ptr**, strict ownership is constrained by syntax. The copy constructor and the copy assignment operator are inaccessible. Instead, **unique_ptr** meets the **MoveConstructible** and **MoveAssignable** requirements ([C++0x §20.2.1]) and move semantics are enabled in order to transfer ownership. Therefore, a function can explicitly receive or return ownership. A template parameter accepts a client-supplied deleter that becomes part of the type. Disposal is indirected to this deleter. **unique_ptr** can manage objects obtained using allocators other than new. Because of this additional flexibility, the memory footprint may be larger compared to **auto_ptr**.

The **shared_ptr** class template implements semantics of shared ownership ([C++0x §20.9.10.2], [BoostSmartPtr]). More than one **shared_ptr** object can own the same object, satisfying the **CopyConstructible** and the **CopyAssignable** requirements. A **shared_ptr** object typically maintains an internal reference counter. The **make_shared** and **allocate_shared** helper template functions may efficiently combine the allocation of the reference counter and the owned object, as most implementations allocate the reference counter dynamically. Overloaded constructors accept a client-supplied deleter as well, if disposal using delete is not appropriate. Thanks to clever type elimination, this deleter is not part of the type (contrary to **unique_ptr**). Once the reference counter drops to zero, the owned object is disposed.

Semantics of strict and shared ownership can lead to cycles (e.g. objects owning each other) where the owned objects will never be disposed. A **weak_ptr** object does not own the object and is used to break cycles of **shared_ptr** objects ([C++0x §20.9.10.3], [BoostSmartPtr]). As opposed to a raw pointer, **weak_ptr** observes its **shared_ptr** and therefore avoids dereferencing a dangling pointer.

The **intrusive_ptr** class template provides a semantics of shared ownership [BoostIntrusivePtr]. As this smart pointer uses intrusive reference counting, the owned object must provide the reference counter. **intrusive_ptr** is not directly aware of this counter. You supply free functions, **intrusive_ptr_add_ref** and **intrusive_ptr_release**, which solely manipulate the counter. The constructors and the destructor of **intrusive_ptr** invoke these functions through unqualified calls with a pointer to the reference-counted object as an argument. The object is not directly disposed by **intrusive_ptr** but by **intrusive_ptr_release**. To sum up, some requirements must be addressed when using **intrusive_ptr**:

- Semantics of **intrusive_ptr_add_ref** and **intrusive_ptr_release**
- Disposal of the object
- Visibility of **intrusive_ptr_add_ref** and **intrusive_ptr_release**

A Naive Approach

I develop a base class template, **ref_counter**, that is efficient and easy to apply with **intrusive_ptr**, and guarantees disposal using delete. As often, the devil is in the details. Let's start with a test case:

```
// test.cpp: A simple class foo using the base class ref_counter

#include <boost/intrusive_ptr.hpp>
#include "ref_counter.h"

namespace foo {
class foo: public intrusive::ref_counter {};
}

int main( )
{
        boost::intrusive_ptr<foo::foo> p1(new foo::foo());
}
and get it running:
// ref_counter.h: A naive definition
#ifndef INTRUSIVE_REF_COUNTER_H_INCLUDED
#define INTRUSIVE_REF_COUNTER_H_INCLUDED
#include <boost/assert.hpp>

namespace intrusive {
class ref_counter
{
public:
        friend void intrusive_ptr_add_ref(ref_counter* p)
        {
                BOOST_ASSERT(p);
                ++p->counter_;
        }
        friend void intrusive_ptr_release(ref_counter* p)
        {
                BOOST_ASSERT(p);
                if (--p->counter_ == 0)
                        delete p;
        }
protected:
        ref_counter(): counter_(0) {}
        virtual ~ref_counter() = 0 {}
private:
        unsigned long counter_;
};
}
#endif
```

The class is fully exception safe, meeting the strongest guarantee, the no-throw guarantee (noexcept in C++0x). But have the requirements I mentioned above been met?

- Semantics: The counter (counter always refers to **ref_counter::counter_**) is initialized with zero, **intrusive_ptr_add_ref** increments the counter, and **intrusive_ptr_release** decrements the counter. Both free functions are friends, as they have to manipulate the encapsulated counter. In addition to the compiler-declared copy constructor and copy assignment operator, they provide the public interface of **ref_counter**. As the class is intended to be used as a base class, it is abstract and the constructor and destructor are placed in the protected section.
- Disposal: The object is disposed if the counter drops to zero. As it is deleted using a pointer to the base class, the destructor is virtual.
- Visibility: The **intrusive_ptr_add_ref** and **intrusive_ptr_release** functions reside in the same namespace scope as the base class **ref_counter**. They do not pollute the global namespace.

> In fact, their names are not even visible during an ordinary lookup [C++03 §14.6.5-2], because they are in the lexical scope of **ref_counter**. Their names are found through argument-dependent lookup (ADL, Koenig lookup [C++03 §3.4.2-2]).

Getting It Right

Did you spot the bug? Do the two free functions solely manipulate the counter? Think about compiler-generated definitions in **ref_counter**:

```
p1 = new foo::foo(*p1); // copy construction of foo
```

This code allocates a new foo-object using the copy constructor of foo, which uses the implicitly defined copy constructor of its **ref_counter** base class ([C++03 §12.8-8]), which copies the non-zero counter. But as a new **foo**-object is constructed, the counter should start over with zero. Remember, only the two free functions should manipulate the counter. This happens in the assignment operator of **intrusive_ptr**, where the old **foo** object is detached and the new one is attached (indirectly calling **intrusive_ptr_release** and **intrusive_ptr_add_ref**) using the "copy construct a temporary and swap" idiom. The copy assignment operator of **ref_counter** is another suspicious candidate:

```
*p1 = foo::foo(); // copy assignment of foo
```

The implicitly defined copy assignment operator performs memberwise assignment of its subobjects ([C++03] §12.8-13). Therefore, the implicitly defined copy assignment of **ref_counter** is called. It overwrites the left counter with the right (overwritten with zero in this case). Do not get confused: The **intrusive_ptr** object is not assigned to or changed and so is its pointer member, but the object pointed to is assigned to. The number of referencing pointers did not change. Therefore, the assignment operator of **ref_counter** must not alter the counter, but do nothing. Because **ref_counter** is an abstract base class, the missing explicit definitions are placed in the protected section:

```
// ref_counter.h: Adding copy constructor and assignment operator
class ref_counter
{
// …
protected:
        ref_counter(const ref_counter&) : counter_(0) {}
        ref_counter& operator=(const ref_counter&) { return *this; }
};
```

Always keep assignment and copy construction in mind, especially if the class is managing a resource. In addition, the free **std::swap** function works correctly with classes derived from **ref_counter**, such as **in std::swap(*p1, foo::foo())**. Swapping **ref_counter** does nothing, and a swap-member function looks like:

```
        void ref_counter::swap(ref_counter&) {} // noexcept
```

Cv-qualifier

A smart pointer should operate on both constant and volatile objects. But the compiler complains about a line as innocent as:

```
boost::intrusive_ptr<const foo::foo> p1(new foo::foo());
```

The problem is that the two free functions expect a pointer to a non-**const ref_counter** object, because the counter has to be manipulated. The solution is straightforward:

```
// ref_counter.h: Adding support for constant objects
class ref_counter
{
// …
        friend void intrusive_ptr_add_ref(const ref_counter* p);
        friend void intrusive_ptr_release(const ref_counter* p);
private:
        mutable unsigned long counter_;
};
```

The two free functions accept a pointer to a **const ref_counter** object while the counter is mutable, so it can yet be manipulated. I leave the support for the volatile qualifier as an exercise.

Polymorphism

The **ref_counter** class has a virtual destructor. A virtual destructor forces objects to carry a pointer to a virtual method table along with them. If **intrusive_ptr_release** did not polymorphically delete the object using a pointer to the base class, but using a pointer to the derived class, the virtual destructor could be abandoned. The **intrusive_ptr** class template calls the two free functions with a pointer, whose type is parameterized. This type is the derived class, not the **ref_counter** base class. Perfect. This means that **intrusive_ptr_release** can take a pointer to the derived class, but therefore, it must know the type of the derived class. Coplien's "Curiously Recurring Template Pattern" (CRTP) [C++Templates 16.3] provides the solution to this dilemma: A class, **X**, is derived from a class template instantiation, **base**, and uses itself as the template argument, **class X: base<X>**. Here is the definition of the class template announcing the type of the derived class using CRTP, the free functions taking pointers to the derived class and therefore obviating deletion using a base class pointer:

```
// ref_counter.h: disposal using a pointer to the derived class
template <class Derived>
class ref_counter
{
        friend void intrusive_ptr_add_ref(const Derived* p);
        friend void intrusive_ptr_release(const Derived* p);
// …
protected:
        ~ref_counter() {};     // non-virtual
//~ref_counter() = default; // in C++0x
};
```

The class definition of **foo** has to pass its type to the base class template **ref_counter**:

```
class foo: public intrusive::ref_counter<foo> {};
```

Meyers uses CRTP to provide a static counter for different derived types [Counting]. Here, I use CRTP in combination with the "friend name injection" (a component of the Barton-Nackman Trick) [C++Templates 9.2.2 and 11.7] to inject the two free non-template functions into namespace scope, accepting pointers to the derived class instead of pointers to the base

class. The **ref_counter** class template does not need a virtual destructor anymore. Of course the derived class may need one.

Enabling Private Inheritance

A derived foo class has no conceptual relationship to **ref_counter**, there is no reason for **ref_counter** to add to **foo**'s interface. Instead, **foo** uses the generic reference-counting implementation of **ref_counter**, it is implemented in terms of **ref_counter**. This is expressed with private inheritance, while public inheritance supplies an interface, not an implementation [EffectiveC++ item 42].

A client could gather a pointer to the base class with an implicit upcast. Is that sensible? Or is it even a violation of the Liskov Substitution Principle [LSP]? Clearly, **foo** should not be used as a **ref_counter**, **foo** IS-NOT-A **ref_counter**. So, it should not be possible to provide a **foo** where a **ref_counter** is expected. Class design should prevent the user from misuse. Prohibiting the implicit conversion from the derived class, **foo**, to the base class, **ref_counter**, can be done by private inheritance:

```
class foo: intrusive::ref_counter<foo> {};
```

Now the compiler complains of an inaccessible conversion from a pointer to the derived class to a pointer to the base class. The problem is that the two free functions have to access the counter in the base class through a pointer to the derived class. An implicit conversion and a **static_cast** from a pointer to the derived class to a pointer to the inaccessible base class both are ill-formed. Obviously, **dynamic_cast** (downcast) or **const_cast** (cv-qualifier conversion) won't do the trick. A **reinterpret_cast** is not only dangerous and implementation dependent, but it will fail in multiple inheritance. This leaves us with the dangerous explicit C-style cast. It will work even if the base class type is not accessible [C++03 §5.4-7 and §11.2-3], but the class type must be complete — otherwise, the explicit C-style cast may be interpreted as a **reinterpret_cast** instead of a **static_cast**. By the way, the **delete** expression also expects a complete class type if the class has a non-trivial destructor [C++03 §5.3.5-5]. **ref_counter** is complete, as within the class member specification, the class is regarded as complete within function bodies [C++03 §9.2-2]. Declarations inside the base class template are instantiated when the derived class is declared, but their bodies are instantiated after the full declaration or instantiation of the derived class, so **Derived** is complete [TMP 9.8]. Therefore, inside the friend function bodies, both object-types are completely defined. The C-style cast and the delete expression are safe and well defined:

```
//  ref_counter.h: enabling private inheritance
template <class Derived>
class ref_counter
{
        friend void intrusive_ptr_add_ref(const Derived* p)
        {
                BOOST_ASSERT(p);
                ++((const ref_counter*) p)->counter_;
        }
        friend void intrusive_ptr_release(const Derived* p)
        {
                BOOST_ASSERT(p);
                if (--((const ref_counter*) p)->counter_ == 0)
                        delete p;
        }
```

```
//…
};
```

Private inheritance is enabled and so is the **foo** IS-NOT-A **ref_counter** property.

Concurrency

Thread-safety is a complex mission. In C++0x, it is easy to make the counter atomic. Just include the <atomic> header and have the counter a type of **std::atomic_size_t**. The increment and decrement operations are atomic now. Now, the **ref_counter** class template offers the same level of thread safety as built-in types. Simultaneous reads are fine, but beware of other access patterns, for example:

```
// shared global
intrusive_ptr<foo> ps(new foo());

// Thread 1
intrusive_ptr<foo> p1(ps);    // read

// Thread 2
ps.reset();    // write
```

This is a data race that breaks invariances, even though the counter is atomic. The problem is that the **intrusive_ptr** operations are not atomic as a whole. Here is the copy constructor of **intrusive_ptr**:

```
intrusive_ptr( intrusive_ptr const & rhs ): px( rhs.px )
    {
        if( px != 0 ) intrusive_ptr_add_ref( px );
    }
```

Note that the pointer and the counter are not updated atomically, so, an invalid state can be observed. Just assume the following interleave:

- Thread 1 calls the copy constructor and performs the copy of the pointer. The invariants are broken, as the pointer was copied and the counter is not yet incremented.
- Thread 2 resets **ps**. This operation decrements the counter, which drops to zero, and thus the owned object is disposed.
- Thread 1 completes the copy construction, uses the dangling pointer, and accesses the counter. Bang! undefined behavior based on a data race.

The race cannot be avoided in **ref_counter**, but rather in **intrusive_ptr**. [AtomicRCP] provides a lock-free non-portable non-C++0x implementation for PowerPC. Boost's **shared_ptr** uses a spin-lock pool, **intrusive_ptr** can easily use a similar implementation. **shared_ptr** provides operations like **atomic_load, atomic_store, atomic_exchange,** or **atomic_compare_exchange** ([C++0x 20.9.10.5, N2674]). The interface of **intrusive_ptr** should offer these operations. A compatible interface has the advantage of easy refactoring between **intrusive_ptr** and **shared_ptr**.

Customized Disposal

The disposal of the object can be hard coded or customizable. The customization can be carried in the type (like in **unique_ptr**) or not (using type elimination like in **shared_ptr**). When it comes to intrusive reference counting, disposal gets intrusive as well. The interface of the class must expose customizable disposal or should include creation (and loosing flexibility), otherwise creation and disposal are not adhered. A clear architectural flaw. It is your choice to leave hard-coded disposal, add custom disposal in the type (maybe as in **unique_ptr** [C++0x 20.9.9.1] or use CRTP and empty-base-optimization for disposal), or use type elimination (using the function class template).

Alternative Implementations

A base class is not the only possibility to develop a reusable reference counter. If a reference-counting class is added as a class member, it would necessitate some member functions to reach the counter and the two free functions had to be delivered separately. This is too demanding. Another possibility is to add reference counting by deriving from the **foo** class:

```
template <class Base>
class add_ref_counter : public Base {/*…*/};
```

and to use it like this:

```
intrusive_ptr<add_ref_counter<foo> >
  p(new add_ref_counter<foo>());
```

A generic implementation of **add_ref_counter** would make use of C++0x variadic templates and perfect forwarding. The downside of such a technique would be a broken upcast in a class hierarchy, because it adds **add_ref_counter**-leaves. The **intrusive_ptr** object holds a pointer to a **add_ref_counter<foo>** object, but not to a **foo** object. Although a **foo_child** may be used as a **foo**, an **add_ref_counter<foo_child>** cannot be used as an **add_ref_counter<foo>**. On the other hand, this technique makes it possible to add reference counting and customized disposal to classes afterwards.

Results

Intrusive reference counting may be an efficient way to manage shared resources. I presented a way to implement a reusable and easy to apply base class. However, intrusive reference counting has its downsides. Its inflexible requirements force the managed object to hold the reference counter and to know something about disposal. This is a clear flaw, as it separates creation and disposal. Intrusive reference counting can be used only with classes designed this way. Finally, the interface of **intrusive_ptr** must be extended to provide thread safety. I prefer **shared_ptr<X> px(CreateX(), DestroyX)**, adhering creation with disposal. **X** does not have to know anything about **shared_ptr**, creation or disposal. Additionally, the interface of **shared_ptr** features thread-safe functions. Finally, **shared_ptr** is part of the C++0x standard library. I recommend using **shared_ptr** for shared ownership and **unique_ptr** for strict ownership. Only if you can prove that **intrusive_ptr** solves problems that you have with **shared_ptr** should you revert to **intrusive_ptr**. Here is the definition of the **ref_counter** class template for single-threaded in C++03:

```
template <class Derived>
class ref_counter
{
```

```
public:
        friend void intrusive_ptr_add_ref(const Derived* p)
        {
                ++((const ref_counter*) p)->counter_;
        }
        friend void intrusive_ptr_release(const Derived* p)
        {
                if (--((const ref_counter*) p)->counter_ == 0)
                        delete p;
        }
protected:
        ref_counter(): counter_(0) {}
        ref_counter(const ref_counter&) : counter_(0) {}
        ref_counter& operator=(const ref_counter&) { return *this; }
        ~ref_counter() {};
        void swap(ref_counter&) {};
private:
        mutable unsigned long counter_;
};
```

References

[C++03] ISO/IEC 14882:2003: "Programming Language C++," 2003, Wiley.

[C++0x] ISO/IEC Working Draft: "Standard for Programming Language C++," 2010, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf.

[BoostSmartPtr] Colvin, Dawes, Adler: "Boost smart pointers", http://www.boost.org/doc/libs/1_45_0/libs/smart_ptr/smart_ptr.htm.

[BoostIntrusive] Dimov: "Boost **intrusive_ptr** class template", http://www.boost.org/doc/libs/1_45_0/libs/smart_ptr/intrusive_ptr.html.

[C++Templates] Vandevoorde, Josuttis: "C++ Templates – The Complete Guide," 2003, Addison-Wesley Professional.

[Counting] Meyers: "Counting Objects in C++," 1998, http://www.drdobbs.com/184403484.

[EffectiveC++] Meyers: "Effective C++," 2nd ed., 1997, Addison-Wesley Professional.

[LSP] Martin: "The Liskov Substitution Principle," 1996, http://www.objectmentor.com/resources/articles/lsp.pdf.

[TMP] Abrahams, Gurtovoy: "C++ Template Metaprogramming," 2005, Addison-Wesley Professional.

[AtomicRCP] Reinholtz: "Atomic Reference Counting Pointers," 2004, http://www.drdobbs.com/184401888.

[N2674] Dimov, Dawes: "**shared_ptr** atomic access," revision 1, 2008, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2674.htm.

Dr.Dobb's

go parallel

Translating Multicore Power into Application Performance

Use Threads Correctly = Isolation + Asynchronous Messages

"Up-level" this low-level tool

By Herb Sutter Dr.Dobb's Journal
March 16, 2009
URL : http://drdobbs.com/go-parallel/article/215900465

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

Explicit threads are undisciplined. They need some structure to keep them in line. In this column, we're going to see what that structure is, as we motivate and illustrate best practices for using threads -- techniques that will make our concurrent code easier to write correctly and to reason about with confidence.

Where Threads Fit (and Thread Pools, Sometimes)

In [1], I described the three pillars of concurrency. The first two pillars summarize the two main kinds of concurrency we need to be able to express: 1. Keep things separate, so that independent parts of the program can run asynchronously. 2. Use more cores to get the answer faster using data-parallel and similar techniques. (The third pillar is about controlling concurrency once it has been expressed, using tools like locks and atomics.)

Table 1 summarizes these two pillars, and also summarizes how well each is served by four major tools at our disposal today for expressing concurrency: threads, thread pools, work stealing runtimes, and data-parallel facilities like OpenMP.

	I. Isolation Via Asynchronous Agents	II. Scalability Via Concurrent Collections
Tagline	Keep things separate	Re-enable the free lunch
Summary	Stay responsive and avoid blocking by running tasks in isolation and communicating via messages	Use more cores to get the answer faster by running operations on groups of things; exploit parallelism in data/algorithm structures
Examples	GUIs, web services, background print/compile, pipelining	Trees, quicksort, compilation
Threads	**OK (primary, Options 1 and 2):** For independent work that is long-running (e.g., background worker) or likely to wait/block	**No:** If you're using threads for this, chances are you're trying to write your own thread pool
Thread pools	**OK (secondary, Option 3):** A short-running thread that rarely waits/blocks can instead "rent-a-thread" by running as a pool work item	**OK:** Medium overhead (context switch per work item), medium flexibility
OpenMP	**No:** OpenMP tasks are not guaranteed to run asynchronously (on a different thread)	**OK:** Low overhead, low flexibility (intended principally for parallelizing Fortran and C loops)
Work stealing pools (Cilk, TBB, PPL, TPL, PLINQ, Fork/Join)	**No:** Tasks are not guaranteed to run asynchronously (on a different thread)	**Best:** Low overhead, high flexibility

Table 1: Expressing two kinds of concurrency.

Threads are about expressing Pillar 1 only, and this article will focus on that column: How to effectively use today's tools, notably threads and in some cases thread pools, to express independent work. We'll look at Pillar 2 in a future article.)

Threads: In a Nutshell

Here are the key things to know about threads:

- Threads are for expressing asynchronous work. The point of being asynchronous is to let the units of independent work in the application all run at their own speeds and better tolerate each other's latency.
- Threads are a low-level tool. Threads are just "sequential processes that share memory," and that kind of freewheeling anything-goes model doesn't provide any abstraction or guard rails to make good practices easy and bad practices hard. As aptly criticized by Edward Lee in his paper "The Problem with Threads" [2], threads let you do anything, and do it nondeterministically by default.
- "Up-level" them by replacing shared data with asynchronous messages. As much as possible, prefer to keep each thread's data isolated (unshared), and let threads instead communicate via asynchronous messages that pass copies of data. This best practice inherently encourages writing threads that are event-driven message processing loops, which gives inherent structure and synchronization and also improves determinism:
-

```
// An idealized thread mainline
//
do {
   message = queue.Receive();
   // this could block (wait)
   // ...
   // handle the message
   // ...
   } while( !done );
// check for exit
```

Ideally, each thread's logic should be built around the model of servicing its message queue, whether a simple FIFO queue or a priority queue (the latter if some messages should be given priority even if they arrive later).

- Highly responsive threads should not perform significant work directly. Some threads are responsible for interacting with the user (e.g., GUI threads) or with other processes or machines (e.g., socket and communications threads), or for other reasons need to respond to messages quickly. Such threads should perform nearly all their work asynchronously by posting the work to one or more helper threads or, where appropriate, to a pool thread. In particular, highly responsive threads should never block by waiting for a message or trying to acquire a lock.

A final note: In some cases, a thread that is short-running and will not block (wait idly for other events, including inbound messages or locks) can be expressed instead as a thread pool work item for efficiency, to avoid the overhead of creating a new thread from scratch. That's the one valid use of a thread pool for Pillar 1.

Example 1: GUI

GUI threads are a classic example of message pump-driven code. A GUI program is driven by queued events coming in from the user or elsewhere, and its code is organized as a set of responders providing the right thing to do in response to anything from the user pressing a button (e.g., Save, Print, search and replace) to a system event (e.g., window repaint, timer pulse, file system change notification). Many GUI systems offer priority queueing that lets later messages execute sooner even if other messages in the queue arrived earlier (e.g., **SendMessage** vs. **PostMessage** on Windows).

Consider the following simplified GUI pseudocode:

```
// Example 1: Sample synchronous GUI
// (not recommended)
//
while( message = queue.Receive() ) {
        // this could block
  if( it's a "save document" request ) {
    TurnSavingIconOn();
    SaveDocument();
       // bad: synchronous call
    TurnSavingIconOff();
  }
```

```
  else if( it's a "print document" request ) {
    TurnPrintingIconOn();
    PrintDocument();
      // bad: synchronous call
    TurnPrintingIconOff();
  }
else
  ...
}
```

Running nontrivial work synchronously on the GUI thread like this is a classic mistake, because GUIs are supposed to be responsive, and to be responsive requires not just responding to events but responding to them quickly. What if the GUI thread is busy processing a **PrintDocument** for several seconds, and in the meantime the user tries to resize or move the window? The resize or move request will be dutifully enqueued as a message that waits to be processed in its turn once the printing is complete; but in the meantime the user sees no effect and may try the same action, or give up and try something else instead, only to have all of the potentially duplicated or contradictory commands performed in sequence when the GUI thread is finally able to service them -- often with unintended effects. Synchronous execution leads to poor responsiveness and a poor user experience.

Threads that must be responsive should never execute high-latency work directly. This includes not just work that may take a lot of processing time, but also work that might have to wait for another thread, process, or computer -- including communications or trying to acquire a lock.

There are three major ways we can run high-latency work synchronously to move it off a thread that needs to stay responsive. Using the GUI thread as a case in point, let's examine them in turn and consider when each one is appropriate.

Async Option 1: Dedicated Background Thread

Our first option for moving work off the GUI thread is to execute it instead on a dedicated background worker thread. Because there is exactly one background worker for all the GUI-related grunt work, one potentially desirable effect is that the background work will be processed sequentially, one item at a time. This can be a good thing when two pieces of work would conflict (e.g., want to use the same mutable data) and so benefit from running sequentially; it can be a drawback when earlier tasks block later tasks that could have run sooner independently (see Option 2 below). Note also that messages don't have to go only one way: The asynchronous work can send back notifications ranging from a simple "I'm done" to intermediate information like progress status.

Figure 1 shows an example of this arrangement, where the user presses Print (dark blue) and then Save (light blue), and the two pieces of work are delegated to the worker where they can run asynchronously while the user continues to perform other operations such as moving windows or editing text (gray).

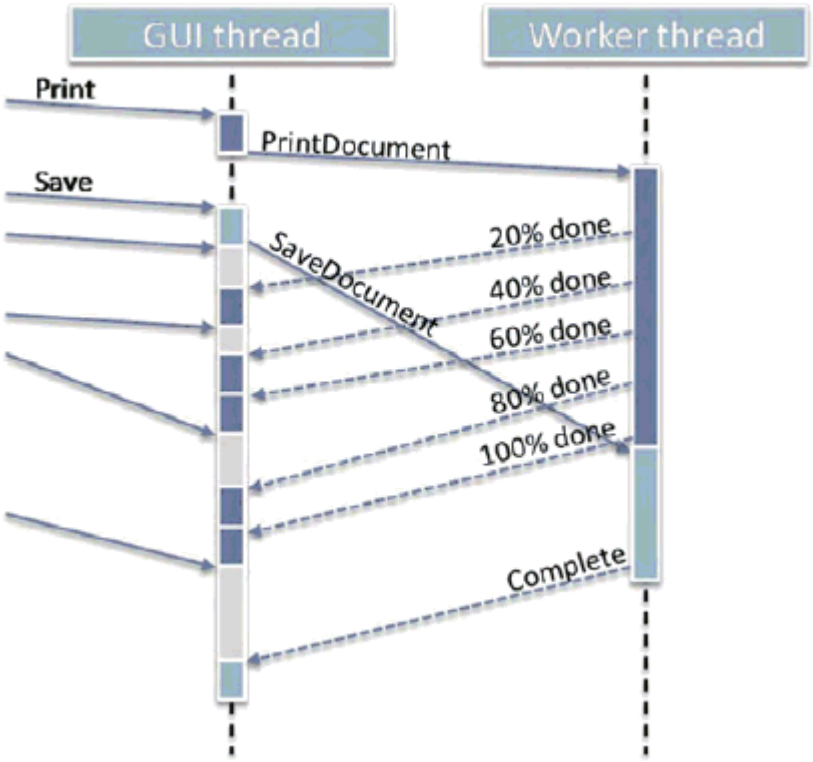

Figure 1: Getting work off the GUI thread using a dedicated background worker.

Let's consider two common ways to express the code for this option. First, we can arrange for the GUI thread to send tags representing the work that needs to be done:

```
// Option 1(a): Queueing work tags for a dedicated
// background thread. Suitable where tasks need
// to run sequentially with respect to each other (note
// corollary: an earlier task can block later tasks).

// GUI thread
//
while( message = queue.Receive() ) { // this could block
   if( it's a "save document" request ) {
      TurnSavingIconOn();
      worker.Send( new SaveMsg() ); // send async request
   }
   else if( it's a "save document" completion notification ) {
     TurnSavingIconOff();// receive async
// notification
   }
   else if( it's a "print document" request ) {
      TurnPrintingStatusOn();
      worker.Send( new PrintMsg() ); // send async request
   }
   else if( it's a "print document" progress notification ) {
      if( percent < 100 ) // receive async
// notification
         DisplayPrintPercentComplete( percent );
      else
         TurnPrintingStatusOff();
```

```
   }
   else
   ...
}
// Dedicated worker thread: Just interprets
// the tags on its queue and performs the
// appropriate action for each tag.
//
while( message = workqueue.Receive() ) {// this could block
   if( it's a "save document" request )
      SaveDocument(); // now sends completion
                                          // notification
   else if( it's a "print document" request )
      PrintDocument(); // now sends progress
                                          // notifications
   else
      ...                                 // etc., and check for
                                          // termination
}
```

Alternatively, instead of sending tags we can send executable messages, such as Java runnable objects, C++ function objects or lambdas, C# delegates or lambda expressions, or even C function pointers. This helps simplify the worker thread mainline:

```
// Option 1(b): Queueing runnable work for a dedicated
// background thread. Suitable where tasks need
// to run serially with respect to each other (note
// corollary: an earlier task can block later tasks).

// GUI thread
//
while( message = queue.Receive() ) { // this could block
   if( it's a "save document" request ) {
      TurnSavingIconOn();
      worker.Send( [] { SaveDocument(); } ); // send async work
   }
   else if( it's a "print document" request ) {
      TurnPrintingStatusOn();
      worker.Send( [] { PrintDocument(); } );  // send async work
   }
   else if( it's a "save document" notification ) { ... }
                                         // as before
   else if( it's a "print document" progress notification ) { ... }
                                         // as before
   else
      ...
}
// Simplified dedicated worker thread:
// Just executes the work it's being given.
//
while( message = workqueue.Receive() ) { // this could block
   message();                 // execute the given
                                          // work
   ...                                    // check for
                                          // termination
}
```

Async Option 2: Background Thread Per Task

Our second option is a variant of the first: Instead of having only one dedicated background worker, we can choose to launch each piece of asynchronous work as its own new thread. This makes sense when the work items really are independent and won't interfere with each other, though it can mean that work that is launched later can finish earlier, as illustrated in Figure 2.

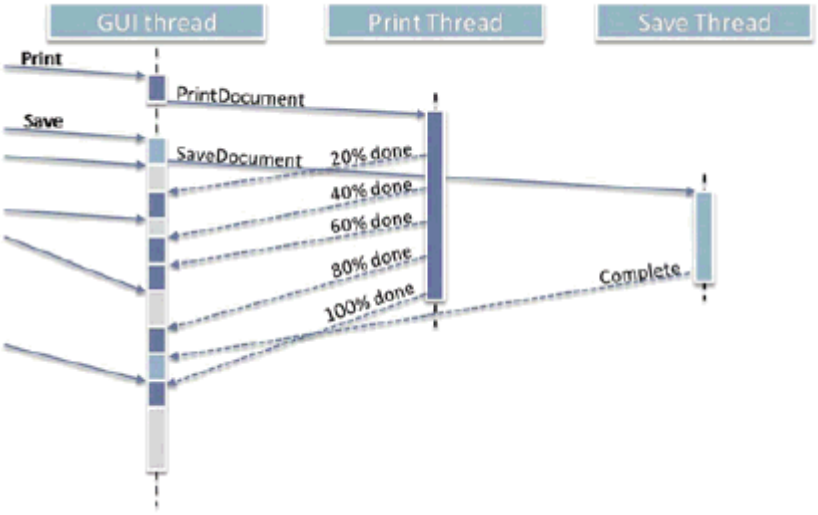

Figure 2: Getting work off the GUI thread using separate background workers.

Here's sample code that sketches how we can write this option, where red code again highlights the code that is different from the previous example.

```
// Option 2: Launching a new background thread
// for each task. Suitable where tasks don't need
// to run sequentially with respect to each other.
//
while( message = queue.Receive() ) { // this could block
   if( it's a "save document" request ) {
      TurnSavingIconOn();
      ...  new Thread( [] { SaveDocument(); } ); // run async request
   }
   else if( it's a "print document" request ) {
      TurnPrintingStatusOn();
       new Thread( [] { PrintDocument(); } );// run async request
   }
   else if( it's a "save document" notification ) { ... }
                                        // as before
   else if( it's a "print document" progress notification ) { ... }
                                        // as before
   else
      ...
}
```

The ellipsis before **new Thread** stands for any housekeeping we might do to keep track of the threads. In languages with garbage collection we would normally just launch a **new**

Thread and let it go; otherwise, without garbage collection we might maintain a list of launched threads and clean them up periodically.

Async Option 3: Run One-Shot Independent Tasks On a Thread Pool

Thread pools are about expressing independent work that will get run on a set of threads whose number is automatically chosen to match the number of cores available on the machine. Pools are mainly intended to enable scalability (Pillar 2), but can sometimes also be used for running what would otherwise be short threads.

Figure 3 shows how a short-running thread that rarely waits/blocks can instead "rent-a-thread" to run as a pool work item. Again, this technique is for simple one-shot work only, and it's very important not to run work on a thread pool if the work could block, such as wait to acquire a mutex or wait to receive a message (sending out messages is okay because that doesn't block).

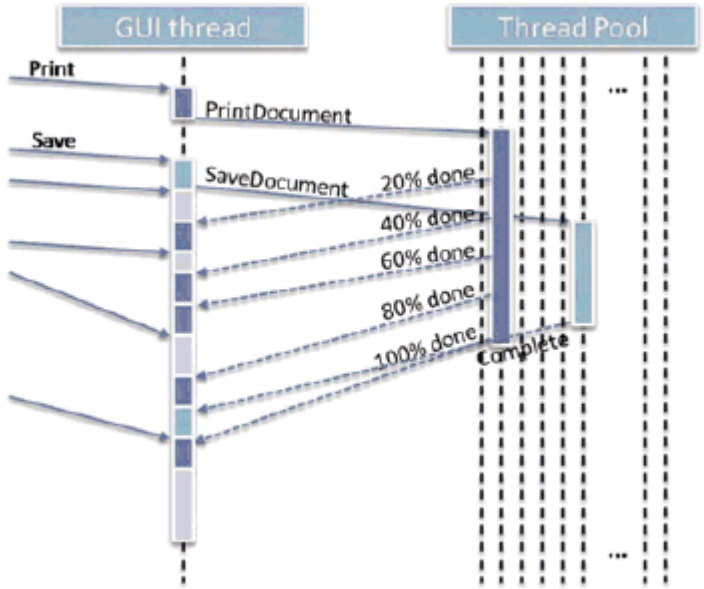

Figure 3: Getting work off the GUI thread using a thread pool (appropriate for shorter and non-blocking work).

Here's some sample code, which again should be considered pseudocode given that the specific spelling of "run this work on that pool" varies from one language and operating system to another:

```
// Option 3: Launching each task in a thread pool.
// Suitable for one-shot independent tasks that
// don't block for locks or communication.
//
while( message = queue.Receive() ) {   // this could block
   if( it's a "save document" request ) {
      TurnSavingIconOn();
      pool.run( [] { SaveDocument(); } ); // async call
   }
   else if( it's a "print document" request ) {
```

```
      TurnPrintingStatusOn();
      pool.run( [] { PrintDocument(); } ); // async call
   }
   else if( it's a "save document" notification ) { ... }
                                          // as before
   else if( it's a "print document" progress notification ) {  ... }
                                         // as before
   else
      ...
}
```

A Word About OpenMP and Work-Stealing Runtimes

Table 1 shows two other tools that are available (or becoming available) to express potentially asynchronous work, but for a different purpose:

- Work-stealing runtimes. Work stealing, pioneered by Cilk [3], is the engine behind the emerging crop of next-generation runtimes that target providing scalable concurrency (Pillar 2). They make it easy and efficient to express work that might be done in parallel if there are enough hardware resources to run it in parallel, yet add hardly any overhead if there aren't extra cores handy because the default implementation is to run the work sequentially in the original thread. Products based on work stealing include Intel's Threading Building Blocks [4], and at least four upcoming products: the Visual C++ 2010 Parallel Patterns Library (PPL) [5], the .NET 4.0 Task Parallel Library (TPL) [6] and Parallel LINQ (PLINQ) [7], and the Java 7 Fork/Join framework [8].
- OpenMP. Long available and now in version 3.0, OpenMP is all about parallelizing Fortran- and C-style loops, running them on a sort of thread pool under the covers to get data-parallel scalability. [9]

I mention these tools here primarily to say that they're meant for Pillar 2, as Table 1 shows. They're all about different ways to split and subdivide work across available cores to get the answer faster on machines having more cores, from doing loop iterations in parallel, to working on subranges of the data in parallel, to performing recursive decomposition (divide-and-conquer algorithms) in parallel.

These are not the right tools for Pillar 1, which means that they neither compete with threads nor replace them. The reason that work-stealing runtimes and OpenMP are unsuitable for running work asynchronously is because the work is not guaranteed to actually run off the original thread -- i.e., it's not guaranteed to be asynchronous at all.

For completeness, though, here's a taste of the syntax:

```
// Option 4 (NOT recommended for PrintDocument):
// Run on a work stealing or OpenMP runtime.
// Using Visual C++ 2010 Parallel Patterns Library (PPL)
// with the convenience of ISO C++0x lambdas
//
taskgroup.run( [] { PrintDocument(); } );
// Using .NET 4.0 Task Parallel Library (TPL)
// with the convenience of C# lambdas
//
Parallel.Invoke( () => { PrintDocument(); } );
```

```
// Using Java 7 ForkJoin with an explicit runnable object
//
class PrintTask extends ForkJoinTask {
   public void run() { PrintDocument(); }
   ...
}
...
fjpool.execute( new PrintTask(); );
```

OpenMP is not based on work stealing, but does offer a similar construct with the same caveat:

```
// Using OpenMP (again, NOT recommended for PrintDocument)
//
#pragma omp task
{
   PrintDocument();
}
```

Example 2: Sockets

Another classic example of event-driven code is client-server communications. A server listens for incoming connection requests; as each connection arrives, the server has to accept it and then process inbound and outbound traffic on that connection until the connection is closed by one side or the other. At any given time, the server has to be able to deal with multiple active connections from different clients, each at a different stage in its processing.

As with the previous GUI example, we have the option of handling the work synchronously or asynchronously. The synchronous code looks something like this in a classic socket-oriented API. The methods are spelled a little differently in POSIX, Java, and other environments, but all follow the basic operation.

```
// Example 2: Doing it all on one thread/process
// (pseudocode, and assumes select returns single events)
//
while( /*... */ ) {
   select( our socket list ); // blocks until something is ready
   if( a connection request came in )
      accept() and add it into to our socket list
   else if( a connection closed )
      close() and remove it from our socket list
   else if( one of our read sockets is ready )
                           // synchronous processing
      read more data from that socket and process it
   else if( one of our write sockets is ready )
                           // synchronous processing
   write the next chunk of data to that socket
}
```

A key problem with this is that all the work is being handled by a single thread. For one thing, that tends to make the code more complex; one thread has to do it all, so all the code to

handle all the different kinds of clients appears in one place. For another, running all the work on one thread means that all the work will run sequentially on one core (or, equivalently, one core at a time if the operating system migrates the thread from core to core). If the total work saturates the core, then additional work will be throttled and slow down the server's responsiveness for all clients as later requests queue up and wait for earlier work to complete.

A common solution is to use Option 2: Let the server service connectionscsimultaneously by launching a new thread to handle each incoming client. Here's the basic logic:

```
// Applying Option 2 (pseudocode)
// Listener thread: Only accepts connections
// and launches new threads to deal with them
//

while( /**/ ) {
   wait for connection request
   accept()
   ... = new Thread( [] { HandleIt( theConnection) ); } );
}
// Each handler thread
//
void HandleIt( ... ) {
   while( wait for event on this connection ) {
                        // this could block
      if( this connection was closed ) {
        close()
        return;
      }
      else if( incoming data ready to read )
                    // synchronous processing
        read more data from this connection and process it
      else if( connection is ready to write )
                   // synchronous processing
        write the next chunk of data to this connection
}
```

This code is both clearer and potentially faster. The program is clearer because the code to handle each kind of client is nicely wrapped up in its own method or class, instead of being intermingled. The program is faster because it keeps different connections asynchronous and independent, so that the work on one connection doesn't have to wait to be processed sequentially behind work on another connection. This gives better responsiveness even on a single-core server that isn't heavily loaded, and it delivers better scalability under load on servers that do have parallel hardware.

Example 3: Pipeline Stages

A third classic example of independent work that should run asynchronously is pipelining, and the independent pieces of work are the pipeline stages. In a nutshell, here's the idea: We have a series of pieces of data to be processed in order (e.g., series of packets to prepare for sending). Each piece has to go through several stages of processing (e.g., decorate, compress, encrypt) that need to be applied to a given piece of data in order.

The stages are otherwise independent (e.g., the compressor can run independently of the decorator and encryptor), other than waiting for available work to arrive from the previous stage. As shown in Figure 4, we can express the pipeline with one thread per stage, connected by asynchronous message queues to let the stages run independently and tolerate latency as the different stages typically run at different speeds. Each stage just takes each incoming packet from the previous stage, does its own processing, and throws it over to the next stage.

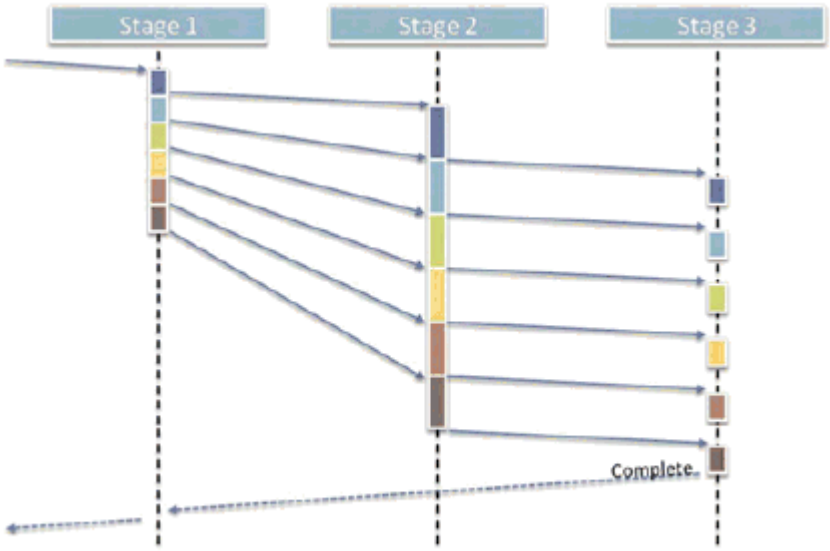

Figure 4: Using three threads to express three pipeline stages.

Summary

Threads are a low-level tool for expressing asynchronous work. "Uplevel" them by applying discipline: strive to make their data private, and have them communicate and synchronize using asynchronous messages. Each thread that needs to get information from other threads or from people should have a message queue, whether a simple FIFO queue or a priority queue, and organize its work around an event-driven message pump mainline; replacing spaghetti with event-driven logic is a great way to improve the clarity and determinism of your code.

Notes

[1] H. Sutter. "The Pillars of Concurrency" (Dr. Dobb's Journal, August 2007). http://www.ddj.com/architect/200001985.

[2] E. Lee. "The Problem with Threads" (EECS Department Technical Report, University of California, Berkeley, 2006). http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html.

[3] The Cilk Project website: http://supertech.csail.mit.edu/cilk/.

[4] Intel Threading Building Blocks website: http://www.threadingbuildingblocks.org/.

[5] K. Kerr. "Visual C++ 2010 and the Parallel Patterns Library (MSDN Magazine, February 2009). http:// msdn.microsoft.com/en-us/magazine/dd434652.aspx.

[6] D. Leijen and J. Hall. "Optimize Managed Code For Multi-Core Machines" (MSDN Magazine, October 2007). http://msdn.microsoft.com/en-us/magazine/cc163340.aspx.

[7] J. Duffy and E. Essey. "Running Queries On Multi-Core Processors" (MSDN Magazine, October 2007). http://msdn.microsoft.com/en-us/magazine/cc163329.aspx.

[8] D. Lea. "Package jsr166y" (Draft JSR166y documentation). http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/.

[9] OpenMP Application Program Interface Version 3.0 (OpenMP.org, May 2008). http://www.openmp.org/mp-documents/spec30.pdf.

Dr.Dobb's

Sharing Is the Root of All Contention

Sharing requires waiting and overhead, and is a natural enemy of scalability

By Herb Sutter Dr.Dobb's Journal
February 13, 2009
URL : http://drdobbs.com/go-parallel/article/214100002

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

Quick: What fundamental principle do all of the following have in common?

- Two children drawing with crayons.
- Three customers at Starbucks adding cream to their coffee.
- Four processes running on a CPU core.
- Five threads updating an object protected by a mutex.
- Six threads running on different processors updating the same lock-free queue.
- Seven modules using the same reference-counted shared object.

Answer: In each case, multiple users share a resource that requires coordination because not everyone can use it simultaneously -- and *such sharing is the root cause of all resulting contention* that will degrade everyone's performance. (Note: The only kind of shared resource that doesn't create contention is one that inherently requires no synchronization because all users can use it simultaneously without restriction, which means it doesn't fall into the description above. For example, an immutable object can have its unchanging bits be read by any number of threads or processors at any time without synchronization.)

In this article, I'll deliberately focus most of the examples on one important case, namely mutable (writable) shared objects in memory, which are an inherent bottleneck to scalability on multicore systems. But please don't lose sight of the key point, namely that "sharing causes contention" is a general principle that is not limited to shared memory or even to computing.

The Inherent Costs of Sharing

Here's the issue in one sentence: Sharing fundamentally requires waiting and demands answers to expensive questions.

As soon as something is shared in a way that doesn't allow unlimited simultaneous use, we are forced to partially or fully serialize access to it. The word "serialize" should leap out as a bright red flag -- it is the inverse and antithesis of "parallelize," and a fundamental enemy of scalability. That the very act of sharing a thing demands overhead is inherent in the notion of sharing itself, not only in software; it applies equally to "one child at a time can use the blue crayon" as to "one process at a time can write to the file" and "one writer at a time can use the shared object."

That we're forced to serialize access, in turn, means two things: we are forced to (potentially) wait, and we are also forced to start answering expensive questions like: "*Can I use it now?*" Computing the answer to any question in software costs CPU cycles. But these are more expensive questions, because answering them requires agreement at many levels of the system -- across software threads and processes, and across hardware processors and memory subsystems -- and getting that agreement can cost substantial communication and synchronization overhead that's not even visible in the source code.

A Roadmap

Table 1 helps to map out these costs by breaking down their sources (rows) and how they manifest (columns), and giving examples of each case. The rows categorize the sources into three kinds of sharing overhead, where the first two are fundamental and the third arises when attempts to optimize these costs fail:

- **Blocking progress (fundamental): Sharing requires waiting.** Clearly, when one client has exclusive access to the resource, some or all other clients must wait idly and are unable to make progress. (The worst case is when a client could starve, or be forced to wait indefinitely.)
- **Slowing progress (fundamental): Sharing demands answers to expensive questions**. Beyond actual waiting, there is usually a cost for just asking for the coordination -- whether or not it is actually needed.

- **Wasting progress (secondary): Resolving contention can mean throwing away work.** Additionally, some protocols perform optimistic execution to mitigate the first two costs and perform faster in the case of no contention. But when contention actually happens, the protocol may have to undo and redo otherwise -- useful work. The more contention we encounter, the more effort we waste.

	A. Software, visible in your source code	B. Software, invisible in your source code	C. Hardware, invisible in your source code
1. Blocking progress: The basic waiting requirement. When it's one's turn the others inherently must wait	Blocking calls like mutex.lock(), semaphore.wait() Opening a file for write Writer locking a RW mutex "CAS loop" in lock-free & obstruction-free algorithms	Threads sharing a CPU core: context switches Synchronized I/O (e.g., console)	Write to shared memory: exclusive ownership False sharing: two *possibly-unshared* objects in a cache line Tasks using the same spindle (e.g., HDD, DVD) Tasks accessing the same server
2. Slowing progress: Coordination / synchronization overhead. Extra cost of synchronization that we wouldn't otherwise need	mutex.unlock() Atomic variable read or write (esp. write), even in wait-free algorithms Compare-and-swap (CAS)	Threads sharing a CPU core: context switches Readers locking a RW mutex: CAS Sharing a reference counted object: CAS Inhibits optimizations	Write to shared memory: sync after write, propagate values to other caches/memory CAS: full memory sync before & after to guarantee atomicity Inhibits optimizations
3. Wasting progress: Throwing away work. Having to undo/redo	Lock-free algorithms involving a retrying "CAS loop"	Backoff/retry protocols	CAS implemented with loop Tasks sharing a cache: eviction

Table 1: Examples of contention penalties

These penalties also manifest in various ways, shown in the columns. The simplest case is when the potential cost is visible in our source code (column A). For example, just by looking at the code we know that the expression **mutex.lock()** might cause our thread to wait idly. But each penalty also manifests in invisible ways, both in software and in hardware (columns B and C), particularly on multicore hardware.

Visible Costs (A1, A2, A3)

Consider the following code which takes locks for synchronization to control access to a shared object **x**:

```
// Example 1: Good, if a little boring;
// plain-jane synchronized code using locks

// Thread 1
mutX.lock();           // A1, A2: potentially blocking call
Modify( x, thisWay );
mutX.unlock();         // A2: more expense for synchronization

// Thread 2
mutX.lock();           // A1, A2: potentially blocking call
Modify( x, thatWay );  // ok, no race
mutX.unlock();         // A2: more expense for synchronization

// Thread 3
```

```
mutX.lock();           // A1, A2: potentially blocking call
Modify( x, anotherWay ); // ok, no race
mutX.unlock();         // A2: more expense for synchronization
```

The obvious waiting costs (type A1) are clear: You can read them right in the source code where it says **mutX.lock()**. If these threads try to run concurrently, the locks force them to wait for each other; only one thread may execute in the critical section at a time, and the use of **x** is properly serialized.

The A1 waiting characteristics may vary. For example, if **mutX** is a reader/writer mutex, that will tend to improve concurrency among readers, and readers' lock acquisitions will succeed more often and so be cheaper on average; but acquiring a write lock will tend to block more often as writers are more likely to have to wait for readers to clear. We could also rewrite the synchronization in terms of atomic variables and compare-and-swap (CAS) operations, perhaps with a CAS loop that optimistically assumes no contention and goes back and tries again if contention happens, which is common in all but the best lock-free algorithms (lock-free and obstruction-free; only wait-free algorithms avoid A1 entirely). [2]

We also incur the performance tax of executing the synchronization operations themselves (A2). Some of these are probably obvious; for example, taking a lock on a contended mutex clearly requires spinning or performing a context switch which can be expensive. But even taking an uncontended lock on an available mutex isn't free, because it requires a synchronized memory operation such as a compare-and-swap (CAS). And, perhaps surprisingly, so does releasing a lock with **mutX.unlock()**, because *releasing* a lock requires writing a value that signifies the mutex is now free, and that write has to be synchronized and propagated across the system to other caches and cores. (We'll have more to say about memory synchronization when we get to cases C1-C3.)

Even the best wait-free algorithms can't avoid paying this toll, because wait-free algorithms are all about avoiding the waiting costs (A1); they still rely on synchronized atomic memory operations and so must pay the overhead taxes (A2). On many mainstream systems, the write to unlock a mutex, or to write or CAS an atomic variable in lock-free code, is significantly more expensive than a normal memory write for reasons we'll consider in a moment, even when there are no other contending threads on other processors that will ever care to look at the new value.

Finally, some algorithms may actually have to redo their work (A3), particularly if they perform optimistic execution. Lock-free code to change a data structure will frequently do its work assuming that no other writers are present and that its final atomic "commit" will succeed, and perform the commit using a CAS operation that lets it confirm whether the key variable still has its original expected value; if not, then the commit attempt failed and the code will need to loop to try again and attempt a CAS commit again (hence, the term "CAS loop") until it succeeds. Each time the code has to retry such a loop, it may need to partially or entirely throw away the work it did the previous time through. The more contention there is, the more often it will have to re-execute. On top of losing the work, there is often an extra overhead if abandoning the work requires that we do some cleanup (e.g., deallocation or destruction/disposal of scratch resources) or perform compensating actions to "undo" or "roll back" the work.

Besides shared memory operations, other kinds of shared resources cause similar problems. For example, if two processes try to open a file for exclusive access with a blocking call, one must wait (A1), for the same reason one child must wait for another to finish using a shared green crayon. Opening a file for exclusive use is also usually a more expensive operation than opening it for read-only use because it incurs more coordination overhead (A2); it's the same kind of overhead as when two coffee customers are waiting for the cream, and after waiting for the previous customer to finish they have to make eye contact and maybe even talk to decide which of the waiters gets to go next.

In all these cases, the key point is that the potential blocking or overhead or retrying is clearly visible in the source code. You can point to the lines that manipulate a mutex or an atomic variable or perform CAS loops and see the costs. Unfortunately, life isn't always that sweet.

Invisible Costs in Software (B1, B2, B3)

Column B lists examples of some costs of sharing that still occur in software, but typically aren't visible by reading your own source code.

Two pieces of code may have to wait for each other because of synchronization being done under the covers by low-level libraries or by the operating system (B1). This can arise because they share a resource (not just memory): For example, two threads may try to write to the console, or append to the same file, using a synchronized I/O mode; if they try to execute their conflicting actions at the same time, the I/O library will serialize them so that one will block and wait until the other is done before proceeding. As another example, if two threads are running on the same CPU core, the operating system will let them share the core and have the illusion of executing simultaneously by making them take turns, interleaving their execution so that each periodically is preempted and waits for the other to get and use its next quantum.

These kinds of invisible sharing can also impose invisible synchronization overhead costs (B2). In the case of two threads sharing a CPU core, in addition to waiting they incur the overhead of context switches as the operating system has to save and restore the registers and other essential state of each thread; thanks to that tax, the total CPU time the threads receive adds up to less than 100 percent of the processor's capacity.

While we're speaking of B2, I have some mixed news for fans of reader/writer mutexes and reference counting. A reader/writer mutex is a common tool to achieve better concurrency for a "read-mostly" shared object. The idea is that, since readers dominate and won't interfere with each other, we should let them run concurrently with each other. This can work pretty well, but taking a read lock requires at least an expensive CAS on the shared reference count, and the CAS expense grows with the number of participating threads and processors. (Note that many reader/writer mutexes require much more overhead than just that because they maintain other lists, counters, and flags under the covers, so that they can block readers when a writer is waiting or active, allow priority entry, and maintain fairness and other properties.) [12]. The shorter the reader critical section and the more frequently that it's called, the more the synchronization overhead will begin to dominate. A similar problem arises with plain old reference counting, where the CAS operations to maintain the shared reference count itself can limit scalability. We'll consider the reasons why shared writes in general, and CAS in particular, are so expensive in more detail when talking about column C (see below).

Sharing also inhibits optimizations (still B2). When we share a resource, and notably when we share an object in memory, directly or indirectly we have to identify critical sections in the code to acquire and release the resource. As I described in detail in [2] and [3], compilers (and processors and caches, in C2) have to restrict the optimizations they can perform across those barriers, and that can result in a measurable penalty especially in inner loops.

Finally, for similar reasons as those we saw with A3, backoff algorithms and optimistic algorithms that are implemented under the covers inside libraries or by the operating system can result in invisible wasted work (B3).

That concludes our tour of software-related costs... but the story would be grossly incomplete without looking in detail at costs in hardware. Let's focus first once again on the important case of shared memory objects, then broaden the net again to remember that the principle is fully general.

Shared Objects Cause Overhead -- Even In a Race (C1, C2)

Here's an interesting fact, and perhaps surprising: Writing to a shared memory object from multiple processors makes those processors wait for each other (C1) and incurs synchronization overhead (C2), even if you fail to do any locking or other synchronization -- that is, *even in a race*. To see why, let's dig a little.

If we change the Example 1 code to remove the locking, then we have a classic race condition:

```
// Example 2: Evil code for demonstration only
// (closed course, professional driver)

// Thread 1
Modify( x, thisWay );     // race condition

// Thread 2
Modify( x, thatWay );     // race condition

// Thread 3
Modify( x, anotherWay );  // race condition
```

Take a moment to consider this question: Have we really removed all waiting (row 1) and overhead (row 2)? Is the performance of this code now going to be the same as if the threads were using only unshared local objects?

Perhaps surprisingly, on mainstream multicore systems the answer is No. We've definitely removed the obvious and visible synchronization (column A in Table 1). But on mainstream multicore systems, we are still incurring invisible costs (column C) just because we're writing to a shared object through potentially complex levels of cache. Achieving cache coherence across the system inherently requires coordination and overhead, to hide the complexity of the memory hierarchy and maintain the illusion that the data only exists in one place (its expected memory location) when it actually exists in multiple places (the actual memory location plus all the copies in various caches). [4] For more about the memory hierarchy and its costs, see also [5].

Consider how threads 1-3 might execute on the first three cores of a typical modern mainstream multicore processor: an Intel Dunnington, whose simplified block diagram is shown in Figure 1. There are six CPU cores and three levels of cache on the die; each core has 96KB of private level 1 (L1) cache; each pair of processors shares 3MB of L2 cache; and all six processors share up to 16MB of L3 cache in common. (If you already have experience with NUMA architectures, bladed servers, and such, this may look very familiar. All mainstream commodity hardware will be basically NUMA from now on.)

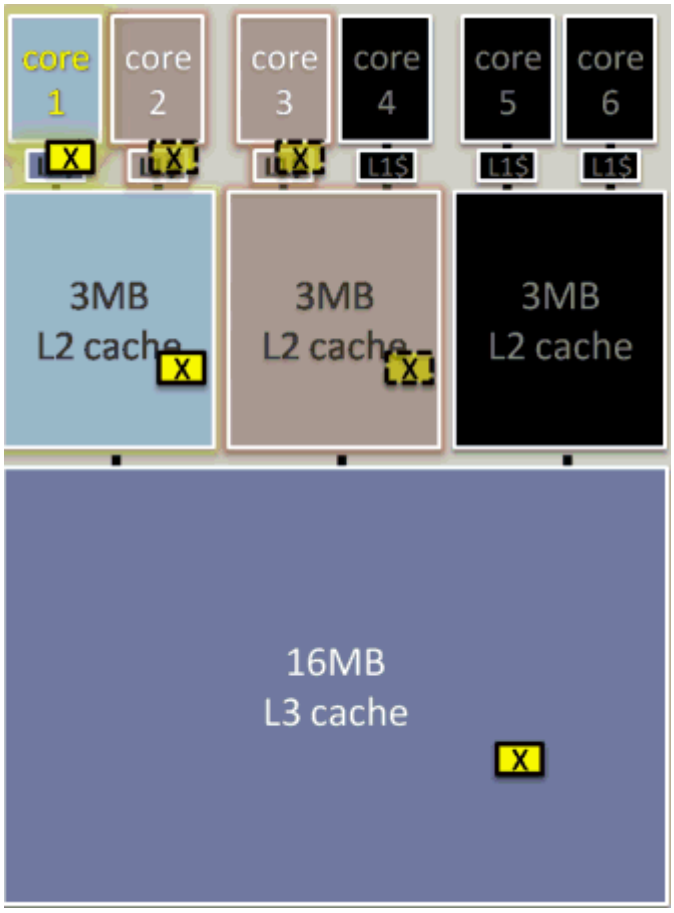

Figure 1: Using x on three cores of an Intel Dunnington processor; core 1 has exclusive access

When a core uses object **x**, the memory containing the accessed parts of **x** must be loaded from RAM up through the pyramid of cache leading to that core. In particular, caches manipulate memory in small contiguous chunks called cache lines, so asking for one byte requires loading the entire cache line containing that byte. In Figure 1, if cores 1, 2, and 3 request access to **x**, the hardware has to load the cache line(s) containing **x** into the processor-wide L3 cache, then into the two L2 caches for cores 1-2 and 3-4, then into the three L1 caches for cores 1, 2, and 3.

For an unshared or read-only shared memory location, it's enough to have each core load the cache line(s) containing that location all the way up into its local cache. So if **x** were immutable and all the accesses were reads, we'd be done and there would be no contention; each core could simply load a copy of the data into its cache and read it independently.

But, in our case, cores 1-3 are modifying **x**, and that's where it gets interesting because to *write* to a memory location a core must additionally have exclusive ownership of the cache

line containing that location. While one core has exclusive use, all other cores trying to write the same memory location must wait and take turns -- that is, they must run serially. Conceptually, it's as if each cache line were protected by a hardware mutex, where only one core can hold the hardware lock on that cache line at a time. Here, let's say core 1 gets to go first: It acquires exclusive use of the cache line, which typically includes following some procedure for invalidating that cache line in at least core 3's L2 cache and possibly also in core 2's and core 3's L1 caches; it performs its update to the appropriate parts of the cache line, then flushes the result out to L2, L3, and RAM for other cores to see; and then, when memory has been sufficiently synchronized, it releases ownership of the cache line. In the meantime, cores 2 and 3 must wait idly (C1), waiting for their turn to reload the cache line with the new value and then perform the same exclusion dance in turn. The deeper the memory hierarchy, the "farther apart" two cores can be and the greater the cost of synchronizing them (C2). [6]

It can be discouraging to see how the overheads of writing to a shared memory location add up. To avoid race conditions, our software should be using mutexes or other synchronization, which means impeding threads' progress as they wait for each other (A1 in Table 1) and incurring the overhead of software synchronization operations (A2 in Table 1). But even in a race, the mere act of sharing a writable memory location causes contention in hardware: We block other cores' and threads' progress by incurring waiting on the memory location (C1), and we incur the cost of synchronization to flush and propagate each write to cores near and far across the rest of the cache hierarchy (C2).

All Sharing Is Bad -- Even of "Unshared" Objects (C1, C2)

It's bad enough that writing to the same shared object inherently throttles scalability, but what's worse is that writing to nearby objects can have the same effect thanks to false sharing.

False sharing (aka "ping-ponging") occurs when two different objects -- shared or not! -- happen to live in the same cache line, and the cache system treats them as a single lump for caching purposes. I've discussed some of the perils of false sharing before, in [5] and Example 4 of [7], but it arises again here: Sadly, the C1 and C2 costs of writing to shared memory can cost you even when you're not actually writing to the same shared object.

Figure 2: Sample memory layout for four objects across 32-byte cache lines

Figure 2 shows a sample memory layout where each row represents a 32-byte cache line; note that objects **obj2** and **obj3**, and part of **obj4**, all reside on the same line. This can cause a real performance problem if **obj2**, **obj3**, and **obj4** are not intended to be shared together, because it is typically impossible for most mainstream multicore hardware to let two different cores simultaneously write to different parts of the same cache line without serializing those writes. For example, a core that wants to update (or in some cases even read from) **obj2** must wait for another core that happens to be concurrently writing to the first part of **obj4**. So these two threads end up competing at least enough to get these writes serialized, even if they're

supposed to be able to run fully concurrently -- if, say, **obj2** and **obj4** are protected by different mutexes or, worse still, because **obj2** and **obj4** are actually supposed to be unshared private objects used only by its respective thread!

Will shuffling objects around or adding padding significantly affect performance? It sure can. Here are two examples: First, in Example 4 of [7], I demonstrated code and performance measurements where merely adding padding increased scalability of a lock-free queue on a 24-core benchmark machine from peaking at 15 cores to peaking at over 20 cores. Second, a few days before I wrote this article, the PLINQ [8] team discovered that moving a single field of a data structure to get it off a popular cache line improved the scalability of one benchmark from 5.2x to 14.8x on a 16-core machine. Incidentally, you usually won't notice such scalability differences on machines with too few cores, because first you have to get enough hardware concurrency for the scalability to matter; on a 4-core machine, the improvement from that same change was only from 3.8x to 3.9x.

The moral(s) of the story? Joe Duffy put it well:

1. Test on bigger machines wherever possible. [*After all, a "big" machine today is probably similar to a "typical" machine your customer will be running tomorrow. -hs*]
2. Validate any performance fixes or experiments on the larger machines, else you will miss cache-related regressions and you might write off potential performance improvements that would have showed a lot of gain had you only run it on a big machine.
3. False sharing is deadly. And even more so on big machines. We knew that, but you'd be surprised how tiny a change led to the cache problems: the movement of a single field [moving a single writable field out of a cache line that otherwise contained only read-only immutable data, so that the cache line went from "read-mostly" to "read-only" and the writer thread stopped interfering with otherwise-independent reader threads]. [9]

Today's performance profiling tools are poor at helping us find these penalties. For example, in all common profiling tools and simple utilities like Windows' Task Manager, your CPU appears busy even while idly waiting for memory due to miss latency and cache line exclusion, so that even when all cores appear to be pegged at 100% the cores may easily be waiting for memory 90% of the time or more. The industry is still working on providing decent tools to let programmers detect these and similar sharing-related performance issues, particularly those due to hardware effects.

Be aware of false sharing, and "if variables A and B are liable to be used by two different threads, keep them on separate cache lines" to avoid penalizing scalability. [5]

Other Shared Memory Costs in Hardware (C1, C2, C3)

Is a shared read less expensive than a shared write? We saw that a shared write is expensive because it needs exclusion (C1), and must be propagated across the memory subsystem and conceptually requires a synchronization after the write (C2). We might expect that reads are cheaper, because they conceptually don't require exclusivity or propagation, and in general writes are more expensive than reads. But the answer depends on the processor: On many processors, shared reads are indeed cheaper, close or equal in cost to an unshared read; on other processors, in some circumstances even a shared read must incur full hardware synchronization overhead, such as that a sequentially consistent read requires a hwsync instruction on current PowerPC processors. [10]

What about the cost CAS compared to a shared write? A CAS also requires exclusion (C1) and conceptually requires synchronization not only after, but also before, the operation (C2), so we would expect a CAS to be at least as expensive, and probably more expensive, than a shared write. That's usually the case, and on some processors the most efficient implementation of CAS is to generate a loop that can actually perform, not just one, but repeated heavyweight synchronizations (C3). [10] (Note that this isn't the same as the "CAS loop" lock-free coding technique I mentioned earlier, which meant writing an explicit loop in your code that performs repeated CAS operations; here I mean that the implementation of the CAS operation itself can have a loop buried inside by which it could be forced to retry and multiple heavy sync instructions each time the software tries to perform a single CAS.)

Other Invisible Costs in Hardware (C1, C2, C3)

We've focused on synchronizing memory operations, but there are other examples of costs in column C.

Consider two threads or processes sharing a spindle. If two threads or processes try to read or write files on the same hard disk drive, or read files on the same DVD-ROM, their operations will have to wait for each other (C1), much as processes do when scheduled on the same core. Additionally, every time the spindle device switches to serve a different user request, it will experience overhead for moving the (usually singular) head across the media and probably some rotational delay for the right part of the media to reach the head (C2), much as processes sharing a core experience will context switch overhead; the amount of time the clients each get access to the hardware will add up to less than 100% due to this overhead.

As mentioned under B2, sharing also inhibits optimizations at the hardware level (still C2). Having critical sections in code prevents processors from reordering instructions, and caches from performing optimizations, that would use memory and other resources more efficiently.

Finally, consider a very different kind of memory sharing: cache residency and eviction under contention (C3). Two threads or processes may share a cache in common at any level of the hierarchy; for example, in Figure 1, they may share only L3 in common (e.g., if running on cores 1 and 6), share both L3 and L2 in common (e.g., if running on cores 5 and 6), or share L3, L2, and even L1 in common (e.g. if running on the same core). Each of the two threads or processes has a given hot working set of data it needs to perform its current work. When the shared cache is big enough for one or both of the contending threads' hot data taken by itself, but not big enough for the total, then running the threads simultaneously means that memory will appear to be slower as they more often evict both their own and each other's data from the shared cache. For example, in Figure 1 and given two threads running memory-intensive inner loops each using 2MB of hot data, the threads can run on cores 1 and 6 without experiencing cache eviction interference, because 4MB fits into L3 cache (assume that the rest of the system's work fits in the remainder). If they run on cores 5 and 6, however, their demand for 4MB will contend for 3MB of L2 cache, which doesn't fit without regular eviction and reloading from L3. If they run on the same core, interleaved with context switches, then on each context switch from one thread to the other we can expect L1 cache to be completely evicted and reloaded, which is a kind of "undo/redo" C3 cost over and above the B2 cost of context switching.

Stepping Back: A Perspective on Symptoms vs. Causes

I frequently see assertions like these, which are basically correct but focus attention on symptoms rather than on the root cause:

- "Taking an exclusive lock kills scalability."
- "Compare-and-swap kills scalability."

Yes, using locks and CAS incurs waiting and overhead expenses, but those all arise from the root cause of having a mutable shared object at all. Once we have a mutable shared object, we have to have some sort of concurrency control to prevent races where two writers corrupt the shared object's bits; instead of trying to get exclusive use of each individual byte of an object, we typically employ a mutex that conveniently stands for the object(s) it protects and synchronize on that instead; and mutexes in turn are ultimately built on CAS operations. Alternatively, we might bypass mutexes and directly use CAS operations in lock-free code. Whichever road we take, we end up in the same place and ultimately have to pay the CAS tax and all the other costs of synchronizing memory. And, as we saw in column C, even in a race we'd still have real contention in hardware and degraded performance -- for no other reason than that we're writing to a shared object.

I wouldn't say "locks kill scalability" or "CAS kills scalability" for the same reason I wouldn't say something like "Stop signs and traffic lights kill throughput." The underlying problem that throttles throughput is having a shared patch of pavement that requires different users to acquire exclusive access; once you have that, you need some way to control the sharing, especially in the presence of attempted concurrent access. [11] So, once again, it's *sharing* that kills scalability, the ability to increase throughput using more concurrent workers -- whether we're sharing "the file," "the object," or "the patch of pavement." Even having to ask for "the crayon" kills scalability for a family's artistic output as we get more and more little artists.

Summary

Sharing is the root cause of all contention, and it is an enemy of scalability. In all of the cases listed at the outset, the act of sharing a resource forces its users to wait idly and to incur overhead costs -- and, often enough, somebody ends up crying. Much the same drama plays out whether it's because kids both want the one blue crayon at the same time, processes steal a shared CPU's cycles or evict each other's data from a shared cache, or threads possibly retry work and are forced to ping-pong the queue's memory as only one processor can have exclusive access at a time. Some of the costs are visible in our high-level code; others are invisible and hidden in lower layers of our software or hardware, though equally significant.

Prefer isolation: Make resources private and unshared, where possible; sometimes duplicating resources is the answer, like providing an extra crayon or a copy of an object or an additional core. Otherwise, prefer immutability: Make shared resources immutable, where possible, so that no concurrency control is required and no contention arises. Finally, use mutable shared state when you can't avoid it, but understand that it's fundamentally an enemy of scalability and minimize touching it in performance-sensitive code including inner loops. Avoid false sharing by making sure that objects that should be usable concurrently by different threads stay on different cache lines.

On Deck

The fact that sharing is the root cause of contention and an enemy of scalability adds impetus to pursuing isolation. Next month, we'll look at the question of isolation in more detail as we turn to the topic of the correct use of threads (hint: threads should try hard to communicate only through messages). Stay tuned.

Acknowledgments

Thanks to Joe Duffy for his comments and data on this topic.

Notes

[1] For details on wait-free vs. lock-free vs. obstruction-free algorithms, see: M. Herlihy. "Obstruction-Free Synchronization: Double-Ended Queues as an Example" (Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003). Available at http://www.cs.brown.edu/~mph/HerlihyLM03/main.pdf.

[2] H. Sutter. "Use Critical Sections (Preferably Locks) to Eliminate Races" (Dr. Dobb's Journal, 32(10), October 2007). Available online at http://www.ddj.com/cpp/201804238.

[3] H. Sutter. "Apply Critical Sections Consistently" (Dr. Dobb's Journal, 32(11), November 2007). Available online at http://www.ddj.com/hpc-high-performance-computing/202401098.

[4] For simplicity I'm going to talk about the way caches tend to operate on mainstream desktop and server hardware, but there's a lot of variety out there.

[5] H. Sutter. "Maximize Locality, Minimize Contention." (Dr. Dobb's Journal, 33(6), June 2008.) Available online at http://www.ddj.com/architect/208200273.

[6] For added fun, imagine the synchronization involved in adding two more levels to the memory hierarchy: having a server containing some system-wide RAM and several "blades," where each blade contains two or four Dunnington chips and additional cache or RAM that is local to the processors on that blade.

[7] H. Sutter "Measuring Parallel Performance: Optimizing a Concurrent Queue" (Dr. Dobb's Journal, 34(1), January 2009). Available online at http://www.ddj.com/cpp/211800538.

[8] J. Duffy and E. Essey. "Parallel LINQ: Running Queries On Multi-Core Processors" (MSDN Magazine, October 2007).

[9] Joe Duffy, personal communication.

[10] P. McKenney and R. Silvera. "Example POWER Implementation for C/C++ Memory Model" (ISO/IEC C++ committee paper JTC1/SC22/WG21/N2745, August 2008). Available online at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2745.html.

[11] I find it useful to compare a mutable shared object to the patch of pavement in an intersection. You don't want timing "races" because they result in program crashes. Sharing requires control: Locks are like stop signs or traffic lights. Wait-free code is often like building a cloverleaf -- it can eliminate some contention by removing direct sharing (sometimes by duplicating resources), at the cost of more elaborate architecture and more

space and pavement (that you won't always have room for), and paying very careful attention to getting the timing right as you automatically merge the cars on and off the freeway (which can be brittle and have occasional disastrous consequences if you don't get it right).

[12] As this article was going to press, Joe Duffy posted a nice overview of some of the issues of reader/writer mutexes: J. Duffy. "Reader/writer locks and their (lack of) applicability to fine-grained synchronization" (Blog post, February 2009). Available online at http://www.bluebytesoftware.com/blog/default,date,2009-02-11.aspx.

Measuring Parallel Performance: Optimizing a Concurrent Queue

When it comes to scalability and concurrency, more is always better.

By Herb Sutter
December 01, 2008
URL:http://drdobbs.com/high-performance-computing/212201163

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

How would you write a fast, internally synchronized queue, one that callers can use without any explicit external locking or other synchronization? Let us count the ways...or four of them, at least, and compare their performance. We'll start with a baseline program and then successively apply three optimization techniques, each time stopping to measure each change's relative performance for queue items of different sizes to see how much each trick really bought us.

All versions of the code we'll consider have one thing in common: They control concurrency using (the equivalent of) two spinlocks, one for each end of the queue. Our goal in refining the code is to allow separate threads calling *Produce* and/or *Consume* to run as concurrently as possible, while also improving scalability to get more total throughput through the queue when we use larger numbers of producer and/or consumer threads.

I'll analyze the throughput of each version of the code by varying three values:

- The number of producer threads. In my test harness, each producer thread is a tight loop that calls *Produce* with no other work.

- The number of consumers. Each consumer thread is a tight loop that calls *Consume* with no other work.
- The size of the queued objects: I'll test queues of objects of two different sizes—"small" and "large" objects that each contain 10 and 100 strings, respectively, of length 26 characters, so that enqueueing or dequeueing an object costs 10 or 100 deep string copies. This actually lets us accomplish two things: a) to measure how performance depends on the size and copying cost of the queued objects; and b) to indirectly simulate the performance effect of how much "other work" the producer and consumer threads may be doing on real-world workloads, which don't usually call *Produce* and *Consume* in tight loops that just throw away the values without using them.

Let's dive in.

Example 1: Baseline

The baseline program is adapted from the code in [1] by adding a spinlock on each end of the queue to allow only one producer and one consumer to update the queue at a time. We will allow some concurrency among producer threads because the producer spinlock doesn't cover the entire body of the *Produce* function, but we will allow only one call to *Consume* to run at a time. This code follows the policy in [1] that only producer threads ever actually change the queue; consumed nodes aren't freed by the consumer, but are lazily cleaned up by the next producer.

The contained objects are held by value:

```
// Example 1: Baseline code
//
template <typename T>
struct LowLockQueue {
private:
  struct Node {
    Node( T val ) : value(val), next(nullptr) { }
    T value;
    atomic<Node*> next;
  };
```

The control variables include pointers to the *first* and *last* nodes in the underlying list, and a pointer to a *divider* element that marks the boundary between the producer and consumer:

```
  Node *first, *last;          // for producer only
  atomic<Node*> divider;               // shared
  atomic<bool> producerLock;   // shared by producers
  atomic<bool> consumerLock;   // shared by consumers
```

Note that, as described in [1], we always maintain at least one dummy node at the front, so the first unconsumed node is actually the one after *divider*. Any nodes before *divider* are already consumed nodes that can be lazily freed by the next producer. The constructor sets up that invariant, and the destructor (in Java or .NET, the *disposer*) simply walks the list to free it:

```
public:
  LowLockQueue() {
    first = divider = last = new Node( T() );
    producerLock = consumerLock = false;
  }
  ~LowLockQueue() {
    while( first != nullptr ) {
      Node* tmp = first;
      first = tmp->next;
      delete tmp;
    }
  }
```

Consume copies the value contained in the first unconsumed node to the caller and updates *divider* to mark that the node has been consumed. Note that the entire body of *Consume* is inside the critical section, which means we get no concurrency among consumers in this version of the code:

```
  bool Consume( T& result ) {
    while( consumerLock.exchange(true) )
      { }                     // acquire exclusivity

    if( divider->next != nullptr ) {  // if queue is nonempty
      result = divider->next->value;  // copy it back to the caller
      divider = divider->next;        // publish that we took an item
      consumerLock = false;           // release exclusivity
      return true;            // and report success
    }

    consumerLock = false;             // release exclusivity
    return false;                     // queue was empty
  }
```

Produce allocates a new node and adds *it* to the tail of the list, then performs the lazy cleanup of consumed nodes at the front of the list, if any. Note that not all of the body of *Produce* is inside the exclusive critical section—many producers can concurrently be allocating their new nodes and copying their new item's value into them, which allows some concurrency among producers:

```
bool Produce( const T& t ) {
  Node* tmp = new Node( t );   // do work off to the side

  while( producerLock.exchange(true) )
    { } // acquire exclusivity

  last->next = tmp;           // publish the new item
  last = last->next;          // (more about this later)

  while( first != divider ) {         // lazy cleanup
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }

  producerLock = false;               // release exclusivity
```

```
  return true;
 }
};
```

How well would you expect this version of the code to scale for different numbers of producers and consumers, or for different kinds of queued objects? Let's find out.

Measuring Example 1

Figure 1 shows the results of running Example 1 using different numbers of producer and consumer threads, and the two sizes of queued objects. The larger the circle, the better the throughput; and each graph has a corner note showing the "max" (largest circle) and "min" (smallest circle) values on that graph. Note the size of each circle represents the relative throughput for that graph only; in this example, even the smallest result on the left-hand graph (where the minimum value is 18,700 objects moved through the queue per second, where each object contains 10 strings) is better than the largest circle on the right-hand graph (11,300 objects per second, where each object contains 100 strings).

This test already illustrates four important properties to consider for parallel throughput. The first key property is overall throughput, or the total amount of work the system is able to accomplish. Here, we can move up to 88,000 small objects or 11,300 objects through the Example 1 queue every second.

The second key property is scalability, or the ability to use more cores to get more work done. For both sizes of queued objects, Example 1 isn't very scalable at all: We get our best throughput at two producers and one consumer, and adding more producers and consumers doesn't get more objects through the queue.

[Click image to view at full size]

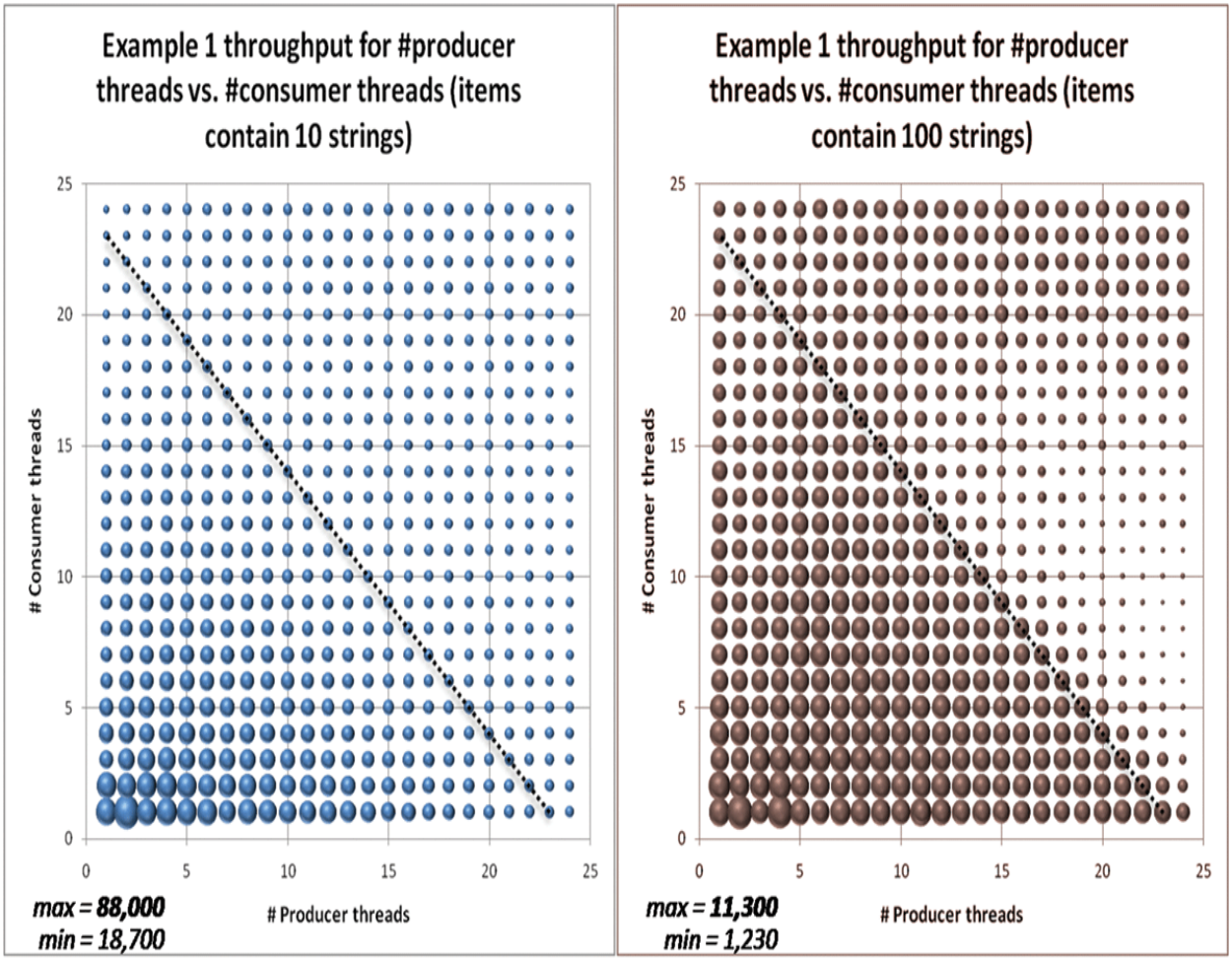

Figure 1: Example 1 throughput (total items through queue per second).

The third key property is contention, or how much different threads interfere with each other by fighting for resources. The small-object case (left graph) shows acute contention, because total throughput drops dramatically when more producers and/or consumers are added—even when those producers and consumers are running on otherwise-idle cores, which unfortunately is a great example of negative scalability. For larger queued objects, the problem is still visible but much less acute; throughput does not drop off as rapidly, at least not until we get to the dashed line.

Incidentally, speaking of that dashed line, I never described the hardware I used to run these tests. If you've been wondering about how many cores were available on my test machine, you can read the answer off the right-hand graph: The dashed line where we see a major performance cliff is where *#consumers* + *#producers* = 24, the number of CPU cores on the test system.

The right-hand graph's diagonal dropoff is a classic artifact of the fourth key property, the cost of oversubscription, or having more CPU-bound work ready to execute than available hardware to execute it. Clearly, we can't scale to more cores than actually exist on a given machine, but there can be a real cost to exceeding that number. In this example, especially when more than half the available threads are producers, oversubscription causes a dramatic increase in churn and contention and loss of overall system throughput. In this test case, a

larger number of consumers had less effect, as illustrated by the "spillover" across the top part of the graph.

Let's see if we can improve total throughput and scalability, while driving down contention and the cost of oversubscription.

Example 2: Shrinking the Consumer Critical Section

Our first optimization involves having each node allocate its contained queue item on the heap and hold it by pointer instead of by value. To experienced parallel programmers, performing an extra heap allocation might seem like a bad idea at first, because heap allocations are notorious scalability busters on many of today's memory allocators. But experience is no substitute for measurement: It turns out that, even on a system with a nonscalable allocator, the benefits often outweigh the advantages. Here, holding the queued object by pointer lets us get greater concurrency and scalability among the consumer threads, because we can take the work of actually copying its value out of the critical section of code that updates the shared data structure.

I'll show only the changed lines. In *Node* itself, of course, we need to hold the values by pointer, and adjust the constructor slightly to match:

```
// Example 2 (diffs from Example 1):
// Moving the copying work out of
// the consumer's critical section
//
  struct Node {
    Node( T* val ) : value(val), next(nullptr) { }
    T* value;
    atomic<Node*> next;
  };

  LowLockQueue() {
    first = divider = last = new Node( nullptr );
    producerLock = consumerLock = false;
    //size = maxSize = 0;
  }
```

The key changes we've enabled are in *Consume*. The key is that we can move the copying of the dequeued object, and the deletion of the value, outside the critical section:

```
bool Consume( T& result ) {
    while( consumerLock.exchange(true) )
      {}       // acquire exclusivity

    if( divider->next != nullptr ) {  // if queue is nonempty

      T* value = divider->next->value;  // take it out
      divider->next->value = nullptr;  // of the Node
      divider = divider->next;  // publish that we took an item
      consumerLock = false;    // release exclusivity
```

```
    result = *value;         // now copy it back to the caller
    delete value;
    return true;             // and report success
  }

  consumerLock = false;              // release exclusivity
  return false;              // queue was empty
}
```

Produce only changes in its first line, where *new Node(t)* becomes *new Node(new T(t)).*

This change should allow better concurrency among consumers. But will it? Before reading on, ask yourself: How would you expect this to affect the magnitude and shape of the performance graphs? How will it affect overall throughput, scalability, contention, and the cost of oversubscription?

Measuring Example 2

Figure 2 shows the results of running Example 2 in the same test harness and on the same hardware. Again, larger circles mean better throughput on the same graph, and pay attention to each graph's "max" (largest circle) and "min" (smallest circle) notes.

Clearly, both throughput and scalability have improved across the board. Our peak throughput is better by a factor of 3 for small objects, and by an order of magnitude for large objects. From this we can deduce that Example 1's throttling of consumers was a severe limiting factor. But the reason peak throughput improved is because we improved scalability: We reach the peak now at larger numbers of producers and consumers. For large objects, we're able to saturate the 24-core machine, and get peak throughput at 15 producer threads and 9 consumer threads.

Unfortunately, for small objects we peak too early and can only usefully harness about half of the available cores before contention is the limiting factor, and adding more producers and consumers, even on otherwise-idle cores, actually accomplishes less total work. For small objects, we don't even reach the dashed line. Even for large objects, contention among consumers still seems to be a limiting factor as shown by the thinness of the top half of the graph.

[Click image to view at full size]

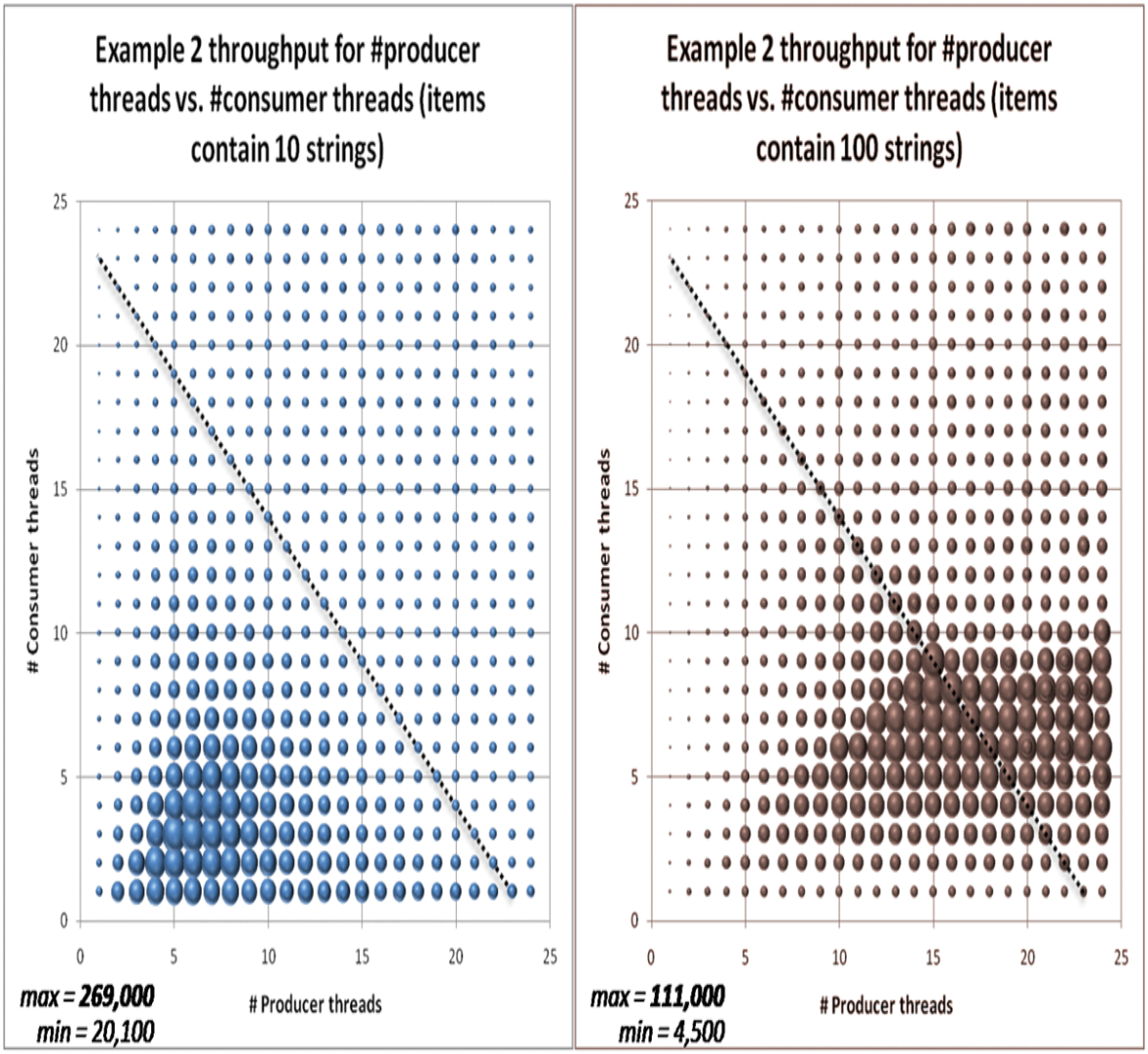

Figure 2: Example 2 throughput (total items through queue per second).

Finally, note that for large objects, not only did we reach the dashed line with throughput still rising, but we spilled across it in the lower fewer-consumers region without losing much throughput. The cost of oversubscription is lessened, at least when adding extra producers; more producers doesn't get more total work done, but they don't negatively impact total work much either. Both contention and oversubscription are still bad, however, when we add extra consumers.

That's a good first step, but let's keep going.

Example 3: Reducing Head Contention

In Examples 1 and 2, the producer was responsible for lazily removing the nodes consumed since the last call to *Produce*. But that's bad for performance for several reasons, notably because it forces a producer to touch both ends of the queue—and every thread that uses the queue, whether producer or consumer, has to touch the queue's head end. Even though a producer and a consumer don't use the same spinlocks and so can run fully concurrently with respect to each other, the fact that they touch the same memory inherently adds invisible

contention, as updates to the memory containing the head nodes have to be propagated to all threads on other cores, not just to consumer threads that naturally have to touch the head end to do their work.

In Example 3, we'll let each consumer be responsible for trimming the node it consumed (which it was touching anyway) and this gives better locality. The first thing we notice is that we can get rid of *divider*—itself a source of contention because it was used by both consumers and producers:

```
// Example 3 (diffs from Example 2):
// Moving cleanup to the consumer
//
  LowLockQueue() {
    first = last = new Node( nullptr ); // no more divider
    producerLock = consumerLock =        false;
  }
```

Consume now doesn't need to deal with *divider*, but must add the work to clean up the previous now-unneeded *first* dummy node when it consumes an item:

```
bool Consume( T& result ) {
  while( consumerLock.exchange(true) )
    { } // acquire exclusivity

  if( first->next != nullptr ) {      // if queue is nonempty
    Node* oldFirst = first;
    first = first->next;
    T* value = first->value;          // take it out
    first->value = nullptr;           // of the Node
    consumerLock = false;             // release exclusivity

    result = *value;                  // now copy it back
    delete value;                     // and clean up
    delete oldFirst;                  // both allocations
    return true;                      // and report success
  }

  consumerLock = false;               // release exclusivity
  return false;               // queue was empty
}
```

Next, *Produce* becomes simpler because we can eliminate the lazy cleanup code. However, just eliminating that code leads to a very subtle pitfall because one existing line also has to change. Can you see why?

```
  bool Produce( const T& t ) {
    Node* tmp = new Node( t );        // do work off to the side

    while( producerLock.exchange(true) )
      { }                             // acquire exclusivity
```

```
    last->next = tmp;                 // A: publish the new item
    last = tmp;                // B: not "last->next"

    producerLock = false;             // release exclusivity
    return true;
  }
```

Changing Responsibilities Can Introduce Bugs

Note that line *B* used to be *last = last->next;*. That was always slightly inefficient because it needlessly reread *last* (a holdover from the original code written by someone else). Now, if left unchanged, it becomes something much worse: a small race window. Now that there's no *divider* and consumers clean up consumed nodes, the way consumers know there's an item available to be consumed is to check *first->next;* if it's not null, it's okay to go ahead and consume a node—and delete what used to be the first one because that node is no longer needed. The trouble arises when a sequence like the following occurs:

- Initially: queue is empty, *first == last*
- The producer (from Example 2 code, without the Example 3 correction): l*ast->next = tmp; // A: publish*
- The consumer performs an entire call to *Consume* the just-published node, including deleting the now unnecessary previous first node before it
- Then the producer dereferences *last*

  last = last->next; // B: update last
  // oops: accesses freed memory.

The key is that the act of publishing the new node (line *A)* not only advertises that the new node is ready to be consumed, but also implicitly transfers ownership of the preceding node to the consumer. Hence, line *B* must not dereference *last* again, but should just assign from *tmp* directly.

"But," someone might object, "will this interleaving really happen? After all, *A*-*B* is a very small window for a call to *Consume* to fit into." True, it won't happen often. Based on experience, however, I can report that under heavy stress on a multicore system, this tends to fail once for every few tens of millions of items moving through the queue. This was the only race I wrote (that I know of) when putting these examples together, and it was a real pain to reproduce and diagnose.

Moral: When you change responsibilities for cleanup, code that used to be innocuous can suddenly turn into a subtle race window.

Measuring Example 3

But back to the main event: How well does moving the cleanup responsibility and reducing contention on the head of the queue really help? Again, before seeing my results, consider how much, and why, you think this is likely to affect throughput, scalability, contention, and the oversubscription penalty.

Figure 3 shows the Example 3 performance results. The effects are mainly on the left-hand small object graph, with only incremental improvements for large objects. For small objects, peak throughput has improved by nearly another factor of two, and we've again improved scalability and actually get close to reaching the dashed line, which represents our capacity for getting more work done using more cores. There is some dropoff due to contention as we exceed about 20 active threads (e.g., 12 producers and 8 consumers), and for the first time we can actually see the oversubscription wall on the left-hand graph beyond 24 threads. Although we'd like to scale that wall, right now we're happy to just be able to approach it in the first place!

[Click image to view at full size]

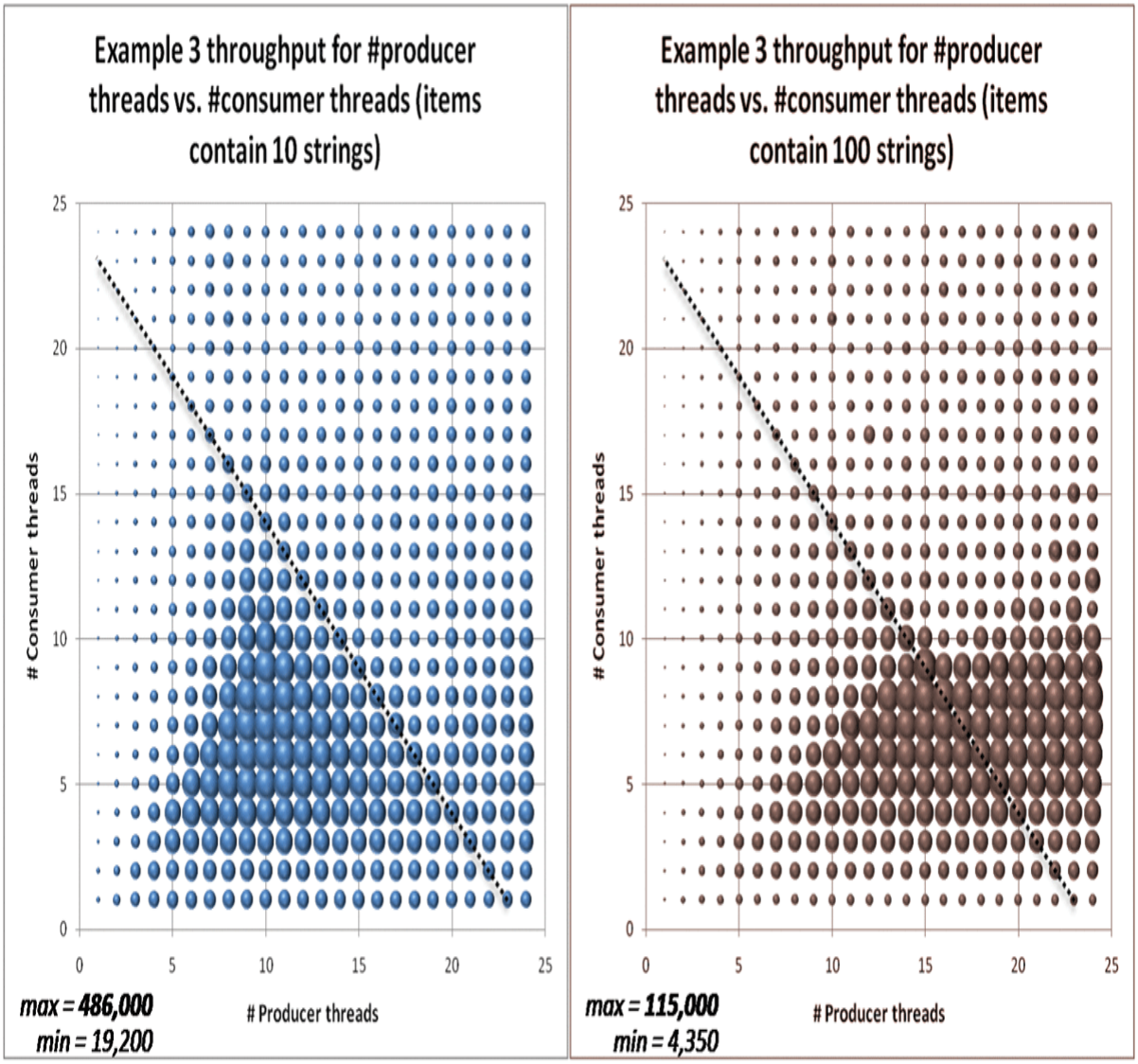

Figure 3: Example 3 throughput (total items through queue per second).

Example 4: Padding To Avoid False Sharing

The changes for our final Example 4 are extremely simple compared to the changes in Examples 2 and 3. In fact, we're not going to make any changes to the program logic at all:

We're simply going to follow the advice that "if variables *A* and *B* are...liable to be used by two different threads, keep them on separate cache lines" to avoid false sharing or "ping-ponging," which limits scalability [2]. In this case, we want to add padding to ensure that different nodes (notably the *first* and *last* nodes), the *first* and *last* pointers to those nodes, and the *producerLock* and *consumerLock* variables are all on different cache lines:

```
struct Node {
  Node( T* val ) :        value(val), next(nullptr) { }
  T* value;
  atomic<Node*> next;
  char pad[CACHE_LINE_SIZE - sizeof(T*) - sizeof(atomic<Node*>)];
};

char pad0[CACHE_LINE_SIZE];
Node* first;                 // for one consumer at a time
char pad1[CACHE_LINE_SIZE - sizeof(Node*)];
atomic<bool> consumerLock;   // shared among consumers
char pad2[CACHE_LINE_SIZE - sizeof(atomic<bool>)];
Node* last;                          // for one producer at a time
char pad3[CACHE_LINE_SIZE - sizeof(Node*)];
atomic<bool> producerLock    // shared among producers
char pad4[CACHE_LINE_SIZE - sizeof(atomic<bool>)];
```

That seems like a pretty trivial change. Is it really worth its own section? Again, ask yourself: Is this likely to have any effect on throughput, scalability, contention, or oversubscription? If so, how much, and why?

Measuring Example 4

Figure 4 shows the Example 4 performance results. Again, the effects are less noticeable for large objects, where the cost of copying dominates and was already moved out of the critical section in earlier code revisions.

For small objects, peak throughput has improved by another 20 percent over Example 3 by extending scalability right to the dashed line representing the total number of hardware cores. As a bonus, we've also reduced the contention dropoff to the dashed line and beyond, smoothing the transition into oversubscription. As in every case, the sparse top half of each graph shows that larger numbers of consumers are still causing significant contention, but we've pushed the consumer contention effect out (upwards) somewhat for both small and large objects.

[Click image to view at full size]

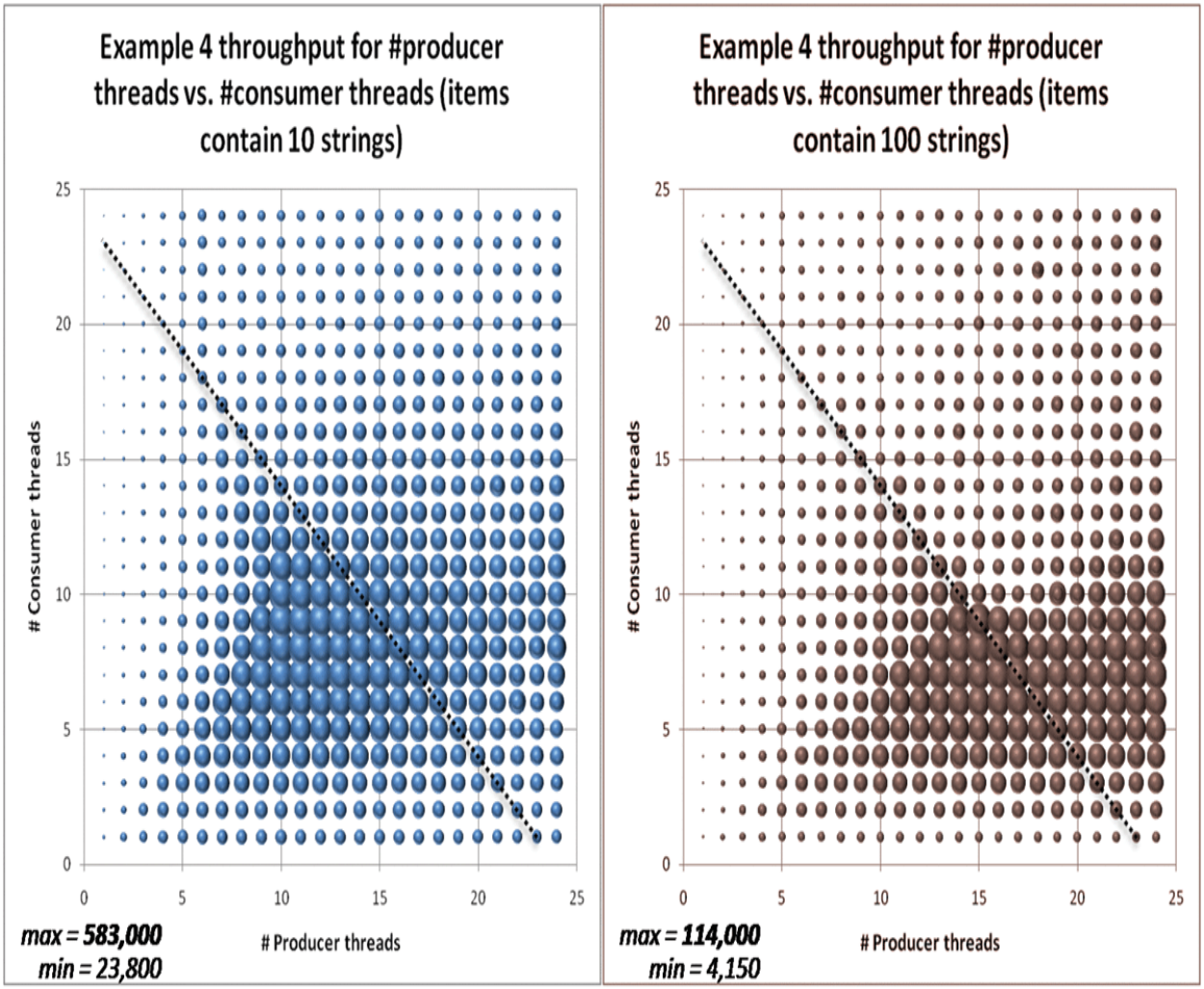

Figure 4: Example 4 throughput (total items through queue per second).

Direct Comparisons

Graphs like Figures 1-4 are useful to get a sense of the shape of our code's scalability for varying amounts of different kinds of work, and allow some visual comparison among different versions of the code. But let's consider a more direct comparison by graphing Examples 1-4 together on a line graph. Because all of the graphs so far show that our code tends to favor having more producers than consumers, I've chosen to analyze the slice of data along the line *#producers* = 2 * *#consumers* (i.e., 2/3 of threads are producers, 1/3 are consumers), which goes through the fat high-throughput part of each of the previous graphs to allow for fairer comparison.

Figure 5 shows each example's throughput along this selected slice for small queue items, with automatically generated polynomial trend lines to connect the dots. Notice that each vertical slice contains several dots of the same example; this is because I ran each test several times, and the vertical spread in dots shows performance variability in repeated executions of the same test.

First, consider the left-hand graph in Figure 5, which shows the straightforward throughput numbers: As we now know to expect, Example 1 clearly has the worst performance, and each

of the other examples successively improve both the peak throughput (the magnitude of each curve's high point) and scalability (how far to the right each peak is). You can visually see how each successive change we made removed an invisible barrier and let the curve continue growing a bit further. The downslope on the right-hand side of each curve shows the throughput-slowing effects of contention. Oversubscription would show up on this graph as a discontinuity as we pass 24 threads; here, with small objects, it's not very visible except for Example 3, as we also saw earlier.

[Click image to view at full size]

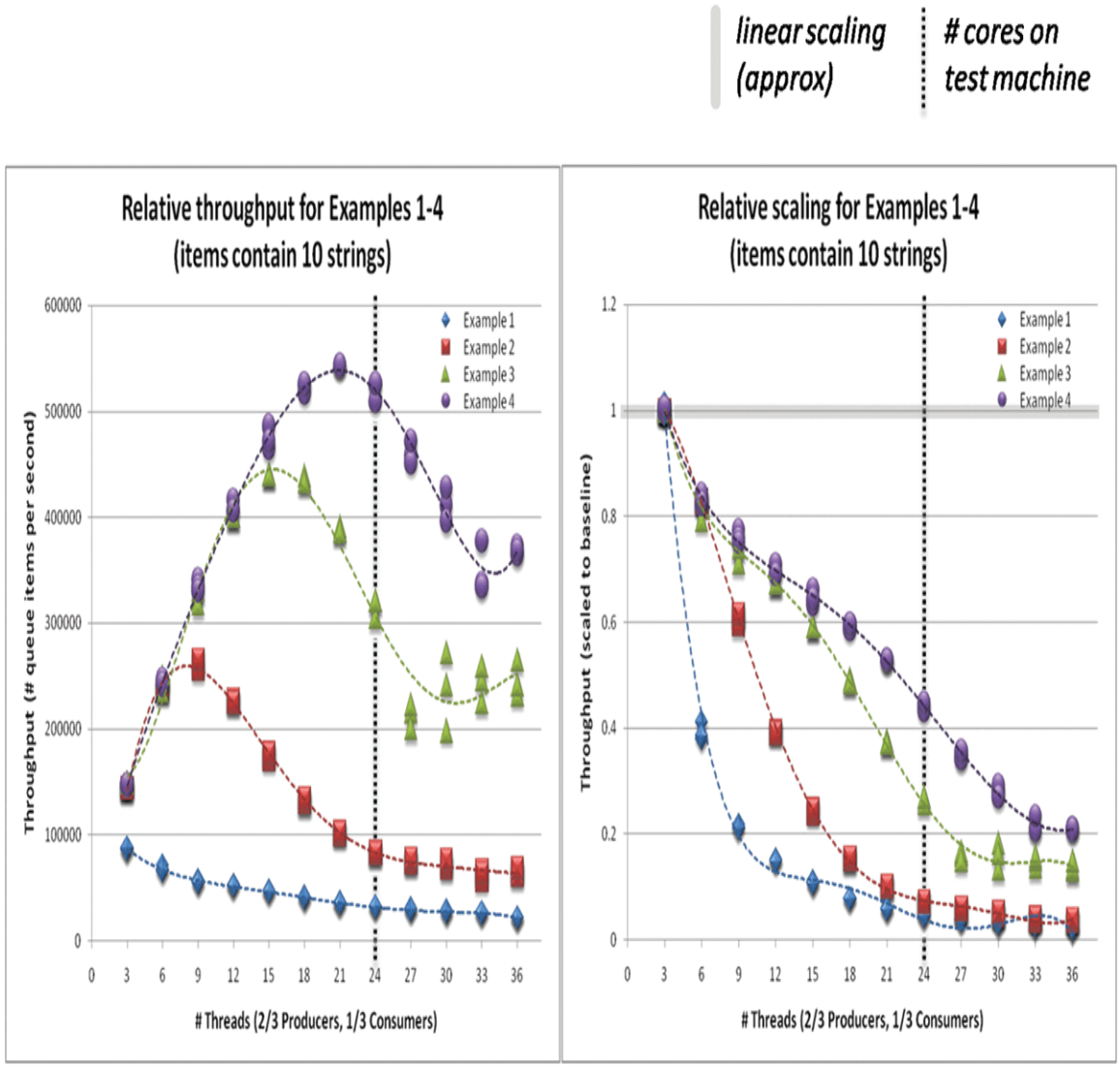

Figure 5: Comparing throughput and scaling for Examples 1-4 along the line #Producers = 2 * #Consumers (small queue item size).

But there's another useful way to scale the graph. Still in Figure 5, the right-hand graph shows exactly the same data as the left-hand graph, only we scale it by dividing each line's data by that same line's initial value and the number of threads. This is a more direct way to measure the relative scalability of each version of the code—the ideal is the grey horizontal bar, which shows linear scalability, and we want our curve to float as high and as close to that as possible. As before, Example 1 is the worst, because it quickly dives to awful depths.

Example 4 is the best: It's still delivering over 50 percent scalability until just before the machine is fully saturated—getting nearly 4 times the performance using 7 times the cores (21 versus the baseline 3) may not be linear scaling, but it's still fairly decent scaling and nothing to sneeze at.

Now consider Figure 6, which shows each example's throughput along the same slice, but this time for large queue items. Again, the dotted curves are automatically generated polynomial trendlines to connect the dots; we'll come back to these in a moment.

[Click image to view at full size]

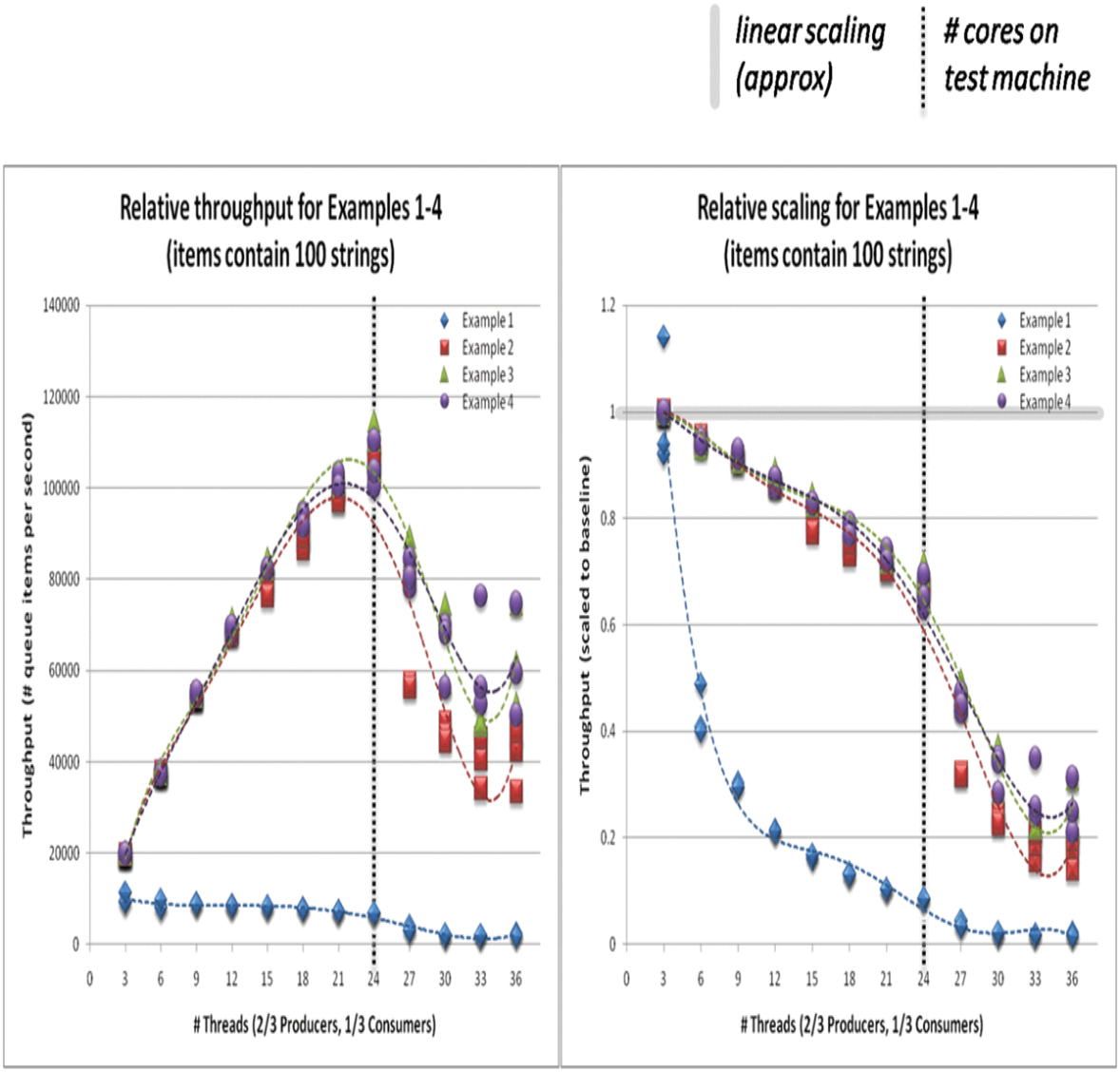

Figure 6: Comparing throughput and scaling for Examples 1-4 along the line #Producers = 2 * #Consumers (large queue item size).

What can we conclude from Figure 6? There's much less difference among Examples 2 through 4 for larger queued items where the cost of copying dominates—or, equivalently, when the producer and consumer threads are doing more work than just madly inserting and removing queue items as quickly as possible and throwing them away. Most of the benefit

came from just applying the change in Example 2 to hold the queued item by pointer in order to gain more concurrency in the consumer, thereby removing what evidently was a consumer bottleneck. On the right-hand graph it's evident that all examples scale about equally well for large objects, and all are much better than they were in Figure 5 for small objects—70 percent scaling until right before we achieve machine saturation. And those pretty polynomial trendlines help us visualize what's going on better...

Or do they?

A Word About Trendlines

Beware automatic trendlines: They are often useful, but they can also lie.

Look closely again at Figure 6: The trendlines actually obscure what's really going on by creating a false visual smoothness and continuity that our human eyes are all too willing to believe. Try to imagine the graph without any trendlines, and ask yourself: Are these really the trendlines you would draw? Yes, they're close to the data up to about 21 threads. But they under-represent throughput and scalability at the machine-saturation point of 24 threads, and then don't account for the sudden drop and increased variability beyond saturation.

One of the weaknesses of curve-fitting trendlines is that they expect all of the data to fit one consistent pattern, but that is often only true within certain regions. Trendlines don't deal well with discontinuities, where we shift from one region to another one with fundamentally different characteristics and assumptions. Clearly, trying to use fewer cores or more cores than exist on the actual hardware are two different regions, and what the data actually correctly shows is that there's a jump between these regions).

If we replace the automatically generated trendlines with hand-fitted ones, a very different and much truer picture emerges, as shown in Figure 7.

[Click image to view at full size]

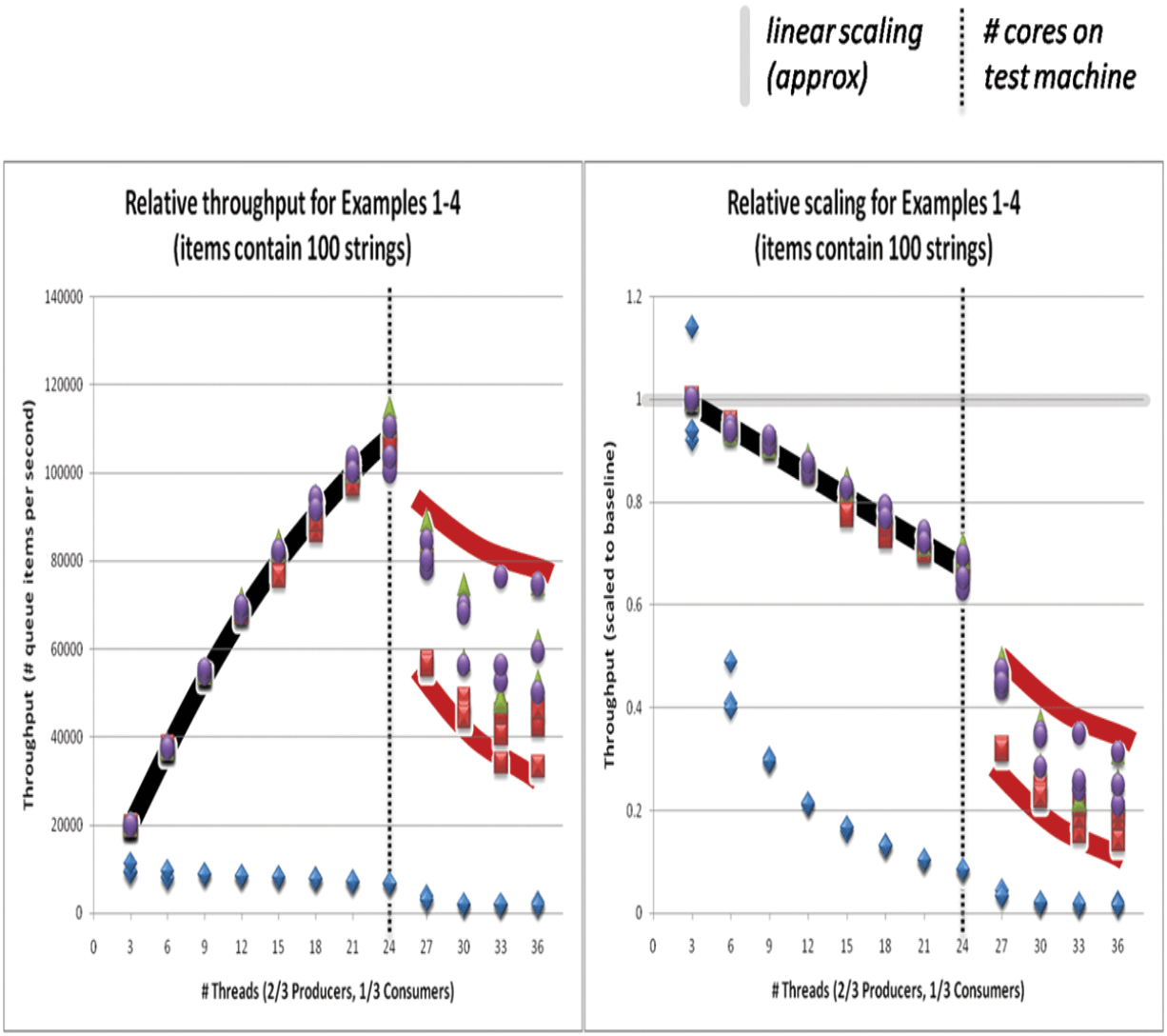

Figure 7: Another view of Figure 6, without misleading automatic trendlines.

Now the discontinuity is glaring and clear: On the left-hand graph, we move from a region of consistent and tight throughput increase through a wall, beyond which we experience a sudden drop and find ourselves in a new region where throughput is both dramatically lower and far less consistent and predictable—for example, multiple runs of the same Example 4 code at >24 threads shows dramatically variable results. On the right-hand graph, we move from a region of good and linearly decreasing scalability through a sudden drop, beyond which lies a new region where scalability too is both much lower and more variable.

What Have We Learned?

To improve scalability, we need to minimize contention:

- Reduce the size of critical sections so that we can get less contention and more concurrency among client threads.
- Reduce sharing of the same data by isolating threads to use different parts of a data structure. In our case, we moved the responsibility for cleanup from the producer to the consumer so that consumers touch only the head and producers touch only the tail.

- Reduce false sharing of different data on the same cache line, by adding padding to ensure that two separate variables that should be able to be used concurrently by different threads are also far enough apart in memory.

To understand our code's scalability, we need to know what to measure and what to look for in the results:

- Identify the key different kinds of work (here, producer threads and consumer threads), and then use stress tests to measure the impact of having different quantities and combinations of these in our workload.
- Identify the different kinds of data (here, representative "small" and "large" queue items), and then vary those to measure their impact.
- Measure total throughput, or items handled per unit time.
- Look for scalability, or the change in throughput as we add more threads. Does using more threads do more total work? Why or why not? In what directions, and for what combinations of workloads?
- Look for contention, or the interference between multiple threads trying to do work concurrently.
- Watch for the cost of oversubscription, and eliminate it either algorithmically or by limiting the actual amount of concurrency to avoid it altogether.
- Beware of overreliance on automated trendlines. Apply them only after first examining the raw data.

Be a scientist: Gather data. Analyze it. Especially when it comes to parallelism and scalability, there's just no substitute for the advice to measure, measure, measure, and understand what the results mean. Putting together test harnesses and generating and analyzing numbers is work, but the work will reward you with a priceless understanding of how your code actually runs, especially on parallel hardware—an understanding you will never gain from just reading the code or in any other way. And then, at the end, you will ship high-quality parallel code not because you think it's fast enough, but because you know under what circumstances it is and isn't (there will always be an "isn't"), and why.

Notes

[1] H. Sutter. "Writing Lock-Free Code: A Corrected Queue" (*DDJ*, October 2008).

[2] H. Sutter. "Maximize Locality, Minimize Contention" (*DDJ*, September 2008). www.ddj.com/architect/208200273.

Writing a Generalized Concurrent Queue

Herb tackles the general problem of supporting multiple producers and multiple consumers with as much concurrency as possible.

By Herb Sutter
October 29, 2008
URL:http://drdobbs.com/high-performance-computing/211601363

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

Last month [1], I showed code for a lock-free queue that supported the limited case of exactly two threads—one producer, and one consumer. That's useful, but maybe not as exciting now that our first rush of lock-free coding glee has worn off. This month, let's tackle the general problem of supporting multiple producers and multiple consumers with as much concurrency as possible. The code in this article uses four main design techniques:

First, we'll use (the equivalent of) two locks: One for the head end of the queue to regulate concurrent consumers, and one for the tail to regulate concurrent producers. We'll use ordered atomic variables (C++0x *atomic<>*, Java/.NET *volatile*) directly instead of prefabricated mutexes, but functionally we're still writing spinlocks; we're just writing them by hand. Although this means it's not a purely "lock-free" or nonblocking algorithm, it's still quite concurrent because we'll arrange the code to still let multiple consumers and multiple producers make progress at the same time by arranging to do as much work as possible outside the small critical code region that updates the head and tail, respectively.

Second, we'll have the nodes allocate the contained *T* object on the heap and hold it by pointer instead of by value. [2] To experienced parallel programmers this might seem like a bad idea at first, because it means that when we allocate each node we'll also need to perform an extra heap allocation, and heap allocations are notorious scalability busters on many of today's memory allocators. It turns out that, even on a system with a nonscalable allocator, the benefits typically outweigh the advantages: Holding the *T* object by pointer let us get greater concurrency and scalability among the consumer threads, because we can take the work of actually copying the *T* value out of the critical section of code that updates the shared data structure.

Third, we don't want to have the producer be responsible for lazily removing the nodes consumed since the last call to *Produce*, because this is bad for performance: It adds contention on the queue's head end, and it needlessly delays reclaiming consumed nodes. Instead, we'll let each consumer be responsible for trimming the node it consumed, which it was touching anyway and so gives better locality.

Fourth, we want to follow the advice that "if variables *A* and *B* are not protected by the same mutex and are liable to be used by two different threads, keep them on separate cache lines" to avoid false sharing or "ping-ponging" which limits scalability. [3] In this case, we want to

add padding to ensure that different nodes (notably the first and last nodes), the *first* and *last* pointers into the list, and the *producerLock* and *consumerLock* variables are all on different cache lines.

A Two-Lock Multiproducer/Consumer Queue

The queue data structure itself is a singly linked list. To make the code simpler for the empty-queue boundary case, the list always contains a dummy node at the head, and so the *first* element logically in the queue is the one contained in the node after first. Figure 1 shows the layout of an empty queue, and Figure 2 shows a queue that holds some objects.

Each node holds the *T* object by pointer and adds padding:

```
template <typename T>
struct LowLockQueue {
private:
  struct Node {
    Node( T* val ) : value(val), next(nullptr) { }
    T* value;
    atomic<Node*> next;
    char pad[CACHE_LINE_SIZE - sizeof(T*)- sizeof(atomic<Node*>)];
  };
```

Like any shared variable, the *next* pointer needs to be protected by a mutex or, as here, declared as an ordered atomic type (C++0x *atomic<>* or Java/.NET *volatile*). The padding here is to ensure that two *Node* objects won't occupy the same cache line; in particular, in a nonempty queue, having the first and last nodes in the same cache line would penalize performance by causing invisible contention between the producer and consumer. Note that the amount of padding shown here and later on errs on the side of being too conservative: Each *Node* will be allocated on the heap, and a heap allocator may add its own extra overhead to each allocated block for alignment and housekeeping information, which effectively adds extra padding. If so, and if we know how much that will be, we could reduce our internal padding proportionately to make a heap-allocated *Node* exactly fill one cache line.

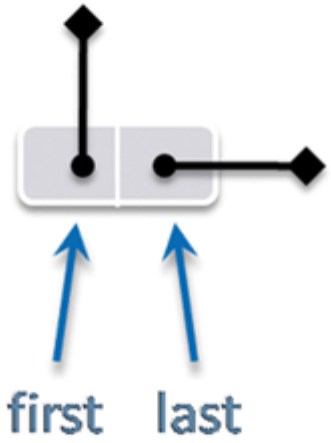

Figure 1: Structure of an empty queue.

[Click image to view at full size]

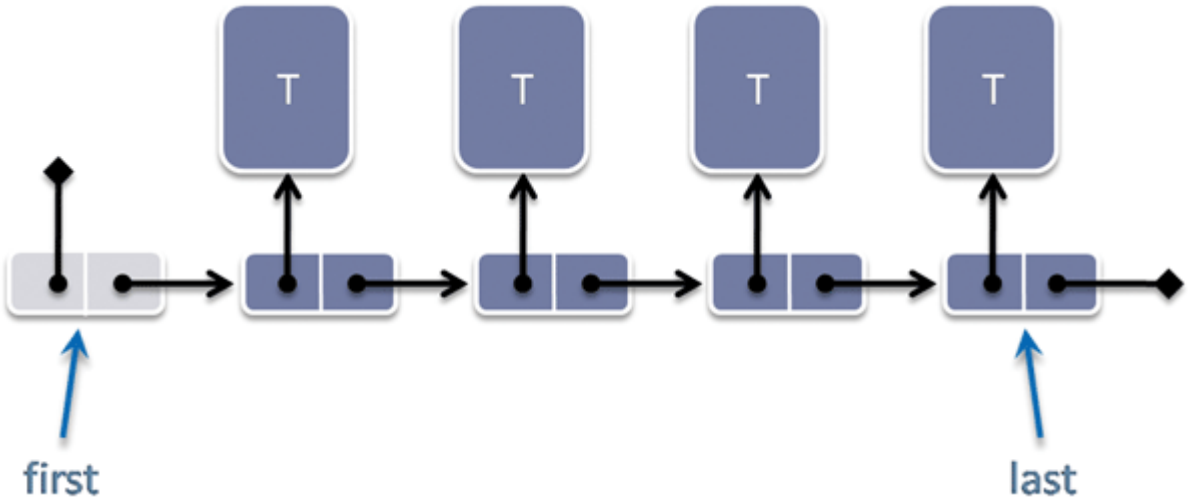

Figure 2: A queue containing four T objectsempty queue.

Next, here are our queue control variables:

```
    char pad0[CACHE_LINE_SIZE];

  // for one consumer at a time
  Node* first;

  char pad1[CACHE_LINE_SIZE
           - sizeof(Node*)];

  // shared among consumers
  atomic<bool> consumerLock;

  char pad2[CACHE_LINE_SIZE
           - sizeof(atomic<bool>)];

  // for one producer at a time
  Node* last;

  char pad3[CACHE_LINE_SIZE
           - sizeof(Node*)];

  // shared among producers
  atomic<bool> producerLock;

  char pad4[CACHE_LINE_SIZE
           - sizeof(atomic<bool>)];
```

Again, we add padding to make sure hat data used by different threads stay physically separate in memory and cache. Clearly, we want the consumer-end data and the producer-end data to be on separate cache lines; but even though only one producer and one consumer will be active at a time, we want to keep the locking variable separate so that waiting consumers spinning on *consumerLock* won't contend on the cache line that contains *first* which the active consumer is updating, and that waiting producers spinning on *producerLock* won't slow down the active producer who is updating *last*.

The constructor just sets up the initial empty state, and the destructor (in .NET or Java, this would be the disposer) just walks the internal list and tears it down:

```
public:
  LowLockQueue() {
    first = last = new Node( nullptr );
    producerLock = consumerLock = false;
  }
  ~LowLockQueue() {
    while( first != nullptr ) {              // release the list
      Node* tmp = first;
      first = tmp->next;
      delete tmp->value;             // no-op if null
      delete tmp;
    }
```

```
}
```

Produce

Now let's look at the first of the two key methods: *Produce*. The goal is to allow multiple producers, and to let them run as concurrently as possible:

```
void Produce( const T& t ) {
  Node* tmp = new Node( new T(t) );
  while( producerLock.exchange(true) )
    { }      // acquire exclusivity
  last->next = tmp;                         // publish to consumers
  last = tmp;                         // swing last forward
  producerLock = false;               // release exclusivity
}
```

First, we want to do as much work as possible outside the critical section of code that actually updates the queue. In this case, we can do all of the allocation and construction of the new node and its value concurrently with any number of other producers and consumers.

Second, we "commit" the change by getting exclusive access to the tail of the queue. The *while* loop keeps trying to set the *producerLock* to true until the old value was false because while the old value was true, it means someone else already has exclusivity. The way to read this *while* loop is, "until I get to be the one to change *producerLock* from false to true," which means that this thread has acquired exclusivity. Then we can update *last->next* and *last* itself, which are two separate writes and cannot be done as a single atomic operation on most processors without some sort of lock. Finally, we release exclusivity on the tail of the queue by setting *producerLock* to false.

Consume

Likewise, we want to support any number of threads calling *Consume*, and let them run as concurrently as possible. First, we get exclusivity, this time on the head end of the queue:

```
bool Consume( T& result ) {
  while( consumerLock.exchange(true) )
    { }       // acquire exclusivity
```

Next, we read the head node's *next* pointer. If it's not null, we need to take out the first value but we want to do as little work as possible here inside the exclusive critical section:

```
  Node* theFirst = first;
  Node* theNext = first->next;
  if( theNext != nullptr ) {               // if queue is nonempty
    T* val = theNext->value;               // take it out
```

```
      theNext->value = nullptr;              // of the Node
      first = theNext;                       // swing first forward
      consumerLock = false;           // release exclusivity
```

Now we're done touching the list, and other consumers can make progress while we do the remaining copying and cleanup work off to the side:

```
      result = *val;          // now copy it back
      delete val;             // clean up the value
      delete theFirst;                // and the old dummy
      return true;                    // and report success
    }
```

Otherwise, if *theNext* was null, the list was empty and we can immediately release exclusion and return that status:

```
    consumerLock = false;                    // release exclusivity
    return false;      // report queue was empty
  }
};
```

Fully Nonblocking Multiproducer/Consumer Queues

The above code still uses two locks, albeit sparingly. How might we eliminate the producer and consumer locks for a fully nonblocking queue implementation that allows multiple concurrent producers and consumers? If that intrigues you, and you're up for some down-and-dirty details, here are two key papers you'll be interested in reading.

In 1996, Michael and Scott published a paper that presented two alternatives for writing an internally synchronized queue. [4] One alternative really is nonblocking; the other uses a producer lock and a consumer lock, much like the examples in this article. In 2003, Herlihy, Luchango and Moir pointed out scalability limitations in Michael and Scott's approach, and presented their own obstruction-free queue implementation. [5]

Both of these papers featured approaches that require a double-width compare-and-swap operations (also known as "DCAS") that can treat a pointer plus an integer counter together as a single atomic unit. That is problematic because not all platforms have a DCAS operation, especially mainstream processors in 64-bit mode which would essentially require a 128-bit CAS. [6] One also requires a special free list allocator to work properly.

Coming Up

We applied four techniques:

1. Having two locks, one for each end of the queue.
2. Allocating objects on the heap to let us make consumers more concurrent.

3. Having consumers remove consumed nodes one at a time for better locality, less contention at the head, and more immediate cleanup than having producers lazily clean up consumed nodes.
4. Adding padding to keep data used by different threads on different cache lines, avoiding memory performance penalties due to false sharing or "ping-pong."

But just how much did each of those help, and how much did each help depending on the size of the queued objects? Next month, I'll break down the four techniques by analyzing the successive performance impact of each of these techniques with some pretty graphs. Stay tuned.

Notes

[1] H. Sutter. "Lock-Free Code: A False Sense of Security" (*DDJ*, June 2008). Available online at http://ddj.com/architect/208200273.

[2] Note that this happens naturally in Java and .NET for reference types, which are always held indirectly via a pointer (which is called an object reference in those environments).

[3] H. Sutter. "Maximize Locality, Minimize Contention" (*DDJ*, September 2008). Available online at http://ddj.com/architect/208200273.

[4] M. Michael and M. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" (*Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 1996)

[5] M. Herlihy, V. Luchango and M. Moir. "Obstruction-Free Synchronization: Double-Ended Queues As an Example" (*Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003).

[6] You could try to make it work in 64 bits via heroic efforts to steal from the 64-bit address space, taking ruthless advantage of the knowledge that on mainstream systems today the operating system usually doesn't actually use all 64 bits of addresses and might not notice if you use a few or even a dozen for your own ends. However, that's inherently brittle and nonportable, and a lot of other people (including probably your OS's developers) have had the same idea and tried to grab those bits too in the frenzied 64-bit land rush already in progress. In reality, you generally only get to play this kind of trick if you're the operating system or its close friend.

Writing Lock-Free Code: A Corrected Queue

Herb continues his exploration of lock-free code--this time focusing on creating a lock-free queue.

By Herb Sutter
September 29, 2008
URL:http://drdobbs.com/high-performance-computing/210604448

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

As we saw last month [1], lock-free coding is hard even for experts. There, I dissected a published lock-free queue implementation [2] and examined why the code was quite broken. This month, let's see how to do it right.

Lock-Free Fundamentals

When writing lock-free code, always keep these essentials well in mind:

- Key concepts. Think in transactions. Know who owns what data.
- Key tool. The ordered atomic variable.

When writing a lock-free data structure, "to think in transactions" means to make sure that each operation on the data structure is atomic, all-or-nothing with respect to other concurrent operations on that same data. The typical coding pattern to use is to do work off to the side, then "publish" each change to the shared data with a single atomic write or compare-and-swap. [3] Be sure that concurrent writers don't interfere with each other or with concurrent readers, and pay special attention to any operations that delete or remove data that a concurrent operation might still be using.

Be highly aware of who owns what data at any given time; mistakes mean races where two threads think they can proceed with conflicting work. You know who owns a given piece of shared data right now by looking at the value of the ordered atomic variable that says who it is. To hand off ownership of some data to another thread, do it at the end of a transaction with a single atomic operation that means "now it's your's."

An ordered atomic variable is a "lock-free-safe" variable with the following properties that make it safe to read and write across threads without any explicit locking:

- Atomicity. Each individual read and write is guaranteed to be atomic with respect to all other reads and writes of that variable. The variables typically fit into the machine's native word size, and so are usually pointers (C++), object references (Java, .NET), or integers.
- Order. Each read and write is guaranteed to be executed in source code order. Compilers, CPUs, and caches will respect it and not try to optimize these operations the way they routinely distort reads and writes of ordinary variables.
- Compare-and-swap (CAS) [4]. There is a special operation you can call using a syntax like *variable.compare_exchange(expectedValue, newValue)* that does the following as an atomic operation: If *variable* currently has the value *expectedValue*, it sets the value to *newValue* and returns true; else returns false. A common use is

if(variable.compare_exchange(x,y)), which you should get in the habit of reading as, "if I'm the one who gets to change variable from *x* to *y*."

Ordered atomic variables are spelled in different ways on popular platforms and environments. For example:

- *volatile* in C#/.NET, as in *volatile int*.
- *volatile* or ** Atomic** in Java, as in *volatile int*, *AtomicInteger*.
- *atomic<T>* in C++0x, the forthcoming ISO C++ Standard, as in *atomic<int>*.

In the code that follows, I'm going to highlight the key reads and writes of such a variable; these variables should leap out of the screen at you, and you should get used to being very aware of every time you touch one.

If you don't yet have ordered atomic variables yet on your language and platform, you can emulate them by using ordinary but aligned variables whose reads and writes are guaranteed to be naturally atomic, and enforce ordering by using either platform-specific ordered API calls (such as Win32's *InterlockedCompareExchange* for compare-and-swap) or platform-specific explicit memory fences/barriers (for example, Linux *mb*).

A Corrected One-Producer, One-Consumer Lock-Free Queue

Now let's tackle the lock-free queue using our essential tools. In this first take, to allow easier comparison with the original code in [2], I'll stay fairly close to the original design and implementation, including that I'll continue to make the same simplifying assumption that there is exactly one Consumer thread and one Producer thread, so that we can easily arrange for them to always work in different parts of the underlying linked list. In Figure 1, the first "unconsumed" item is the one after the *divider*. The consumer increments *divider* to say it has consumed an item. The producer increments *last* to say it has produced an item, and also lazily cleans up consumed items before the *divider*.

[Click image to view at full size]

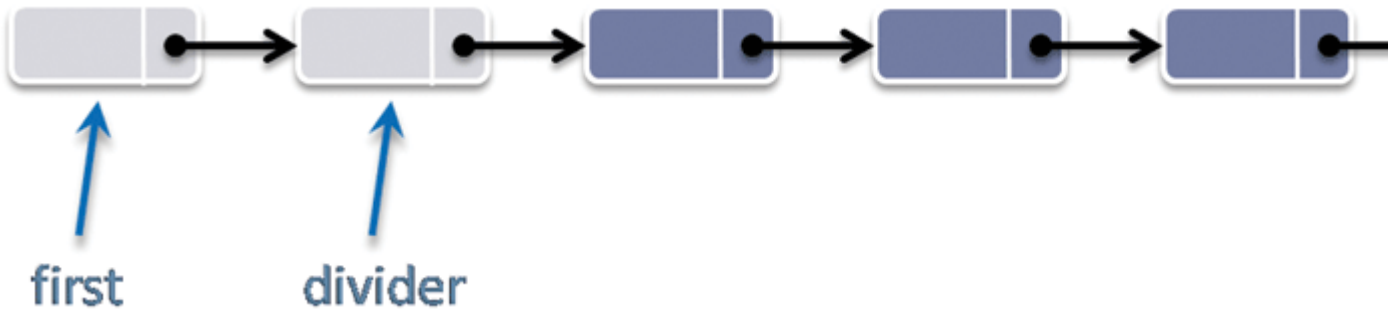

Figure 1: The lock-free queue data structure.

Here's the class definition, which carefully marks shared variables as being of an ordered atomic type (using C++ to most closely follow the original code in [2]):

```
template <typename T>
```

```
class LockFreeQueue {
private:
  struct Node {
    Node( T val ) : value(val), next(nullptr) { }
    T value;
    Node* next;
  };
  Node* first;                    // for producer only
  atomic<Node*> divider, last;          // shared
```

The constructor simply initializes the list with a dummy element. The destructor (in C# or Java, the *dispose* method) releases the list. In a future column, I'll discuss in detail why constructors and destructors of a shared object don't need to worry about concurrency and races with methods of the same object; the short answer for now is that creating or tearing down an object should always run in isolation, so no internal synchronization needed.

```
public:
  LockFreeQueue() {
    first = divider = last =
      new Node( T() );           // add dummy separator
  }
  ~LockFreeQueue() {
    while( first != nullptr ) {   // release the list
      Node* tmp = first;
      first = tmp->next;
      delete tmp;
    }
  }
```

Next, we'll look at the key methods, *Produce* and *Consume*. Figure 2 shows another view of the list by who owns what data by color-coding: The producer owns all nodes before *divider*, the *next* pointer inside the *last* node, and the ability to update *first* and *last*. The consumer owns everything else, including the values in the nodes from *divider* onward, and the ability to update *divider*.

[Click image to view at full size]

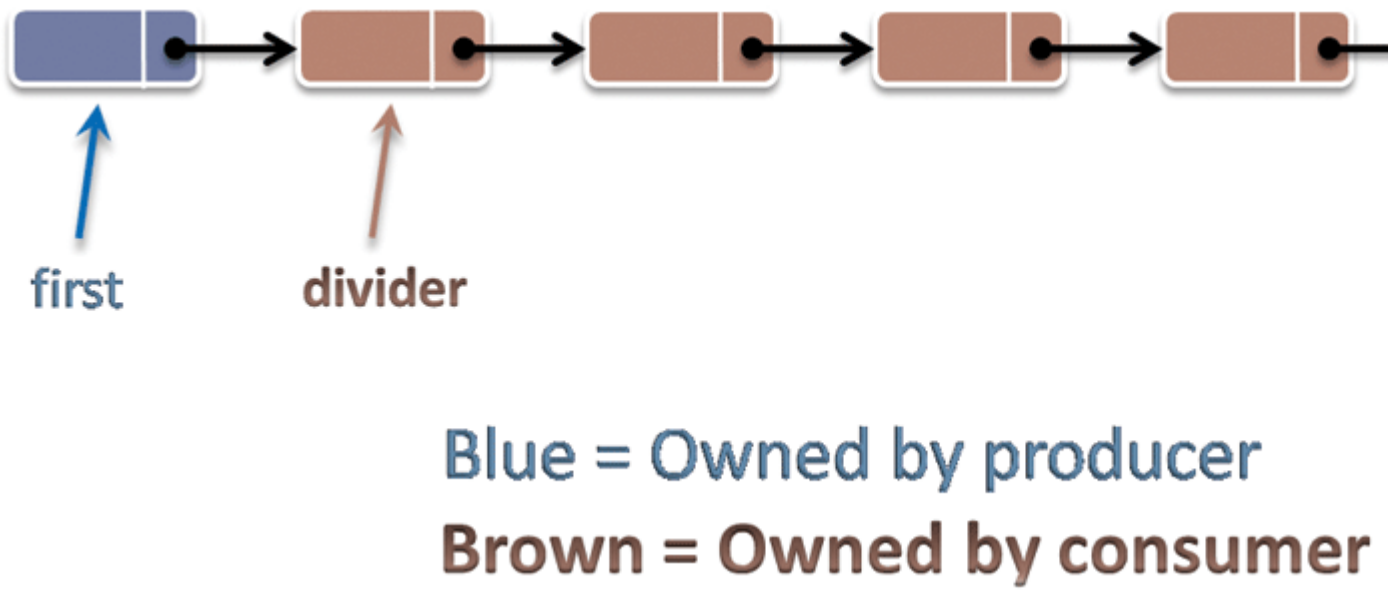

Figure 2: Ownership rules of the road.

The Producer

Produce is called on the producer thread only:

```
void Produce( const T& t ) {
  last->next = new Node(t);          // add the new item
  last = last->next;         // publish it
  while( first != divider ) {        // trim unused nodes
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
}
```

First, the producer creates a new *Node* containing the value and links it to the current *last* node. At this point, the node is not yet shared, but still private to the producer thread even though there's a link to it; the consumer will not follow that link unless the value of *last* says it may follow it. Finally, when all the real work is done—the node exists, its value is completely initialized, and it's correctly connected—then, and only then, do we write to *last* to "commit" the update and publish it atomically to the consumer thread. The consumer reads *last*, and either sees the old value (and ignores the new partly constructed element even if the *last->next* pointer might already have been set) or the new value that officially blesses the new node as an approved part of the queue, ready to be used.

Finally, the producer performs lazy cleanup of now-unused nodes. Because we always stop before *divider*, this can't conflict with anything the consumer might be doing later in the list. What if while we're in the loop, the consumer is consuming items and changing the value of *divider*? No problem: Each time we read *divider*, we see it either before or after any concurrent update by the consumer, both of which let the producer see the list in a consistent state.

The Consumer

Consume is called on the consumer thread only:

```
  bool Consume( T& result ) {
    if( divider != last ) {          // if queue is nonempty
      result = divider->next->value;  // C: copy it back
      divider = divider->next;        // D: publish that we took it
      return true;             // and report success
    }
    return false;              // else report empty
  }
};
```

First, the consumer checks that the list is nonempty by atomically reading *divider*, atomically reading *last*, and comparing them. This one-time check is safe because although *last*'s value may be changed by the producer while we are running the rest of this method, if the check is true once, it will stay true even if *last* moves, because *last* never backs up; it can only move forward to publish new tail nodes—which doesn't affect the consumer, who only cares about the first node after the *divider*. If there is a valid node after *divider*, the consumer copies its value and then, finally, advances *divider* to publish that the queue item was removed.

Yes, we could eliminate the need to make the *last* variable shared: The consumer only uses the value of *last* to check whether there's another node after the *divider*, and we could instead have the consumer just test whether *divider->next* is non-null. That would be fine, and it would let us make *last* an ordinary variable; but if we do that, we must also remember that this change would make each *next* member a shared variable instead, and so to make it safe, we would also have to change *next*'s type to *atomic<Node*>*. I'm leaving *last* as is for now to make it easier to compare this code with the original version in [2], which did use such a tail iterator to communicate between the two threads.

Do Work, Then Publish

You might also have noticed that the original code in [2] did the equivalent of lines C (copy) and D (*divider* update) in the reverse order. You should always be alert and suspicious when you see code that tries to do things backwards: Remember, we're supposed to do all the work off to the side (line C) and only then publish that we did it (line D), as previously shown.

I'm sure someone is about to point out that we could actually get away with writing D then C in this code. Yes, but don't; it's a bad habit. It's true that, in this particular case and now that *divider* is an ordered atomic variable (which wasn't true in the original code), it just so happens that we could get away with writing D then C due to the happy accident of a detail of the implementation combining with a design restriction:

- We always maintain one placeholder *divider* element between the producer and the consumer, so "publishing" the change to *divider* what would otherwise be one step too soon, so that refers to an unconsumed node rather than to a consumed node, happens to be innocuous as long as we're only one step ahead.

- There's exactly one consumer thread, so multiple calls to *Consume* must run in sequence and can never get two steps ahead.

But it's still a bad habit to get into. It's not a good idea to cut corners by relying on "happy" accidents, especially because there's not much to be gained here from breaking the correct pattern. Besides, even if we wrote D then C now, it might be just another thing we'd have to change anyway next month, because...

Coming Up

Next month, we will consider how to generalize the queue for multiple producer and consumer threads. Your homework: What new issues does this raise? What parts of the code we just considered would be broken in the presence of multiple consumers alone and why? What about multiple producers? What about both? Once you've discovered the problems, what would you need to change in the code and in the queue data structure itself to address them?

You have one month. Think of how you would approach it, and we'll take up the challenge when we return.

Notes

[1] H. Sutter. "Lock-Free Code: A False Sense of Security" (DDJ, September 2008). (www.ddj.com/cpp/210604448).

[2] P. Marginean. "Lock-Free Queues" (*DDJ*, July 2008). (www.ddj.com/208801974).

[3] This is just like a canonical exception safety pattern—do all the work off to the side, then commit to accept the new state using nonthrowing operations only. "Think in transactions" applies everywhere, and should be ubiquitous in the way we write our code.

[4] Compare-and-swap (CAS) is the most widely available fundamental lock-free operation and so I'll focus on it here. However, some systems instead provide the equivalently powerful load-linked/store-conditional (LL/SC) instead.

Lock-Free Code: A False Sense of Security

Writing lock-free code can confound anyone—even expert programmers, as Herb shows this month.

By Herb Sutter
September 08, 2008
URL:http://drdobbs.com/cpp/210600279

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

Given that lock-based synchronization has serious problems [1], it can be tempting to think lock-free code must be the answer. Sometimes that is true. In particular, it's useful to have libraries provide hash tables and other handy types whose implementations are internally synchronized using lock-free techniques, such as Java's *ConcurrentHashMap*, so that we can use those types safely from multiple threads without external synchronization and without having to understand the subtle lock-free implementation details.

But replacing locks wholesale by writing your own lock-free code is not the answer. Lock-free code has two major drawbacks. First, it's not broadly useful for solving typical problems—lots of basic data structures, even doubly linked lists, still have no known lock-free implementations. Coming up with a new or improved lock-free data structure will still earn you at least a published paper in a refereed journal, and sometimes a degree.

Second, it's hard even for experts. It's easy to write lock-free code that appears to work, but it's very difficult to write lock-free code that is correct and performs well. Even good magazines and refereed journals have published a substantial amount of lock-free code that was actually broken in subtle ways and needed correction.

To illustrate, let's dissect some peer-reviewed lock-free code that was published here in *DDJ* just two months ago [2]. The author, Petru Marginean, has graciously allowed me to dissect it here so that we can see what's wrong and why, what lessons we should learn, and how to write the code correctly. That someone as knowledgable as Petru, who has published many good and solid articles, can get this stuff wrong should be warning enough that lock-free coding requires great care.

A Limited Lock-Free Queue

Marginean's goal was to write a limited lock-free queue that can be used safely without internal or external locking. To simplify the problem, the article imposed some significant restrictions, including that the queue must only be used from two threads with specific roles: one Producer thread that inserts into the queue, and one Consumer thread that removes items from the queue.

Marginean uses a nice technique that is designed to prevent conflicts between the writer and reader:

- The producer and consumer always work in separate parts of the underlying list, so that their work won't conflict. At any given time, the first "unconsumed" item is the one after the one *iHead* refers to, and the last (most recently added) "unconsumed" item is the one before the one *iTail* refers to.

- The consumer increments *iHead* to tell the producer that it has consumed another item in the queue.
- The producer increments *iTail* to tell the consumer that another item is now available in the queue. Only the producer thread ever actually modifies the queue. That means the producer is responsible, not only for adding into the queue, but also for removing consumed items. To maintain separation between the producer and consumer and prevent them from doing work in adjacent nodes, the producer won't clean up the most recently consumed item (the one referred to by *iHead*).

The idea is reasonable; only the implementation is fatally flawed. Here's the original code, written in C++ and using an STL doubly linked *list<T>* as the underlying data structure. I've reformatted the code slightly for presentation, and added a few comments for readability:

```
// Original code from [1]
// (broken without external locking)
//
template <typename T>
struct LockFreeQueue {
private:
  std::list<T> list;
  typename std::list<T>::iterator iHead, iTail;

public:
  LockFreeQueue() {
    list.push_back(T());              // add dummy separator
    iHead = list.begin();
    iTail = list.end();
  }
```

Produce is called on the producer thread only:

```
  void Produce(const T& t) {
    list.push_back(t);         // add the new item
    iTail = list.end();               // publish it
    list.erase(list.begin(), iHead);  // trim unused nodes
  }
```

Consume is called on the consumer thread only:

```
  bool Consume(T& t) {
    typename std::list<T>::iterator iNext = iHead;
    ++iNext;
    if (iNext != iTail) {             // if queue is nonempty
      iHead = iNext;           // publish that we took an item
      t = *iHead;              // copy it back to the caller
      return true;             // and report success
    }
    return false;              // else report queue was empty
  }
};
```

The fundamental reason that the code is broken is that it has race conditions on both would-be lock-free variables, iHead and iTail. To avoid a race, a lock-free variable must have two key properties that we need to watch for and guarantee: atomicity and ordering. These variables are neither.

Atomicity

First, reads and writes of a lock-free variable must be atomic. For this reason, lock-free variables are typically no larger than the machine's native word size, and are usually pointers (C++), object references (Java, .NET), or integers. Trying to use an ordinary *list<T>::iterator* variable as a lock-free shared variable isn't a good idea and can't reliably meet the atomicity requirement, as we will see.

Let's consider the races on *iHead* and *iTail* in these lines from *Produce* and *Consume*:

```
void Produce(const T& t) {
  ...
  iTail = list.end();
  list.erase(list.begin(), iHead);
}

bool Consume(T& t) {
  ...
  if (iNext != iTail) {
    iHead = iNext;
  ...   }
```

If reads and writes of *iHead* and *iTail* are not atomic, then *Produce* could read a partly updated (and therefore corrupt) *iHead* and try to dereference it, and *Consume* could read a corrupt *iTail* and fall off the end of the queue. Marginean does note this requirement:

"Reading/writing list<T>::iterator is atomic on the machine upon which you run the application." [2]

Alas, atomicity is necessary but not sufficient (see next section), and not supported by *list<T>::iterator*. First, in practice, many *list<T>::iterator* implementations I examined are larger than the native machine/pointer size, which means that they can't be read or written with atomic loads and stores on most architectures. Second, in practice, even if they were of an appropriate size, you'd have to add other decorations to the variable to ensure atomicity, for example to require that the variable be properly aligned in memory.

Finally, the code isn't valid ISO C++. The 1998 C++ Standard said nothing about concurrency, and so provided no such guarantees at all. The upcoming second C++ standard that is now being finalized, C++0x, does include a memory model and thread support, and explicitly forbids it. In brief, C++0x says that the answer to questions such as, "What do I need to do to use a *list<T> mylist* thread-safely?" is "Same as any other object"—if you know that an object like *mylist* is shared, you must externally synchronize access to it, including via iterators, by protecting all such uses with locks, else you've written a race [3]. (Note: Using C++0x's *std::atomic<>* is not an option for *list<T>::iterator*, because

atomic<T> requires *T* to be a bit-copyable type, and STL types and their iterators aren't guaranteed to be that.)

Ordering Problems in Produce

Second, reads and writes of a lock-free variable must occur in an expected order, which is nearly always the exact order they appear in the program source code. But compilers, processors, and caches love to optimize reads and writes, and will helpfully reorder, invent, and remove memory reads and writes unless you prevent it from happening. The right prevention happens implicitly when you use mutex locks or ordered atomic variables (C++0x *std::atomic*, Java/.NET *volatile*); you can also do it explicitly, but with considerably more effort, using ordered API calls (e.g., Win32 *InterlockedExchange*) or memory fences/barriers (e.g., Linux *mb*). Trying to write lock-free code without using any of these tools can't possibly work.

Consider again this code from *Produce*, and ignore that the assignment *iTail* isn't atomic as we look for other problems:

```
list.push_back(t);     // A: add the new item
iTail = list.end();    // B: publish it
```

This is a classic publication race because lines *A* and *B* can be (partly or entirely) reordered. For example, let's say that some of the writes to the *T* object's members are delayed until after the write to *iTail*, which publishes that the new object is available; then the consumer thread can see a partly assigned *T* object.

What is the minimum necessary fix? We might be tempted to write a memory barrier between the two lines:

```
// Is this change enough?
list.push_back(t);     // A: add the new item
mb();          // full fence
iTail = list.end();    // B: publish it
```

Before reading on, think about it and see if you're convinced that this is (or isn't) right.

Have you thought about it? As a starter, here's one issue: Although *list.end* is probably unlikely to perform writes, it's possible that it might, and those are side effects that need to be complete before we publish *iTail*. The general issue is that you can't make assumptions about the side effects of library functions you call, and you have to make sure they're fully performed before you publish the new state. So a slightly improved version might try to store the result of *list.end* into a local unshared variable and assign it after the barrier:

```
// Better, but is it enough?
list.push_back(t);
tmp = list.end();
mb();          // full fence
iTail = tmp;
```

Unfortunately, this still isn't enough. Besides the fact that assigning to *iTail* isn't atomic and that we still have a race on *iTail* in general, compilers and processors can also invent writes to *iTail* that break this code. Let's consider write invention in the context of another problem area: *Consume*.

Ordering Problems in Consume

Here's another reordering problem, this time from *Consume*:

```
if (iNext != iTail) {
  iHead = iNext;               // C
  t = *iHead;          // D
```

Note that *Consume* updates *iHead* to advertise that it has consumed another item before it actually reads the item's value. Is that a problem? We might think it's innocuous, because the producer always leaves the *iHead* item alone to stay at least one item away from the part of the list the consumer is using.

It turns out this code is broken regardless of which order we write lines *C* and *D*, because the compiler or processor or cache can reorder either version in unfortunate ways. Consider what happens if the consumer thread performs a consecutive two calls to *Consume*: The memory reads and writes performed by those two calls could be reordered so that *iHead* is incremented twice before we copy the two list nodes' values, and then we have a problem because the producer may try to remove nodes the consumer is still using. Note: This doesn't mean the compiler or processor transformations are broken; they're not. Rather, the code is racy and has insufficient synchronization, and so it breaks the memory model guarantees and makes such transformations possible and visible.

Reordering isn't the only issue. Another problem is that compilers and processors can invent writes, so they could inject a transient value:

```
// Problematic compiler/processor transformation
if (iNext != iTail) {
  iHead = 0xDEADBEEF;
  iHead = iNext;
  t = *iHead;
```

Clearly, that would break the producer thread, which would read a bad value for *iHead*. More likely, the compiler or processor might speculate that most of the time *iNext != iTail*:

```
// Another problematic transformation
__temp = iHead;
iHead = iNext;     // speculatively set to iNext
if (iNext == iTail) {      // note: inverted test!
  iHead = __temp;  // undo if we guessed wrong
} else {
```

```
t = *iHead;
```

But now *iHead* could equal *iTail*, which breaks the essential invariant that *iHead* must never equal *iTail*, on which the whole design depends.

Can we solve these problems by writing line *D* before *C*, then separating them with a full fence? Not entirely: That will prevent most of the aforementioned optimizations, but it will not eliminate all of the problematic invented writes. More is needed.

Next Steps

These are a sample of the concurrency problems in the original code. Marginean showed a good algorithm, but the implementation is broken because it uses an inappropriate type and performs insufficient synchronization/ordering. Fixing the code will require a rewrite, because we need to change the data structure and the code to let us use proper ordered atomic lock-free variables. But how? Next month, we'll consider a fixed version. Stay tuned.

Notes

[1] H. Sutter, "The Trouble With Locks," *C/C++ Users Journal*, March 2005. (www.ddj.com/cpp/184401930).

[2] P. Marginean, "Lock-Free Queues," *Dr. Dobb's Journal*, July 2008. (www.ddj.com/208801974).

[3] B. Dawes, et al., "Thread-Safety in the Standard Library," *ISO/IEC JTC1/SC22/WG21 N2669*, June 2008. (www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2669.htm).

The Many Faces of Deadlock

Deadlock can happen whenever there is a blocking (or waiting) cycle among concurrent tasks.

By Herb Sutter

July 31, 2008
URL:http://drdobbs.com/high-performance-computing/209900973

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

Quick: What is "deadlock"? How would you define it? One common answer is: "When two threads each try to take a lock the other already holds." Yes, we do indeed have a potential deadlock anytime two threads (or other concurrent tasks, such as work items running on a thread pool; for simplicity, I'll say "threads" for the purposes of this article) try to acquire the same two locks in opposite orders, and the threads might run concurrently. The potential deadlock will only manifest when the threads actually do run concurrently, and interleave in an appropriately bad way where each successfully takes its first lock and then waits forever on the other's, as in the execution *A->B->C->D* in the following example:

```
// Thread 1
mut1.lock();           // A
mut2.lock();           // C: blocks

// Thread 2
mut2.lock();           // B
mut1.lock();           // D: blocks
```

That's the classic deadlock example from college. Of course, two isn't a magic number. An improved definition of deadlock is: "When *N* threads enter a locking cycle where each tries to take a lock the next already holds."

Deadlock Among Messages

"But wait," someone might say. "I once had a deadlock just like the code you just showed, but it didn't involve locks at all—it involved messages." For example, consider this code, which is similar to the lock-based deadlock example:

```
// Thread 1
mq1.receive();         // blocks
mq2.send( x );

// Thread 2
mq2.receive();         // blocks
mq1.send( y );
```

Now Thread 1 is blocked, waiting for a message that Thread 2 hasn't sent yet. Unfortunately, Thread 2 is also blocked, waiting in turn for a message that Thread 1 hasn't sent yet. The result: A deadlock that is qualitatively the same as the one that arose from locks. So, then, what is a complete definition of deadlock?

Dead(b)locks

The key thing to recognize is that deadlock doesn't arise from a locking cycle exclusively. Any blocking (or, waiting) cycle will do. So a complete definition of deadlock could be put something like this:

deadlock, n.: When N concurrent tasks enter a cycle where each one is blocked waiting for the next.

Clearly, blocking operations include all kinds of blocking synchronization: acquiring a lock (of course); waiting for a semaphore or condition variable to be signaled; waiting to join with another thread or task...But it also includes any other kind of call that waits for a result, including:

- Acquiring exclusive access to any resource (notably I/O, such as opening a file for writing).
- Waiting for a future or write-once variable to be available (the result of an asynchronous task).
- Waiting for any other kind of message to arrive from elsewhere, whether in-process (waiting for a queue container to become nonempty, waiting for a message to be placed into a GUI message queue) or out-of-process or out-of-box (performing a blocking receive call on a TCP socket, for instance).
- Anything else that waits for some other condition to be satisfied.

Complex Combinations

For a more complex example, consider Figure 1. There are four things going on:

- Thread 1 is a GUI window message handler that's in the middle of processing a message. As part of that processing, it needs to use some shared state and so wants to take a lock on mutex *mut*—which is currently held by Thread 4.
- Thread 4 is doing something with the same shared state and so has acquired *mut* already, and is just waiting to receive a message from a message queue—a message that is yet to be sent from Thread 2. (Doing communication inside a critical section is problematic and should be avoided where possible. See [2] for more about why to avoid doing communication while holding a lock.)
- Thread 2 hasn't got around to sending that message yet, because it's waiting to open a file for writing, which is an exclusive mode—a file currently opened for writing and appending on Thread 3.
- Thread 3 is working with the file, and posts a message to the window, which is a synchronous call that blocks to wait for the message to be handled—but Thread 1 is already handling a message and so can't pump and handle this one.

[Click image to view at full size]

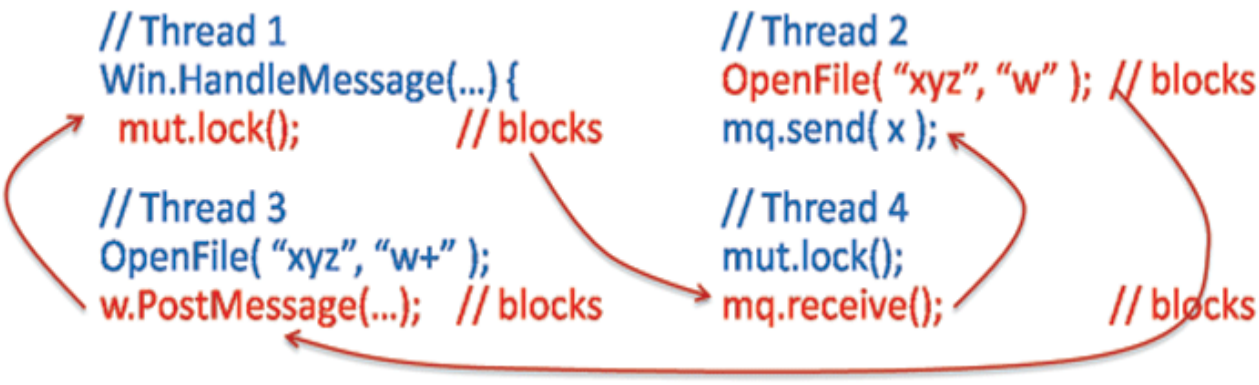

Figure 1: Sample deadlock among locks, message pumps, messages queues, and file access.

Finally, for extra fun and just to emphasize the point that any kind of blocking operation will do, consider that deadlock can involve a human—you! For example:

```
// Thread 1 (running in the CPU hardware)
mut.lock();
char = ReadKeystroke();                        // blocks

// Task 2 (running in your brain's wetware)
Wait for the prompt to appear.                 // blocks
OK, now start typing.

// Thread 3 (back in the hardware again)
mut1.lock();                    // blocks
Print ( "Press 'Any' key to continue..." );
```

Here, Thread 1 can't make progress because it's blocked waiting for you to press a key. You haven't pressed a key because you're still waiting for a prompt. Thread 3 can't print the prompt until it returns from being blocked waiting to lock a mutex...held by Thread 1. Around and around we go.

Dealing With Deadlock

The good news is that we have tools and techniques to detect and prevent many kinds of deadlock. In particular, consider ways to deal with two popular forms of blocking: Waiting to acquire a mutex, and waiting to receive a message.

First, to prevent locking cycles within the code you control, use lock hierarchies and other ordering techniques to ensure that your program always acquires all mutexes in the same order [1]. Note: Not all locks need to participate directly in a lock hierarchy; for example, some groups of related locks already have a total ordering among themselves, such as a chain of locks always traversed in the same order (hand-over-hand locking along a singly linked list, for instance).

Second, to prevent many kinds of message cycles, disciplines have been proposed to express contracts on communication channels, which helps to impose a known ordering on messages. One example is WS-CDL [3]. The general idea is to define rules for the orders in which messages must be sent and received, which can be enforced statically or dynamically to ensure that components communicating via messages won't tie themselves into a knot. A message ordering contract is typically expressed as some form of state machine. For example, here's how we might express a simple interface in pseudocode for a buyer requesting a purchase order:

```
contract PORequest {
// messages from requester to
// provider (arbitrarily
// choose "in" to be from the
// provider's point of view)
  in void Request( Info );

// messages from provider
```

```
// to requester
  out void Ack();
  out void Response( Details );

// now declare states and transitions
  state Start {
   Request -> RequestSent;
  }
  state RequestSent {
   Ack -> RequestReceived;
  }
  state RequestReceived {
   Response -> End;
  }
}
```

The only valid message order is *Request->Ack->Response.* If a provider could mistakenly send a *Response* without first sending an *Ack*, a requester could hang indefinitely waiting for the missing *Ack* message. Expressing the message order contract gives us a way to guarantee that a provider fulfills the contract, or at least to detect violations.

General Deadlock Detection

The bad news is that, as far as I know, there's no tool on the planet that identifies all kinds of blocking cycles for you, especially ones that consist of more than one kind of blocking. Lock hierarchies only guarantee freedom from deadlock among locks in the code you control; message contracts on communication channels only guarantee freedom from deadlock among messages.

Probably the best you can do today is to roll your own deadlock detection in code, by adopting a discipline like the following:

- Identify every condition or resource that can be waited for (a mutex, a message, a value of an atomic variable, an exclusive use of a file) and give it a unique name by creating a dummy object to represent it.
- Instrument every "start wait" and "end wait" point in your code by calling two helper functions: The first records that a condition or resource is about to be waited for (e.g,. StartWait(thing)) and should internally record the current thread's ID and the thing being waited for; it can also check to see if there is now a waiting cycle among threads and resources, and report the deadlock if that occurs. The second records that a wait has ended (e.g., EndWait(thing)) and will remove the waiting edge.

To illustrate how we can apply such a discipline, consider this deadlock between a mutex and a message that arises in the execution *A->B->C*:

```
// Global data
Mutex mut;
MessageQueue queue;

// Thread 1
mut.lock();            // A
queue.receive();               // B: blocks
mut.unlock();
```

```
// Thread 2
mut.lock();            // C: blocks
queue.send( msg );
```

We could apply a wait start/end instrumentation discipline as follows. Here the implementation of *StartWait* and *EndWait* is left for the reader, but should record which threads are waiting for which objects as described above:

```
// Global data
Mutex mut;
WaitableObject w_mut;
MessageQueue queue;
WaitableObject w_queue;
// Thread 1
StartWait( w_mut );
mut.lock();              // A
EndWait( w_mut );
StartWait( w_queue );
queue.receive();         // B: blocks
EndWait( w_queue );
mut.unlock();

// Thread 2
StartWait( w_mut );
mut.lock();              // C: blocks
EndWait( w_mut );
queue.send( msg );
```

That's a sketch of the idea. It's only a coding discipline, but it's an approach that can help you to instrument all waiting in a unified way, at least within the code you control.

Summary

Deadlock can arise whenever there is a blocking (or waiting) cycle among concurrent tasks, where each one is waiting for the next to produce some value or release some resource. Eliminate deadlocks as much as possible by applying ordering techniques like lock hierarchies and message contracts; these techniques are important, even though they are incomplete because each one deals with only a specific kind of waiting. Then consider adding your own deadlock detection by instrumenting the wait points in your code in a uniform way.

Whimsically, we might say that a more correct name for deadlock could be "deadblock"...but the world has already adopted a common spelling that's one letter shorter, and this isn't the time to try to change that. When reasoning about deadlock, remember not to forget the important "b", even though it's silent in pronunciation and in the common spelling.

Notes

[1] H. Sutter. "Use Lock Hierarchies To Avoid Deadlock" (*DDJ*, January 2008).

[2] H. Sutter. "Avoid Calling Unknown Code While Inside a Critical Section" (*DDJ*, December 2007).

[3] Web Services Choreography Description Language (WS-CDL) (*W3C*, 2005) (www.w3.org/TR/ws-cdl-10/).

Choose Concurrency-Friendly Data Structures

Linked Lists and Balanced Search Trees are familiar data structures, but can they make the leap to parallelized environments?

By Herb Sutter
June 27, 2008
URL:http://drdobbs.com/high-performance-computing/208801371

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

What is a high-performance data structure? To answer that question, we're used to applying normal considerations like Big-Oh complexity, and memory overhead, locality, and traversal order. All of those apply to both sequential and concurrent software.

But in concurrent code, we need to consider two additional things to help us pick a data structure that is also sufficiently concurrency-friendly:

- In parallel code, your performance needs likely include the ability to allow multiple threads to use the data at the same time. If this is (or may become) a high-contention data structure, does it allow for concurrent readers and/or writers in different parts of the data structure at the same time? If the answer is, "No," then you may be designing an inherent bottleneck into your system and be just asking for lock convoys as threads wait, only one being able to use the data structure at a time.
- On parallel hardware, you may also care about minimizing the cost of memory synchronization. When one thread updates one part of the data structure, how much memory needs to be moved to make the change visible to another thread? If the answer is, "More than just the part that has ostensibly changed," then again you're asking for a potential performance penalty, this time due to cache sloshing as more data has to move from the core that performed the update to the core that is reading the result.

It turns out that both of these answers are directly influenced by whether the data structure allows truly localized updates. If making what appears to be a small change in one part of the data structure actually ends up reading or writing other parts of the structure, then we lose locality; those other parts need to be locked, too, and all of the memory that has changed needs to be synchronized.

To illustrate, let's consider two common data structures: linked lists and balanced trees.

Linked Lists

Linked lists are wonderfully concurrency-friendly data structures because they support highly localized updates. In particular, as illustrated in Figure 1, to insert a new node into a doubly linked list, you only need to touch two existing nodes; namely, the ones immediately adjacent to the position the new node will occupy to splice the new node into the list. To erase a node, you only need to touch three nodes: the one that is being erased, and its two immediately adjacent nodes.

This locality enables the option of using fine-grained locking: We can allow a potentially large number of threads to be actively working inside the same list, knowing that they won't conflict as long as they are manipulating different parts of the list. Each operation only needs to lock enough of the list to cover the nodes it actually uses.

For example, consider Figure 2, which illustrates the technique of hand-over-hand locking. The basic idea is this: Each segment of the list, or even each individual node, is protected by its own mutex. Each thread that may add or remove nodes from the list takes a lock on the first node, then while still holding that, takes a lock on the next node; then it lets go of the first node and while still holding a lock on the second node, it takes a lock on the third node; and so on. (To delete a node requires locking three nodes.) While traversing the list, each such thread always holds at least two locks—and the locks are always taken in the same order.

[Click image to view at full size]

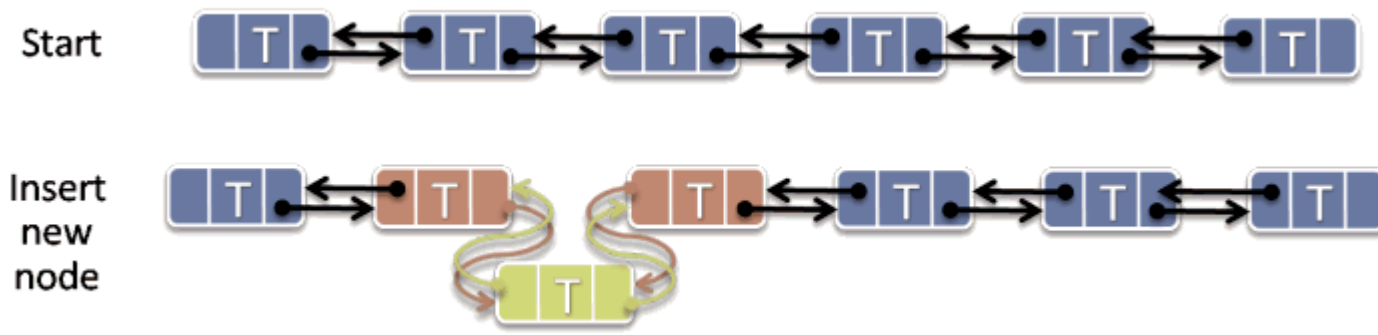

Figure 1: Localized insertion into a linked list.

[Click image to view at full size]

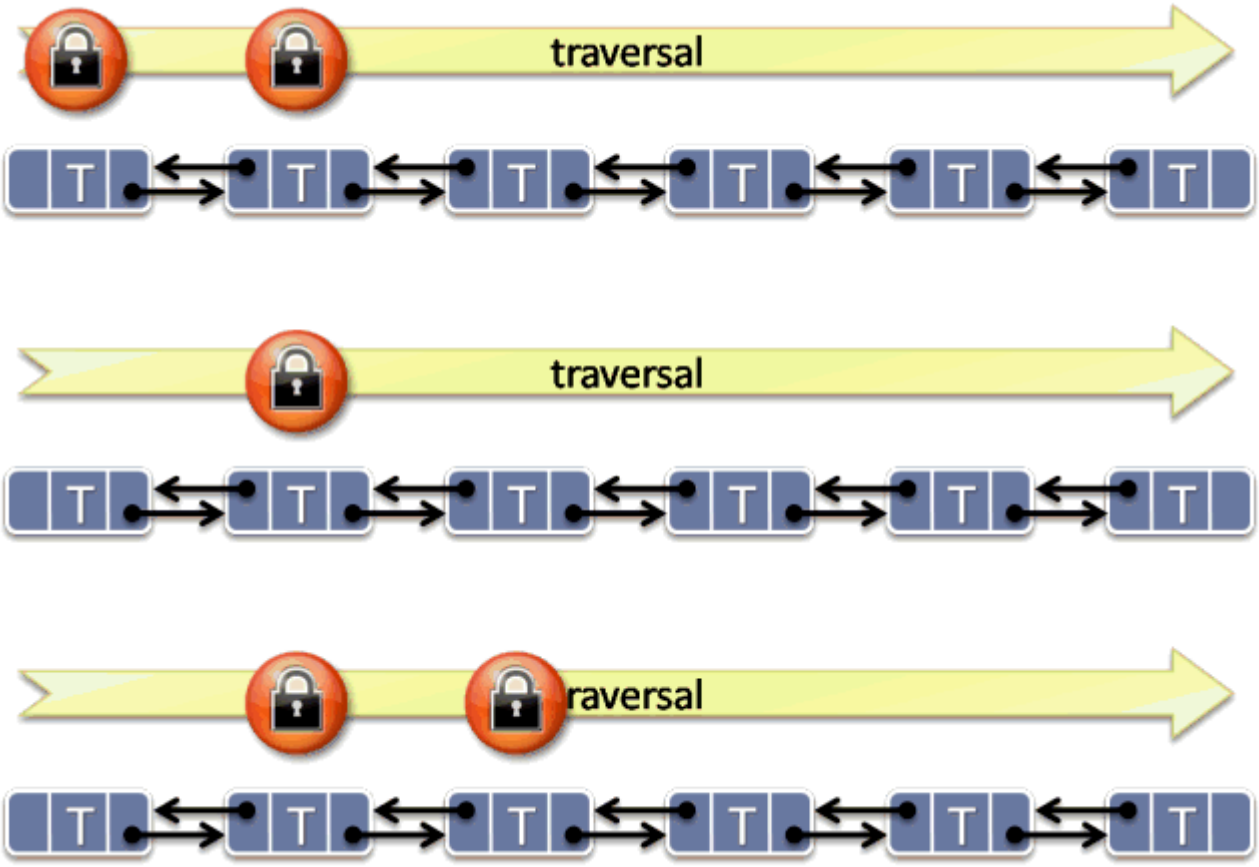

Figure 2: Hand-over-hand locking in a linked list.

This technique delivers a number of benefits, including the following:

- Multiple readers and writers can be actively doing work in the same list.
- Readers and writers that are traversing the list in the same order will not pass each other. This can be useful to get deterministic results in concurrent code. In particular, the list's semantics will be the same as if each thread acquired complete exclusion on the list and performed its complete pass in isolation, which is easy to reason about.
- The locks taken on parts of the list won't deadlock with each other, because multiple locks are acquired in the same order.
- We can readily tune the code for better concurrency vs. lower locking overhead by choosing a suitable locking granularity: one lock for the whole list (no concurrency), a lock for each node in the list (maximum concurrency), or a lock for each chunk of some fixed or variable length (something in between).

Aside: If we always traverse the list in the same order, why does the figure show a doubly linked list? Because not all operations need to take multiple locks; those that use individual segments or nodes in-place one at a time without taking more than one node's or chunk's lock at a time can traverse the list in any order without deadlock. (For more on avoiding deadlock, see [1].)

Besides being well suited for concurrent traversal and update, linked lists also are cache-friendly on parallel hardware. When one thread removes a node, for example, the only

memory that needs to be transferred to every other core that subsequently reads the list is the memory containing the two adjacent nodes. If the rest of the list hasn't been changed, multiple cores can happily store read-only copies of the list in their caches without expensive memory fetches and synchronization. (Remember, writes are always more expensive than reads because writes need to be broadcast. In turn, "lots of writes" are always more expensive than "limited writes.")

Clearly, one benefit lists enjoy is that they are node-based containers: Each element is stored in its own node, unlike an array or vector where elements are contiguous and inserting or erasing typically involves copying an arbitrary number of elements to one side or the other of the inserted or erased value. We might therefore anticipate that perhaps all node-based containers will be good for concurrency. Unfortunately, we would be wrong.

Balanced Search Trees

The story isn't nearly as good for another popular data structure: the balanced search tree. (Important note: This section refers only to balanced trees; unbalanced trees that support localized updates don't suffer from the problems we'll describe next.)

Consider a red-black tree: The tree stays balanced by marking each node as either "red" or "black," and applying rules that call for optionally rebalancing the tree on each insert or erase to avoid having different branches of the tree become too uneven. In particular, rebalancing is done by rotating subtrees, which involves touching an inserted or erased node's parent and/or uncle node, that node's own parent and/or uncle, and so on to the grandparents and granduncles up the tree, possibly as far as the root.

For example, consider Figure 3. To start with, the tree contains three nodes with the values *1*, *2*, and *3*. To insert the value *4*, we simply make it a child of node *3*, as we would in a nonbalanced binary search tree. Clearly, that involves writing to node *3*, to set its right-child pointer. However, to satisfy the red-black tree mechanics, we must also change node *3*'s and node *1*'s color to black. That adds overhead and loses some concurrency; for example, inserting *4* would conflict with adding 1.5 concurrently, because both inserts would need to touch both nodes *1* and *3*.

[Click image to view at full size]

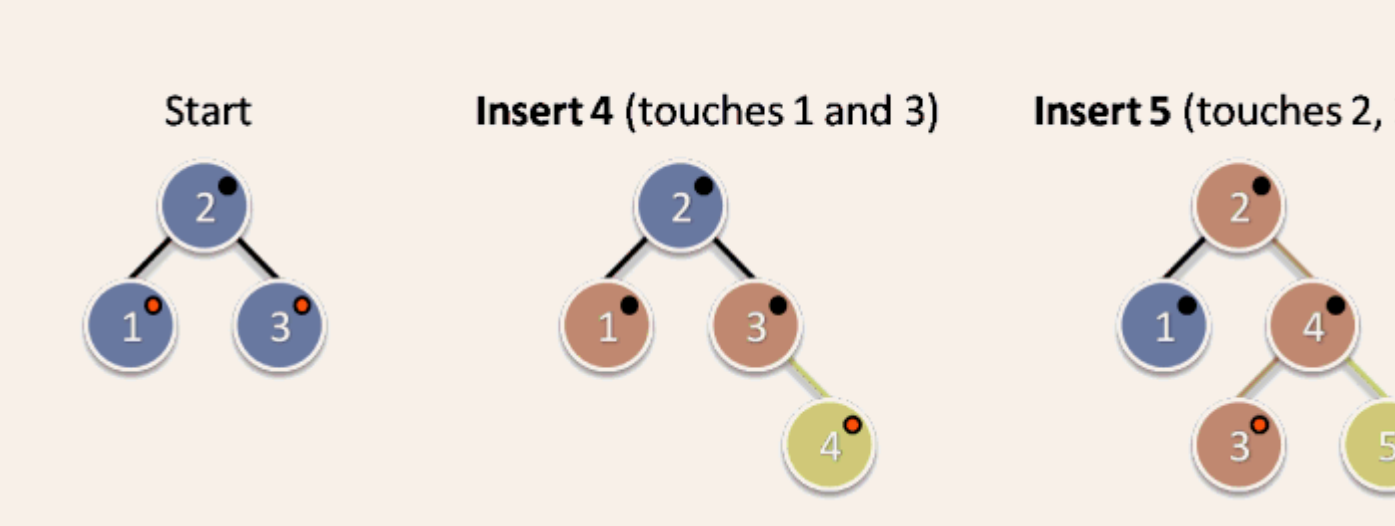

Figure 3: Nonlocalized insertion into a red-black tree.

Next, to insert the value *5*, we need to touch all but one of the nodes in the tree: We first make node *4* point to the new node *5* as its right child, then recolor both node *4* and node *3*, and then because the tree is out of balance we also rotate *3-4-5* to make node *4* the root of that subtree, which means also touching node *2* to install node 4 as its new right child.

So red-black trees cause some problems for concurrent code:

- It's hard to run updates truly concurrently because updates arbitrarily far apart in the tree can touch the same nodes—especially the root, but also other higher-level nodes to lesser degrees—and therefore contend with each other. We have lost the ability to make truly localized changes.
- The tree performs extra internal housekeeping writes. This increases the amount of shared data that needs to be written and synchronized across caches to publish what would be a small update in another data structure.

"But wait," I can hear some people saying, "why can't we just put a mutex inside each node and take the locks in a single direction (up the tree) like we could do with the linked list and hand-over-hand locking? Wouldn't that let us regain the ability to have concurrent use of the data structure at least?" Short answer: That's easy to do, but hard to do right. Unlike the linked list case, however: (a) you may need to take many more locks, even all the way up to the root; and (b) the higher-level nodes will still end up being high-contention resources that bottleneck scalability. Also, the code to do this is much more complicated. As Fraser noted in 2004: "One superficially attractive solution is to read-lock down the tree and then write-lock on the way back up, just as far as rebalancing operations are required. This scheme would acquire exclusive access to the minimal number of nodes (those that are actually modified), but can result in deadlock with search operations (which are locking down the tree)." [2] He also proposed a fine-grained locking technique that does allow some concurrency, but notes that it "is significantly more complicated." There are easy answers, but few easy and correct answers.

To get around these limitations, researchers have worked on alternative structures such as skip lists [4], and on variants of red-black trees that can be more amenable to concurrency, such as by doing relaxed balancing instead of rebalancing immediately when needed after each update. Some of these are significantly more complex, which incurs its own costs in both performance and correctness/maintainability (for example, relaxed balancing was first suggested in 1978 but not implemented successfully until five years later). For more information and some relative performance measurements showing how even concurrent versions can still limit scalability, see [3].

Conclusions

Concurrency-friendliness alone doesn't singlehandedly trump other performance requirements. The usual performance considerations of Big-Oh complexity, and memory overhead, locality, and traversal order all still apply. Even when writing parallel code, you shouldn't choose a data structure only because it's concurrency-friendly; you should choose the right one that meets all your performance needs. Lists may be more concurrency-friendly than balanced trees, but trees are faster to search, and "individual searches are fast" can

outbalance "multiple searches can run in parallel." (If you need both, try an alternative like skip lists.)

Remember:

- In parallel code, your performance needs likely include the ability to allow multiple threads to use the data at the same time.
- On parallel hardware, you may also care about minimizing the cost of memory synchronization.

In those situations, prefer concurrency-friendly data structures. The more a container supports truly localized updates, the more concurrency you can have as multiple threads can actively use different parts of the data structure at the same time, and (secondarily but still sometimes importantly) the more you can avoid invisible memory synchronization overhead in your high-performance code.

Notes

[1] H. Sutter. "Use Lock Hierarchies to Avoid Deadlock" (*Dr. Dobb's Journal*, January 2008).

[2] K. Fraser. "Practical lock-freedom" (*University of Cambridge Computer Laboratory Technical Report #579*, February 2004).

[3] S. Hanke. "The Performance of Concurrent Red-Black Tree Algorithms" (*Lecture Notes in Computer Science*, 1668:286-300, Springer, 1999).

[4] M. Fomitchev and E. Ruppert. "Lock-Free Linked Lists and Skip Lists" (*PODC '04*, July 2004).

Maximize Locality, Minimize Contention

Source Code Accompanies This Article. Download It Now.

- lock.txt

Want to kill your parallel application's scalability? Easy: Just add a dash of contention.

By Herb Sutter
May 23, 2008

URL:http://drdobbs.com/high-performance-computing/208200273

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

In the concurrent world, locality is a first-order issue that trumps most other performance considerations. Now locality is no longer just about fitting well into cache and RAM, but to avoid scalability busters by keeping tightly coupled data physically close together and separately used data far, far apart.

Of Course, You'd Never Convoy On a Global Lock

Nobody would ever willingly write code like this in a tight loop.

```
// Threads 1-N
while( ... ) {
  globalMutex.lock();
  DoWork();
  globalMutex.unlock();
}
```

This is clearly foolish, because all the work is being done while holding a global lock and so only one thread can make progress at a time. We end up with a classic lock convoy: At any given time, all of the threads save one are piled up behind the lock as each waits idly for its turn to come.

Convoys are a classic way to kill parallel scalability. In this example, we'll get no parallel speedup at all because this is just a fancy way to write sequential code. In fact, we'll probably get a minor performance hit because of taking and releasing the lock many times and incurring context switches, and so we would be better off just putting the lock around the whole loop and making the convoy more obvious.

True, we sometimes still gain some parallel benefit when only part of each thread's work is done while holding the lock:

```
// Threads 1-N
while( ... ) {
  DoParallelWork();     // p = time spent here,
                        // parallel portion
  globalMutex.lock();
  DoSequentialWork();   // s = time spent here,
                        // sequential portion
  globalMutex.unlock();
}
```

Now at least some of the threads' work can be done in parallel. Of course, we still hit Amdahl's Law [1]: Even on infinitely parallel hardware with an infinite number of workers,

the maximum speedup is merely *(s+p)/s*, minus whatever overhead we incur to perform the locking and context switches. But if we fail to measure (ahem) and the time spent in *p* is much less than in *s*, we're really gaining little and we've again written a convoy.

We'd never willingly do this. But the ugly truth is that we do it all the time: Sometimes it happens unintentionally; for example, when some function we call might be taking a lock on a popular mutex unbeknownst to us. But often it happens invisibly, when hardware will helpfully take exactly such an exclusive lock automatically, and silently, on our behalf. Let's see how, why, and what we can do about it.

Recap: Chunky Memory

For high-performance code, we've always had to be aware of paging and caching effects. Now, hardware concurrency adds a whole new layer to consider.

When you ask for a byte of memory, the system never retrieves just one byte. We probably all know that nearly all computer systems keep track—not of bytes, but of chunks of memory. There are two major levels at which chunking occurs: the operating system chunks virtual memory into pages, each of which is managed as a unit, and the cache hardware further chunks memory into cache lines, which again are each handled as a unit. Figure 1 shows a simplified view. (In a previous article, we considered some issues that arise from nonshared caches, where only subsets of processors share caches in common [2].)

First, consider memory pages: How big is a page? That's up to the OS and varies by platform, but on mainstream systems the page size is typically 4K or more (see Figure 2). So when you ask for just one byte on a page that's not currently in memory, you incur two main costs:

- Speed: A page fault where the OS has to load the entire page from disk.
- Space: Memory overhead for storing the entire page in memory, even if you only ever touch one byte from the page.

Second, consider cache lines: How big is a line? That's up to the cache hardware and again varies, but on mainstream systems the line size is typically 64 bytes (see Figure 2). So when you ask for just one byte on a line that's not currently in cache, you incur two main costs:

- Speed: A cache miss where the cache hardware has to load the entire line from memory.
- Space: Cache overhead for storing the entire line in cache, even if you only ever touch one byte from the line.

And now comes the fun part. On multicore hardware, if one core writes to a byte of memory, then typically, as part of the hardware's cache coherency protocol, that core will automatically (read: invisibly) take an exclusive write lock on that cache line. The good news is that this prevents other cores from causing trouble by trying to perform conflicting writes. The sad news is that it also means, well, taking a lock.

[Click image to view at full size]

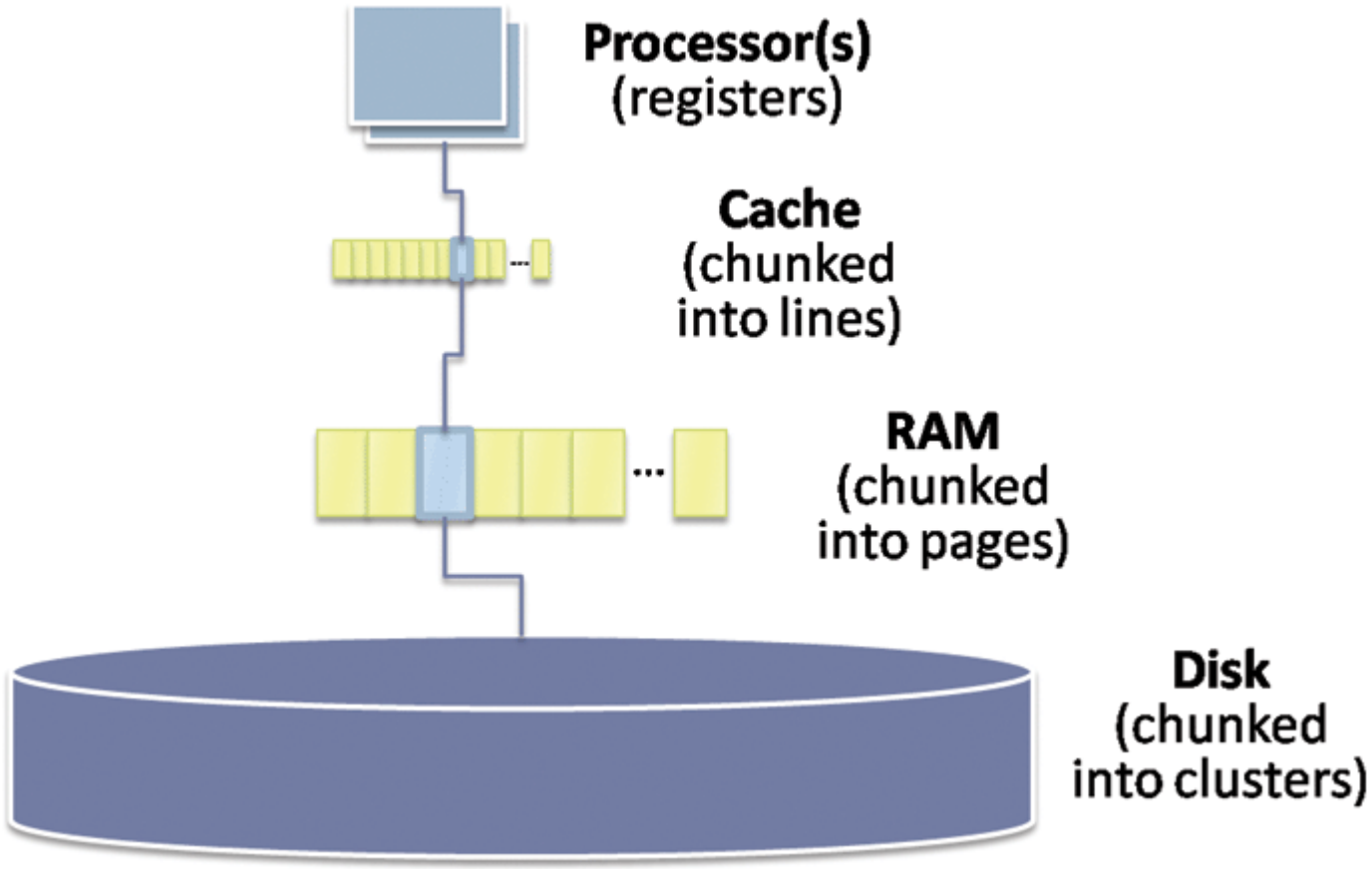

Figure 1: Chunking in the memory hierarchy.

[Click image to view at full size]

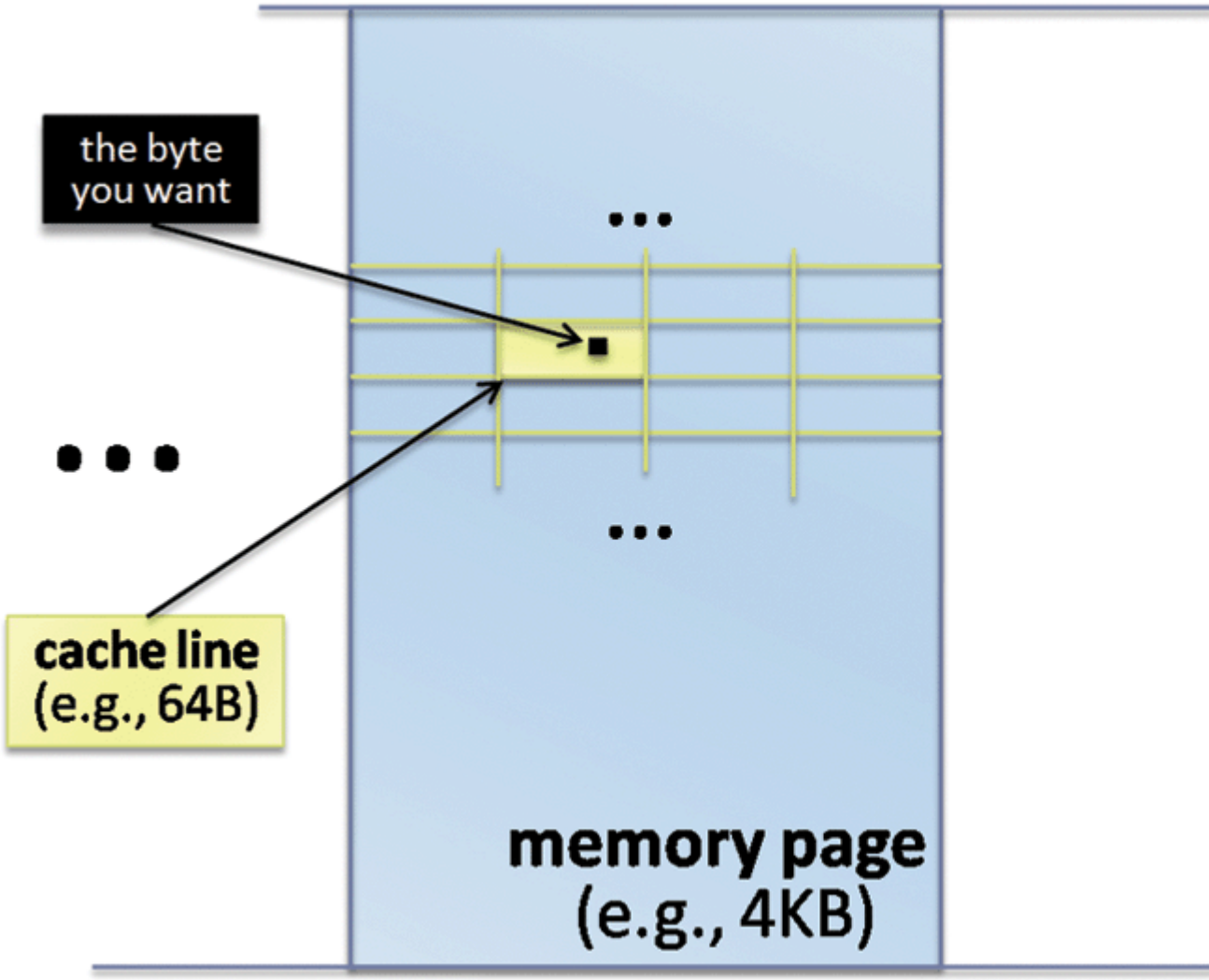

Figure 2: Load a byte = load a line + load a page.

Sharing and False Sharing (Ping-Pong)

Consider the following code where two threads update two distinct global integers *x* and *y*, and assume we've disabled optimizations to prevent the optimizer from eliminating the loops entirely in this toy example:

```
// Thread 1
for( i = 0; i < MAX; ++i ) {
  ++x;
}

// Thread 2
for( i = 0; i < MAX; ++i ) {
  ++y;
}
```

Question: What relative performance would you expect if running Thread 1 in isolation versus running both threads:

- On a machine with one core?
- On a machine with two or more cores?

On a machine with one core, the program would probably take twice as long to run, as we'd probably get the same throughput (additions/sec), maybe with a little overhead for context switches as the operating system schedules the two threads interleaved on the single core.

On a machine with two or more cores, we'd probably expect to get a 2x throughput improvement as the two threads each run at full speed on their own cores. And that is what in fact will happen...but only if *x* and *y* are on different cache lines.

If *x* and *y* are on the same cache line, however, only one core can be updating the cache line at a time, because only one core can have exclusive access at a time—it's as if the cache line is a token being passed between the threads/cores to say who is currently allowed to run. So the situation is exactly as if we had explicitly written:

```
// Thread 1
for( i = 0; i < MAX; ++i ) {
  lightweightMutexForXandY.lock();
  ++x;
  lightweightMutexForXandY.unlock();
}

// Thread 2
for( i = 0; i < MAX; ++i ) {
  lightweightMutexForXandY.lock();
  ++y;
  lightweightMutexForXandY.unlock();
}
```

Which of course is exactly what we said we would never willingly do: Only one thread can make progress at a time. This effect is called "false sharing" because, even though the cores are trying to update different parts of the cache line, that doesn't matter; the unit of sharing is the whole line, and so the performance effect is the same as if the two threads were trying to share the same variable. It's also called ping-ponging because that's an apt description of how the cache line ownership keeps hopping back and forth madly between the two cores.

Even in this simple example, what could we do to ensure *x* and *y* are on different cache lines? First, we can rearrange the data: For example, if *x* and *y* are data values inside the same object, perhaps we can rearrange the object's members so that *x* and *y* are sufficiently further apart. Second, we can add padding: If we have no other data that's easy to put adjacent to *x*, we can ensure *x* is alone on its cache line by allocating extra space, such as by allocating a larger object with *x* as a member (preceded/followed by appropriate padding to fill the cache line) instead of allocating *x* by itself as a naked integer. This is a great example of how to deliberately waste space to improve performance.

False sharing arises in lots of hard-to-see places. For example:

- Two independent variables or objects are too close together, as in the above examples.
- Two node-based containers (lists or trees, for example) interleave in memory, so that the same cache line contains nodes from two containers.

Cache-Conscious Design

Locality is a first-order issue that trumps much of our existing understanding of performance optimization. Most traditional optimization techniques come after "locality" on parallel hardware (although a few are still equally or more important than locality, such as big-Oh algorithmic complexity for example).

Arrange your data carefully by following these three guidelines, starting with the most important:

First: Keep data that are not used together apart in memory. If variables *A* and *B* are not protected by the same mutex and are liable to be used by two different threads, keep them on separate cache lines. (Add padding, if necessary; it's a great way to "waste" memory to make your code run faster.) This avoids the invisible convoying of false sharing (ping-pong) where in the worst case only one contending thread can make progress at all, and so typically trumps other important cache considerations.

Second: Keep data that is frequently used together close together in memory. If a thread that uses variable *A* frequently also needs variable *B*, try to put *A* and *B* in the same cache line if possible. For example, *A* and *B* might be two fields in the same object that are frequently used together. This is a traditional technique to improve memory performance in both sequential and concurrent code, which now comes second to keeping separate data apart.

Third: Keep "hot" (frequently accessed) and "cold" (infrequently accessed) data apart. This is true even if the data is conceptually in the same logical object. This helps both to fit "hot" data into the fewest possible cache lines and memory pages and to avoid needlessly loading the "colder" parts. Together these effects reduce (a) the cache footprint and cache misses, and (b) the memory footprint and virtual memory paging.

Next Steps

Achieving parallel scalability involves two things:

1. Express it: Find the work that can be parallelized effectively.
2. Then don't lose it: Avoid visible and invisible scalability busters like the ones noted in this article.

We've seen some ways to avoid losing scalability unwittingly by invisibly adding contention. Next time, we'll consider one other important way we need to avoid invisibly adding contention and losing scalability: Choosing concurrency-friendly data structures, and avoiding concurrency-hostile ones. Stay tuned.

Notes

[1] H. Sutter. "Break Amdahl's Law!" (www.ddj.com/cpp/205900309)

[2] H. Sutter. "Super Linearity and the Bigger Machine" (www.ddj.com/architect/206903306).

Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Interrupt Politely

Stopping threads or tasks you no longer need is important for efficiency. But how do you do it?

By Herb Sutter
April 09, 2008
URL:http://drdobbs.com/high-performance-computing/207100682

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ standards committee. He can be contacted at www.gotw.ca.

We want to be able to stop a running thread or task when we discover that we no longer need or want to finish it. As we saw in the last two columns, in a simple parallel search we can stop other workers once one finds a match, and when speculatively running two alternative algorithms to compute the same result we can stop the longer-running one once the first finds a result. [1,2] Stopping threads or tasks lets us reclaim their resources, including locks, and apply them to other work.

But how do you stop a thread or task you longer need or want? Table 1 summarizes the four main ways, and how they are supported on several major platforms. Let's consider them in turn.

[Click image to view at full size]

	1. Kill	2. Tell, don't take no for an answer	3. Ask politely, and accept rejection	4. Set flag politely, let it poll if it wants
Tagline	Shoot first, check invariants later	Fire him, but let him clean out his desk	Tap him on the shoulder	Send him an email
Summary	A time-honored way to randomly corrupt your state and achieve unde-fined behavior	Interrupt at well-defined points and allow a handler chain (but target can't refuse or stop)	Interrupt at well-defined points and allow a handler chain, but request can be ignored	Target actively checks a flag – can be manual, or provided as part of #2 or #3
Pthreads	pthread_kill pthread_cancel (async)	pthread_cancel (deferred mode)	*n/a*	Manual
Java	Thread.destroy Thread.stop	*n/a*	Thread.interrupt	Manual, or Thread.interrupted
.NET	Thread.Abort	*n/a*	Thread.Interrupt	Manual, or Sleep(0)
C++0x	*n/a*	*n/a*	*n/a*	Manual
Guidance	**Avoid,** almost certain to corrupt transaction(s)	OK for languages without exceptions and unwinding...	Good, conveniently automated	Good, but requires more cooperative effort (can be a plus!)

Table 1: Major cancellation/interruption options.

Option 1: (Thou Shalt Not) Kill

The first option, which is nearly always wrong, is to kill the target thread or task immediately right in the middle of whatever it happens to be doing. This form of reckless slaughter is available in most platform APIs and frameworks, including the venerable UNIX *kill -9*, Pthreads' *pthread_kill* (or *pthread_cancel* in async mode), Java *Thread.destroy* or *Thread.stop*, and .NET's *Thread.Abort*.

Every major platform has reinvented this trap because it seems like a simple idea at first, until you realize it's nearly impossible to write correct code whose execution can be abruptly killed at arbitrary and unpredictable points.

The main trouble with Option 1 is that it is an extreme measure with extreme consequences: There's rarely such a thing as killing just one thread or task. Doing that is liable not only to stop that particular work, but also to corrupt the entire process and possibly other processes. Chances are, the thread will be partway through an operation where it's taking an object or data from one valid state to another. For example, data may be partly written into a buffer; or a money-transferring task may have taken money out of one account and not yet put it into the target account. Now mix in compiler optimizers, processors, and cache subsystems that routinely transform your code and execute it out of order, and you typically have no idea just from reading the source code what memory values might be read or written, and in what

orders, and therefore, no way to predict the consequences of interrupting that execution at a random point.

Killing a thread or task in the middle of doing some work usually means that we will leave behind state that has been corrupted, typically in an apparently random and unpredictable way; and/or we will lose resources the thread or task held, such as any locks it held.

Consider for a moment the specific issue of locks: If the killed thread was holding a lock, it's because it was using (and possibly changing) some data protected by that lock. Killing it in that state has two possible outcomes. First, on some platforms, the lock is released, which makes corrupted state visible to other parts of the program. Second, on other platforms, the lock is not released, which will deadlock any other parts of the program that are already waiting, or subsequently try to wait, for that same lock. Perhaps surprisingly, the second outcome is usually better, because at least it prevents the rest of the system from seeing the data that was left in a corrupt state. Of course, better still is not pulling the trigger and corrupting the data in the first place.

In short: Please, let's stop the slaughter. Option 1 is nearly always wrong because it is likely to corrupt at least the entire process, and might also corrupt other processes—including even processes on other machines if the killed thread was in the middle of performing some important I/O. Most of the time when someone tries to use *pthread_kill*, *Thread.stop*, and their ilk, the programmer is unaware of the extreme measure they're really signing up for. Be aware, and don't use it unless you really intend to take down the process or the machine without any attempt at graceful cleanup.

There are two use cases where Option 1 can be appropriate, one rare and one very rare:

- If you can prove that the target thread is doing nothing but reading memory and that it owns no resources, it may be safe to kill it.
- If you deliberately intend to terminate and restart the entire target process (not just thread) and possibly even the entire target machine, without even trying to clean up corrupted state, then killing may be appropriate. For example, in a system that uses three redundant and independent computers or processors that do not share data, when one misbehaves, it can be appropriate to kill and restart it in isolation.

Interlude: Cancellation/Interruption Points

Unlike Option 1, all of the three remaining alternatives share one vital point in common: The target thread or task can be stopped only at well-defined points in its execution, called "cancellation points" or "interruption points", which are typically when the thread is blocked doing one of the following things:

- Waiting to acquire a mutex, get a signal on a semaphore, or other synchronization.
- Joining with another thread or task.
- Sleeping.

Under Options 2, 3, and 4, these are the points at which a thread or task could be interrupted. This still imposes a burden on the author of the thread's or task's code: The code has to be ready to be interrupted at such points, and especially you have to either reestablish your invariants before you make any such calls at which you could be interrupted, or arrange for

the invariants to be reestablished if you are actually interrupted. Let's see what this looks like under the remaining three Options.

Option 2: Peremptory, Don't Take No For an Answer

Option 2 is to follow the model of POSIX threads (Pthreads) deferred cancellation via *pthread_cancel*: Wait until the target thread reaches its next well-defined cancellation point, then stop it and run the chain of cancellation handlers that the program installed (if any) which serves a similar purpose as destructor/dispose functions in modern languages. This is much better than Option 1.

The key drawback of Option 2 is that Pthreads cancellation requests cannot be ignored or caught; the target has no choice but to be stopped at its next cancellation point, and once cancellation has begun it cannot be stopped. This is a reasonable design for a language that does not have exceptions or objects with destructor/dispose functions (the cancellation handlers simulate the latter), but it is largely inappropriate for modern languages which have exception handling and know how to catch and recover from errors and continue correct execution. So Option 2 is appropriate in languages like C and Fortran if it is acceptable to force target threads to stop, but is less well suited for use with modern languages that have more sophisticated error-recovery mechanisms or your threads or tasks may legitimately want to handle the cancellation request and continue or ignore it entirely, neither of which is permitted under Option 2.

Option 3: Ask Politely

Option 3 is to follow the interruption model common to modern languages and frameworks, including Java and .NET, which spell it as *Thread.interrupt* and *Thread.Interrupt*, respectively. Like Option 2, the target thread continues to run until it reaches its next interruption point, at which point in most systems implementing Option 3 the interruption manifests as an exception thrown from the wait/join/sleep call. Then, unlike Option 2, the target thread can catch and handle the exception like any other exception, including that it has more options:

As in Pthreads, it can simply let destructors/disposers and *finally* clauses unwind the stack entirely and exit. Unlike Pthreads, the target thread can choose to unwind its stack partway until it finds a handler that catches and handles the exception, and then continue normal operations. Also unlike Pthreads, the target thread can immediately catch and ignore the exception entirely.

This is polite interruption, the state of the art in automated interruption facilities.

Option 4: Cooperate

Finally, Option 4, which you can and should use together with Option 3, is a fully cooperative model where the target thread can check to see whether someone has asked for it to interrupt work. This checking can be in between between interruption points (if you want to use both Options 3 and 4 together), or instead of interruption points (if you want to use Option 4 alone). We saw Option 4 in action in the previous two columns [1,2]: In a simple parallel search, once one worker finds an answer and records it in a shared location, the other

workers can periodically check that shared location and stop their own work when they see that someone else has already found the answer.

What About Library/OS Calls?

What should you do about library calls that are not interruptible? If it doesn't cooperate, it doesn't cooperate. Don't shoot! Violence is not the answer.

What should you do if you need to call an OS (possibly kernel-mode) function that isn't interruptible? The answer is the same: If it doesn't cooperate, it doesn't cooperate. Don't shoot. Incidentally, you may have noticed a recent trend: More recent operating systems are on the road to making all calls interruptible. For example, in Windows Vista, nearly all file and I/O APIs support interruption, so that you can stop them without just waiting for them to return. This shouldn't be surprising, since we've been considering the importance of interruption in concurrent code.

Summary

Interrupt politely. Always use Options 3 and 4, which allow the thread or task to participate in the decision about whether and how it should clean up its work and/or continue on. Notify a thread of interruption requests only at well-defined predictable wait/join/sleep points, and make sure you write your code to be safe if interruption does happen at those points. Note that both Options 3 and 4 provide a strict superset of what is possible in Option 2: Anything you can code in Option 2, you can code in Option 3 or 4 as well. Avoid the peremptory Option 2 of not letting the thread participate in the decision. Even if you are running on Pthreads which does not support Option 3, you have the option of writing Option 4 yourself.

Finally, never kill a thread or task as in Option 1, unless you can prove you're truly in one of the rare cases where this questionable practice is safe and it's okay to take down the whole process (or more) without any graceful cleanup at all. Most real-world attempts to kill a thread at arbitrary points are indefensible; every major threading library or environment started here, but now we know better—violence is not the answer.

Notes

[1] H. Sutter. "Going Superlinear" (*Dr. Dobb's Journal*, March 2008).

[2] H. Sutter. "Super Linearity and the Bigger Machine" (*Dr. Dobb's Journal*, March 2008).

Use Lock Hierarchies to Avoid Deadlock

Need to avoid deadlock in the code you control? Try using lock hierarchies.

By Herb Sutter
December 11, 2007
URL:http://drdobbs.com/high-performance-computing/204801163

Herb is a software architect at Microsoft and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

In the first two Effective Concurrency columns—"The Pillars of Concurrency" (*DDJ*, August 2007) and "How Much Scalability Do You Have or Need?" (*DDJ*, September 2007)—we saw the three pillars of concurrency and what kinds of concurrency they express:

- Pillar 1: Isolation via asynchronous agents. This is all about doing work asynchronously to gain isolation and responsiveness, using techniques like messaging. Some Pillar 1 techniques, such as pipelining, also enable limited scalability, but this category principally targets isolation and responsiveness.
- Pillar 2: Scalability via concurrent collections. This is all about reenabling the free lunch of being able to ship applications that get faster on machines with more cores, using more cores to get the answer faster by exploiting parallelism in algorithms and data.
- Pillar 3: Consistency via safely shared resources. While doing all of the above, we need to avoid races and deadlocks when using shared objects in memory or any other shared resources.

My last few columns have focused on Pillar 3, and I've referred to lock levels or lock hierarchies as a way to control deadlock. A number of readers have asked me to explain what those are and how they work, so I'll write one more column about Pillar 3 (for now).

The Problem: Anatomy of a Deadlock

Your program contains a potential deadlock if:

- One part of your program tries to acquire exclusive use of two shared resources (such as mutexes) *a* and *b* at the same time by acquiring first *a* and then *b.*
- Some other part of your program tries to do the same by acquiring *b* and then *a* in the reverse order.
- The two pieces of code could ever execute concurrently.

For convenience, from now on I'm going to talk about just locks, but the issues and techniques apply to any shared resource that needs to be used exclusively by one piece of code at a time. The following code shows the simplest example of a potential deadlock:

```
// Thread 1: a then b   // Thread 2: b then a
a.lock();                 b.lock();
b.lock();                 a.lock();
 ...                       ...
// unlock a and b     // unlock a and b
//   in either order  //   in either order
```

The only way to eliminate such a potential deadlock is to make sure that all mutexes ever held at the same time are acquired in a consistent order. But how can we ensure this in a way that will be both usable and correct? For example, we could try to figure out which groups of mutexes might ever be held at the same time, and then try to define pairwise ordering rules that cover each possible combination. But that approach by itself is prone to accidentally missing unexpected combinations of locks; and even if we did it perfectly, the result would still be at best "DAG spaghetti"—a directed acyclic graph (DAG) that nobody could comprehend as a whole. And every time we want to add a new mutex to the system, we would have to find a way fit it into the DAG without creating any cycles.

We can do better by directly exploiting the knowledge we already have about the structure of the program to regularize the mess and make it understandable.

A Solution: Lock Hierarchies and Layering

The idea of a lock hierarchy is to assign a numeric level to every mutex in the system, and then consistently follow two simple rules:

- Rule 1: While holding a lock on a mutex at level *N*, you may only acquire new locks on mutexes at lower levels <*N*.
- Rule 2: Multiple locks at the same level must be acquired at the same time, which means we need a "lock-multiple" operation such as *lock(mut1, mut2, mut3, ...).* This operation internally has the smarts to make sure it always takes the requested locks in some consistent global order. [1] Note that any consistent order will do; for example, one typical strategy is to acquire mutexes at the same level in increasing address order.

If the entire program follows these rules, then there can be no deadlock among the mutex acquire operations, because no two pieces of code can ever try to acquire two mutexes *a* and *b* in opposite orders: Either *a* and *b* are at different levels and so the one at the higher level must be taken first; or else they are at the same level and they must be requested at the same time, and the system will automatically acquire them in the same order. The two simple rules have provided a convenient and understandable way to conveniently express a total order on all locking performed in the system.

But where do we find the levels? The answer is: You probably already have them. Mutexes protect data, and the data is already in layers.

Lock levels should directly leverage and mirror the layering already in place in the modular structure of your application. Figure 1 illustrates a typical example of layering (or "hierarchical decomposition" and "into a directed acyclic graph," if you prefer five-dollar words), a time-tested technique to control the dependencies in your software. The idea is to group your code into modules and the modules into layers, where code at a given layer can only call code at the same or lower layers, and should avoid calling upward into higher layers.

[Click image to view at full size]

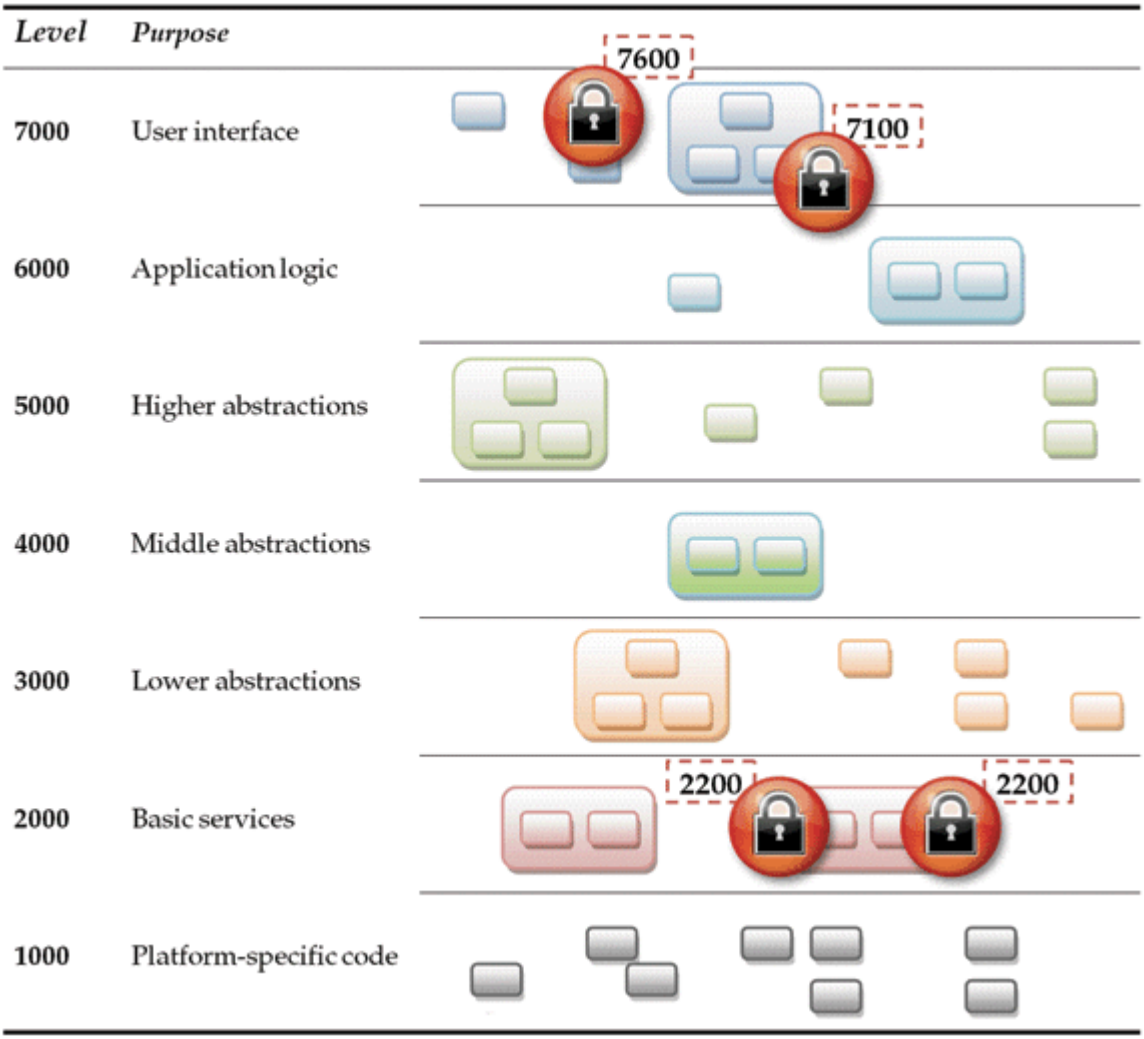

Figure 1: Sample module/layer decomposition.

If that sounds a lot like the Two Rules of lock hierarchies, that's no coincidence. After all, both the layering and the mutexes are driven by the same goal: to protect and control access to the encapsulated data that is owned by each piece of code, and to keep it free from corruption by maintaining its invariants correctly. As in Figure 1, the levels you assign to mutexes will normally closely follow the levels in your program's layered structure. A direct consequence of Rule 1 is that locks held on mutexes at lower levels have a shorter duration than locks held at higher levels; this is just what we expect of calls into code at lower layers of a layered software system.

Software can't always be perfectly layered, but exceptions should be rare. After all, if you can't define such layers, it means that there is a cycle among the modules somewhere that includes code in what should be a lower level subsystem calling into higher level code somewhere, such as via a callback, and you have the potential for reentrancy even in single-threaded code. And remember, reentrancy is a form of concurrency, so the program can observe corrupt state even in single-threaded code. If higher level code is in the middle of taking the system from one valid state to another, thus temporarily breaking some invariant, and calls into lower level code, the trouble is that if that call could ultimately call back into the higher level code it might see the broken invariant. Layering helps to solve this single-

threaded concurrency problem for the same reasons it helps to solve the more general multithreaded version.

Frameworks and Lock Hierarchies

It is a curious thing that major frameworks that supply mutexes and locks do nothing to offer any direct support for lock hierarchies. Everyone is taught that lock hierarchies are a best practice, but then are generally told to go roll their own.

The frameworks vendors will undoubtedly fix this little embarrassment in the future, but for now, here's a useful recipe to follow as you do roll your own level-aware mutex wrapper. You can adapt this simple sketch to your project's specific needs (for example, to suit details such as whether your *lock* operation is a method or a separate class):

- Write a wrapper around each of your favorite language- or platform-specific mutex types, and let the wrapper's constructor(s) take a level number parameter that it saves in a *myLevel* member. Use these wrappers everywhere. (Where practical, save time by making the wrapper generic—as a C++ template, or a Java or .NET generic—so that it can be instantiated to wrap arbitrary mutex types that have similar lock/unlock features. You might only have to write it once.)
- Give the wrapper class a thread-local static variable called *currentLevel*, initialized to a value higher than any valid lock level.
- In the wrapper's *lock* method (or similar), assert that *currentLevel* is greater than *myLevel*, the level of the mutex that you're about to try to acquire. Remember, if the previous value of *currentLevel* is using another member variable, then set *currentLevel = myLevel*; and acquire the lock.
- In the wrapper's *unlock* method (or similar), restore the previous value of *currentLevel*.
- As needed, also wrap other necessary methods you need to be able to use, such as *try_lock*. Any of these methods that might try to acquire the lock should do the same things as *lock* does.
- Finally, write a "lock-multiple" method *lock(m1, m2, ...)* that takes a variable number of lockable objects, asserts that they are all at the same level, and locks them in their address order (or their GUID order, or some other globally consistent order).

The reason for using assertions in the *lock* methods is so that, in a debug build, we force any errors to be exposed the first time we execute the code path that violates the lock hierarchy rules. That way, we can expect to find violations at test time and have high confidence that the program is deadlock-free based on code path coverage. Enabling such deterministic test-time failures is a great improvement over the way concurrency errors usually manifest, namely as nondeterministic runtime failures that can't be thoroughly tested using code path coverage alone. But often our test-time code path coverage isn't complete, either because it's impossible to cover all possible code path combinations or because we might forget a few cases; so prefer to also perform the tests in release builds, recording violations in a log or diagnostic dump that you can review later if a problem does occur.

A Word About Composability

Although lock hierarchies address many of the flaws of locks, including the possibility of deadlock, they still share the same Achilles Heel: Like locks themselves, lock hierarchies are not in general composable without some extra discipline and effort. After all, just because

you use a lock hierarchy discipline correctly within the code you control, a separately authored module or plug-in that you link with won't necessarily know anything about your lock hierarchy unless you somehow inform them about your layers and how they should fit into the hierarchy.

For example, look at the sample application architecture in Figure 1 again, and consider: What if the application wants to allow plugins that are called from the 5000s layer? First, of course, the program and all the plug-ins should make every effort to avoid calling unknown code (in this case, each other) while holding a lock. But, as we saw last month ("Avoid Calling Unknown Code While Inside a Critical Section", *DDJ*, December 2007), we can encounter trouble even if a plug-in takes no locks of its own but may call back into the main program and create a code path that unexpectedly calls up into higher level code that takes a higher level lock. So the plug-ins must be aware of the layering of the application and be told that they are to operate at (say) level 4999, and may only call APIs below level 4999.

Summary

Keep it on the level: Use lock hierarchies to avoid deadlock in the code you control. Assign each shared resource a level that corresponds to its architectural layer in your application, and follow the two rules: While holding a resource at a higher level, acquire only resources at lower levels; and acquire multiple resources at the same level all at once.

If your program will call external code, especially plug-ins, then document your public API sufficiently for plug-in authors to see what level their plug-in is expected to operate at, and therefore the API calls they can and can't make (those below their level and those at or above their level, respectively).

Next month, we'll start to look into issues and techniques for writing scalable manycore applications. Stay tuned.

Notes

[1] Or otherwise gets the same effect. For example, it is possible to just start trying to acquire the locks in some randomly selected order using try_lock operations, and if we can't acquire them all just back-off (unlock the ones already acquired) and try a different order until we find one that works. Surprisingly, this can be more efficient than taking the locks in a hard-coded global order, although any backoff-and-retry strategy has to take care that it doesn't end up prone to livelock problems in-stead. But we can leave all this to the implementer; the key is that the programmer simply writes *lock(/* whatever */)* and is insulated from the details of determining the best way to keep the order consistent.

Avoid Calling Unknown Code While Inside a Critical Section

The good news is that today's software is built on modular, composable software. The bad news is that locks, and other forms of synchronization, aren't.

By Herb Sutter
November 06, 2007
URL:http://drdobbs.com/cpp/202802983

Herb is a software architect at Microsoft and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

Our world is built on modular, composable software. We routinely expect that we don't need to know about the internal implementation details of a library or plug-in to be able to use it correctly.

The problem is that locks, and most other kinds of synchronization we use today, are not modular or composable. That is, given two separately authored modules, each of which is provably correct but uses a lock or similar synchronization internally, we generally can't combine them and know that the result is still correct and will not deadlock. There are only two ways to combine such a pair of lock-using modules safely:

- Option 1. Each module must know about the complete internal implementation of any function it calls in the other module (generally impossible for code you don't control).
- Option 2. Each module must be careful not to call into the other module while the calling code is inside a critical section (while holding a lock, for example).

Let's examine why Option 2 is generally the only viable choice, and what consequences it has for how we write concurrent code. For convenience, I'm going to cast the problem in terms of locks, but the general case arises whenever:

- The caller is inside one critical section.
- The callee directly or indirectly tries to enter another critical section, or performs another blocking call.
- Some other thread could try to enter the two critical sections in the opposite order.

Quick Recap: Deadlock

A deadlock (or deadly embrace) can happen anytime two different threads can try to acquire the same two locks (or more generally, acquire exclusive use of the resources protected by the same two synchronization objects) in opposite orders. Therefore, anytime you write code where a thread holds one lock *L1* and tries to acquire another lock *L2*, that code is liable to be one arm of a potential deadly embrace—unless you can eliminate the possibility that some other thread might try to acquire the locks in the other order.

We can use techniques such as lock hierarchies to guarantee that locks are taken in order. Unfortunately, those techniques do not compose either. For example, you can use a lock hierarchy and guarantee that the code you control follows the discipline, but you can't guarantee that other people's code will know anything about your lock levels, much less follow them correctly.

Example: Two Modules, One Lock Each

One fine day, you decide to write a new web browser that lets users write plug-ins to customize the behavior or rendering of individual page elements. Consider the following possible code, where we simply protect all the data structures representing elements on a given page using a single mutex *mutPage*:

```
// Example 1: Thread 1 of a potential deadly embrace
//
class CoreBrowser {
  ... other methods ...
  private void RenderElements() {
    mutPage.lock();  // acquire exclusion on the page elements
    try {
      for( each PageElement e on the page ) {
        DoRender( e );       // do our own default processing
        plugin.OnRender( e ); // let the plug-in have a crack at it
      }
    } finally {
      mutPage.unlock();        // and then release it
    }
  }
}
```

Do you see the potential for deadlock? The trouble is that if inside the call to *plugin.OnRender* the plug-in might acquire some internal lock of its own, which could be one arm of a potential deadly embrace. For example, consider this plug-in implementation that does some basic instrumentation of how many times certain actions have been performed and protects its internal data with a single mutex *mutMyData*:

```
class MyPlugin {
   ... other methods ...
  public void OnRender( PageElement e ) {
    mutMyData.lock();  // acquire exclusion on some internal shared data
    try {
      renderCount[e]++;        // update #times e has been rendered
    } finally {
      mutMyData.unlock(); // and then release it
```

```
    }
  }
}
```

Thread 1 can therefore acquire *mutPage* and *mutMyData* in that order. Thread 1 is potential deadlock-bait, but the trouble will only manifest if some other Thread 2 that could run concurrently with the aforementioned code performs something like the following:

```
// Example 2: Thread 2 of a potential deadly embrace
//
class MyPlugin {
   ... other methods ...
  public void RefreshDisplay( PageElement e ) {
    mutMyData.lock();  // acquire exclusion on some internal shared data
    try {              // display stuff in a debug window
      for( each element e we've counted ) {
        listRenderedCount.Add( e.Name(), renderCount[e] );
      }
      textHiddenCount = browser.CountHiddenElements();
    } finally {
      mutMyData.unlock();  // and then release it
    }
  }
}
```

Notice how the plugin calls code unknown to it, namely *browser.CountHiddenElements*? You can probably see the trouble coming on like a steamroller:

```
class CoreBrowser {
  ... other methods ...
  public int CountHiddenElements() {
    mutPage.lock();  // acquire exclusion on the page elements
    try {
      int count = 0;
      for( each PageElement e on the page ) {
        if( e.Hidden() ) count++;
      }
      return count;
    } finally {
      mutPage.unlock();       // and then release it
    }
  }
}
```

Threads 1 and 2 can therefore acquire *mutPage* and *mutMyData* in the opposite order, so this is a deadlock waiting to happen if Threads 1 and 2 can ever run concurrently. For added fun, note that each mutex is purely an internal implementation detail of its module that is never exposed in the interface; neither module knows anything about the internal lock being used within the other. (Nor, in a better programming world than the one we now inhabit, should it have to.)

Example: Two Modules, But Only One Has Locks

Note that this kind of thing can happen even if both locks are in the same module, but control flow passes through another module so that you don't know what locks are taken. Consider the following modification, where the browser protects each page element using a separate mutex, which can be desirable to allow different parts of the page to be rendered concurrently:

```
// Example 3: Thread 1 of an alternative potential deadly embrace
//
class CoreBrowser {
  ... other methods ...
  private void RenderElements() {
    for( each PageElement e on the page ) {
      e.mut.lock();  // acquire exclusion on this page element
      try {
        DoRender( e ); // do our own default processing
        plugin.OnRender( e ); // let the plug-in have a
                              // crack at it
      } finally {
        e.mut.unlock();       // and then release it
      }
    }
  }
}
```

And consider a plug-in that does no locking of its own at all:

```
class MyPlugin {
  ... other methods ...
  public void OnRender( PageElement e ) {
    GuiCoord cPrev = browser.GetElemPosition( e.Previous() );
    GuiCoord cNext = browser.GetElemPosition( e.Next() );
    Use( e, cPrev, cNext ); // do something with the coords
  }
}
```

But which calls back into:

```
class CoreBrowser {
  ... other methods ...
  public GuiCoord GetElemPosition( PageElement e2 ) {
    e2.mut.lock();  // acquire exclusion on this page element
    try {
```

```
      return FigureOutPositionFor( e2 );
    } finally {
      e2.mut.unlock(); // and then release it
    }
  }
}
```

The order of mutex acquisition is:

```
for each element e
  acquire e.mut
    acquire e.Previous().mut
    release e.Previous().mut
    acquire e.Next().mut
    release e.Next().mut
  release e.mut
```

Perhaps the most obvious issue is that any pair of locks on adjacent elements can be taken in both orders by Thread 1; so this cannot possibly be part of a correct lock hierarchy discipline.

Because of the interference of the plug-in code, which does not even have any locks of its own, this code has a latent deadlock if any other concurrently running thread (including perhaps another instance of Thread 1 itself) can take any two adjacent elements' locks in any order. The deadlock-proneness is inherent in the design, which fails to guarantee a rigorous ordering of locks. In this respect, the original Example 1 was better, even though its locking was coarse-grained and less friendly to concurrent rendering of different page elements.

Consequences: What Is "Unknown Code"?

It's one thing to say "avoid calling unknown code while holding a lock" or while inside a similar kind of critical section. It's another to do it, because there are so many ways to get into "someone else's code." Let's consider a few. While inside a critical section, including while holding a lock:

- **Avoid calling library functions.** A library function is the classic case of "someone else's code." Unless the library function is documented to not take any locks, deadlock problems can arise.
- **Avoid calling plug-ins.** Clearly, a plug-in is "someone else's code."
- **Avoid calling other callbacks, function pointers, functors, delegates, and so on.** C function pointers, C++ functors, C# delegates, and the like can also fairly obviously lead to "someone else's code." Sometimes, you know that a function pointer, functor, or delegate will always lead to your own code, and calling it is safe; but if you don't know that for certain, avoid calling it from inside a critical section.
- **Avoid calling virtual methods.** This may be less obvious and quite surprising, even Draconian; after all, virtual functions are common and pervasive. But every virtual function is an extensibility point that can lead to executing code that doesn't even exist today. Unless you control all derived classes (for example, the virtual function is internal to your module and not exposed for others to derive from), or you can somehow enforce or document that overrides must not take locks, deadlock problems can arise if it is called while inside a critical section.

- **Avoid calling generic methods, or methods on a generic type.** When writing a C++ template or a Java or .NET generic, we have yet another way to parameterize and intentionally let "someone else's code" be plugged into our own. Remember that anything you call on a generic type or method can be provided by anyone, and unless you know and control all the types with which your template or generic can be instantiated, avoid calling something generic from within a critical section.

Some of these restrictions may be obvious to you; others may be surprising at first.

Avoidance: Noncritical Calls

So you want to remove a call to unknown code from a critical section. But how? What can you do? Four options are: (a) move the call out of the critical section if you didn't need the exclusion anyway; (b) make copies of data inside the critical section and later pass the copies; (c) reduce the granularity or power of the critical section being held at the point of the call; or (d) instruct the callee sternly and hope for the best.

We can apply the first approach directly to Example 2. There is no reason the plugin needs to call *browser.CountHiddenElements*() while holding its internal lock. That call should simply be moved to before or after the critical section.

The second approach is to pass copies of data, which solves the correctness problem at the expense of space and performance. Variants of this approach include passing a subset of the data, and passing the copies via messages to run the callee asynchronously.

To improve Example 1, for instance, it might be appropriate to change the *RenderElements* method to hold the lock only long enough to take copies of the necessary shared information in a local container, then doing processing outside the lock, passing the copied elements. (This could be inappropriate if the data is very expensive to copy, or the callee needs to work on the real data.) Alternatively, perhaps the callee doesn't really need all the information it gets from being given direct access to the protected object, and it would be both sufficient and efficient to pass copies of just those parts of the data the callee does need.

The third option is to reduce the power or granularity of the critical section, which implicitly trades off ease-of-use because making your synchronization finer-tuned and/or finer-grained also makes it harder to code correctly. One example of reducing the power of the critical section is to replace a mutex with a reader-writer mutex so that multiple concurrent readers are allowed; if the only deadlocks could arise among threads that are only performing reads of the protected resources, then this can be a valid solution by enabling the use of a read-only lock instead of a read-write lock. And an example of making the critical section finer-grained is to replace a single mutex protecting a large data structure with mutexes protecting parts of the structure; if the only deadlocks possible are among threads that use different parts of the structure, then this can be a valid solution (Example 1 is not such a case).

The fourth option is to tell the callee not to block, which trades off enforceability. In particular, if you have the power to impose requirements on the callee (as you do with plug-ins to your software, but not with simple calls into existing third-party libraries), then you can require them to not take locks or otherwise perform blocking actions. Alas, these requirements are typically going to be limited to documentation, and are typically not

enforceable automatically. Tell the callee what (not) to do, and hope he follows the yellow brick road.

Summary

Be aware of the many opportunities modern languages give us to call "someone else's code," and eliminate external opportunities for deadlock by not calling unknown code from within a critical section. If you additionally eliminate internal opportunities for deadlock by applying a lock hierarchy discipline within the code you control, your use of locks will be highly likely to be correct...and we'll consider lock hierarchies next month. Stay tuned.

Apply Critical Sections Consistently

Source Code Accompanies This Article. Download It Now.

- critical.txt

Critical sections are our One True Tool for guaranteeing mutual exclusion on shared variables.

By Herb Sutter
October 10, 2007
URL:http://drdobbs.com/high-performance-computing/202401098

Herb is a software architect at Microsoft and chair of the ISO C++ Standards committee. He can be contact

---

The critical section is our One True Tool for guaranteeing mutual exclusion on mutable shared variables. Tab express exclusive critical sections (see last month's column [1] for details). A useful way to think about the sy chain of these "release-acquire" handoffs that stitch all the threads' critical sections together into some linear cumulative work done and is "released" by all the others that preceded it.

Like most tools, these must be applied consistently, and with the intended meanings. A program must ensure properly protected using exactly one of these mechanisms at any given point in its lifetime. Chaos can erupt i (e.g., trying to abuse taking a lock or reading an atomic variable as a "critical section exit" operation; see Exa synchronization techniques at the same time.

Let's consider some examples to illustrate the proper and improper uses of critical sections, to get us used to

synchronization points should appear in the code.

Synchronization Type	To Enter a Critical Section	To Exit a Critical Section	Notes
Locks	Acquire lock	Release lock	Preferr
Condition variables	Wait for cv	Notify cv	Useful v middle
Semaphores	Acquire semaphore	Release semaphore	Use jud
Ordered atomics (e.g., Java/.NET volatile, C++0x atomic)	Read from variable *	Write to variable **	Use jud
Unordered atomics and explicit fences (e.g., aligned integers and Linux mb or Win32 MemoryBarrier)	Read from variable followed by fence***	Fence followed by write to variable***	Avoid,

* *or equivalent, such as calling a compare-and-swap function having at least acquire semantics*

** *or equivalent, such as calling a compare-and-swap function having at least release semantics*

*** *when in doubt, use a full fence*

Table 1: Common ways to express critical sections.

Example 1: One Producer, Many Consumers, Using Locks

Imagine we have a single Producer thread that produces a number of tasks for Consumer threads to perform. *queue*, and finally pushes a special *done* sentinel to mark the end of the work. The *queue* object is protected a Producer holds only as long as necessary for a single *push*, so that Consumers can start executing their tasks efficiency, to avoid making the Consumers busy-wait whenever the queue is initially or temporarily empty, t *cv* every time something has been added to the queue:

```
// One Producer thread
//
while( ThereAreMoreTasks() ) {
  task = AllocateAndBuildNewTask();
  mut.lock();          // enter critical section
```

```
  queue.push( task );  // add new task
  mut.unlock();// exit critical section
  cv.notify();
}
mut.lock();            // enter critical section
queue.push( done );    // add sentinel value; that's all folks
mut.unlock();  // exit critical section
cv.notify();
```

On the consumption side, the Consumer threads each pull individual tasks off the completed queue. Here, *my*
Consumer:

```
// K Consumer threads
//
myTask = null;
while( myTask != done ) {
  mut.lock();  // enter critical section
  while( queue.empty() )       // if nothing's there, to avoid busy-waiting we'll
    cv.wait( mut );    // release and reenter critical section
  myTask = queue.first();      // take a copy of the task
  if( myTask != done )         // remove it from queue unless it was the sentinel,
    queue.pop();       // which we want to leave there for others to see
  mut.unlock();// exit critical section
  if( myTask != done )
    DoWork( myTask );
}
```

All of these critical sections are easy to see. This code simply protects the data structure using the same mute
check that you did it right—just look to make sure the code only refers to *queue* when inside some critical se

The primary expression of critical sections is the explicit locking done on the mutex. The condition variable i
lets us express an efficient wait point in the middle of a locked section so that the Consumer race cars don't h
tires (in this case, their *lock/unlock* loop) while waiting for the Producer to give them the green light.

Example 2: One Producer, Many Consumers, Using Lock-Free Mailboxes

Next, let's consider a similar example, but instead of locks ,we'll use ordered atomic variables to synchronize
synchronization at all for interConsumers contention.

The idea in this second approach is to use a set of ordered atomic variables (e.g., Java/.NET *volatile*, propose
will serve as "mailbox" slots for the Consumers individually. Each mail slot can hold exactly one item at a ti
he puts it in the next mailbox that doesn't already contain an item using an atomic write; this lets the Consum
check their slots and perform work while the Producer is still manufacturing and assigning later tasks. For fu
busy-wait whenever its slot is initially or temporarily empty, we'll also give each Consumer a semaphore *sem*
a task into the Consumer's mail slot.

Finally, once all the tasks have been assigned, the Producer starts to put "done" dummy tasks into mailboxes
mailbox has received one. Here's the code:

```
// One Producer thread: Changes any box from null to nonnull
```

```
//
curr = 0;  // keep a finger on the current mailbox

// Phase 1: Build and distribute tasks (use next available box)
while( ThereAreMoreTasks() ) {
  task = AllocateAndBuildNewTask();
  while( slot[curr] != null )  // acquire null: look for next
        curr = (curr+1)%K;            // empty slot
  slot[curr] = task;           // release nonnull: "You have mail!"
  sem[curr].signal();
}

// Phase 2: Stuff the mailboxes with"done" signals
numNotified = 0;
while( numNotified < K ) {
  while( slot[curr] == done )  // acquire: look for next not-yet-
       curr = (curr+1)%K;             // notified slot
  if( slot[curr] == null ) {          // acquire null: check that the
                                      // mailbox is empty
    slot[curr] = done;         // release done: write sentinel
    sem[curr].signal();
    ++numNotified;
  }
}
```

On the other side of the mailbox wall, each Consumer checks only its own mailbox. If its slot holds a new tas slot to "empty," and then processes the task. If the mailbox holds a "done" dummy task, the Consumer knows

```
// K Consumer threads (mySlot = 0..K-1):
// Each changes its own box from non-null to null
//
myTask = null;
while( myTask != done ) {
  while( (myTask = slot[mySlot]) == null )    // acquire nonnull,
    sem[mySlot].wait();        // without busy-waiting
  if( myTask != done ) {
    slot[mySlot] = null;       // release null: tell
                               // him we took it
    DoWork( myTask );
  }
}
```

A few notes: First, you can easily generalize this to replace each slot with a bounded queue; the previous exa maximum size of 1. Second, this Producer code can needlessly busy-wait when it's trying to write a new task no free slots. This can be addressed by adding another semaphore, which

the Producer waits for when it has no free slots and that is signaled by each Consumer when it resets its slot t the example clearer.

In Example 2, it's not quite as easy to check that we did it right as it was when we used locks back in Exampl convince ourselves that it is correct. First, note how each Consumer's read of its slot "acquires" whatever valu Producer's read of each slot "acquires" whatever value the corresponding Consumer last wrote there. We avo owned by the Producer (only the Producer can change a slot from null to non-null); and while it's nonnull, it's

the Consumer can change its slot from nonnull to null).

Example 3: Don't Try to Turn Critical Sections Inside Out

Although every entry/exit of a critical section implies a use of locks or atomics, not every use of locks or ato
Consider this evil, smelly code of questionable descent, where *x* is initially zero (Note: this example is drawn

```
// Thread 1
x = 1;  // a
mut.lock();    // b
... etc. ...
mut.unlock();

// Thread 2: wait for Thread 1 to take the lock, then use x
while( mut.trylock() ) // try to take the lock...
  mut.unlock();        // ... if we succeeded, unlock and loop
r2 = x;        // not guaranteed to see the value 1!
```

This programmer is certainly nobody we would know or associate with. But what is he doing? He is trying to
operation, and is visible in principle to all threads.

Specifically, a thread can use a *trylock*-like facility to find out whether some other thread currently holds the
that attempt succeeded or failed. Pretty much every threading library has a way to determine if someone else
you can use *Lock.tryLock*; on Windows and .NET, there's *Monitor.TryEnter* or *WaitOne* with a timeout; in pr
pthreads, *pthread_mutex_trylock* and *pthread_mutex_timedlock*.

Thread 2, which we might now refer to as Machiavelli, doesn't really want the lock. Machiavelli only wants t
room, listening with a *trylock* glass against the wall until he hears the telltale sound that means Thread 1 got t
set *x*.

The most obvious red flag in this code is that the read and write of *x* are outside critical sections; that is, whil
only works when there's enough synchronization to hand off an object from one thread to another (it belongs
another thread after the synchronization), and there isn't enough synchronization here to do that. Let's see wh

The problem is that this anti-idiom is trying to abusively invert the usual meanings of a locked section. Threa
entering a critical section as a release event, which it isn't. Remember, entering a critical section is an acquire
release event. On this, the whole world depends, as we saw last month [1]. In particular, compiler and proces

normal critical section boundaries, and therefore not change the semantics of correctly synchronized code; sp
be reordered to move into, but not out of, a critical section. In the context of Thread 1, that means an optimiz
after line *b*... and therefore, Thread 2 could well see a value of 0 for *x*, despite its attempts to abuse the global

In reality, the code may happen to work all the time on a given system that doesn't happen to reorder lines *a* a
gone by the time some other poor sod tries to port it to a new compiler or platform that does manifest the race
the intermittent problem over a holiday weekend. (Note that, besides the issues we've discussed, this code has
2 can wait forever if Thread 1's lock is taken too early.)

Treat a critical section as you would treat another person: Turning either inside out is at least cruel, and usual
unsavory details, see [3].

Summary

Apply critical sections consistently to protect shared objects: Enter a critical section by taking a lock or readi
critical section by releasing a lock or writing to an ordered *atomic* variable. Never pervert these meanings; in
a lock acquire in another thread act like the end of a critical section.

Notes

[1] H. Sutter. "Use Critical Sections (Preferably Locks) to Eliminate Races," *Dr. Dobb's Journal*, October 20

[2] H. Sutter. "The Trouble With Locks," *C/C++ Users Journal*, March 2005. Available at http://gotw.ca/pul

[3] H. Boehm. "Reordering Constraints for Pthread-Style Locks," *Proceedings of the 12th ACM SIGPLAN Sy
Parallel Programming (PPoPP'07)*, March 2007.

volatile vs. volatile

A tale of two similar but different tools

By Herb Sutter

January 08, 2009
URL:http://drdobbs.com/high-performance-computing/212701484

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

What does the **volatile** keyword mean? How should you use it? Confusingly, there are two common answers, because depending on the language you use volatile supports one or the other of two different programming techniques: lock-free programming, and dealing with 'unusual' memory. (See Figure 1.)

[Click image to view at full size]

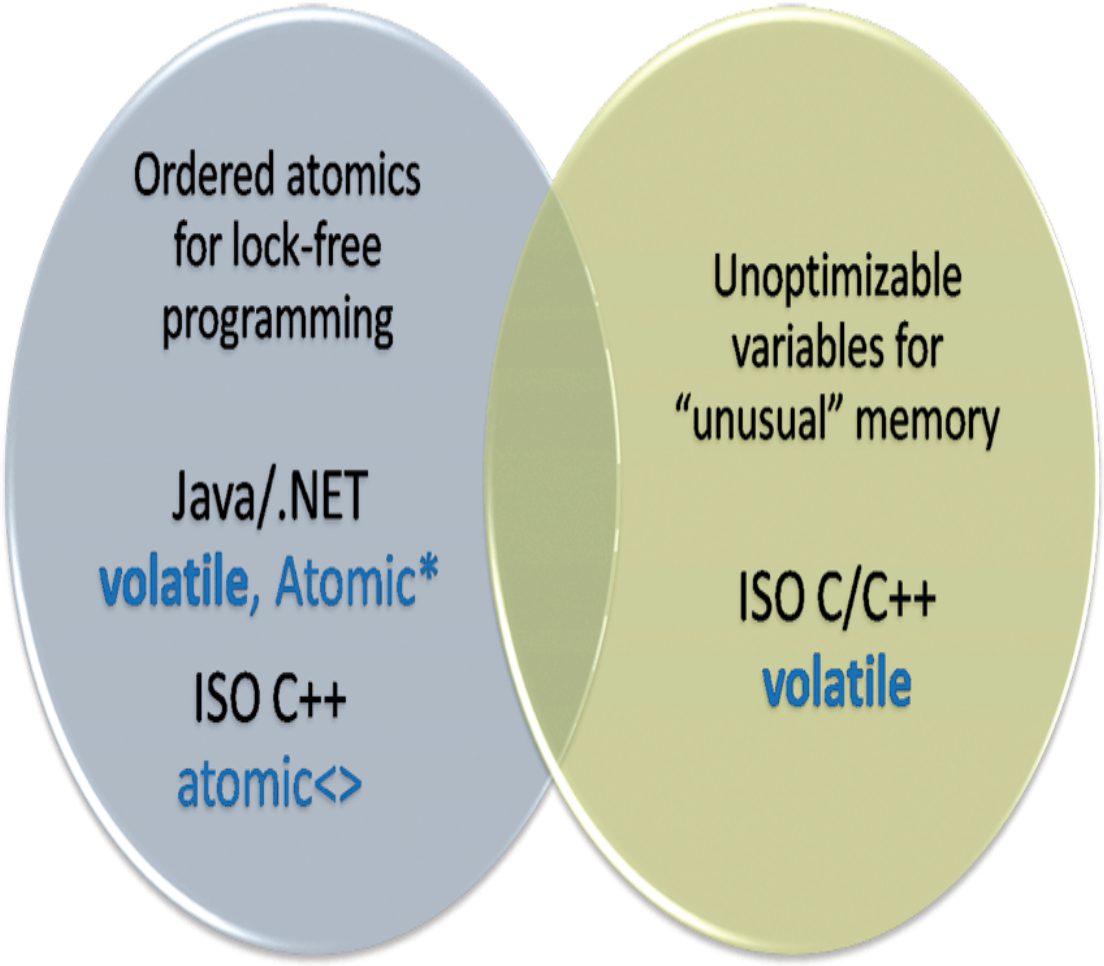

Figure 1: A tale of two requirements.

Adding to the confusion, these two different uses have overlapping requirements and impose overlapping restrictions, which makes them appear more similar than they are. Let's define and understand them clearly, and see how to spell them correctly in C, C++, Java and C# -- and not always as **volatile**.

[Click image to view at full size]

	I. Lock-free programming → Ordered atomic variable (Java/.NET "volatile T", ISO C++ atomic<T>)	II. Unusual memory semantics (e.g,. HW register, setjmp-safety, memory at more than one address) → Unoptimizable variable (ISO C/C++ "volatile")
Atomicity: Are special loads and stores guaranteed to be all-or-nothing?	**Yes**, either for types T up to a certain size (Java and .NET) or for all T (ISO C++)	**No**, in fact sometimes they cannot be naturally atomic (e.g., HW registers that must be unaligned or larger than CPU's native word size)
Optimizing ordinary memory operations: Can ordinary memory operations be reordered across these special ones?	**Some (1)**: in one direction only, down across an ordered atomic load or up across an ordered atomic store	**Some (2)**: one reading of the standard is that ordinary loads may be reordered across a volatile load or store, but not ordinary writes
Optimizing these special operations: Can these special loads and stores themselves be reordered/ invented/ elided?	**Some** optimizations are allowed, such as combining two adjacent stores to the same location	**No** optimization possible; the compiler is not allowed to assume it knows anything about the type, not even "v = 1; r1 = v;" -> "v = 1; r1=1;"

Table 1: Comparing the overlapping but different requirements.

Case 1: Ordered Atomic Variables For Lock-Free Programming

Lock-free programming is all about doing communication and synchronization between threads using lower-level tools than mutex locks. In the past and even today, however, these tools are all over the map. In rough historical order, they include explicit fences/barriers (e.g., Linux's **mb()**), order-inducing special API calls (e.g., Windows' **InterlockedExchange**), and various flavors of special atomic types. Many of these tools are tedious and/or difficult, and

their wide variety means that lock-free code ends up being written differently in different environments.

The last few years, however, have seen a major convergence across hardware and software vendors: The computing industry is coalescing around sequentially consistent ordered atomic variables as the default or only way to write lock-free code using the major languages and OS platforms. In a nutshell, *ordered atomic variables* are safe to read and write on multiple threads at the same time without doing any explicit locking because they provide two guarantees: their reads and writes are guaranteed to be executed in the order they appear in your program's source code; and each read or write is guaranteed to be atomic, all-or-nothing. They also have special operations such as **compareAndSet** that are guaranteed to be executed atomically. See [1] for further details about ordered atomic variables and how to use them correctly.

Ordered atomic variables are available in Java, C# and other .NET languages, and the forthcoming ISO C++ Standard, but under different names:

- Java provides ordered atomics under the **volatile** keyword (e.g., **volatile int**), and solidified this support in Java 5 (2004). Java additionally provides a few named types in **java.util.concurrent.atomic**, such as **AtomicLongArray**, that you can use for the same purpose.
- .NET mostly added them in Visual Studio 2005, also under the volatile keyword (e.g., **volatile int**). These are suitable for nearly all lock-free code uses, except for rare examples similar to Dekker's algorithm. .NET is fixing these remaining corner cases in Visual Studio 2010, which is in early beta as of this writing.
- ISO C++ added them to the C++0x draft Standard in 2007, under the templated name **atomic<T>** (e.g., **atomic**). They started to become available beginning in 2008 in the Boost project and other implementations. [2]. The ISO C++ atomics library also provides a C-compatible way to spell those types and their operations (e.g., **atomic_int**), and these appear to be likely to be adopted by ISO C in the future.

A Word About Optimization

We're going to look at how ordered atomics restrict the optimizations that compilers, CPUs, cache effects, and other parts of your execution environment might perform. So let's first briefly review some basic rules of optimization.

The most fundamental rule of optimization in pretty much any language is this: Optimizations that rearrange ('transform') your code's execution are always legal if they don't change the meaning of the program, so that the program can't tell the difference between executing the original code and the transformed code. In some languages, this is also known as the 'as if' rule -- which gets its name from the fact that the transformed code has the same observable effects 'as if' the original source code had been executed as written.

This rule cuts two ways: First, an optimization must never make it possible to get a result that wasn't possible before, or break any guarantees that the original code was allowed to rely on, including language semantics. If we produce an impossible result, after all, the program and the user certainly can tell the difference, and it's not just 'as if' we'd executed the original untransformed code.

Second, optimizations are permitted to reduce the set of possible executions. For example, an optimization might make some potential (but not guaranteed) interleavings never actually happen. This is okay, because the program couldn't rely on them happening anyway.

Ordered Atomics and Optimization

Using ordered atomic variables restricts the kinds of optimizations your compiler and processor and cache system can do. [3] There are two kinds of optimizations to consider:

- Optimizations on the ordered atomic reads and writes themselves.
- Optimizations on nearby ordinary reads and writes.

First, all of the ordered atomic reads and writes on a given thread must execute exactly in source code order, because that's one of the fundamental guarantees of ordered atomic variables. However, we can still perform some optimizations, in particular, optimizations that have the same effect as if this thread just always executed so quickly that another thread didn't happen to ever interleave at certain points.

For instance, consider this code, where a is an ordered atomic variable:

```
a = 1;          // A
a = 2;          // B
```

Is it legal for a compiler, processor, cache, or other part of the execution environment to transform the above code into the following, eliminating the redundant write in line A?

```
                      // A': OK: eliminate line A entirely
a = 2;          // B
```

The answer is, 'Yes.' This is legal because the program can't tell the difference; it's as if this thread always ran so fast that no other thread accessing a concurrently ever got to interleave between lines A and B to see the intermediate value. [4]

Similarly, if **a** is an ordered atomic variable and **local** is an unshared local variable, it is legal to transform

```
a = 1;          // C: write to a
local = a;            // D: read from a
```

to

```
a = 1;          // C: write to a
local = 1;            // D': OK, apply "constant propagation"
```

which eliminates the read from **a**. Even if another thread is concurrently trying to write to **a**, its as if this thread always ran so fast that the other thread never got to interleave between lines **C** and **D** to change the value before we can read our own back into **local**.

Second, nearby ordinary reads and writes can still be reordered around ordered atomic reads and writes, but subject to some restrictions. In particular, as described in [3], ordinary reads and writes can't move upward across (from after to before) an ordered atomic read, and can't move downward across (from before to after) an ordered atomic write. In brief, that could move them out of a critical section of code, and you can write programs that can tell the difference. For more details, see [3].

That's it for lock-free programming and ordered atomics. What about the other case that some "volatiles" address?

Case 2: Semantic-Free Variables for "Unusual" Memory Semantics

The second requirement is to deal with "unusual" memory that goes beyond a given language's memory model, where the compiler must assume that the variable can change value at any time and/or that reads and writes may have unknowable semantics and consequences. Classic examples include:

- Hardware registers, part 1: Asynchronous changes. For example, consider a memory location M on a custom board that is connected to an instrument that directly writes to **M**. Unlike ordinary memory that is only modified by the program itself, the value stored in **M** can change at any time even if no program thread writes to it; therefore, the compiler cannot make any assumptions that the value will be stable.
- Hardware registers, part 2: Semantics. For example, consider a memory location **M** on a custom board where writing to that location always automatically increments by one. Unlike an ordinary RAM memory location, the compiler can't even assume that performing a write to **M** and then immediately following it with a read from **M** will necessarily read the same value that was written.
- Memory having more than one address. If a given memory location is accessible using two different addresses **A1** and **A2**, a compiler or processor may not be able to know that writing to location **A1** can change the value at location **A2**. Any optimization that assumes a write to **A1** does not change the value of **A2** will break the program, and must be prevented.

Variables in such memory locations are *unoptimizable variables*, because the compiler can't safely make any assumptions about them at all. Put another way, the compiler needs to be told that such a variable doesn't participate in the normal type system, even if it appears to have a given type. For example, if memory location **M** or **A1**/**A2** in the aforementioned examples are typed as an "**int**" in the program, what does that really mean? It means at most that it has the size and layout of an **int**, but it cannot mean that it behaves like an **int** -- after all, **int**s don't autoincrement themselves when you write to them, or mysteriously change their values when you write to what looks like a different variable at some other address.

We need a way to turn off all optimizations on their reads and writes. ISO C and C++ have a portable, standard way to tell the compiler that this is such a special variable that it must not optimize: **volatile**.

Java and .NET have no comparable concept. Managed environments are supposed to know the full semantics of the program they execute, after all, so it's not surprising that they don't support memory with "unknowable" semantics. But both Java and .NET do provide escape hatches to leave the managed environment and invoke native code: Java provides the Java Native Interface (JNI) and .NET provides Platform Invoke (P/Invoke). The JNI specification [5] is silent about **volatile**, however, and doesn't mention either Java **volatile** or C/C++ **volatile** at all; similarly, the P/Invoke documentation doesn't mention interactions with .NET **volatile** or C/C++ **volatile**. So to correctly access an unoptimizable memory location in either Java or .NET, you must write C/C++ functions that use C/C++ volatile to do the needed work on behalf of their managed calling code, make sure they entirely encapsulate and hide the **volatile** memory (i.e., neither take nor return anything **volatile**), and call those functions through JNI and P/Invoke.

Unoptimizable Variables and (Not) Optimization

All reads and writes of unoptimizable variables on a given thread must execute exactly as written; no optimizations are allowed at all, because the compiler can't know the full semantics of the variable and when and how its value can change. This is a stronger statement than for ordered atomics, which only need to execute in source code order.

Consider again the two transformations we considered before, but this time replacing the ordered atomic variable a with the unoptimizable (C/C++ volatile) variable **v**:

```
v = 1;  // A
v = 2;  // B
```

Is it legal to transform this as follows to remove the apparently redundant write in line **A**?

```
                // A': invalid, cannot eliminate write
v = 2;  // B
```

The answer is no, because the compiler cannot possibly know that eliminating line **A**'s write to **v** won't change the meaning of the program. For example, **v** could be a location accessed by custom hardware that expects to see the value 1 before the value 2 and won't work correctly otherwise.

Similarly, if **v** is an unoptimizable variable and local is an unshared local variable, it is not legal to transform

```
v = 1;              // C: write to v
local = v;             // D: read from v
```

to

```
a = 1;               // C: write to v
```

```
local = 1;l ;  // D': invalid, cannot do
// "constant propagation"
```

to eliminate the read from **v**. For example, **v** could be a hardware address that automatically increments itself every time it's written, so that writing 1 will yield the value 2 on the next read.

Second, what about nearby ordinary reads and writes -- can those still be reordered around unoptimizable reads and writes? Today, there is no practical portable answer because C/C++ compiler implementations vary widely and aren't likely to converge anytime soon. For example, one interpretation of the C++ Standard holds that ordinary reads can move freely in either direction across a C/C++ **volatile** read or write, but that an ordinary write cannot move at all across a C/C++ **volatile** read or write -- which would make C/C++ **volatile** both less restrictive and more restrictive, respectively, than an ordered atomic. Some compiler vendors support that interpretation; others don't optimize across **volatile** reads or writes at all; and still others have their own preferred semantics.

Summary

To safely write lock-free code that communicates between threads without using locks, prefer to use ordered atomic variables: Java/.NET **volatile**, C++**0x atomic<T>**, and C-compatible **atomic_T**.

To safely communicate with special hardware or other memory that has unusual semantics, use unoptimizable variables: ISO C/C++ volatile. Remember that reads and writes of these variables are not necessarily atomic, however.

Finally, to express a variable that both has unusual semantics and has any or all of the atomicity and/or ordering guarantees needed for lock-free coding, only the ISO C++0x draft Standard provides a direct way to spell it: **volatile atomic<T>**.

Notes

[1] H. Sutter. "Writing Lock-Free Code: A Corrected Queue" (*DDJ*, October 2008). Available online at www.ddj.com/hpc-high-performance-computing/210604448.

[2] See www.boost.org.

[3] H. Sutter. "Apply Critical Sections Consistently" (*DDJ*, November 2007). Available online at www.ddj.com/hpc-high-performance-computing/202401098.

[4] A common objection at this point is: "In the original code, it was possible for another thread to see the intermediate value, but that is impossible in the transformed code. Isn't that a change in observable behavior? " The answer is, "No, " because the program was never guaranteed to ever actually interleave just in time to see that value; it was already a legal outcome for this thread to just always run so fast that the interleaving happened to never manifest. Again, what this optimization does is reduce the set of possible executions, which is always legal.

[5] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. (Prentice Hall, 1999). Available online at java.sun.com/docs/books/jni/.

Dr.Dobb's
go parallel
Translating Multicore Power into Application Performance

Design for Manycore Systems

Why worry about "manycore" today?

By Herb Sutter Dr.Dobb's Journal
August 11, 2009
URL : http://drdobbs.com/go-parallel/article/219200099

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.

Dual- and quad-core computers are obviously here to stay for mainstream desktops and notebooks. But do we really need to think about "many-core" systems if we're building a typical mainstream application right now? I find that, to many developers, "many-core" systems still feel fairly remote, and not an immediate issue to think about as they're working on their current product.

This column is about why it's time right now for most of us to think about systems with lots of cores. In short: Software is the (only) gating factor; as that gate falls, hardware parallelism is coming more and sooner than many people yet believe.

Recap: What "Everybody Knows"

Figure 1 is the canonical "free lunch is over" slide showing major mainstream microprocessor trends over the past 40 years. These numbers come from Intel's product line, but every CPU vendor from servers (e.g., Sparc) to mobile devices (e.g., ARM) shows similar curves, just shifted slightly left or right. The key point is that Moore's Law is still generously delivering transistors at the rate of twice as many per inch or per dollar every couple of years. Of course, any exponential growth curve must end, and so eventually will Moore's Law, but it seems to have yet another decade or so of life left.

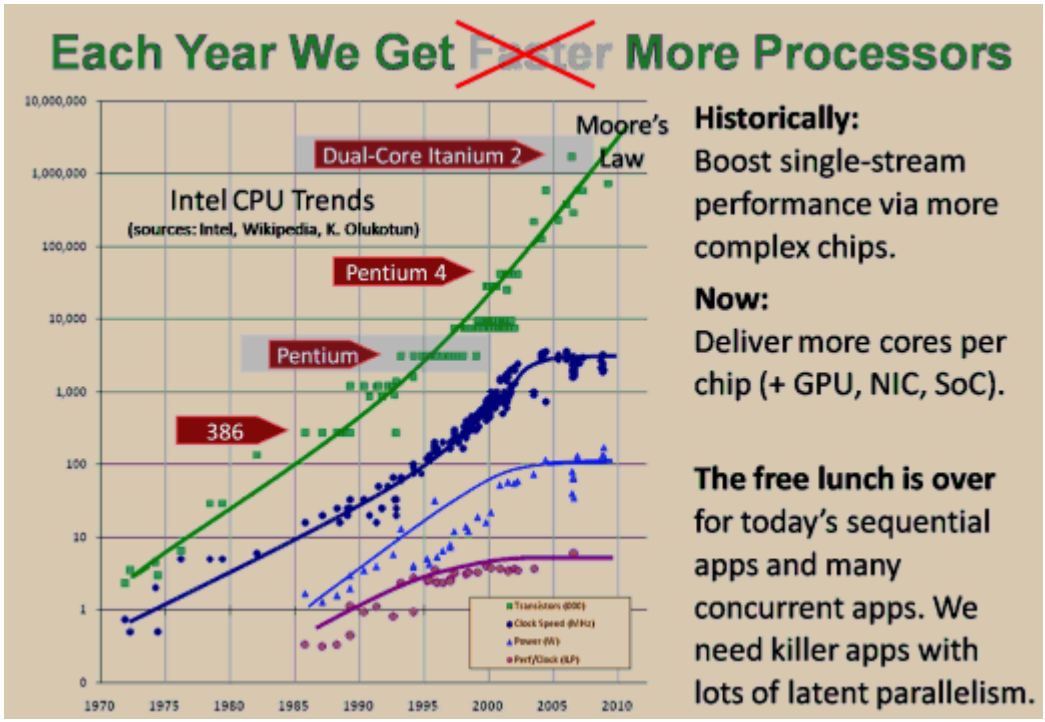

Figure 1: Canonical "free lunch is over" slide. Note Pentium vs. dual-core Itanium transistor counts.

Mainstream microprocessor designers used to be able to use their growing transistor budgets to make single-threaded code faster by making the chips more complex, such as by adding out-of-order ("OoO") execution, pipelining, branch prediction, speculation, and other techniques. Unfortunately, those techniques have now been largely mined out. But CPU designers are still reaping Moore's harvest of transistors by the boatload, at least for now. What to do with all those transistors? The main answer is to deliver more cores rather than more complex cores. Additionally, some of the extra transistor real estate can also be soaked up by bringing GPUs, networking, and/or other functionality on-chip as well, up to putting an entire "system on a chip" (aka "SoC") like the Sun UltraSPARC T2.

How Much, How Soon?

How quickly can we expect more parallelism in our chips? The nave answer would be: Twice as many cores every couple of years, just continuing on with Moore's Law. That's the baseline projection approximated in Figure 2, assuming that some of the extra transistors aren't also used for other things.

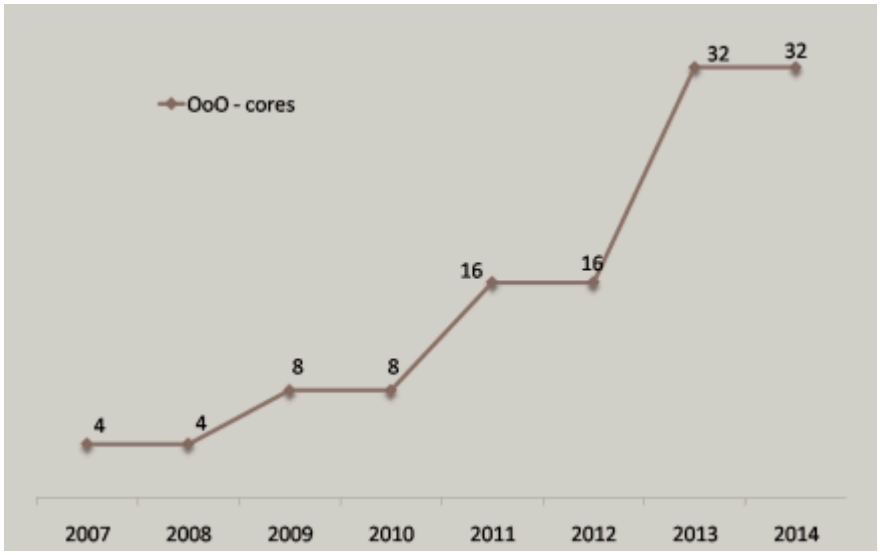

Figure 2: Simple extrapolation of "more of the same big cores" (not counting some transistors being used for other things like on-chip GPUs, or returning to smaller cores).

However, the naive answer misses several essential ingredients. To illustrate, notice one interesting fact hidden inside Figure 1. Consider the two highlighted chips and their respective transistor counts in million transistors (Mt):

- 4.5Mt: 1997 "Tillamook" Pentium P55C. This isn't the original Pentium, it's a later and pretty attractive little chip that has some nice MMX instructions for multimedia processing. Imagine running this 1997 part at today's clock speeds.
- 1,700Mt: 2006 "Montecito" Itanium 2. This chip handily jumped past the billion-transistor mark to deliver two Itanium cores on the same die. [1]

So what's the interesting fact? (Hint: 1,700 ÷ 4.5 = ???.)

In 2006, instead of shipping a dual-core Itanium part, with exactly the same transistor budget Intel could have shipped a chip that contained 100 decent Pentium-class cores with enough space left over for 16 MB of Level 3 cache. True, it's more than a matter of just etching the logic of 100 cores on one die; the chip would need other engineering work, such as in improving the memory interconnect to make the whole chip a suitably balanced part. But we can view those as being relatively 'just details' because they don't require engineering breakthroughs.

Repeat: Intel could have shipped a 100-core desktop chip with ample cache -- in 2006. So why didn't they? (Or AMD? Or Sun? Or anyone else in the mainstream market?) The short answer is the counter-question: Who would buy it? The world's popular mainstream client applications are largely single-threaded or nonscalably multithreaded, which means that existing applications create a double disincentive:

- They couldn't take advantage the extra cores, because they don't contain enough inherent parallelism to scale well.
- They wouldn't run as fast on a smaller and simpler core, compared to a bigger core that contains extra complexity to run single-threaded code faster.

Astute readers might have noticed that when I said, "why didn't Intel or Sun," I left myself open to contradiction, because Sun (in particular) did do something like that already, and Intel is doing it now. Let's find out what, and why.

Hiding Latency: Complex Cores vs. Hardware Threads

One of the major reasons today's modern CPU cores are so big and complex, to make single-threaded applications run faster, is that the complexity is used to hide the latency of accessing glacially slow RAM -- the "memory wall."

In general, how do you hide latency? Briefly, by adding concurrency: Pipelining, out-of-order execution, and most of the other tricks used inside complex CPUs inject various forms of concurrency within the chip itself, and that lets the CPU keep the pipeline to memory full and well-utilized and hide much of the latency of waiting for RAM. (That's a very brief summary. For more, see my machine architecture talk, available on Google video. [2])

So every chip needs to have a certain amount of concurrency available inside it to hide the memory wall. In 2006, the memory wall was higher than in 1997; so naturally, 2006 cores of any variety needed to contain more total concurrency than in 1997, in whatever form, just to avoid spending most of their time waiting for memory. If we just brought the 1997 core as-is into the 2006 world, running at 2006 clock speeds, we would find that it would spend most of its time doing something fairly unmotivating: just idling, waiting for memory.

But that doesn't mean a simpler 1997-style core can't make sense today. You just have to provide enough internal hardware concurrency to hide the memory wall. The squeezing-the-toothpaste-tube metaphor applies directly: When you squeeze to make one end smaller, some other part of the tube has to get bigger. If we take away some of a modern core's concurrency-providing complexity, such as removing out-of-order execution or some or all pipeline stages, we need to provide the missing concurrency in some other way.

But how? A popular answer is: Through hardware threads. (Don't stop reading if you've been burned by hardware threads in the past. See the sidebar "Hardware Threads Are Important, But Only For Simpler Cores.")

Hardware Threads Are Important, But Only For Simpler Cores

Hardware threads have acquired a tarnished reputation. Historically, for example, Pentium hyperthreading has been a mixed blessing in practice; it made some applications run something like 20% faster by hiding some remaining memory latency not already covered in other ways, but made other applications actually run slower because of increased cache contention and other effects. (For one example, see [3].)

But that's only because hardware threads are for hiding latency, and so they're not nearly as useful on our familiar big, complex cores that already contain lots of other latency-hiding concurrency. If you've had mixed or negative results with hardware threads, you were probably just using them on complex chips where they don't matter as much.

Don't let that turn you off the idea of hardware threading. Although hardware threads are a mixed bag on complex cores where there isn't much remaining memory latency left to hide, they are absolutely essential on simpler cores that aren't hiding nearly enough memory latency in other ways, such as simpler in-order CPUs like Niagara and Larrabee. Modern GPUs take the extreme end of this design range, making each core very simple (typically not even a general-purpose core) and relying on lots of hardware threads to keep the core doing useful work even in the face of memory latency.

—HS

Toward Simpler, Threaded Cores

What are hardware threads all about? Here's the idea: Each core still has just one basic processing unit (arithmetic unit, floating-point unit, etc.) but can keep multiple threads of execution "hot" and ready to switch to quickly as others stall waiting for memory. The switching cost is just a few cycles; it's nothing remotely similar to the cost of an operating system-level context switch. For example, a core with four hardware threads can run the first thread until it encounters a memory operation that forces it to wait, and then keep doing useful work by immediately switching to the second thread and executing that until it also has to wait, and then switching to the third until it also waits, and then the fourth until it also waits -- and by then hopefully the first or second is ready to run again and the core can stay busy. For more details, see [4].

The next question is, How many hardware threads should there be per core? The answer is: As many as you need to hide the latency no longer hidden by other means. In practice, popular answers are four and eight hardware threads per core. For example, Sun's Niagara 1 and Niagara 2 processors are based on simpler cores, and provide four and eight hardware threads per core, respectively. The UltraSPARC T2 boasts 8 cores of 8 threads each, or 64 hardware threads, as well as other functions including networking and I/O that make it a "system on a chip." [5] Intel's new line of Larrabee chips is expected to range from 8 to 80 (eighty) x86-compatible cores, each with four or more hardware threads, for a total of 32 to 320 or more hardware threads per CPU chip. [6] [7]

Figure 3 shows a simplified view of possible CPU directions. The large cores are big, modern, complex cores with gobs of out-of-order execution, branch prediction, and so on.

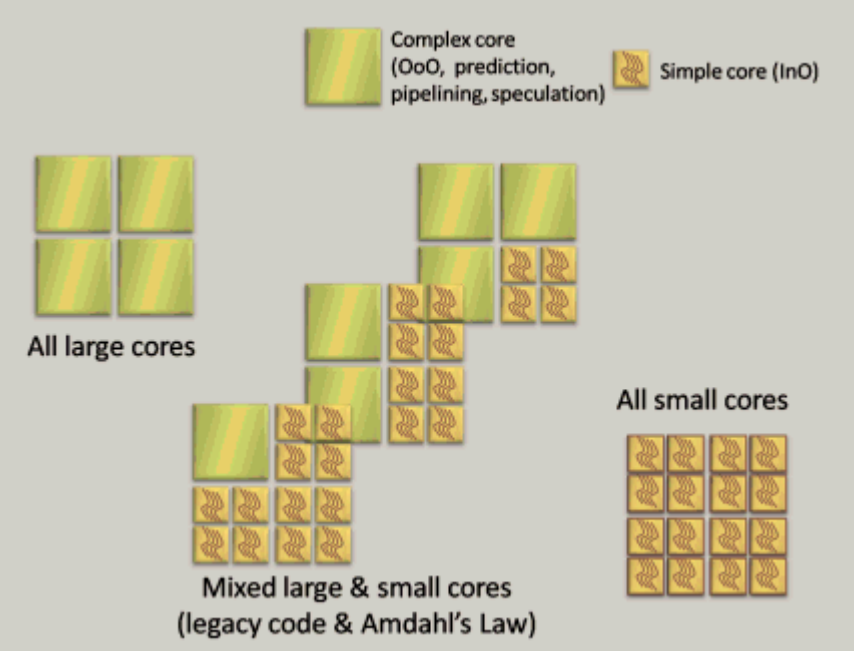

Figure 3: A few possible future directions.

The left side of Figure 3 shows one possible future: We could just use Moore's transistor generosity to ship more of the same -- complex modern cores as we're used to in the mainstream today. Following that route gives us the projection we already saw in Figure 2.

But that's only one possible future, because there's more to the story. The right side of Figure 3 illustrates how chip vendors could swing the pendulum partway back and make moderately simpler chips, along the lines that Sun's Niagara and Intel's Larrabee processors are doing.

In this simple example for illustrative purposes only, the smaller cores are simpler cores that consume just one-quarter the number of transistors, so that four times as many can fit in the same area. However, they're simpler because they're missing some of the machinery used to hide memory latency; to make up the deficit, the small cores also have to provide four hardware threads per core. If CPU vendors were to switch to this model, for example, we would see a one-time jump of 16 times the hardware concurrency -- four times the number of cores, and at the same time four times as many hardware threads per core -- on top of the Moore's Law-based growth in Figure 2.

What makes smaller cores so appealing? In short, it turns out you can design a small-core device such that:

- 4x cores = 4x FP performance: Each small, simple core can perform just as many floating-point operations per second as a big, complex core. After all, we're not changing the core execution logic (ALU, FPU, etc.); we're only changing the supporting machinery around it that hides the memory latency, to replace OoO and predictors and pipelines with some hardware threading.
- Less total power: Each small, simple core occupies one-quarter of the transistors, but uses less than one-quarter the total power.

Who wouldn't want a CPU that has four times the total floating-point processing throughput and consumes less total power? If that's possible, why not just ship it tomorrow?

You might already have noticed the fly in the ointment. The key question is: Where does the CPU get the work to assign to those multiple hardware threads? The answer is, from the same place it gets the work for multiple cores: From you. Your application has to provide the software threads or other parallel work to run on those hardware threads. If it doesn't, then the core will be idle most of the time. So this plan only works if the software is scalably parallel.

Imagine for a moment that we live in a different world, one that contains several major scalably parallel "killer" applications -- applications that a lot of mainstream consumers want to use and that run better on highly parallel hardware. If we have such scalable parallel software, then the right-hand side of Figure 3 is incredibly attractive and a boon for everyone, including for end users who get much more processing clout as well as a smaller electricity bill.

In the medium term, it's quite possible that the future will hold something in be-tween, as shown in the middle of Figure 3: heterogeneous chips that contain both large and small cores. Even these will only be viable if there are scalable parallel ap-plications, but they offer a nice migration path from today's applications. The larger cores can run today's applications at full speed, with ongoing incremental improve-ments to sequential performance, while the smaller cores can run tomorrow's applica-tions with a reenabled "free lunch" of exponential improvements to CPU-bound per-formance (until the program becomes bound by some other factor, such as memory or network I/O). The larger cores can also be useful for faster execution of any unavoidably sequential parts of new parallel applications. [8]

How Much Scalability Does Your Application Need?

So how much parallel scalability should you aim to support in the application you're working on today, assuming that it's compute-bound already or you can add killer features that are compute-bound and also amenable to parallel execution? The answer is that you want to match your application's scalability to the amount of hardware parallelism in the target hardware that will be available during your application's expected production or shelf lifetime. As shown in Figure 4, that equates to the number of hardware threads you expect to have on your end users' machines.

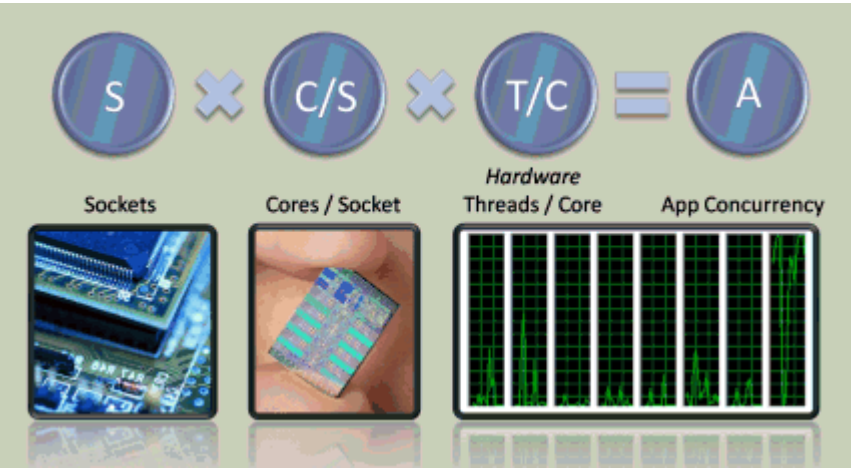

Figure 4: How much concurrency does your program need in order to exploit given hardware?

Let's say that YourCurrentApplication 1.0 will ship next year (mid-2010), and you expect that it'll be another 18 months until you ship the 2.0 release (early 2012) and probably another 18 months after that before most users will have upgraded (mid-2013). Then you'd be interested in judging what will be the likely mainstream hardware target up to mid-2013.

If we stick with "just more of the same" as in Figure 2's extrapolation, we'd expect aggressive early hardware adopters to be running 16-core machines (possibly double that if they're aggressive enough to run dual-CPU workstations with two sockets), and we'd likely expect most general mainstream users to have 4-, 8- or maybe a smattering of 16-core machines (accounting for the time for new chips to be adopted in the marketplace).

But what if the gating factor, parallel-ready software, goes away? Then CPU vendors would be free to take advantage of options like the one-time 16-fold hardware parallelism jump illustrated in Figure 3, and we get an envelope like that shown in Figure 5.

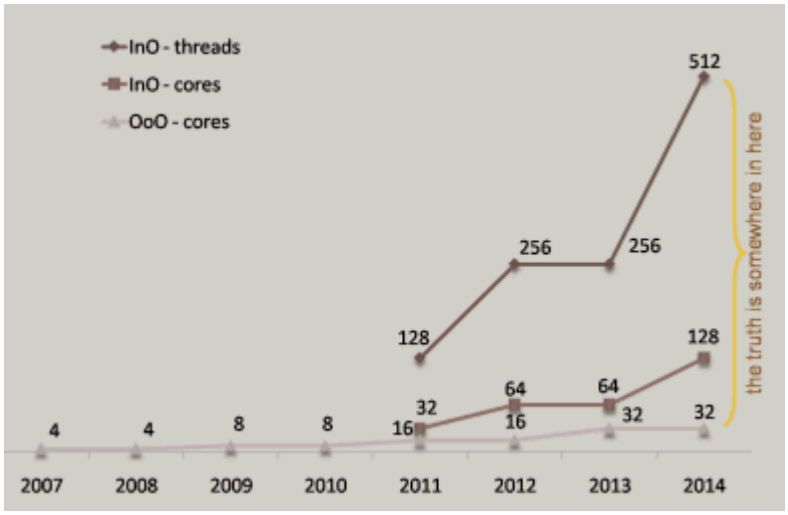

Figure 5: Extrapolation of "more of the same big cores" and "possible one-time switch to 4x smaller cores plus 4x threads per core" (not counting some transistors being used for other things like on-chip GPUs).

Now, what amount of parallelism should the application you're working on now have, if it ships next year and will be in the market for three years? And what does that answer imply for the scalability design and testing you need to be doing now, and the hardware you want to be using at least part of the time in your testing lab? (We can't buy a machine with 32-core mainstream chip yet, but we can simulate one pretty well by buying a machine with four eight-core chips, or eight quad-core chips It's no coincidence that in recent articles I've often shown performance data on a 24-core machine, which happens to be a four-socket box with six cores per socket.)

Note that I'm not predicting that we'll see 256-way hardware parallelism on a typical new Dell desktop in 2012. We're close enough to 2011 and 2012 that if chip vendors aren't already planning such a jump to simpler, hardware-threaded cores, it's not going to happen. They typically need three years or so of lead time to see, or at least anticipate, the availability of parallel software that will use the chips, so that they can design and build and ship them in their normal development cycle.

I don't believe either the bottom line or the top line is the exact truth, but as long as sufficient parallel-capable software comes along, the truth will probably be somewhere in between, especially if we have processors that offer a mix of large- and small-core chips, or that use some chip real estate to bring GPUs or other devices on-die. That's more hardware parallelism, and sooner, than most mainstream developers I've encountered expect.

Interestingly, though, we already noted two current examples: Sun's Niagara, and Intel's Larrabee, already provide double-digit parallelism in mainstream hardware via smaller cores with four or eight hardware threads each. "Manycore" chips, or perhaps more correctly "manythread" chips, are just waiting to enter the mainstream. Intel could have built a nice 100-core part in 2006. The gating factor is the software that can exploit the hardware parallelism; that is, the gating factor is you and me.

Summary

The pendulum has swung toward complex cores nearly far as it's practical to go. There's a lot of performance and power incentive to ship simpler cores. But the gating factor is software that can use them effectively; specifically, the availability of scalable parallel mainstream killer applications. The only thing I can foresee that could prevent the widespread adoption of manycore mainstream systems in the next decade would be a complete failure to find and build some key parallel killer apps, ones that large numbers of people want and that work better with lots of cores. Given our collective inventiveness, coupled with the parallel libraries and tooling now becoming available, I think such a complete failure is very unlikely.

As soon as mainstream parallel applications become available, we will see hardware parallelism both more and sooner than most people expect. Fasten your seat belts, and remember Figure 5.

References

[1] Montecito press release (Intel, July 2006) www.intel.com/pressroom/archive/releases/20060718comp.htm.

[2] H. Sutter. "Machine Architecture: Things Your Programming Language Never Told You" (Talk at NWCPP, September 2007). http://video.google.com/videoplay?docid=-4714369049736584770

[3] "Improving Performance by Disabling Hyperthreading"(Novell Cool Solutions feature, October 2004). www.novell.com/coolsolutions/feature/637.html

[4] J. Stokes. "Introduction to Multithreading, Superthreading and Hyperthreading" (Ars Technica, October 2002). http://arstechnica.com/old/content/2002/10/hyperthreading.ars

[5] UltraSPARC T2 Processor(Sun). www.sun.com/processors/UltraSPARC-T2/datasheet.pdf

[6] L. Seiler et al. "Larrabee: A Many-Core x86 Architecture for Visual Computing" (ACM Transactions on Graphics (27,3), Proceedings of ACM SIGGRAPH 2008, August 2008). http://download.intel.com/technology/architecture-silicon/Siggraph_Larrabee_paper.pdf

[7] M. Abrash. "A First Look at the Larrabee New Instructions" (Dr. Dobb's, April 2009). www.ddj.com/hpc-high-performance-computing/216402188

[8] H. Sutter. "Break Amdahl's Law!" (Dr. Dobb's Journal, February 2008). www.ddj.com/cpp/205900309.