<http://www.drdobbs.com/parallel/lock-free-queues/208801974>

**Lock-Free Queues**

By Petru Marginean, July 01, 2008

[**3 Comments**](http://www.drdobbs.com/parallel/lock-free-queues/208801974#disqus_thread)

**Source Code Accompanies This Article. Download It Now.**

* [lockfree.txt](http://www.drdobbs.com/parallel/lock-free-queues/parallel/sourcecode/lock-free-queues/30000034)
* [lockfreequeue.zip](http://www.drdobbs.com/parallel/lock-free-queues/parallel/sourcecode/lock-free-queues/30000035)

**One thread can write and another read—at the same time!**

*Petru is a Vice President for Morgan Stanley, where he works as a C++ senior programmer in investment banking. He can be contacted at* [*petru.marginean@gmail.com*](mailto:petru.marginean@gmail.com)*.*

*This article as written assumes a sequentially consistent model. In particular, the code relies on specific order of instructions in both Consumer and Producer methods. However, without inserting proper memory barrier instructions, these instructions can be reordered with unpredictable results (see, for example, the classic* [*Double-Checked Locking*](http://www.ddj.com/184405726) *problem).*

*Another issue is using the standard* ***std::list<T>****. While the article mentions that it is the developer responsibility to check that the reading/writing* ***std::list<T>::iterator*** *is atomic, this turns out to be too restrictive. While gcc/MSVC++2003 has 4-byte iterators, the MSVC++2005 has 8-byte iterators in Release Mode and 12-byte iterators in the Debug Mode.*

*The solution to prevent this is to use memory barriers/volatile variables. The downloadable code featured at the top of this article has fixed that issue.*

*Many thanks to Herb Sutter who signaled the issue and helped me fix the code. --P.M.*

Queues can be useful in a variety of systems involving data-stream processing. Typically, you have a data source producing data—requests coming to a web server, market feeds, or digital telephony packets—at a variable pace, and you need to process the data as fast as possible so there are no losses. To do this, you can push data into a queue using one thread and process it using a different thread—a good utilization of resources on multicore processors. One thread inserts data into the queue, and the other reads/deletes elements from the queue. Your main requirement is that a high-rate data burst does not last longer than the system's ability to accumulate data while the consumer thread handles it. The queue you use has to be threadsafe to prevent race conditions when inserting/removing data from multiple threads. For obvious reasons, it is necessary that the queue mutual exclusion mechanism add as little overhead as possible.

In this article, I present a lock-free queue (the source code for the *lockfreequeue* class is available online; see [www.ddj.com/code/](http://www.ddj.com/code/)) in which one thread can write to the queue and another read from it—at the same time without any locking.

To do this, the code implements these requirements:

* There is a single writer (*Producer*) and single reader (*Consumer*). When you have multiple producers and consumers, you can still use this queue with some external locking. You cannot have multiple producers writing at the same time (or multiple consumers consuming the data simultaneously), but you can have one producer and one consumer (2x threads) accessing the queue at the same time (**Responsibility: developer**).
* When inserting/erasing to/from an *std::list<T>*, the iterators for the existing elements must remain valid (**Responsibility: library implementor**).
* Only one thread modifies the queue; the producer thread both adds/erases elements in the queue (**Responsibility: library implementor**).
* Beside the underlying *std::list<T>* used as the container, the lock-free queue class also holds two iterators pointing to the not-yet-consumed range of elements; each is modified by one thread and read by the other (**Responsibility: library implementor**).
* Reading/writing *list<T>::iterator* is atomic on the machine upon which you run the application. If they are not on your implementation of STL, you should check whether the raw pointer's operations are atomic. You could easily replace the iterators to be mentioned shortly with raw pointers in the code (**Responsibility: machine**).

Because I use Standard C++, the code is portable under the aforementioned "machine" assumption:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | template <typename T>  struct LockFreeQueue  {    LockFreeQueue();    void Produce(const T& t);    bool Consume(T& t);  private:    typedef std::list<T> TList;    TList list;    typename TList::iterator iHead, iTail;  }; |

Considering how simple this code is, you might wonder how can it be threadsafe. The magic is due to design, not implementation. Take a look at the implementation of the *Produce()* and *Consume()* methods. The *Produce()* method looks like this:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void Produce(const T& t)  {    list.push\_back(t);    iTail = list.end();    list.erase(list.begin(), iHead);  } |

To understand how this works, mentally separate the data from *LockFreeQueue<T>* into two groups:

* The list and the *iTail* iterator, modified by the *Produce()* method (*Producer* thread).
* The *iHead* iterator, modified by the *Consume()* method (*Consumer* thread).

*Produce()* is the only method that changes the list (adding new elements and erasing the consumed elements), and it is essential that only one thread ever calls *Produce()*—it's the *Producer* thread! The iterator (*iTail*) (only manipulated by the *Producer* thread) changes it only after a new element is added to the list. This way, when the *Consumer* thread is reading the *iTail* element, the new added element is ready to be used. The *Consume()* method tries to read all the elements between *iHead* and *iTail* (excluding both ends).

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | bool Consume(T& t)  {    typename TList::iterator iNext = iHead;    ++iNext;    if (iNext != iTail)    {      iHead = iNext;      t = \*iHead;      return true;    }    return false;  } |

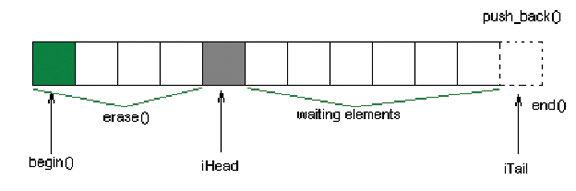
This method reads the elements, but doesn't remove them from the list. Nor does it access the list directly, but through the iterators. They are guaranteed to be valid after *std::list<T>* is modified, so no matter what the *Producer* thread does to the list, you are safe to use them.

The *std::list<T>* maintains an element (pointed to by *iHead*) that is considered already read. For this algorithm to work even when the queue was just created, I add an empty *T()* element in the constructor of the *LockFreeQueue<T>* (see Figure 1):

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | LockFreeQueue()  {    list.push\_back(T());    iHead = list.begin();    iTail = list.end();  } |

[Click image to view at full size]



**Figure 1: Adding an empty T() element in the constructor of the LockFreeQueue<T>.**

*Consume()* may fail to read an element (and return false). Unlike traditional lock-based queues, this queue works fast when the queue is not empty, but needs an external locking or polling method to wait for data. Sometimes you want to wait if there is no element available in the queue, and avoid returning false. A naive approach to waiting is:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | T Consume()  {    T tmp;    while (!Consume(tmp))      ;    return tmp;  } |

This *Consume()* method will likely heat up one of your CPUs red-hot to 100-percent use if there are no elements in the queue. Nevertheless, this should have good performance when the queue is not empty. However, if you think of it, a queue that's almost never empty is a sign of systemic trouble: It means the consumer is unable to keep pace with the producer, and sooner or later, the system is doomed to die of memory exhaustion. Call this approach *NAIVE\_POLLING*.

A friendlier *Consume()* function does some pooling and calls some sort of *sleep()* or *yield()* function available on your system:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | T Consume(int wait\_time = 1/\*milliseconds\*/)  {    T tmp;    while (!Consume(tmp))    {      Sleep(wait\_time/\*milliseconds\*/);    }    return tmp;  } |

The *DoSleep()* can be implemented using *nanosleep()* (POSIX) or *Sleep()* (Windows), or even better, using *boost::thread::sleep(),* which abstracts away system-dependent nomenclature. Call this approach *SLEEP*. Instead of simple polling, you can use more advanced techniques to signal the *Consumer* thread that a new element is available. I illustrate this in Listing One using a *boost::condition* variable.

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | #include <boost/thread.hpp>  #include <boost/thread/condition.hpp>  #include <boost/thread/xtime.hpp>    template <typename T>  struct WaitFreeQueue  {      void Produce(const T& t)      {          queue.Produce(t);          cond.notify\_one();      }      bool Consume(T& t)      {          return queue.Consume(t);      }      T Consume(int wait\_time = 1/\*milliseconds\*/)      {          T tmp;          if (Consume(tmp))              return tmp;          // the queue is empty, try again (possible waiting...)          boost::mutex::scoped\_lock lock(mtx);          while (!Consume(tmp)) // line A          {              boost::xtime t;              boost::xtime\_get(&t, boost::TIME\_UTC);              AddMilliseconds(t, wait\_time);              cond.timed\_wait(lock, t); // line B          }          return tmp;      }  private:      LockFreeQueue<T> queue;      boost::condition cond;      boost::mutex mtx;  }; |

**Listing One**

I used the *timed\_wait()* instead of the simpler *wait()* to solve a possible deadlock when *Produce()* is called between line *A* and line *B* in Listing One. Then *wait()* will miss the *notify\_one()* call and have to wait for the next produced element to wake up. If this element never comes (no more produced elements or if the *Produce()* call actually waits for *Consume()* to return), there's a deadlock. Call this approach *TIME\_WAIT*.

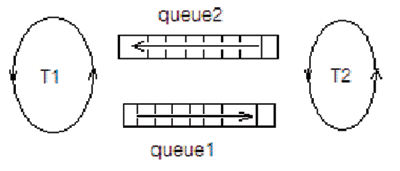
The lock is still wait-free as long as there are elements in the queue. In this case, the *Consumer()* thread does no waiting and reads data as fast as possible (even with the *Producer()* that is inserting new elements). Only when the queue is exhausted does locking occur.

**One thread can write and another read—at the same time!**

**The Ping-Pong Test**

To compare the three approaches (*NAIVE\_POLLING, SLEEP,* and *TIME\_WAIT*), I implemented a test called "Ping-Pong" that is similar to the game of table tennis (the source code is available online). In Figure 2, there are two identical queues between the threads *T1* and *T2*. You first load one of the queues with a number of "balls," then ask each thread to read from one queue and write to the other. The result is a controlled infinite loop. By limiting the game to a fixed number of reads/writes ("shots"), you get an understanding of how the queue behaves when varying the waiting/sleep time and strategy and the number of "balls." The faster the game, the better the performance. You should also check CPU usage to see how much of it is used for real work.

* "No ball" means "do nothing" (like two players waiting for the other to start). This gives you an idea of how good the queues are when there is no data—how nervous the players are. Ideally, CPU usage should be zero.
* "One ball" is like the real ping-pong game: Each player shoots and waits for the other to shoot.
* "Two (or more) balls" means both players could shoot at the same time, modulo collision and waiting issues.



**Figure 2: The Ping-Pong test.**

In a wait-free system, the more balls in the game, the better the performance gain compared to the classic locking strategy. This is because wait-free is an optimistic concurrency control method (works best when there is no contention), while classical lock-based concurrency control is pessimistic (assumes contention happens and preemptively inserts locking).

Ready to play? Here is the Ping-Pong test command line:

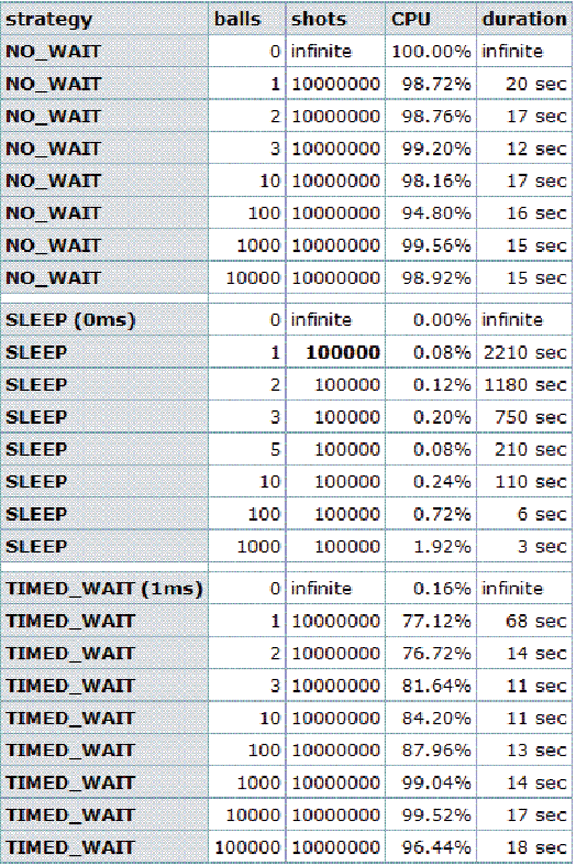
[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974?pgno=2)

|  |  |
| --- | --- |
| 1  2 | $> ./pingpong [strategy] [timeout] [balls] [shots] |

When you run the program, the tests show the results in the table shown in Figure 3:

* The best combination is the *timed\_wait()* with a small wait time (1ms in the test for *TIMED\_WAIT*). It has a very fast response time and almost 0 percent CPU usage when the queue is empty.
* Even when the sleep time is 0 (*usleep(0)*), the worst seems to be the *sleep()* method, especially when the queue is likely to be empty. (The number of shots in this case is 100-times smaller than the other cases because of the long duration of the game.)
* The *NO\_WAIT* strategy is fast but behaves worst when there are no balls (100-percent CPU usage to do nothing). It has the best performance when there is a single ball.

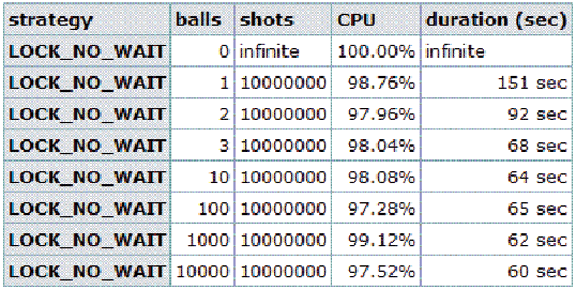
[Click image to view at full size]



**Figure 3: Ping-Pong test results.**

Figure 4 presents a table with the results for a classic approach (see *SafeQueue*). These results show that this queue is, on average, more than four-times slower than the *LockFreeQueue*. The slowdown comes from the synchronization between threads. Both *Produce()* and *Consume()* have to wait for each other to finish. CPU usage is almost 100 percent for this test (similar to the *NO\_WAIT* strategy, but not even close to its performance).

[Click image to view at full size]



**Figure 4: Classic approach results.**

**One thread can write and another read—at the same time!**

**Final Considerations**

The single-threaded code below shows the value of the *list.size()* when Producing/ Consuming elements:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | LockFreeQueue<int> q;   // list.size() == 1  q.Produce(1);    // list.size() == 2  int i;  q.Consume(i);  // list.size() == still 2!;                 // Consume() doesn't modify the list  q.Produce(i);    // list.size() == 2 again; |

The size of the queue is 1 if *Produce()* was never called and greater than 1 if any element was produced.

No matter how many times *Consume()* is called, the list's size will stay constant. It is *Produce()* that is increasing the size (by 1); and if there were consumed elements, it will also delete them from the queue. In a way, *Produce()* acts as a simple garbage collector. The whole thread safety comes from the fact that specific data is modified from single threads only. The synchronization between threads is done using iterators (or pointers, whichever has atomic *read/write* operation on your machine). Also consider this code:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974?pgno=3)

|  |  |
| --- | --- |
| 1  2 | usleep(1000);    // sleep 1 microsecond |

On the face of it, this line of code makes a thread sleep for 1 microsecond, and then continue. In reality, 1 microsecond is just a lower bound to the duration of the call.

The man page for *usleep()* says, "The *usleep()* function suspends execution of the calling process for (at least) usec microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers," or if you use the *nanosleep()* function. "Therefore, *nanosleep()* always pauses for at least the specified time; however, it can take up to 10 ms longer than specified until the process becomes runnable again."

So if the process is not scheduled under a real-time policy, there's no guarantee when your thread will be running again. I've done some tests and (to my surprise) there are situations when code such as:

[?](http://www.drdobbs.com/parallel/lock-free-queues/208801974?pgno=3)

|  |  |
| --- | --- |
| 1  2 | cond.timed\_wait(lock, x);    // x = e.g. 1 millisecond |

will actually wait for more than 1 second.

**Acknowledgments**

Many thanks to Andrei Alexandrescu who took the time to review this article. Also, thanks to Radu Duta for making useful corrections.

위 글은 Single Producer Single Consumer인 경우에 해당하는 글이네. 단지 그림 3이 잘 이해가 안 된다. 그림 1에 소개된 기법, 트릭이 흥미롭다.

유명하신 Herb Sutter가 Dr.Dobb’s journal에 기고한 글.

**Lock-Free Code: A False Sense of Security**

By Herb Sutter, September 08, 2008

[**Post a Comment**](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279#disqus_thread)

**Writing lock-free code can confound anyone—even expert programmers, as Herb shows this month.**

*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*](http://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.

**Writing lock-free code can confound anyone—even expert programmers, as Herb shows this month.**

**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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // 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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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.

**Writing lock-free code can confound anyone—even expert programmers, as Herb shows this month.**

**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*:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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.)

**Writing lock-free code can confound anyone—even expert programmers, as Herb shows this month.**

**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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=4)

|  |  |
| --- | --- |
| 1  2  3 | 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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=4)

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Is this change enough?  list.push\_back(t);  // A: add the new item  <font color="#FF0000">mb();       // full fence</font>  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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=4)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Better, but is it enough?  list.push\_back(t);  <font color="#FF0000">tmp</font> = list.end();  <font color="#FF0000">mb();       // full fence</font>  iTail <font color="#FF0000">= tmp;</font> |

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*.

**Writing lock-free code can confound anyone—even expert programmers, as Herb shows this month.**

**Ordering Problems in Consume**

Here's another reordering problem, this time from *Consume*:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=5)

|  |  |
| --- | --- |
| 1  2  3  4 | 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:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=5)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Problematic compiler/processor transformation  if (iNext != iTail) {    <font color="#FF0000">iHead = 0xDEADBEEF;</font>    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*:

[?](http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279?pgno=5)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Another problematic transformation  <font color="#FF0000">\_\_temp = iHead;  iHead = iNext;  // speculatively set to iNext</font>  if (iNext == iTail) {   // note: inverted test!    <font color="#FF0000">iHead = \_\_temp;   // undo if we guessed wrong  } else {</font>    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](http://www.ddj.com/cpp/184401930)).

[2] P. Marginean, "Lock-Free Queues," *Dr. Dobb's Journal*, July 2008. ([www.ddj.com/208801974](http://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](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2669.htm)).

**Herb continues his exploration of lock-free code--this time focusing on creating a lock-free queue.**

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>.

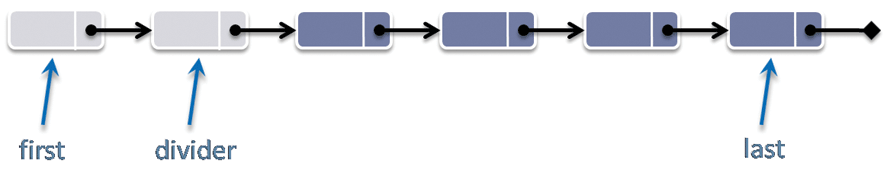
In the code that follows, I'm going to highlight the key reads and writes of such a variable; these variables should leap out of the screen at you, and you should get used to being very aware of every time you touch one.

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]):

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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 |

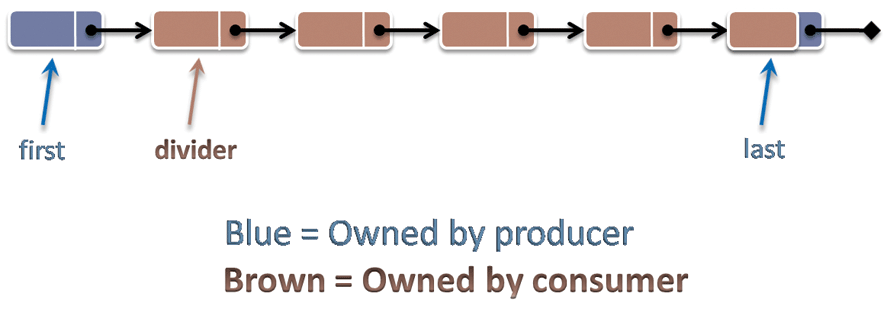
The constructor simply initializes the list with a dummy element. The destructor (in C# or Java, the dispose method) releases the list. In a future column, I'll discuss in detail why constructors and destructors of a shared object don't need to worry about concurrency and races with methods of the same object; the short answer for now is that creating or tearing down an object should always run in isolation, so no internal synchronization needed.

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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:

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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 two threads.

**Herb continues his exploration of lock-free code--this time focusing on creating a lock-free queue.**

**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/210600279](http://www.ddj.com/cpp/210600279)).

[2] P. Marginean. "Lock-Free Queues" (DDJ, July 2008). ([www.ddj.com/208801974](http://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.

**Acknowledgment**

Thanks to Tim Harris for his comments on drafts of this article.

**Writing Lock-Free Code: A Corrected Queue**

By Herb Sutter, September 29, 2008

[7 Comments](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448#disqus_thread)

**Herb continues his exploration of lock-free code--this time focusing on creating a lock-free queue.**

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).

**Writing Lock-Free Code: A Corrected Queue**

By Herb Sutter, September 29, 2008

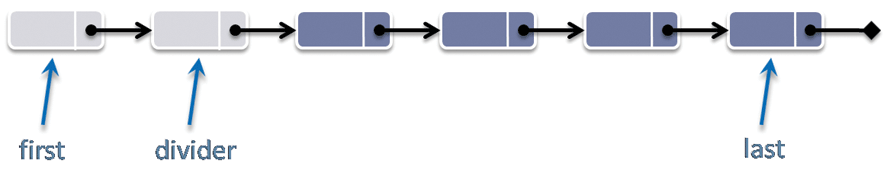
[7 Comments](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448#disqus_thread)

**Herb continues his exploration of lock-free code--this time focusing on creating a lock-free queue.**

**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]):

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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 |

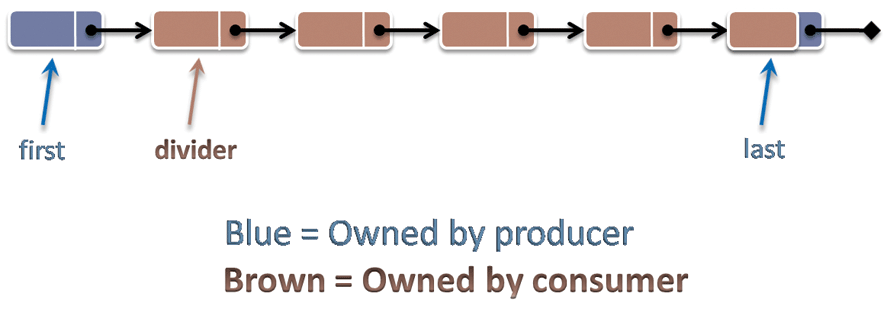
The constructor simply initializes the list with a dummy element. The destructor (in C# or Java, the dispose method) releases the list. In a future column, I'll discuss in detail why constructors and destructors of a shared object don't need to worry about concurrency and races with methods of the same object; the short answer for now is that creating or tearing down an object should always run in isolation, so no internal synchronization needed.

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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:

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:

[?](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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 two threads.

**Writing Lock-Free Code: A Corrected Queue**

By Herb Sutter, September 29, 2008

[7 Comments](http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448#disqus_thread)

**Herb continues his exploration of lock-free code--this time focusing on creating a lock-free queue.**

**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/210600279](http://www.ddj.com/cpp/210600279)).

[2] P. Marginean. "Lock-Free Queues" (DDJ, July 2008). ([www.ddj.com/208801974](http://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.

**Acknowledgment**

Thanks to Tim Harris for his comments on drafts of this article.

*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*](http://www.gotw.ca)*.*

**Writing a Generalized Concurrent Queue**

By Herb Sutter, October 29, 2008

[5 Comments](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363#disqus_thread)

**Herb tackles the general problem of supporting multiple producers and multiple consumers with as much concurrency as possible.**

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.

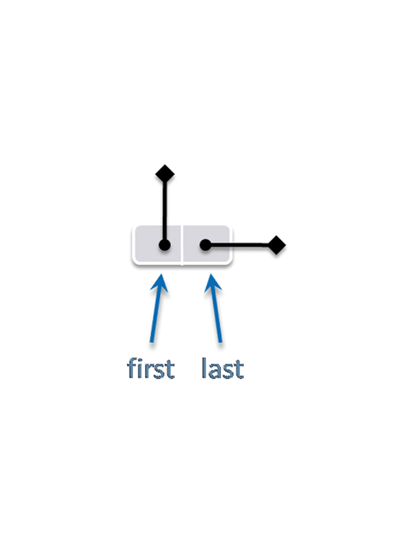
Second, we'll have the nodes allocate the contained T object on the heap and hold it by pointer instead of by value. [2] To experienced parallel programmers this might seem like a bad idea at first, because it means that when we allocate each node we'll also need to perform an extra heap allocation, 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.

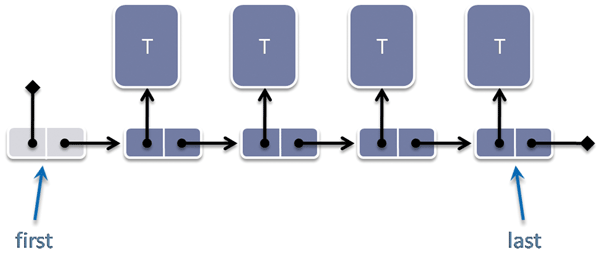
Fourth, we want to follow the advice that "if variables A and B are not protected by the same mutex and are liable to be used by two different threads, keep them on separate cache lines" to avoid false sharing or "ping-ponging" which limits scalability. [3] In this case, we want to add padding to ensure 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.



**Figure 1: Structure of an empty queue.**



**Figure 2: A queue containing four T objectsempty queue.**

Each node holds the T object by pointer and adds padding:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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.

Next, here are our queue control variables:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | 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:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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;      }    } |

**Writing a Generalized Concurrent Queue**

By Herb Sutter, October 29, 2008

[5 Comments](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363#disqus_thread)

**Herb tackles the general problem of supporting multiple producers and multiple consumers with as much concurrency as possible.**

**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:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

[?](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4 | consumerLock = false;   // release exclusivity      return false;                  // report queue was empty    }  }; |

**Writing a Generalized Concurrent Queue**

By Herb Sutter, October 29, 2008

[5 Comments](http://www.drdobbs.com/parallel/writing-a-generalized-concurrent-queue/211601363#disqus_thread)

**Herb tackles the general problem of supporting multiple producers and multiple consumers with as much concurrency as possible.**

**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.

*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*](http://www.gotw.ca)*.*