Lock-Free Code: A False Sense of Security

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

September 08, 2008
URL:http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/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).