C++

February 26, 2015

Mike Spertus

mike_spertus@symantec.com

Lock ordering

- If you want to avoid deadlocks, you want to acquire locks in the same order!
  - Suppose thread 1 acquires lock A and then lock B
  - Suppose thread 2 acquires lock B and then lock A
  - There is a window where we could deadlock with thread 1 owning lock A and waiting for lock B while thread 2 owns lock B and is waiting for lock A forever
- The usual best practice is to document an order on your locks and always acquire them consistent with that order
- See http://www.ddj.com/hpc-high-performance-computing/204801163

Sometimes it is hard to fix a lock order

- From http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-7-locking-multiple-mutexes.html
- Consider

```
class account {
 mutex m;
 currency_value balance;
public:
 friend void transfer(account& from,account& to,
                      currency_value amount) {
   lock_guard<mutex> lock_from(from.m);
   lock_guard<mutex> lock_to(to.m);
   from.balance -= amount;
   to.balance += amount;
 }
};
```

- If one thread transfers from account A to account B at the same time as another thread is transferring from account B to account A: Deadlock!

C++ provides a solution

- `std::lock(…)` allows you to acquire multiple locks “at the same time”
  - It actually will try releasing locks and then acquiring in different orders until no deadlock occurs
- ```
  class account {
   mutex m;
   currency_value balance;
  public:
   friend void transfer(account& from,account& to,
                        currency_value amount) {
     lock(from.m, to.m);
     lock_guard<mutex> lock_from(from.m, adopt_lock);
     lock_guard<mutex> lock_to(to.m, adopt_lock);
     from.balance -= amount;
     to.balance += amount;
   }
  };
  ```
- After acquiring the lock, “adopt” it into the lock_guard to manage the lifetime of the lock.
  ```

C++ Threads vs Operating System Threads

- Since C++ would like to work with as many operating systems as possible, it provides a “least common denominator” approach
- As a result, it may not support the specific threading features on your platform
- For example, C++11 has no notion of thread priority, but if your operating system supports it, you may want to take advantage of the ability to prioritize threads

Accessing non-portable thread functionality

- If you need to access some OS-provided thread functionality that is not built into C++, use `thread`'s `native_handle` method
  - ```
    thread t(thrFunc);
    pthread_setschedprio
       (t.native_handle(), 12);
    ```
    ```

Beware that advanced thread features have a lot of pitfalls

- For example, setting thread priority like above can cause a “priority inversion” deadlock
- This almost bricked the Mars Lander
  - Low priority once-per-day meteorological data gathering thread locks the message bus when it send the data
  - High priority information thread blocked waiting to get the message bus lock
  - Medium priority communication thread that doesn’t need the lock is higher priority than the meteorological thread, so it gets all the CPU cycles
  - The meteorological thread is starved and never completes its work, so it never releases the lock
- Some operating systems offer locks with “priority inheritance” to fix this, but the C++ standard says nothing about whether your C++ mutexes have priority inheritance enabled
  - Note that C++ mutexes do have a `native_handle()` method, which is helpful in cases like this

C++11 atomics

- Sometimes you just want a variable that you can read and update from multiple threads
- Using locks seems a little too complicated for that
- Fortunately, C++11 has a library of atomic types that can be shared between threads

An atomic counter

- You can read an atomic with its `load()` method, write it with its `store()` method and (usually) increment or decrement it with `++` or `--`
- Here's how you'd allow a bunch of threads to increment a global task counter
  - ```
    atomic<unsigned> tasksCompleted;
    void doTask() {
      /* ... */
      // Next line gives right result even if
      // called from multiple threads simultaneously
      tasksCompleted++;
    }
    void reportsTasksCompleted() {
      cout << tasksCompleted.load();
    }
    ```
    ```

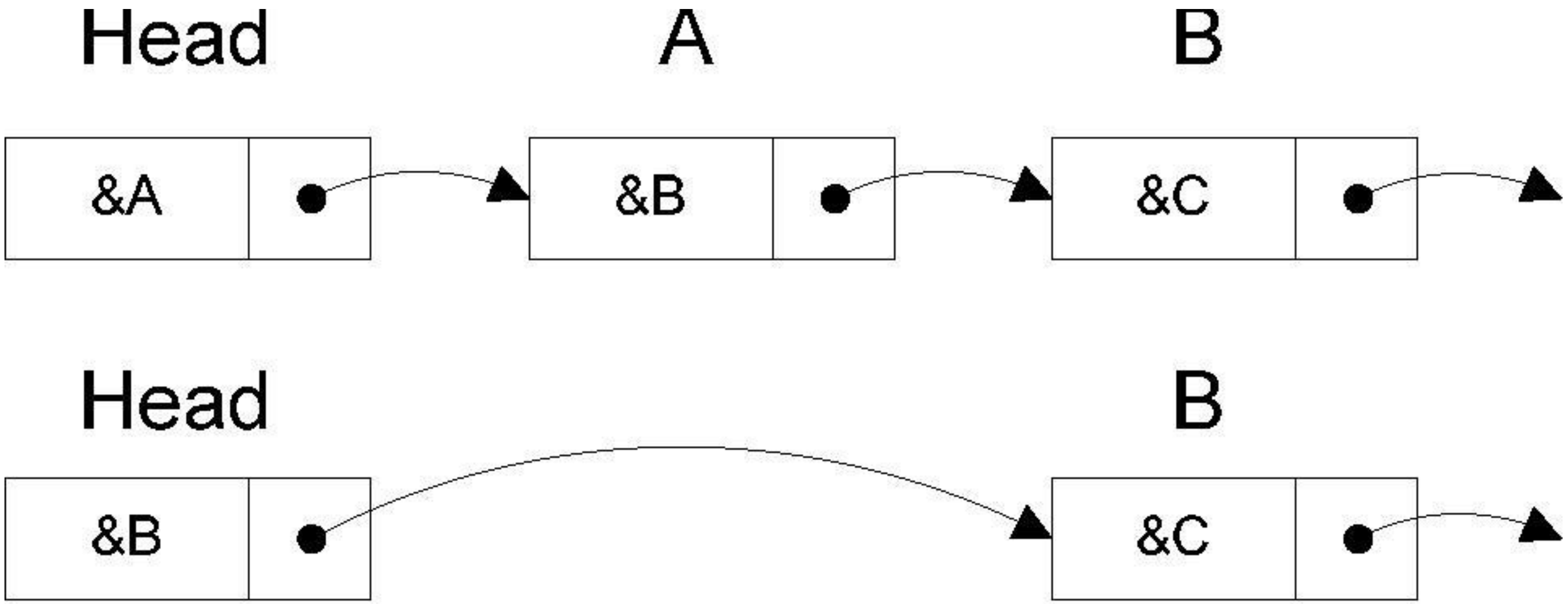
CASE STUDY ON THE RISKS AND REWARDS OF TRYING TO (OVER?) OPTIMIZE MULTITHREADED CODE

Background: How to quickly allocate objects of a fixed size?

- Say we're allocating 32-byte objects from 4096-byte pages
- Divide each page in our memory pool into 128 objects in a linked list
- Now, allocate and deallocate 32-byte objects from the list by pushing and popping
  - Fewer than a dozen instructions vs hundreds in a conventional allocator
  - Make sure you lock for thread-safety
- You will implement such a lock-based stack as an exercise

Allocating an object

- Pop the first object off the list

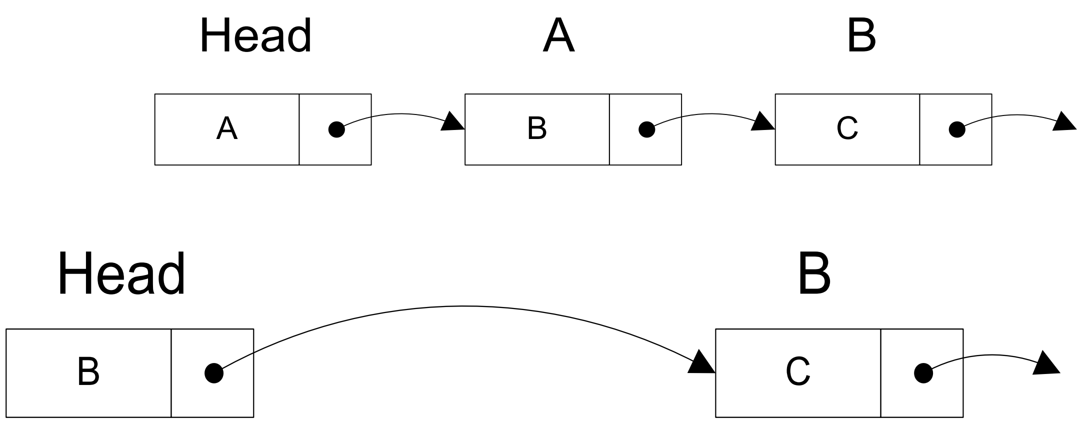


A True Story with a Twist—The Bad Beginning

- A programmer released an application using a linked list allocator like in the previous slide
  - It appeared to speed up his program considerably
- His customers reported that the application become slow as the number of threads increased into the hundreds
- Even though the lock only protects a few instructions, if a thread holding the lock loses its quantum, the list is unavailable until that thread gets another timeslice (perhaps hundreds of quanta later)
- Not acceptable

Can we make a thread-safe list without locks?

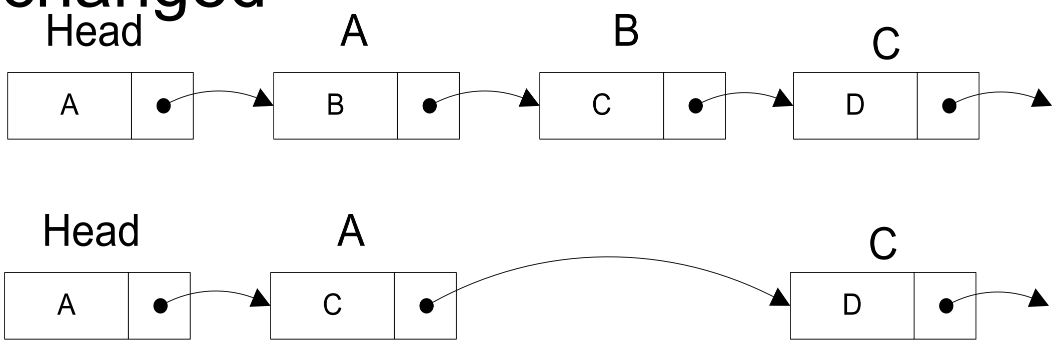
- To Remove an element



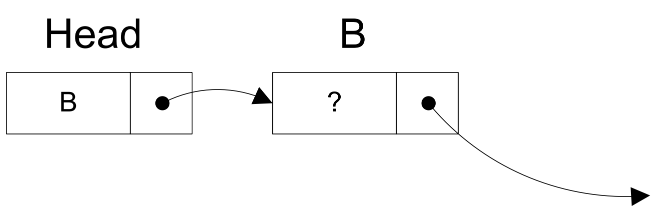
- We need a lock because we need to both return A and update the head to point to B (i.e., A's link) atomically
- Or do we?
- C++11 has an atomic `compare_and_exchange_weak` primitive that does a swap, but only if the target location has the value that we expect
  - Then our update would fail if someone messed with the list in the critical section
  - If so, just loop back and try again

Oops! Doesn't quite work

- Some other thread could do two pops and one push during the critical section, leaving the head unchanged

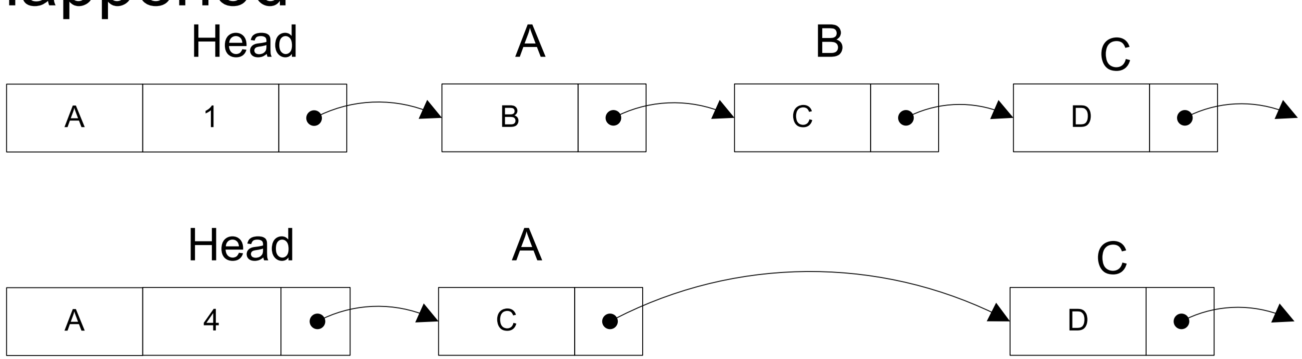


- After the `compare_and_exchange_weak`, B is erroneously back on the list



We can fix this

- Add a “list operation counter” to the head
- Update with 64-bit compare and exchange (on a 32-bit program), which C++ conveniently provides (and maps onto a single x86 instruction provided for just this reason)
- Now the compare and swap fails if intervening list ops happened



What's the point?

- This is much better
- No need for memory barrier
- Only one atomic operation instead of two
- If thread loses its quantum while doing the list operation, other threads are free to manipulate the list
  - This is the big one
- Works on x86-32, x86-64, and Sparc

How is this implemented in C++?

- See lockFreeStack.h in chalk
- Let’s look at it now

What about PPC and Itanium?

- Even better, PPC and Itanium have Linked Load and Store Conditional (LLSC)
- lwarx instruction loads from a memory address and “reserves” that address
- stwcx instruction only does a store if no intervening writes have been made to that address since the reservation
- Exactly what we want

What about push?

- The same techniques work for pushing onto the list
  - Exercise to see if you understand
- Not just restricted to lists
  - Many other lock-free data structures are known
  - See the references

A True Story with a Twist—A Happy Ending?

- The programmer switched to using Compare and Exchange-based atomic lists on Sparc
- The customers were happy with the performance
- But wait…

No happy ending?

- The customers started to experience extremely intermittent list corruption
- Virtually impossible to debug
  - He ran 100 threads doing only list operations for hours between failures
- The problem was that Solaris interrupt handlers only saved the bottom 32-bits of some registers
  - Timer interrupts in the critical section corrupted the compare and exchange
  - Fix: Restrict list pointers to specific registers
- Moral: The first rule of optimization is “Don’t!”
  - These techniques are powerful but only used where justified

But wait, there’s more

- Later, the program started being used on massively SMP systems, and it started to exhibit performance problems
  - The Compare and Exchange locked the bus to be thread-safe but that is expensive as the number of processors went up (this results in a surprising implementation of the Windows Interlocked exchange primitive).
- Since they no longer needed many more threads than processors, they went back to a lock-based list

So should you do a class-specific allocator?

- Do you really want to pollute your class with deep assumptions about the HW and OS?
- Do you want to update it everytime there is a new OS rev?
  - Early version of this before threading inadvertently made classes thread unsafe
- The answer is almost always, “No,” but…

No way! Except…

- My friend’s product wouldn’t have been usable without a custom memory manager
- He wouldn’t have sold his company for a large sum of money without usable products
- Use it when necessary, but only if you can justify the costs of maintaining your code over every present and future OS/hardware revision
- This story illustrates the real power and danger of using C++
  - Know the difference between “use” and “abuse”

C++11 Memory Model

- Sequential Consistency in the absence of race conditions
  - This basically means that if data is shared between threads, you must use an atomic or lock
- Herb Sutter atomic<> Weapons
  - http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

Memory model best practices

- Here are the takeaways
  - Try to avoid sharing data between threads except when necessary
  - When you share data between threads, always use locks or atomics to ensure both threads have a coherent view of the shared data
- A good reference
  - Boehm, Adve, "*You Don't Know Jack about Shared Variables of Memory Models: Data Races are Evil*" Communications of the ACM 55, 2 Feb. 2012
  - http://queue.acm.org/detail.cfm?id=2088916

Constructors for thread-safe classes

- Writing thread-safe copy constructors is difficult because it is difficult to lock the source

```
struct A {
  A(A const &a) : i(a.i) {
    // Too late, because we read
    // a.i before we locked
    lock_guard<mutex> l(a.mtx);
  }
  int &i;
  mutex mtx
};
```

New feature: Delegating constructors

- Often one constructor is a special case of another
- In C++98, the following is illegal

```
struct A {
  A(int i) { … // Do a lot }
  A(string s) : A(s.size()) {}
};
```

- In C++11, it just works
- More info at [Delegating Constructors (revision 3)]()

Solution by delegating copy constructor

- ```
  struct A {
     A(A const &a)
        : A(a, lock_guard<mutex>(a.mtx)) {}
   private:
     A(A const &a, const lock_guard<mutex>& )
       : i(a.i) {
     }
     int &i;
     mutex mtx
  };
  ```
- See http://www.justsoftwaresolutions.co.uk/threading/thread-safe-copy-constructors.html for details
  ```

C++-specific threading best practices

RAII

- Use a scoped locking class whose destructor releases the lock to make sure locks get released even when exceptions bypass normal control flow
  - Typically, this means to use the std::lock_guard class, like we do in the false sharing example
  - At work, I (Mike) just had a critical customer defect this week because manual unlocking code was bypassed by an exception.
    - Moral: Don’t rely on manual unlocking code!

Thread arguments

- For most thread systems, the thread creation function takes an arbitrary pointer argument.
  - This allows you to pass thread-specific creation info
- Never pass the address of a local variable because an exception in the creating function will cause a dangling pointer in the new thread

shared_ptr

- Since shared_ptrs delete their target whenever the reference count goes to zero, it is very difficult to know what locks will be held when the target classes destructor is called.
- Great care (or even handle/proxy classes that schedule destruction in a different thread) may be necessary to avoid violating lock ordering.
- When possible, avoid this complexity by not locking in destructors of class that may be managed by shared_ptrs.

Code that works for single and multithreaded

- It is common to write library classes that need to work for both single and multi-threaded cases
  - Single-threaded: don’t lock
  - Multi-threaded: synchronize methods
- How do we do with a single codebase?

Solution use a policy class!

```
struct SingleThreading {
  // No locking
  typedef void *mutex; // dummy
 static void lock(mutex &m) {}
 static void unlock(mutex &m) {}
};
struct MultiThreading {
  typedef std::mutex mutex;
  void lock(mutex &m) {
    m.lock();
  }
  void unlock(mutex &m) {
    m.unlock();
  }
};

template<class ThreadingModel>
class MyClass
{
public:
  void myMethod() {
    ThreadingModel::lock(myMutex);
    ...
    ThreadingModel::unlock(myMutex);
  }
  typename ThreadingModel::mutex myMutex;
}
```

Homework

HW 8-1

- Is the following code OK? If not, how would you fix it?

```
#include<thread>
#include<mutex>
#include<iostream>
#include<ofstream>
using namespace std;
mutex coutMutex;
mutex outpMutex;
ofstream outp("output.txt"); // Open file as ostream

void thrFunc1() {
  lock_guard<mutex> coutLock(coutMutex);
  lock_guard<mutex> outpLock(outpMutex);
  cout << "thrFunc1 console output" << endl;
  outp << "thrFunc1 file output" << endl;
}

void thrFunc2() {
  lock_guard<mutex> outpLock(outpMutex);
  lock_guard<mutex> coutLock(coutMutex);
  cout << "thrFunc2 console output" << endl;
  outp << "thrFunc2 file output" << endl;
}

int main() {
  thread t1(thrFunc1);
  thread t2(thrFunc2);
  t1.join();
  t2.join();
}
```

HW 8-2

- Complete the implementation of lock free stacks by implement the push() method in LockFreeStack.h (posted on Chalk).

HW 8-3

- Since this lecture is on low-level systems programming and memory, it is a good chance to remind ourselves that computer memory stores numbers in binary
- Learn to count in binary on your fingers
  - See http://en.wikipedia.org/wiki/Finger_binary
  - We'll test this in class
- How high can you count on both hands?
- Extra credit: Count to 31 in 15 seconds or less

HW 8-4—Extra Credit

- There is a whopper of a mistake in the following article about constructor delegation and threadsafe constructors
  - http://www.justsoftwaresolutions.co.uk/threading/thread-safe-copy-constructors.html
- What is the mistake in article?
  - Please don't read the comments after the article until you have submitted this assignment