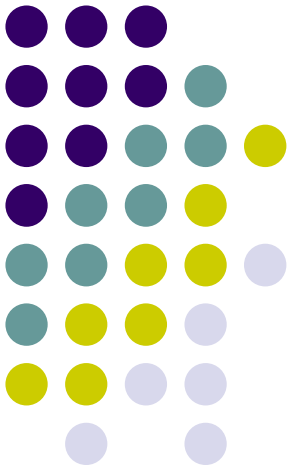
C++

February 5, 2013

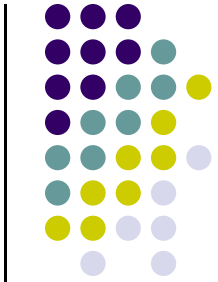
Mike Spertus

mike_spertus@symantec.com

YIM: spertus

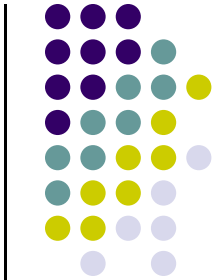


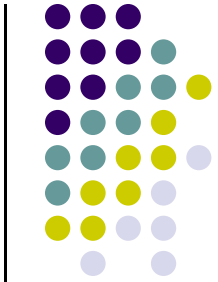
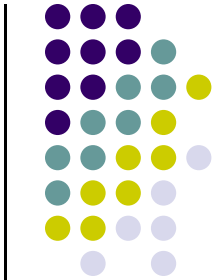
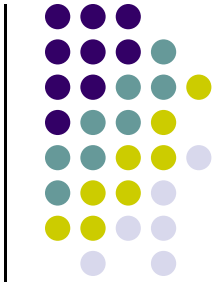
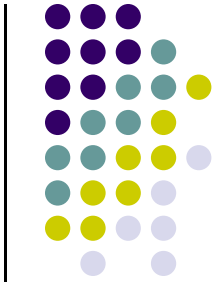
Exception Specifications



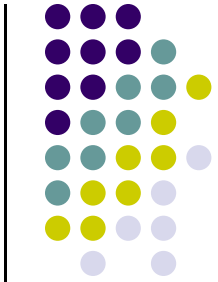
- You can declare what exceptions a method can throw
  - Sounds like a good idea
  - But they are broken
  - So badly that they're deprecated in C++11
- Summary: Don't use them
- http://www.gotw.ca/publications/mill22.htm

Noexcept

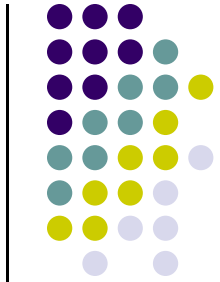


- As a partial replacement, there is a new noexcept specifier in C++11 that you can use to say your function won't throw
- http://en.cppreference.com/w/cpp/language/noexcept_spec:
- ```
  // whether foo is declared noexcept depends on if the
  expression
  // T() will throw any exceptions
  template <class T>
  void foo() noexcept(noexcept(T())) {}
  void bar() noexcept(true) {}
  void baz() noexcept { throw 42; }
  // noexcept is the same as noexcept(true)
  int main() {
   foo<int>();
   // noexcept(noexcept(int())) => noexcept(true), so fine
    bar(); // fine
    baz(); // compiles, but calls std::terminate at runtime
  }
  ```
  ```

Lambdas



- We have used C++11 lambdas in class
- [](int x, int y) { return x < y; } is an anonymous boolean valued “function” returning true if x < y
- “Function” is in quotes because it is actually an unspecified type. You can only assign to a function pointer if there is no capture list (see following slides), but you can always store as follows
  - auto f = [](int x, int y) { return x < y; }
  - function<bool(int, int)> f = [](int x, int y) { return x < y; }

Lambda return values



- If your lambda just consists of a return statement, the compiler infers the type.
- If not, you give it using the “unified function syntax”

```
[](int x, int y) -> int {
  int z = x + y; return z + x;
}
```

- This syntax avoids parsing problems

Capture lists



- To capture local variables by reference, use [&]
- To capture local variables by value, use [=]
- You can get finer-grained if necessary

Lambdas and algorithms



- Now all of the standard library algorithms can be used as easy as for loops
- ```
  std::vector<int> someList; // Wikipedia
  int total = 0;
  std::for_each
   (someList.begin(), someList.end(),
    [&](int x) { total += x; });
  ```
  ```

Parallel accumulate

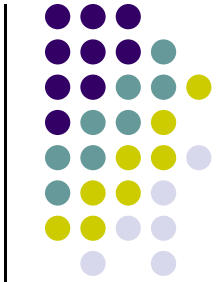


- It would be really nice to have an implementation of std::accumulate that breaks up its input into pieces, adds up each piece in parallel and then adds up the results from each of the pieces
- Let’s do this with futures
- async_accumulate_function.cpp



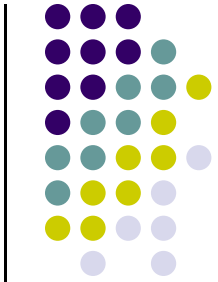
MEMORY MODEL

Why we need a memory model



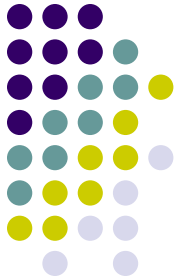
- The following slides are based on Hans Boehm's PLDI paper "Threads cannot be implemented as a library"
- A memory model says how changes made to memory by one thread are seen in another

WARNING!



- The next several slides are very confusing
- You do not need to learn them in detail (or at all) as long as you follow the best practices given in slide 41
- However, we give these slides for two reasons
  - Without seeing such bizarre unexpected behavior, one would be tempted to ignore the “obviously too strict” guidelines in slide 41
  - They are very interesting

What can r1 and r2 end up as? (Boehm)



Initially x = y = 0;

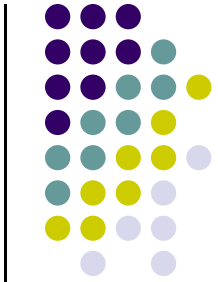
Thread 1

```
x = 1;
r1 = y;
```

Thread 2

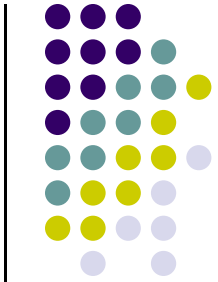
```
y = 1;
r2 = x;
```

Answer: Any combinations of 0 and 1!



- Intuitively r1 == r2 == 0 impossible
- Practically, the compiler (or the hardware) may reorder the statements because it doesn't matter within a given thread which order the assignments take place
- However, it does matter if the variables are used by another thread at the same time and we could end up with both r1 and r2 being 0
  - Note: Under pthreads rules this is simply illegal

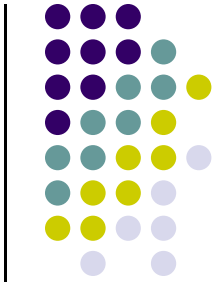
If q = 0, what can another thread see count as? (Boehm)



```
[count is global]
for (p = q; p!= 0; p = p->next) {
    count++;
}
```

- Other threads may see `count == 1`!
- Compiler may rewrite code by speculatively incrementing `count` before the loop, and decrementing if necessary at the end!
- Even gcc –O2 does this.

Is this code correct?



```
class A {
public:
    virtual void f();
};
A *a; // Global variable
```

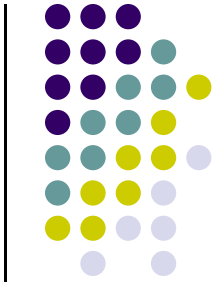
Thread 1

```
a = new A;
```

Thread 2

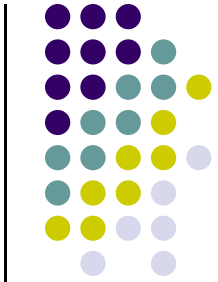
```
if (a) a->f();
```

Not on modern multiprocessor computers!



- Writes made on one processor may not be seen in the same order on another processor!
  - Allows microprocessor designers to use write buffers, instruction execution overlap, out-of-order memory accesses, lockup-free caches, etc.
- Thread 2 may see the assignment to a before it sees the vtable of the new A object!
- If that happens, the `a->f()` call will crash!
- Modern processors use *Weak Consistency*

Weak memory consistency

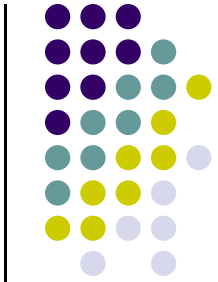


In a multiprocessor system, storage accesses are weakly ordered if (1) accesses to global synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed, and if (3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

—Dubois, Scheurich, Briggs (1986)

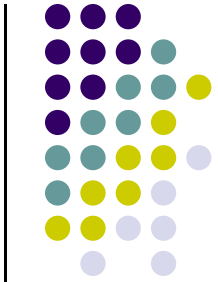
If the compiler does not have a notion of synchronizing variables, the above says nothing! Prior to C++0x, this is addressed non-portably by vendor-specific synchronization extensions to C++.

C++11 Memory Model

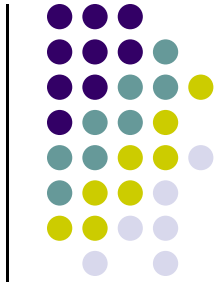


- Sequential Consistency in the absence of race conditions

This is too complicated

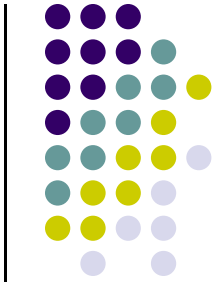


- I agree
- Here are the takeaways
  - Try to avoid sharing data between threads except when necessary
  - When you share data between threads, always use locks (or atomics, which we will discuss next week) to explicitly serialize accesses to the shared data



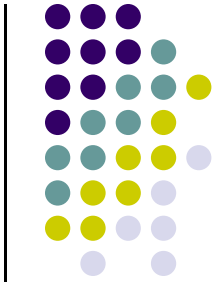
CACHE CONSIDERATIONS

Warning



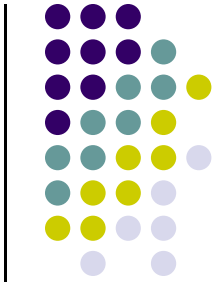
- Again, the next several slides are very scary, so we end with a distilled set of best practice rules on slides 50 and 51

Caches



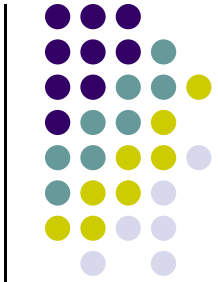
- Accessing main memory can take a processors hundreds of cycles
- Therefore, processors use high-speed caches to maintain local copies of data
  - See http://mugur.roz.md/computer-science/computer-science/images/fig06_19_0.jpg
  - If another processor needs to read/write that memory, it needs to force other processors to flush or invalidate any cached copies of the memory
    - See http://en.wikipedia.org/wiki/Cache_coherency

Cache lines and false sharing



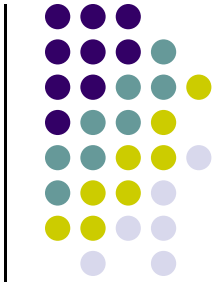
- When data is moved from main memory to cache, enough data is always moved to fill a “cache line.”
  - The size of a cache line varies by processor and needs to be looked up in the processor datasheet. A typical size would be 32 bytes, but it varies greatly.
- As a result, if two processors are modifying data within 32 bytes, they are constantly forcing each other to invalidate their cache (“false sharing”)

False sharing example



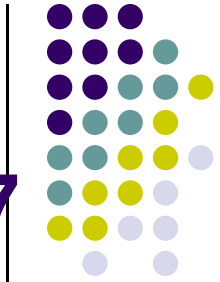
- Look at false_sharing.cpp from chalk
- On my laptop (single processor), changing `padSize` has no effect on performance
- On my desktop (2 quad-core processors), it runs 6 times as fast with `padSize=512` as `padSize=1`!
  - False sharing!
- This is very insidious because code that performs well on development machines often performs very badly on multiprocessor servers!

Direct-Mapped Caches



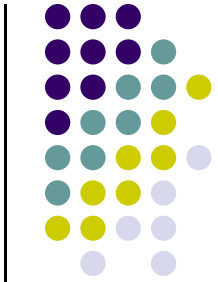
- Often a single memory location can only be mapped to one or two possible cache lines
  - See http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/direct.html
- Why do I need to know this?
  - Later in this lecture we'll discuss how direct-mapped caches caused the widely-used performance-sensitive 'ghostscript' program to spend one third of its time in cache misses.

Cache-conscious programming (Adapted from Herlihy&Shavit p. 477



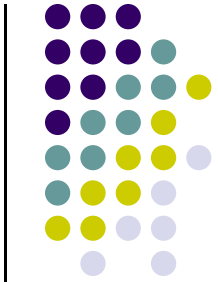
- Objects or fields that are accessed independently should be aligned and padded so they end up on different cache lines.
- Keep read-only data separate from data that is modified frequently.
- When possible, split an object into thread-local pieces. For example, a counter used for statistics could be split into an array of counters, one per thread, each one residing on a different cache line. While a shared counter would cause invalidation traffic, the split counter allows each thread to update its own replica without causing cache coherence traffic.
- If a lock protects data that is frequently modified, then keep the lock and the data on distinct cache lines, so that threads trying to acquire the lock do not interfere with the lock-holder's access to the data.
- If a lock protects data that is frequently uncontended, then try to keep the lock and the data on the same cache lines, so that acquiring the lock will also load some of the data into the cache.
- If a class or struct contains a large chunk of data whose size is divisible by a high power of two, consider separating it out of the class and holding it with an `auto_ptr` to avoid the Ghostscript problem from lecture 5.

Lock ordering

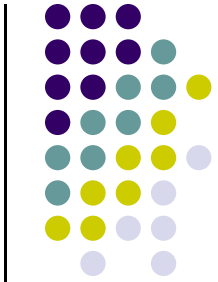


- If you want to avoid deadlocks, you must always acquire locks in the same order!
- See
  http://www.ddj.com/hpc-high-performance-computing/204801163

Warning: `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`.

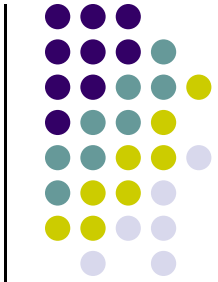


CASE STUDY ON THE RISKS AND REWARDS OF TRYING TO (OVER?) OPTIMIZE MULTITHREADED CODE

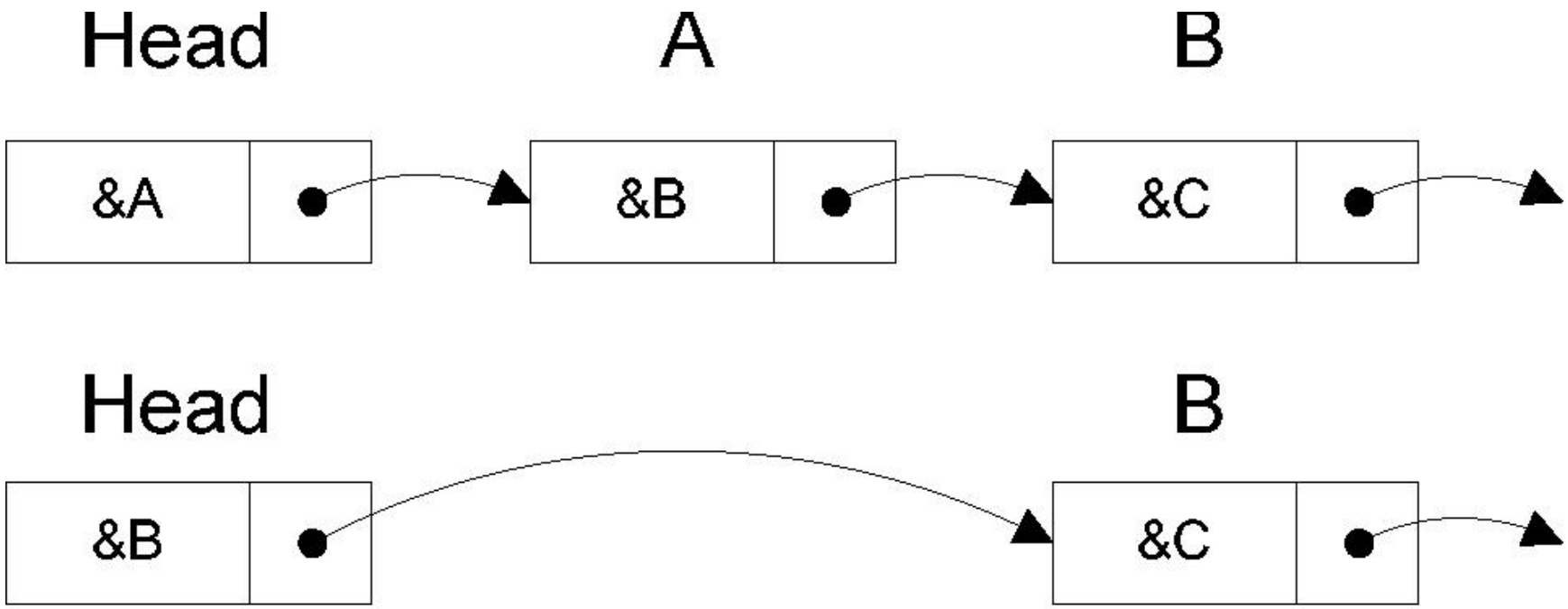
Background: How to quickly allocate objects of a fixed size?

- 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

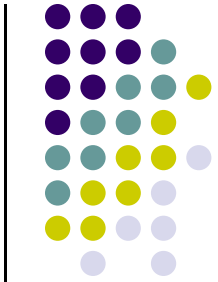
Allocating an object



- Pop the first object off the list



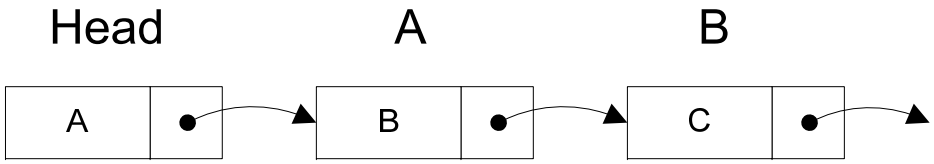
A True Story with a Twist—The Bad Beginning

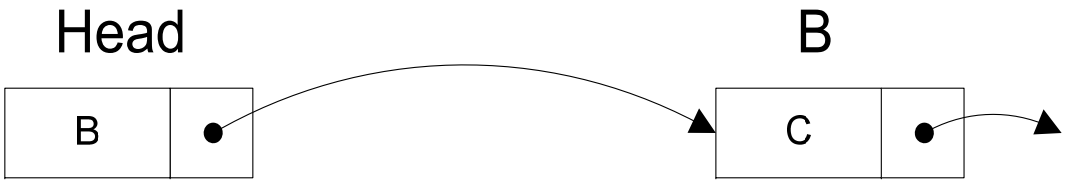


- A programmer released an application using a linked list allocator like in the previous slide
  - It appeared to speed up his program considerably
- His customers reported that the application become slow as the number of threads increased into the hundreds
- Even though the lock only protects a few 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?

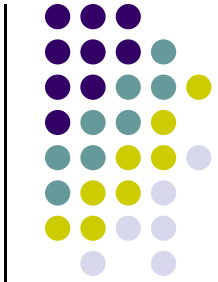
- To Remove an element



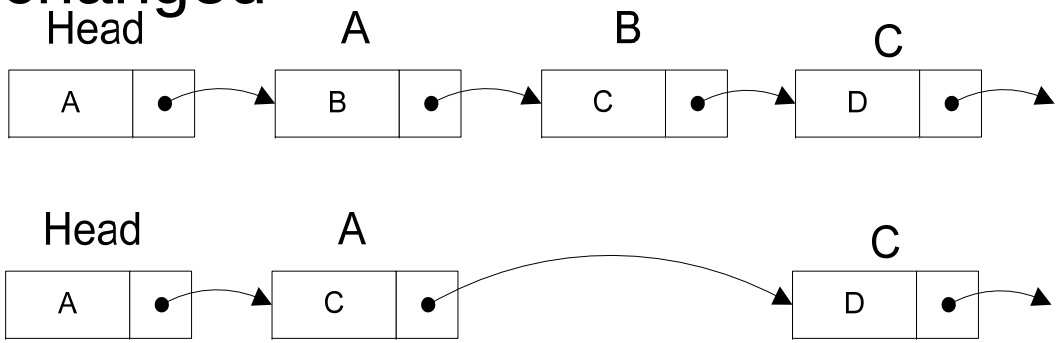


- We need a lock because we need to both return A and update the head to point to B (i.e., A's link) atomically
- Or do we?
- C++11 has an atomic `compare_and_exchange_weak` primitive that does a swap, but only if the target location has the value that we expect
  - Then our update would fail if someone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

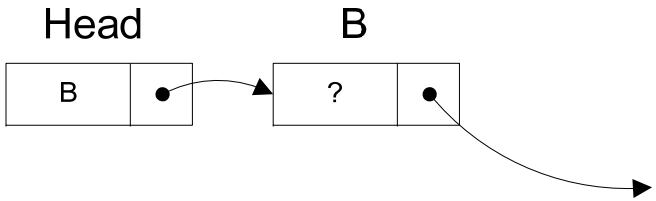
Oops! Doesn't quite work



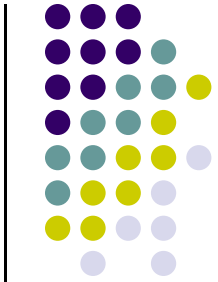
- Some other thread could do two pops and one push during the critical section, leaving the head unchanged



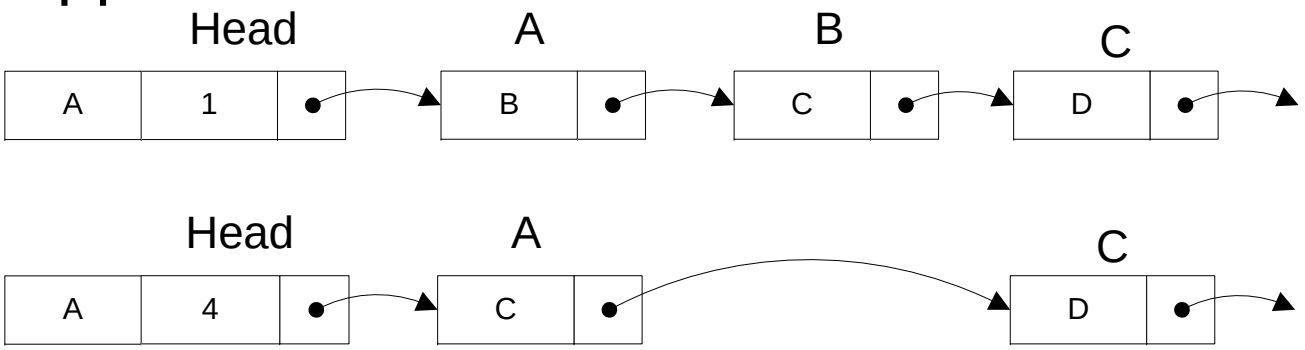
- After the `compare_and_exchange_weak`, B is erroneously back on the list



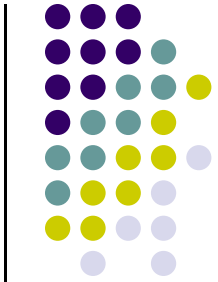
We can fix this



- 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

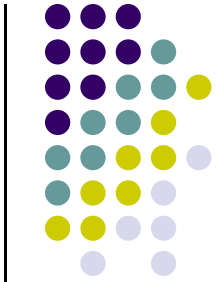


What's the point?



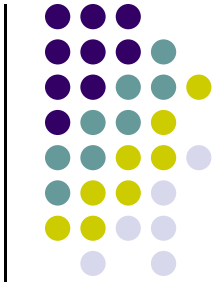
- This is much better
- No need for memory barrier
- Only one atomic operation instead of two
- If thread loses its quantum while doing the list operation, other threads are free to manipulate the list
  - This is the big one
- Works on x86-32, x86-64, and Sparc

What about PPC and Itanium?



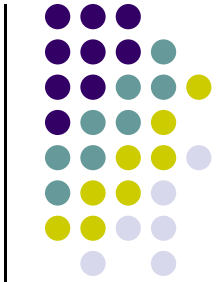
- Even better, PPC and Itanium have Linked Load and Store Conditional (LLSC)
- lwarx instruction loads from a memory address and “reserves” that address
- stwcx instruction only does a store if no intervening writes have been made to that address since the reservation
- Exactly what we want

What about push?



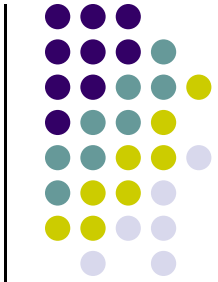
- The same techniques work for pushing onto the list
  - Exercise to see if you understand
- Not just restricted to lists
  - Many other lock-free data structures are known
  - See the references

A True Story with a Twist—A Happy Ending?



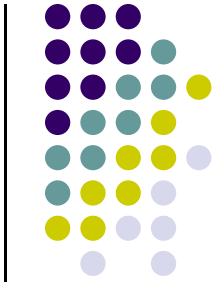
- The programmer switched to using Compare and Exchange-based atomic lists on Sparc
- The customers were happy with the performance
- But wait…

No happy ending?



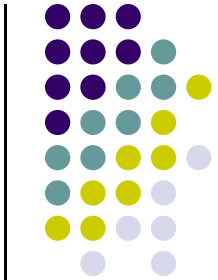
- The customers started to experience extremely intermittent list corruption
- Virtually impossible to debug
  - He ran 100 threads doing only list operations for hours between failures
- The problem was that Solaris interrupt handlers only saved the bottom 32-bits of some registers
  - Timer interrupts 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



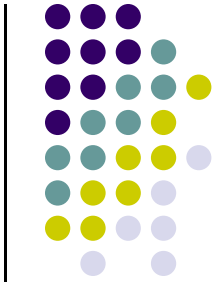
- Later, the program started being used on massively SMP systems, and it started to exhibit performance problems
  - The Compare and Exchange locked the bus to be thread-safe but that is expensive as the number of processors went up (this results in a surprising implementation of the Windows Interlocked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

Another example



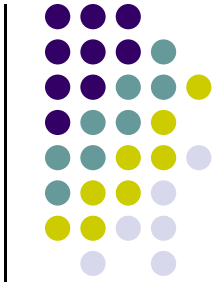
- The popular postscript rendering program ghostscript was originally written by Peter Deutsch, who wrote a custom memory manager. It is certainly true that malloc()/free() performance is critical in postscript and Peter Deutsch was a memory management expert, having coauthored the first high-performance Smalltalk implementation.
- Peter Deutsch used a custom allocator along the lines we just discussed, with a free list of pages that could be subdivided into fixed size objects
- Tests 10 years later showed that ghostscript's memory manager was actually slowing it down by 30%

What went wrong?



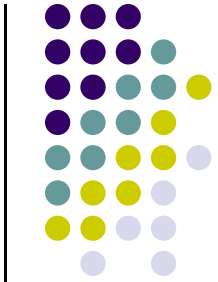
- The custom allocator maintained a pool of free-pages in a linked list, with the first word of each free page as a pointer to the next free page. As this code was developed on a machine without a direct-mapped cache, it ran fine. However, on machines with direct-mapped caches, all of the freelist pointers mapped to the same cache line causing a cache miss on each step of walking through the freelist. Ghostscript was spending about a third of its time in cache misses from walking through the page freelist.
- Note: You don’t need to understand this as long as you understand the moral

So should you do a class-specific allocator?



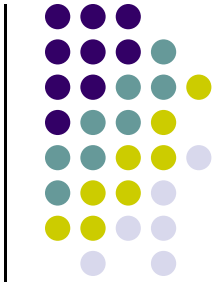
- Do you really want to pollute your class with deep assumptions about the HW and OS?
- Do you want to update it everytime there is a new OS rev?
- Early version of this before threading inadvertently made classes thread unsafe
- Are you smarter than Peter Deutch?
- The answer is almost always, “No,”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

No way!
Except…



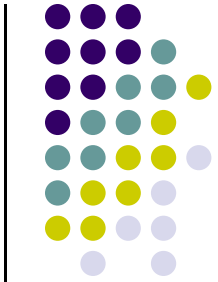
- My friend’s product wouldn’t have been usable without a custom memory manager
- He wouldn’t have sold his company for a large sum of money to IBM without usable products
- Use it when necessary, but only if you can justify the costs of maintaining your code over every present and future OS/hardware revision

HW 5-1



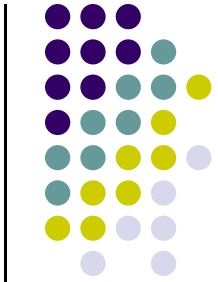
- Use a lambda to sort a vector of ints by absolute value

HW 5-2



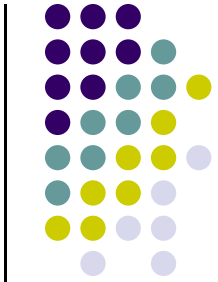
- Write a thread-safe stack using locks
  - Just like the lock-free stack, the only operations will be push and pop, but use locks instead of compare and exchange

HW 5-3



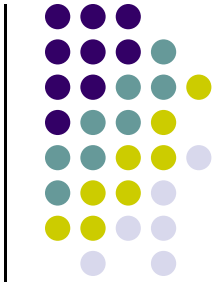
- Suppose your program has a counter that counts how many pieces of work have been done by a `doWork()` function.
- The simplest implementation would be to have `doWork()` increment a global variable "`unsigned workUnits`"
  - Of course, use a lock or atomics to increment it.
  - Why is this a bad design if you have many threads doing work?
- How would you fix this?
  - You may want to look up the C++11' `thread_local` keyword

HW 5-4 Extra credit



- The file lockFreeStack.h implements the lock-free pop method described in the lecture.
- Complete the definition of the push method to do a lock-free push.

HW 5-5: Extra credit exploration



- Here is a cautionary example on both the danger of trusting what you read on the internet (even if it is from a reputable source) and the trickiness of getting locking code correct. Unfortunately, this example is in C#, but you should be able to get the gist of it.
  - Moral: Don't blithely roll your own concurrency primitives and expect to get them right without significant careful and formal analysis
- Microsoft's gives documentation for how to implement a Producer-Consumer idiom at http://msdn.microsoft.com/en-us/library/yy12yx1f.aspx
  - Producer-Consumer means a set of producer threads add work to the queue, while a set of consumer threads remove work for the queue. It is one of the most common concurrency idioms.
- Is Microsoft's code is incorrect (you can assume multiple producer and consumer threads)? Why?
  - You may find it useful to carefully read the documentation on AutoResetEvents at http://msdn.microsoft.com/en-us/library/system.threading.autoresetevent.aspx. Although we discussed Auto-reset events in class, some of the particulars of Microsoft's implementation of them make a big difference here.
- For even more extra credit, show how to implement a correct producer-consumer idiom either in C# or C++