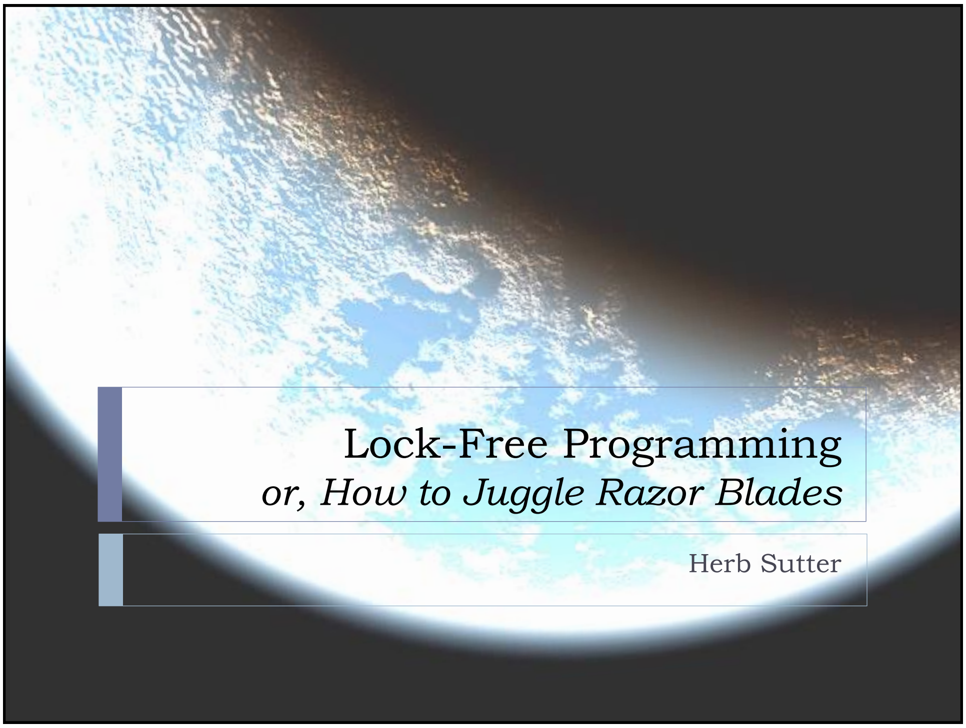


Synchronization Choices Overview

- Typical uses/techniques, and **examples** we'll consider:

	Synchronized externally (by the caller)	Synchronized internally	Both
Using locks	**Default, esp. when transaction involves multiple objects** Producer/Consumer: locks	"Synchronized" objects (Generally problematic)	
Using atomics ("lock-free")	Atomic handoffs Producer/Consumer: atomic mail slots Versioned state	"Lock-free" data structures slist	*(e.g., copy-on-write strings)*
Both	Different tools at different times Double-Checked Locking (DCL) Producer/Consumer: locks + atomics	config_map	

Why Lock-Free Code?

- **Concurrency and scalability.**
  - Eliminate/reduce blocking/waiting in algorithms and data structures.
- Avoid the troubles with (b)locking:

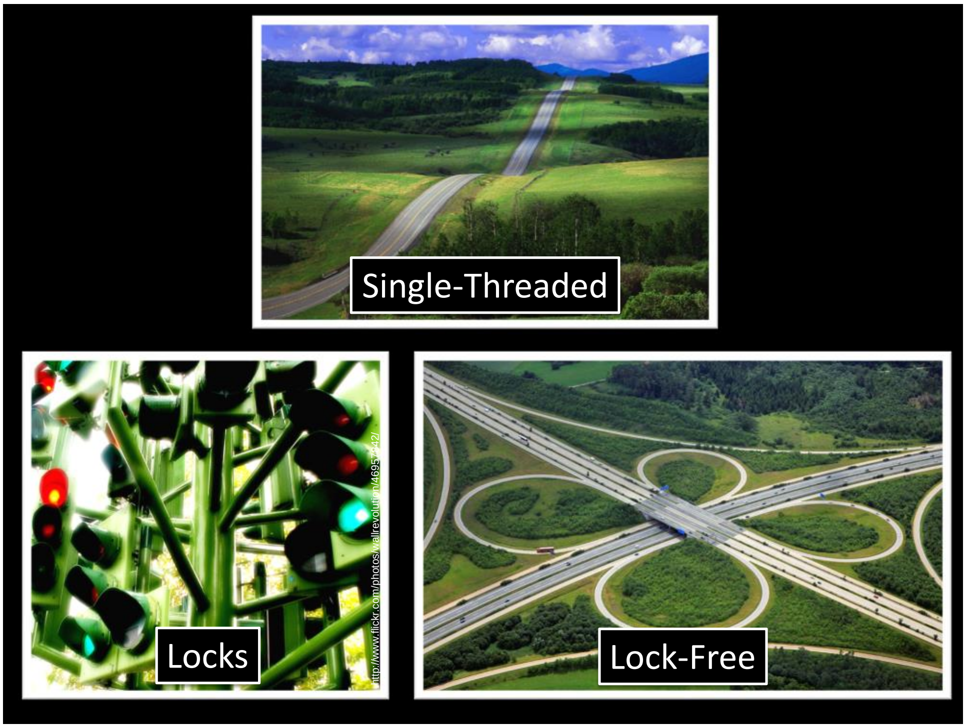
```
{
 lock_guard<mutex> lock1{ mutTable1 };
 lock_guard<mutex> lock2{ mutTable2 };
 table1.erase( x );
 table2.insert( x );
} // release mutTable2 and mutTable1
```

  - **Races:** Forgot to lock, or locked the wrong thing.
  - **Deadlock:** Locked in incompatible orders on different threads.
  - **Simplicity vs. scalability (convoying, priority inversion)?** Coarse-grained locking is simpler to program, but creates bottlenecks to kill scalability.
  - **Not composable.** In today's world, this is a deadly sin.

Important assumptions

(1) **You have already measured** performance/scalability and proven you have a high-contention data structure, before resorting to the techniques described in this talk.

(2) **You will measure again** after you write a hopefully-more-concurrent replacement using these techniques to ensure that it is actually an improvement.



Roadmap

- **Two Basic Tools**
  - **Transactional thinking + atomic<T>**
- Basic Example: Double-Checked Locking
  - It’s pretty easy to do right, but you still have to do it right
- Producer-Consumer Variations
  - Using locks, locks + lock-free, and fully lock-free
- A Singly Linked List: This Stuff Is Harder Than It Looks
  - Just find, push_front, and pop: How hard could it be?

enter critical region

exit critical region

Lock-Free Fundamental #1

- **Your key concept: Think in transactions (ACID).**
  - Atomicity:
    - A transaction is all-or-nothing; "commit" is atomic. Other code must not be able to see the data in a partially-updated state (i.e., a corrupt state).
    - [LF] Publish each change using one atomic write (read-modify-write).
  - Consistency, Isolation, Durability:
    - A transaction takes the data from one consistent state to another.
    - Two transactions never simultaneously operate on the same data.
    - A committed transaction is never overwritten by second transaction that did not see the results of the first transaction. (The "lost update" problem.)
    - [LF] Make sure concurrent updates don't interfere with each other (especially think about deletes!) or with concurrent readers.

Lock-Free Fundamental #2

- **Your key tool: The ordered atomic variable.**
  - C++11 "atomic<T>" and C11 "atomic_*".
  - Java "volatile T" and Atomic* (e.g., AtomicLong).
  - .NET "volatile T".
- Semantics and operations:
  - Each individual read and write is atomic, no locking required.
  - Reads/writes are guaranteed not to be reordered.
  - Compare-and-swap (CAS)… ***conceptually*** an atomic execution of:

```
bool atomic<T>::compare_exchange_strong( T& expected, T desired ) {
  if( this->value == expected ) { this->value = desired;    return true;  }
  else /* it's not */           { expected = this->value;  return false; }
}
```

  - + **compare_exchange_weak** for use in loops (is allowed to fail spuriously)
  - + **exchange** for when a "blind write" that returns the old value is sufficient
- Notes:
  - Limited to certain types that can be manipulated atomically.
  - An 'atomic T' may not have the same layout (e.g., alignment) as a plain T.

atomic<T> Notes

- **Lock-free** vs. lock-based implementations:
  - If T is a small type, including most built-ins, atomic<T> is implemented without locks (typically, platform-specific instructions).
  - For larger types, atomic<T> is implemented using a lock.
- **Initialization:** Remember to explicitly initialize – `atomic<int> ai{ 0 };`
- **Interleaving:** The state of the atomic<T> can change at any time between successive calls on this thread due to interleaved calls on other threads.
- **Granularity:** Logical transactions often operate on multiple objects, or on multiple calls to the same object. Example:

```
atomic<int> account1_balance = …, account2_balance = …;
account1_balance += amount;
account2_balance –= amount;
```

  - Those two lines still need to be externally locked, if some invariant doesn't hold in between the two calls.

Aside: Three Levels of "Lock-Freedom"

- **Wait-free (strongest, "no one ever waits"):** Every operation will complete in a bounded #steps no matter what else is going on.
  - Guaranteed system-wide throughput + starvation-freedom.
- **Lock-free ("someone makes progress"):** Every step taken achieves global progress (for some sensible definition of progress).
  - Guaranteed system-wide throughput.
  - All wait-free algorithms are lock-free, but not vice versa.
- **Obstruction-free (weakest, "progress if no interference"):** At any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a bounded number of steps will complete its operation.
  - No thread can be blocked by delays or failures of other threads.
  - Doesn't guarantee progress while two or more threads run concurrently (e.g., *deadlock* is impossible, but *livelock* could be possible).
  - All lock-free algorithms are obstruction-free, but not vice versa.

- Informally, "lock-free" ≈ "doesn't use mutexes" == any of these.

Roadmap

- Two Basic Tools
  - Transactional thinking + atomic<T>
- **Basic Example: Double-Checked Locking**
  - **It's pretty easy to do right, but you still have to do it right**
- Producer-Consumer Variations
  - Using locks, locks + lock-free, and fully lock-free
- A Singly Linked List: This Stuff Is Harder Than It Looks
  - Just find, push_front, and pop: How hard could it be?

Bad DCL

- The pattern relies on the flag value being atomic and ordered. But an ordinary variable doesn't guarantee these properties.

```
Widget* Widget::pInstance = nullptr;
Widget* Widget::Instance() {
  if( pInstance == nullptr ) {              // 1: first check
    lock_guard<mutex> lock( mutW );         // 2: acquire lock (crit sec enter)
    if( pInstance == nullptr ) {            // 3: second check
      pInstance = new Widget();             // 4: create and assign
    }
  }                                         // 5: release lock
  return pInstance;                         // 6: return pointer
}
```

- Issues:
  - **Race on pInstance: Anything can happen.** There is a race between lines 1 and 4. This could cause a wild pointer (e.g., pointer value tearing), possibly leading to random code execution.
  - **Reordering: Seeing partly-constructed objects.** If the parts of line 4 are reordered, pInstance could point to a partly-constructed object.

Correct Double-Checked Locking

- The Double-Checked Locking (DCL) pattern (un-“broken”).

```
atomic<Widget*> Widget::pInstance{ nullptr };
Widget* Widget::Instance() {
  if( pInstance == nullptr ) {              // 1: first check
    lock_guard<mutex> lock{ mutW };// 2: acquire lock (crit sec enter)
    if( pInstance == nullptr ) {            // 3: second check
      pInstance = new Widget();             // 4: create and assign
    }
  }                                         // 5: release lock (crit sec exit)
  return pInstance;                         // 6: return pointer
}
```

- Four key points, involving both atomicity and ordering:
  - 1: Test pInstance **atomically**.
  - 2: **Then**, if that fails, take the lock.
  - 3-4a: **Then** repeat the test and construct the object.
  - 4b: **Then** assign its this pointer **atomically** to pInstance.

Slight Optimization

- This may be slightly faster (1 vs. 2 atomic loads in the main case):

```
atomic<Widget*> Widget::pInstance{ nullptr };
Widget* Widget::Instance() {
  Widget* p = pInstance;
  if( p == nullptr ) {                      // 1: first check
    lock_guard<mutex> lock{ mutW };// 2: acquire lock (crit sec enter)
    if( (p = pInstance) == nullptr ) {      // 3: second check
      pInstance = p = new Widget();         // 4: create and assign
    }
  }                                         // 5: release lock (crit sec exit)
  return p;                                 // 6: return pointer
}
```

- The compiler is allowed to do this optimization for you, but:
  - it isn’t required to, and
  - it’s not common yet AFAIK.

Even Better: There's a Tool For That

- The general-purpose way to spell lazy initialization in C++11 is:

```
static unique_ptr<widget> widget::instance;
static std::once_flag widget::create;

widget& widget::get_instance() {
   std::call_once( create, [=]{ instance = make_unique<widget>(); } );
   return instance;
}
```

  - No raw *, automatic cleanup, and much lower boilerplate-to-real code ratio.

Best of All: There's a Tool For That

- The special-purpose way that you should use when you can (aka the Meyers Singleton!) is this:

```
widget& widget::get_instance() {
   static widget instance;
   return instance;
}
```

Roadmap

- Two Basic Tools
  - Transactional thinking + atomic<T>
- Basic Example: Double-Checked Locking
  - It's pretty easy to do right, but you still have to do it right
- **Producer-Consumer Variations**
  - **Using locks, locks + lock-free, and fully lock-free**
- A Singly Linked List: This Stuff Is Harder Than It Looks
  - Just find, push_front, and pop: How hard could it be?

Locks and Atomics In Combination

- The key requirement is that access to a given shared mutable object is synchronized consistently…
  - Using traditional locking. (Preferred, but sometimes problematic because locks don't compose well.)
  - Using a lock-free atomic<> discipline. (Less deadlock, but this style tends to be really hard today.)
- … at every given point in time.
  - It doesn't have to be the same for the lifetime of the object.
  - For example, consider handoff situations:
    - Threads 1..N share object x, synchronizing via mutex m1.
    - Then x is handed off and never looked at again by those threads.
    - Then Threads N+1..M shared x, synchronizing via mutex m2 (or via a lock-free discipline, or some other way).

Create and Publish Queue Items: 1 Producer, Many Consumers, Using Locks

- Thread 1 (producer):

```
while( ThereAreMoreTasks() ) {
  task = AllocateAndBuildNewTask();
  {
    lock_guard<mutex> lock{mut};  // enter critical section
    queue.push( task );
    cv.notify();                  //
  }                               // exit critical section
}

{
  lock_guard<mutex> lock{mut};    // enter critical section
  queue.push( done );             // add sentinel; that's all folks
  cv.notify();                    //
}                                 // exit critical section
```

Create and Publish Queue Items: 1 Producer, Many Consumers, Using Locks

- Threads 2..N (consumers):

```
myTask = null;
while( myTask != done ) {
  {
    lock_guard<mutex> lock{mut};  // enter critical section
    while( queue.empty() )        // if not ready, don't busy-wait,
      cv.wait( mut );             // release and re-enter crit sec
    myTask = queue.first();       // take task
    if( myTask != done )          // remove it if not the sentinel,
      queue.pop();                //   which others need to see
  }                               // exit critical section
  if( myTask != done )
    DoWork( myTask );
}
```

Quick Quiz: Where Must Those Pesky Lock-Protected Invariants Hold, Again?

- Threads 2..N (consumers):

```
myTask = null;
while( myTask != done ) {
  {
                          INVARIANTS HOLD
    lock_guard<mutex> lock{mut};  // enter critical section
    while( queue.empty() )        // if not ready, don't busy-wait,
                          INVARIANTS HOLD
      cv.wait( mut );             // release and re-enter crit sec
                          INVARIANTS HOLD
    myTask = queue.first();       // take task
    if( myTask != done )          // remove it if not the sentinel,
      queue.pop();                //   which others need to see
  }                               // exit critical section
                          INVARIANTS HOLD
  if( myTask != done )
    DoWork( myTask );
}
```

Questions & Answers

- Why was *mut.unlock()* not enough to exit the critical section?
  - Unlock often is enough to exit a critical section, but we have extra semantics: "We knew" that consumers are waiting on the condition variable too.
  - If we don't *cv.notify()*, the consumers will never wake up.
- But why *cv.notify()* on only the Producer critical section exits?
  - Because the condition variable is only to notify of new additions to the queue.
  - We don't need to wake up other consumers when we've taken a task away. They're only waiting for tasks to arrive.
- Could we make unlock-and-notify a single operation by default?
  - What an interesting suggestion! Exercise for the reader…

Create and Publish Queue Items: 1 Producer, Many Consumers (Locks + LF)

- This variant uses an **atomic<Task*> head** that points to a lock-free slist, using lock-free coordination for step 1 (producer → consumers), then a lock among consumers **(sketch)**:
  - Thread 1 (producer):

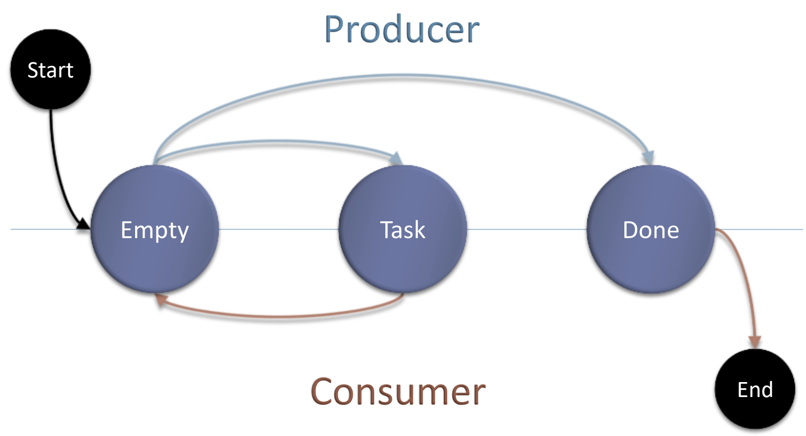
```
... build task list ...                                           Step 1 release
head = head of task queue;   // publish that complete list exists
```

  - Threads 2..N (consumers) spin until the list is there, then swarm:

```
while( myTask == null ) {
  lock_guard<mutex> lock{mut};                        Step 2 critical region
  if( head != null ) {          // check if list exists yet    Step 1 acquire
    myTask = head;              // take task
    head = head->next;          // remove it
  }
}
... = myTask->data;
```

- Note: In a real implementation you'd want to avoid busy-waiting.

A Mail Slot State Machine



Create and Publish Queue Items: 1 Producer, Many Consumers, Lock-Free

- 1 Producer thread: Changes any box from null to non-null.

```
curr = 0;                          // keep a finger on the current mailbox
// Phase 1: Build and distribute tasks
while( ThereAreMoreTasks() ) {
  task = AllocateAndBuildNewTask();
  while( slot[curr] != null )      // acquire null: look for next empty slot
    curr = (curr+1)%K;
  slot[curr] = task;               // release non-null: "You have mail!"
  sem[curr].signal();
}
// Phase 2: Stuff the mailboxes with "done" signals
numNotified = 0;
while( numNotified < K ) {
  while( slot[curr] != null )      // acquire null: look for next notifiable slot
    curr = (curr+1)%K;
  slot[curr] = done;               // release done: write sentinel
  sem[curr].signal();
  ++numNotified;
}
```

Create and Publish Queue Items: 1 Producer, Many Consumers, Lock-Free

- K Consumer threads (mySlot = 0..K-1): Each changes its own box from non-null to null.

```
myTask = null;
while( myTask != done ) {
  while( (myTask = slot[mySlot]) == null ) // acquire non-null,
    sem[mySlot].wait();                    // without busy-wait
  if( myTask != done ) {
    slot[mySlot] = null;     // release null: tell that we took it
    DoWork( myTask );        // good practice: prefer to do work
  }                          // outside the critical section
}
```

Create and Publish Queue Items: 1 Producer, Many Consumers, Lock-Free

- K Consumer threads (mySlot = 0..K-1): Each changes its own box from non-null to null.

```
myTask = null;
while( myTask != done ) {
  while( (myTask = slot[mySlot]) == null ) // acquire non-null,
    sem[mySlot].wait();                    // without busy-wait
  if( myTask != done ) {
    slot[mySlot] = null;
    DoWork( myTask );
  }                          // outside the critical section
}
```

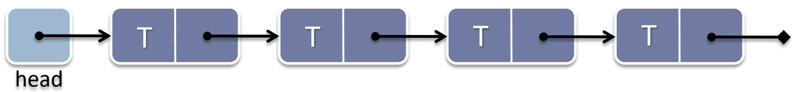
Q: Could it make sense to swap these two lines? Why?

Roadmap

- Two Basic Tools
  - Transactional thinking + atomic<T>
- Basic Example: Double-Checked Locking
  - It's pretty easy to do right, but you still have to do it right
- Producer-Consumer Variations
  - Using locks, locks + lock-free, and fully lock-free
- **A Singly Linked List: This Stuff Is Harder Than It Looks**
  - **Just find, push_front, and pop: How hard could it be?**

Example: Singly-Linked List

- A singly-linked list (aka “slist<T>”) is one of the simplest possible data structures:



- Simplifying assumptions:
  - Only four operations: Construct, destroy, find, push_front.
- **Challenge:** Write a lock-free implementation that callers can safely use without any external locks.
  - C’mon, how hard could it be?

A Lock-Free Singly-Linked List: First Cut

- Here is the interface declaration, and the internals we’ll use:

```
template<typename T>
class slist {
public:
   slist();
   ~slist();
   T* find( const T& t ) const;           // return pointer to first equal T
   void push_front( const T& t );         // insert at the front of the list
private:
   struct Node { T t; Node* next; };  // no “atomic” needed here
   atomic<Node*> head{ nullptr };     // but “atomic” is needed here:
                                      // “head” is mutable shared data
   slist(slist&) =delete;
   void operator=(slist&) =delete;
};
```

slist<T> Constructor

- The constructor is easy:

```
template<typename T>
slist<T>::slist()
  { }                              // or just "=default"
```

- Concurrency issues:
  - None.
  - Note: As usual, the caller has to know he can't use an object concurrently while he's constructing it. But this isn't an "external synchronization" issue as much as it's a lifetime management issue – he can't use the slist *before* it's constructed either.

slist<T> Destructor

- The destructor has to traverse:

```
template<typename T>
slist<T>::~slist() {
  auto first = head.load();    // good habit: access head once
  while( first ) {             //   (not needed here, but good habit...)
    auto unlinked = first;
    first = first->next;
    delete unlinked;
  }
}
```

- Concurrency issues:
  - None.
  - Note: As usual, the caller has to know he can't use an object concurrently while he's destroying it. But this isn't an "external synchronization" issue as much as it's a lifetime management issue – he can't use the slist *after* it's destroyed either.

slist<T>::find

- Return a pointer to the first equal element, or nullptr if there isn't one:

```
template<typename T>
T* slist<T>::find( const T& t ) const {
   auto p = head.load();
   while( p && p->t != t )
      p = p->next;
   return p ? &p->t : nullptr;
}
```

- Concurrency issues:
  - None.
  - As long as the constructor and destructor aren't running, this can freely run concurrently with other **find** operations… and should be safe to run concurrently with **insert** operations.

Group Exercise

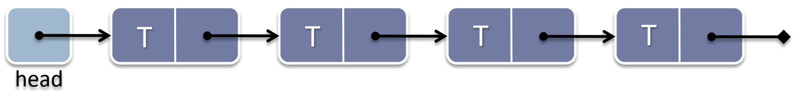
- Implement push_front to insert a node with a copy of the given value:

```
template<typename T>
void slist<T>::push_front( T t ) {




}
```

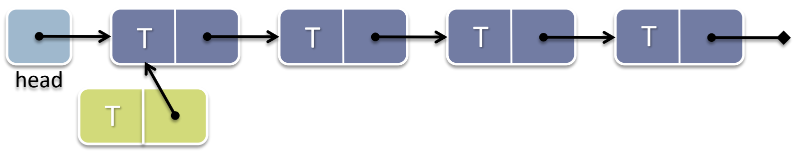
- You have 10 minutes.

A Look At push_front

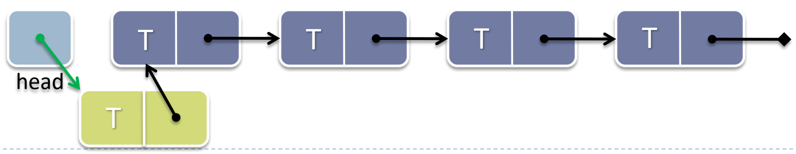
- Initial state:



- Intermediate state:



- Final state:



slist<T>::push_front (Flawed)

- Insert a node with a copy of the given value:

```
template<typename T>
void slist<T>::push_front( const T& t ) {
   auto p = new Node;               // create the new node
   p->t = t;                        // set its element value
   p->next = head;                  // set its place in the list
   head = p;                        // publish it at the head
}
```

  - **Q: What's wrong with this code?**

slist<T>::push_front (Flawed)

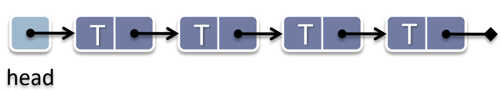
- Insert a node with a copy of the given value:

```
template<typename T>
void slist<T>::push_front( const T& t ) {
   auto p = new Node;            // create the new node
   p->t = t;                     // set its element value
   p->next = head;               // set its place in the list
   head = p;                     // publish it at the head
}
```

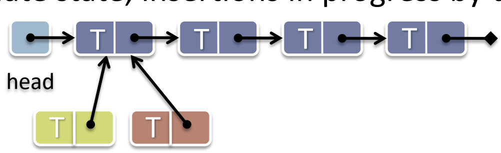
  - **Q: What's wrong with this code?**
- Concurrency issues:
  - None for any readers: The insertion of the new node is atomic. A concurrent reader will see either the old value or the new value, and in either case has a valid list to traverse.
  - **Problem for writers:** What if two threads try to insert at the same time?

A Look At the Problem

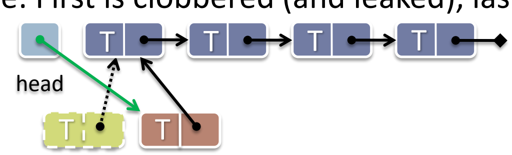
- Initial state:



- Intermediate state, insertions in progress by two threads:



- Final state: First is clobbered (and leaked), last one wins.



slist<T>::push_front

- Insert a node with a copy of the given value:

```
template<typename T>
void slist<T>::push_front( const T& t ) {
   auto p = new Node;                     // create the new node
   p->t = t;                              // set its element value
   p->next = head;                        // set its place in the list and
   while( !head.compare_exchange_weak(p->next, p) )
      { }                                 // try to swap it in until successful
}
```

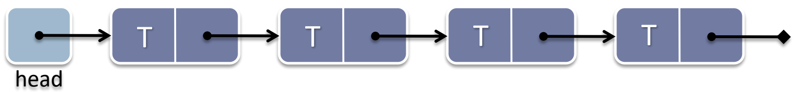
  - The "**CAS loop**" is a common construction in lock-free code.
    - Loop until "we get to be the one" to update head from 'expected' to 'desired'.
- Concurrency issues:
  - None for any readers: The insertion of the new node is atomic. A concurrent reader will see either the old value or the new value, and in either case has a valid list to traverse.
  - None for writers: The CAS loop makes concurrent writers safe (for now).

Well, that was easy…

So how about adding just one more little member function?

Revised Example: Pop Goes the List

- We'll stick with our singly-linked list, one of the simplest possible data structures:



- Simplifying assumptions:
  - Original operations: Construct, destroy, find, push_front.
  - **New operation: pop** to erase the first element from the list.
- **Same challenge:** Write a lock-free implementation that callers can safely use without any external locks.
  - C'mon, how hard could it be?

Group Exercise

- Implement pop_front to erase the first node:

```
template<typename T>
void slist<T>::pop_front() {



}
```

- You have 10 minutes.

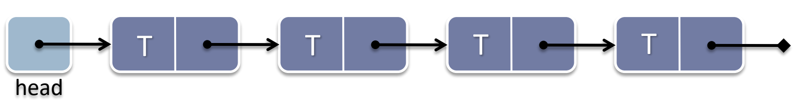
A Lock-Free Singly-Linked List: First Cut

- Here is the interface declaration, and the internals we'll use:

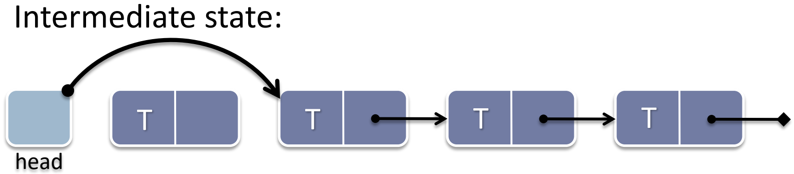
```
template<typename T>
class slist {
public:
   slist();
   ~slist();
   T* find( const T& t ) const;          // return pointer to first equal T
   void push_front( const T& t );        // insert at the front of the list
   void pop_front();                     // remove first element
private:
   struct Node { T t; Node* next; };  // no "atomic" needed here
   atomic<Node*> head{ nullptr };     // but it's needed here, because
                                      // "head" is mutable shared data
   slist(slist&) =delete;
   void operator=(slist&) =delete;
};
```

A Look At pop

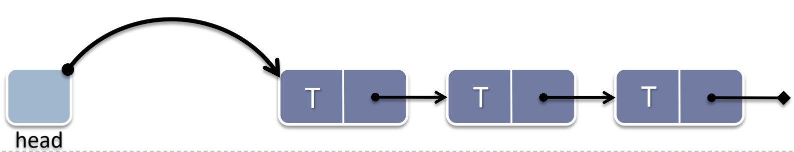
- Initial state:



- Intermediate state:



- Final state:



slist<T>::pop (Flawed)

- Remove the first node:

```
template<typename T>
void slist<T>::pop_front() {
   auto p = head.load();          // remember current first node
   if( p ) head = p->next;        // set head to the second node
   delete p;                      // and clean up old first node
}
```

  - **Q: What's wrong with this code?**

slist<T>::pop (Flawed)

- Remove the first node:

```
template<typename T>
void slist<T>::pop_front() {
   auto p = head.load();          // remember current first node
   if( p ) head = p->next;        // set head to the second node
   delete p;                      // and clean up old first node
}
```

  - **Q: What's wrong with this code?**

- Concurrency issues:
  - **Problem for readers:** What if a concurrent reader doing a find() is pointing to the first node and about to read its next pointer?
  - **Problem for writers:** What if a concurrent writer is trying to insert? What if a concurrent writer is trying to erase?

slist<T>::pop (Attempt #2, Still Flawed)

- Remove the first node:

```
template<typename T>
void slist<T>::pop_front() {
   auto p = head.load();                  // important: read head once
   while( p && !head.compare_exchange_weak(p, p->next) )
     { }                                  // NB: relies on short-circuit eval
   delete p;                              // and clean up
}
```

  - **Q: What's wrong with this code?**

slist<T>::pop (Attempt #2, Still Flawed)

- Remove the first node:

```
template<typename T>
void slist<T>::pop_front() {
   auto p = head.load();                  // important: read head once
   while( p && !head.compare_exchange_weak(p, p->next) )
     { }                                  // NB: relies on short-circuit eval
   delete p;                              // and clean up
}
```

  - **Q: What's wrong with this code?**

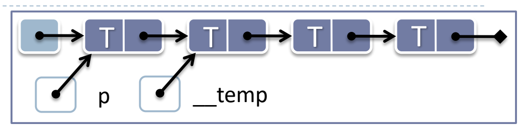
- Concurrency issues:
  - **Same problem for readers:** What if a concurrent reader doing a find() is pointing to the first node and about to read its next pointer?
  - **<u>Subtle</u> problem for writers:** What if a concurrent writer is trying to insert? What if a concurrent writer is trying to erase? Let's look at the "ABA problem"…

The "ABA Problem"

- Original state:

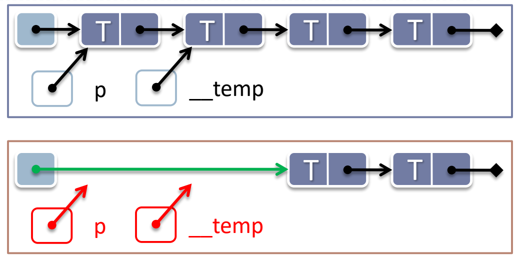
The "ABA Problem"

- Step 1 of deletion:
  p = head;
  __temp = p->next;



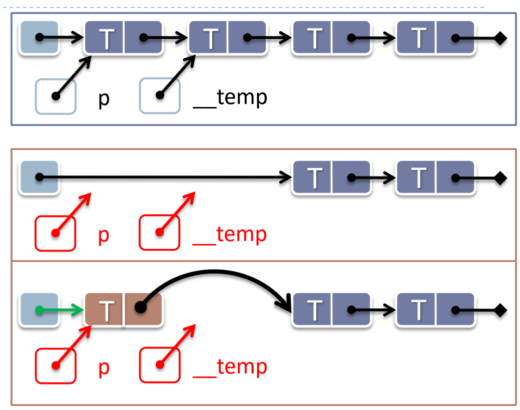
The "ABA Problem"

- Step 1 of deletion:
  p = head;
  __temp = p->next;
- Concurrently, another thread deletes the p two nodes.



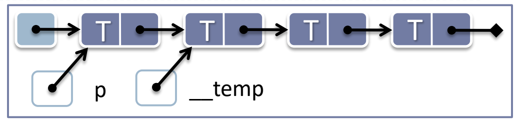
The "ABA Problem"

- Step 1 of deletion:
  p = head;
  __temp = p->next;
- Concurrently, another thread deletes the p two nodes.
- Then someone inserts a new node, and the allocator **reuses the same memory block**.

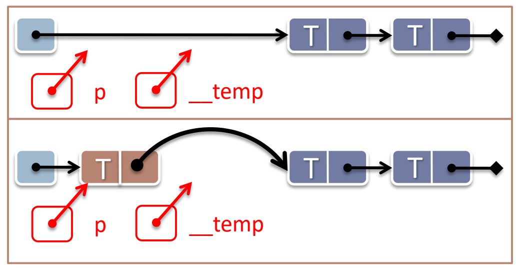


The “ABA Problem”

- Step 1 of deletion:
  p = head;
  __temp = p->next;



- Concurrently, another thread deletes the p two nodes.
- Then someone inserts a new node, and the allocator **reuses the same memory block**.



- Step 2 (CAS) “succeeds”:
  head.*c_e_w*(p, __temp);
  delete p;
  and we’re in trouble.

ABA Solutions (sketch)

- We need to solve the ABA issue: Two nodes with the same address, but different identities (existing at different times).
- Option 1: Use lazy garbage collection.
  - Solves the problem. Memory can’t be reused while pointers to it exist.
  - But: Not an option (yet) in portable C++ code, and destruction of nodes becomes nondeterministic.
- Option 2: Use reference counting (garbage collection).
  - Solves the problem in cases without cycles. Again, avoids memory reuse.
- Option 3: Make each pointer unique by appending a serial number, and increment the serial number each time it’s set.
  - This way we can always distinguish between A and A’.
  - But: Requires an atomic compare-and-swap on a value that’s larger than the size of a pointer. Not available on all hardware & bit-nesses.
- Option 4: Use hazard pointers.
  - Maged Michael and Andrei Alexandrescu have covered this in detail.
  - But: It’s very intricate. Tread with caution.

Delete-While-Traversing Solutions (sketch)

- We also need to resolve the deletion issue: We can't delete a node if a concurrent reader/writer might be pointing to it.
  - A concurrent member function, like find.
  - (!) A concurrent users of a T* we handed out.
- Option 1: Use lazy garbage collection.
  - Solves the problem because memory can't be reused while any pointers to it exist. However, destruction of nodes becomes nondeterministic.
- Option 2: Use reference counting (garbage collection).
  - Solves the problem in cases without cycles.
- Option 3: Never actually delete a node (only logically delete).
  - Can work when deleting is rare.
- Option 4: Put auxiliary nodes in between actual nodes.
  - Contains a next pointer only, no data. These links don't move. Enables operations on adjacent nodes to run without interference.

Psst…

Interested in a correct *slist* that fell off the back of my friend's truck, for cheap?

A Lock-Free Singly-Linked List: Second Cut

- Judicious tweaks to eliminate raw *: ref counting + *reference*.

```
template<typename T> class slist {
   struct Node { T t; shared_ptr<Node> next; };
   atomic_shared_ptr<Node> head;
   slist(slist&) =delete;
   void operator=(slist&) =delete;
public:
   slist() =default;
   ~slist() =default;
   class reference { /*…*/ };
   auto find( const T& t ) const
      auto p = head.load();
      while( p && p->t != t )
         p = p->next;
      return reference(move(p)
   }
```

Q: How would you implement class reference?

```
class reference {
   shared_ptr<Node> p;
public:
   reference(shared_ptr<Node> p_) : p{p_} { }
   T& operator*() { return p->t; }
   T* operator->() { return &p->t; }
};
```

A Lock-Free Singly-Linked List: Second Cut

- Continued:

```
   void push_front( const T& t ) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;
      while( !head.compare_exchange_weak(p->next, p) )
         { }
   }

   void pop_front() {
      auto p = head.load();
      while( p && !head.compare_exchange_weak(p, p->next) )
         { }
   }
};
```

Q: Where is the “delete”?

What Happens In the Case of Concurrent…

- Pop

```
void pop_front() {
   auto p = head.load();
   while( p &&
         !head.c_e_w(p, p->next) )
      { }
}
```

- Pop

```
void pop_front() {
   auto p = head.load();
   while( p &&
         !head.c_e_w(p, p->next) )
      { }
}
```

- The only competing modifying operations are *head.compare_exchange*.
  - One will happen-before the other, and succeed.
  - The other will fail, and retry.
  - ABA can't happen because no delete+recycling.

What Happens In the Case of Concurrent…

- Insert

```
void push_front( const T& t ) {
   auto p = make_shared<Node>();
   p->t = t;
   p->next = head;
   while( !head.c_e_w(p->next, p) )
      { }
}
```

- Pop

```
void pop_front() {
   auto p = head.load();
   while( p &&
         !head.c_e_w(p, p->next) )
      { }
}
```

- The only competing modifying operations are *head.compare_exchange*.
  - One will happen-before the other, and succeed.
  - The other will fail, and retry.

What Happens In the Case of Concurrent…

- Find

```
auto find( const T& t ) const {
   auto p = head.load();
   while( p && p->t != t )
      p = p->next;
   return reference(p);
}
```

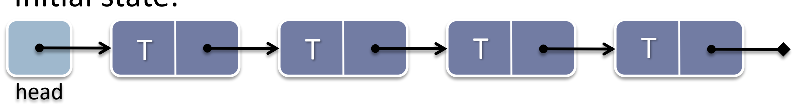
- Pop

```
void pop_front() {
   auto p = head.load();
   while( p &&
            !head.c_e_w(p, p->next) )
      { }
}
```

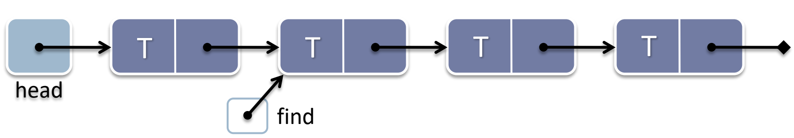
- Thanks to ref counting, *find* keeps its current node (and successors) alive.
  - *find* sees list “as if” *pop* waited for *find* to finish!
  - Important concept: **Linearizability**.

A Look At *find* + *pop*

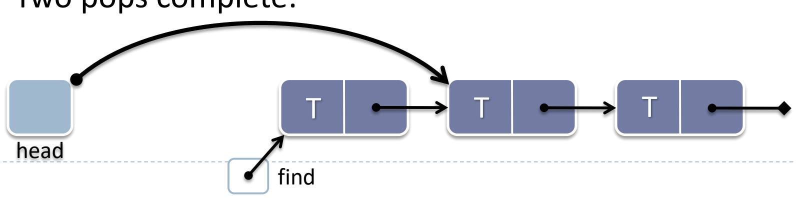
- Initial state:



- Find begins and gets partway through the list:



- Two pops complete:



Translation Guide

- The code we just saw.
  - From the Concurrency TS:

```
atomic_shared_ptr<T> a;

auto p = a.load();

a.compare_exchange_weak(e,d);
```

- What you actually write today.
  - The “atomic<>” is a comment, and remember to write an *atomic_** call for <u>every</u> use of the *shared_ptr*.

```
shared_ptr<T> a;
  // remember "atomic" – rely on discipline

auto p = atomic_load(&a);

atomic_compare_exchange_weak(&a,&e,d);
```

Q: How long are nodes kept alive?

A: Until the last *shared_ptr<Node>* (including in a returned *::reference*) to that node <u>or an earlier *Node*</u> goes away.

If calling code stores *::reference*s we can “leak” *Node*s in the Java sense of “leak” (delay their cleanup).

One option: Use custom refcounting to distinguish “navigating” strong refs that keep the following *Node*s alive, and ordinary refs to only the *T* and that can *->next.reset()*.

A more standard option follows… ++simplicity vs. --space/time.

HT: Tomasz Kamiński

A Lock-Free Singly-Linked List: Third Cut

- + Proverbial level of indirection (NB: space/alloc overhead):

```
template<typename T> class slist {
   struct Node { shared_ptr<T> t; shared_ptr<Node> next; };
   atomic_shared_ptr<Node> head;
   slist(slist&) =delete;
   void operator=(slist&) =delete;
public:
   slist() =default;
   ~slist() =default;
   // no more 'reference' class
   auto find( const T& t ) const {
      auto p = head.load();
      while( p && p->t != t )
         p = p->next;
      return p.t;
   }
```

A Lock-Free Singly-Linked List: Third Cut

- Continued:

```
   void push_front( const T& t ) {
      auto p = make_shared<Node>();
      p->t = make_shared<T>(t);
      p->next = head;
      while( !head.compare_exchange_weak(p->next, p) )
         { }
   }

   void pop_front() {
      auto p = head.load();
      while( p && !head.compare_exchange_weak(p, p->next) )
         { }
   }
};
```

Roadmap

- Two Basic Tools
  - Transactional thinking + atomic<T>
- Basic Example: Double-Checked Locking
  - It's pretty easy to do right, but you still have to do it right
- Producer-Consumer Variations
  - Using locks, locks + lock-free, and fully lock-free
- A Singly Linked List: This Stuff Is Harder Than It Looks
  - Just find, push_front, and pop: How hard could it be?

Questions?

Bonus Slides

Example 1

- Baseline code.

```
template <typename T>
struct LowLockQueue {
  struct Node {
    Node( T val ) : value(val), next(nullptr) { }
    T value;                       // objects are held by value
    atomic<Node*> next;
  };
```

  - *first* and *last* point to the before-the-first and last nodes
  - *divider* points to a boundary between producer and consumer

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

Example 1 (continued)

- Construct.

```
public:
  LowLockQueue() {
    first = divider = last = new Node( T() );
    producerLock = consumerLock = false;
  }
```

- Destroy.

```
~LowLockQueue() {
    while( first != nullptr ) {
      Node* tmp = first;
      first = tmp->next;
      delete tmp;
    }
  }
```

Example 1 (continued)

- *Consume* returns the value in the first unconsumed node.
  - Note: The entire body of *Consume* is inside the critical section, so we get no concurrency among consumers in this code.

```
bool Consume( T& result ) {
  while( consumerLock.exchange(true) )
    { }                                      // acquire exclusivity
  if( divider->next != nullptr ) {          // if queue is nonempty
    result = divider->next->value;          // copy it back to the caller
    divider = divider->next;                // publish that we took an item
    consumerLock = false;                   // release exclusivity
    return true;                            // and report success
  }
  consumerLock = false;                     // release exclusivity
  return false;                             // queue was empty
}
```

Example 1 (continued)

- *Produce* adds a new nodeto the tail, then lazily cleans up any consumed nodes at the front of the list.
  - Note: Not all of the body of *Produce* is inside the critical section, so there is some concurrency among producers in this code.

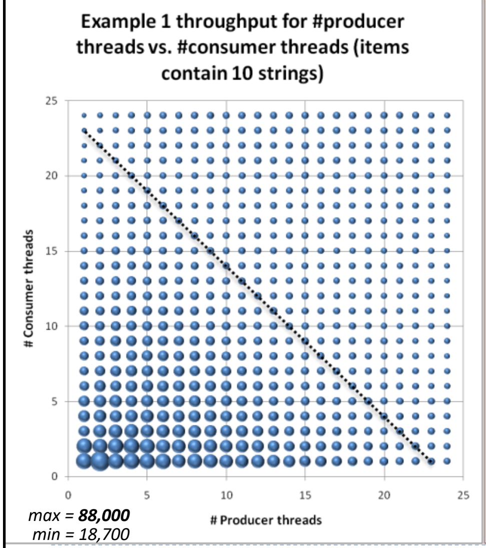
```
  bool Produce( const T& t ) {
    Node* tmp = new Node( t );          // do work off to the side
    while( producerLock.exchange(true) )
      { }                               // acquire exclusivity
    last = last->next = tmp;            // publish the new item
    while( first != divider ) {         // lazy cleanup
      Node* tmp = first;
      first = first->next;
      delete tmp;
    }
    producerLock = false;               // release exclusivity
    return true;
  }
};
```

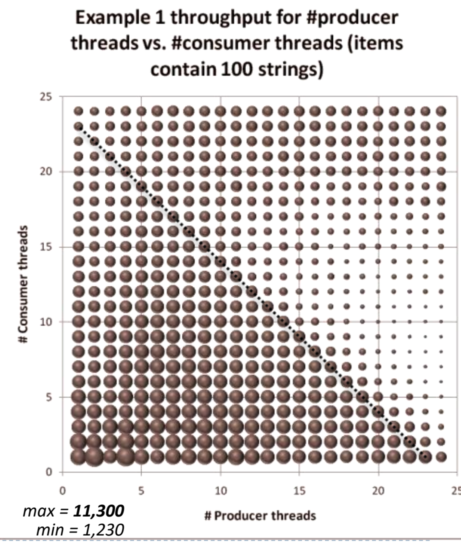
How Fast Is It?

- Key properties to look for:
  - **Throughput:** Total work, here #objects that can pass through the queue.
  - **Scalability:** Ability to use more hardware (cores) to get more work done.
- These are affected by the effects of:
  - **Contention:** How much threads interfere with each other by fighing for resources.
  - **Oversubscription:** What happens if there is more CPU-bound work ready to execute than available hardware to execute it.

Example 1 Measurements

 # cores on test machine





Ex. 2: Shrinking Consumer Critical Section

- For better performance, add heap-allocation…
  … *what, what?*
  - Lets us move the copying work out of the critical section.
- Diffs from Example 1 (part 1 of 2):

```
struct Node {
   Node( T* val ) : value(val), next(nullptr) { }
   T* value;
   atomic<Node*> next;
};
LowLockQueue() {
   first = divider = last = new Node( nullptr );
   producerLock = consumerLock = false;
}
```

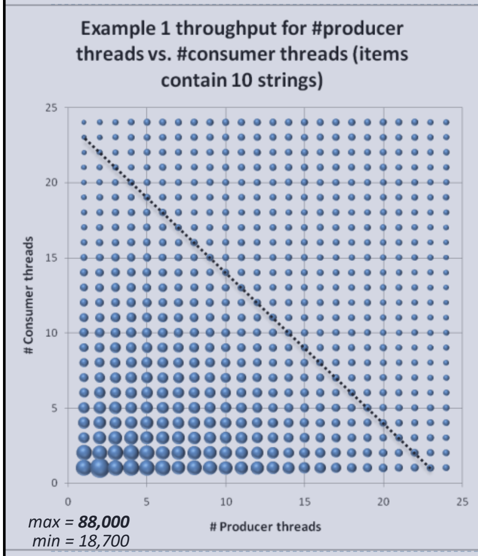
Ex. 2: Shrinking Cons Crit Sec (continued)

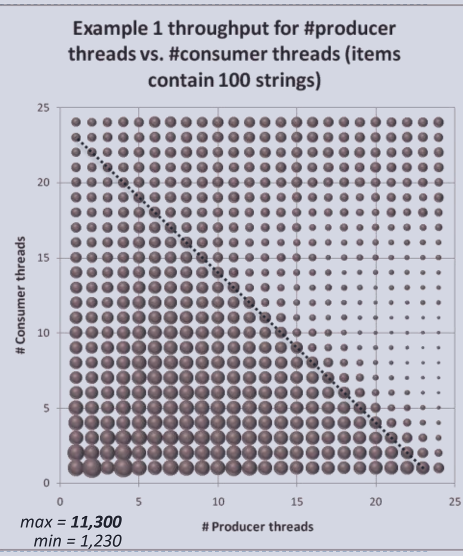
- Diffs from Example 1 (part 2 of 2):
  - Move the copying of the dequeued object, and the deletion of the value, outside the critical section.

```
bool Consume( T& result ) {
  while( consumerLock.exchange(true) )
    {}                                  // acquire exclusivity
  if( divider->next != nullptr ) {      // if queue is nonempty
    T* value = divider->next->value;    // take it out
    divider->next->value = nullptr;     // of the Node
    divider = divider->next;            // publish that we took an item
    consumerLock = false;               // release exclusivity
    result = *value;                    // now copy it back to the caller
    delete value;
    return true;                        // and report success
  }
  consumerLock = false;                 // release exclusivity
  return false;                         // queue was empty
}
```

Example 1 Measurements

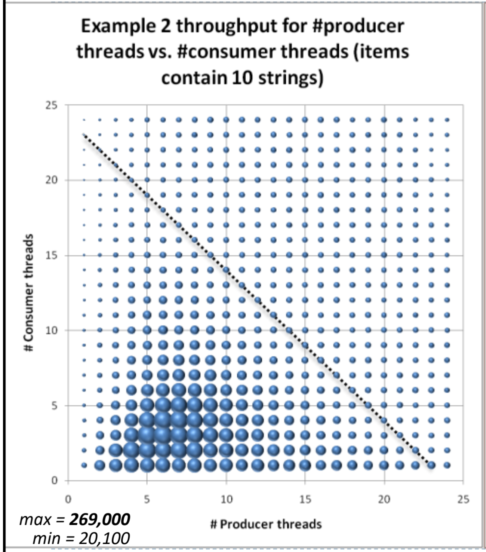
 # cores on test machine

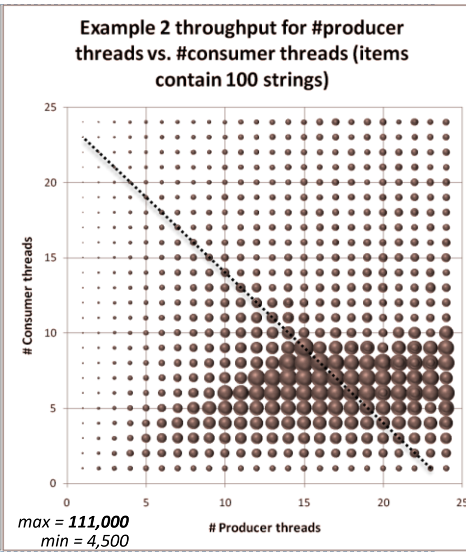




Example 2 Measurements

 # cores on test machine





Ex. 3: Reducing Head Contention

- Ex. 1 & 2: Producer lazily removed consumed nodes.
  - Forces producer to touch both ends of the queue.
  - All threads (producers and consumers) have to touch head.
  - Even though producers and consumers use different locks and can run concurrently w.r.t. each other, this results in invisible contention in the memory system.
- Idea: Let each consumer trim the nodes it consumed.
  - Which it was touching anyway ⇒ better **locality**.
  - Bonus: No more *divider*.
- Diffs from Example 3 (part 1 of 3):

```
LowLockQueue() {
  first = last = new Node( nullptr );     // no more divider
  producerLock = consumerLock = false;
}
```

Ex. 3: Reducing Head Contention (cont'd)

- Diffs from Example 3 (part 2 of 3):

```
bool Consume( T& result ) {
  while( consumerLock.exchange(true) )
    { }                          // acquire exclusivity
  if( first->next != nullptr ) { // if queue is nonempty
    Node* oldFirst = first;
    first = first->next;
    T* value = first->value;     // take it out
    first->value = nullptr;      // of the Node
    consumerLock = false;        // release exclusivity
    result = *value;             // now copy it back
    delete value;                // and clean up
    delete oldFirst;             // both allocations
    return true;                 // and report success
  }
  consumerLock = false;      // release exclusivity
  return false;          // queue was empty
}
```

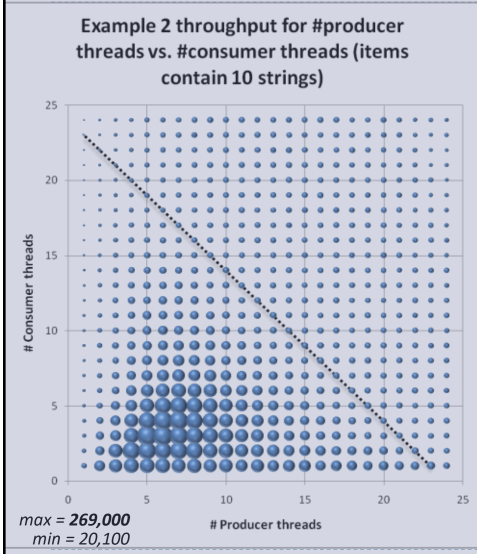
Ex. 3: Reducing Head Contention (cont'd)

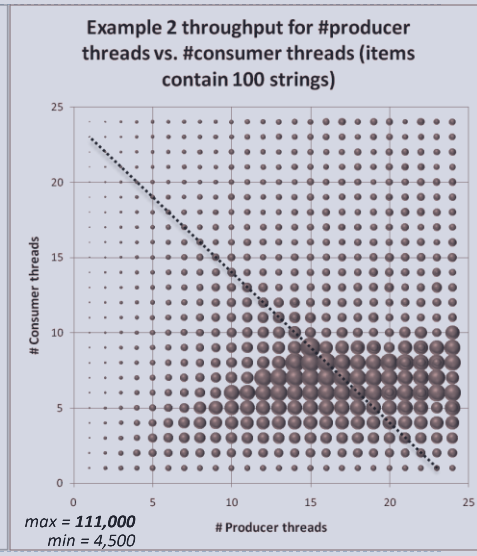
- Diffs from Example 3 (part 3 of 3):
  - *Producer* is simpler.

```
bool Produce( const T& t ) {
  Node* tmp = new Node( t );    // do work off to the side
  while( producerLock.exchange(true) )
    { }                         // acquire exclusivity
  last->next = tmp;             // A: publish the new item
  last = tmp;                   // B: not "last->next"
  producerLock = false;         // release exclusivity
  return true;
}
```

Example 2 Measurements

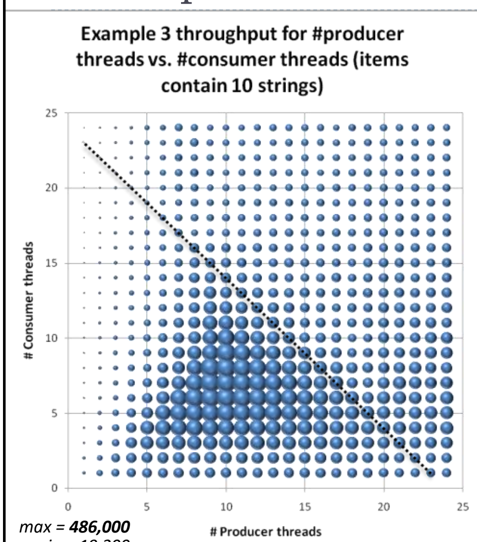
 # cores on test machine

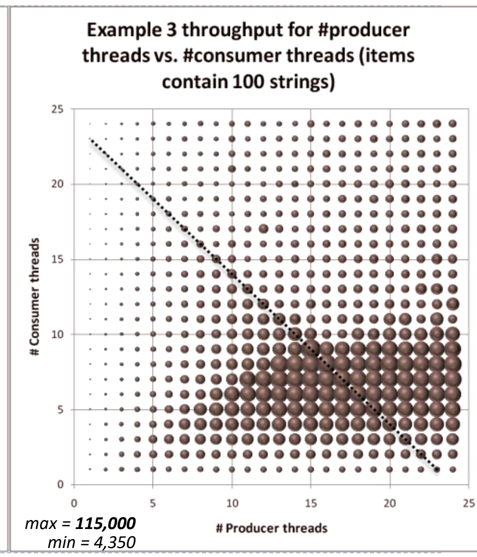




Example 3 Measurements

 # cores on test machine





Ex. 4: Do Nothing… or, “Add Nothing”

- Keep data that is *not* used together *apart*.
  - If variables A and B are liable to be used on different threads, keep them on separate cache line.
- Diffs from Example 4:

```
struct                          Node {
   Node( T* val ) : value(val), next(nullptr) { }
   T* value;
   atomic<Node*> next;
};
                          Node* first;
                          atomic<bool> consumerLock;
                          Node* last;
                          atomic<bool> producerLock;
```

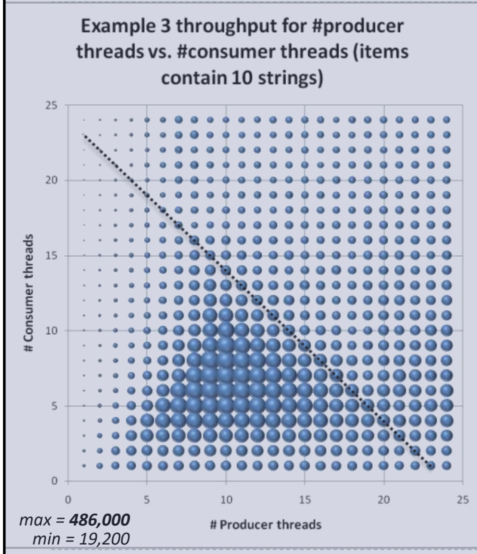
Ex. 4: Do Nothing… or, “Add Nothing”

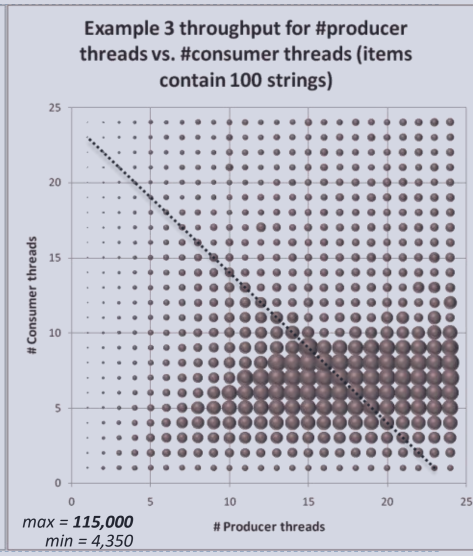
- Keep data that is *not* used together *apart*.
  - If variables A and B are liable to be used on different threads, keep them on separate cache line.
- Diffs from Example 4:

```
struct alignas(CACHE_LINE_SIZE) Node {
   Node( T* val ) : value(val), next(nullptr) { }
   T* value;
   atomic<Node*> next;
};
alignas(CACHE_LINE_SIZE) Node* first;
alignas(CACHE_LINE_SIZE) atomic<bool> consumerLock;
alignas(CACHE_LINE_SIZE) Node* last;
alignas(CACHE_LINE_SIZE) atomic<bool> producerLock;
```

Example 3 Measurements

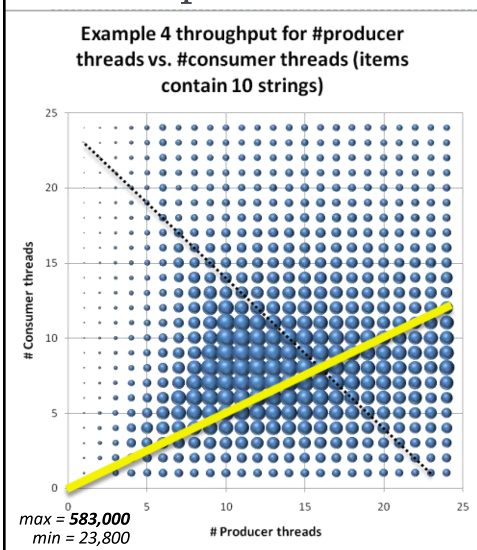
 # cores on test machine

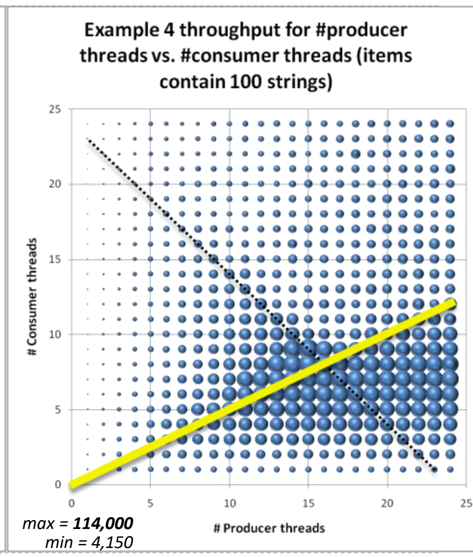




Example 4 Measurements

 # cores on test machine

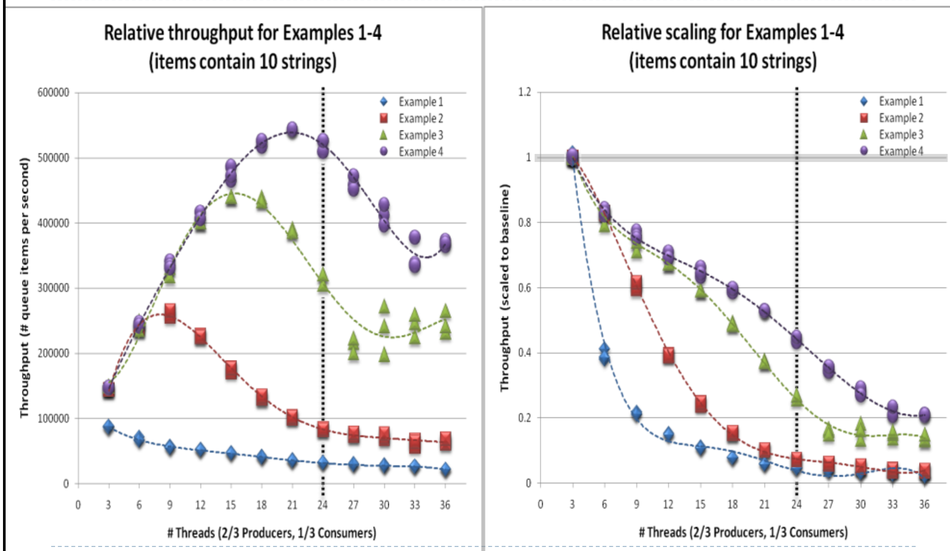




Direct Comparisons of Examples 1-4 ("Small")

linear scaling

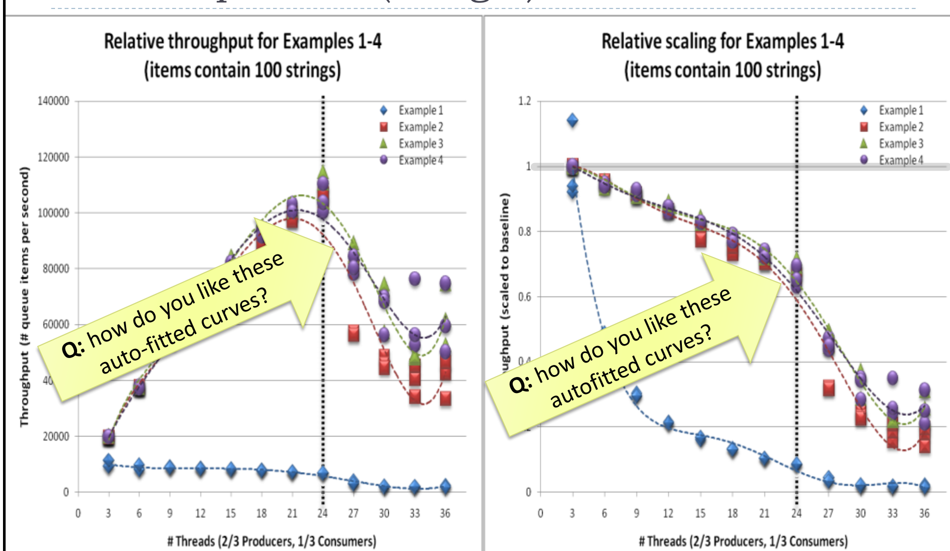
 # cores on test machine



Direct Comparisons of Examples 1-4 ("Large")

linear scaling

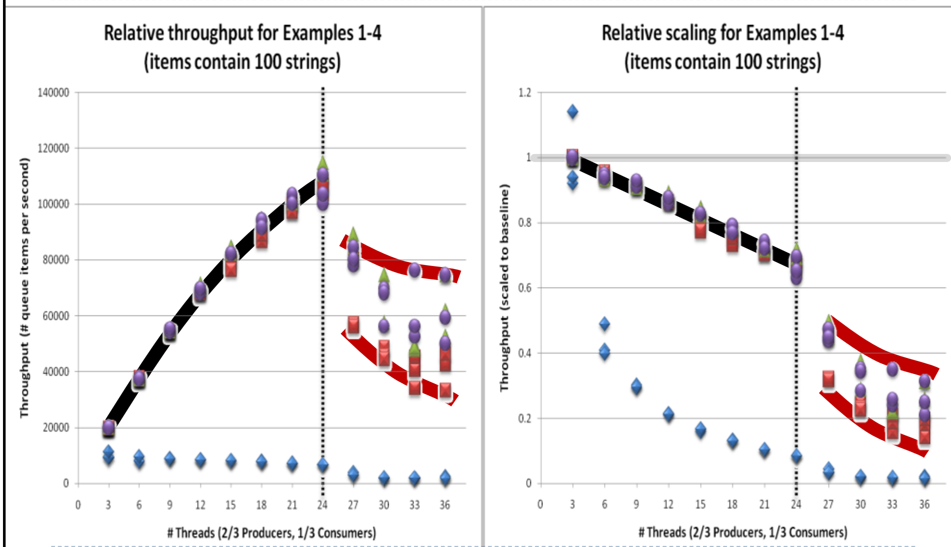
 # cores on test machine



When Fitted Curves Go Bad

linear scaling

 # cores on test machine



What Have We Learned?

- To improve **scalability**, we need to minimize **contention**:
  - Reduce the size of critical sections $\Rightarrow$ more concurrency.
  - Reduce sharing by isolating threads to use different parts of the data structure.
    - Moving cleanup from producer to consumer lets consumers touch only the head, producers touch only the tail.
  - Reduce false sharing of different data on the same cache line, but adding alignment padding.
    - Separate variables that should be able to be used concurrently by different threads should be far enough apart in memory.

What Have We Learned? (2)

- To understand scalability, need to know what to measure:
  - **Identify the key different kinds of work:** Here, producer threads and consumer threads. **Use stress tests to measure** the impact of having different quantities and combinations of these in our workload.
  - **Identify the different kinds of data:** Here, representative "small" and "large" queue items). **Vary those** to measure their impact.
  - **Measure total throughput**, or items handled per unit time.
  - **Look for scalability**, or the change in throughput as we add more threads. Does using more threads do more total work? Why or why not? In what directions, and for what combinations of workloads?
  - **Look for contention**, or the interference between multiple threads trying to do work concurrently.
  - **Watch for the cost of oversubscription**, and eliminate it either algorithmically or by limiting the actual amount of concurrency to avoid it altogether.
  - **Beware of overreliance on automated trendlines / fitted curves.** Apply them only after first examining the raw data.

Questions?