Concurrent Queues and Stacks

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

The Five-Fold Path

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization

We saw 4 approaches to concurrent data structure design.

Another Fundamental Problem

- We told you about
  - Sets implemented using linked lists
- Next: queues
- Next: stacks

We have already talked about concurrent lists, and today we look at another ubiquitous data structure: queues. Queues provide enqueue and dequeue methods and they are used to route requests from one place to another. For example, a web site might queue up requests for pages, for disk writes, for credit card validations and so on.

Queues & Stacks

- Both: pool of items
- Queue
  - enq() & deq()
  - First-in-first-out (FIFO) order
- Stack
  - push() & pop()
  - Last-in-first-out (LIFO) order

Bounded vs Unbounded

- Bounded
  - Fixed capacity
  - Good when resources an issue
- Unbounded
  - Holds any number of objects

When designing a pool interface, one choice is whether the make the pool bounded or unbounded. A bounded pool has a fixed capacity (maximum number of objects it holds). Bounded pools are useful when resources are an issue. For example, if you are concerned that the producers will get too far ahead of the consumers, you should bound the pool. An unbounded pool, by contrast, holds an unbounded number of objects.

Blocking vs Non-Blocking

- Problem cases:
  - Removing from empty pool
  - Adding to full (bounded) pool
- Blocking
  - Caller waits until state changes
- Non-Blocking
  - Method throws exception

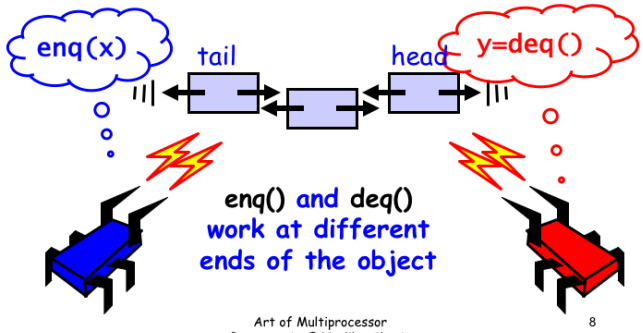
A second choice is what to do when a thread tries to remove an item from an empty queue or to add one to a full bounded queue. One choice is to block the thread until the state changes. This choice makes sense when there is nothing else for the thread to do. The alternative is to throw an exception. This choice makes sense when the thread can profitably turn its attention to other activities.

This Lecture

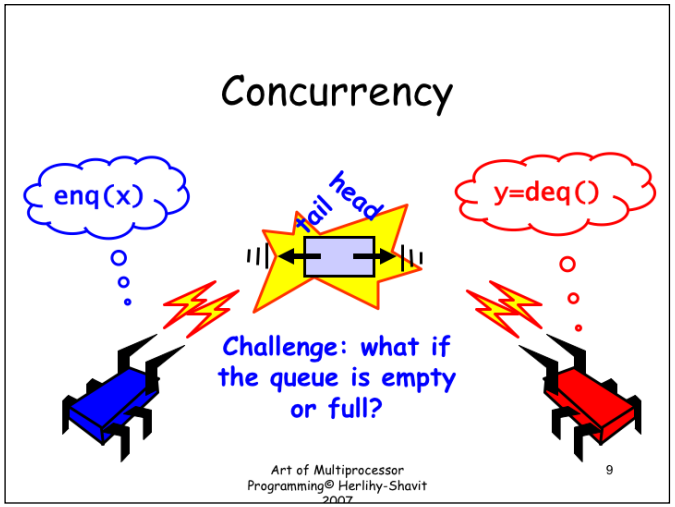
- Bounded, Blocking, Lock-based Queue
- Unbounded, Non-Blocking, Lock-free Queue
- ABA problem
- Unbounded Non-Blocking Lock-free Stack
- Elimination-Backoff Stack

In this lecture we are going to look at two alternative highly-concurrent queue implementations. The first is a bounded, blocking, lock-based queue. This implementation uses a combination of techniques due to Doug Lea and to Maged Michael and Michael Scott. The second implementation is an unbounded, non-blocking, lock-free implementation due to Michael and Scott. We will then continue to examine the design of concurrent stacks, show Treiber's lock-free stack and then how one can add parallelism to a stack through a technique called elimination.

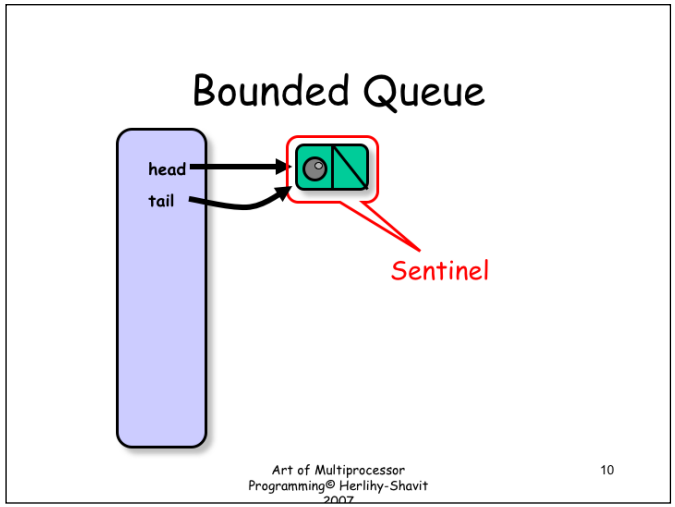
Queue: Concurrency



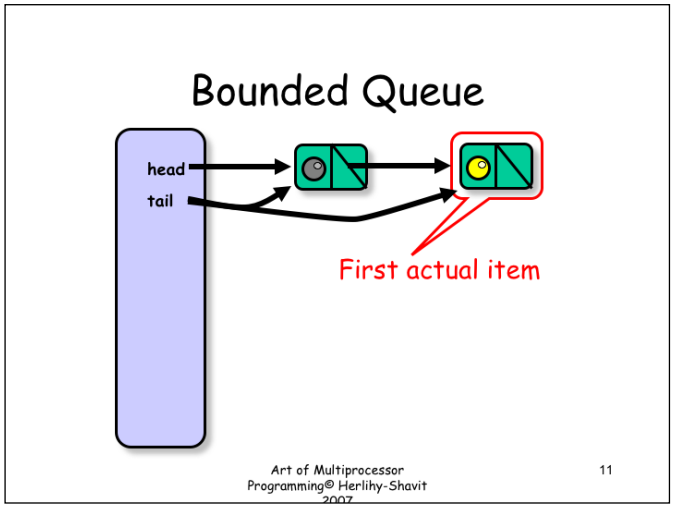
Making a queue concurrent is quite a challenge. Very informally, it seems that it should be OK for one thread to enqueue an item at one end of the queue while another thread dequeues an item from the other end, since they are working on disjoint parts of the data structure. Does that kind of concurrency seem easy to realize? (Answer: next slide)



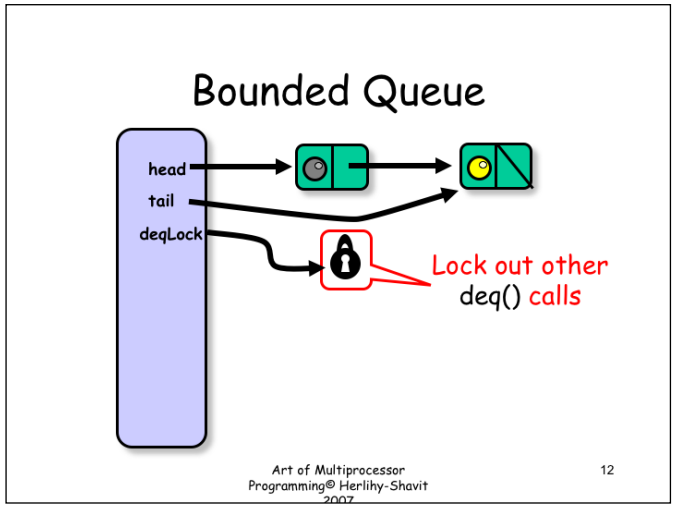
Concurrent enqueue and dequeue calls should in principle be able to proceed without interference, but it gets tricky when the queue is nearly full or empty, because then one method call should affect the other. The problem is that whether the two method calls interfere (that is, need to synchronize) depends on the dynamic state of the queue. In pretty much all of the other kinds of synchronization we considered, whether or not method calls needed to synchronize was determined statically (say read/write locks) or was unlikely to change very often (resizable hash tables).



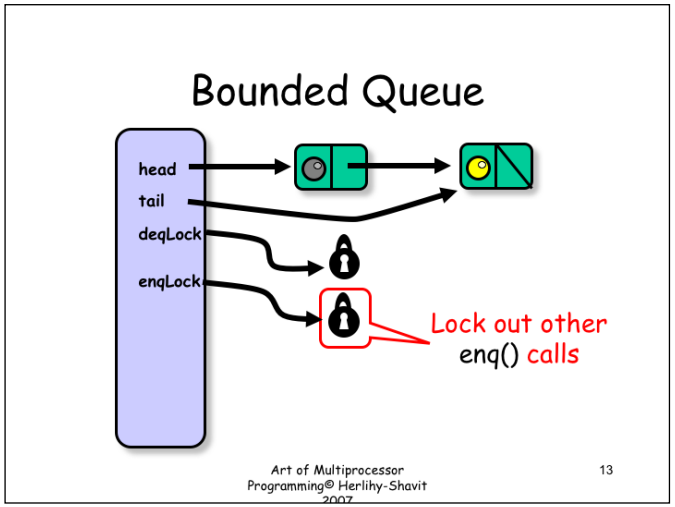
Let's use a list-based structure, although arrays would also work. We start out with head and tail fields that point to the first and last entries in the list. The first Node in the list is a sentinel Node whose value field is meaningless. The sentinel acts as a placeholder.



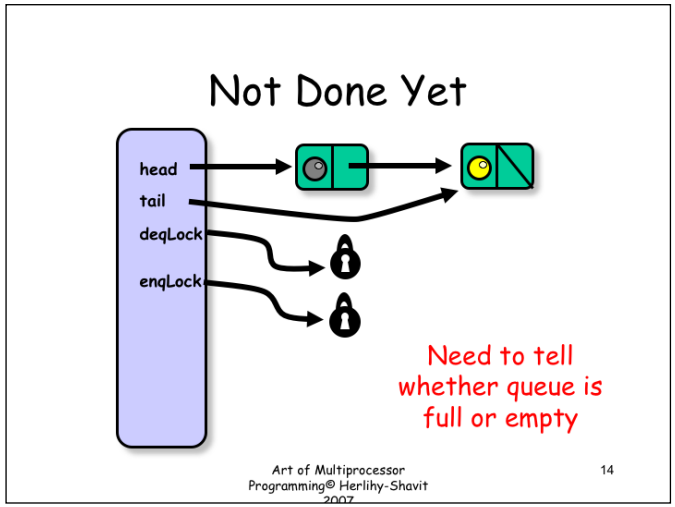
When we add actual items to the queue, we will hang them of the sentinel Node at the head.



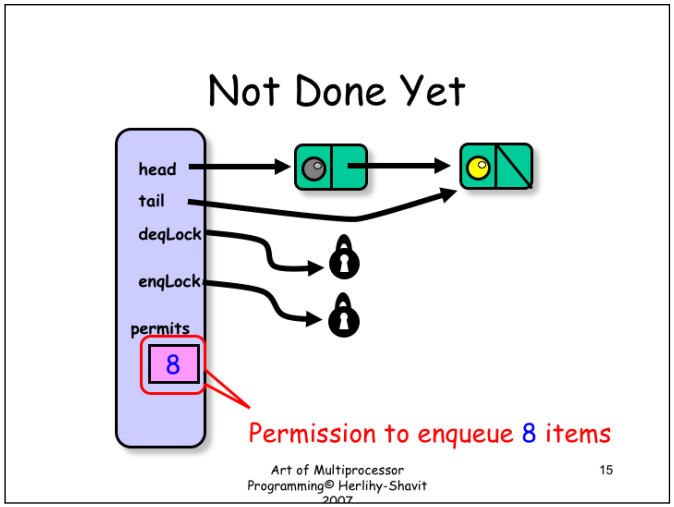
The most straightforward way to allow concurrent enq and deq calls is to use one lock at each end of the queue. To make sure that only one dequeuer is accessing the queue at a time, we introduce a dequeue lock field. Could we get the same effect just by locking the tail Node? (Answer: there's a race condition between when a thread reads the tail field and when it acquires the lock. Another thread might have enqueued something in that interval. Of course you could check after you acquire the lock that the tail pointer hadn't changed, but that seems needlessly complicated.)



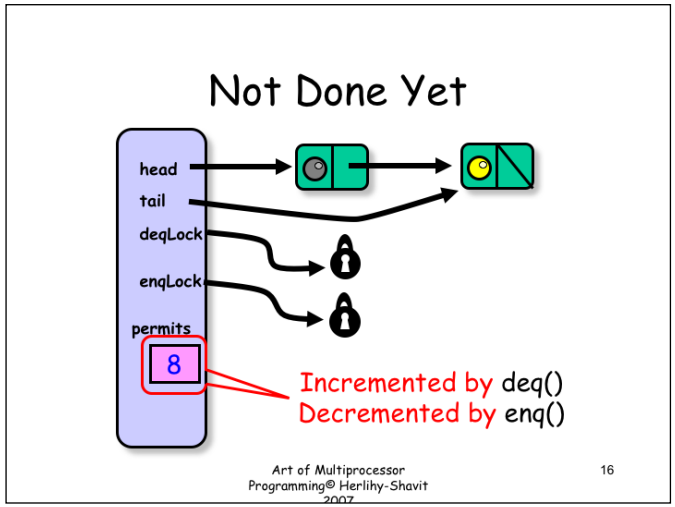
In the same way, we introduce an explicit enqLock field to ensure that only one enqueuer can be accessing the queue at a time. Would it work if we replaced the enqLock by a lock on the first sentinel Node? (Answer: Yes, it would because the value of the head field never changes. Nevertheless we use an enqLock field to be symmetric with the deqLock.)



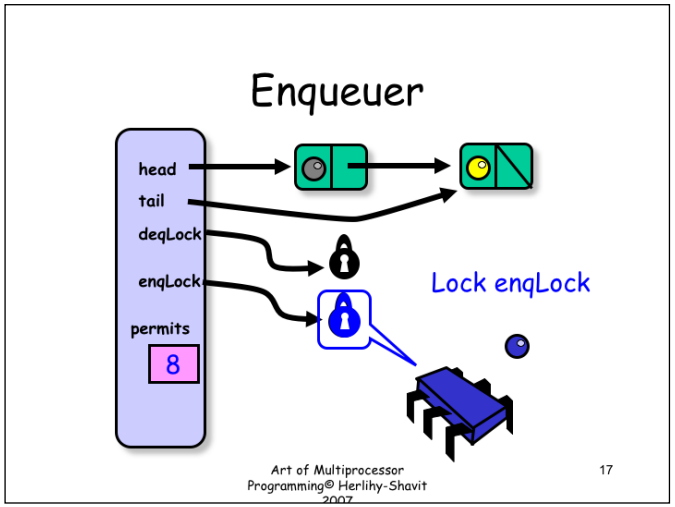
In the same way, we introduce an explicit enqLock field to ensure that only one enqueuer can be accessing the queue at a time. Would it work if we replaced the enqLock by a lock on the first sentinel Node? (Answer: Yes, it would because the value of the head field never changes. Nevertheless we use an enqLock field to be symmetric with the deqLock.)



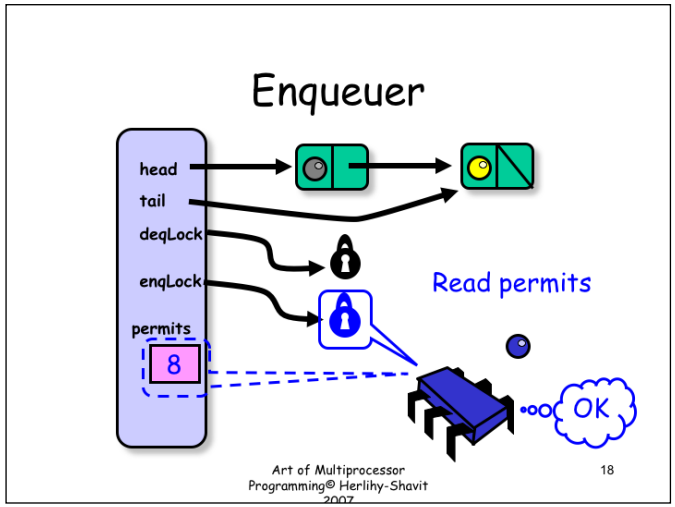
Let's add another field, which we will think of as keeping track of permissions to enqueue items. We could also think of this field as counting the number of empty slots, but calling it a permission will seem more natural later on when we look at some additional fine-grained tricks.



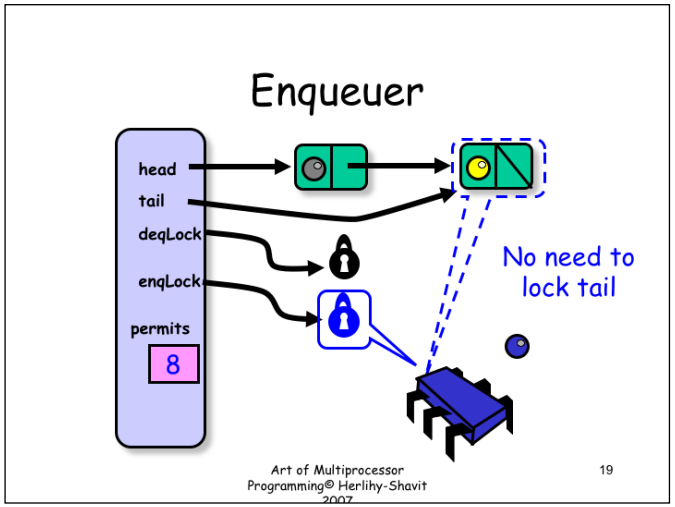
This field is incremented by the deq method (which creates a space) and decremented by the enq method.



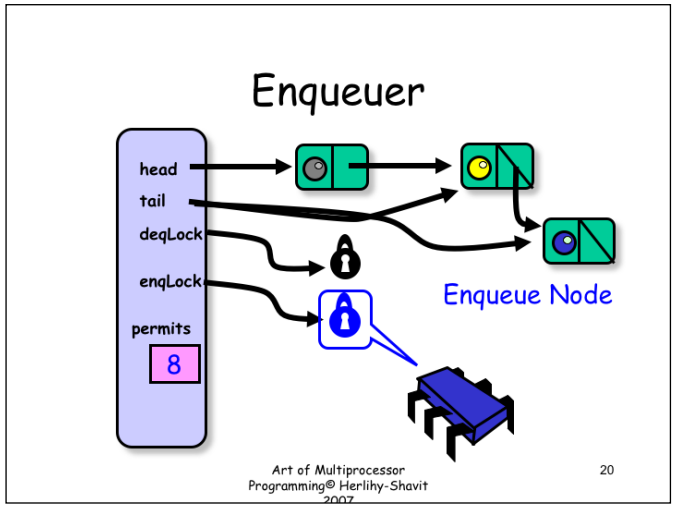
Let's walk through a very simple implementation. A thread that wants to enqueue an item first acquires the enqLock. At this point, we know there are no other enq calls in progress.



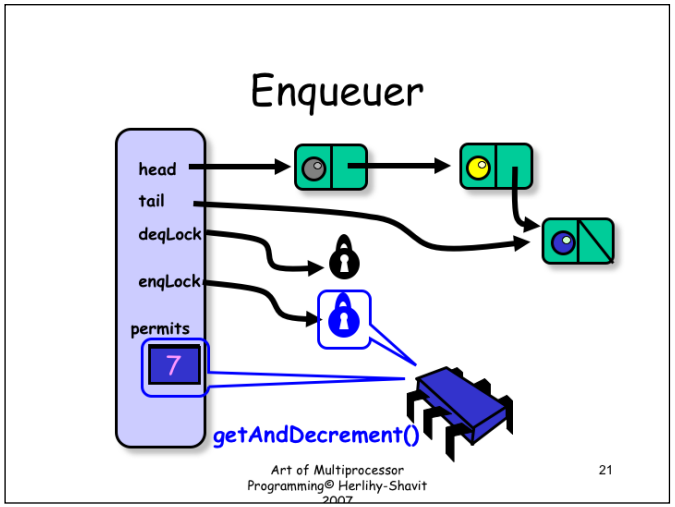
Next, the thread reads the number of permits. What do we know about how this field behaves when the thread is holding the enqLock? (Answer: it can only increase. Dequeuers can increment the counter, but all the enqueuers are locked out.) If the enqueuing thread sees a non-zero value for permits, it can proceed.



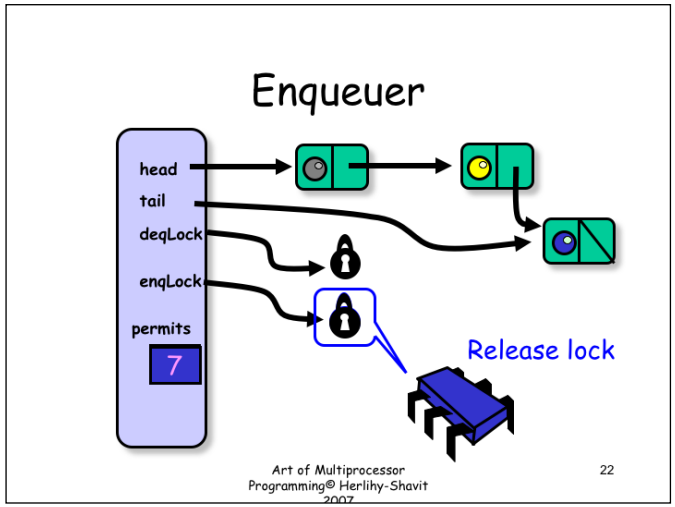
The thread does not need to lock tail Node before appending a new Node. Clearly, there is no conflict with another enqueuer and we will see a clever way to avoid conflict with a dequeuer.



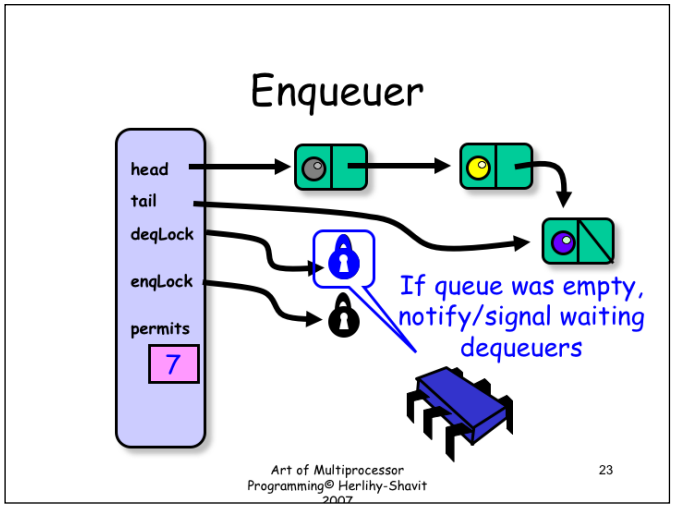
Still holding the enqLock, the thread redirects both the tail Node's next field and the queue's tail field to point to the new Node.



Still holding the enqLock, the thread calls getAndDecrement() to reduce the number of permits. Remember that we know the number is positive, although we don't know exactly what it is now. Why can't we release the enqLock before the increment? (Answer: because then other enqueuers won't know when they have run out of permits.)

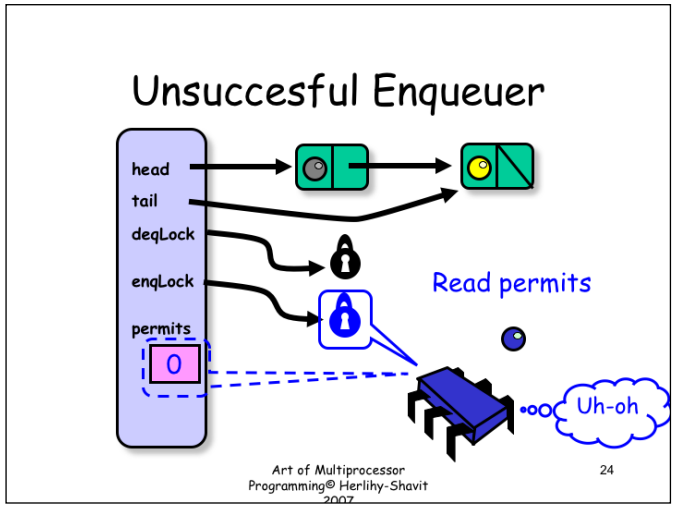


Finally, we can release the enqLock and return.



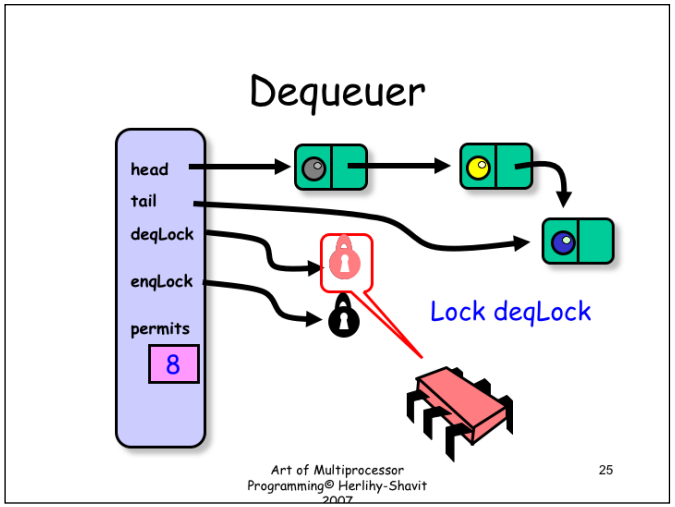
There is one more issue to consider. If the queue was empty, then one or more dequeuers may be waiting for an item to appear. If the dequeuer is spinning then we don't need to do anything, but if the dequeuers are suspended in the usual Java way, then we need to notify them. There are many ways to do this, but for simplicity we will not try to be clever, and simply require any enqueuer who puts an item in an empty queue to acquire the deqLock and notify any waiting threads.

TALKING POINT; Notice that Java requires you to acquire the lock for an object before you can notify threads waiting to reacquire that lock. Not clear if this design is a good idea. Does it invite deadlock? Discuss.

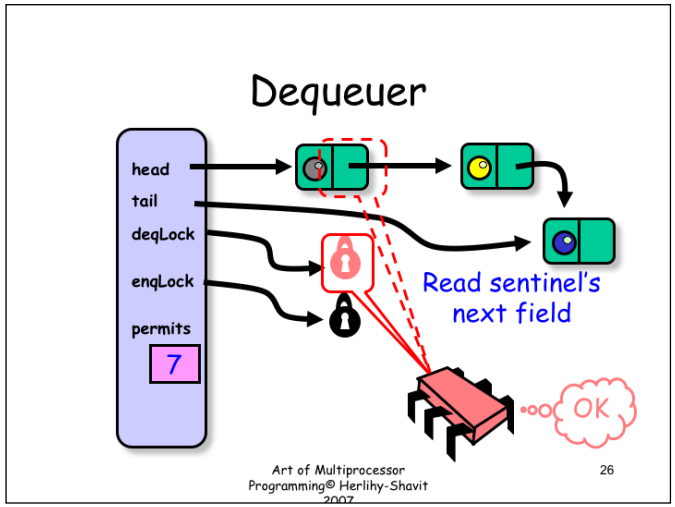


Let's rewind and consider what happens if the enqueing thread found the queue full. Suppose the threads synchronize by blocking, say using the Java wait() method. In this case, the thread can wait() on the enqLock, provided the first dequeuer to to remove an item notifies the waiting thread.

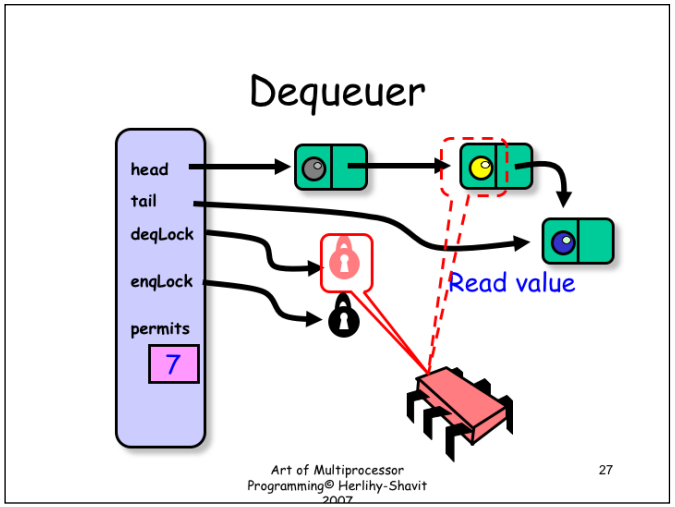
An alternative is for the enqueuer to spin waiting for the number of permits to be non-zero. In general, it is a dangerous practice to spin while holding a lock. It so happens that it works in this case, provided the dequeuer also spins when the queue is empty. Spinning here is cache-friendly, since the thread is spinning on a locally-cached value, and stops spinning as soon as the field value changes.



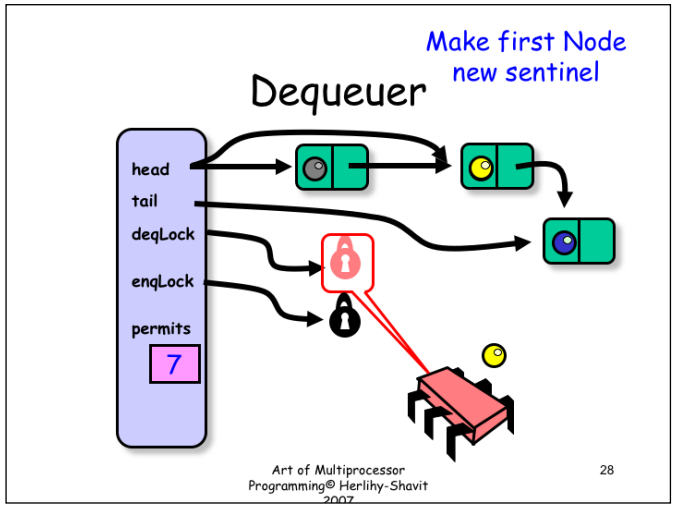
Now let's walk through a similar deq() implementation. The code is similar, but not at all identical. As a first step, the dequeuer thread acquires the deqLock.



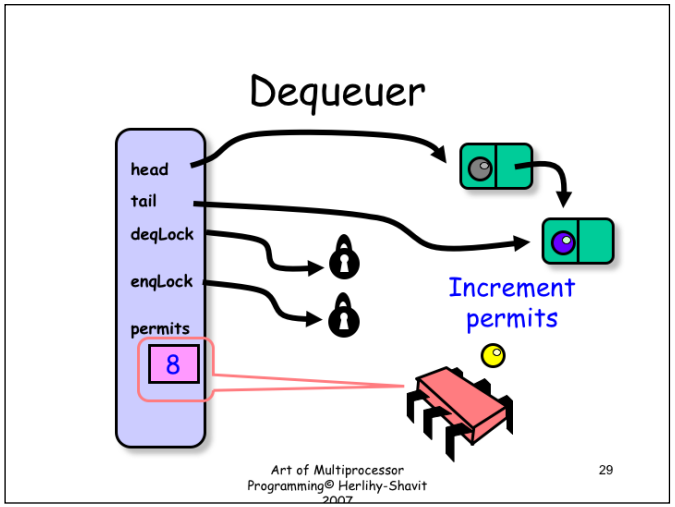
Next, the thread reads the sentinel node's next field. If that field is non-null, the queue is non-empty. Once the enqueuer sees a non-empty next field, that field will remain that way. Why? (ANSWER: the deq() code, which we have seen, changes null references to non-null, but never changes a non-null reference). The enqueuers are all locked out, and no dequeuer will modify a non-null field. There is no need to lock the Node itself, because it is implicitly protected by the enqLock.



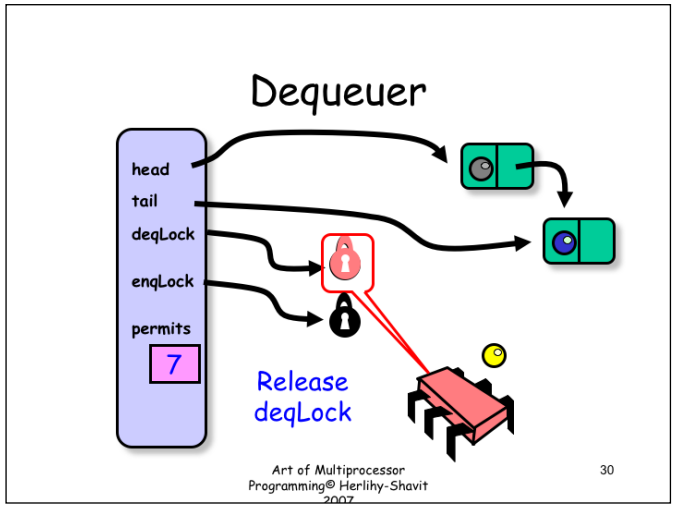
Next, the thread reads the first Node's value, and stores it in a local variable.



The thread then makes the first Node the new sentinel, and discards the old sentinel. This step may seem obvious, but, in fact, it is enormously clever. If we had tried to physically remove the Node, we would soon become bogged down in quagmire of complications. Instead, we discard the prior sentinel, and transform the prior first real Node into a sentinel. Brilliant.

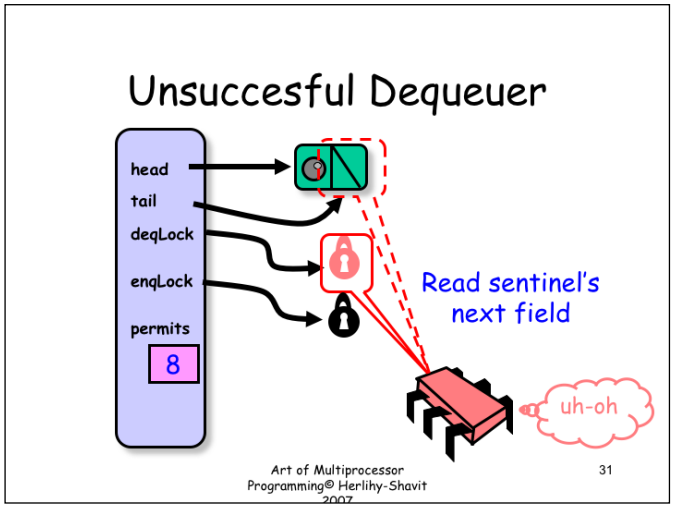


Finally, we increment the number of permits. By contrast with enqueuers, we do not need to hold the lock while we decrement the permits. Why? (Answer: we had to hold the lock while enqueuing to prevent lots of enqueuers from proceeding without noticing that the capacity had been exceeded. Dequeuers will notice the queue is empty when they observe that the sentinel's next field is null.)

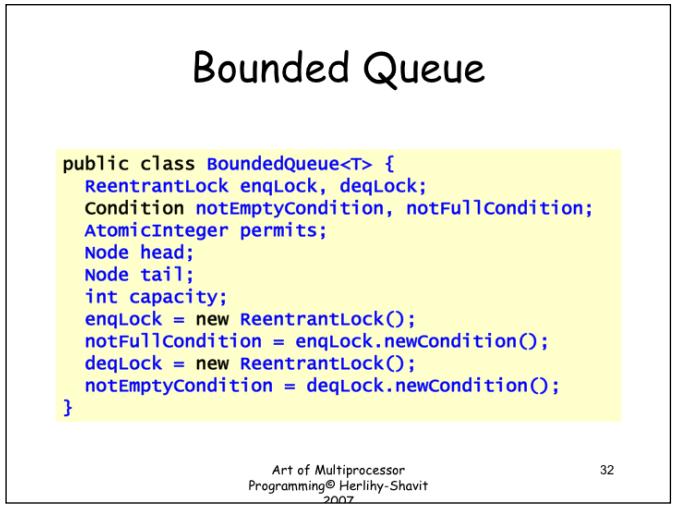


Next we can release the deqLo

ck.



If the dequeuer observes that the sentinel's next field is null, then it must wait for something to be enqueued. As with the dequeuer, the simplest approach is just to wait() on the deqLock, with the understanding that any enqueuer who makes the queue non-empty will notify any waiting threads. Spinning works too, even though the thread is holding a lock. Clearly, mixing the two strategies will not work: if a dequeuer spins while holding the deq lock and an enqueuer tries to acquire the deq lock to notify waiting threads, then a deadlock ensues.



Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq & deq locks

Digression: Monitor Locks

- Java Synchronized objects and Java ReentrantLocks are monitors
- Allow blocking on a condition rather than spinning
- Threads:
  - acquire and release lock
  - wait on a condition

In Java, every object provides a **wait()** method that unlocks the object and suspends the caller for a while. While the caller is waiting, another thread can lock and change the object. Later, when the suspended thread resumes, it locks the object again before it returns from the **wait()** call.

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Acquire lock

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Release lock

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Try for lock, but not too hard

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Create condition to wait on

The Java Lock Interface

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  void unlock;
}
```

Guess what this method does?

Java allows threads to interrupt one another. Interrupts are not pre-emptive: a thread typically must check whether it has been interrupted. Some blocking methods, like the built-in wait() method throw InterruptedException if the thread is interrupted while waiting. The lock method of the lock interface does not detect interrupts, presumably because it is more efficient not to do so. This variant does.

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Release lock and wait on condition

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Wake up one waiting thread

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Wake up all waiting threads

Await

```
q.await()
```

- Releases lock associated with q
- Sleeps (gives up processor)
- Awakens (resumes running)
- Reacquires lock & returns

In Java, every object provides a **wait()** method that unlocks the object and suspends the caller for a while. While the caller is waiting, another thread can lock and change the object. Later, when the suspended thread resumes, it locks the object again before it returns from the **wait()** call.

Signal

```
q.signal();
```

- Awakens one waiting thread
  - Which will reacquire lock

The **notify()** method (similarly notify() for synchronized methods) wakes up one waiting thread, chosen arbitrarily from the set of waiting threads. When that thread awakens, it competes for the lock like any other thread. When that thread reacquires the lock, it returns from its **wait()** call. You cannot control which waiting thread is chosen.

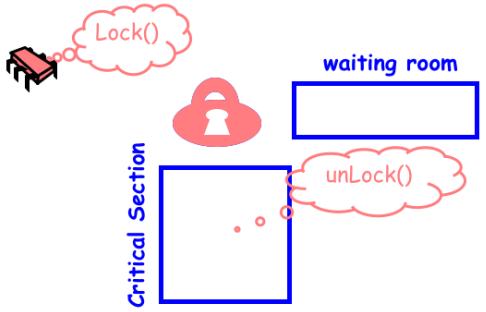
Signal All

```
q.signalAll();
```

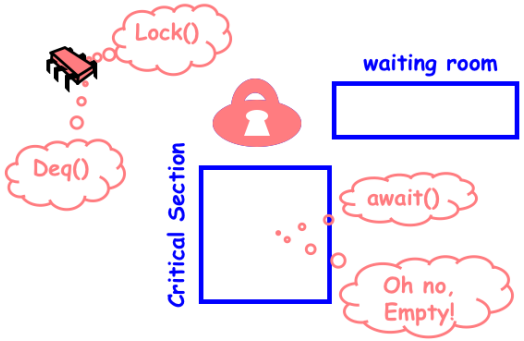
- Awakens all waiting threads
  - Which will each reacquire lock

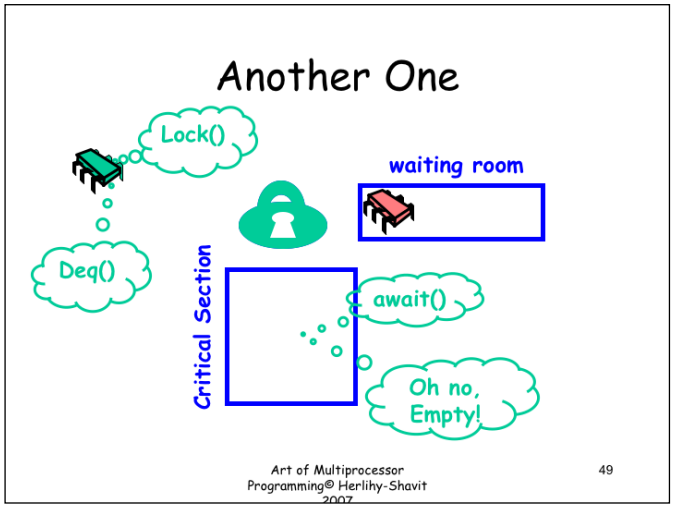
The **signalAll()** method wakes up all waiting threads. Each time the object is unlocked, one of these newly- wakened threads will reacquire the lock and return from its **wait()** call. You cannot control the order in which the threads reacquire the lock.

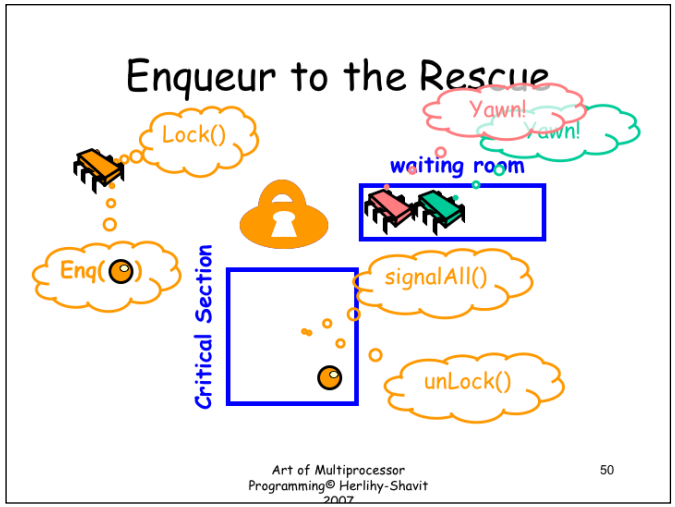
A Monitor Lock

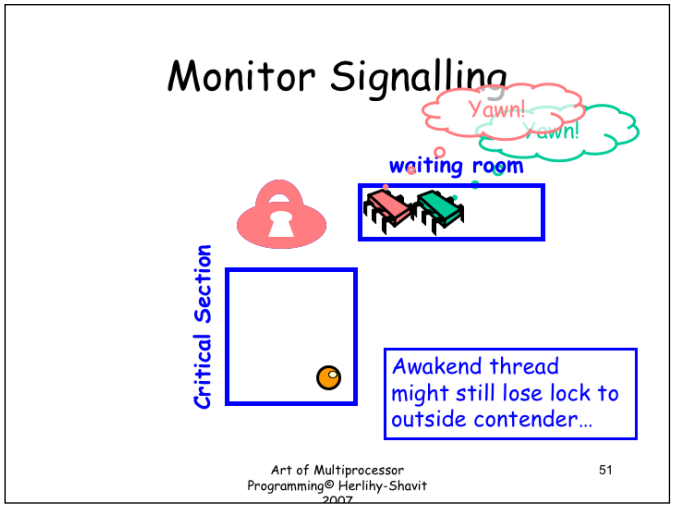


Unsuccessful Deq

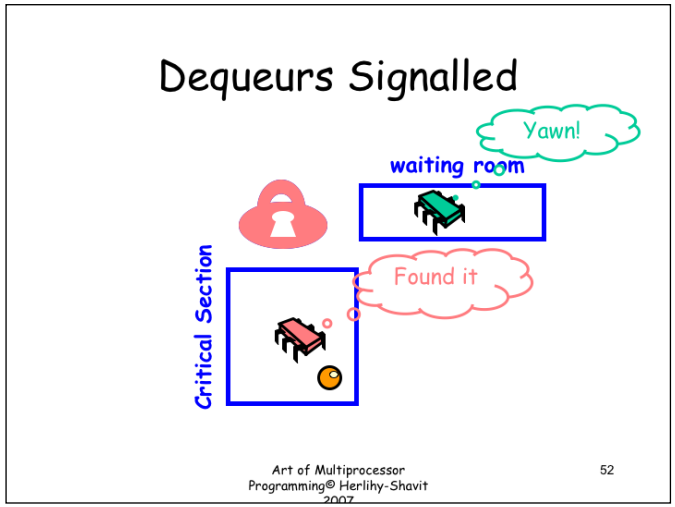



Another One
Lock()
waiting room
Deq()
Critical Section
await()
Oh no,
Empty!
Art of Multiprocessor
Programming© Herlihy-Shavit
2007
49

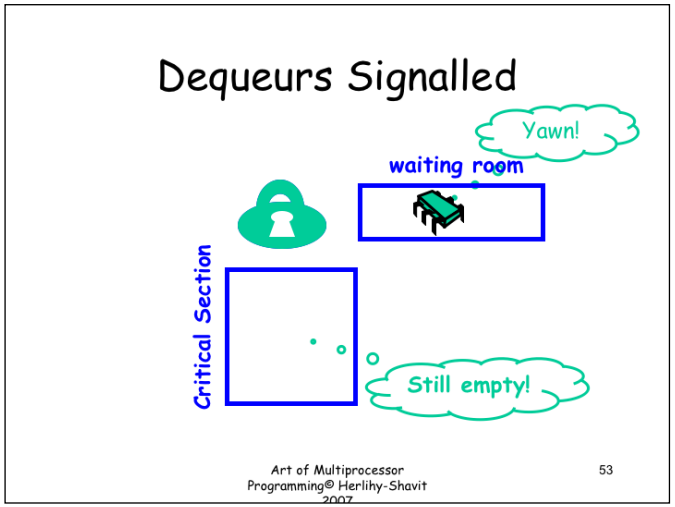

Enqueur to the Rescue
Yawn!
Yawn!
Lock()
waiting room
Enq()
Critical Section
signalAll()
unLock()
Art of Multiprocessor
Programming© Herlihy-Shavit
2007
50



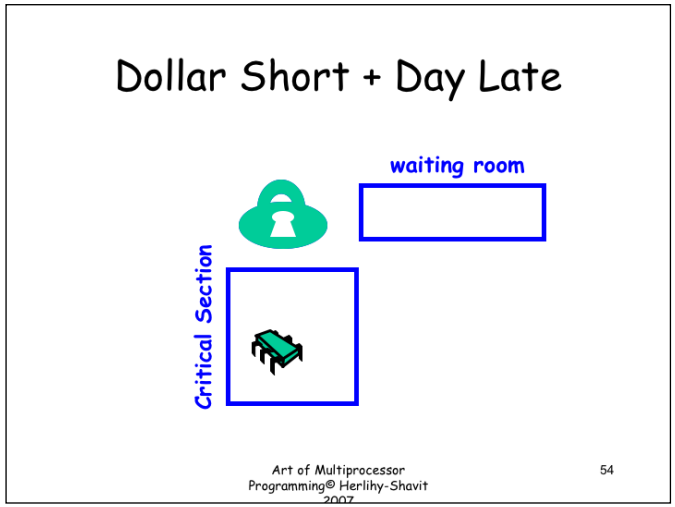
Notice, there may be competition from other threads attempting to lock for the thread. The FIFO order is arbitrary, different monitor locks have different ordering of waiting threads, that is, notify could release the earliest or latest any other waiting thread.



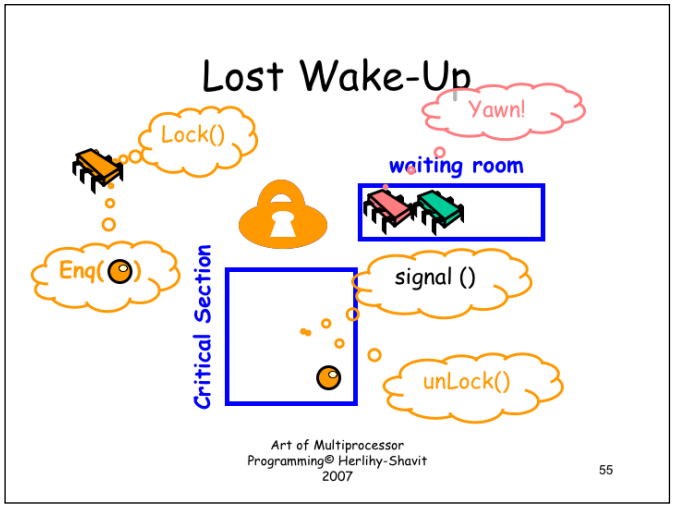
Notice, there may be competition from other threads attempting to lock for the thread. The FIFO order is arbitrary, different monitor locks have different ordering of waiting threads, that is, notify could release the earliest or latest any other waiting thread.



Notice, there may be competition from other threads attempting to lock for the thread. The FIFO order is arbitrary, different monitor locks have different ordering of waiting threads, that is, notify could release the earliest or latest any other waiting thread.



Notice, there may be competition from other threads attempting to lock for the thread. The FIFO order is arbitrary, different monitor locks have different ordering of waiting threads, that is, notify could release the earliest or latest any other waiting thread.



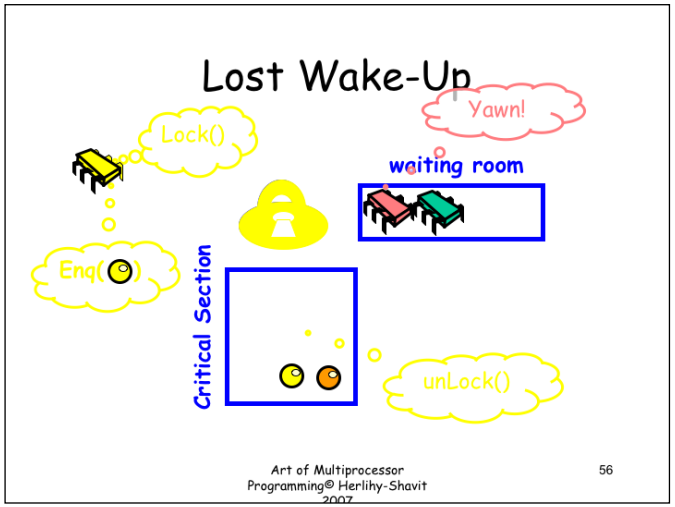
Just as locks are inherently vulnerable to deadlock,
\cCondition{} objects are inherently vulnerable to \emph{lost wakeups},
in which one or more threads wait forever without realizing that the condition
for which they are waiting has become true.

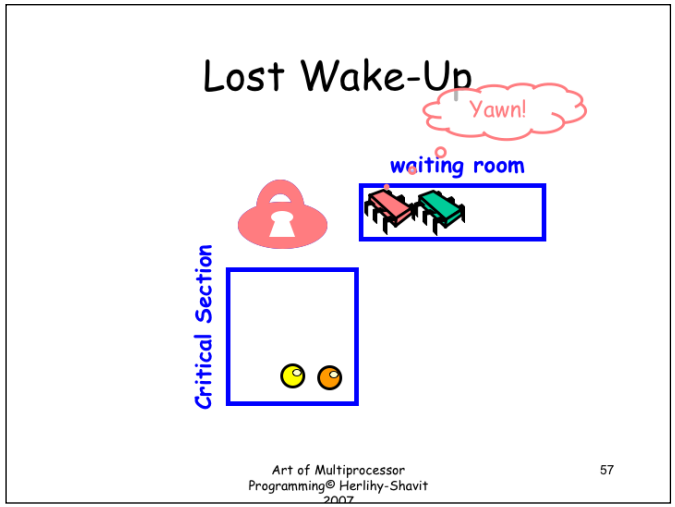
Lost wakeups can occur in subtle ways.
Figure \ref{figure:monitor:lost} shows an ill-considered optimization of the
\cQueue{T} class.
Instead of signaling the \fNotEmpty{} condition each time \mEnq{} enqueues an item,
would it not be more efficient to signal the condition only when
the queue actually transitions from empty to non-empty?
This optimization works as intended if there is only one producer and one consumer,
but it is incorrect if there are multiple producers or consumers.
Consider the following scenario:

consumers A and B both try to dequeue an item from an empty queue,
both detect the queue is empty,
and both block on the \fNotEmpty{} condition.
Producer C enqueues an item in the buffer, and signals \fNotEmpty{},
waking A.
Before A can acquire the lock, however,
another producer D puts a second item in the queue,
and because the queue is not empty, it does not signal \fNotEmpty{}.
Then A acquires the lock, removes the first item,
but B, victim of a lost wakeup,
waits forever even though there is an item in the buffer to be consumed.

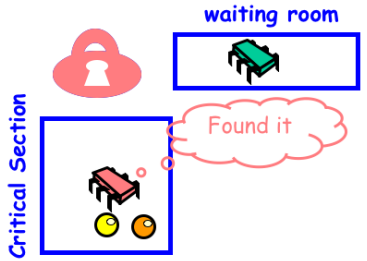
Although there is no substitute for reasoning carefully about your program,
there are simple programming practices that will minimize vulnerability to
lost wakeups.

55

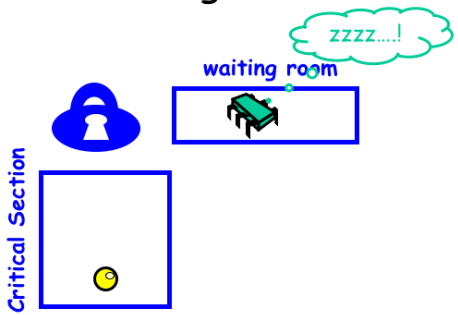

Lost Wake-Up
Lock()
Yawn!
waiting room
Enq()
Critical Section
unLock()
Art of Multiprocessor Programming© Herlihy-Shavit 2007
56


Lost Wake-Up
Yawn!
waiting room
Critical Section
Art of Multiprocessor Programming© Herlihy-Shavit 2007
57

Lost Wake-Up



What's Wrong Here?



Solution to Lost Wakeup

- Always use signalAll and notifyAll
- Not signal and notify

Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail - head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```


Synchronized methods use a monitor lock also.

Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail - head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Each object has an implicit lock with an implicit condition

Synchronized methods use a monitor lock also.

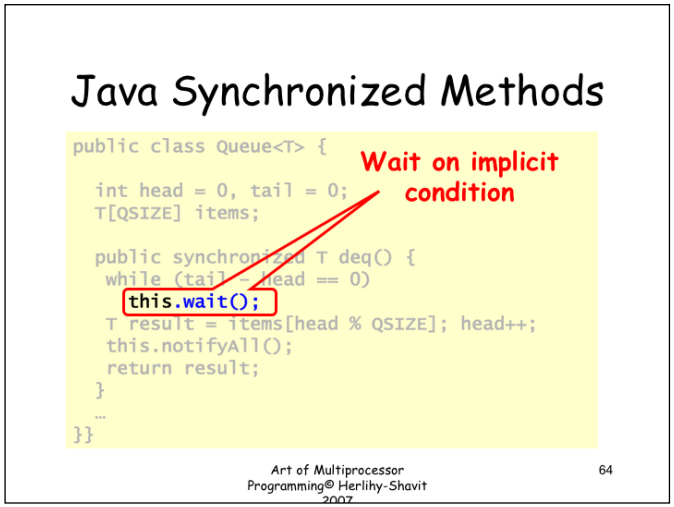
Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail - head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Lock on entry, unlock on return

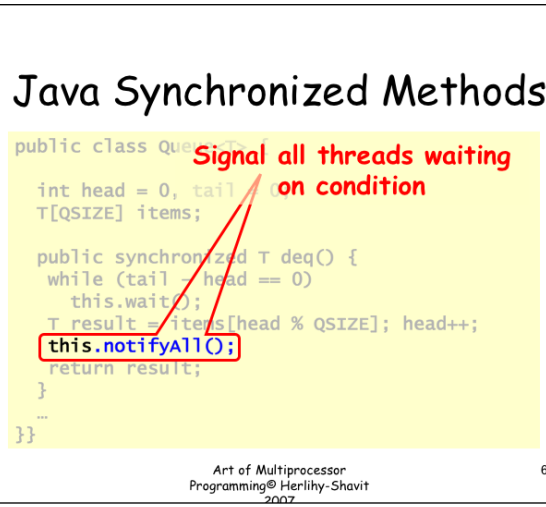
Synchronized methods use a monitor lock also.

Java Synchronized Methods



Synchronized methods use a monitor lock also.

Java Synchronized Methods



Synchronized methods use a monitor lock also.

(Pop!) The Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```


Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq & deq locks

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq lock's associated condition

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Num permits: 0 to capacity

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Head and Tail

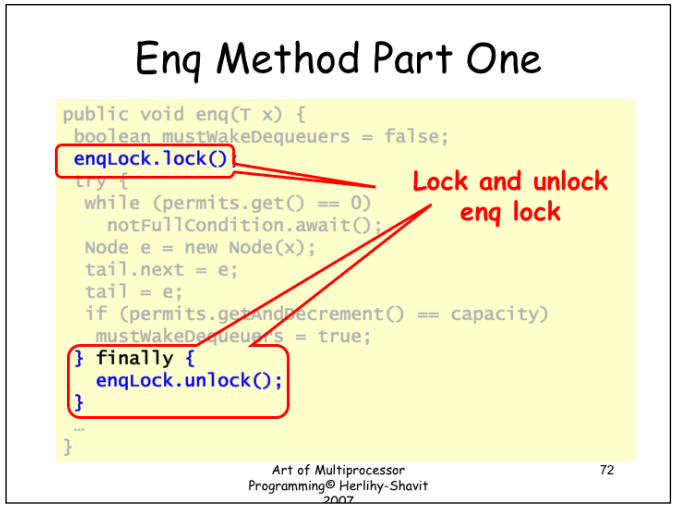
Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```


A remarkable aspect of this queue implementation is that the methods are subtle, but they fit on a single slide.

The \mEnq{} method (Figure~\ref{figure:boundedQueue:enq})
works as follows.
A thread acquires the \fEnqLock{}
(Line \ref{line:bounded:lock}),
and repeatedly reads the \fPermits{} field
(Line \ref{line:bounded:permits}).
While that field
is zero, the queue is full, and the enqueuer must wait until a
dequeuer makes room.
The enqueuer waits by waiting on the \fNotFullCondition{} field
(Line \ref{line:bounded:notfull}),
which releases the enqueue lock temporarily,
and blocks until the condition is signaled.
Each time the thread awakens
(Line \ref{line:bounded:notfull}),

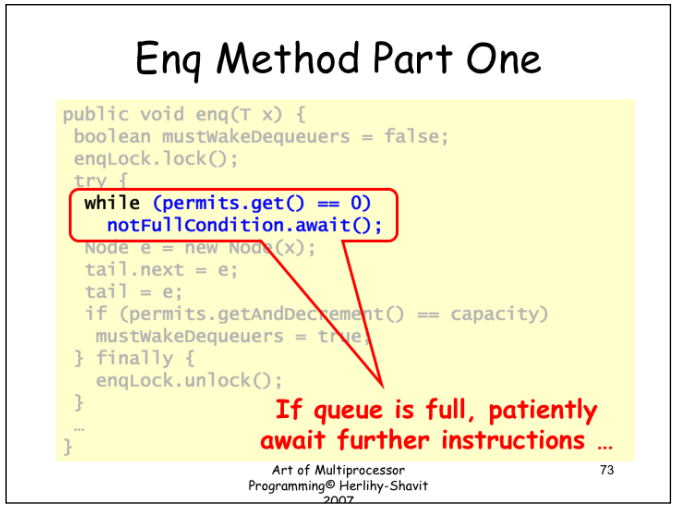
it checks whether the
\fPermits{} field is positive (that is, the queue is not empty)
and if not, goes back to sleep.



A remarkable aspect of this queue implementation is that the methods are subtle, but they fit on a single slide.

The \mEnq{} method (Figure~\ref{figure:boundedQueue:enq})
works as follows.
A thread acquires the \fEnqLock{}
(Line \ref{line:bounded:lock}),
and repeatedly reads the \fPermits{} field
(Line \ref{line:bounded:permits}).
While that field
is zero, the queue is full, and the enqueuer must wait until a
dequeuer makes room.
The enqueuer waits by waiting on the \fNotFullCondition{} field
(Line \ref{line:bounded:notfull}),
which releases the enqueue lock temporarily,
and blocks until the condition is signaled.
Each time the thread awakens
(Line \ref{line:bounded:notfull}),

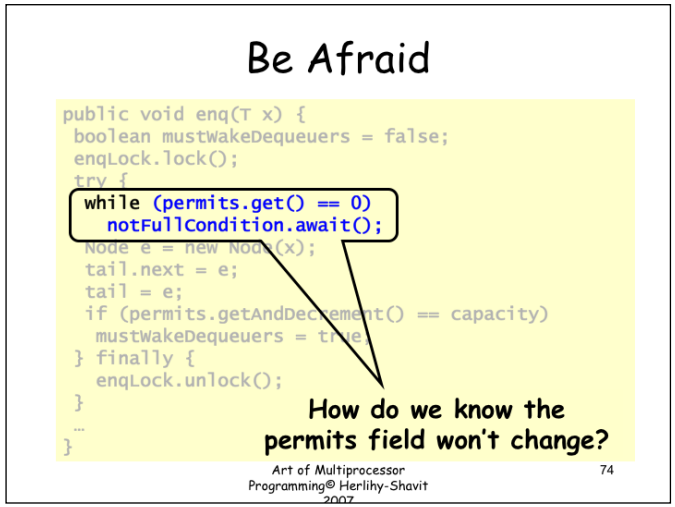
it checks whether the
\fPermits{} field is positive (that is, the queue is not empty)
and if not, goes back to sleep.



Once the number of permits exceeds zero, however, the enqueuer may proceed.
Note that once the enqueuer observes a positive number of permits,
then while the enqueue is in progress no other thread can cause the number of
permits to fall back to zero,
because all the other enqueuers are locked out,
and a concurrent dequeuer can only increase the number of permits.

We must check carefully that this implementation does not suffer from a ``lost-wakeup'' bug. Care is needed because an enqueuer encounters a full queue in two steps: first, it sees that \fPermits{} is zero, and second, it waits on the \fNotFullCondition{} condition until there is room in the queue. When a dequeuer changes the queue from full to not-full, it acquires \fEnqLock{} and signals the \fNotFullCondition{} condition. Even though the \fPermits{} field is not protected by

the \fEnqLock{}, the dequeuer acquires the \fEnqLock{} before it signals the condition, so the dequeuer cannot signal between the enqueuer's two steps.



Once the number of permits exceeds zero, however, the enqueuer may proceed.

Note that once the enqueuer observes a positive number of permits, then while the enqueue is in progress no other thread can cause the number of

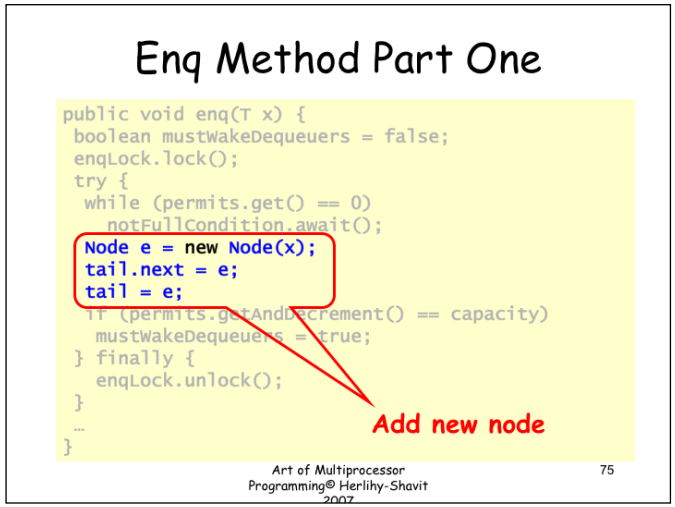
permits to fall back to zero,

because all the other enqueuers are locked out,

and a concurrent dequeuer can only increase the number of permits.

We must check carefully that this implementation does not suffer from a ``lost-wakeup'' bug. Care is needed because an enqueuer encounters a full queue in two steps: first, it sees that \fPermits{} is zero, and second, it waits on the \fNotFullCondition{} condition until there is room in the queue. When a dequeuer changes the queue from full to not-full, it acquires \fEnqLock{} and signals the \fNotFullCondition{} condition. Even though the \fPermits{} field is not protected by

the \fEnqLock{}, the dequeuer acquires the \fEnqLock{} before it signals the condition, so the dequeuer cannot signal between the enqueuer's two steps.



Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

If queue was empty, wake frustrated dequeuers

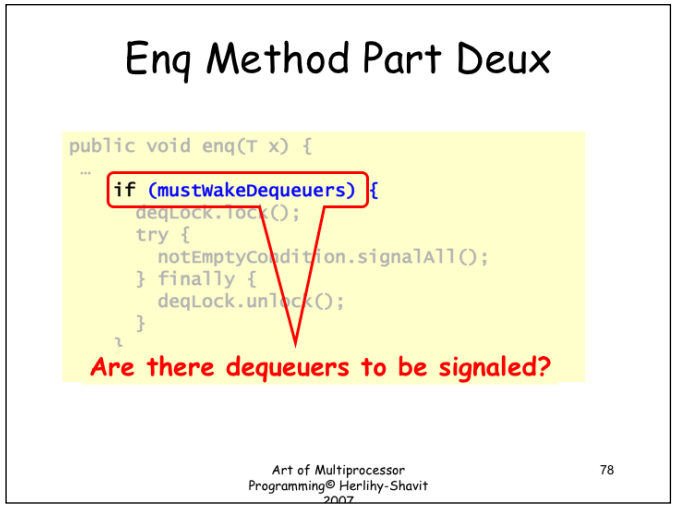
Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers) {
      deqLock.lock();
      try {
        notEmptyCondition.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
  }
```


A remarkable aspect of this queue implementation is that the methods are subtle, but they fit on a single slide.

The \mEnq{} method (Figure~\ref{figure:boundedQueue:enq}) works as follows.
A thread acquires the \fEnqLock{}
(Line \ref{line:bounded:lock}),
and repeatedly reads the \fPermits{} field
(Line \ref{line:bounded:permits}).
While that field
is zero, the queue is full, and the enqueuer must wait until a dequeuer makes room.
The enqueuer waits by waiting on the \fNotFullCondition{} field
(Line \ref{line:bounded:notfull}),
which releases the enqueue lock temporarily,
and blocks until the condition is signaled.
Each time the thread awakens
(Line \ref{line:bounded:notfull}),

it checks whether the
\fPermits{} field is positive (that is, the queue is not empty)
and if not, goes back to sleep.



A remarkable aspect of this queue implementation is that the methods are subtle, but they fit on a single slide.

The \mEnq{} method (Figure~\ref{figure:boundedQueue:enq})
works as follows.
A thread acquires the \fEnqLock{}
(Line \ref{line:bounded:lock}),
and repeatedly reads the \fPermits{} field
(Line \ref{line:bounded:permits}).
While that field
is zero, the queue is full, and the enqueuer must wait until a
dequeuer makes room.
The enqueuer waits by waiting on the \fNotFullCondition{} field
(Line \ref{line:bounded:notfull}),
which releases the enqueue lock temporarily,
and blocks until the condition is signaled.
Each time the thread awakens
(Line \ref{line:bounded:notfull}),

it checks whether the
\fPermits{} field is positive (that is, the queue is not empty)
and if not, goes back to sleep.

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers) {
      deqLock.lock();
      try {
        notEmptyCondition.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
  }
```

Lock and unlock deq lock

The deq() method is symmetric.

Enq Method Part Deux

Signal dequeuers that queue no longer empty

```
    deqLock.lock();
    try {
      notEmptyCondition.signalAll();
    } finally {
      deqLock.unlock();
    }
  }
}
```


The deq() method is symmetric.

The Enq() & Deq() Methods

- Share no locks
  - That's good
- But do share an atomic counter
  - Accessed on every method call
  - That's not so good
- Can we alleviate this bottleneck?

The key insight is that the enqueuer only decrements the counter, and really cares only whether or not the counter value is zero. Symmetrically, the dequeuer cares only whether or not the counter value is the queue capacity.

NB: the deq() method does not explicitly check the permits field, but relies on testing the sentinel Node's next field. Same thing.

Split the Counter

- The **enq()** method
  - Decrements only
  - Cares only if value is **zero**
- The **deq()** method
  - Increments only
  - Cares only if value is **capacity**

The key insight is that the enqueuer only decrements the counter, and really cares only whether or not the counter value is zero. Symmetrically, the dequeuer cares only whether or not the counter value is the queue capacity.

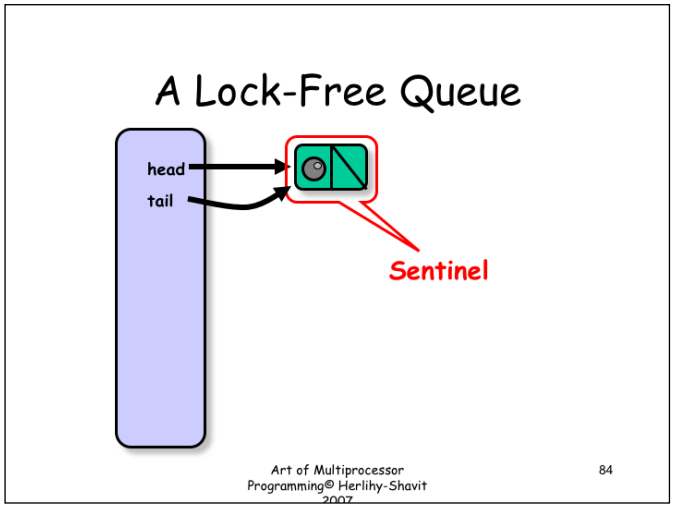
NB: the deq() method does not explicitly check the permits field, but relies on testing the sentinel Node's next field. Same thing.

Split Counter

- Enqueuer decrements enqSidePermits
- Dequeuer increments deqSidePermits
- When enqueuer runs out
  - Locks **deqLock**
  - Transfers permits
- Intermittent synchronization
  - Not with each method call
  - Need both locks! (careful …)

Let's summarize. We split the permits field in to two parts. The enqueuer decrements the enqSidePermits field, and the dequeuer increments the deqSidePermits field. When the enqueuer discovers that its field is zero, then it tries to transfer the dequeuer's permits to itself. Why do this? It replaces synchronization with each method call with sporadic, intermittent synchronization. As a practical matter, an enqueuer that runs out of permits needs to acquire the dequeue lock (why?) (ANSWER: to prevent the dequeuer from incrementing the value at the same time we are trying to copy it).

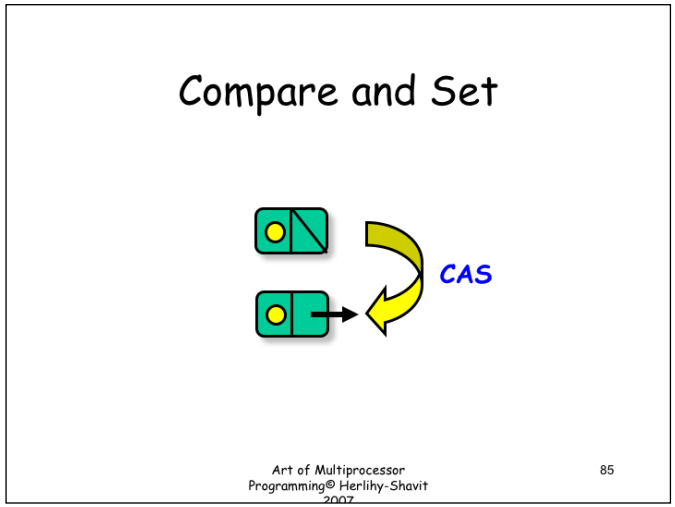
WARNING: Any time you try to acquire a lock while holding another, your ears should prick up (alt: your spider-sense should tingle) because you are just asking for a deadlock. So far, no danger, because all of the methods seen so far never hold more than one lock at a time.



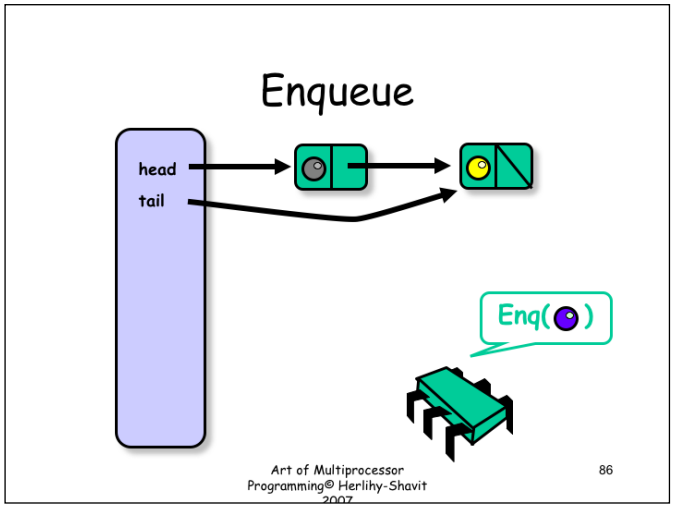
Now we consider an alternative approach that does not use locks. The Lock-free queue is due to Maged Michael & Michael Scott

HUMOR: this construction is due to Maged Michael and Michael Scott. Everyone cites it as Michael Scott, which means that many people (unjustly) think that Michael Scott invented it. The moral is, make sure that your last name is not the same as your advisor's first name.

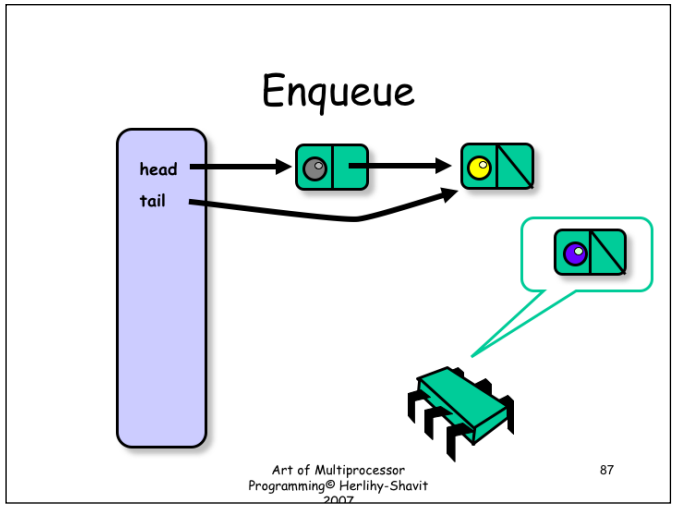
Here too we use a list-based structure. Since the queue is unbounded, arrays would be awkward. As before, We start out with head and tail fields that point to the first and last entries in the list. The first Node in the list is a sentinel Node whose value field is meaningless. The sentinel acts as a placeholder.



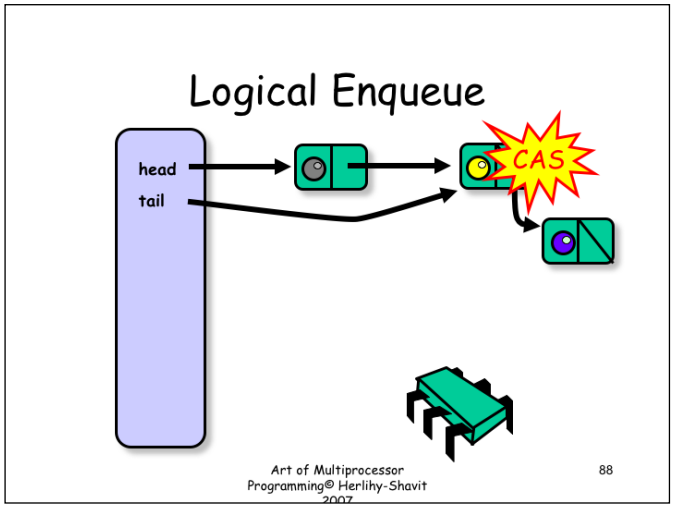
One difference from the lock-based queue is that we use CAS to manipulate the next fields of entries. Before, we did all our synchronization on designed fields in the queue itself, but here we must synchronize on Node fields as well.



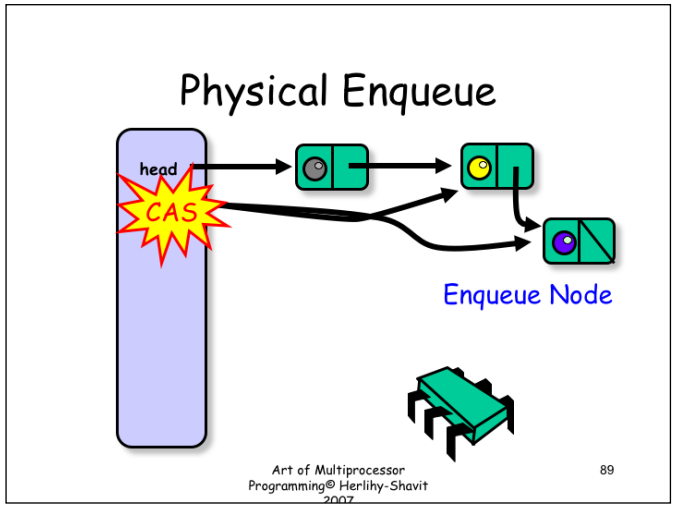
As a first step, the enqueuer calls CAS to sent the tail Node's next field to the new Node.



As a first step, the enqueuer calls CAS to sent the tail Node's next field to the new Node.



As a first step, the enqueuer calls CAS to sent the tail Node's next field to the new Node.



Once the tail Node's next field has been redirected, use CAS to redirect the queue's tail field to the new Node.

Enqueue

- These two steps are not atomic
- The tail field refers to either
  - Actual last Node (good)
  - Penultimate Node (not so good)
- Be prepared!

[pe·nul·ti·mate – adjective -- next to the last: "the penultimate scene of the play"]

As you may have guessed, the situation is more complicated than it appears. The enq() method is supposed to be atomic, so how can we get away with implementing it in two non-atomic steps? We can count on the following invariant: the tail field is a reference to (1) actual last Node, or (2) the Node just before the actual last Node.

Enqueue

- What do you do if you find
  - A trailing **tail**?
- Stop and help fix it
  - If **tail** node has non-*null* next field
  - CAS the queue's **tail** field to **tail.next**
- As in the universal construction

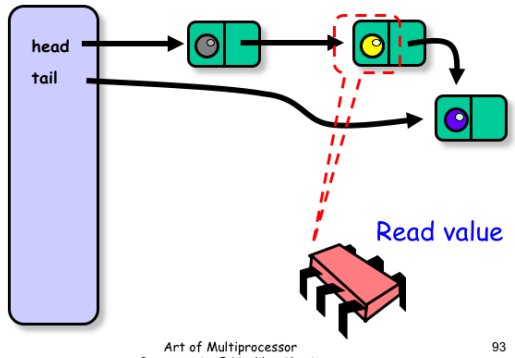
Any method call must be prepared to encounter a tail field that is trailing behind the actual tail. This situation is easy to detect. How? (Answer: tail.next not null). How to deal with it? Fix it! Call compareAndSet to set the queue's tial field to point to the actual last Node in the queuel

When CASs Fail

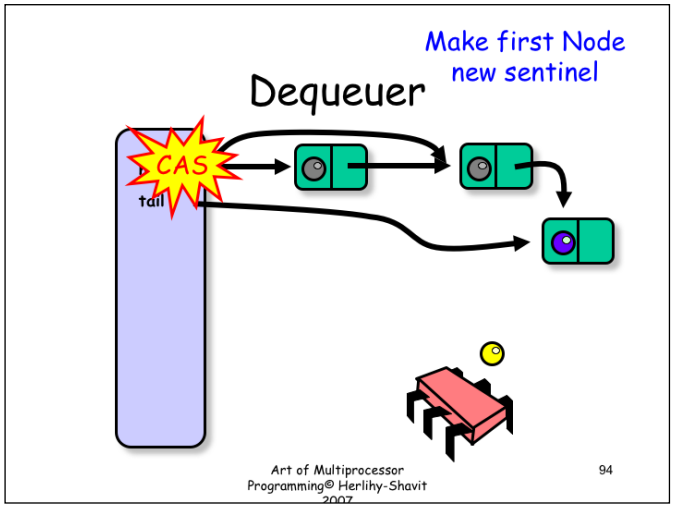
- During logical enqueue
  - Abandon hope, restart
  - Still lock-free (why?)
- During physical enqueue
  - Ignore it (why?)

What do we do if the CAS calls fail? It matters a lot in which CAS call this happened. In Step One, a failed CAS implies a synchronization conflict with another enq() call. Here, the most sensible approach is to abandon the current effort (panic!) and start over. In Step Two, we can ignore a failed CAS, simply because the failure means that some other thread has executed that step on our behalf.

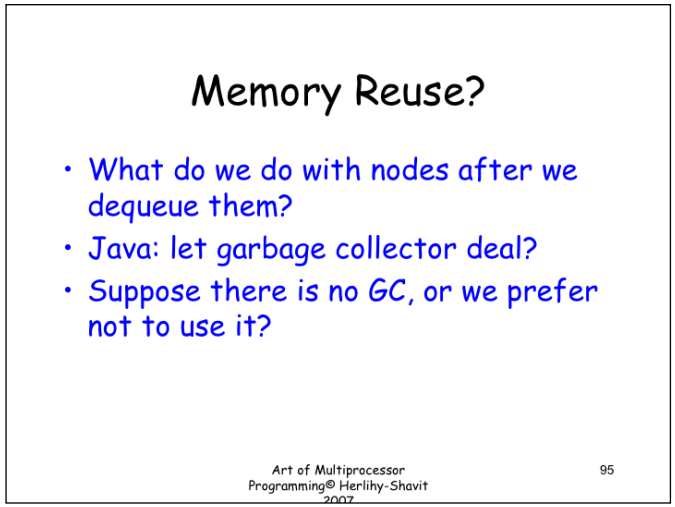
Dequeuer



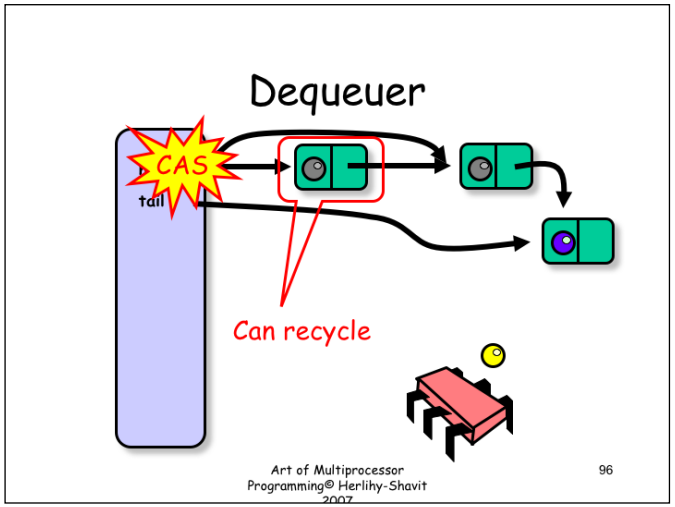
Next, the thread reads the first Node's value, and stores it in a local variable. The slide suggests that the value is nulled out, but in fact there is no need to do so.



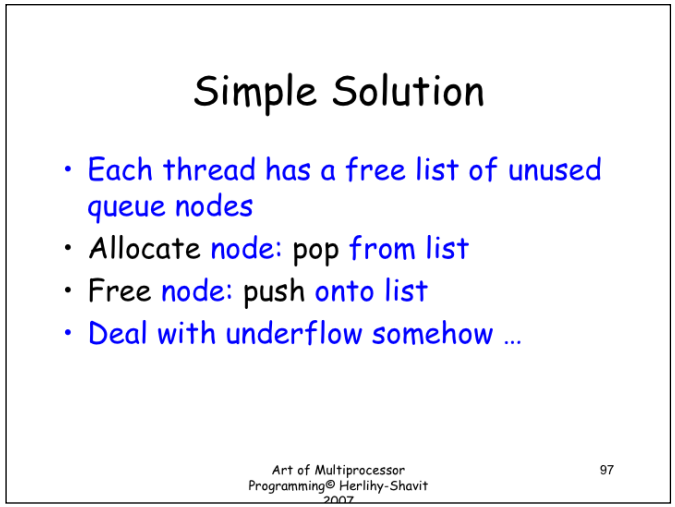
The thread then makes the first Node the new sentinel, and discards the old sentinel. This is the clever trick I mentioned earlier that ensures that we do not need to lock the Node itself. We do not physically remove the Node, we just demote it to sentinel.



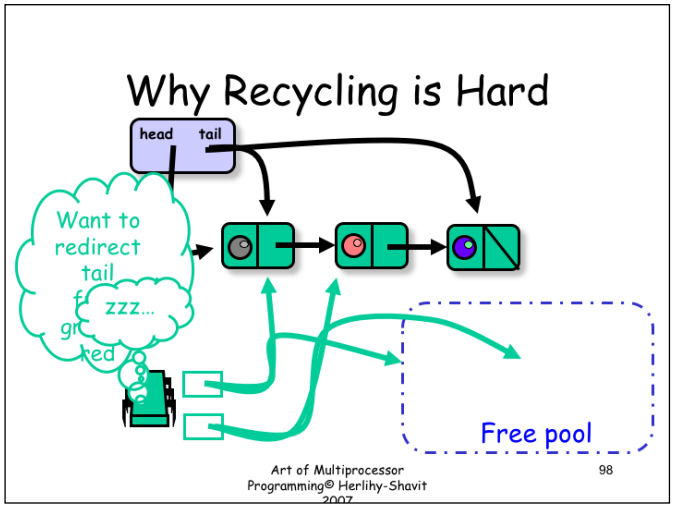
Let's take a brief detour. What do we do with nodes after we remove them from the queue? This is not an issue with the lock-based implementation, but it requires some thought here. In Java, we can just let the garbage collector recycle unused objects. But what if we are operating in an environment where there is no garbage collector, or where we think we can recycle memory more efficiently?



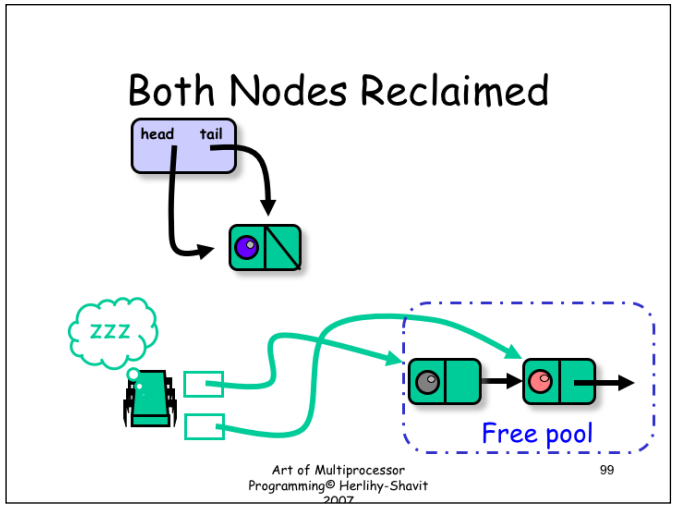
Let us look at the dequeue method from a different perspective. When we promote the prior first Node to sentinel, what do we do with the old sentinel? Seems like a good candidate for recylcling.



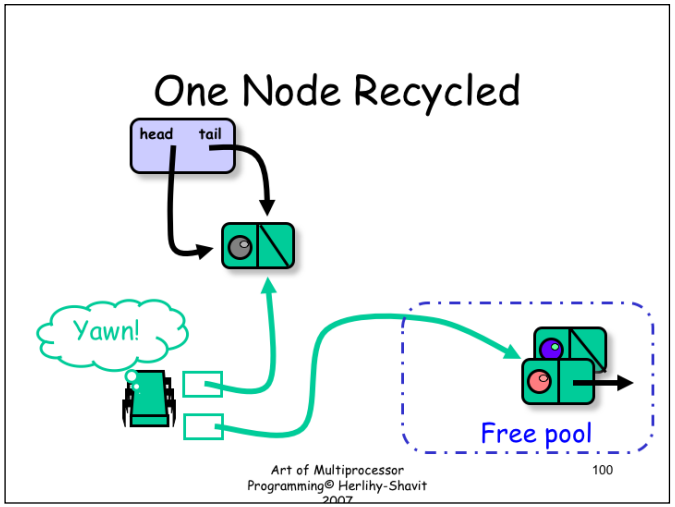
The most reasonable solution is to have each thread manage its own pool of unused nodes. When a thread needs a new Node, it pops one from the list. No need for synchronization. When a thread fees A Node, it pushes the newly-freed Node onto the list. What do we do when a thread runs out of nodes? Perhaps we could just malloc() more, or perhaps we can devise a shared pool. Never mind.



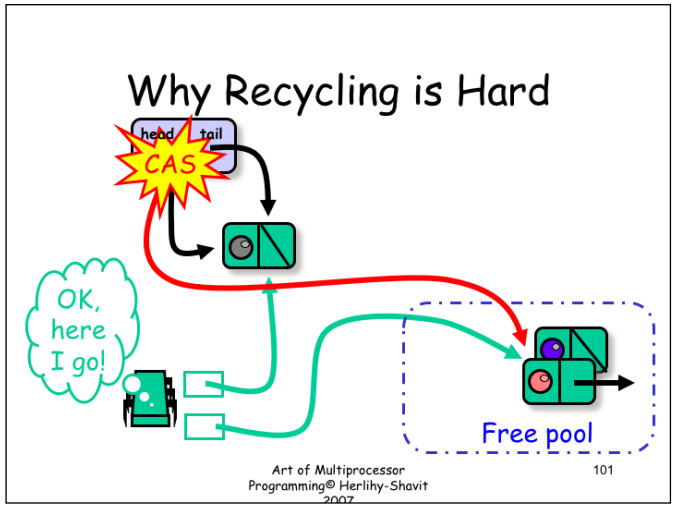
Now the green thread goes to sleep, and other threads dequeue the red object and the green object, sending the prior sentinel and prior red Node to the respective free pools of the dequeuing threads.



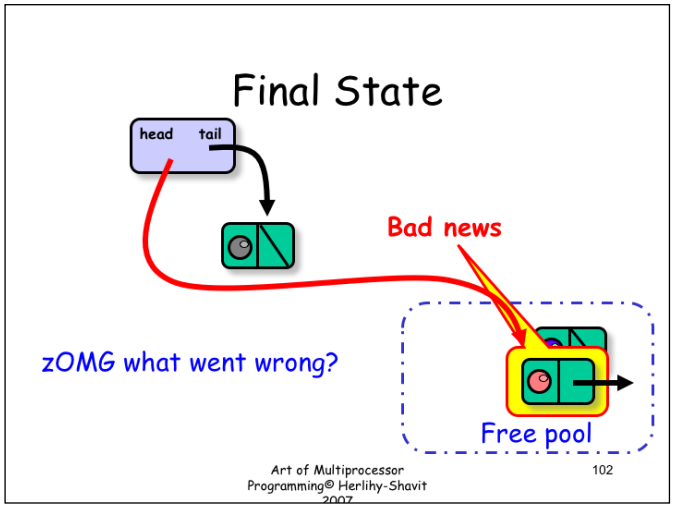
Despite what you might think, we are perfectly safe, because any thread that tries to CAS the head field must fail, because that field has changed. Now assume that the enough deq() and enq() calls occur such that the original sentinel Node is recyled, and again becomes a sentinel Node for an empty queue.



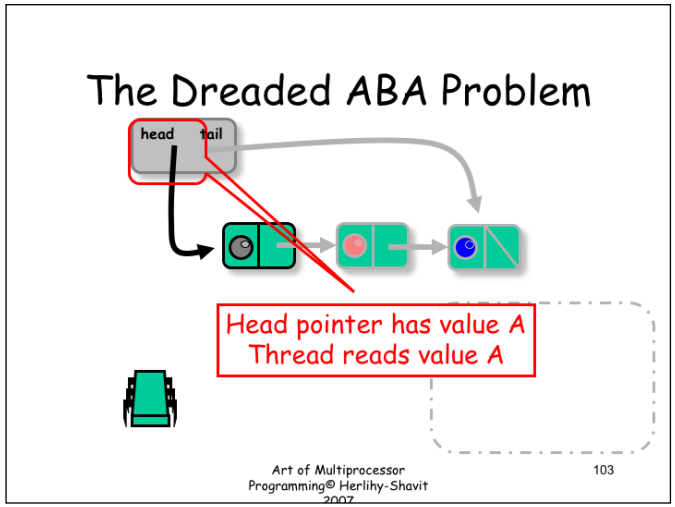
Now the green thread wakes up. It applies CAS to the queue's head field ..



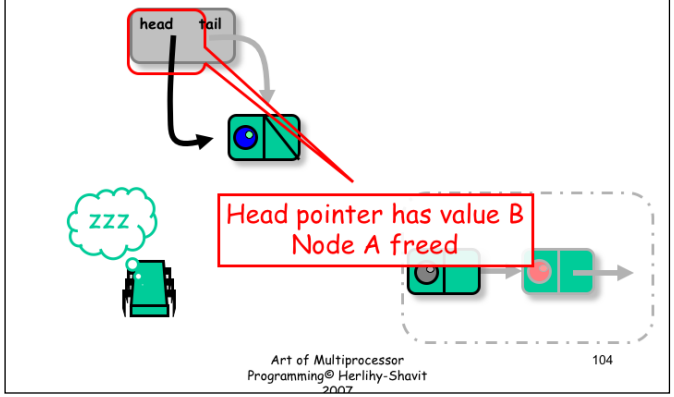
Surprise! It works! The problem is that the bit-wise value of the head field is the same as before, even though its meaning has changed. This is a problem with the way the CAS operation is defined, nothing more.



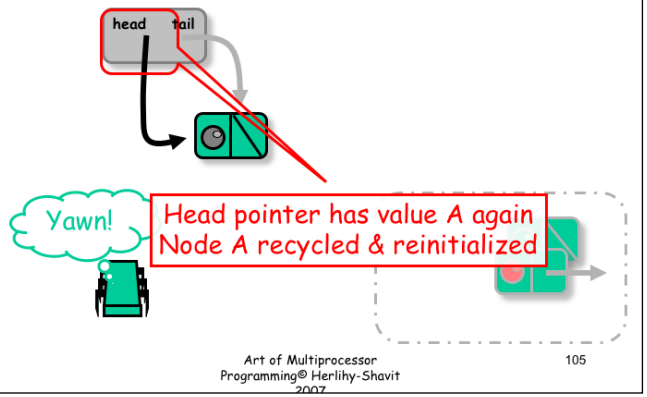
In the end, the tail pointer points to a sentinel node, while the head points to node in some thread's free list. What went wrong?



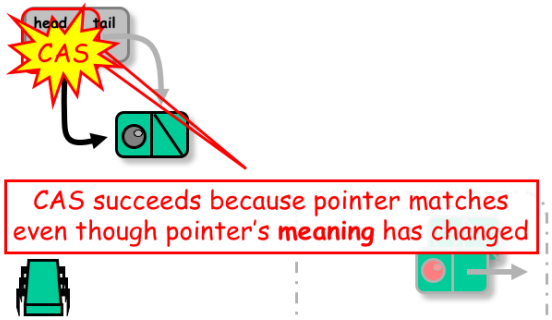
Dreaded ABA continued



Dreaded ABA continued



Dreaded ABA continued



The Dreaded ABA Problem

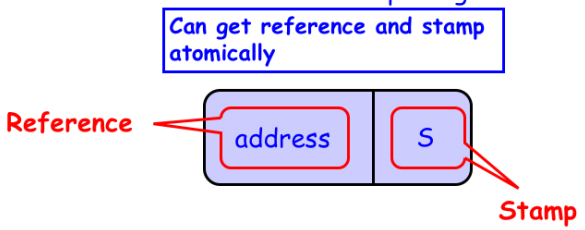
- Is a result of CAS() semantics
  - I blame Sun, Intel, AMD, …
- Not with Load-Locked/Store-Conditional
  - Good for IBM?

Dreaded ABA – A Solution

- Tag each pointer with a counter
- Unique over lifetime of node
- Pointer size vs word size issues
- Overflow?
  - Don't worry be happy?
  - Bounded tags?
- AtomicStampedReference class

Atomic Stamped Reference

- **AtomicStampedReference** class
  - Java.util.concurrent.atomic package

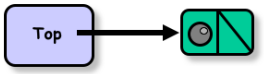


The actual implementation in Java is by having an indirection through a special sentinel node. If the special node exists then the mark is set, and if the node does not, then the bit is false. The sentinel points to the desired address so we pay a level of indirection. In C, using alignment on 32 bit architectures or on 64 bit architectures, we can "steal" a bit from the pointer.

Concurrent Stack

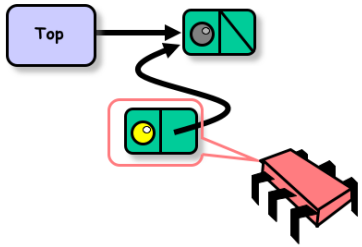
- Methods
  - **push(x)**
  - **pop()**
- Last-in, First-out (LIFO) order
- Lock-Free!

Empty Stack

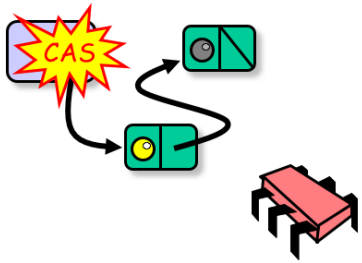


To \mPush{} an item of type \cT{} into the stack
(As in our \cLockFreeQueue{} implementation, we do not allow
insertion of a \Jnull{} value), a
thread creates a new node (Line~\ref{line:elimination:new}),
and then calls the \mTryPush{} method that directs its next pointer to the node
pointed-to by the \fTop{}, and then attempts to swing the \fTop{}
to point to the new node using a \mCompareAndSet{} method call. If \mTryPush{} succeeds,
\mPush{} returns, and if not, the \mTryPush{} attempt is repeated.

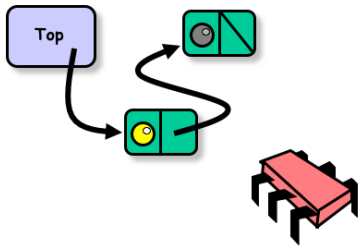
Push



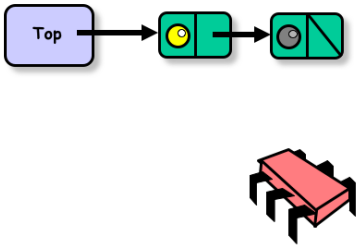
Push



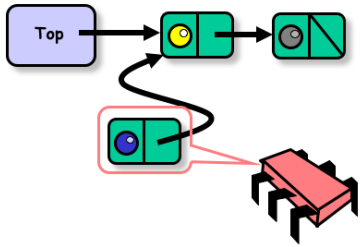
Push



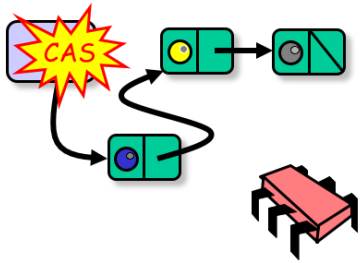
Push



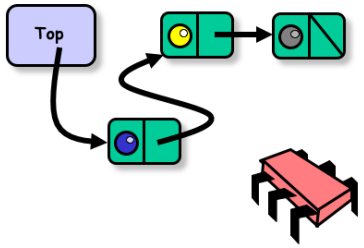
Push



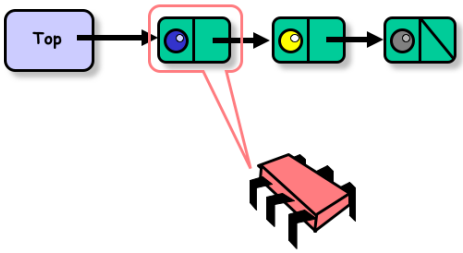
Push



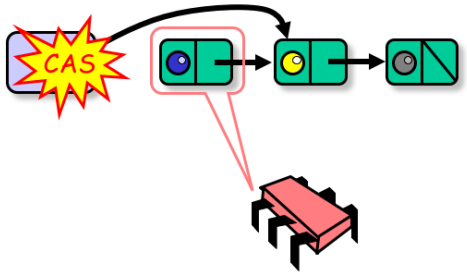
Push



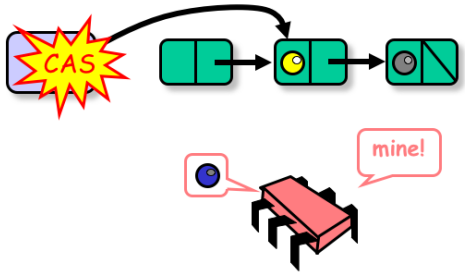
Pop



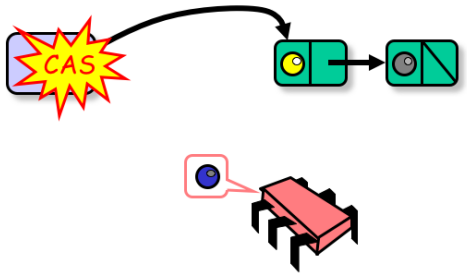
Pop



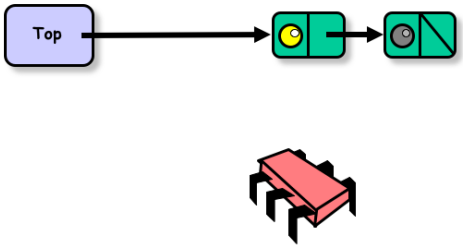
Pop



Pop



Pop



Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top =
    new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
  Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff();
  }}
```


Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public Boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
}}
```

tryPush attempts to push a node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Read top value

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
}}
```

current top will be new node's successor

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
```

Try to swing top, return success or failure

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff();
}}
```

Push calls tryPush

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Create new node

Lock-free Stack

Makes scheduling benevolent so Method is effectively wait-free

If tryPush() fails, back off before retrying

```
public class
  private Atc
AtomicReferen
public boolean tryPush(Node node){
      Node oldT
      node.next
      return(to
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Lock-free Stack

- Good
  - No locking
- Bad
  - Without GC, fear ABA
  - Without backoff, huge contention at top
  - In any case, no parallelism

Big Question

- Are stacks *inherently* sequential?
- Reasons why
  - Every **pop()** call fights for top item
- Reasons why not
  - Stay tuned …

Elimination-Backoff Stack

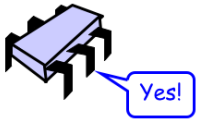
- How to
  - "turn contention into parallelism"
- Replace familiar
  - **exponential backoff**
- With alternative
  - **elimination-backoff**

Observation

Push( )

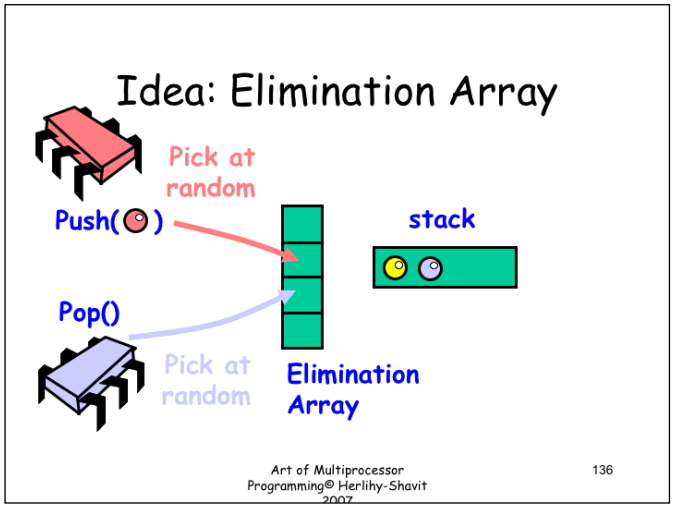
linearizable stack

Pop()

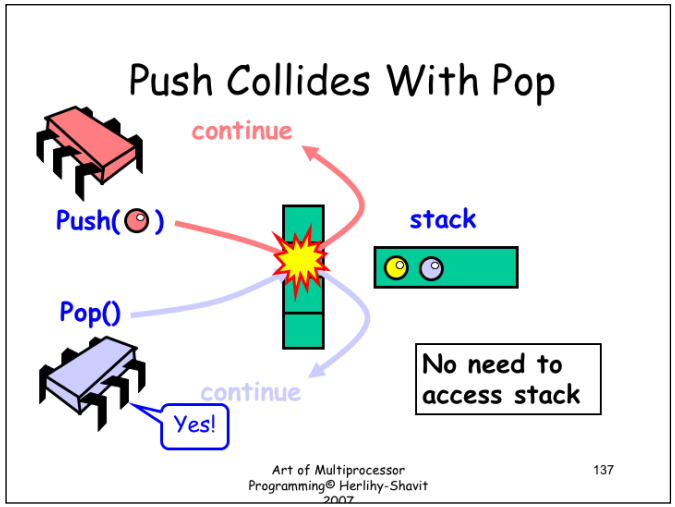


After an equal number of pushes and pops, stack stays the same

Take any linearizable stack, for example, the lock-free queue.

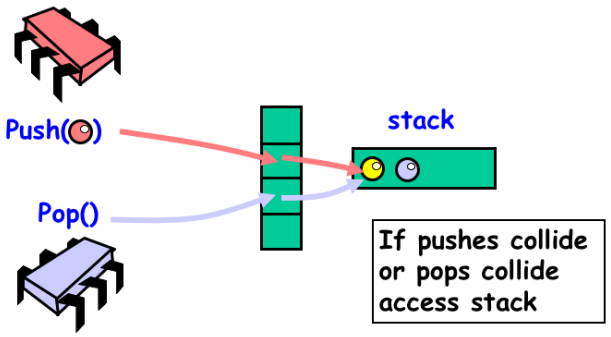


Pick random locations in the array.



No need to access stack since it stays the same anyhow. Just exchange the values.

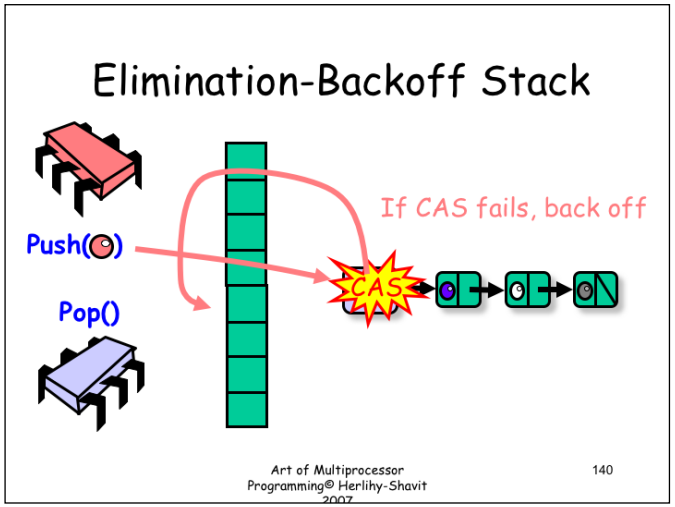
No Collision



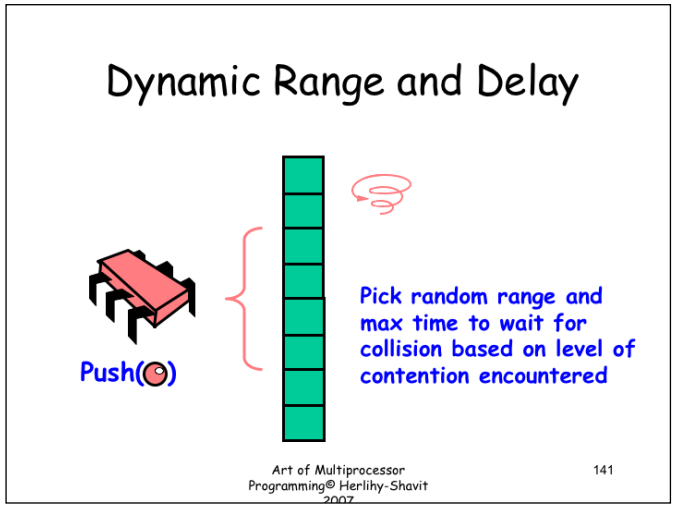
No need to access stack since it stays the same anyhow. Just exchange the values.

Elimination-Backoff Stack

- Lock-free stack + elimination array
- Access Lock-free stack,
  - If **uncontended**, apply operation
  - if **contended**, back off to elimination array and attempt elimination



No need to access stack since it stays the same anyhow. Just exchange the values.

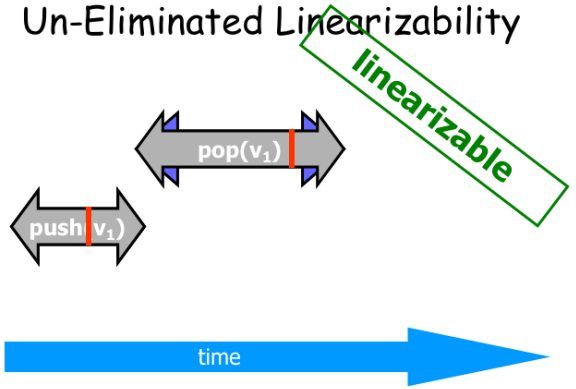


As the contention grows,

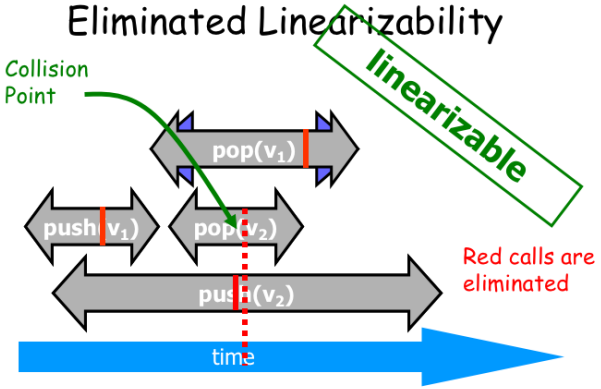
Linearizability

- Un-eliminated calls
  - linearized as before
- Eliminated calls:
  - linearize pop() immediately after matching push()
- Combination is a linearizable stack

Un-Eliminated Linearizability



Eliminated Linearizability



Backoff Has Dual Effect

- Elimination introduces parallelism
- Backoff onto array cuts contention on lock-free stack
- Elimination in array cuts down total number of threads ever accessing lock-free stack

Elimination Array

```
public class EliminationArray {
 private static final int duration = ...;
 private static final int timeUnit = ...;
 Exchanger<T>[] exchanger;
 public EliminationArray(int capacity) {
  exchanger = new Exchanger[capacity];
  for (int i = 0; i < capacity; i++)
   exchanger[i] = new Exchanger<T>();
  …
  }
  …
}
```


Elimination Array

```
public class EliminationArray {
 private static final int duration = ...;
 private static final int timeUnit = ...;
 Exchanger<T>[] exchanger;
 public EliminationArray(int capacity) {
  exchanger = new Exchanger[capacity];
  for (int i = 0; i < capacity; i++)
   exchanger[i] = new Exchanger<T>();
  …
  }
  …
}
```

An array of exchangers

A Lock-Free Exchanger

```
public class Exchanger<T> {
 AtomicStampedReference<T> slot
  = new AtomicStampedReference<T>(null, 0);
```


A Lock-Free Exchanger

```
public class Exchanger<T> {
 AtomicStampedReference<T> slot
  = new AtomicStampedReference<T>(null, 0);
```

Atomically modifiable reference + time stamp

Atomic Stamped Reference

- AtomicStampedReference class
  - Java.util.concurrent.atomic package
- In C or C++:

The actual implementation in Java is by having an indirection through a special sentinel node. If the special node exists then the mark is set, and if the node does not, then the bit is false. The sentinel points to the desired address so we pay a level of indirection. In C, using alignment on 32 bit architectures or on 64 bit architectures, we can "steal" a bit from the pointer.

Extracting Reference & Stamp

```
Public T get(int[] stampHolder);
```

Extracting Reference & Stamp

```
Public T get(int[] stampHolder);
```

Returns reference to object of type T

Returns stamp at array index 0!

Exchanger Status

```
enum Status {EMPTY, WAITING, BUSY};
```

Exchanger Status

```
enum Status {EMPTY, WAITING, BUSY};
```

Nothing yet

Exchange Status

```
enum Status {EMPTY, WAITING, BUSY};
```

Nothing yet

One thread is waiting for rendez-vous

Exchange Status

```
enum Status {EMPTY, WAITING, BUSY};
```

Nothing yet

One thread is waiting for rendez-vous

Other threads busy with rendez-vous

The Exchange

```
public T Exchange(T myItem, long nanos)
   throws TimeoutException {
 long timeBound = System.nanoTime() + nanos;
 int[] stampHolder = {EMPTY};
 while (true) {
  if (System.nanoTime() > timeBound)
    throw new TimeoutException();
  T herItem = slot.get(stampHolder);
  int stamp = stampHolder[0];
  switch(stamp) {
   case EMPTY: …   // slot is free
   case WAITING: … // someone waiting for me
   case BUSY: …    // others exchanging
  }
 }
```

The Exchange

```
public T Exchange(T myItem, long nanos)
   throws TimeoutException {
 long timeBound = System.nanoTime() + nanos;
 int[] stampHolder = {EMPTY};
 while (true) {
  if (System.nanoTime() > timeBound)
    throw new TimeoutException();
  T herItem = slot.get(stampHolder);
  int stamp = stampHolder[0];
  switch(stamp) {
   case EMPTY: …   // slot is free
   case WAITING: … // someone waiting for me
   case BUSY: …    // others exchanging
  }
 }
```

Item & timeout

The Exchange

```
public T Exchange(T myItem, long nanos)
   throws TimeoutException {
 long timeBound = System.nanoTime() + nanos;
 int[] stampHolder = {EMPTY};
 while (true) {
  if (System.nanoTime() > timeBound)
    throw new TimeoutException();
  T herItem = slot.get(stampHolder);
  int stamp = stampHolder[0];
  switch(stamp) {
   case EMPTY: …   // slot is free
   case WAITING: … // someone waiting for me
   case BUSY: …    // others exchanging
  }
 }
```

Array to hold timestamp

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {0};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp) {
        case EMPTY:   // slot is free
        case WAITING: // someone waiting for me
        case BUSY:    // others exchanging
        }
    }}
```

Loop until timeout

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {0};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp) {
        case EMPTY:   // slot is free
        case WAITING: // someone waiting for me
        case BUSY:    // others exchanging
        }
    }}
```

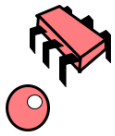
Get other's item and timestamp

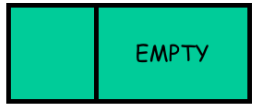
The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
  long timeBound = System.nanoTime() + nanos;
  while (true) {
    if (System.nanoTime() > timeBound)
      throw new TimeoutException();
    T herItem = slot.get(stampHolder);
    int stamp = stampHolder[0];
    switch(stamp) {
      case EMPTY: …   // slot is free
      case WAITING: … // someone waiting for me
      case BUSY: …    // others exchanging
      }
  }}
```

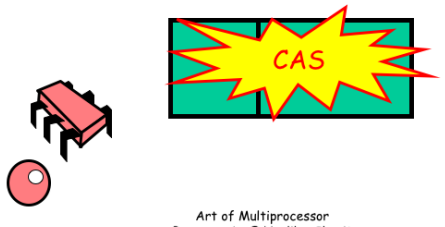
Exchanger slot has three states

Lock-free Exchanger

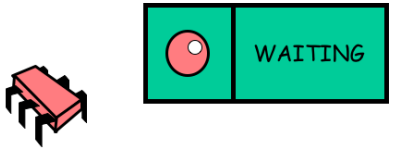


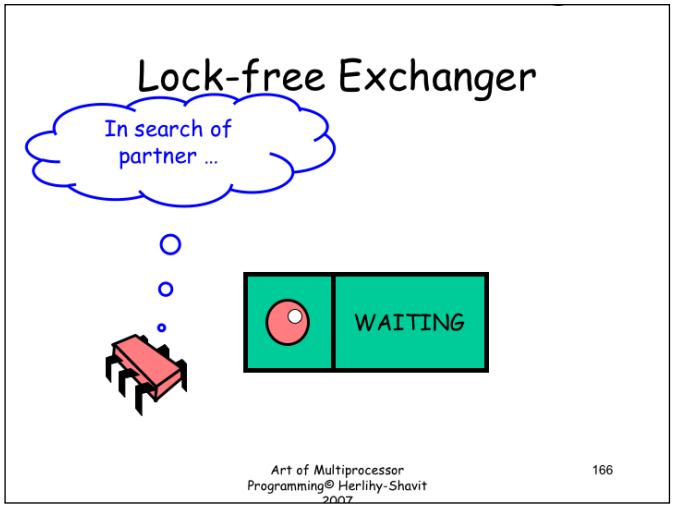


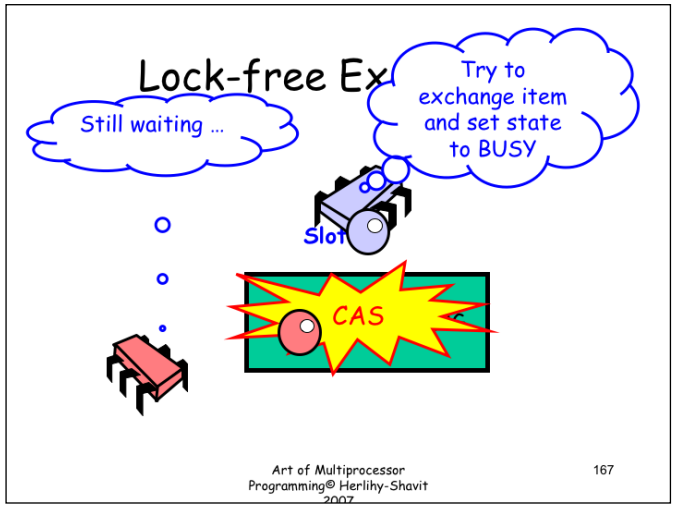
Lock-free Exchanger



Lock-free Exchanger

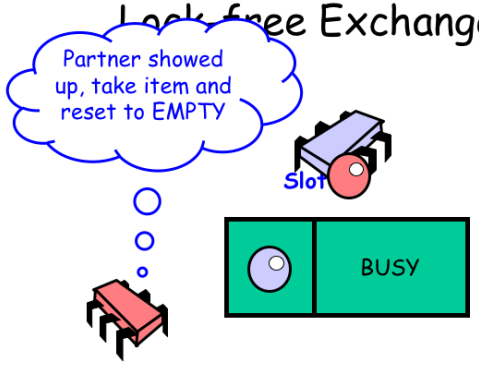




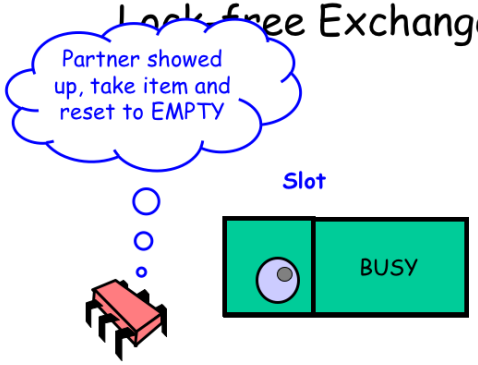


Obviously some other thread can try and set the state to 2 also and only the winner does the exchange.

Lock-free Exchanger



Lock-free Exchanger



Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, EMPTY);
         return herItem;
     }
} break;
```


We present algorithm with timestamps so that you will not worry about possible ABA but actually can implement this scheme with counter that has just 3 values 0,1, and 2.

Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, EMPTY);
         return herItem;
     }
} break;
```

Slot is free, try to insert myItem and change state to WAITING

Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, EMPTY);
         return herItem;
     }
} break;
```

Loop while still time left to attempt exchange

Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, WAITING,
BUSY)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, EMPTY);
         return herItem;
     }
} break;
```

Get item and stamp in slot and check if state changed to BUSY

Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, EMPTY);
         return herItem;
     }
} break;
```

If successful reset slot state to EMPTY

Notice that we do not need a CAS here, only a set which is a write, because only the thread that changed the state from 0 to 1 can reset it from 2 to 0.

Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, WAITING,
BUSY)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, EMPTY);
         return herItem;
     }
} break;
```

and return item found in slot

Exchanger State EMPTY

Otherwise we ran out of time, try to reset state to EMPTY, if successful time out

```
em, myItem, EMPTY,
imeBound){
lder);
) {
  slot.set(null, EMPTY);
  return herItem;
}}
if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
} else {
  herItem = slot.get(stampHolder);
  slot.set(null, EMPTY);
  return herItem;
}
} break;
```


Exchanger State EMPTY

If reset failed, someone showed up after all, take that item

```
WAITING,
while (System.nanoTime() < timeBound){
  herItem = slot.get(stampHolder);
  if (stampHolder[0] == BUSY) {
    slot.set(null, EMPTY);
    return herItem;
  }}
if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
} else {
  herItem = slot.get(stampHolder);
  slot.set(null, EMPTY);
  return herItem;
}
} break;
```

Exchanger State EMPTY

Set slot to EMPTY with new timestamp and return item found

```
case EMPTY: // slot is free
    if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
     } else {
       herItem = slot.get(stampHolder);
       slot.set(null, EMPTY);
       return herItem;
    }
} break;
```

Again notice that we do not need a CAS here, only a set which is a write, because only the thread that changed the state from 0 to 1 can reset it from 2 to 0…

Exchanger State EMPTY

If initial CAS failed then someone else changed state from EMPTY to WAITING so retry from start

```
case EMPTY: // slot is free
    if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
     } else {
       herItem = slot.get(stampHolder);
       slot.set(null, EMPTY);
       return herItem;
    }
} break;
```

Again notice that we do not need a CAS here, only a set which is a write, because only the thread that changed the state from 0 to 1 can reset it from 2 to 0…

States WAITING and BUSY

```
case WAITING:   // someone waiting for me
        if (slot.compareAndSet(herItem, myItem,
WAITING, BUSY))
          return herItem;
        break;
case BUSY:   // others in middle of exchanging
        break;
default:  // impossible
        break;
      }
    }
  }
}
```


States WAITING and BUSY

```
case WAITING:   // someone waiting for me
        if (slot.compareAndSet(herItem, myItem,
                               WAITING, BUSY))
          return herItem;
        break;
case BUSY:   // others in middle of exchanging
        break;
default:
      }
    }
  }
}
```

state WAITING means someone is waiting for an exchange, so attempt to CAS my item in and change state to BUSY

States WAITING and BUSY

```
case WAITING:   // someone waiting for me
        if (slot.compareAndSet(herItem, myItem,
                               WAITING, BUSY))
          return herItem;
        break;
case BUSY:   // others in middle of exchanging
        break;
default:  // impossible
        break;
      }
    }
  }
}
```

If successful return her item, state is now BUSY, otherwise someone else took her item so try again from start

States WAITING and BUSY

```
case WAITING:   // someone waiting for me
        if (slot.compareAndSet(herItem, myItem,
WAITING, BUSY))
          return herItem;
        break;
case BUSY:   // others in middle of exchanging
        break;
default:  // impossible
        break;
      }
    }
  }
}
```

If state is BUSY then some other threads are using slot to exchange so start again

The Exchanger Slot

- Exchanger is lock-free
- Because the only way an exchange can fail is if others repeatedly succeeded or no-one showed up
- The slot we need does not require symmetric exchange

Notice that The slot we need does not require symmetric exchange, only oush needs to pass item to pop.

Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int Range) throws
TimeoutException {
    int slot = random.nextInt(Range);
    int nanodur = convertToNanos(duration, timeUnit));
    return (exchanger[slot].exchange(value, nanodur)
}}
```

Elimination Array

```
public class EliminationArray {

public T visit(T value, int Range) throws
TimeoutException {
    int slot = random.nextInt(Range);
    int nanodur = convertToNanos(duration, timeUnit));
    return (exchanger[slot].exchange(value, nanodur)
}}
```

visit the elimination array with a value and a range (duration to wait is not dynamic)

Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int Range) throws
TimeoutException {
    int slot = random.nextInt(Range);
    int nanodur = convertToNanos(duration, timeUnit));
    return (exchanger[slot].exchange(value, nanodur)
}}
```

Pick a random array entry

Elimination Array

```
public class EliminationArray {
…
public T Exchange value or time out
TimeoutException {
    int slot = random.nextInt(range);
    int nanodur = convertToNanos(duration, timeUnit));
  return (exchanger[slot].exchange(value, nanodur)
}}
```


Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
      eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
    }
}
```

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
      eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
    }
}
```

First try to push

Elimination Stack Push

```
public void push(T value) {
...
 whi
  if (tryPush(node)) {
    return;
} else try {
    T otherValue =
    eliminationArray.visit(value,policy.Range);
    if (otherValue == null) {
       return;
    }
}
```

If failed back-off to try to eliminate

Elimination Stack Push

```
public void push(T value) {
...
 whil
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
      eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
      }
}
```

Value being pushed and range to try

Elimination Stack Push

```
public void push(T value) {
...
 while (
  if (tr
    return;
  } else try {
      T otherValue =
      eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
      }
}
```

Only a pop has null value so elimination was successful

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
      eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
        return;
    }
}
```

Else retry push on lock-free stack

Elimination Stack Pop

```
public T pop() {
 ...
 while (true) {
  if (tryPop()) {
   return returnNode.value;
   } else
      try {
        T otherValue =
        eliminationArray.visit(null,policy.Range;
        if  otherValue != null) {
         return otherValue;
         }
      }
}}
```

Elimination Stack Pop

```
public T pop() {
 ...
 while
  if
   re
   } else
      try {
        T otherValue =
        eliminationArray.visit(null,policy.Range;
        if ( otherValue != null) {
         return otherValue;
        }
      }
}}
```

Same as push, if non-null other thread must have pushed so elimnation succeeds

Summary

- We saw both lock-based and lock-free implementations of
- queues and stacks
- Don't be quick to declare a data structure inherently sequential
  - Linearizable stack is not inherently sequential
- ABA is a real problem, pay attention