CHALMERS

UNIVERSITY OF
GOTHENBURG

Lock-free programming

Lecture 11 of TDA384/DIT391
Principles of Concurrent Programming

Nir Piterman

Chalmers University of Technology | University of Gothenburg
SP1 2020/2021

Based on course slides by Carlo A. Furia and Sandro Stucki

Today’s menu

Parallel linked queues

Software transactional memory

Synchronization costs

A number of factors challenge designing correct and efficient parallelizations:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

Synchronization costs

A number of factors challenge designing correct and efficient parallelizations:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

In this class, we present:

- a lock-free queue data structure, which involves minimal synchronization costs (in particular, it uses no locking)
- software transactional memory, which supports composability in lock-free programming

Parallel linked queues

Parallel linked queue

We present another example of lock-free data structure: an implementation of a linked queue that support parallel access.

A queue data structure offers obvious opportunities for parallelization – because insertion and removal of nodes occurs at two opposite ends of a linked structure. At the same time, it requires to carefully consider the interleaving of operations, and to take measures to prevent modifications that lead to inconsistent states.

We will use regular Java syntax, without emphasizing opportunities for object-oriented abstraction and encapsulation, so as to have a different presentation style, complementary to the one adopted for linked sets.

The interface of a queue

We use linked lists to implement a lock-free queue data structures with interface:

```
interface Queue<T>
{
   // add ‘item’ to back of queue
   void enqueue(T item);

   // remove and return item in front of the queue
   // raise EmptyException if queue is empty
   T dequeue() throws EmptyException;
}
```

Atomic references

To implement data structures that are correct under concurrent access without using any locks we need to rely on synchronization primitives more powerful than just reading and writing shared variables.

We are going to use a variant of the compare-and-set operation.

```
class AtomicReference<V> {

  V get();                        // current reference
  void set(V newRef);             // set reference to newRef

  // if reference == expectRef, set to newRef and return true
  // otherwise, do not change reference and return false
  boolean compareAndSet(V expectRef, V newRef);
}
```

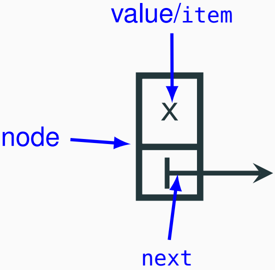
Nodes

The underlying implementations of queues use singly-linked lists, which are made of chains of nodes. Every node:

- stores an `item` – its value
- points to the `next` node in the chain

To build a lock-free implementation, `next` is a reference that supports compare-and-set operations (thus, need not be **`volatile`**).

```
class QNode<T>
{ // value of node
  T value;
  // next node in chain
  AtomicReference<QNode<T>> next;
  QNode(T value)
  { this.value = value;
    next = new AtomicReference<>(null); }
}
```



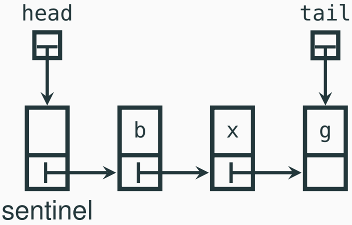
Queues as chains of nodes

A list with a pair of head and tail references implements a queue:

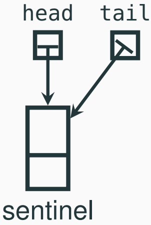
- a sentinel node points to the first element to be dequeued,
- the queue is empty iff the sentinel points to **`null`**,
- `head` points to the sentinel (front of queue),
- `tail` points to the latest enqueued element (back of queue), or the sentinel if the queue is empty.

The sentinel (also called “dummy node”) ensures that `head` and `tail` are never **`null`**.

A non-empty queue:



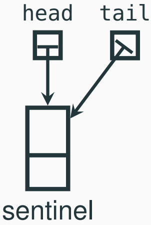
An empty queue:



Head, tail, and empty queue

```
class LockFreeQueue<T> implements Queue<T>
{
  // access to front and back of queue
  protected AtomicReference<QNode<T>> head, tail;

  // empty queue
  public LockFreeQueue() {
      // value of sentinel does not matter
    QNode<T> sentinel = new QNode<>();
    head = new AtomicReference<>(sentinel);
    tail = new AtomicReference<>(sentinel);
  }
```



Enqueue operation

The method `enqueue` adds a new node to the back of a queue – where `tail` points. It requires two updates that modify the linked structure:

1. update last: make the last node in the queue point to the new node,
2. update tail: make `tail` point to the new node.

Each update is individually atomic (it uses compare-and-set), but another thread may interfere between the two updates:

- repeat update last until success;
- try update tail once;
- the implementation should be able to deal with a "half finished" enqueue operation (tail not updated yet), and finish the job – this technique is called helping.

Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

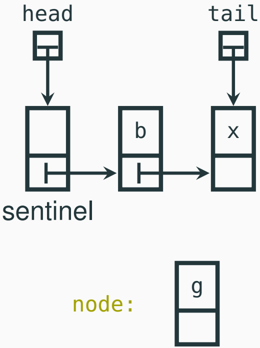
fails only if another thread moves `tail`

helps another thread move `tail`

Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

If `tail` points to actual last:



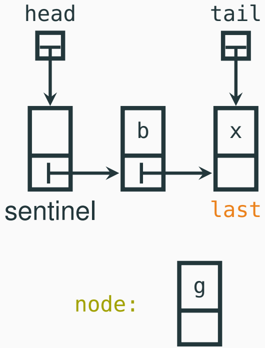
fails only if another thread moves `tail`

helps another thread move `tail`

Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

If `tail` points to actual last:



fails only if another thread moves `tail`

helps another thread move `tail`

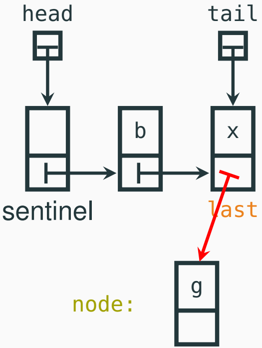
Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

fails only if another thread moves `tail`

helps another thread move `tail`

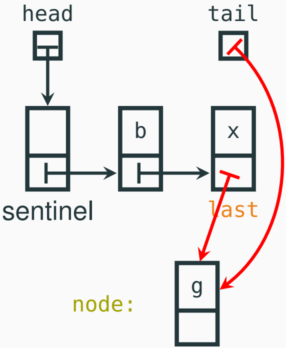
If `tail` points to actual last:



Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

If `tail` points to actual last:



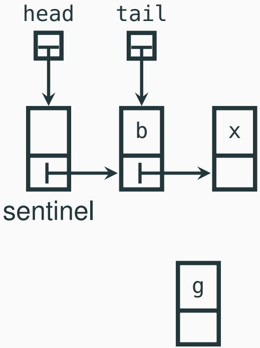
fails only if another thread moves `tail`

helps another thread move `tail`

Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

If `tail` points to old last:



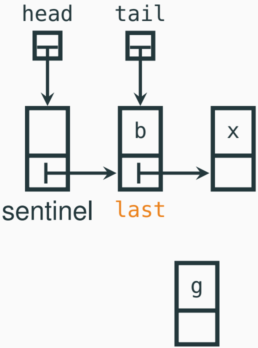
fails only if another thread moves `tail`

helps another thread move `tail`

Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

If `tail` points to old last:



fails only if another thread moves `tail`

helps another thread move `tail`

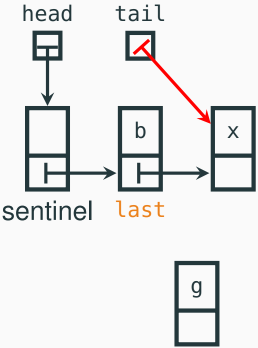
Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

fails only if another thread moves `tail`

helps another thread move `tail`

If `tail` points to old last:



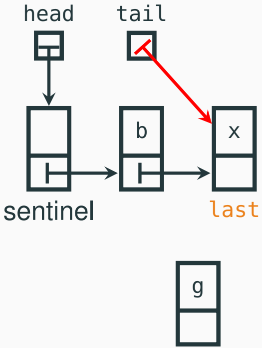
Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

fails only if another thread moves `tail`

helps another thread move `tail`

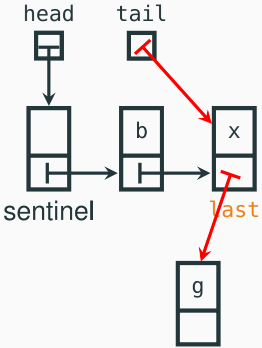
If `tail` points to old last:



Method `enqueue`

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

If `tail` points to old last:



fails only if another thread moves `tail`

helps another thread move `tail`

Dequeue operation

The method `dequeue` removes the node at the head of a queue – where the sentinel points. Unlike enqueue, dequeueing only requires one update to the linked structure:

1. update head: make `head` point the node previously pointed to by the sentinel; the same node becomes the new sentinel and is also returned.

The update is atomic (it uses compare-and-set), but other threads may be updating the head concurrently:

- repeat update head until success,
- if you detect a "half finished" enqueue operation – with the tail pointing to the sentinel about to be removed – help by moving the tail forward.

Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn't point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

Method `dequeue`

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn't point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

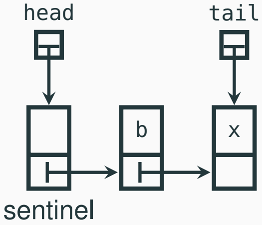
must help move `tail` before updating `head`

must move `head`: no other thread can help

Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

If `tail` needs no update:



must help move `tail` before updating `head`

must move `head`: no other thread can help

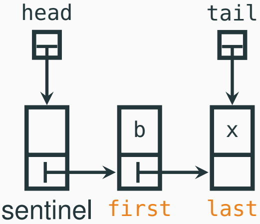
Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
           first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

must help move `tail` before updating `head`

must move `head`: no other thread can help

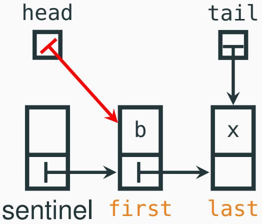
If `tail` needs no update:



Method `dequeue`

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

If `tail` needs no update:



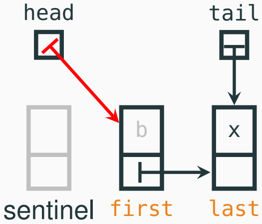
must help move `tail` before updating `head`

must move `head`: no other thread can help

Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

If `tail` needs no update:



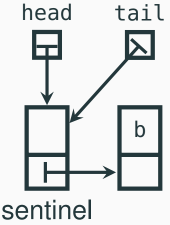
must help move `tail` before updating `head`

must move `head`: no other thread can help

Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

If `tail` needs update:



must help move `tail` before updating `head`

must move `head`: no other thread can help

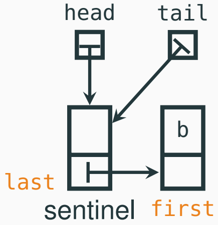
Method `dequeue`

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn't point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

must help move `tail` before updating `head`

must move `head`: no other thread can help

If `tail` needs update:



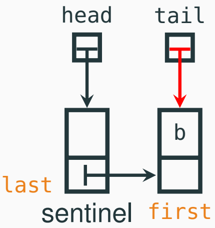
Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

must help move `tail` before updating `head`

must move `head`: no other thread can help

If `tail` needs update:



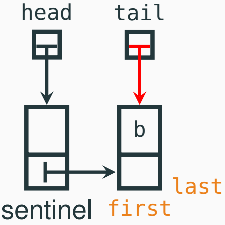
Method dequeue

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
           first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

must help move `tail` before updating `head`

must move `head`: no other thread can help

If `tail` needs update:



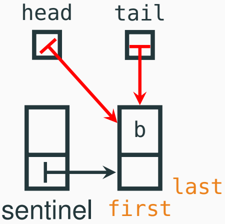
Method `dequeue`

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
           first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

must help move `tail` before updating `head`

must move `head`: no other thread can help

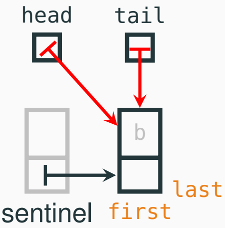
If `tail` needs update:



Method `dequeue`

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn’t point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

If `tail` needs update:

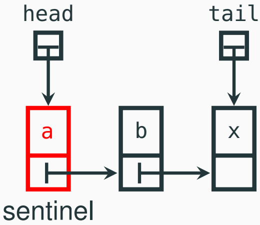


must help move `tail` before updating `head`

must move `head`: no other thread can help

Garbage collection saves the day

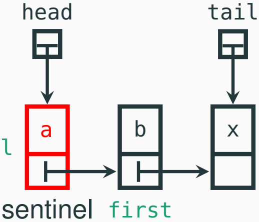
If we were using a language without garbage collection – where objects can be recycled – the following problem could occur:



Garbage collection saves the day

If we were using a language without garbage collection – where objects can be recycled – the following problem could occur:

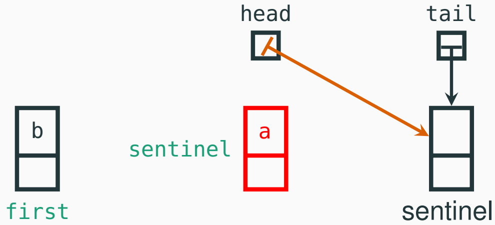
1. *t* is about to CAS `head` from sentinel node `a` to node `b`:
   `head.compareAndSet(sentinel,first)`



Garbage collection saves the day

If we were using a language without garbage collection – where objects can be recycled – the following problem could occur:

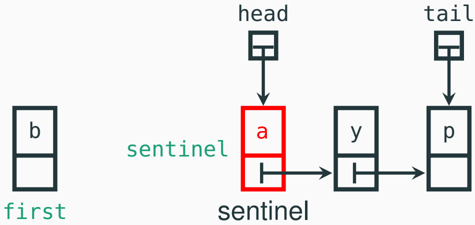
1. *t* is about to CAS `head` from sentinel node `a` to node `b`:
   `head.compareAndSet(sentinel,first)`
2. *u* dequeues `b` and `x`



Garbage collection saves the day

If we were using a language without garbage collection – where objects can be recycled – the following problem could occur:

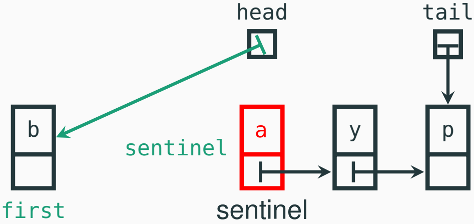
1. *t* is about to CAS `head` from sentinel node `a` to node `b`:
   `head.compareAndSet(sentinel,first)`
2. *u* dequeues `b` and `x`
3. *u* enqueues `a` again (the very same node), enqueues `y`, enqueues `p`, and then dequeues `a` again, so that the same node `a` becomes the sentinel again



Garbage collection saves the day

If we were using a language without garbage collection – where objects can be recycled – the following problem could occur:

1. *t* is about to CAS `head` from sentinel node `a` to node `b`:
   `head.compareAndSet(sentinel,first)`
2. *u* dequeues `b` and `x`
3. *u* enqueues `a` again (the very same node), enqueues `y`, enqueues `p`, and then dequeues `a` again, so that the same node `a` becomes the sentinel again
4. *t* completes CAS successfully (`head` still points to *t*'s local reference `sentinel`), but node `b` is now disconnected!



The ABA problem

The problem we have just seen is known as the ABA problem. It cannot occur in languages that, like Java, feature automatic memory management (garbage collection).

Our `LockFreeQueue` implementation relies on garbage collection for correctness: a thread creates a fresh node (using **`new`**) whenever it enqueues a value, which is guaranteed to have a reference that was not in use before.

Software transactional memory

Transactions

The notion of transaction, which comes from database research, supports a general approach to lock-free programming:

> A transaction is a sequence of steps executed by a single thread, which are executed atomically.

A transaction may:

- succeed: all changes made by the transaction are committed to shared memory; they appear as if they happened instantaneously
- fail: the partial changes are rolled back, and the shared memory is in the same state it would be if the transaction had never executed

Therefore, a transaction either executes completely and successfully, or it does not have any effect at all.

Programming with transactions

The notion of transaction supports a general approach to lock-free programming:

- define a transaction for every access to shared memory
- if the transaction succeeds, there was no interference
- if the transaction failed, retry until it succeeds

Imagine we have a syntactic means of defining transaction code:

```
atomic {
  // transaction code
}
// retry until success
```

```
% execute Function(Arguments)
% as a transaction (retry until success)
atomic(Function, Arguments)
```

Transactions may also support invoking retry and rollback explicitly.

(Note that `atomic` is not a valid keyword in Java or Erlang: we use it for illustration purposes, and later we sketch how it could be implemented as a function in Erlang.)

Transactions are better than locks

Transactional atomic blocks look superficially similar to monitor’s methods with implicit locking, but they are in fact much more flexible:

- since transactions do not lock, there is no locking overhead
- parallelism is achieved without risks of race conditions
- since no locks are acquired, there is no problem of deadlocks (although starvation may still occur if there is a lot of contention)
- transactions compose easily

```
class Account {
  void deposit(int amount)
    { atomic {
      balance += amount; }}
  void withdraw(int amount)
    { atomic {
      balance -= amount; }}
}
```

```
class TransferAccount extends Account {
 // transfer from ‘this’ to ‘other’
 void transfer(int amount,
               Account other)
  { atomic {
      this.withdraw(amount);
      other.deposit(amount); }}
}
```

no locking, so no deadlock is possible!

Transactional memory

A transactional memory is a shared memory storage that supports atomic updates of multiple memory locations.

Implementations of transactional memory can be based on hardware or software:

- hardware transactional memory relies on support at the level of instruction sets (Herlihy & Moss, 1993),
- software transactional memory is implemented as a library or language extension (Shavit & Touitou, 1995).

Software transactional memory implementations are available for several mainstream languages (including Java, Haskell, and Erlang). This is still an active research topic – quality varies!

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and version:

```
-record(var, {name, version = 0, value = undefined}).
```

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and version:

```
-record(var, {name, version = 0, value = undefined}).
```

Clients use an STM as follows:

- at the beginning of a transaction, check out a copy of all variables involved in the transaction;
- execute the transaction, which modifies the values of the local copies of the variables;
- at the end of a transaction, try to commit all local copies of the variables.

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and version:

```
-record(var, {name, version = 0, value = undefined}).
```

The STM's commit operation ensures atomicity:

- if all committed variables have the same version number as the corresponding variables in the STM, there were no changes to the memory during the transaction: the transaction succeeds;
- if some committed variable has a different version number from the corresponding variable in the STM, there was some change to the memory during the transaction: the transaction fails.

The counter example – with software transactional memory

```
int cnt;
```

thread `t`

```
int c;
atomic {
 c = cnt;
 cnt = c + 1;
}
```

thread `u`

```
int c;
atomic {
 c = cnt;
 cnt = c + 1;
}
```

The `atomic` translates into a loop that repeats until the transaction succeeds:

1. check out (`pull`) the current value of `cnt`
2. increment the local variable `c`
3. try to commit (`push`) the new value of `cnt`
4. if `cnt` has changed version when trying to commit, repeat the loop

The counter example: a successful run

$\langle$name: cnt, version: x, value: $y$$\rangle$

thread t

```
int c;
do {
 // check out cnt
● c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
c_t : $\bot$	c_u : $\bot$	cnt : 0_3

The counter example: a successful run

$\langle$`name: cnt`, `version:` x, `value:` $y\rangle$

thread t

```
 int c;
 do {
  // check out cnt
  c = pull(cnt);
● c = c + 1;
 } while (!push(cnt, c));
   // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 0_3$	$c_u : \bot$	$cnt : 0_3$

The counter example: a successful run

$\langle \texttt{name: cnt}, \texttt{version:} x, \texttt{value:} y \rangle$

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
●} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 1_3$	$c_u : \bot$	$\texttt{cnt} : 0_3$

The counter example: a successful run

⟨name: cnt, version: x, value: y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
success	c_u : ⊥	cnt : 1_4

The counter example: a successful run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;     ●
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$\mathtt{c}_{\mathtt{u}} : 1_4$	$\mathtt{cnt} : 1_4$

The counter example: a successful run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c)); ●
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	c_u : 2_4	cnt : 1_4

The counter example: a successful run

⟨`name: cnt`, `version:` x, `value:` y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t's LOCAL	u's LOCAL	STM
done	success	`cnt`: 2_5

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t

```
 int c;
 do {
  // check out cnt
● c = pull(cnt);
  c = c + 1;
 } while (!push(cnt, c));
   // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : \bot$	$c_u : \bot$	cnt : 0_3

The counter example: a retry run

$\langle$`name: cnt`, `version:`x, `value:`$y\rangle$

thread t

```
 int c;
 do {
  // check out cnt
  c = pull(cnt);
● c = c + 1;
 } while (!push(cnt, c));
   // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 0_3$	$c_u : \bot$	$\texttt{cnt} : 0_3$

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

thread t

```
 int c;
 do {
  // check out cnt
  c = pull(cnt);
● c = c + 1;
 } while (!push(cnt, c));
   // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;     ●
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 0_3$	$c_u : 0_3$	$\mathtt{cnt} : 0_3$

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
●} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;     ●
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t : 1_3$	$c_u : 0_3$	$\mathtt{cnt} : 0_3$

The counter example: a retry run

$\langle$`name: cnt`, `version:`x, `value:`$y\rangle$

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
●} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c)); ●
  // commit cnt
```

The subscript in a variable's value indicates its version:

T'S LOCAL	U'S LOCAL	STM
$c_t : 1_3$	$c_u : 1_3$	$\mathtt{cnt} : 0_3$

The counter example: a retry run

$\langle$name: cnt, version: x, value: $y\rangle$

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c)); ●
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
success	c_u : 1_3	cnt : 1_4

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	fail	`cnt`: 1_4

The counter example: a retry run

$\langle$`name: cnt`, `version:` x, `value:` $y\rangle$

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	retry	`cnt`: 1_4

The counter example: a retry run

$\langle \texttt{name: cnt}, \texttt{version:} x, \texttt{value:} y \rangle$

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt); ●
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: \bot$	cnt: 1_4

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;     ●
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u : 1_4$	$cnt : 1_4$

The counter example: a retry run

⟨name: cnt, version: x, value: y⟩

---

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c)); ●
  // commit cnt
```

The subscript in a variable’s value indicates its version:

t’S LOCAL	u’S LOCAL	STM
done	$\mathtt{c}_\mathtt{u} : 2_4$	$\mathtt{cnt} : 1_4$

The counter example: a retry run

⟨`name: cnt`, `version:`x, `value:`y⟩

thread t	thread u

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	success	`cnt`: 2_5

The counter example: a retry run

$\langle$`name: cnt`, `version:`x, `value:`$y$$\rangle$

thread t

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

thread u

```
int c;
do {
 // check out cnt
 c = pull(cnt);
 c = c + 1;
} while (!push(cnt, c));
  // commit cnt
```

The subscript in a variable's value indicates its version:

	t'S LOCAL	u'S LOCAL	STM
	done	done	cnt: 2_5

STM in Erlang

An STM is a server that provides the following main operations:

- `pull(Name)`: check out a copy of variable with name `Name`
- `push(Vars)`: commit all variables in `Vars`; return `fail` if unsuccessful

Clients read and write local copies of variables using:

- `read(Var)`: get value of variable `Var`
- `write(Var, Value)`: set value of variable `Var` to `Value`

We base the STM implementation on the `gserver` generic server implementation we presented in a previous class.

STM: operations

```
create(Tm, Name, Value) ->
  gserver:request(Tm, {create, Name, Value}).
drop(Tm, Name) ->
  gserver:request(Tm, {drop, Name}).
pull(Tm, Name) ->
  gserver:request(Tm, {pull, Name}).
push(Tm, Vars) when is_list(Vars) ->
  gserver:request(Tm, {push, Vars});
read(#var{value = Value}) ->
  Value.
write(Var = #var{}, Value) ->
  Var#var{value = Value}.
```

STM: server handlers

The storage is a dictionary associating variable names to variables; it is the essential part of the server state.

```
stm(Storage, {pull, Name}) ->
  case dict:is_key(Name, Storage) of
    true ->
      {reply, Storage,
       dict:fetch(Name, Storage)};
    false ->
      {reply, Storage, not_found}
   end;
```

```
stm(Storage, {push, Vars}) ->
  case try_push(Vars, Storage) of
    {success, NewStorage} ->
      {reply, NewStorage, success};
    fail ->
      {reply, Storage, fail}
  end.
```

STM: try to push

The helper function `try_push` determines if any variable to be committed has a different version from the corresponding one in the STM.

```
try_push([], Storage) ->
  {success, Storage};
try_push([Var = #var{name = Name, version = Version} | Vars],
         Storage) ->
  case dict:find(Name, Storage) of
    {ok, #var{version = Version}} ->
      try_push(Vars,
               dict:store(Name,
                          Var#var{version = Version + 1},
                          Storage));
    _ -> fail
  end.
```

Using the Erlang STM

Using the STM to create atomic functions is quite straightforward. For example, here are pop and push atomic operations for a list:

```
% pop head element from ‘Name’
qpop(Tm, Name) ->
  Queue = pull(Tm, Name),
  [H|T] = read(Queue),
  NewQueue = write(Queue, T),
  case push(Tm, NewQueue) of
    % push failed: retry!
    fail -> qpop(Tm, Name);
    % push successful: return head
    _ -> H
  end.
```

```
% push ‘Value’ to back of ‘Name’
qpush(Tm, Name, Value) ->
  Queue = pull(Tm, Name),
  Vals = read(Queue),
  NewQueue = write(Queue,
                   Vals ++ [Value]),
  case push(Tm, NewQueue) of
    % push failed: retry!
    fail -> qpush(Tm, Name, Value);
    % push successful: return ok
    _ -> ok
  end.
```

Composable transactions?

The simple implementation of STM we have outlined does not support easily composing transactions:

```
% pop from Queue1 and push to Queue2
qtransfer(Tm, Queue1, Queue2) ->
  Value = qpop(Tm, Queue1),   % another process may interfere!
  qpush(Tm, Queue2, Value).
```

To implement composability, we need to keep track of pending transactions and defer commits until all nested transactions have completed.

See the course's website for an example implementation:

```
% atomically execute Function on arguments Args
atomic(Tm, Function, Args) -> todo.
```

These slides' license