Dual Data Structures

Michael L. Scott

www.cs.rochester.edu/research/synchronization/

jug.ru Hydra Conference
July 2019

Joint work with William N. Scherer, Doug Lea, and Joseph Izraelevitz

The University of Rochester

- Small private research university
- 6400 undergraduates
- 4800 graduate students
- Set on the Genesee River in Western New York State, near the south shore of Lake Ontario
- 250km by road from Toronto; 590km from New York City

The Computer Science Dept.

- Founded in 1974
- 20 tenure-track faculty; 70 Ph.D. students
- Specializing in AI, theory, HCI, and parallel and distributed systems
- Among the best small departments in the US

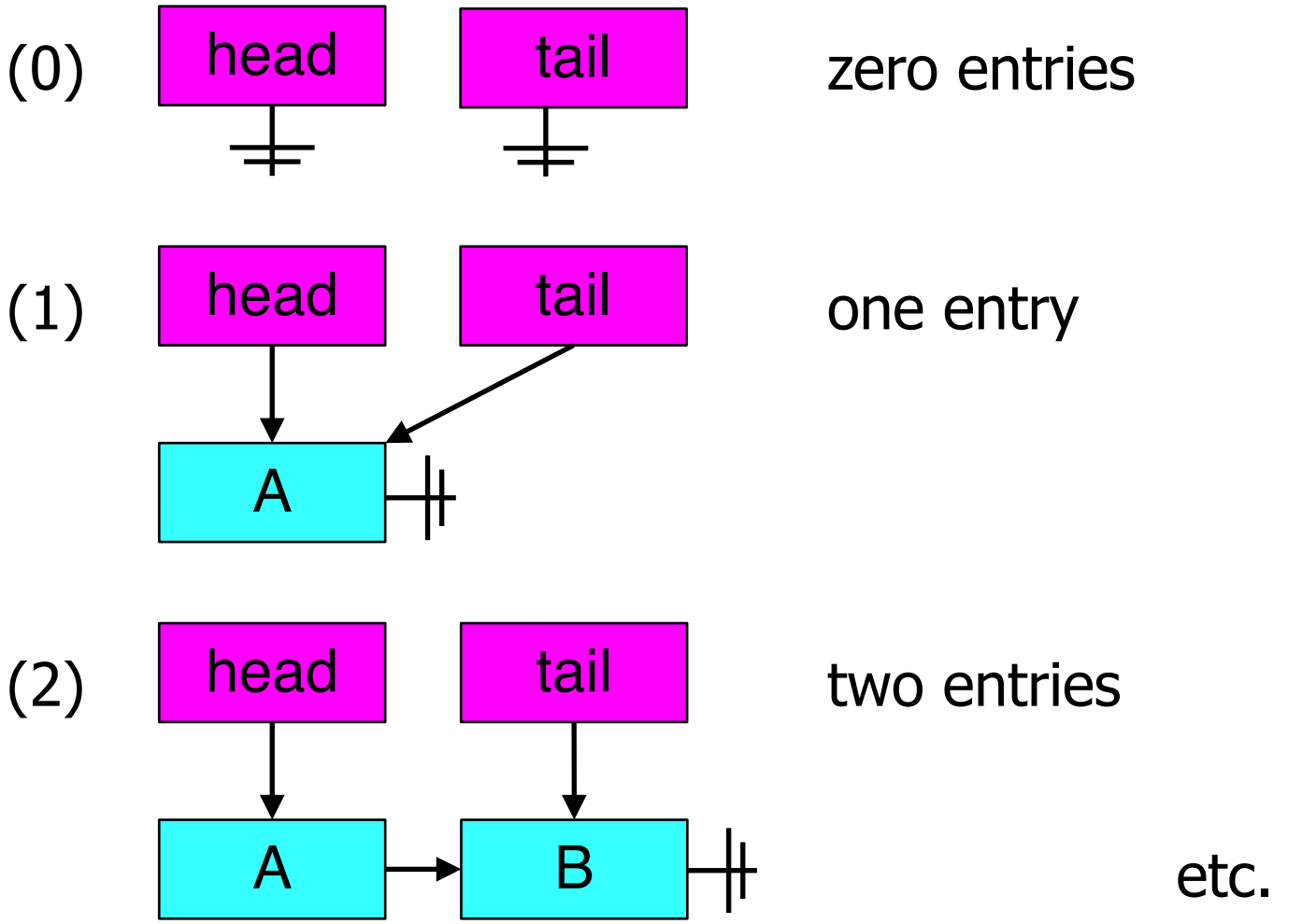
The multicore revolution

- Uniprocessor speed improvements stopped in 2004.
- Since then, all but the most basic processors have had multiple cores on chip.
- Any program that wants to use the full power of the chip must be written with multiple *threads*, which cooperate like a team of people with a common goal.

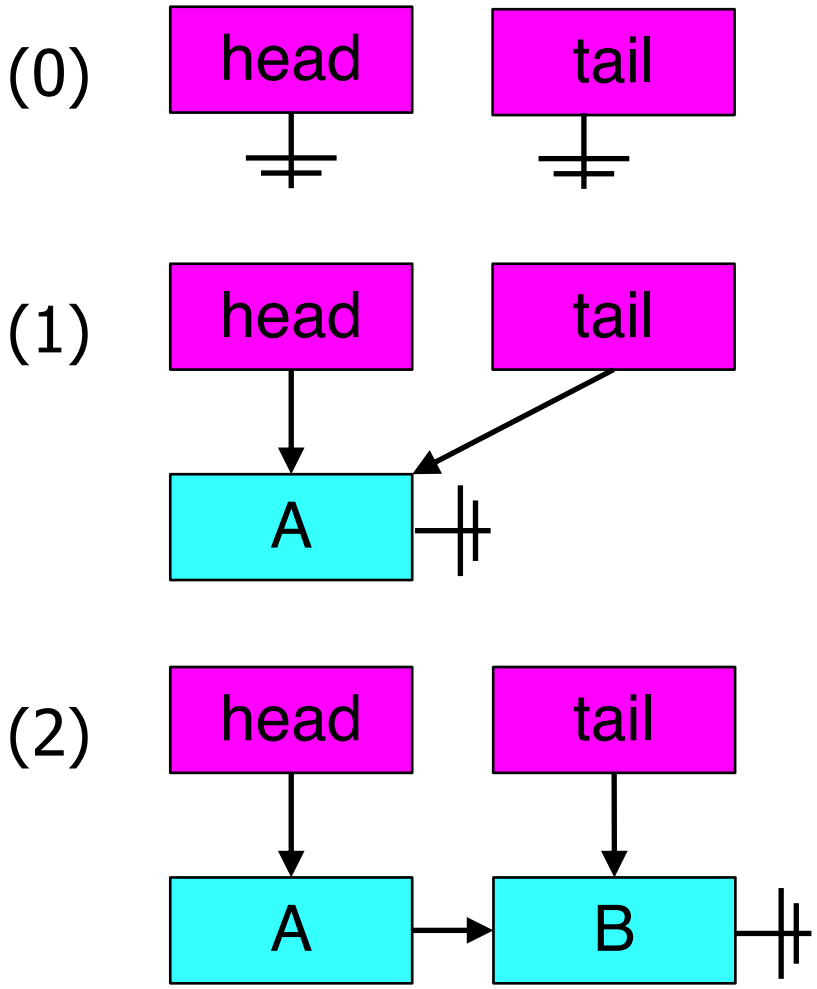
Shared data structures

- Threads interact by calling methods of shared data structures — stacks, queues, linked lists, hash tables, skip lists, many kinds of trees, and more.
- I'll use queues as the example in this talk.
  - Commonly used to pass work from threads in one *stage* of a program to threads in the next stage — like work on a factory assembly line

A typical sequential queue



A typical sequential queue



- Enqueue transforms state (0) to state (1), state (1) to state (2), etc., by allocating a new node and linking it in.
- Dequeue will unlink the head node, if any, and return it.
- Each will read and write multiple locations.

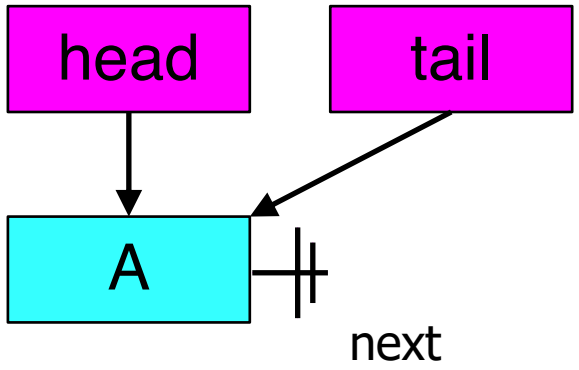
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



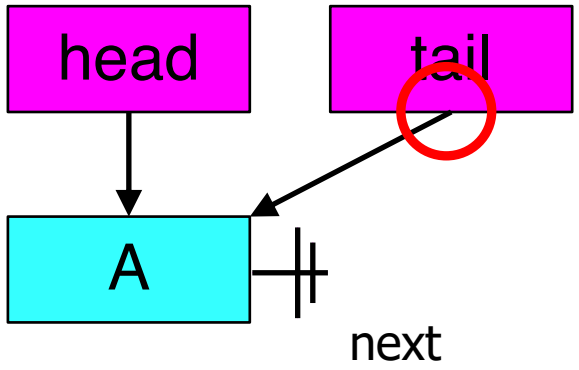
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



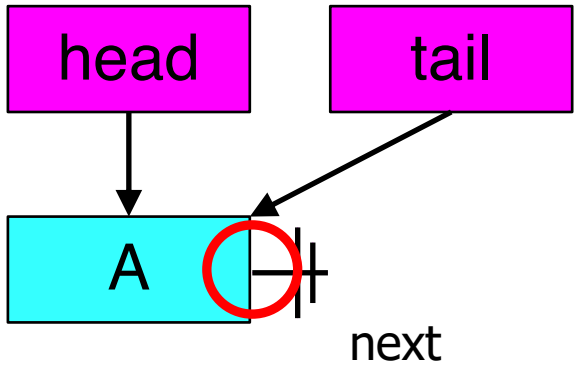
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



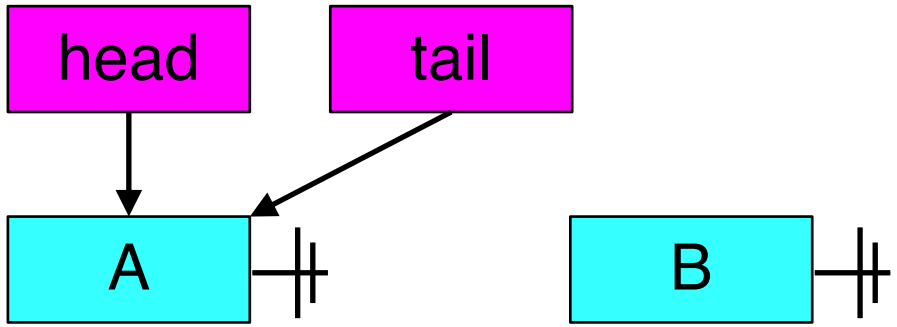
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



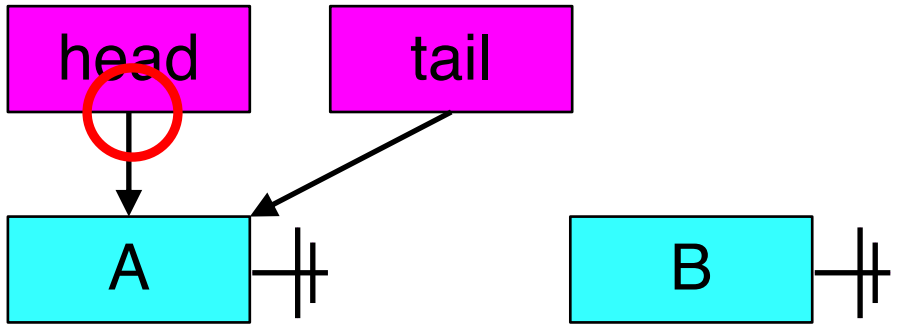
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



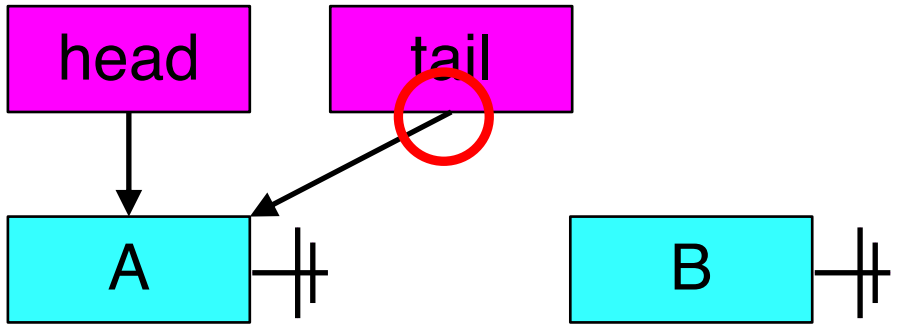
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



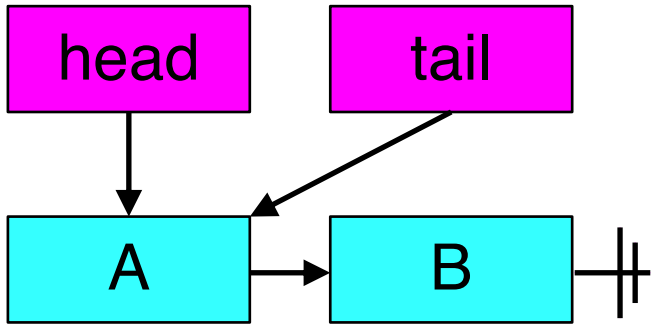
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



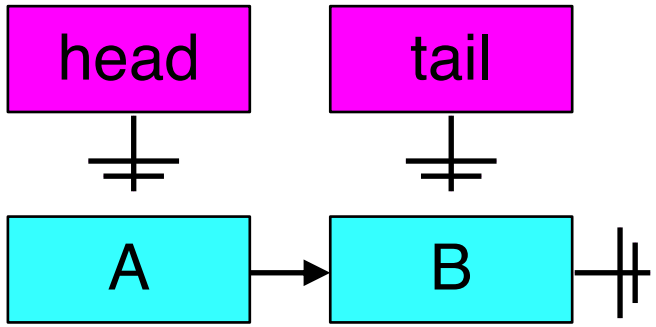
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



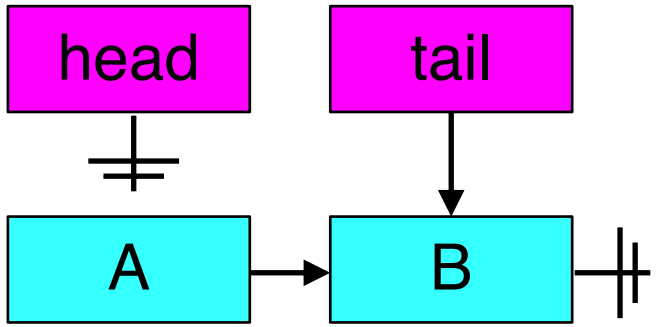
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



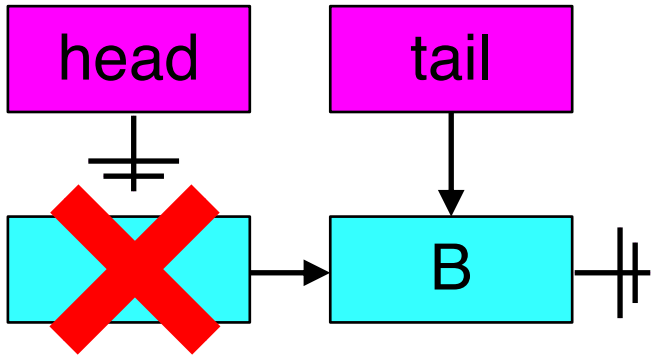
Concurrent updates are unsafe

Thread 1: dequeue A

- reads head
- reads tail (sees single entry)
- sets head and tail to null
- deletes A

Thread 2: enqueue B

- reads tail
- assumes A.next is null
- creates new node B
- sets A.next to B
- sets tail to B



Atomicity

- Correct operations have to appear to happen “all at once”
- Easiest way to do that is with locks or, in Java, synchronized blocks or methods:

```
synchronized(Q) {
    perform operation
}
```

- Performs badly if scheduler preempts a thread that holds a lock

Nonblocking data structures

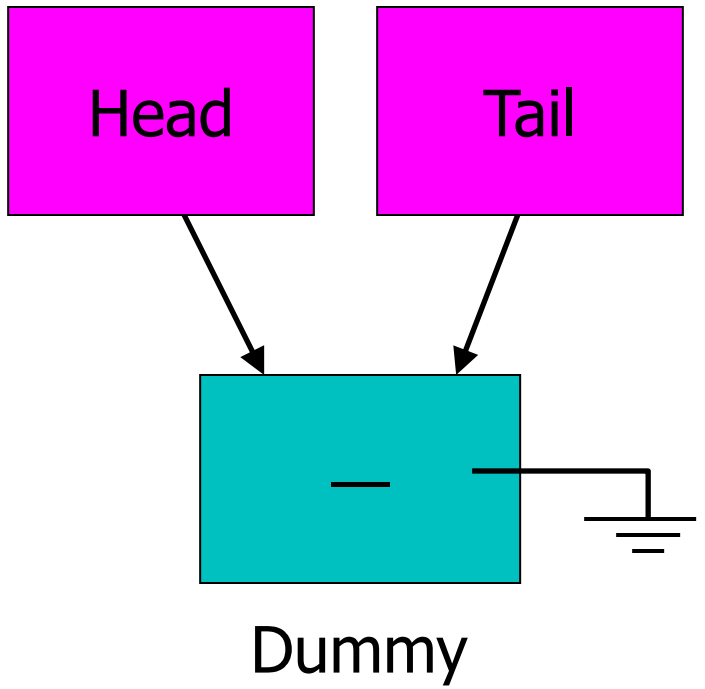
- Never use locks/synchronized blocks; never block progress when a thread is preempted
- Operation seems to happen instantaneously at some "linearizing" instruction — often *compare-and-swap,* expressed in Java as r.compareAndSet(e, n)
  - takes as argument a reference *r*, an expected value e, and a new value *n*
  - replaces contents of *r* with *n* if and only if previous contents was *e*
  - supported by hardware; operates atomically

Instantaneously?

- Everything before the linearizing instruction is harmless preparation
  - » Doesn't change the abstract state of the structure
- Everything after the linearizing instruction is merely clean-up
  - » Can be done by any thread
- Hard to write, but versions exist for almost all common data structures
  - » Again, use a queue as an example
- All interleavings are acceptable!

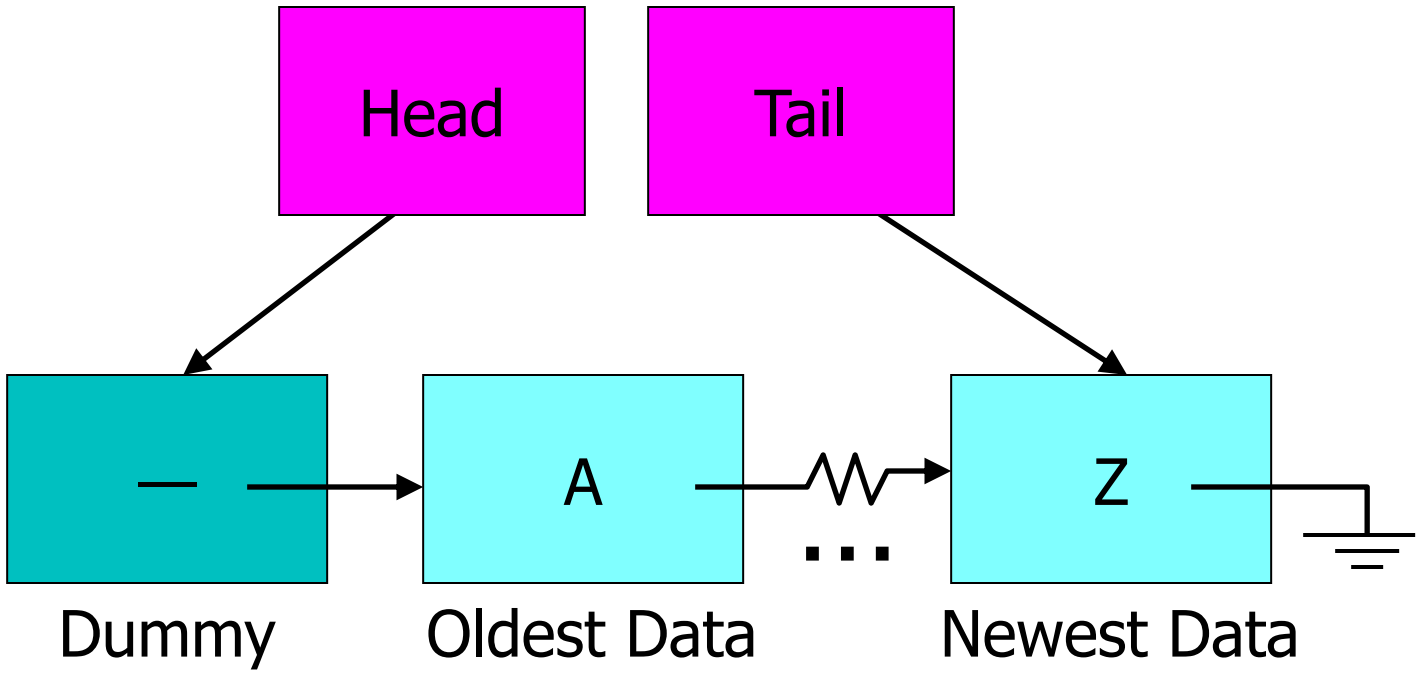
M&S queue

- Empty queue consists of a queue object with head, tail pointers to *dummy* node



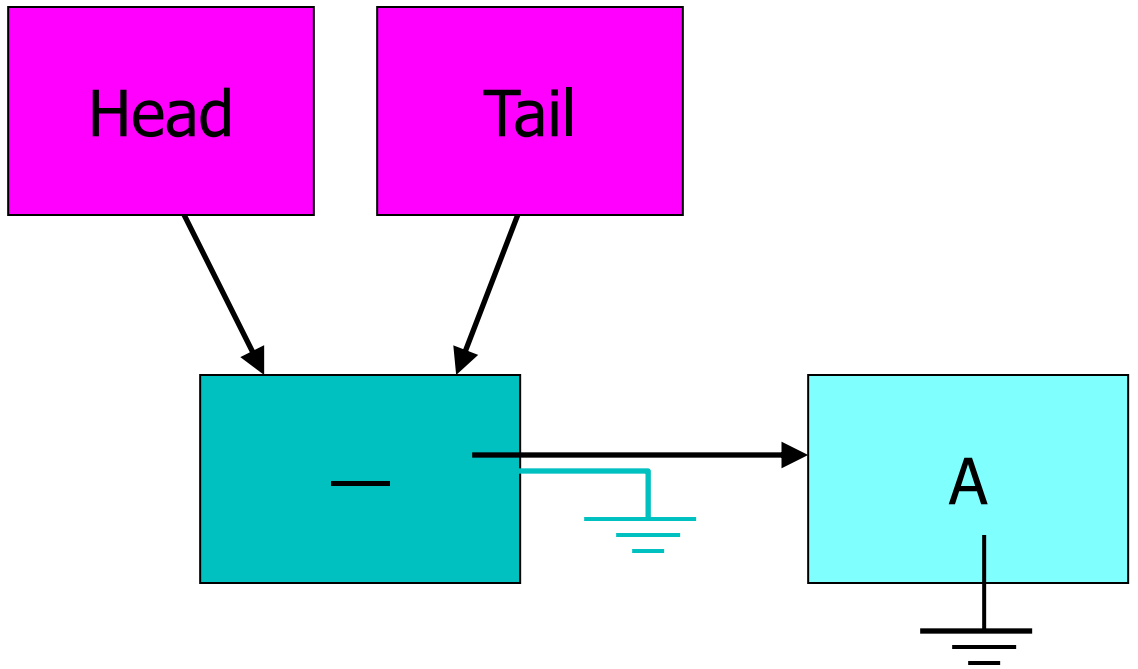
M&S queue with data

- Head of linked list is still a dummy node
- Enqueue adds at tail; dequeue removes at head



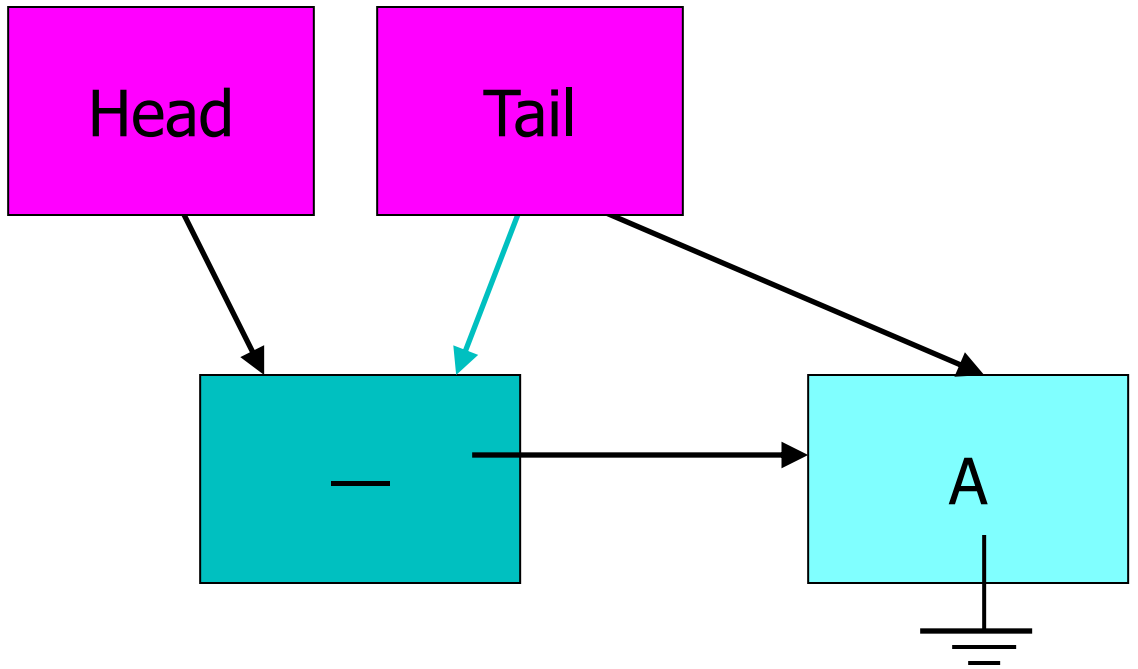
M&S queue: enqueue

1) CAS next pointer of tail node to new node
2) Use CAS to swing tail pointer



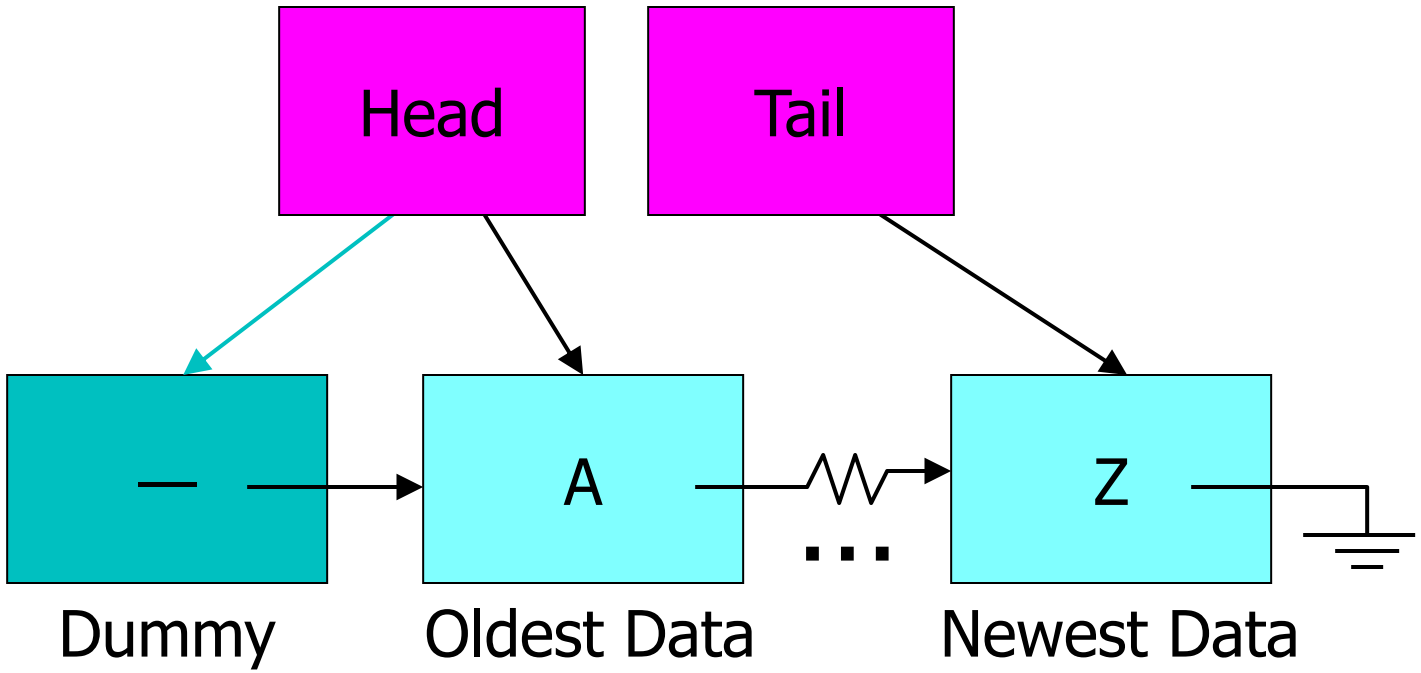
M&S queue: enqueue

1) CAS next pointer of tail node to new node
2) Use CAS to swing tail pointer (any thread can help)



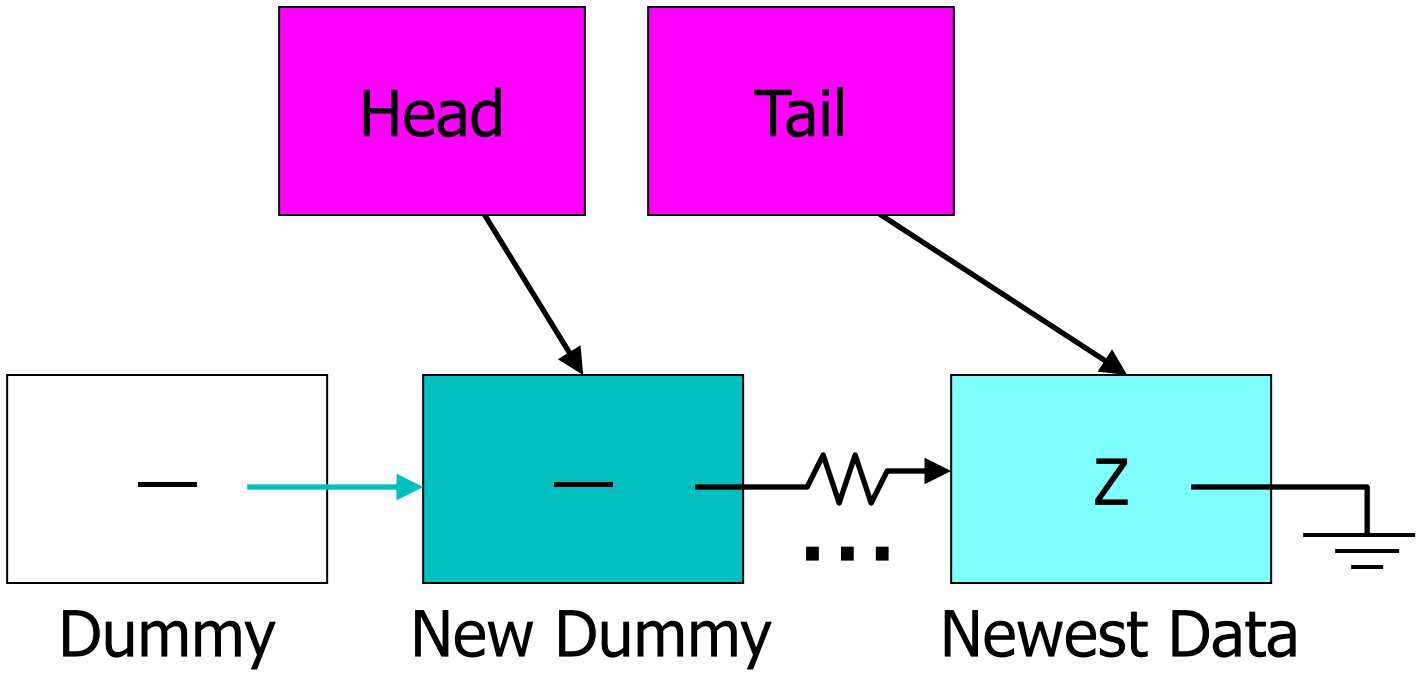
M&S queue: dequeue

1) Read data in dummy's next node
2) CAS head pointer to dummy's next node



M&S queue: dequeue II

3) Discard old dummy node
4) Node from which we read is new dummy



What if the queue is empty?

- With locks, a dequeuing thread can *wait* — tell the scheduler to put it to sleep and release the lock
  - » Some later enqueuing thread, while holding the lock, will tell the scheduler to make the sleeping thread runnable again
- In a nonblocking queue, this hasn't traditionally been possible
  - » Does it even make sense to wait (block?) in a nonblocking structure?

The traditional approach

- Dequeue on an empty queue *fail*s immediately
- Calling thread must *spin*:

```
do {
    t = q.dequeue()
} while (t == ⊥)
```

- This works but with
  - » high contention
  - » no guarantee of fairness (waiting threads can succeed out of order)

Dual data structures

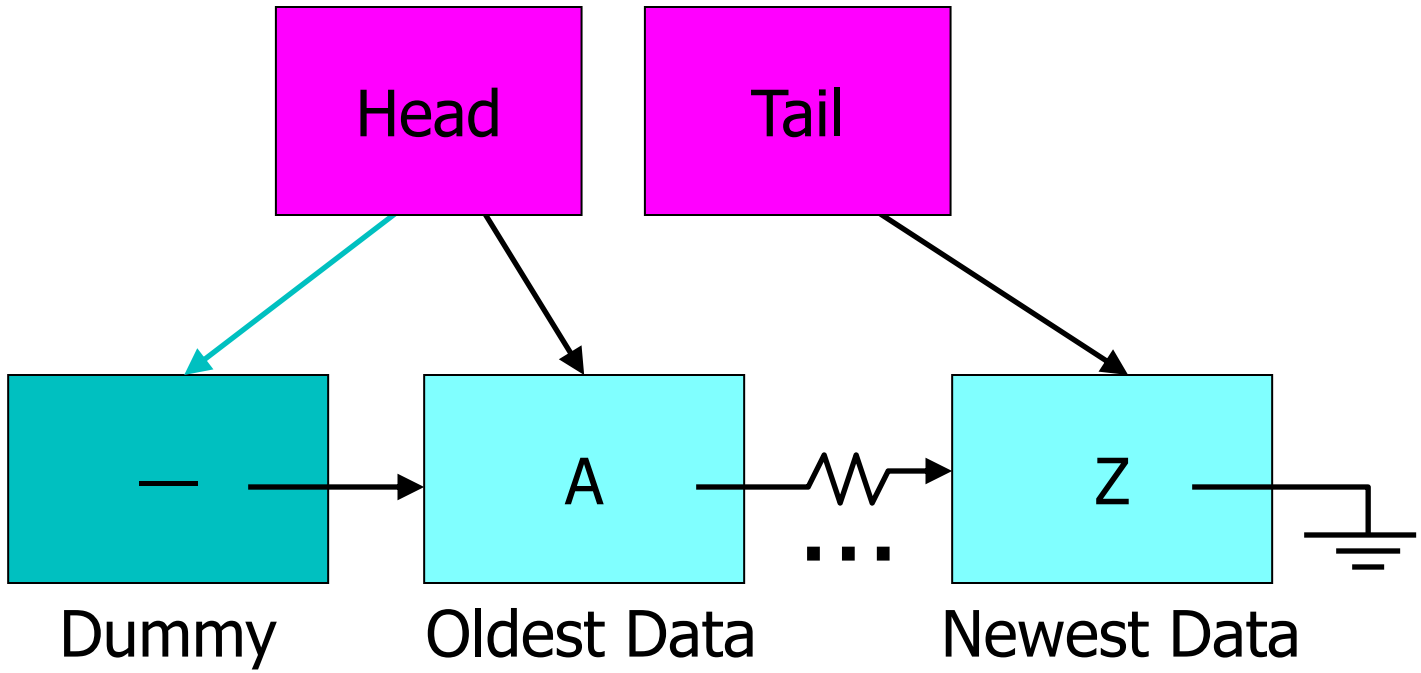
- Data structure holds data or *reservations*
- Dequeuer (in general, consumer) removes data or inserts reservation
- Enqueuer (in general, producer) inserts data or removes and *satisfies* reservation
- Data structure controls which reservation to satisfy — guaranteeing fairness
- Developed dual stack, queue, synchronous variants, exchanger, LCRQ; generic construction
  - » focus here on the queue

The dualqueue

- When trying to dequeue from an empty queue, enqueue a reservation instead
- When enqueuing, satisfy a reservation if present
- Mark pointers to the reservation nodes with a “tag” bit in the pointer
  - Can (mostly) tell queue state from tail pointer
  - Easy in C; requires extra indirection in Java
- Symmetry between enqueues and dequeues
  - Enqueue adds data or removes a reservation
  - Dequeue removes data or adds a reservation

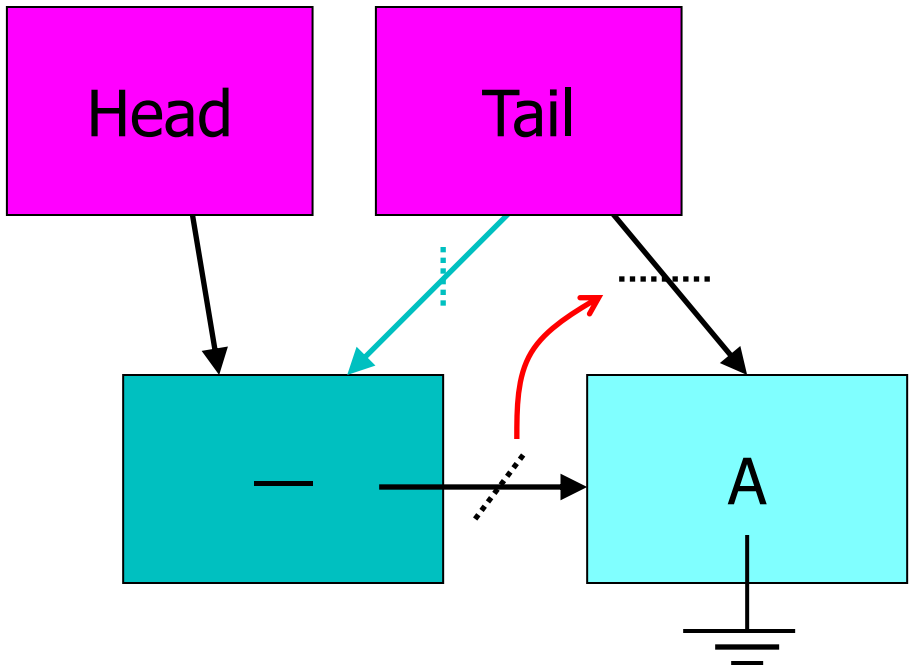
Dualqueue: dequeue

1) Check for queue “empty” or full of reservations
2) If neither, try to dequeue data as before



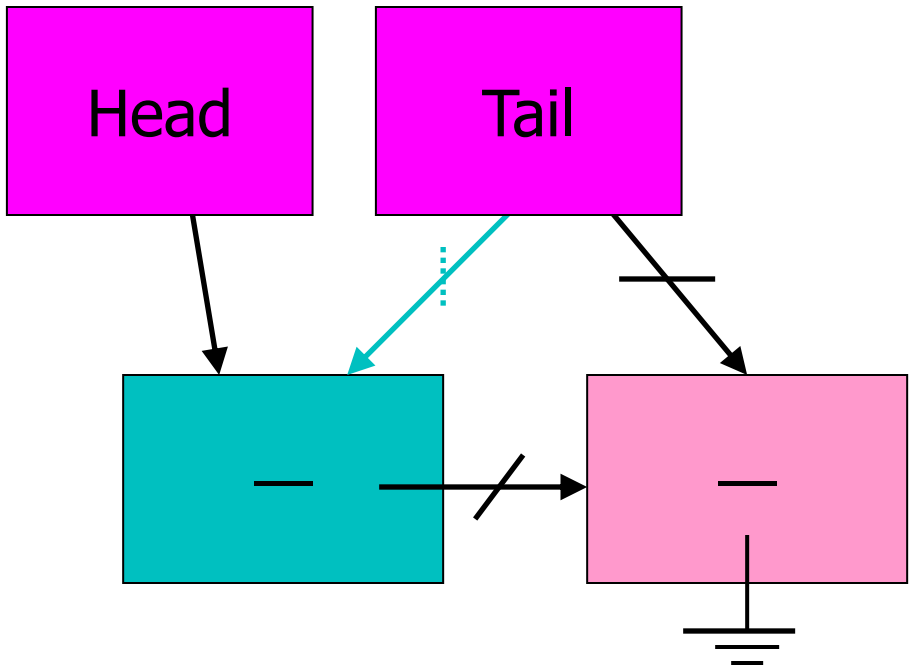
Dualqueue: dequeue II

3) If tail pointer is lagging, swing it and restart. Match tag of tail node's next pointer



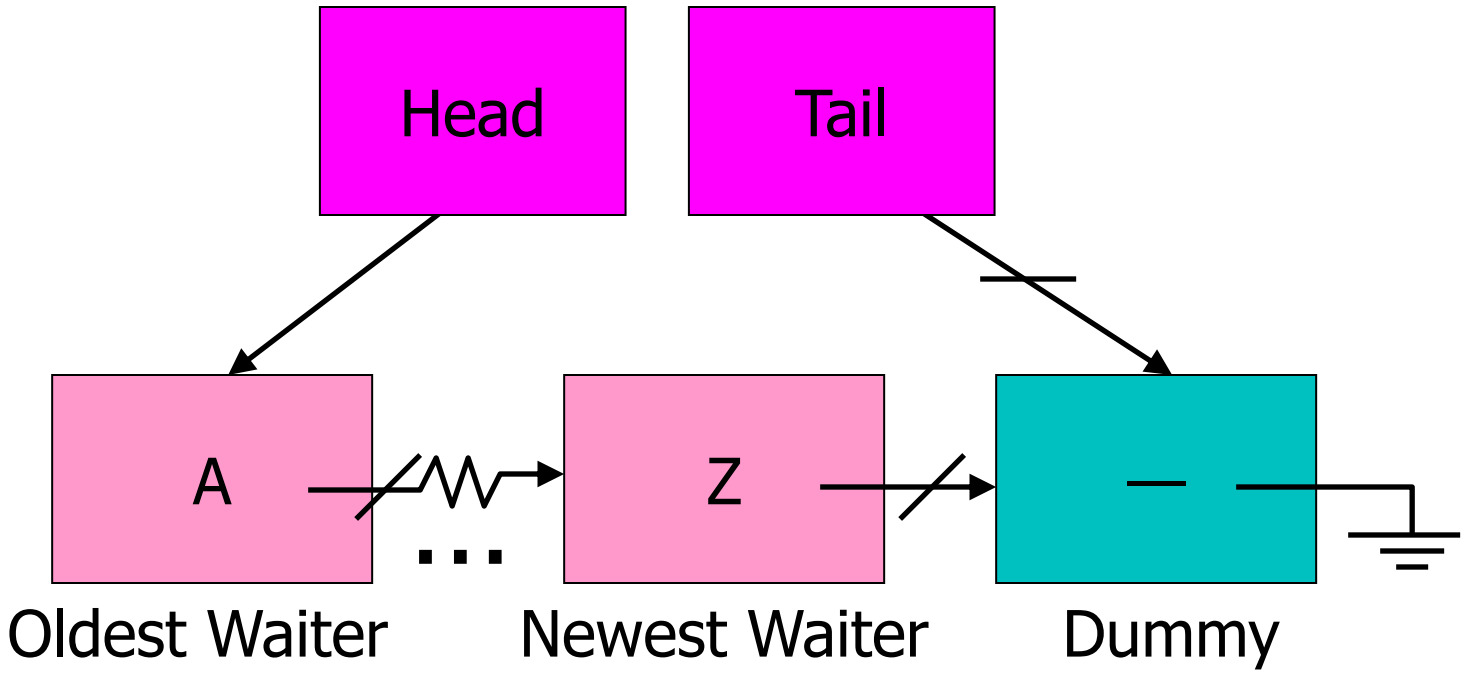
Dualqueue: dequeue III

4) If queue is empty, enqueue a tagged marker node, then swing tail pointer



Dualqueue: dequeue IV

4) Next, spin on the old tail node. Note: when queue holds reservations, dummy node is at *tail* end



Dualqueue: enqueue

1) Read head & tail pointers to see if queue looks empty or has data in it
2) If so, do an enqueue just as in the M&S queue
3) Else, try to satisfy a reservation

Dualqueue: satisfying requests

1) CAS pointer to data node into reservation node, breaking spin

   Alternatively, waiting thread can sleep on a semaphore, to which the awakening thread can post

2) CAS reservation node out of queue (dequeuing thread may help CAS)
3) Dequeuer reads data, frees reservation & data nodes

Synchronous stacks, queues, and exchangers

- Joint work with Doug Lea, chief architect of `java.util.concurrent` libraries
- In synchronous stacks & queues, producer waits for consumer; in exchanger they swap values
- Synchronous dualstack 3x faster than previous
- Synchronous dualqueue 14x faster
- Throughput of `Executor` library increased by 2x in “unfair” mode and 10x in “fair” mode
- Standard part of distribution since Java SE 6

Generic duals

- The dualqueue satisfies reservations in FIFO order; the dualstack in LIFO order
- We can write “quacks” and “steues” that use opposite orders for data and reservations
- More generally, we can pair *any* nonblocking container for data (e.g., a priority queue) with almost any nonblocking container for reservations

Almost any?

- Reservation container must provide two special methods
  - ⟨r, k⟩ = peek()
    returns the highest priority reservation and a *key*
  - s = removeConditional(k)
    removes r if it still has highest priority; returns status
- The ability to peek lets us satisfy reservations in a way that is amenable to *helping*
- We also employ a *handshaking* protocol to coordinate the two containers

How fast?

- Along with the generic construction [TOPC 2016] we also presented dual versions of Morrison & Afek's linked concurrent ring queue (LCRQ), which is based on fetch-and-increment (FAI)
- The resulting C code can sustain over 30M ops/s on an 18-core, 3.6GHz Intel Xeon processor
- That's almost 5x the throughput of the original M&S-based dualqueue
- Difficult to add to Java due to extensive pointer tagging

Conclusions/contributions

- Nonblocking operations really can wait
- Dualism improves the performance of
  - » stacks & queues, synchronous stacks & queues, exchangers, and more
  - » nonblocking and lock-based implementations
- Dualism also offers *fairness*: the data structure chooses which waiting thread to satisfy
- Generic construction allows any container for data to be combined with almost any container for reservations

For more information

- "Nonblocking Concurrent Data Structures with Condition Synchronization." W. N. Scherer III and M. L. Scott. *18th Annual Conf. on Distributed Computing (DISC)*, Oct. 2004.
- "Scalable Synchronous Queues." W. N. Scherer III, D. Lea, and M. L. Scott. *11th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2006; *Communications of the ACM*, May 2009.
- "Generality and Speed in Nonblocking Dual Containers." J. Izraelevitz and M. L. Scott. *ACM Transactions on Parallel Computing*, Mar. 2017.

www.cs.rochester.edu/research/synchronization/

www.cs.rochester.edu/u/scott/