Lecture 13: concurrent queues

queue as mutex, strong FIFO, per-producer FIFO, bounded/unbounded queue, SPSC, MPSC, SPMC, MPMC, total/partial/synchronous API, lock-free queue, ring buffer for bounded queues, concurrent dequeue, work stealing

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l13.pdf

In previous episodes

Threading models[1]

- 1:1 threading (user thread = OS thread, `ThreadFactory`)
- N:1 threading (event loop, `newSingleThreadExecutor`)
- N:M threading (coroutines, `newFixedThreadPool`)
- Lazy/on-demand thread creation (`newCachedThreadPool`)

Heavyweight OS threads affect programming style

- Task pools vs long-running tasks
- Task pools vs blocking methods
- Asynchronous I/O and callback-based programming

User-space scheduling:

- Stackless coroutines (CPS and suspend/resume coloring)
- Stackful coroutines (low-level: scheduling policy, context switch, preemption)

[1] https://en.wikipedia.org/wiki/Thread_(computing)#Threading_models

Lecture plan

1. Queues for mutual exclusion
2. Design space
3. Unbounded lock-free queue
4. Bounded array-based queues
   - SPSC
   - MPSC
   - SPMC
5. Concurrent DEQueue and work stealing
6. Summary

Lecture plan

1. Queues for mutual exclusion
2. Design space
3. Unbounded lock-free queue
4. Bounded array-based queues
   - SPSC
   - MPSC
   - SPMC
5. Concurrent DEQueue and work stealing
6. Summary

Question time

Question: Name queue-based mutexes that you already know

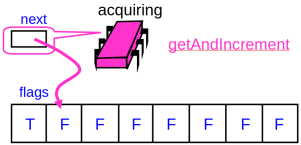
Queue-based mutexes

Queue-based mutexes

- Array-based: Anderson lock

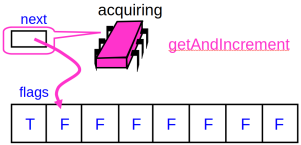
Queue-based mutexes

- Array-based: Anderson lock



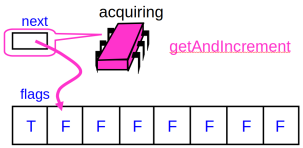
Queue-based mutexes

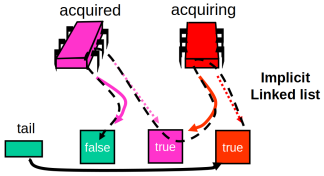
- Array-based: Anderson lock
- LinkedList-based: CLH lock



Queue-based mutexes

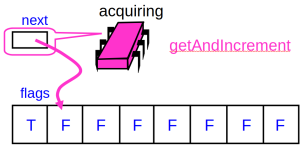
- Array-based: Anderson lock
- LinkedList-based: CLH lock

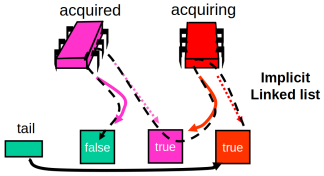




Queue-based mutexes

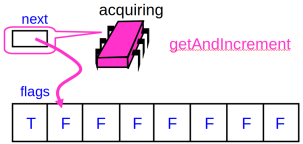
- Array-based: Anderson lock
- LinkedList-based: CLH lock
- LinkedList-based: MCS lock

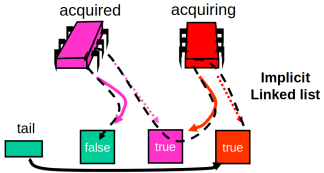


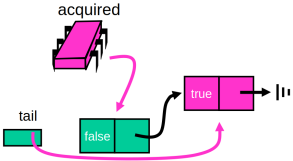


Queue-based mutexes

- Array-based: Anderson lock
- LinkedList-based: CLH lock
- LinkedList-based: MCS lock

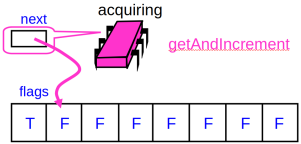


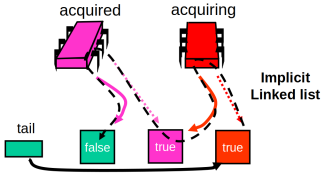


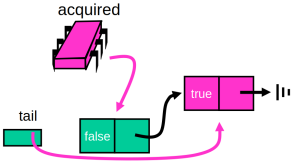


Queue-based mutexes

- Array-based: Anderson lock
- LinkedList-based: CLH lock
- LinkedList-based: MCS lock







Idea: select container (LIFO, FIFO, priority queue) and use it to

- register competitors
- maintain admission policy

FIFO queue for mutual exclusion

```
class FIFOLock implements Lock {
  private Queue q = ...
  void lock() {
    myNode = new Node();
    q.enqueue(myNode);
    while (true) { if (q.peek() == myNode) return; /* it is my turn */ }
  }
  void unlock() {
    prevHead = q.dequeue(); // let other thread go
    assert prevHead == myNode; // could I delete myNode right now?
  }}
```

FIFO queue for mutual exclusion

```
class FIFOLock implements Lock {
  private Queue q = ...
  void lock() {
    myNode = new Node();
    q.enqueue(myNode);
    while (true) { if (q.peek() == myNode) return; /* it is my turn */ }
  }
  void unlock() {
    prevHead = q.dequeue(); // let other thread go
    assert prevHead == myNode; // could I delete myNode right now?
  }}
```

Could be extended with spin-then-park and cancellation[2]

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/LockSupport.html

Question time

Question: How to implement mutex using lock-free LIFO stack?

Queues and mutexes: summary

Thread-safe FIFO container could be transformed into mutex

- admission policy control "out of the box"
- easy to associate data with every entering thread (e.g. thread-local memory for spinning)
- must maintain high level of concurrent consistency
  - Strong FIFO
  - Multiple enqueuers, Single dequeuer

Question time

Question: Name other usage scenarios for concurrent queues

Lecture plan

Queues: where to use

- Concurrency control

Queues: where to use

- Concurrency control
  - `Lock` (`lock` $\approx$ `enqueue`, `unlock` $\approx$ `dequeue`)

Queues: where to use

- Concurrency control
  - `Lock` (`lock` $\approx$ `enqueue`, `unlock` $\approx$ `dequeue`)
  - `ConditionVariable` (`signal` $\approx$ `enqueue`, `await` $\approx$ `dequeue`)

Queues: where to use

- Concurrency control
  - `Lock` (`lock` $\approx$ `enqueue`, `unlock` $\approx$ `dequeue`)
  - `ConditionVariable` (`signal` $\approx$ `enqueue`, `await` $\approx$ `dequeue`)
  - `CountDownLatch` (`countDown` $\approx$ `dequeue`, `await` $\approx$ `isEmpty`)

Queues: where to use

- Concurrency control
  - `Lock` (`lock` $\approx$ `enqueue`, `unlock` $\approx$ `dequeue`)
  - `ConditionVariable` (`signal` $\approx$ `enqueue`, `await` $\approx$ `dequeue`)
  - `CountDownLatch` (`countDown` $\approx$ `dequeue`, `await` $\approx$ `isEmpty`)
  - `Semaphore` (`acquire` $\approx$ `dequeue`, `release` $\approx$ `enqueue`)

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...
- Data transfer

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...
- Data transfer
  - Push-based producer-consumer

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...
- Data transfer
  - Push-based producer-consumer
  - Backpressure/reactive publisher-subscriber

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...
- Data transfer
  - Push-based producer-consumer
  - Backpressure/reactive publisher-subscriber
- Load balancing

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...
- Data transfer
  - Push-based producer-consumer
  - Backpressure/reactive publisher-subscriber
- Load balancing
  - Work arbitrage, work dealing, work stealing

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: where to use

- Concurrency control
  - `Lock` (`lock` ≈ `enqueue`, `unlock` ≈ `dequeue`)
  - `ConditionVariable` (`signal` ≈ `enqueue`, `await` ≈ `dequeue`)
  - `CountDownLatch` (`countDown` ≈ `dequeue`, `await` ≈ `isEmpty`)
  - `Semaphore` (`acquire` ≈ `dequeue`, `release` ≈ `enqueue`)
  - PoisonPill/ShutdownSignal pattern[3]
  - ...
- Data transfer
  - Push-based producer-consumer
  - Backpressure/reactive publisher-subscriber
- Load balancing
  - Work arbitrage, work dealing, work stealing

What *precisely* do we need from thread-safe container?

[3] https://java-design-patterns.com/patterns/poison-pill/#benefits-and-trade-offs-of-poison-pill-pattern

Queues: design space

Queues: design space

- Ordering guarantees

Queues: design space

- Ordering guarantees
  - Strong FIFO

Queues: design space

- Ordering guarantees
  - Strong FIFO (linearizable to sequential queue)

Queues: design space

- Ordering guarantees
  - Strong FIFO (linearizable to sequential queue)
  - Per-producer FIFO

Queues: design space

- Ordering guarantees
  - Strong FIFO (linearizable to sequential queue)
  - Per-producer FIFO (items enqueued by i-th producer are dequeued in FIFO order)

Queues: design space

- Ordering guarantees
  - Strong FIFO (linearizable to sequential queue)
  - Per-producer FIFO (items enqueued by i-th producer are dequeued in FIFO order)
  - Best-effort FIFO

Queues: design space

- Ordering guarantees
  - Strong FIFO (linearizable to sequential queue)
  - Per-producer FIFO (items enqueued by i-th producer are dequeued in FIFO order)
  - Best-effort FIFO
  - No ordering guarantees

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate vs Delayed (e.g. buffering) vs Eventual (e.g. dynamic routing)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate vs Delayed (e.g. buffering) vs Eventual (e.g. dynamic routing)
  - Guaranteed vs Possible loss (e.g. UDP)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate vs Delayed (e.g. buffering) vs Eventual (e.g. dynamic routing)
  - Guaranteed vs Possible loss (e.g. UDP)
  - Unique vs Possible duplicate

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate vs Delayed (e.g. buffering) vs Eventual (e.g. dynamic routing)
  - Guaranteed vs Possible loss (e.g. UDP)
  - Unique vs Possible duplicate
    - Same item visible to competing dequeuers

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate vs Delayed (e.g. buffering) vs Eventual (e.g. dynamic routing)
  - Guaranteed vs Possible loss (e.g. UDP)
  - Unique vs Possible duplicate
    - Same item visible to competing dequeuers
    - Message resent under some condition

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - Single Producer Single Consumer (SPSC)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - Single Producer Single Consumer (SPSC)
  - Multiple Producers Single Consumer (MPSC)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - Single Producer Single Consumer (SPSC)
  - Multiple Producers Single Consumer (MPSC)
  - Single Producer Multiple Consumers (SPMC)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - Single Producer Single Consumer (SPSC)
  - Multiple Producers Single Consumer (MPSC)
  - Single Producer Multiple Consumers (SPMC)
  - Multiple Producers Multiple Consumers (MPMC)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity
  - Bounded, Unbounded

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity
  - Bounded, Unbounded
- API behaviour

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity
  - Bounded, Unbounded
- API behaviour
  - *total* methods do not wait (`dequeue` throws `EmptyException`)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity
  - Bounded, Unbounded
- API behaviour
  - *total* methods do not wait (`dequeue` throws `EmptyException`)
  - *partial* methods may wait for condition (`enqueue` awaits empty slot in bounded queue)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity
  - Bounded, Unbounded
- API behaviour
  - *total* methods do not wait (`dequeue` throws `EmptyException`)
  - *partial* methods may wait for condition (`enqueue` awaits empty slot in bounded queue)
  - *synchronous* methods wait for overlapping method (`enqueue` awaits paired `dequeue`)

Queues: design space

- Ordering guarantees
  - Strong FIFO, Per-producer FIFO, Best-effort FIFO, Nothing
- Delivery guarantees
  - Immediate, Delayed, Eventual, Guaranteed, Unique
- Supported concurrency
  - SPSC, MPSC, SPMC, MPMC
- Capacity
  - Bounded, Unbounded
- API behaviour
  - *total* methods do not wait (`dequeue` throws `EmptyException`)
  - *partial* methods may wait for condition (`enqueue` awaits empty slot in bounded queue)
  - *synchronous* methods wait for overlapping method (`enqueue` awaits paired `dequeue`)

We will look at some points in this design space

Lecture plan

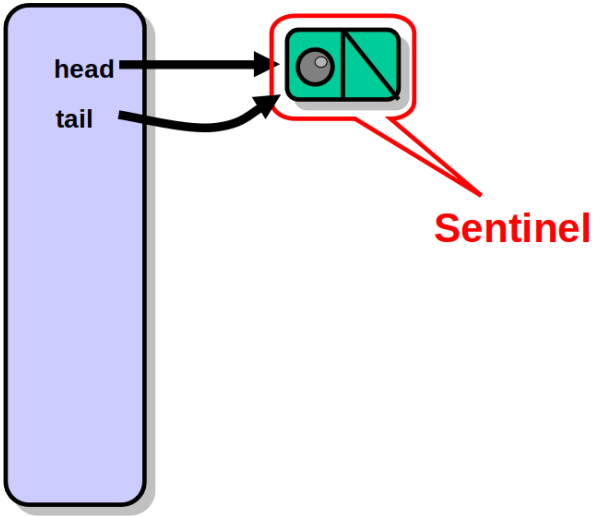
Unbounded MPMC linearizable total queue

- Strong FIFO
- MPMC
- Unbounded
- Total

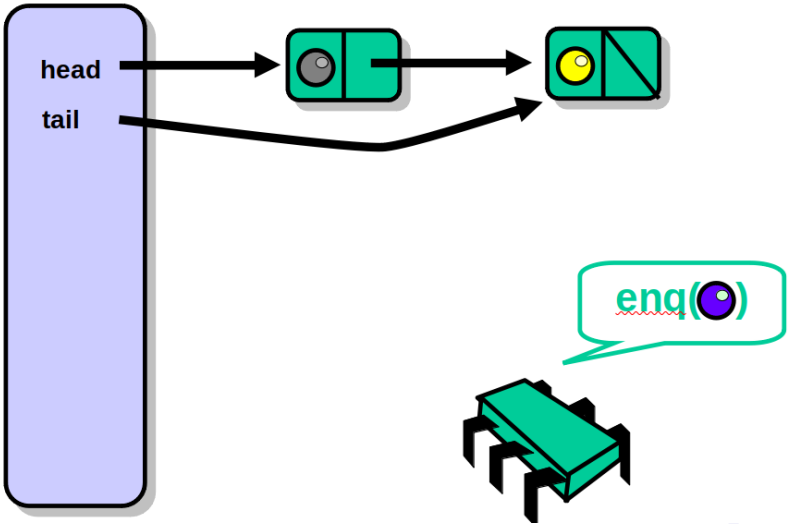
Unbounded MPMC linearizable total queue

- Strong FIFO
- MPMC
- Unbounded
- Total
- Lock-free

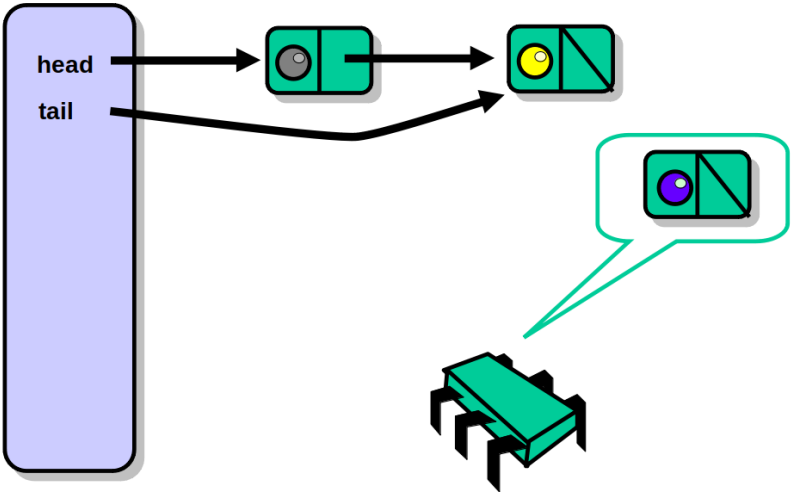
Lock-Free Queue



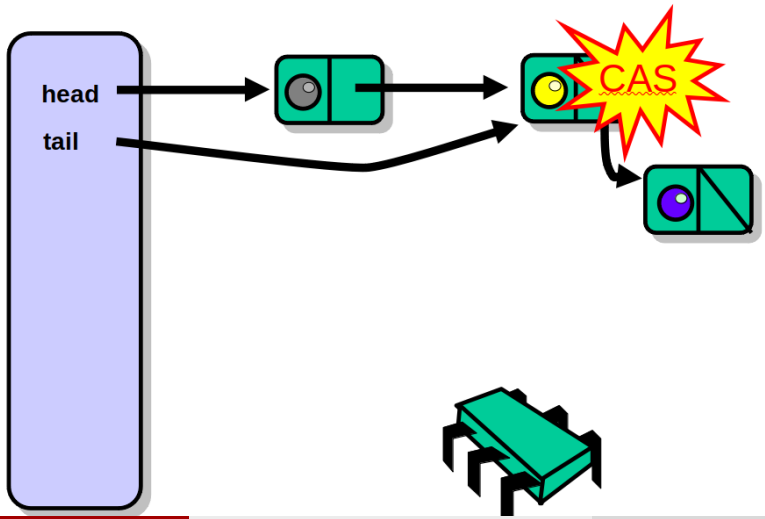
Enqueue



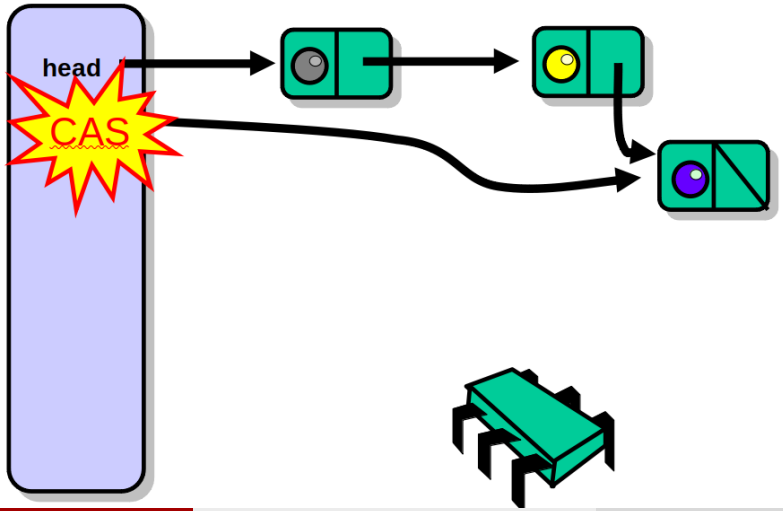
Enqueue



Logical enqueue



Physical enqueue



Enqueue

Logical enqueue and physical enqueue are separate steps

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation
  - But still lock-free (why?)

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation
  - But still lock-free (why?)
- Physical `enqueue`

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation
  - But still lock-free (why?)
- Physical `enqueue`
  - Failed CAS means other thread helped to fix invariants

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation
  - But still lock-free (why?)
- Physical `enqueue`
  - Failed CAS means other thread helped to fix invariants
- Inside `dequeue`

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation
  - But still lock-free (why?)
- Physical `enqueue`
  - Failed CAS means other thread helped to fix invariants
- Inside `dequeue`
  - Actual `tail` lagging behind `head`

Enqueue

Logical enqueue and physical enqueue are separate steps

- `tail` fields refers to either
  - Actual last node (good)
  - "Lagging" node (not so good)

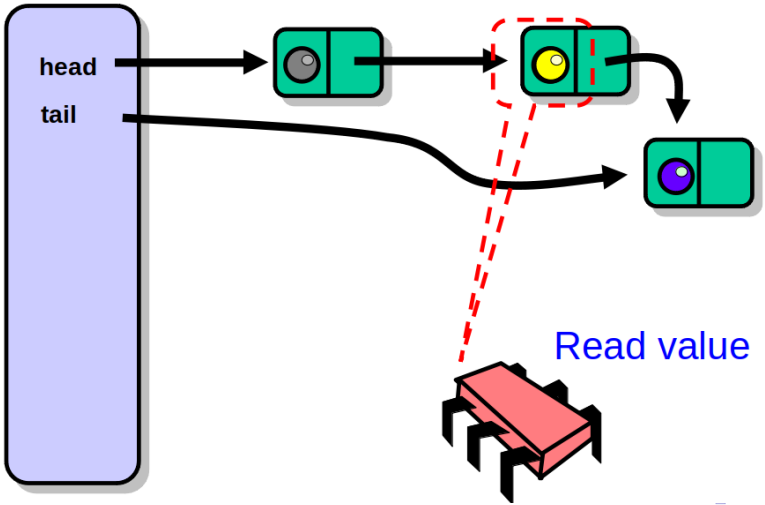
What should we do with "trailing tail"

- Stop and fix it before doing anything else
- If `tail.next != null` then CAS `tail -> tail.next`

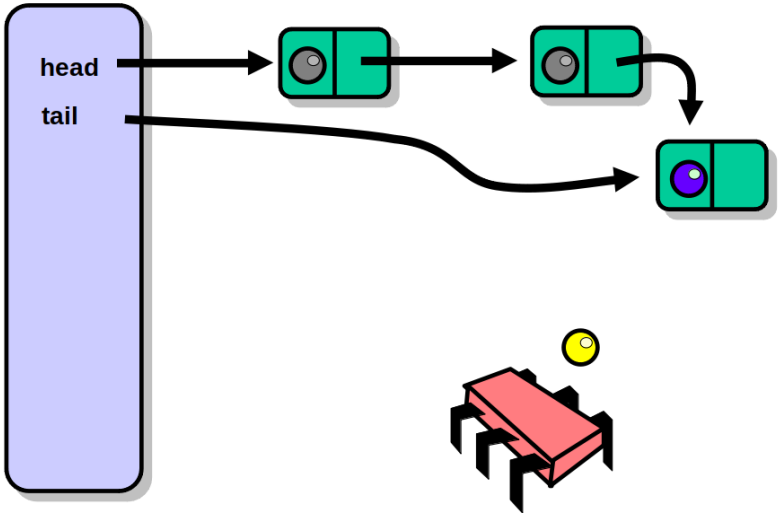
Why is it important

- Logical `enqueue`
  - Failed CAS means we need to restart enqueue operation
  - But still lock-free (why?)
- Physical `enqueue`
  - Failed CAS means other thread helped to fix invariants
- Inside `dequeue`
  - Actual `tail` lagging behind `head`
  - Swapping "sentinel" may delete actual data, make tail up-to-date before dequeue

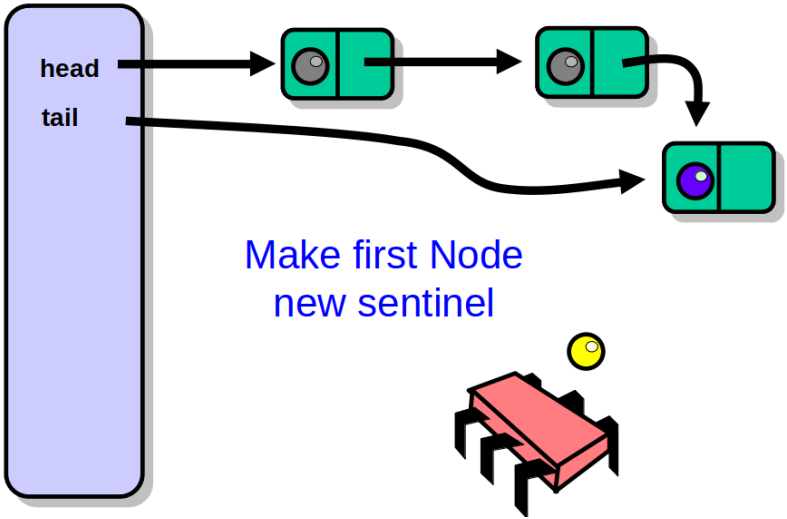
Dequeue



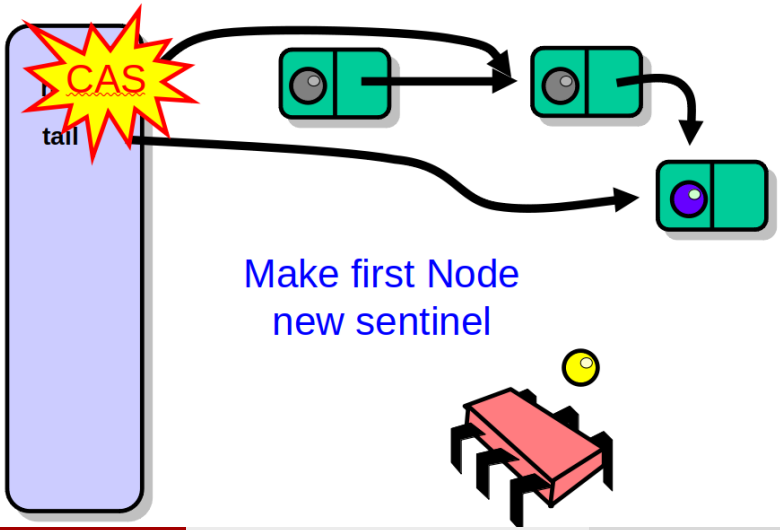
Dequeue



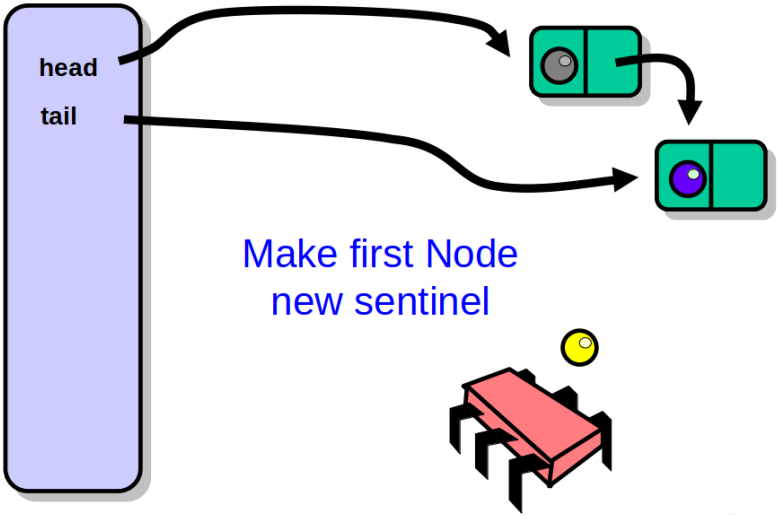
Dequeue



Dequeue



Dequeue



Enqueue: code

```
public void enqueue(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = last.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) { // logical
          tail.compareAndSet(last, node);          // physical
          return;
        }
      } else {
        tail.compareAndSet(last, next); // repair tail and repeat
      }
}}}
```

Dequeue: code

```
public T dequeue() {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) throw new EmptyException();
        tail.compareAndSet(last, next); // repair tail and repeat
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next)) return value;
}}}}
```

Unbounded MPMC linearizable total queue

- LinkedList-based
- Strong FIFO[4]
- Unbounded, needs Garbage Collection to avoid ABA
- Enqueue uses "helping hand" pattern to be lock-free
- Dequeue uses "helping hand" pattern to be correct

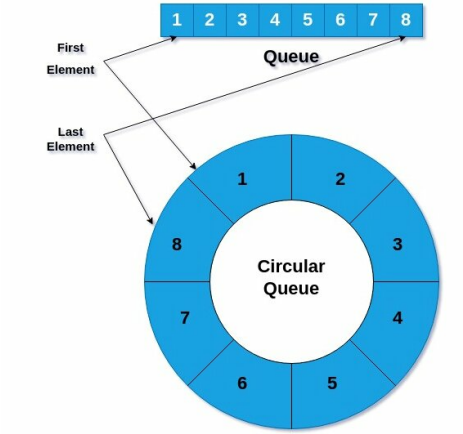
See Section 10.5 "An Unbounded Lock-Free Queue" in "The Art of Multiprocessor Programming" (pages 230 - 233).

[4] Homework: find linearization points

Lecture plan

1. Queues for mutual exclusion
2. Design space
3. Unbounded lock-free queue
4. Bounded array-based queues
   - SPSC
   - MPSC
   - SPMC
5. Concurrent DEQueue and work stealing
6. Summary

Bounded array-based queues



Lecture plan

SPSC

```
class SPSC_WaitFree_Queue<T> {
  volatile int head = 0, tail = 0; T[] items = (T[]) new Object[capacity];
  void enqueue(T x) {
    if (tail - head == items.length) throw new FullException();
    items[tail % items.length] = x;
    tail++; // not even getAndAdd
  }
  T dequeue() {
    if (tail - head == 0) throw new EmptyException();
    T x = items[head % items.length];
    head++; // not even getAndAdd
    return x;
}}
```

SPSC

```
class SPSC_WaitFree_Queue<T> {
  volatile int head = 0, tail = 0; T[] items = (T[]) new Object[capacity];
  void enqueue(T x) {
    if (tail - head == items.length) throw new FullException();
    items[tail % items.length] = x;
    tail++; // not even getAndAdd
  }
  T dequeue() {
    if (tail - head == 0) throw new EmptyException();
    T x = items[head % items.length];
    head++; // not even getAndAdd
    return x;
}}
```

Could you find linearization points? Could you find memory barriers?

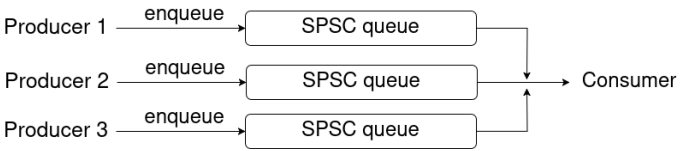
Lecture plan

MPSC

Question time

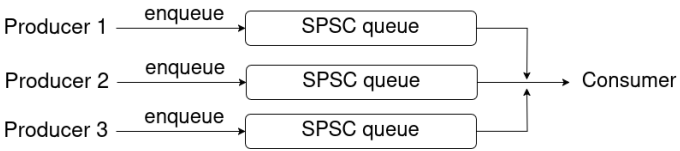
Question: How to create correct MPSC queue from **single** SPSC queue?

MPSC from N SPSC



MPSC from N SPSC

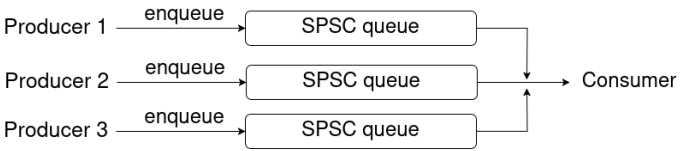
```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Wait-free?

MPSC from N SPSC

```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Wait-free? Thanks to SPSC.

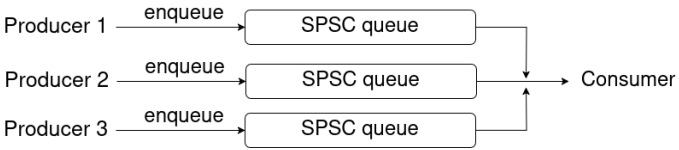
MPSC from N SPSC

```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```

- Wait-free? Thanks to SPSC. But dequeue is O(T).

MPSC from N SPSC

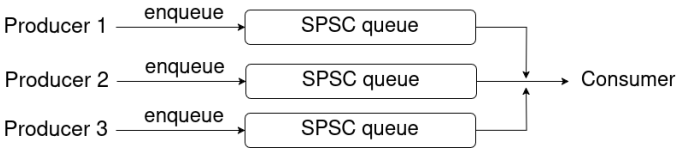
```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Strong FIFO?

MPSC from N SPSC

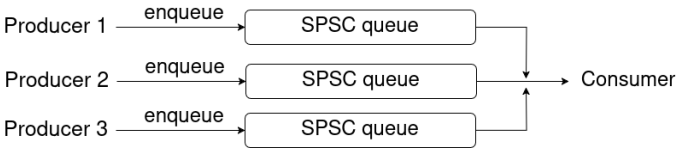
```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Strong FIFO?
  - `t1:enq(x)` → `t2:enq(y)` **does not** imply `deq(x)` → `deq(y)`

MPSC from N SPSC

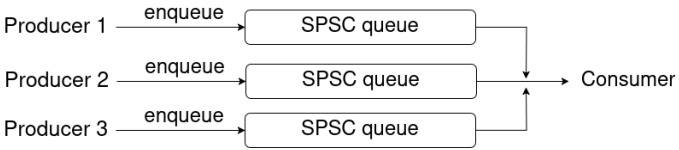
```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Strong FIFO?
  - `t1:enq(x)` → `t2:enq(y)` **does not** imply `deq(x)` → `deq(y)`
  - `t1:enq(x)` → `t1:enq(y)` **does** imply `deq(x)` → `deq(y)`

MPSC from N SPSC

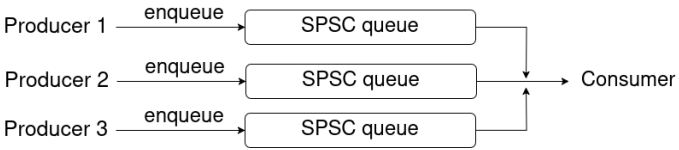
```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Really empty?

MPSC from N SPSC

```
SPSC[] queues = ...
void enqueue(T x) {
  int id = ThreadID.get();
  queues[id].enqueue(x);
}
T dequeue() {
  for (q : queues) if (!q.isEmpty()) return q.dequeue();
  throw new EmptyException();
}
```



- Really empty? Homework: what is *empty* for this queue

MPSC from N SPSC

- Need to register threads in advance

MPSC from N SPSC

- Need to register threads in advance
- Weak ordering guarantees

MPSC from N SPSC

- Need to register threads in advance
- Weak ordering guarantees
- But wait-free!

MPSC from N SPSC

- Need to register threads in advance
- Weak ordering guarantees
- But wait-free!
- `isEmpty` takes O(T)

MPSC from N SPSC

- Need to register threads in advance
- Weak ordering guarantees
- But wait-free!
- `isEmpty` takes O(T)
- Use atomic counters for `size`?

MPSC from N SPSC

- Use atomic counters for O(1) size check

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  size.getAndAdd(1);
  queues[ThreadID.get()].enqueue(x);
}
T dequeue() {
  if (size.get() == 0) throw new EmptyException();
  for (q : queues) if (!q.isEmpty()) {
    size.getAndAdd(-1);
    return q.dequeue();
  }
  assert false;
}
```

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  size.getAndAdd(1);
  queues[ThreadID.get()].enqueue(x);
}
T dequeue() {
  if (size.get() == 0) throw new EmptyException();
  for (q : queues) if (!q.isEmpty()) {
    size.getAndAdd(-1);
    return q.dequeue();
  }
  assert false;
}
```

AssertionError!

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  queues[ThreadID.get()].enqueue(x);
  size.getAndAdd(1);
}
T dequeue() {
  if (size.get() == 0) throw new EmptyException();
  for (q : queues) if (!q.isEmpty()) {
    size.getAndAdd(-1);
    return q.dequeue();
  }
  assert false;
}
```

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  queues[ThreadID.get()].enqueue(x);
  size.getAndAdd(1);
}
T dequeue() {
  if (size.get() == 0) throw new EmptyException();
  for (q : queues) if (!q.isEmpty()) {
    size.getAndAdd(-1);
    return q.dequeue();
  }
  assert false;
}
```

Missing already available element?

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  queues[ThreadID.get()].enqueue(x);
  size.getAndAdd(1);
}
T dequeue() {
  while (true) {
    for (q : queues) if (!q.isEmpty()) {
      size.getAndAdd(-1); return q.dequeue();
    }
    if (size.get() == 0) throw new EmptyException();
}}
```

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  queues[ThreadID.get()].enqueue(x);
  size.getAndAdd(1);
}
T dequeue() {
  while (true) {
    for (q : queues) if (!q.isEmpty()) {
      size.getAndAdd(-1); return q.dequeue();
    }
    if (size.get() == 0) throw new EmptyException();
}}
```

- Wait-free?

MPSC from N SPSC

- Use atomic counters for O(1) size check

```
void enqueue(T x) {
  queues[ThreadID.get()].enqueue(x);
  size.getAndAdd(1);
}
T dequeue() {
  while (true) {
    for (q : queues) if (!q.isEmpty()) {
      size.getAndAdd(-1); return q.dequeue();
    }
    if (size.get() == 0) throw new EmptyException();
}}
```

- Wait-free?
- Negative size?

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking
- *Definitely will be* error-prone

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking
- *Definitely will be* error-prone
- Relaxing consistency requirements helps

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking
- *Definitely will be* error-prone
- Relaxing consistency requirements helps
- Useful to think in terms of linearization points

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking
- *Definitely will be* error-prone
- Relaxing consistency requirements helps
- Useful to think in terms of linearization points
  - Operation takes effect **here**

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking
- *Definitely will be* error-prone
- Relaxing consistency requirements helps
- Useful to think in terms of linearization points
  - Operation takes effect **here**
  - Even if the whole API is not linearizable

Building queues from other queues: takeaways

Building distributed queue from more specialized parts:

- *Could* be scalability-friendly
- *Could* be *really* non-blocking
- *Definitely will be* error-prone
- Relaxing consistency requirements helps
- Useful to think in terms of linearization points
  - Operation takes effect **here**
  - Even if the whole API is not linearizable

Specialized lock-free single-queue algorithms exist:

- JCTools
  - `https://github.com/JCTools/JCTools`
  - `https://www.infoq.com/presentations/jctools-algorithms-optimization`
- 1024 cores
  - `https://www.1024cores.net/home/lock-free-algorithms/queues`

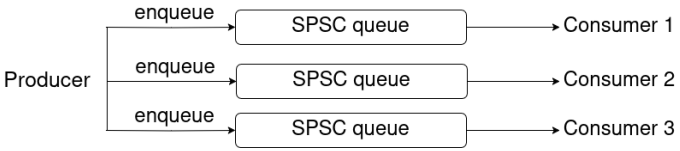
Lecture plan

SPMC

Question time

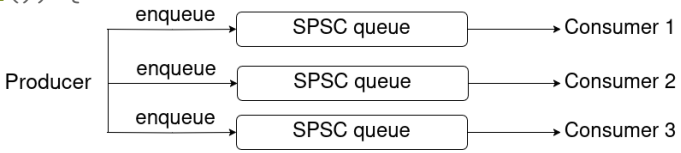
Question: How to build SPMC queue from **one** single-producer k-consumer queue?

SPMC from N SPSC



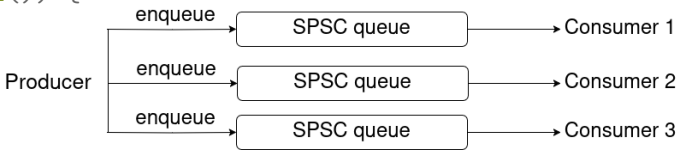
SPMC from N SPSC

```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



SPMC from N SPSC

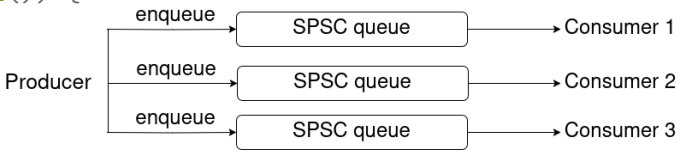
```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- What is empty queue?

SPMC from N SPSC

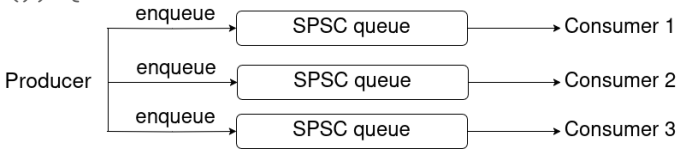
```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- What is empty queue?
  - Producer view: all consumer queues are empty

SPMC from N SPSC

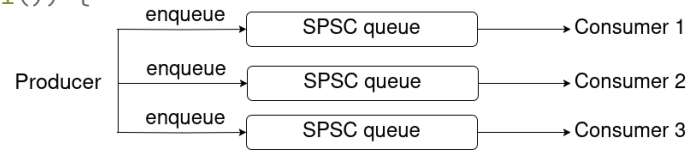
```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- What is empty queue?
  - Producer view: all consumer queues are empty
  - Consumer view: my queue is empty

SPMC from N SPSC

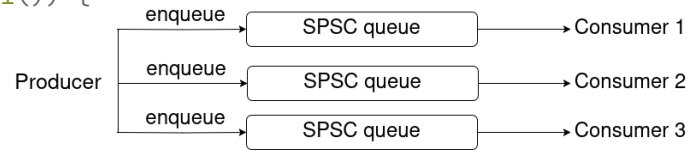
```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- Linearizability?

SPMC from N SPSC

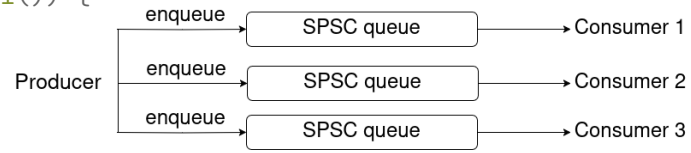
```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- Linearizability? No, but maybe OK

SPMC from N SPSC

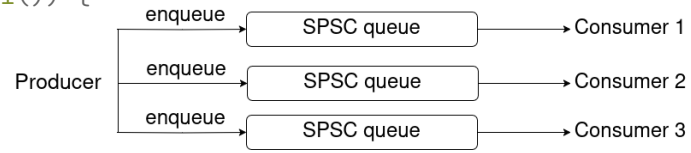
```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- Linearizability? No, but maybe OK
- Load balancing?

SPMC from N SPSC

```
SPSC[] queues = ...
void enqueue(T x) {
  for (q : queues) if (!q.isFull()) {
    q.enqueue(x); return;
  }
  throw new FullException();
}
T dequeue() {
  return queues[ThreadID.get()].dequeue();
}
```



- Linearizability? No, but maybe OK
- Load balancing? No, definitely not OK

SPMC from N SPSC

- Naive work-stealing as load balancing strategy

SPMC from N SPSC

- Naive work-stealing as load balancing strategy

```
void enqueue(T x) {
  for (q : queues) synchronized(q) { if (!q.isFull()) {
    q.enqueue(x); return;
  }}
  throw new FullException();
}
T dequeue() {
  SPSC q = queues[ThreadID.get()];
  synchronized(q) { if (!q.isEmpty()) return q.dequeue(); }
  for(q : queues) synchronized(q) { if (!q.isEmpty()) return q.dequeue(x); }
}
```

SPMC from N SPSC

- Naive work-stealing as load balancing strategy

```
void enqueue(T x) {
  for (q : queues) synchronized(q) { if (!q.isFull()) {
    q.enqueue(x); return;
  }}
  throw new FullException();
}
T dequeue() {
  SPSC q = queues[ThreadID.get()];
  synchronized(q) { if (!q.isEmpty()) return q.dequeue(); }
  for(q : queues) synchronized(q) { if (!q.isEmpty()) return q.dequeue(x); }
}
```

- Real-life algorithms: batch steal, lock-free steal

Consumer contention: takeaways

- Many producers/consumers – maybe strong FIFO is not that necessary

Consumer contention: takeaways

- Many producers/consumers – maybe strong FIFO is not that necessary
- Many consumers – load balancing required

Consumer contention: takeaways

- Many producers/consumers – maybe strong FIFO is not that necessary
- Many consumers – load balancing required
- Non-blocking work stealing looks like hard problem

Consumer contention: takeaways

- Many producers/consumers – maybe strong FIFO is not that necessary
- Many consumers – load balancing required
- Non-blocking work stealing looks like hard problem

Specialized lock-free single-queue algorithms exist:

- JCTools
  - `https://github.com/JCTools/JCTools`
  - `https://www.infoq.com/presentations/jctools-algorithms-optimization`
- 1024 cores
  - `https://www.1024cores.net/home/lock-free-algorithms/queues`

Lecture plan

1. Queues for mutual exclusion
2. Design space
3. Unbounded lock-free queue
4. Bounded array-based queues
   - SPSC
   - MPSC
   - SPMC
5. Concurrent DEQueue and work stealing
6. Summary

Lock-Free Work Stealing

Lock-Free Work Stealing

- Each thread has a pool of ready work

Lock-Free Work Stealing

- Each thread has a pool of ready work
- Remove work without synchronizing

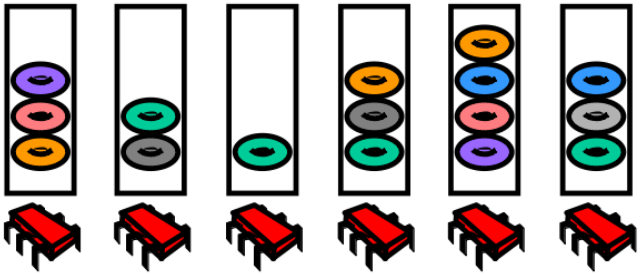
Lock-Free Work Stealing

- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal from someone

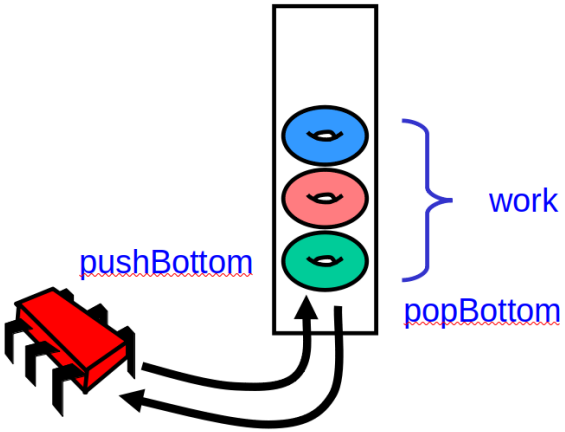
Lock-Free Work Stealing

- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal from someone
- Choose victim at random

Local work pools

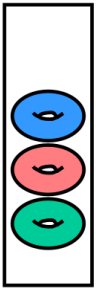


DoubleEndedQueue



Obtain work

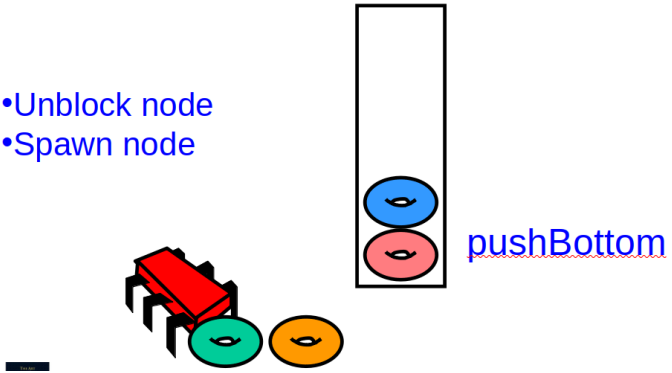
- Obtain work
- Run task until
- Blocks or terminates



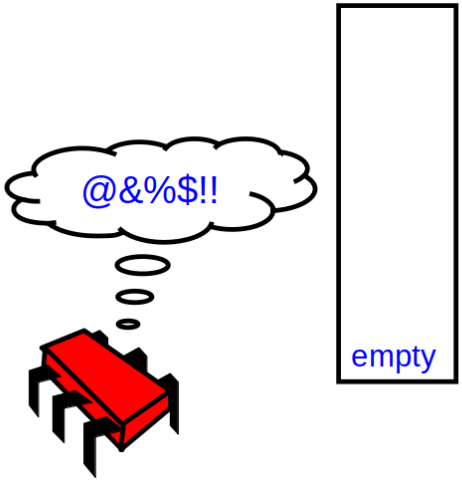
popBottom

New work

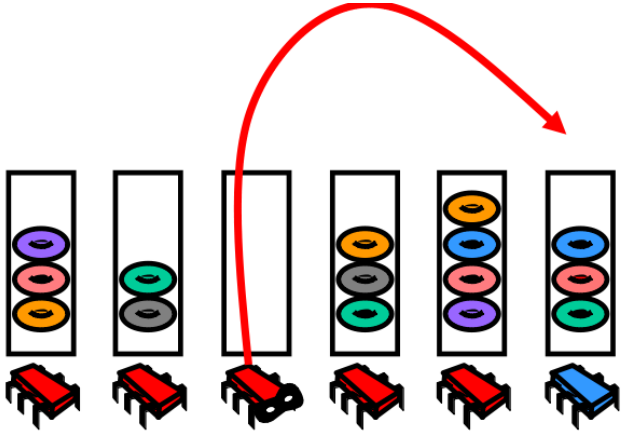
- Unblock node
- Spawn node



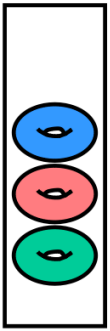
Out of work



Steal from others



Steal task



popTop

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

- make them fast

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

- make them fast
- minimize use of CAS

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

- make them fast
- minimize use of CAS

Ideal requirements:

- Wait-Free, Linearizable, Constant time

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

- make them fast
- minimize use of CAS

Ideal requirements:

- Wait-Free, Linearizable, Constant time

Compromise:

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

- make them fast
- minimize use of CAS

Ideal requirements:

- Wait-Free, Linearizable, Constant time

Compromise: method `popTop` may fail if

Task DEQueue

Methods:

- `pushBottom`, `popBottom`, `popTop`

`pushBottom` and `popBottom` are never concurrent:

- make them fast
- minimize use of CAS

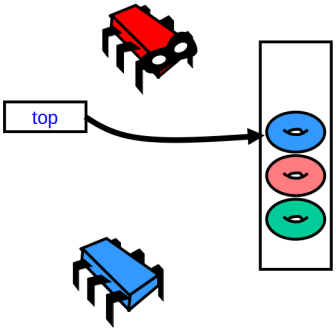
Ideal requirements:

- Wait-Free, Linearizable, Constant time

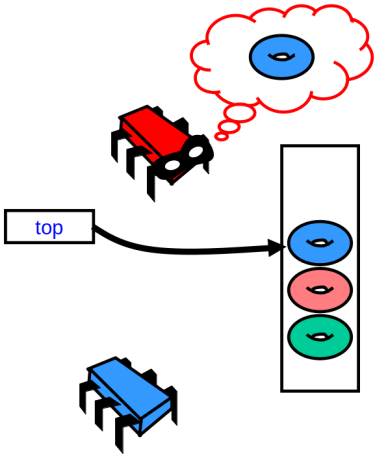
Compromise: method `popTop` may fail if

- Concurrent `popTop` succeeds or
- Concurrent `popBottom` takes last task

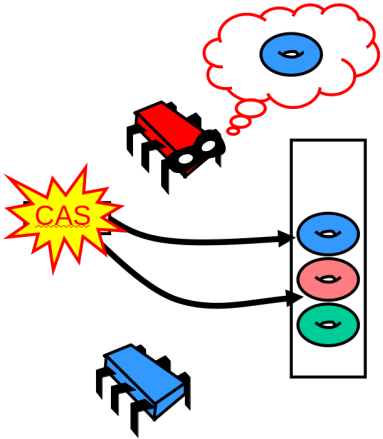
ABA problem



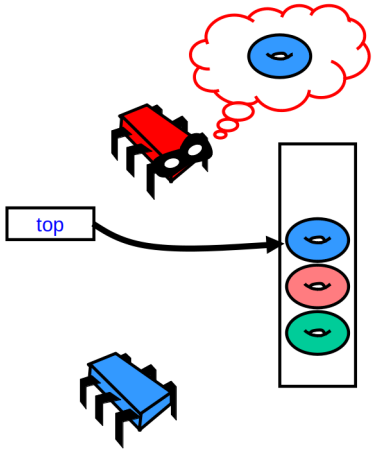
ABA problem



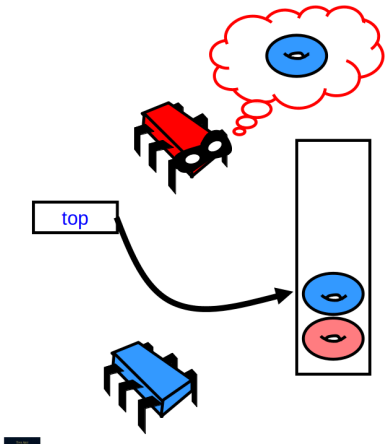
ABA problem



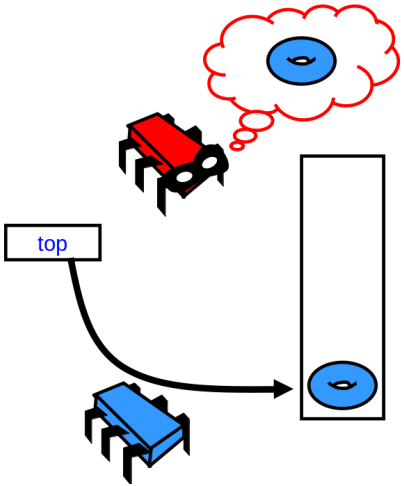
ABA problem



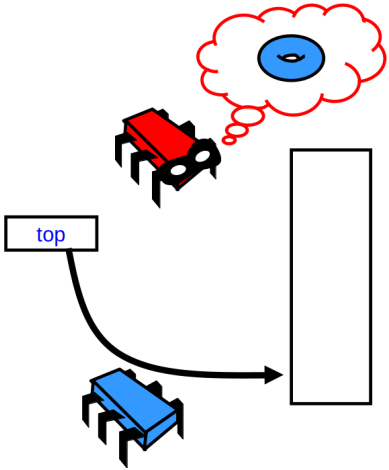
ABA problem



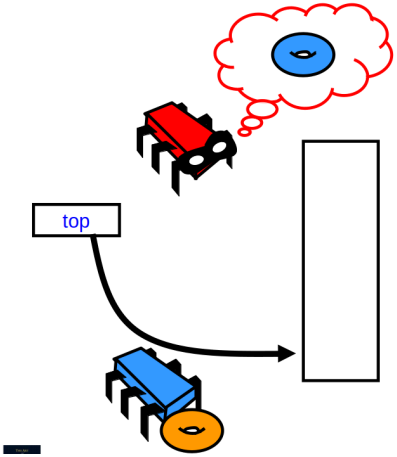
ABA problem



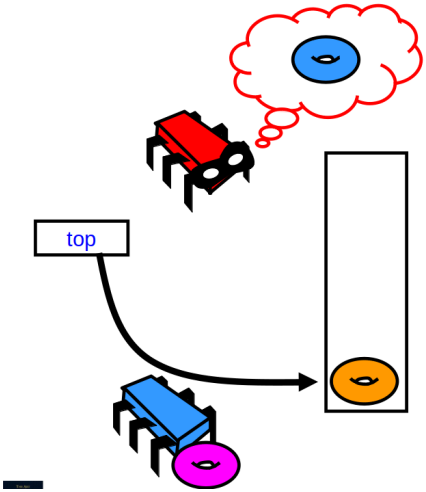
ABA problem



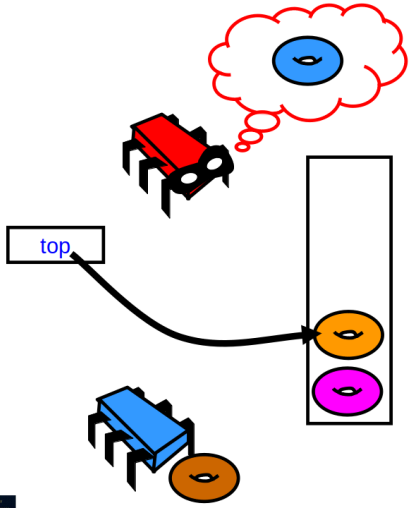
ABA problem



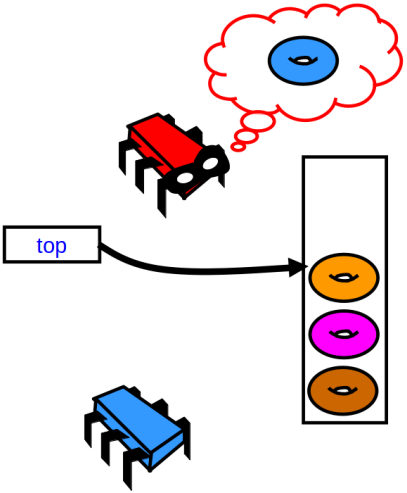
ABA problem



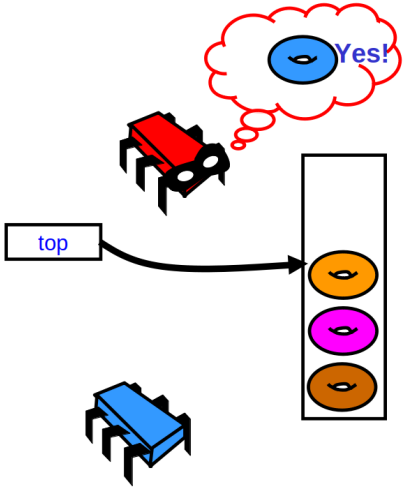
ABA problem



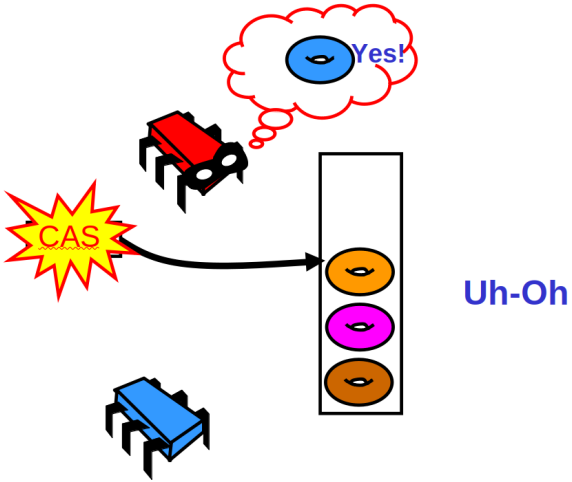
ABA problem



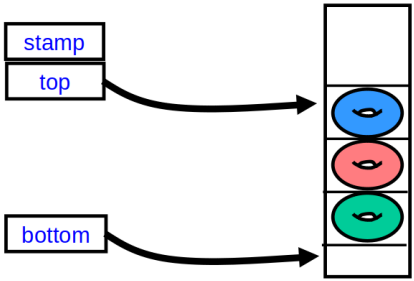
ABA problem



ABA problem



Fix for ABA problem



Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

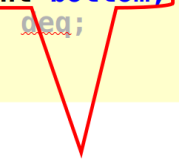
Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

Index & Stamp
(synchronized)

Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] deq;
 …
}
```



index of bottom task
no need to synchronize
memory barrier needed

Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

Array holding tasks

pushBottom

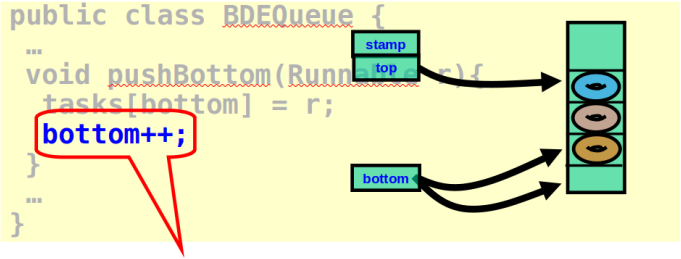
```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```

pushBottom

```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```

Bottom is the index to store the new task in the array

pushBottom


public class BDEQueue {
…
void pushBottom(Runnable r){
tasks[bottom] = r;
bottom++;
}
…
}
stamp
top
bottom
Adjust the bottom index

Steal work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Steal work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Read top (value & stamp)

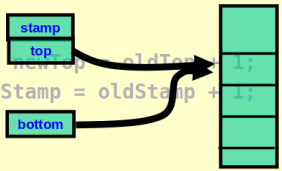
Steal work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Compute new value & stamp

Steal work

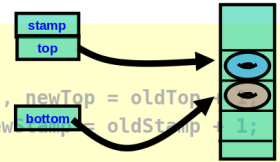
```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom <= oldTop)
  return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```



Quit if queue is empty

Steal work

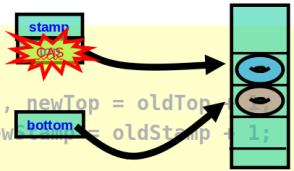
```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
   return r;
 return null;
 }
```



Try to steal the task

Steal work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```



Try to steal the task

Steal work


public Runnable popTop() {
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = oldTop + 1;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom <= oldTop)
return null;
Runnable r = tasks[oldTop];
if (top.CAS(oldTop, newTop, oldStamp, newStamp))
return r;
return null;
}
stamp
CAS
bottom
Try to steal the task

Steal work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
   return r;
return null;
}
```

Give up if conflict occurs

Take work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0; }
```

Take work

```
Runnable popBottom() {
if (bottom == 0) return null;
bottom--;
Runnable r = tasks[bottom];
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = 0;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
  bottom = 0;
  if (top.CAS(oldTop, newTop, oldStamp, newStamp))
    return r;
  }
top.set(newTop,newStamp); return null;
bottom = 0; }
```

Make sure queue is non-empty

Take work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0; }
```

Prepare to grab bottom task

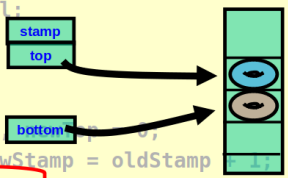
Take work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0; }
```

Read top, & prepare new values

Take work

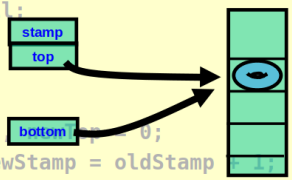
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp)
 int oldStamp = stamp[0], newStamp = oldStamp
if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
top.set(newTop,ne                    null;
 bottom = 0;}
```



If top & bottom one or more apart, no conflict

Take work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp);
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0;}
```



At most one item left

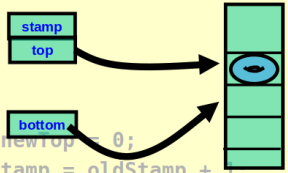
Take work

Try to steal last task.
Reset `bottom` because the DEQueue will be empty even if unsuccessful (why?)

```
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
  bottom = 0;
  if (top.CAS(oldTop, newTop, oldStamp, newStamp))
    return r;
  }
top.set(newTop,newStamp); return null;
bottom = 0;}
```

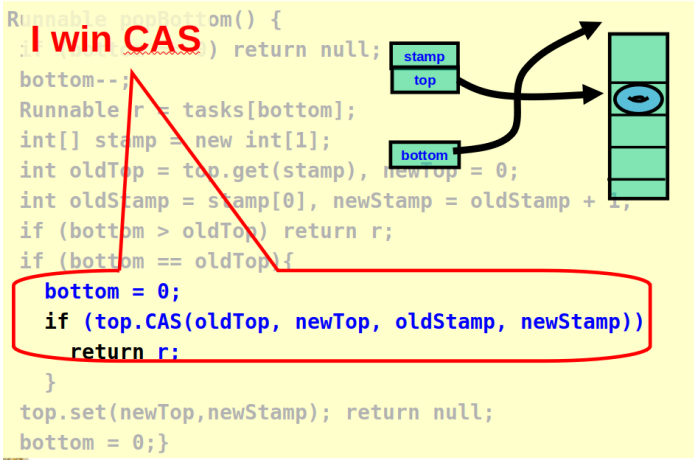
Take work

I win CAS

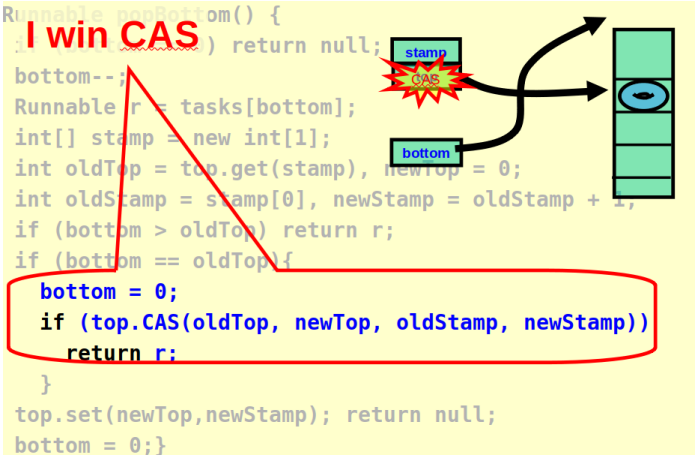



```
Runnable popBottom() {
  if (bottom == 0) return null;
  bottom--;
  Runnable r = tasks[bottom];
  int[] stamp = new int[1];
  int oldTop = top.get(stamp), newTop = 0;
  int oldStamp = stamp[0], newStamp = oldStamp + 1;
  if (bottom > oldTop) return r;
  if (bottom == oldTop){
    bottom = 0;
    if (top.CAS(oldTop, newTop, oldStamp, newStamp))
      return r;
    }
  top.set(newTop,newStamp); return null;
  bottom = 0;}
```

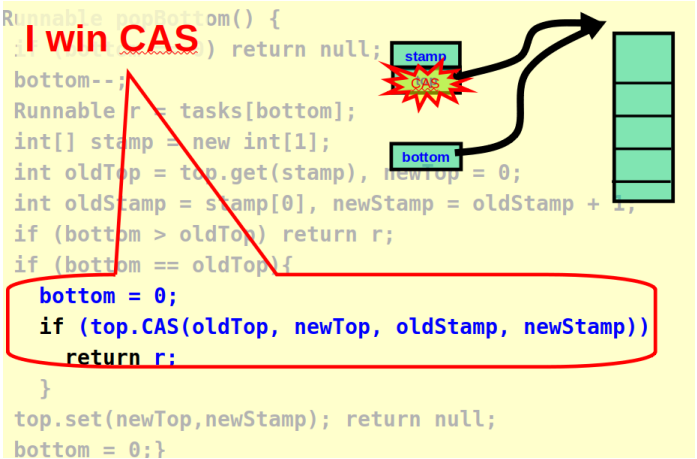
Take work


I win CAS
stamp
top
bottom
Runnable popBottom() {
) return null;
bottom--;
Runnable r = tasks[bottom];
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = 0;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
bottom = 0;
if (top.CAS(oldTop, newTop, oldStamp, newStamp))
return r;
}
top.set(newTop,newStamp); return null;
bottom = 0;}

Take work

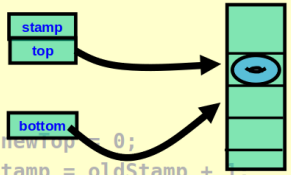

I win CAS
stamp
CAS
bottom
Runnable popBottom() {
if (bottom == 0) return null;
bottom--;
Runnable r = tasks[bottom];
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = 0;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
bottom = 0;
if (top.CAS(oldTop, newTop, oldStamp, newStamp))
return r;
}
top.set(newTop,newStamp); return null;
bottom = 0;}

Take work


I win CAS
stamp
CAS
bottom
Runnable popBottom() {
) return null;
bottom--;
Runnable r = tasks[bottom];
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = 0;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
bottom = 0;
if (top.CAS(oldTop, newTop, oldStamp, newStamp))
return r;
}
top.set(newTop,newStamp); return null;
bottom = 0;}

Take work

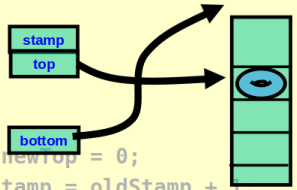
If I lose CAS, thief must have won…




```
Runnable popBottom() {
                                          ull;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0;}
```

Take work

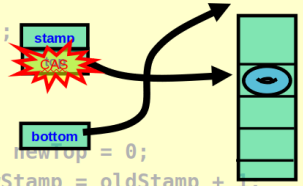
If I lose CAS, thief must have won…




```
Runnable popBottom() {
                                          ull;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
 bottom = 0;}
```

Take work

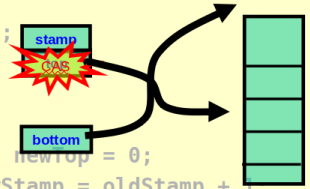
If I lose CAS, thief must have won…




```
Runnable popBottom() {
  Runnable r = tasks[bottom];
  int[] stamp = new int[1];
  int oldTop = top.get(stamp), newTop = 0;
  int oldStamp = stamp[0], newStamp = oldStamp + 1;
  if (bottom > oldTop) return r;
  if (bottom == oldTop){
    bottom = 0;
    if (top.CAS(oldTop, newTop, oldStamp, newStamp))
      return r;
    }
  top.set(newTop,newStamp); return null;
  bottom = 0;}
```

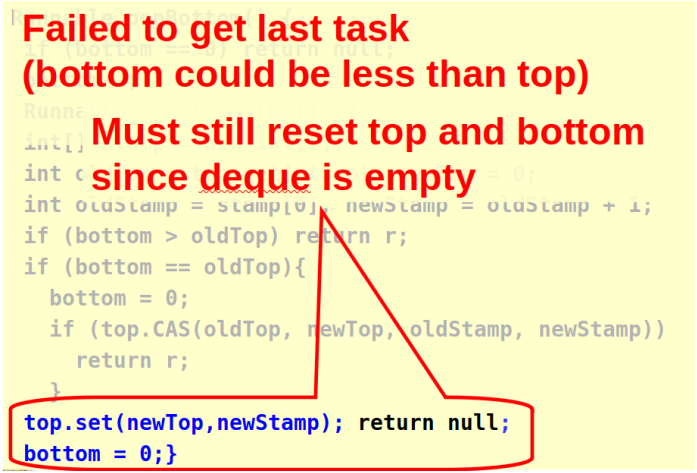
Take work

If I lose CAS, thief must have won…




```
Runnable popBottom() {
if (bottom-oldTop) return null;
Runnable r = tasks[bottom];
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = 0;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
  bottom = 0;
  if (top.CAS(oldTop, newTop, oldStamp, newStamp))
    return r;
  }
top.set(newTop,newStamp); return null;
bottom = 0;}
```

Take work


Failed to get last task
(bottom could be less than top)
Must still reset top and bottom
since deque is empty
int oldStamp = stamp[0]; newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
bottom = 0;
if (top.CAS(oldTop, newTop, oldStamp, newStamp))
return r;
top.set(newTop,newStamp); return null;
bottom = 0;}

Work stealing and balancing

- Trade-offs: fast-path/slow-path, FIFO/LIFO, lock-free/obstruction-free, steal size

Work stealing and balancing

- Trade-offs: fast-path/slow-path, FIFO/LIFO, lock-free/obstruction-free, steal size
- Clean separation between app and scheduling layer

Work stealing and balancing

- Trade-offs: fast-path/slow-path, FIFO/LIFO, lock-free/obstruction-free, steal size
- Clean separation between app and scheduling layer
- Works well when number of processors fluctuates

Work stealing and balancing

- Trade-offs: fast-path/slow-path, FIFO/LIFO, lock-free/obstruction-free, steal size
- Clean separation between app and scheduling layer
- Works well when number of processors fluctuates
- Works on "black-box" operating systems

Summary

Queues are quite universal construct:

- Concurrency control (mutex, latch, semaphore)
- Data transfer
- Load balancing

Design space:

- Strong FIFO, per-producer FIFO, best-effort FIFO
- SPSC, MPSC, SPMC, MPMC
- Bounded, unbounded
- Total, partial, synchronous methods
- Wait-free/lock-free/blocking enqueuers/dequeuers

Load balancing using DoubleEndedQueue

- bottom-oriented LIFO for local tasks
- top-oriented work-stealing