Concurrency: Multi-core Programming & Data Processing

Concurrent Queues and Stacks

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

Institut d'informatique
Université de Neuchâtel

*Based on slides by **Maurice Herlihy** and **Nir Shavit***

Last Lecture

- Five concurrent data structure designs
  - Coarse-grained locking
    - Simple but hotspot + bottleneck
  - Fine-grained locking
    - All delayed by front thread: hotspots + bottleneck
  - Optimistic synchronization
    - Limited hotspots but two traversals
  - Lazy synchronization
    - Lazy add/remove + wait-free contains
  - Lock-free synchronization
    - Lock-free add/remove + wait-free contains

Another Fundamental Problem

- We told you about
  - Sets implemented using linked lists
- Next: queues and stacks
  - Ubiquitous data structure
  - Often used to buffer requests…
- Queue/stacks belongs to broader pool class
- Pool: similar to set but
  - Allows duplicates (multiset)
  - No membership test (no `contains()`)

Pool Flavors

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

- Blocking (remove from empty pool, add to full pool)
  - Caller waits until state changes
- Non-blocking
  - Method throws exception

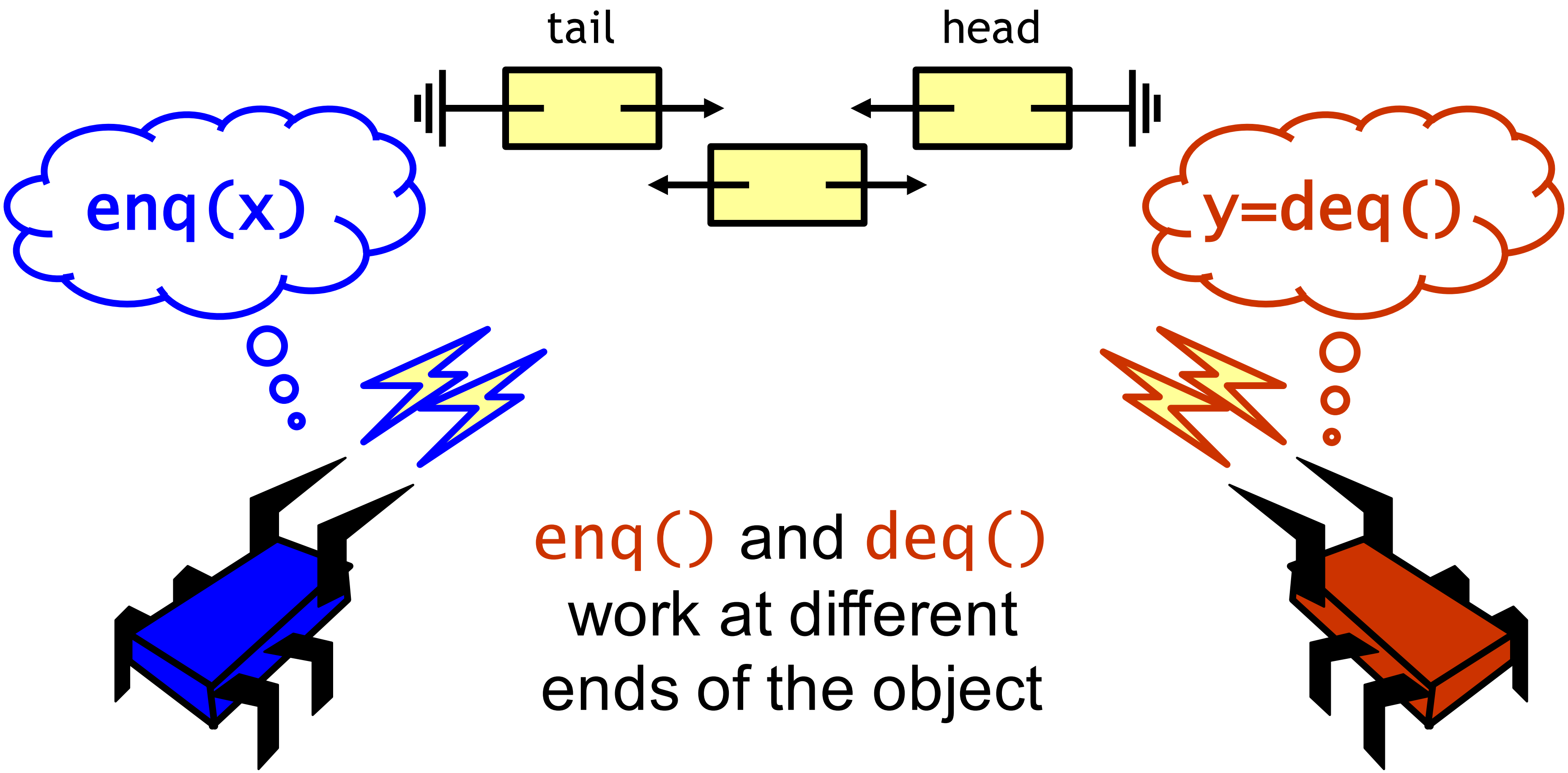
Queues & Stacks

- `add()` and `remove()`
  - Queue: `enq()` and `deq()`
  - Stack: `push()` and `pop()`
- A queue is a pool with FIFO order on enqueues and dequeues
- A stack is a pool with LIFO order on pushes and pops

This Lecture

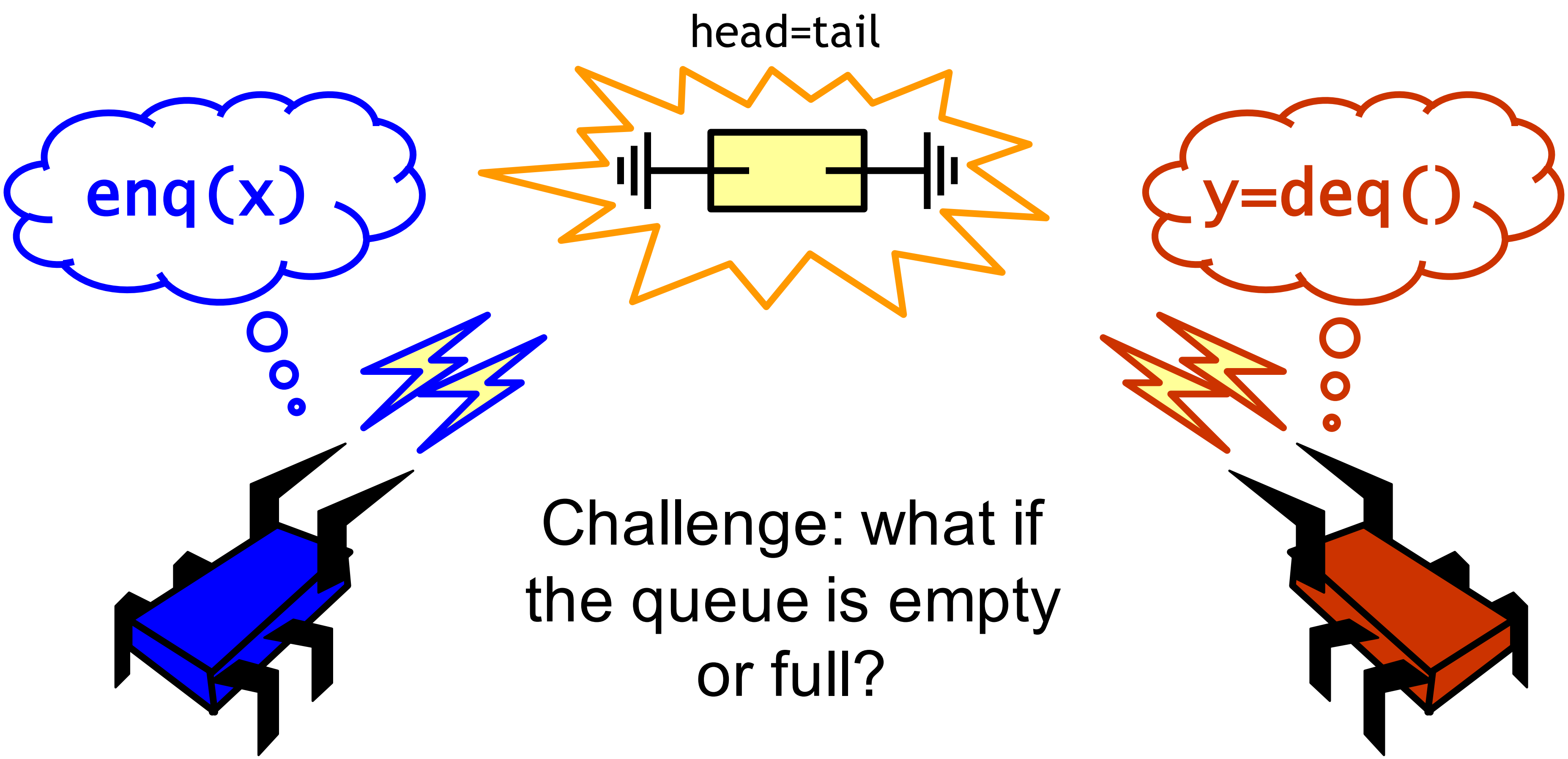
- Bounded, blocking, lock-based queue
- Unbounded, non-blocking, lock-free stack
- Elimination-backoff stack

Queue: Concurrency



enq() and deq() work at different ends of the object

Queue: Concurrency



Challenge: what if the queue is empty or full?

Java Monitor Locks

- The Java `ReentrantLock` is a monitor
  - Allows blocking on a condition rather than spinning
- Threads
  - Acquire lock
  - Release lock
  - Wait on a condition
  - Wake up threads waiting on condition

Java Monitor Locks

```
public interface Lock {
  void lock();                                      Acquire lock
  void lockInterruptibly()
    throw InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();                         Conditions to wait on
  void unlock();                                    Release lock
}
```

Java Lock Conditions

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

Release lock and wait on condition

Signal release of next thread in line

Signal release of all awaiting threads

The `await()` Method

```
c.await();
```

- Releases lock on `c` and sleeps (gives up processor)
  - Move to “waiting room” and wait to be awaken
- Upon being awaken
  - Reacquires lock and continue execution
- The awaiting thread must hold the lock prior to the call

The `signal()` Method

```
c.signal();
```

- Awakens one waiting thread
  - Which will reacquire lock
  - Possibly competing with other threads
- Then returns
- The signaling thread must hold the lock prior to the call

The `signalAll()` Method

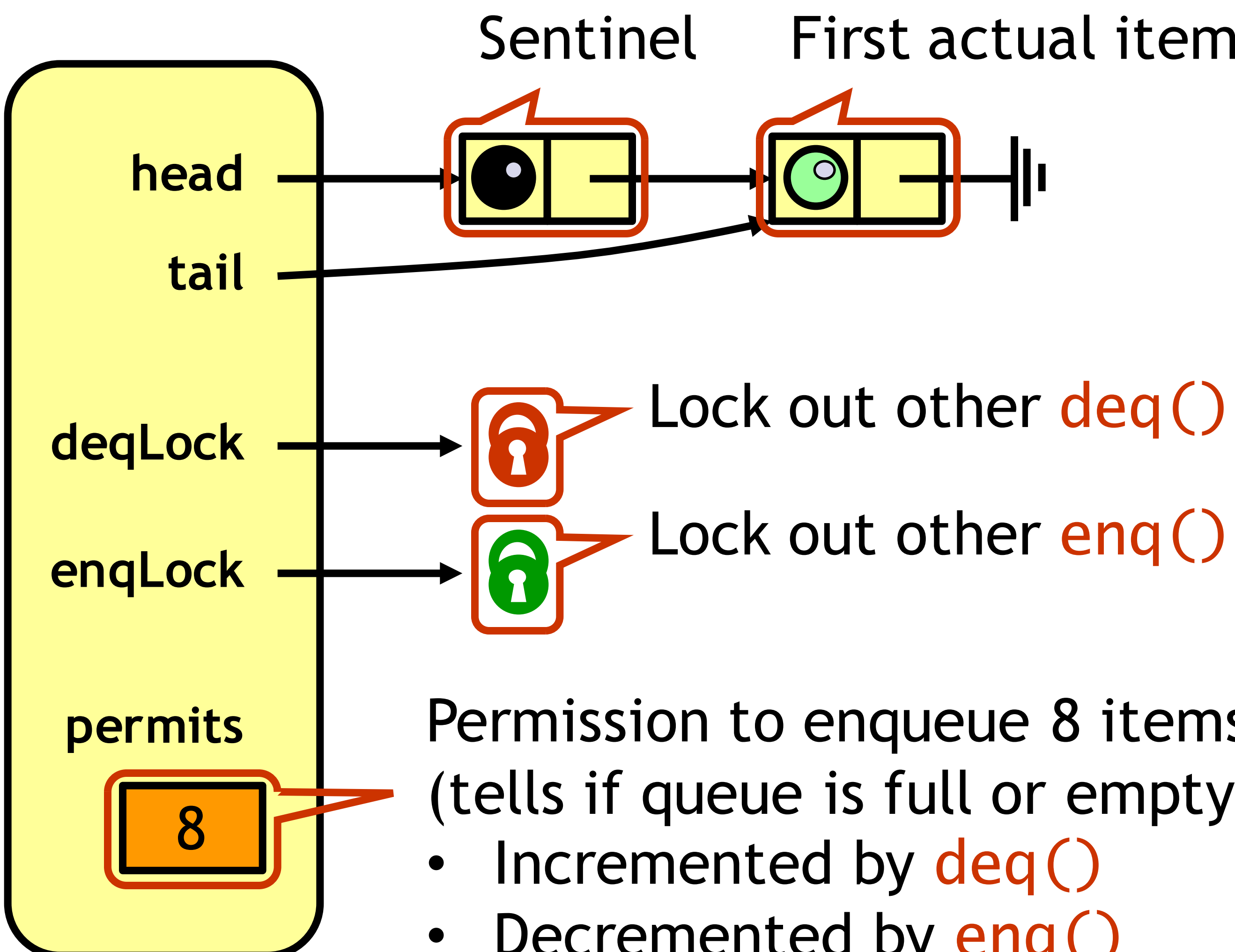
```
c.signalAll();
```

- Awakens all waiting threads
  - Which will reacquire lock
  - Possibly competing with other threads
- Then returns
- The signaling thread must hold the lock prior to the call

Java Synchronized Monitor

- Methods defined on class `Object`
  - `await()` → `wait()`
  - `signal()` → `notify()`
  - `signalAll()` → `notifyAll()`
- Can be called only from synchronized blocks or methods (lock must be held)

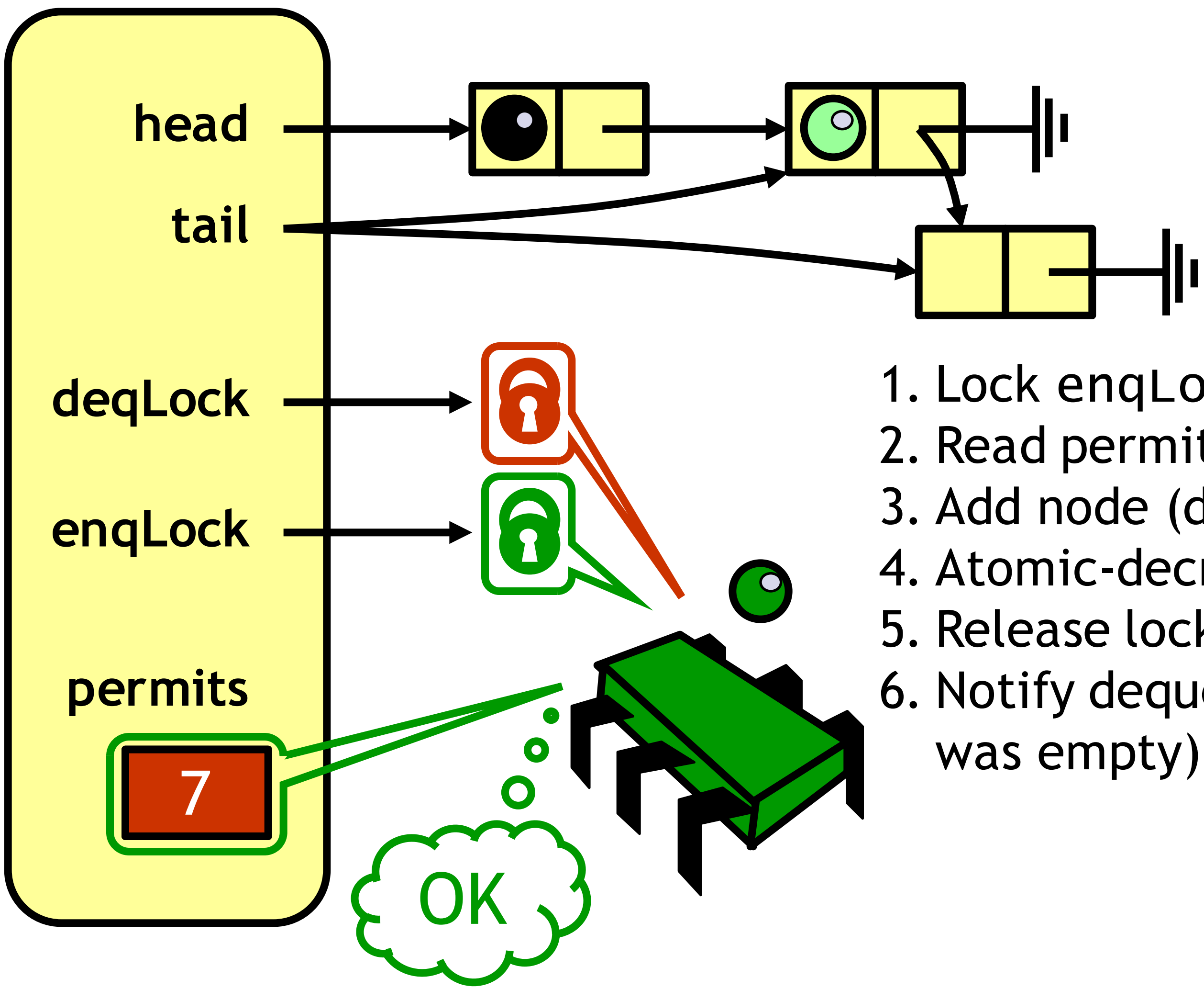
Bounded Queue



Permission to enqueue 8 items (tells if queue is full or empty)

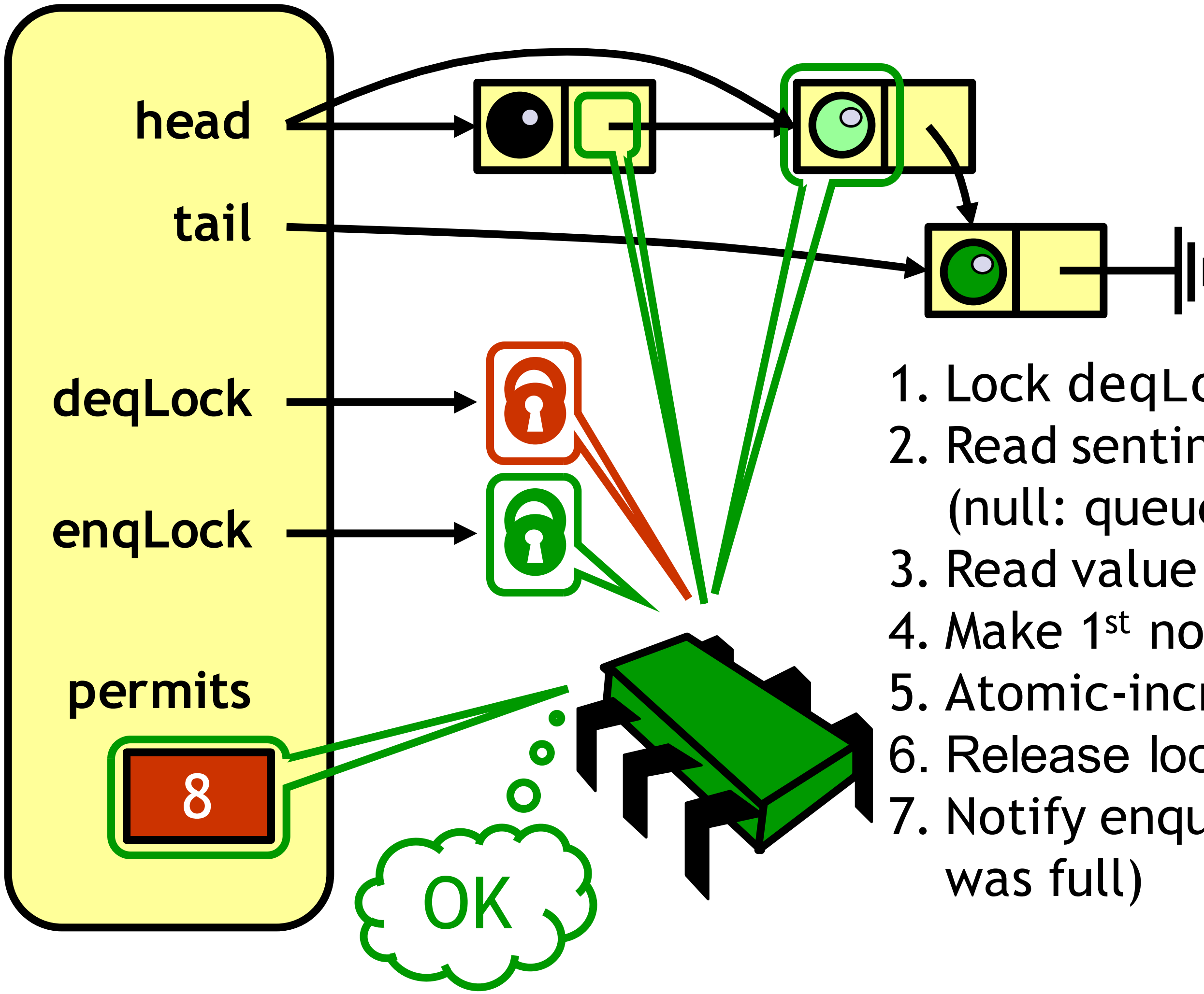
- Incremented by deq()
- Decremented by enq()

Enqueuer



1. Lock enqLock
2. Read permits (0: full)
3. Add node (don't lock tail)
4. Atomic-decrement permits
5. Release lock
6. Notify dequeuers (if queue was empty)

Dequeuer



1. Lock deqLock
2. Read sentinel’s next field (null: queue empty)
3. Read value
4. Make 1st node new sentinel
5. Atomic-increment permits
6. Release lock
7. Notify enqueuers (if queue was full)

Bounded Queue

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

Enqueue & dequeue locks

Condition for threads to wait while enqueuing

Condition for threads to wait while dequeuing

Number of permits from 0 to capacity

Head and tail

Capacity of the queue

Enqueue (part I)

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

Lock enqueue lock

If no permit, wait until `notFullCondition` becomes true then check permits again

Add a new node

If I was the enqueuer that changed queue state from empty to non-empty, need to wake dequeuers

Release enqueue lock

Enqueue (part II)

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

To let the dequeuers know that the queue is non-empty, acquire dequeue lock

Signal all dequeuers waiting that they can attempt to re-acquire dequeue lock

Release dequeue lock

Dequeue (part I)

```
public T deq() {
  boolean mustWakeEnqueuers = false;
  T v;
  deqLock.lock();
  try {
    while (head.next == null)
      notEmptyCondition.await();
    v = head.next.value;
    head = head.next;
    if (permits.getAndIncrement() == 0)
       mustWakeEnqueuers = true;
  } finally { deqLock.unlock(); } …
```

Is queue empty?

Read value of first node

Make first node new sentinel

Dequeue (part II)

```
public T deq() {
  …
  if (mustWakeEnqueuers) {
    enqLock.lock();
    try {
      notFullCondition.signalAll();
    } finally {
      enqLock.unlock();
    }
  }
  return v;
}
```

Return value after waking up enqueuers

The Shared Counter

- The `enq()` and `deq()` methods
  - Do not access the same lock concurrently…
  - But they still share a counter…
  - Which they both increment or decrement on every method call
  - Can we get rid of this bottleneck?

Split the Counter

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

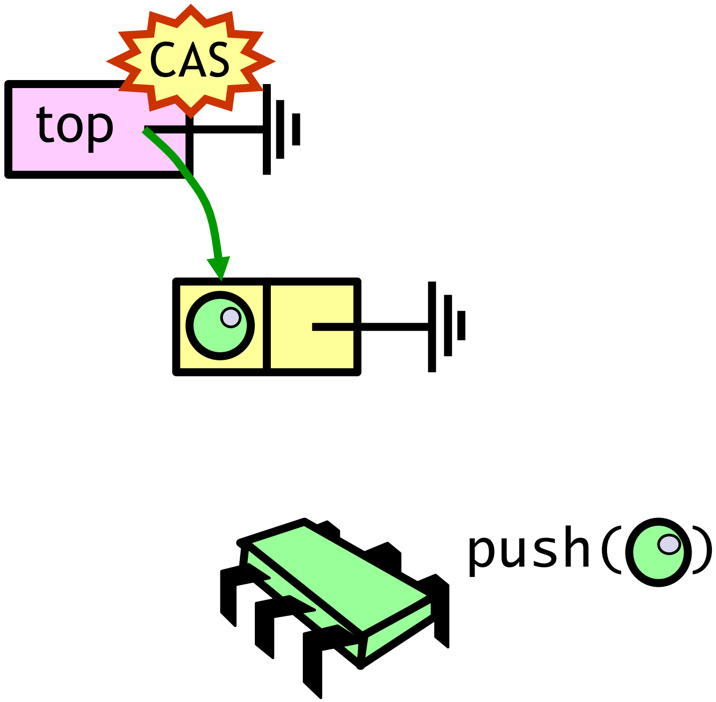
Split Counter

- Enqueuer decrements `enqSidePermits`
  - Initially set to capacity
- Dequeuer increments `deqSidePermits`
  - Initially 0
- When enqueuer runs out
  - Locks `deqLock` (holds both locks!)
  - Transfers permits from dequeue to enqueue side
- Intermittent synchronization
  - Not with each method call

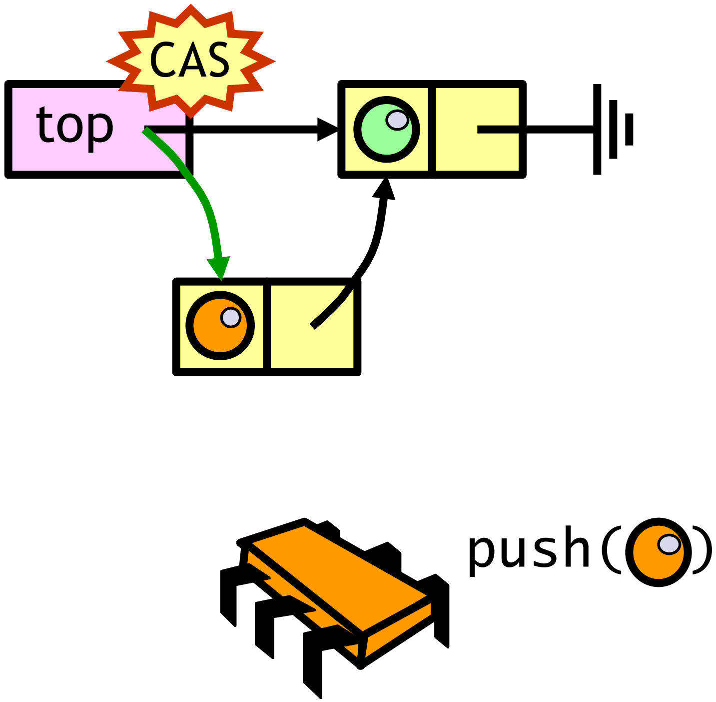
A Concurrent Stack

- `add()` and `remove()` of stack are called `push()` and `pop()`
- A stack is a pool with LIFO order on pushes and pops

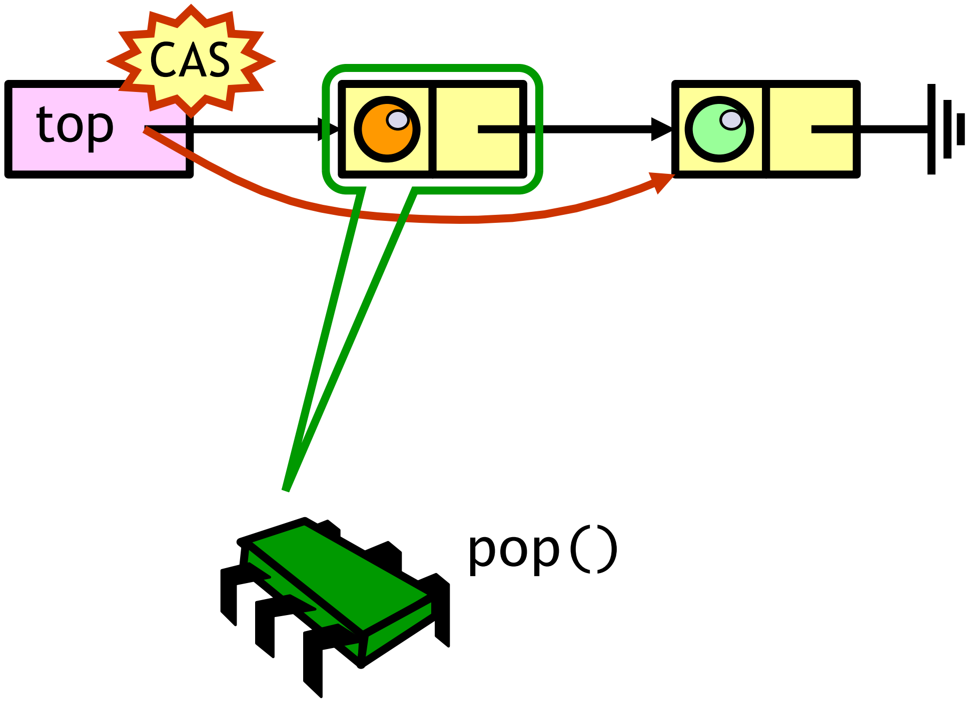
Unbounded Lock-Free Stack



Unbounded Lock-Free Stack



Unbounded Lock-Free Stack



Push

```
public class LockFreeStack {
  AtomicReference top = new AtomicReference();
  public boolean tryPush(Node node) {
    Node n = top.get();
    node.next = n;
    return top.compareAndSet(n, node);
  }
  public void push(T value) {
    Node node = new Node(value);
    while (!tryPush(node))
      backoff();
  }
}
```

Try to push node

Read top value…

…to be new node's successor

Try to swing top to point at new node

Create new node…

…then try to push (upon failure, back off before retrying)

Pop

```
public class LockFreeStack {
  …
  public T pop() {
    Node n;
    while (true) {
      if ((n = top.get()) == null)
        throw new EmptyStackException();
      if (top.compareAndSet(n, n.next))
        return n.value;
      backoff();
    }
  }
}
```

If stack is empty, throw exception

Try to pop top node

Success: return popped value

Failure: back off

Lock-Free Stack

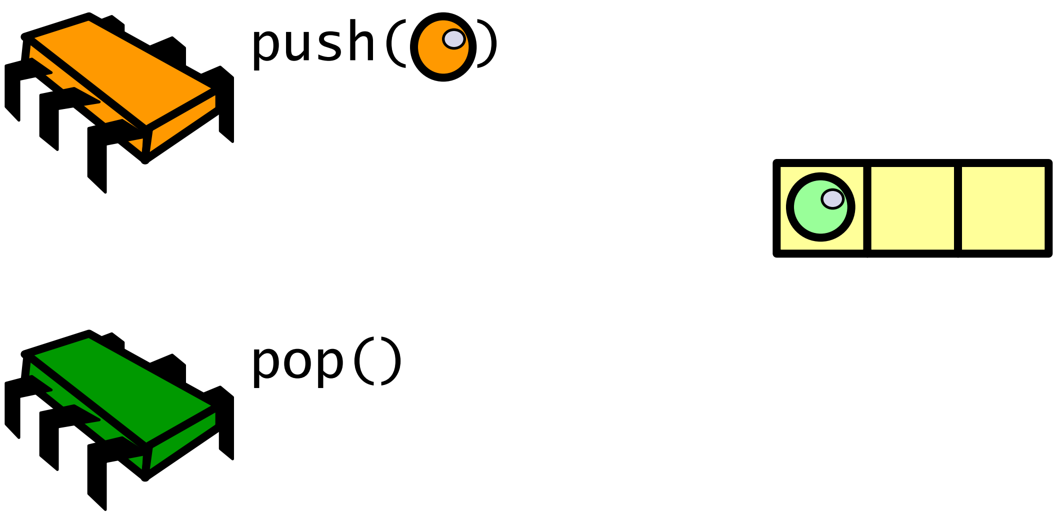
- Good
  - No locking
- Bad
  - Contention on top (add backoff)
  - No parallelism
  - ABA problem (more on that later)
- Is a stack inherently sequential?

Elimination-Backoff Stack

- How to "turn contention into parallelism"
  - Replace regular exponential-backoff…
  - …with an alternative elimination-backoff mechanism

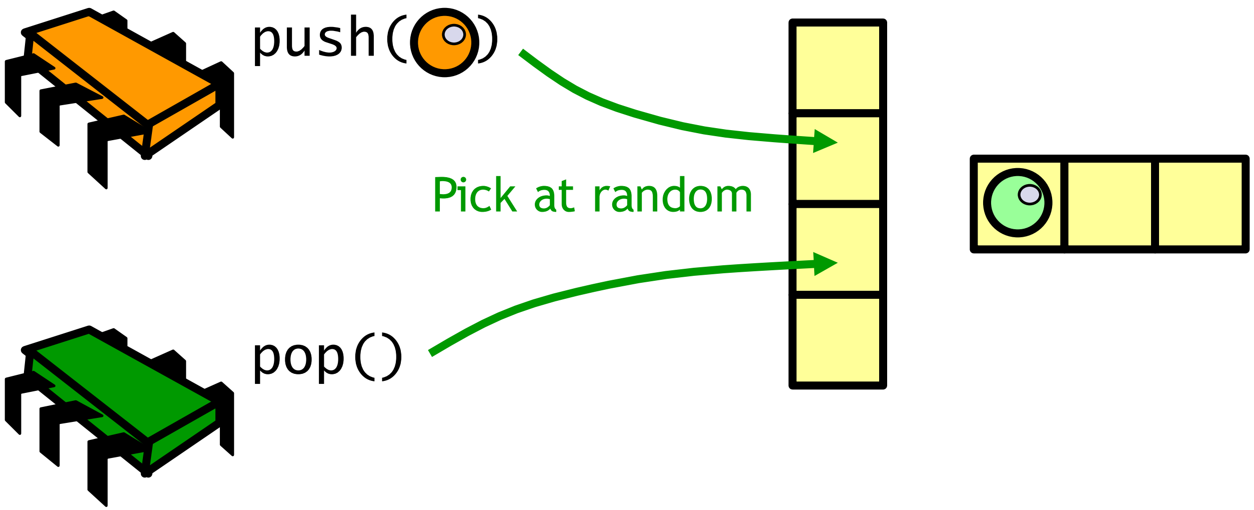
Observation

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



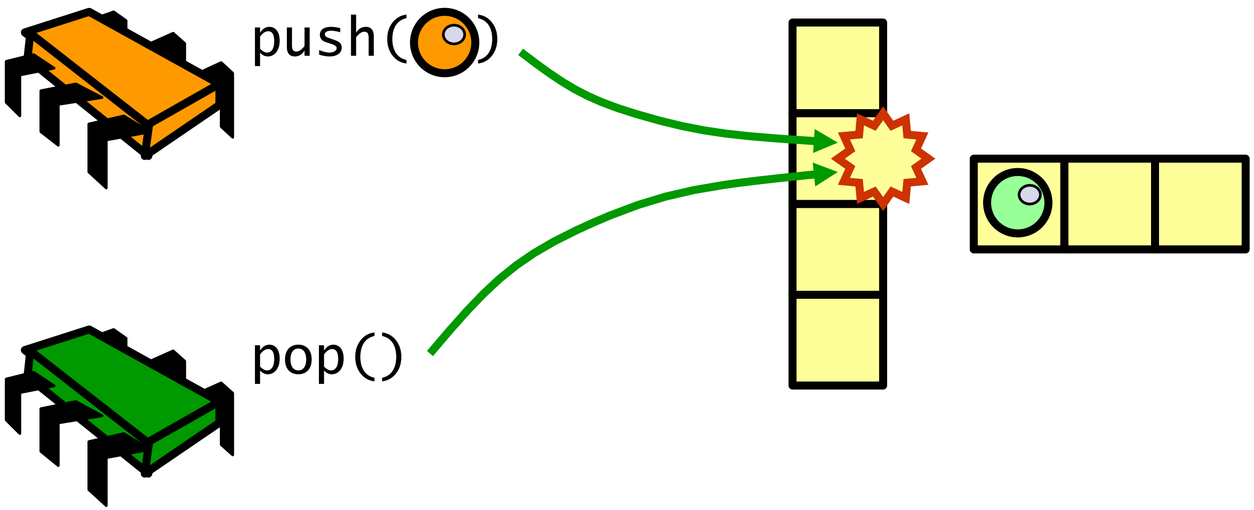
Idea: Elimination Array

- Pick a location at random in elimination array and try to match a push with a pop



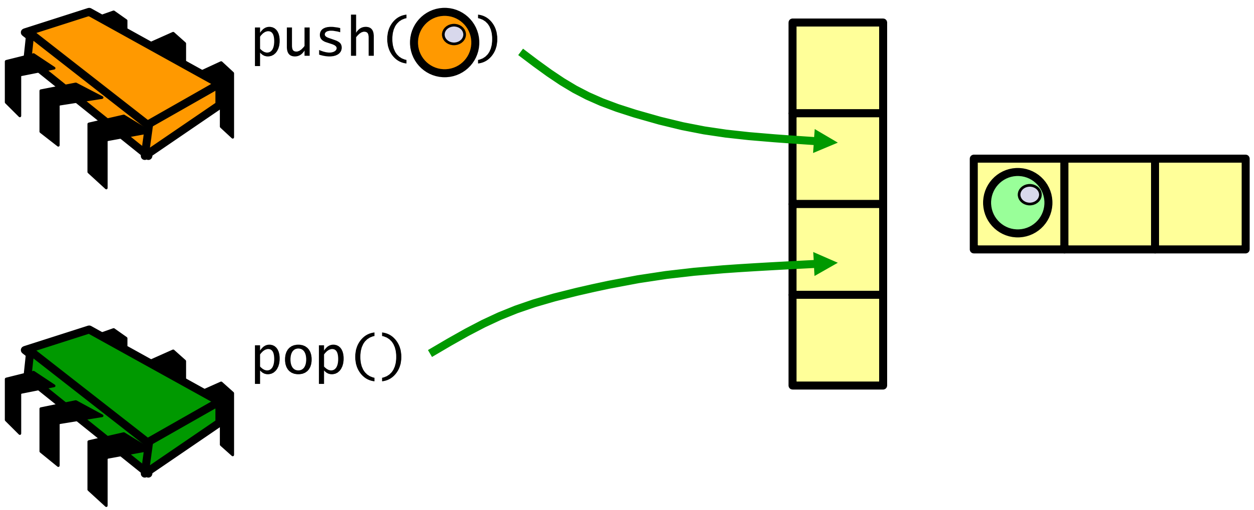
Push Collides With Pop

- If push collides with pop, no need to access the stack



Push Collides With Pop

- If push does not collide with pop, access the stack



Summary

- We saw both
  - Lock-based
  - Lock-free
- Implementations of
  - Queues
  - Stacks
- Do not be quick to declare a data structure inherently sequential
  - Elimination-backoff stack can exploit parallelism