Concurrent Queues and Stacks

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

The Five-Fold Path

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

Another Fundamental Problem

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

Queues & Stacks

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

Bounded vs Unbounded

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

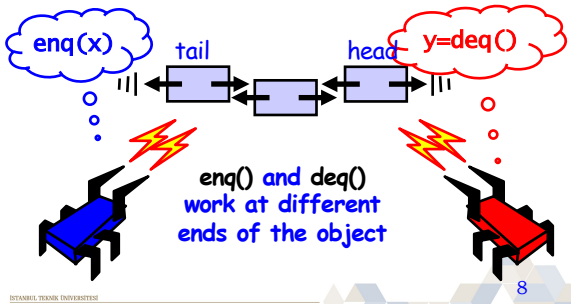
Blocking vs Non-Blocking

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

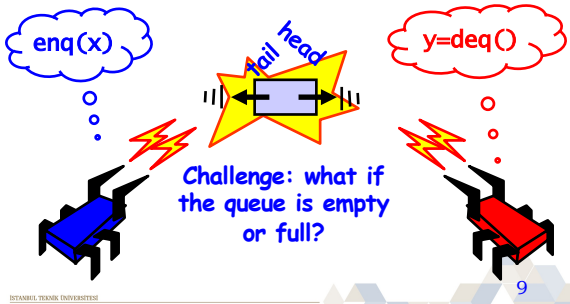
This Lecture

- Bounded, Blocking, Lock-based Queue
- Unbounded, Non-Blocking, Lock-free Queue
- Lost Wake-up problem
- Unbounded Non-Blocking Lock-free Stack
- Elimination-Backoff Stack

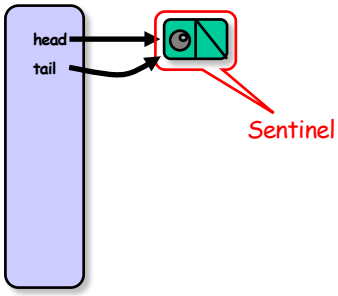
Queue: Concurrency



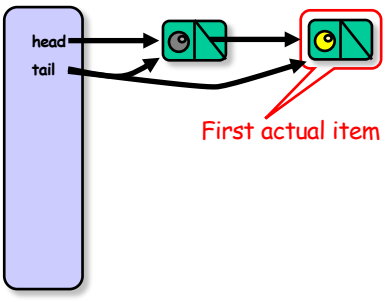
Concurrency



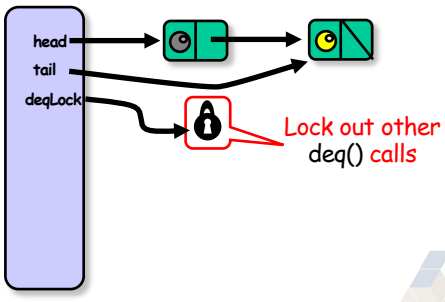
Bounded Queue



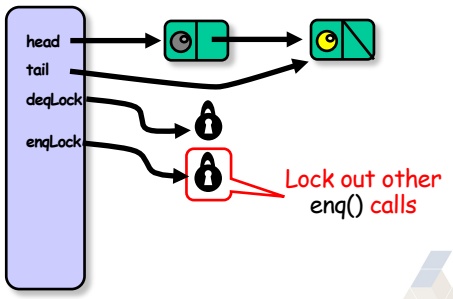
Bounded Queue



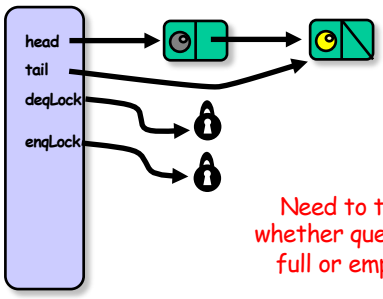
Bounded Queue



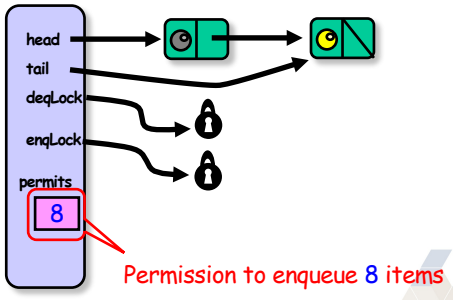
Bounded Queue



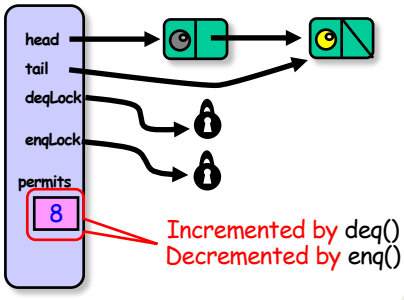
Not Done Yet



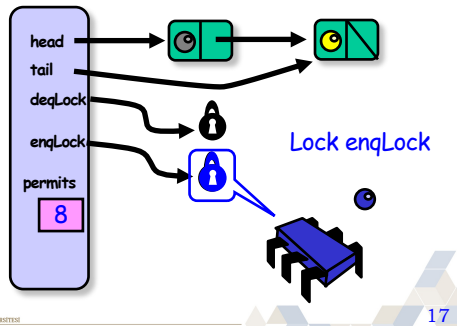
Not Done Yet



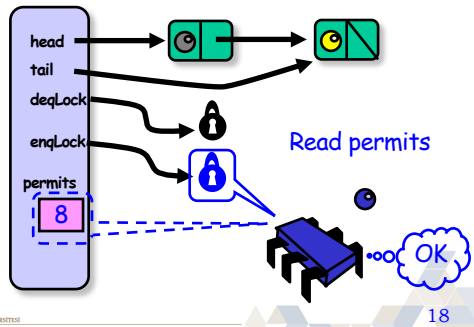
Not Done Yet



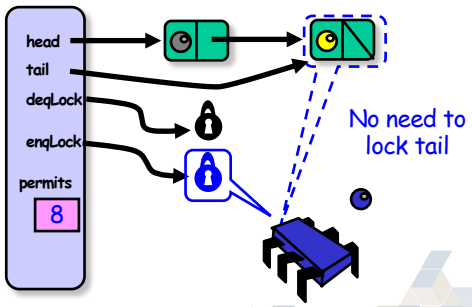
Enqueuer



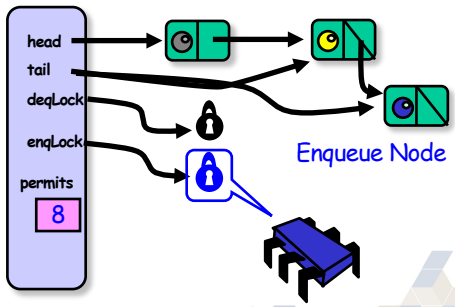
Enqueuer



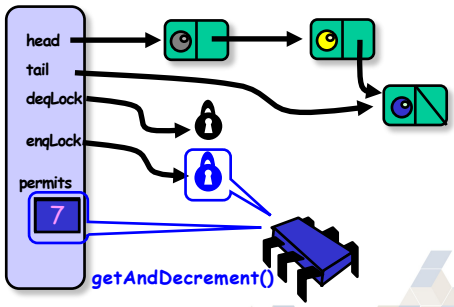
Enqueuer



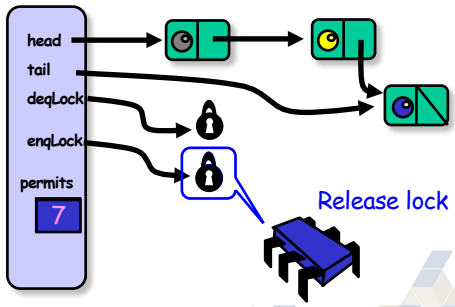
Enqueuer



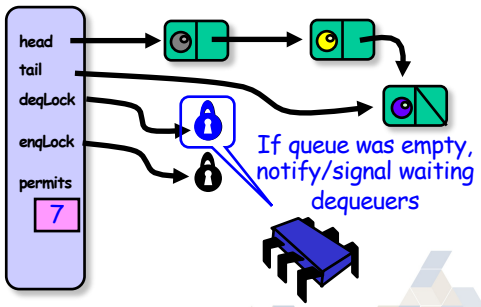
Enqueuer



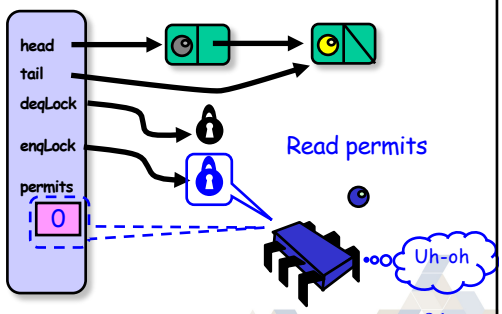
Enqueuer



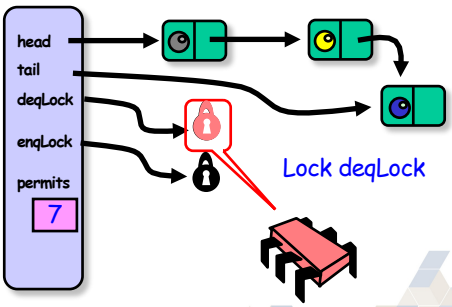
Enqueuer



Unsuccesful Enqueuer

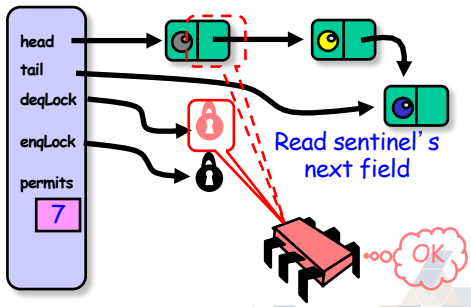


Dequeuer



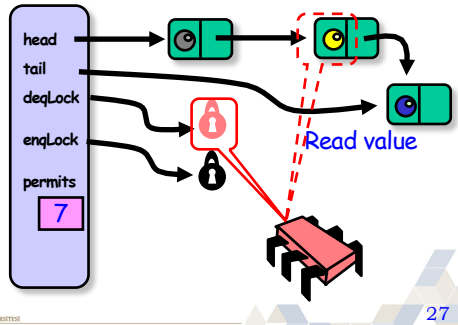
25

Dequeuer



26

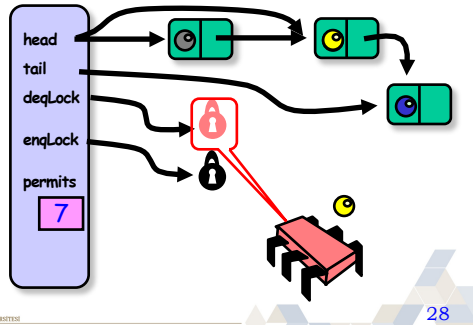
Dequeuer



27

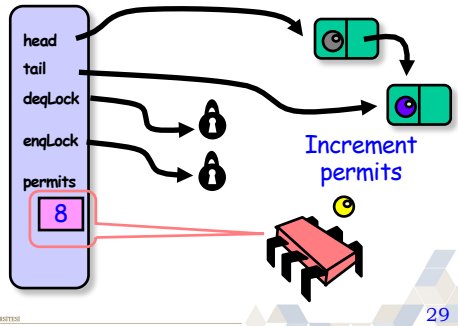
Dequeuer

Make first Node new sentinel



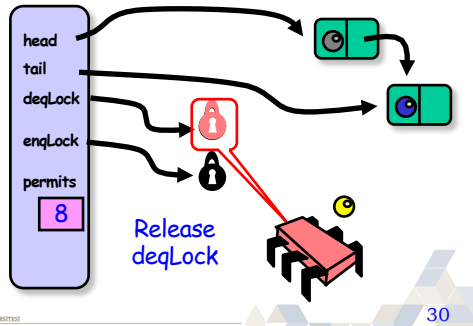
28

Dequeuer



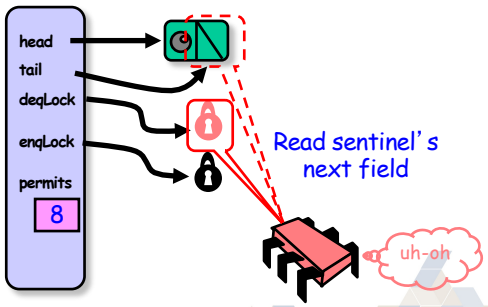
29

Dequeuer



30

Unsuccesful Dequeuer



Bounded Queue

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


Bounded Queue

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

Enq & deq locks

Digression: Monitor Locks

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

The Java Lock Interface

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

Acquire lock

The Java Lock Interface

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

Release lock

The Java Lock Interface

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

Try for lock, but not too hard

The Java Lock Interface

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

Create condition to wait on

The Java Lock Interface

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

Guess what this method does?

Lock Conditions

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


Lock Conditions

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

Release lock and wait on condition

Lock Conditions

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

Wake up one waiting thread

Lock Conditions

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

Wake up all waiting threads

Await

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

```
q.await()
```


Signal

- Awakens one waiting thread
  - Which will reacquire lock

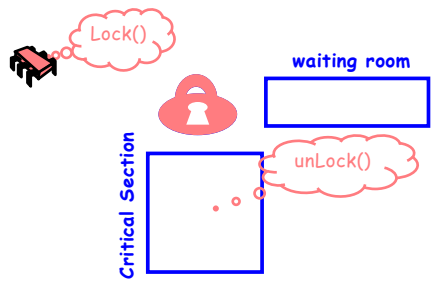
```
q.signal();
```


Signal All

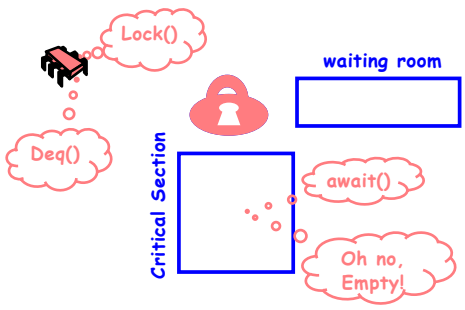
- Awakens all waiting threads
  - Which will each reacquire lock

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

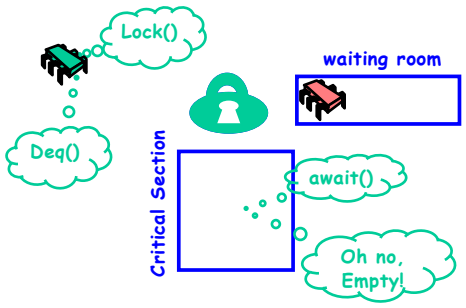

A Monitor Lock



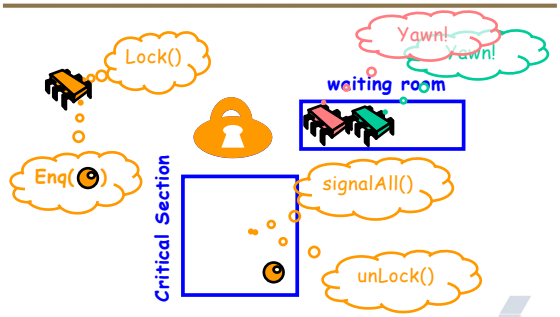
Unsuccessful Deq



Another One

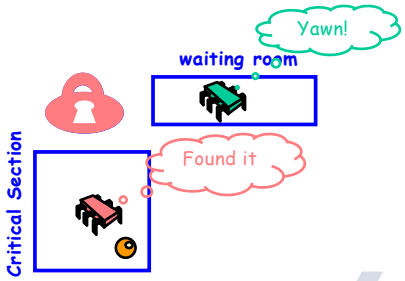


Enqueur to the Rescue



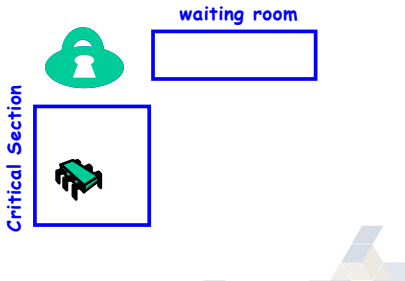
Monitor Signalling

Dequeurs Signalled

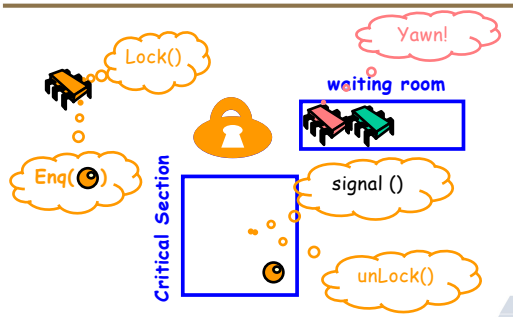


Dequeurs Signalled

Dollar Short + Day Late

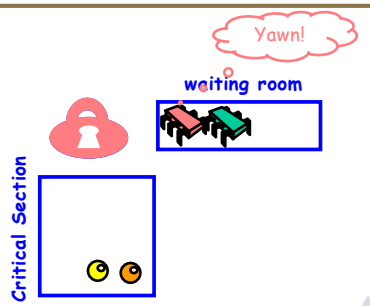


Lost Wake-Up

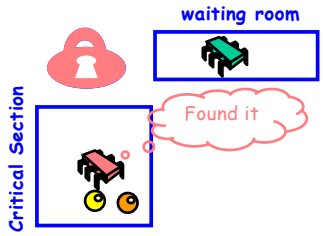


Lost Wake-Up

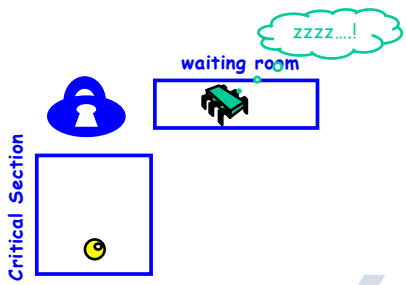
Lost Wake-Up



Lost Wake-Up



What's Wrong Here?



Solution to Lost Wakeup

- Always use signalAll and notifyAll
- Not signal and notify

Java Synchronized Methods

```
public class Queue<T> {

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

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


Java Synchronized Methods

```
public class Queue<T> {

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

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

Each object has an implicit lock with an implicit condition

Java Synchronized Methods

```
public class Queue<T> {

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

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

Lock on entry, unlock on return

Java Synchronized Methods

```
public class Queue<T> {

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

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

Wait on implicit condition

Java Synchronized Methods

```
public class Queue<T> {

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

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

Signal all threads waiting on condition

(Pop!) The Bounded Queue

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

Bounded Queue Fields

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

Enq & deq locks

Bounded Queue Fields

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

Enq lock's associated condition

Bounded Queue Fields

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

Num permits: 0 to capacity

Bounded Queue Fields

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

Head and Tail

Enq Method Part One

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


Enq Method Part One

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

Lock and unlock enq lock

Enq Method Part One

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

If queue is full, patiently await further instructions …

Be Afraid

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

How do we know the permits field won't change?

Enq Method Part One

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

Add new node

Enq Method Part One

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

If queue was empty, wake frustrated dequeuers

Enq Method Part Deux

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


Enq Method Part Deux

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

Are there dequeuers to be signaled?

Enq Method Part Deux

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

Lock and unlock deq lock

Enq Method Part Deux

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

Signal dequeuers that queue no longer empty

The Enq() & Deq() Methods

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

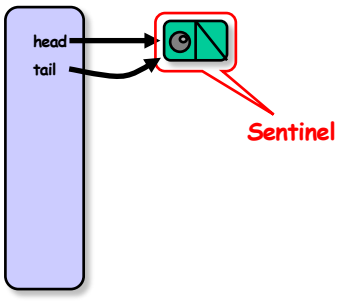
Split the Counter

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

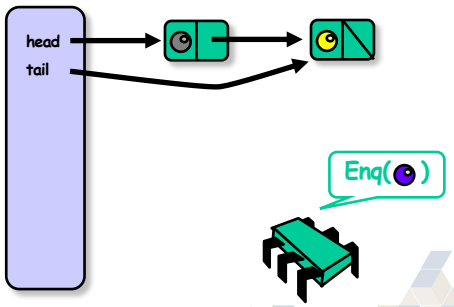
Split Counter

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

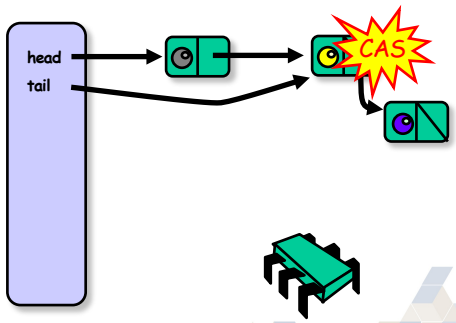
A Lock-Free Queue



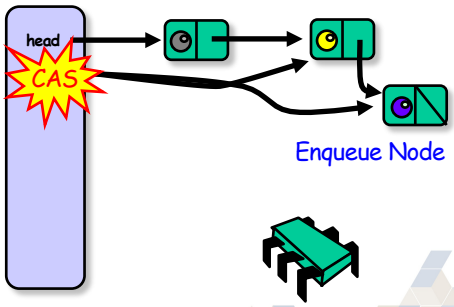
Enqueue



Logical Enqueue



Physical Enqueue



Enqueue

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

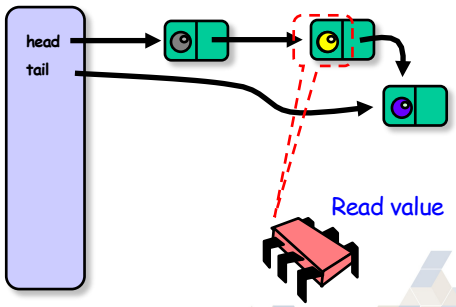
Enqueue

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

When CASs Fail

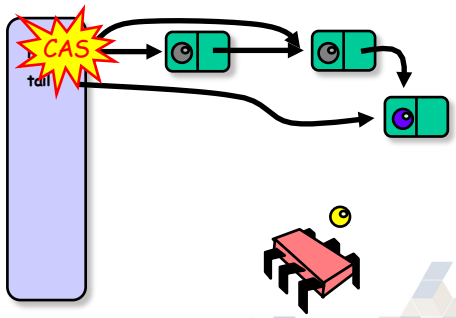
- During logical enqueue
  - Abandon hope, restart
  - Still lock-free
- During physical enqueue
  - Ignore it

Dequeuer



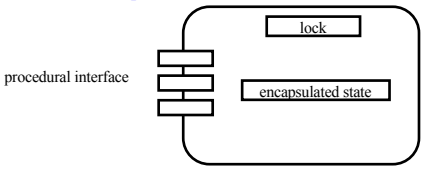
Dequeuer

Make first Node new sentinel



Monitors

- Communication and Synchronization
  - approaches broadly classified as shared-variable or message passing
  - many different models, a popular one is a monitor
  - a monitor can be considered as an object where each of its operation executes in mutual exclusion



Monitors

- Condition Synchronization
  - expresses a constraint on the ordering of execution of operations
  - e.g., data cannot be removed from a buffer until data has been placed in the buffer
- Monitors provide condition variables with two operations which can be called when the lock is held
  - wait: an unconditional suspension of the calling thread (the thread is placed on a queue associated with the condition variable)
  - notify: one thread is taken from the queue and re-scheduled for execution (it must reclaim the lock first)
  - notifyAll: all suspended threads are re-scheduled
  - **notify and notifyAll have no effect if no threads are suspended on the condition variable**

wait(), notify(), notifyAll()

wait() and notify() allows a thread to wait for an event

Instead of "busy wait" or sleep loop!

A call to notifyAll() allows all threads that are on wait() with the same lock to be released

A call to notify() allows one arbitrary thread that is on a wait() with the same lock to be released

wait(), notify(), notifyAll()

```
public class Drop {

  // Message sent from producer to consumer
  private String message;

  // A flag, True if consumer should wait for
  // producer to send message, False if producer
  // should wait for consumer to retrieve message
  private boolean empty = true;

  ...
```

Flag must be used, never count only on the notify

wait(), notify(), notifyAll()

```
public class Drop {
  ...
  public synchronized String take() {
    // Wait until message is available
    while (empty) {
      // we do nothing on InterruptedException
      // since the while condition is checked anyhow
      try { wait(); } catch (InterruptedException e) {}
    }
    // Toggle status and notify on the status change
    empty = true;
    notifyAll();
    return message;
  }
  ...
}
```


wait(), notify(), notifyAll()

```
public class Drop {
  ...
  public synchronized void put(String message) {
    // Wait until message has been retrieved
    while (!empty) {
      // we do nothing on InterruptedException
      // since the while condition is checked anyhow
      try { wait(); } catch (InterruptedException e) {}
    }
    // Toggle status, store message and notify consumer
    empty = false;
    this.message = message;
    notifyAll();
  }
  ...
}
```


The new `wait()` and `notify()`

- Remember **`Lock.newCondition();`** ?
- Condition factors out the Object monitor methods (**`wait`**, **`notify`** and **`notifyAll`**)
  - into distinct objects to give the effect of having multiple wait-sets per object
  - can combine them with the use of arbitrary **`Lock`** implementations.
- Where a **`Lock`** replaces the use of **`synchronized`** methods and statements, a **`Condition`** replaces the use of the Object monitor methods.

Conditions

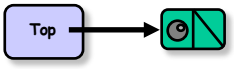
- Allows more than one wait condition per object
  - Even for built-in locks, via Locks utility class
- Allows much simpler implementation of

```
interface Condition {
  void await() throws IE;
  void awaitUninterruptibly();
  long awaitNanos(long nanos) throws IE;
  boolean awaitUntil(Date deadline) throws IE;
  void signal();
  void signalAll();
}
```

Concurrent Stack

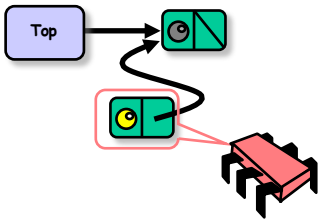
- Methods
  - push(x)
  - pop()
- Last-in, First-out (LIFO) order
- Lock-Free!

Empty Stack



Push

İTÜ

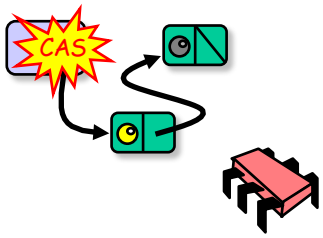


İSTANBUL TEKNİK ÜNİVERSİTESİ

103

Push

İTÜ

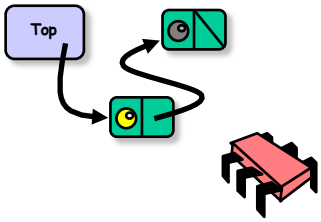


İSTANBUL TEKNİK ÜNİVERSİTESİ

104

Push

İTÜ

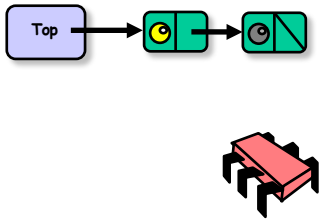


İSTANBUL TEKNİK ÜNİVERSİTESİ

105

Push

İTÜ

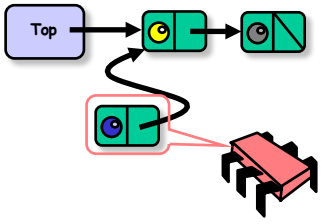


İSTANBUL TEKNİK ÜNİVERSİTESİ

106

Push

İTÜ

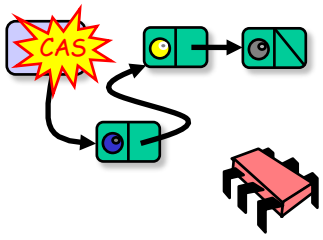


İSTANBUL TEKNİK ÜNİVERSİTESİ

107

Push

İTÜ

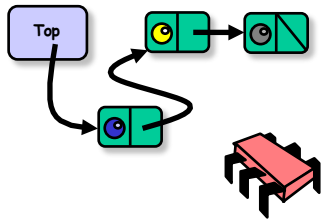


İSTANBUL TEKNİK ÜNİVERSİTESİ

108

Push

İTÜ

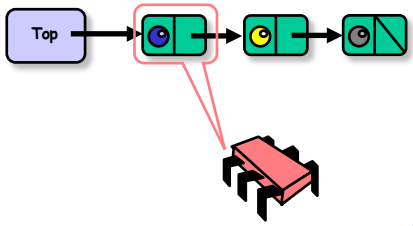


İSTANBUL TEKNİK ÜNİVERSİTESİ

109

Pop

İTÜ

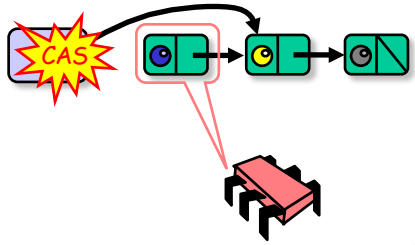


İSTANBUL TEKNİK ÜNİVERSİTESİ

110

Pop

İTÜ

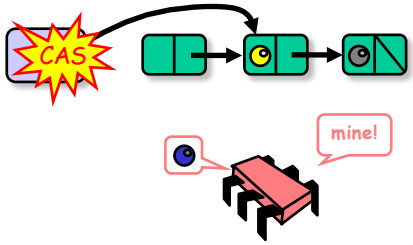


İSTANBUL TEKNİK ÜNİVERSİTESİ

111

Pop

İTÜ

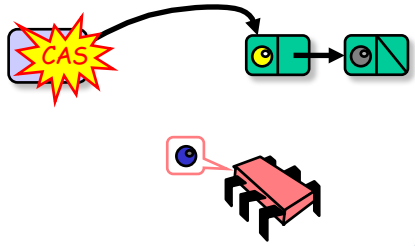


İSTANBUL TEKNİK ÜNİVERSİTESİ

112

Pop

İTÜ

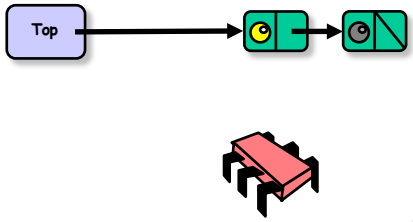


İSTANBUL TEKNİK ÜNİVERSİTESİ

113

Pop

İTÜ



İSTANBUL TEKNİK ÜNİVERSİTESİ

114

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new AtomicReference(null);

  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }

  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      }
      else backoff.backoff();
  }}
```


Lock-free Stack

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

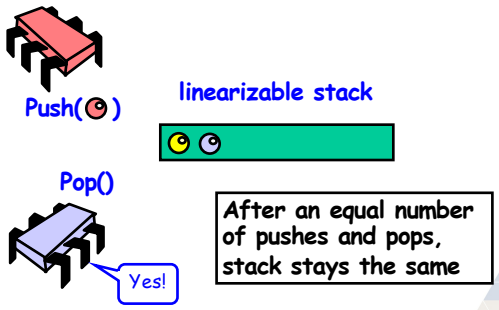
Big Question

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

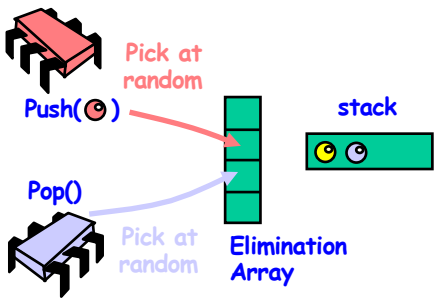
Elimination-Backoff Stack

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

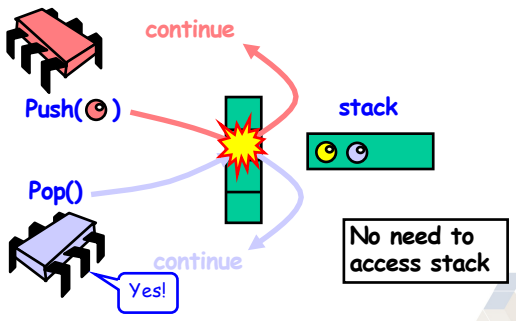
Observation



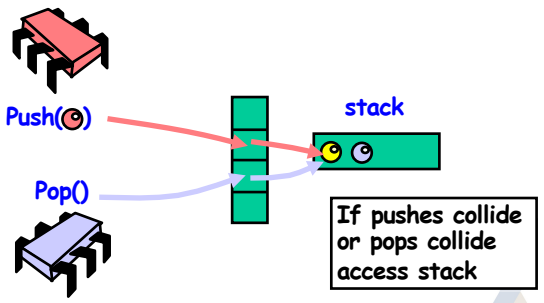
Idea: Elimination Array



Push Collides With Pop



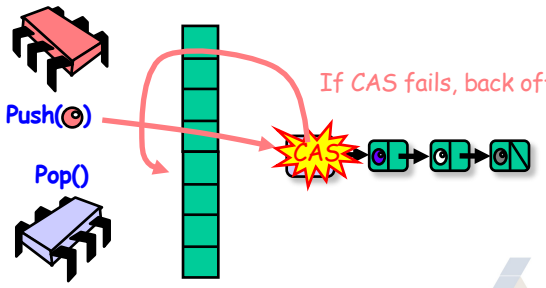
No Collision



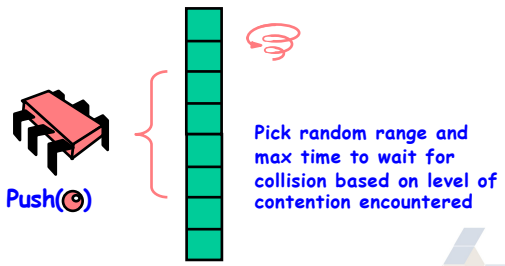
Elimination-Backoff Stack

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

Elimination-Backoff Stack



Dynamic Range and Delay



Linearizability

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

Backoff Has Dual Effect

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

Elimination Array

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


A Lock-Free Exchanger

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


Extracting Reference & Stamp

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

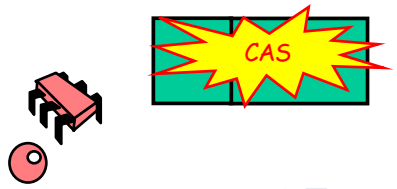

Exchanger Status

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

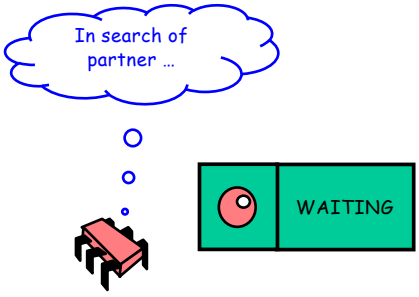

The Exchange

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

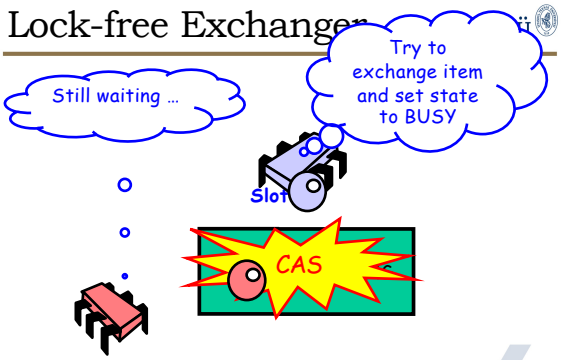
Lock-free Exchanger



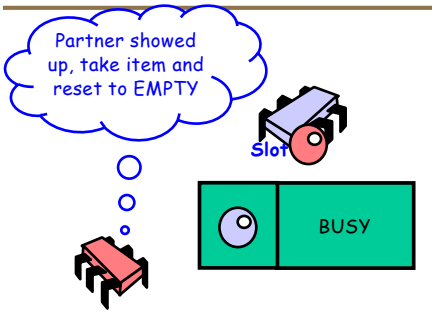
Lock-free Exchanger



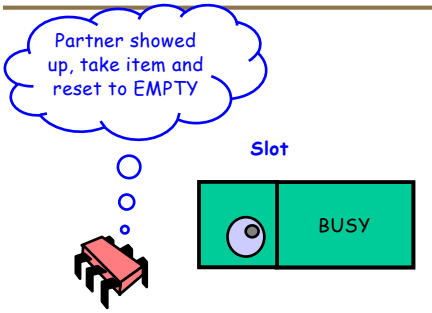
Lock-free Exchanger



Lock-free Exchanger



Lock-free Exchanger



Exchanger State EMPTY

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

States WAITING and BUSY

```
case WAITING:   // someone waiting for me
    if (slot.compareAndSet(herItem, myItem, WAITING, BUSY))
        return herItem;
break;

case BUSY:   // others in middle of exchanging
break;

default:  // impossible
break;

}}}}
```


The Exchanger Slot

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

Elimination Array

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


Elimination Stack Push

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


Elimination Stack Pop

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


Exchangers

- A synchronization point at which two threads can exchange objects
- Can be thought of as a CyclicBarrier with a count of two plus allowing an exchange of state at the barrier
- On calling the exchange() method the thread provides some object as input
- The exchange() methods returns the object entered as input by the second thread to the calling thread
- Useful for example in the case where one thread is filling a buffer (filler thread) and the other emptying (emptying thread)
- On calling exchange() the filler thread is passed an empty buffer
- The emptying thread obtain the newly full buffer from the filler thread

Concurrent Collections

The Concurrency framework provides implementation of several commonly used collections classes optimized for concurrent operations

`public interface Queue<E> extends Collection<E>:`

A collection class that hold elements prior to processing, generally orders element in a FIFO manner (however can also be LIFO etc).

Supports the following functionality

- Offer method inserts an element if possible else return false
- remove() and poll() return and remove the head of the queue
- element() and peek() return but do not remove the head of the queue

Concurrent Collections

The Concurrency framework queue implementations

- CopyOnWriteArrayList: A thread-safe variant of ArrayList
- ConcurrentLinkedQueue : An unbounded thread-safe queue based on linked nodes.
- *BlockingQueue : A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.*

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Concurrent Collections

The Concurrency framework queue implementations

- *BlockingQueue*
  - ArrayBlockingQueue: A bounded blocking queue backed by an array.
  - DelayQueue: An element can only be taken when its delay has expired. Holds elements of *Delayed* instances.
  - LinkedBlockingDeque: An optionally-bounded blocking deque based on linked nodes.
  - LinkedTransferQueue: When transfer is used producing threads may wait until consumed.
  - PriorityBlockingQueue: An unbounded blocking queue that uses comparators for prioritizing.
  - SynchronousQueue: A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

Concurrent Collections

`public interface ConcurrentMap<K,V> extends Map<K,V>:`

An extension to the Map interface that provides support for concurrency

- Supports concurrent putIfAbsent, remove and replace methods
- putIfAbsent(key, value): If the specifed key is not associated with a value associate it with a value. Performed atomically
- remove(key, value): Removes entry for a key if associated with value. Performed atomically
- remove(key, oldvalue, newvalue): Replaces entry for a key if associated with oldvalue with newvalue. Performed atomically

Concurrent Collections

The Concurrency framework queue implementations

- *ConcurrentMap*
  - ConcurrentHashMap: A hash table supporting full concurrency of retrievals and high expected concurrency for updates.
  - ConcurrentNavigableMap: A ConcurrentMap supporting NavigableMap operations, and recursively so for its navigable sub-maps
  - ConcurrentSkipListMap: A scalable concurrent ConcurrentNavigableMap implementation.

Concurrent Collections

```
public class Consumerthread implements Runnable{
   private static int capacity ;
   private BlockingQueue<Integer> intqueue;
   public consumerthread(BlockingQueue<Integer> queue, int cap){
       capacity = cap;
       this.intqueue = queue;
   }
   public void run(){
      int num;
      for(int i = 0; i<capacity; i++){
        try {
           num = intqueue.take();
           if(num == -1) break;
           System.out.println("Square of " + num + ":" + num*num);
        } catch (InterruptedException e) {
              e.printStackTrace();
         }
      }}}
```

Concurrent Collections

```
public class Producerthread implements Runnable {
private static int capacity ;
private BlockingQueue<Integer> intqueue;
public producerthread(BlockingQueue<Integer> queue, int cap){
   capacity = cap;
   this.intqueue = queue;
}
public void run(){
   for(int i = 0; i<capacity-1; i++){
      try {
       intqueue.put(i);
      } catch (InterruptedException e) {
       e.printStackTrace();}
   }
   try {
      intqueue.put(-1);
   } catch (InterruptedException e) {
      e.printStackTrace();}
}}
```

Concurrent Collections

```
public class Tester {
   public static void main(String [] args){
        BlockingQueue<Integer> queue = new
        ArrayBlockingQueue<Integer>(100);

        Consumerthread consumer = new Consumerthread(queue, 100);

        Producerthread producer = new Producerthread(queue, 100);

        new Thread(consumer).start();

        new Thread(producer).start();

   }
}
```

Concurrent Collections

That was an example of using blocking queues to implement producer consumer relationships

- The producer class fills in a queue of integers
- The consumer class pulls integers of this queue and finds the square
- If the queue is empty a take() blocks, if full a put() blocks
- We could extend this to use a thread pool for the producers

Summary

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