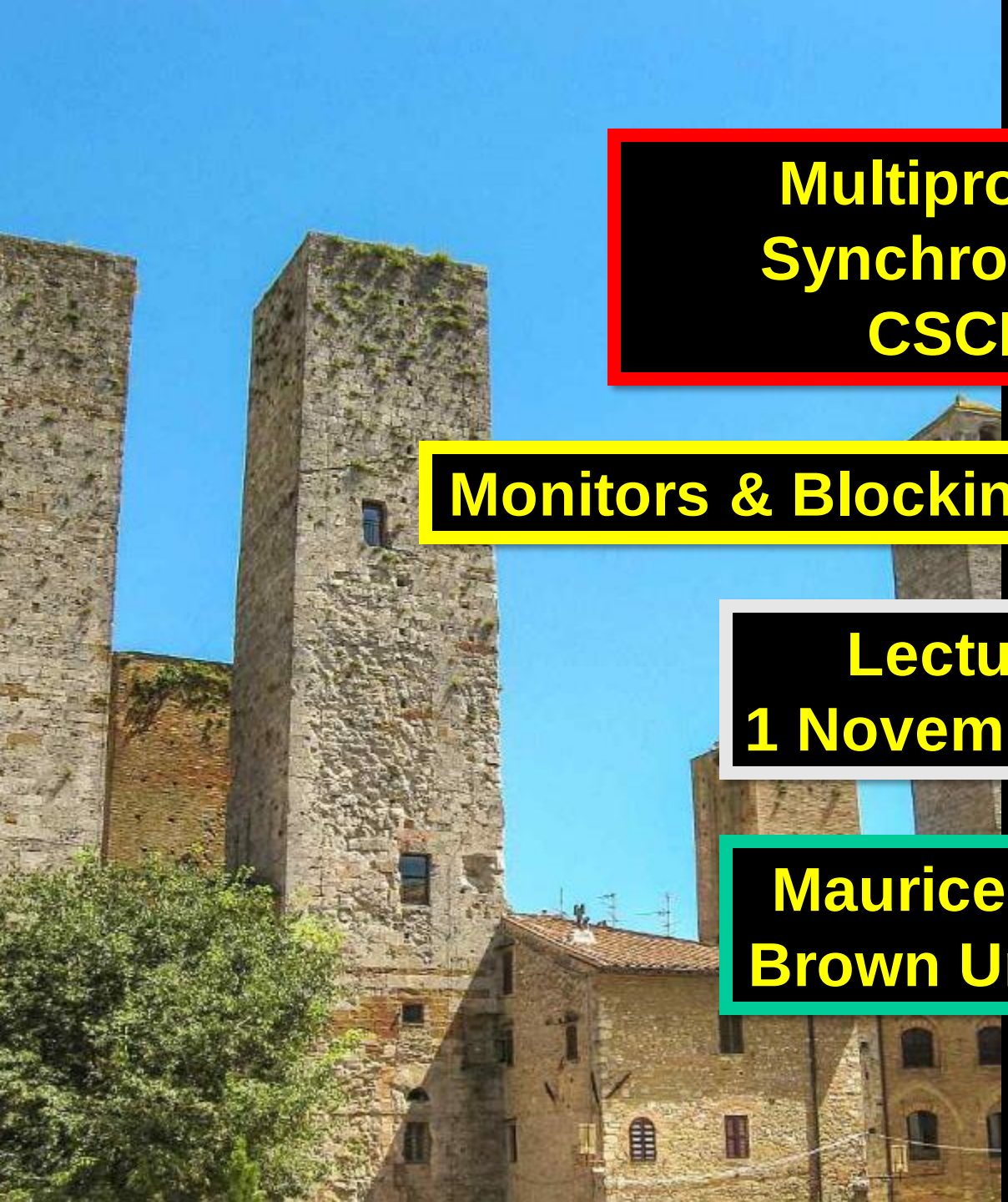


Multiprocessor Synchronization CSCI 176

Monitors & Blocking Synchronization

Lecture 15
1 November 2022

Maurice Herlihy
Brown University

Monitor Locks

Java `synchronized` objects & `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

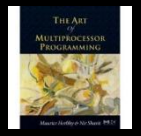
The Java Lock Interface

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

Never mind what this method does

Lock Conditions

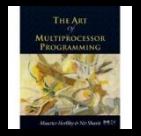
```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```



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

```
q.await()
```

Releases lock associated with q

Sleeps (gives up processor)

Awakens (resumes running)

Reacquires lock & returns

Signal

```
q.signal()
```

Awakens one waiting thread

Which must reacquire lock before running

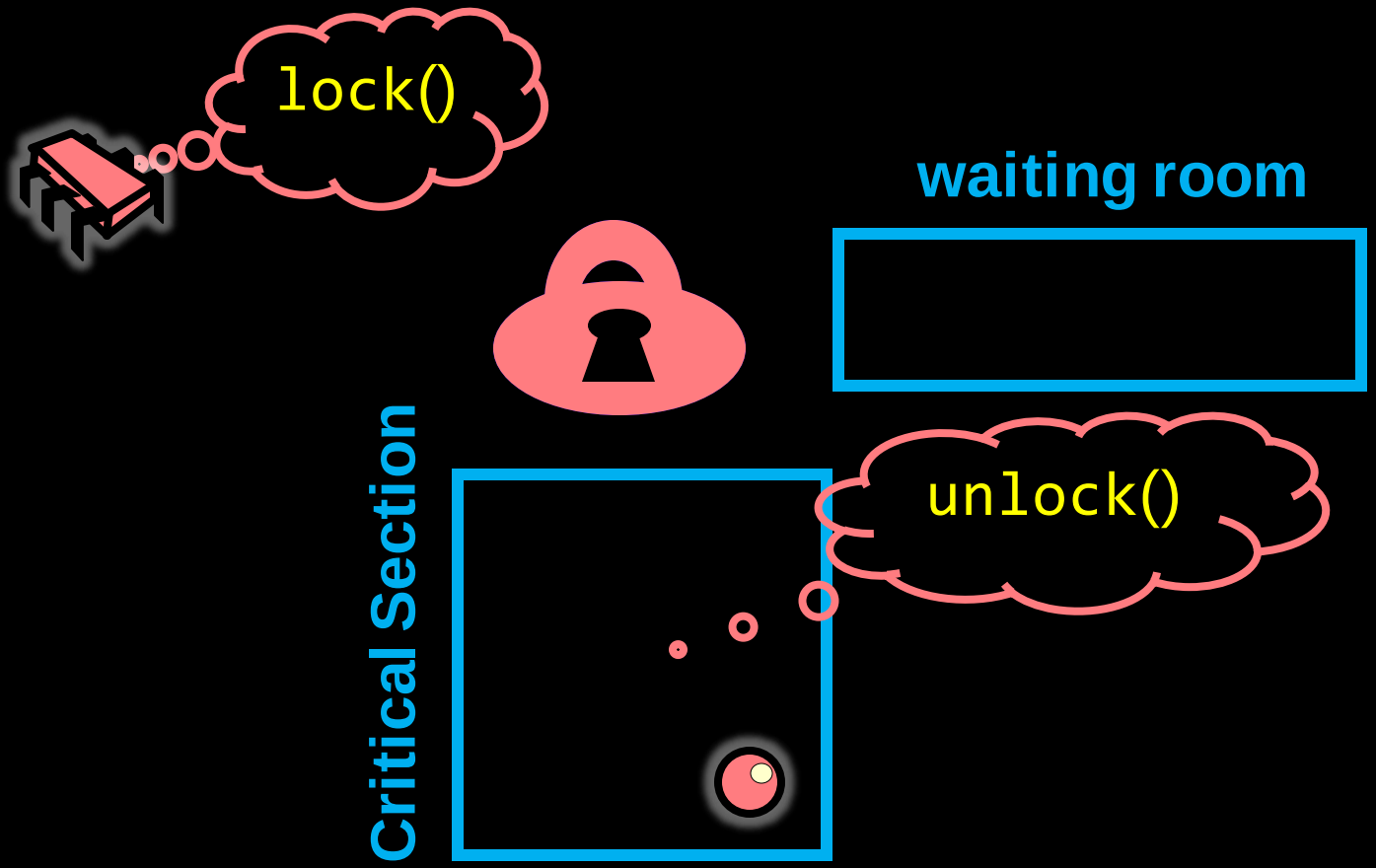
Signal All

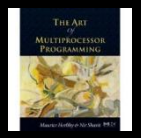
```
q.signalAll()
```

Awakens *all* waiting threads

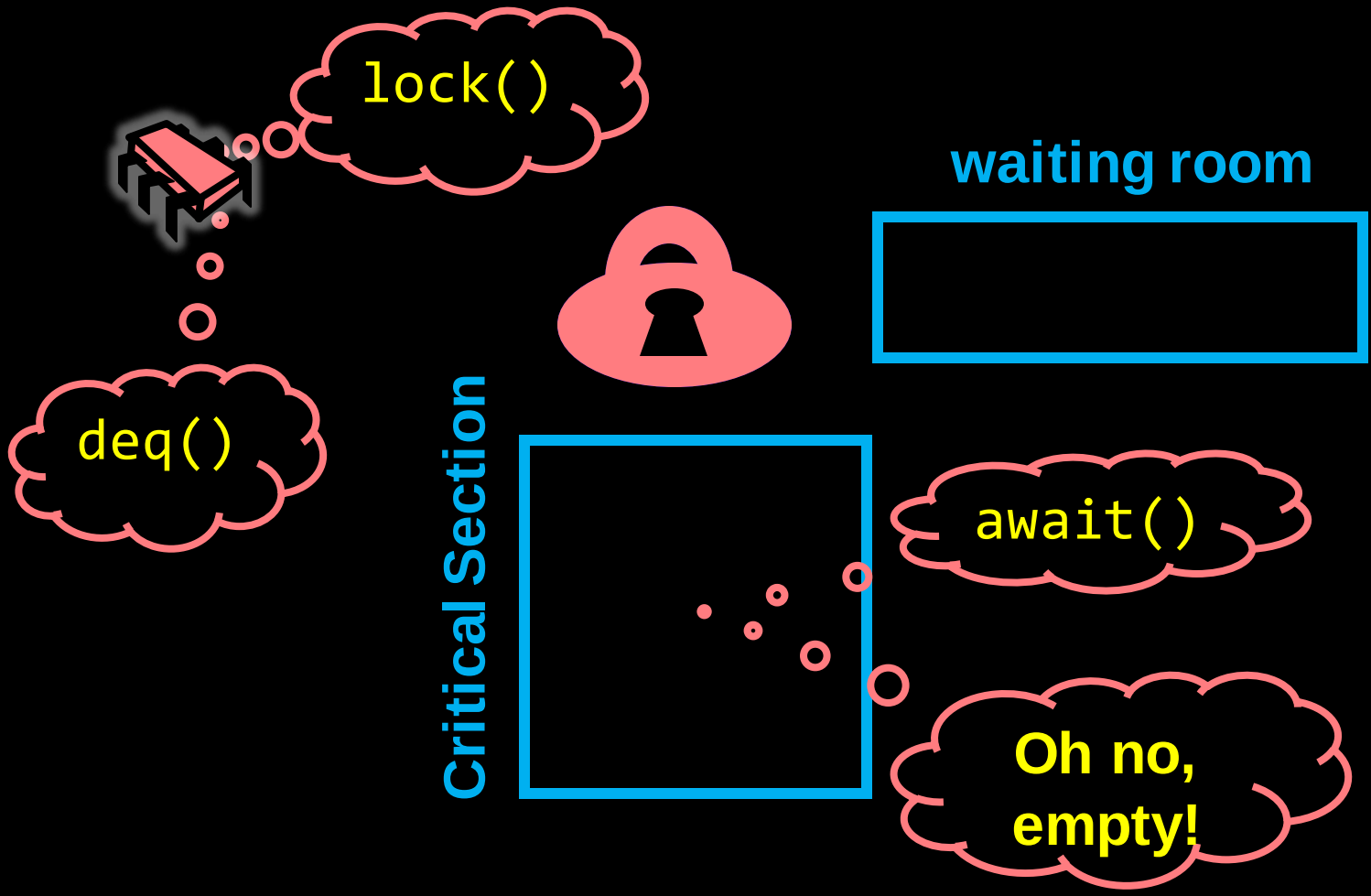
Which must each reacquire lock before running

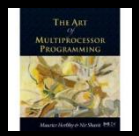
A Monitor Lock for a Queue



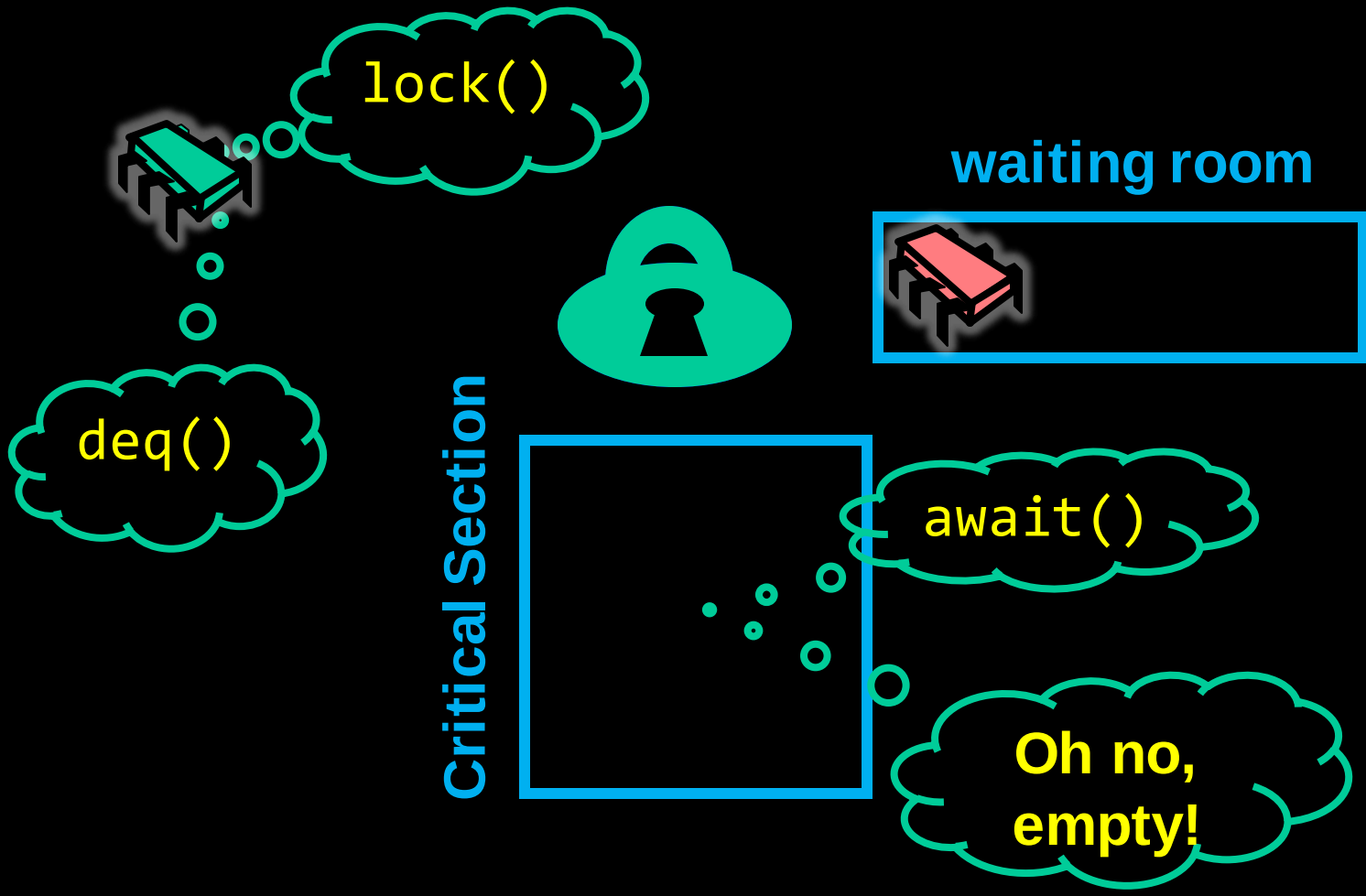


Unsuccessful Deq

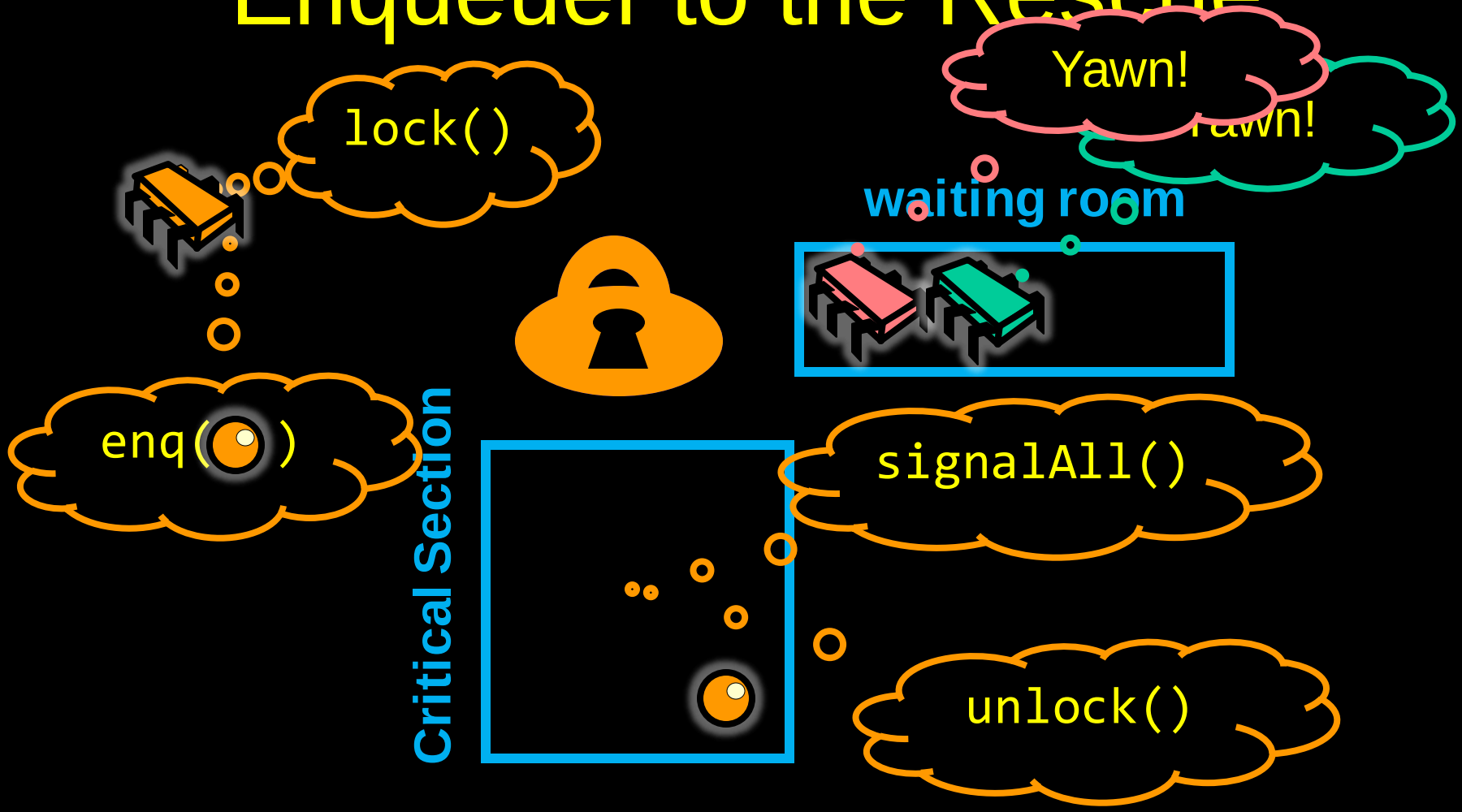


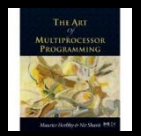


Another One

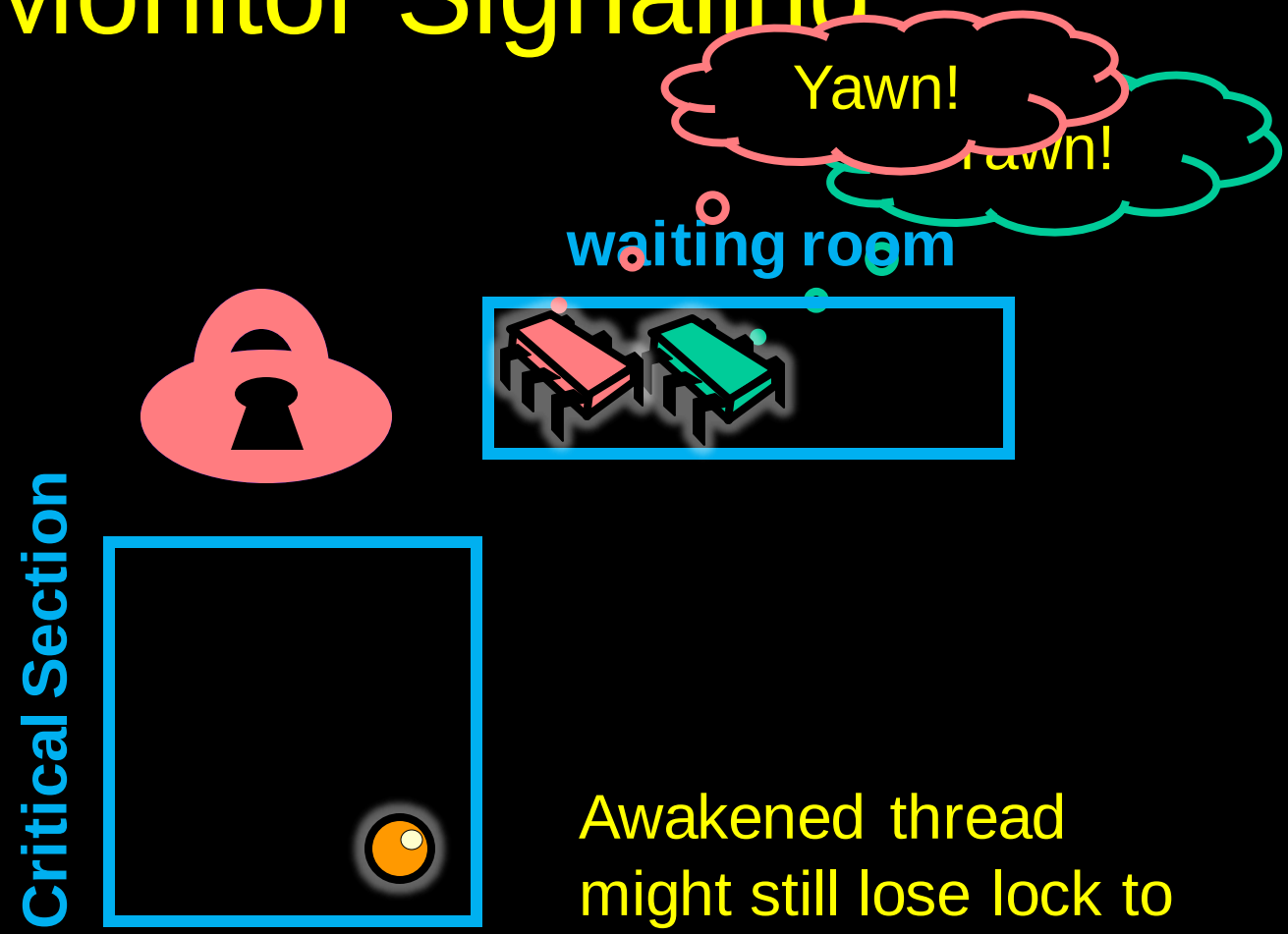


Enqueuer to the Rescue



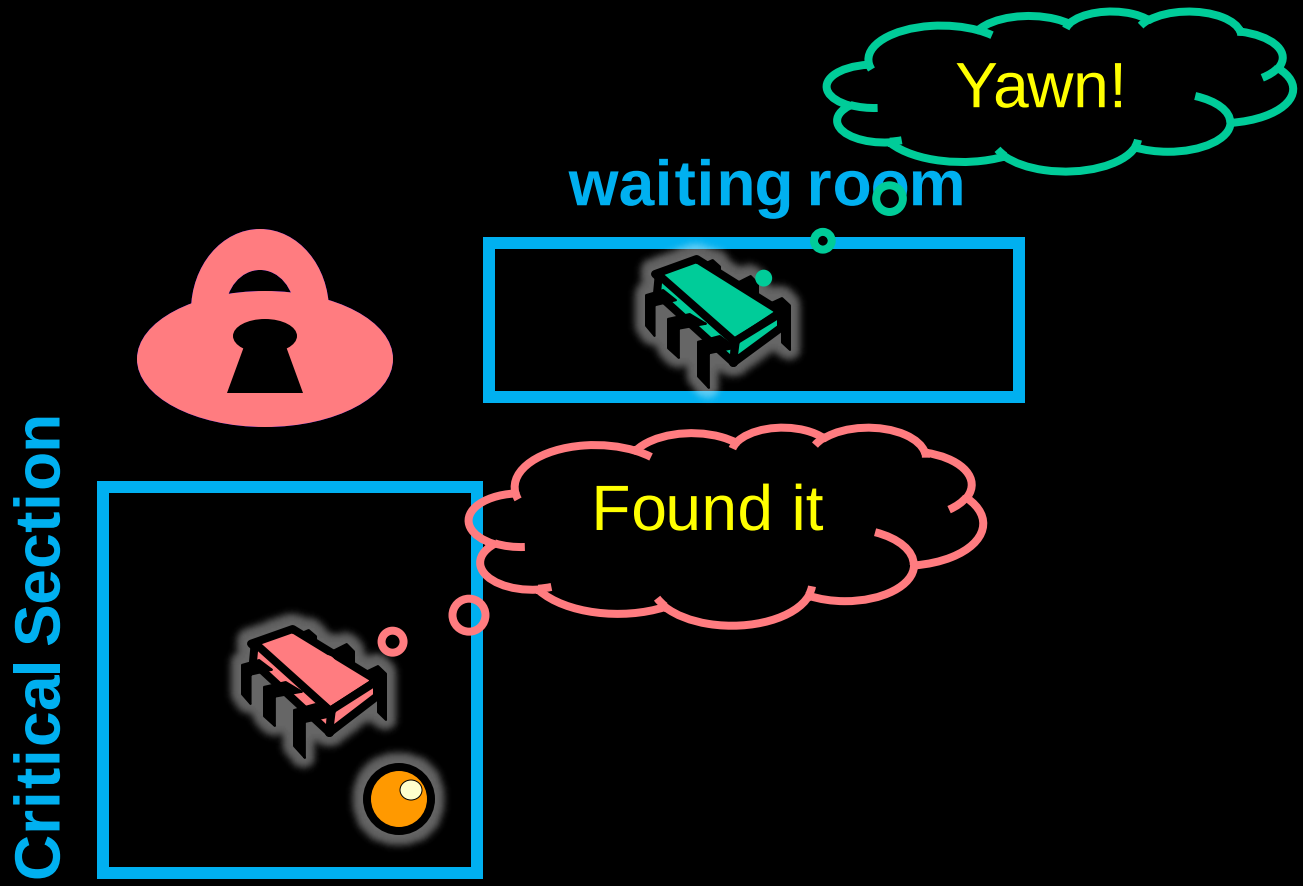


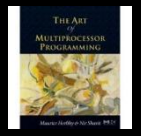
Monitor Signaling



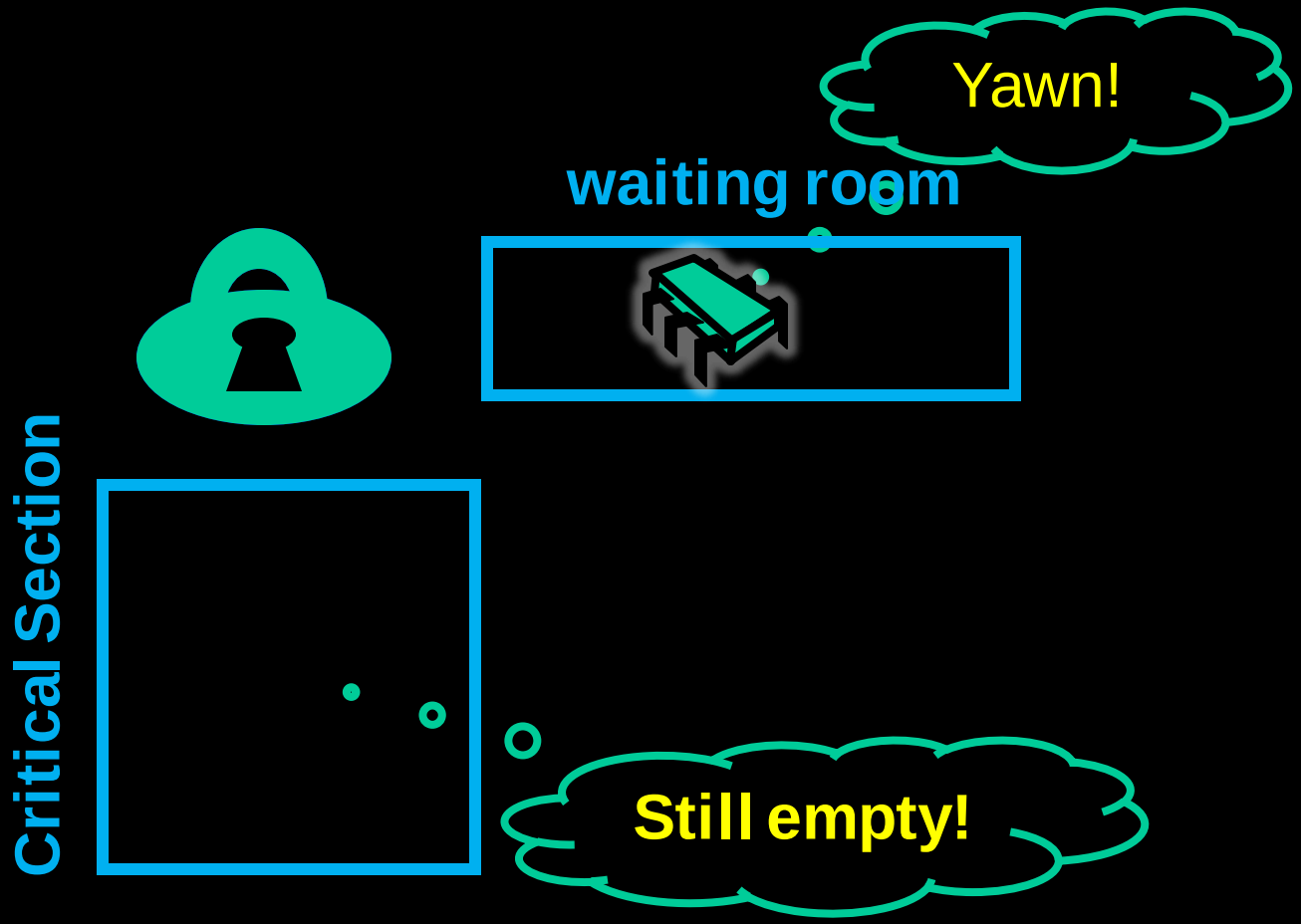
Awakened thread might still lose lock to outside contender…

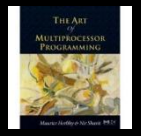
Dequeuers Signaled



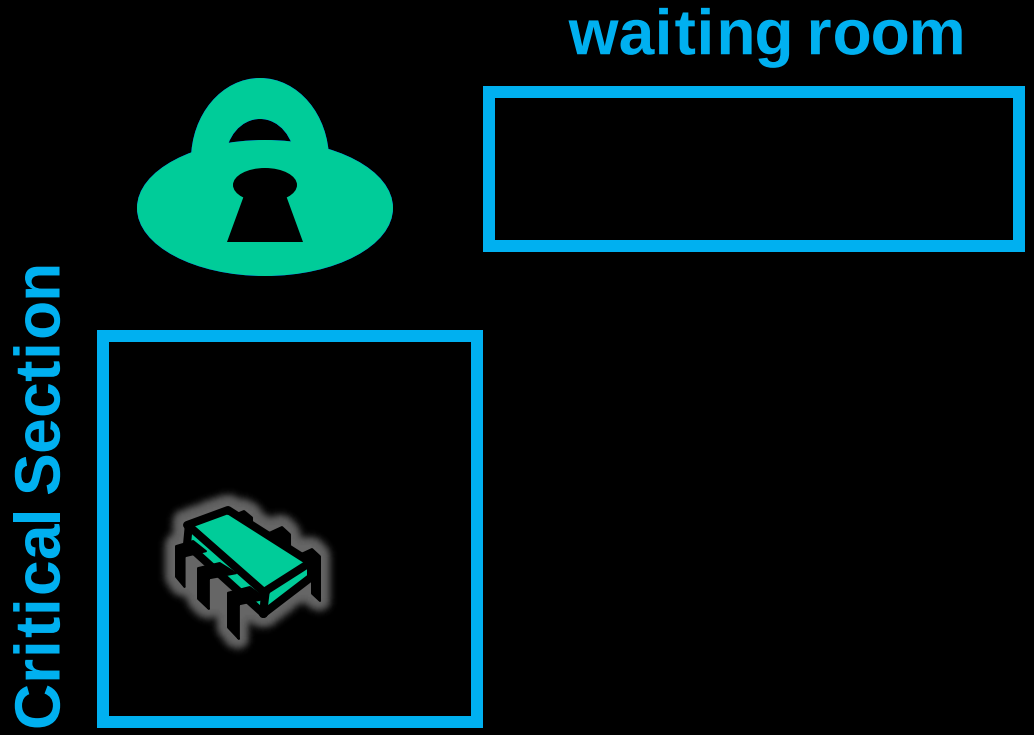


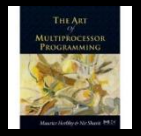
Dequeuers Signaled





Dollar Short + Day Late





Java Synchronized Methods

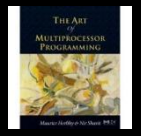
```
public class Queue<T> {

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

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

Art of Multiprocessor Programming

Java Synchronized Methods

```
public class Queue<T> {

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

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

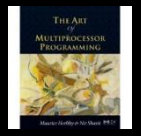
Each object has an implicit lock with an implicit condition



Java Synchronized Methods

```
public class Queue<T> {

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

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

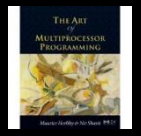
Lock on entry, unlock on return



Java Synchronized Methods

```
public class Queue<T> {

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

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

Wait on implicit condition

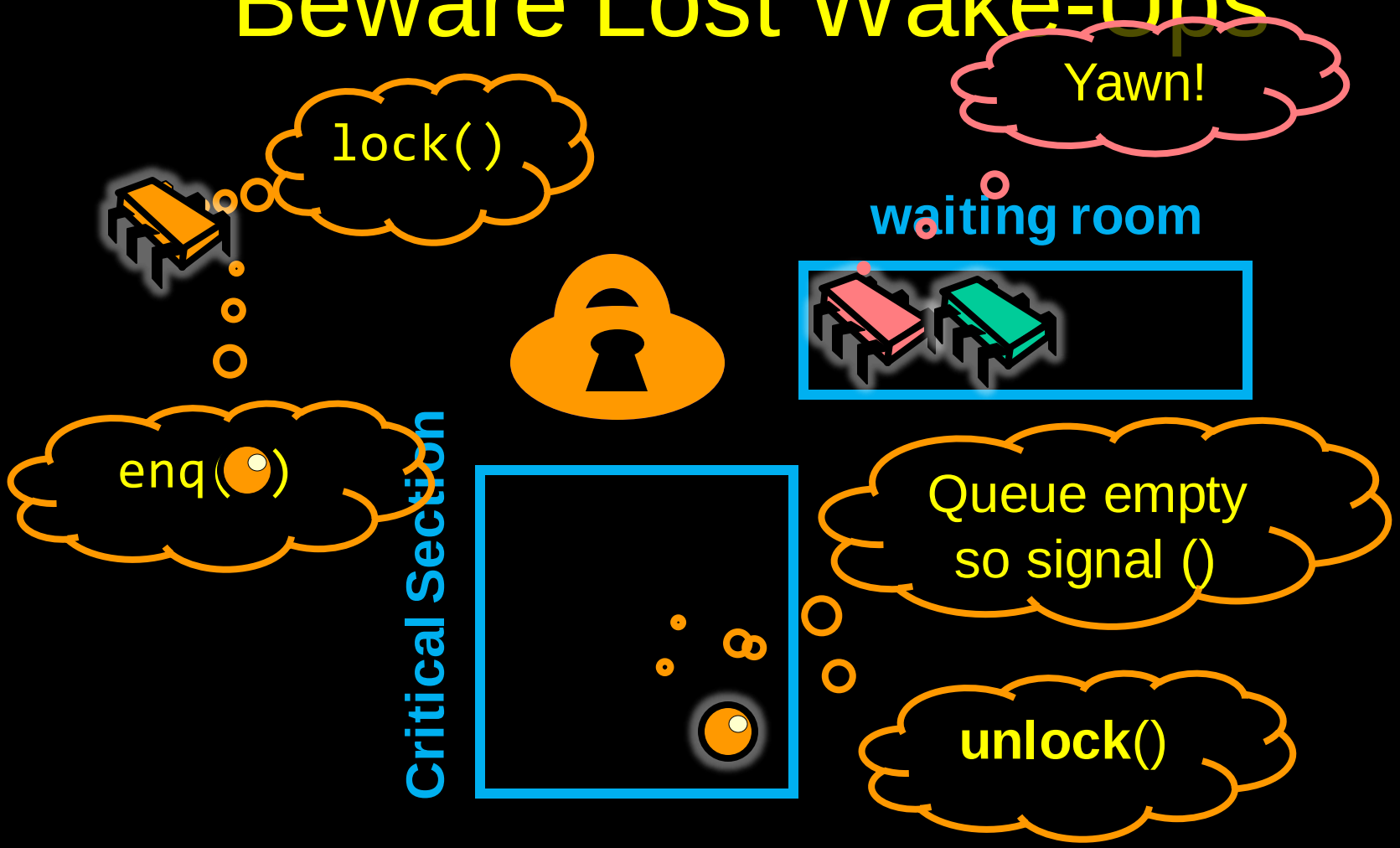


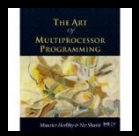
Java Synchronized Methods

```
public class Queue<T> {

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

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

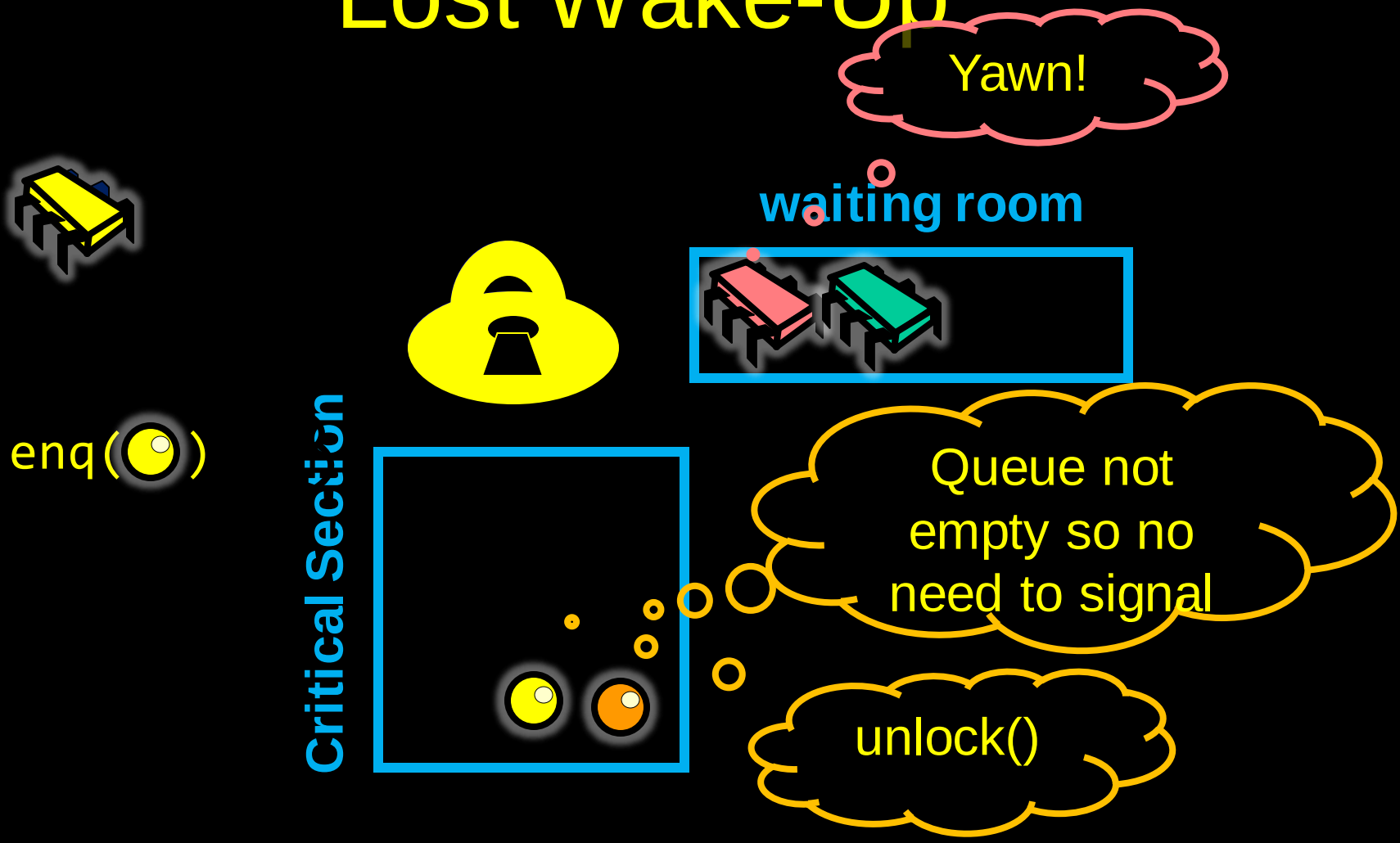
Signal all threads waiting on condition

Beware Lost Wake-Ups

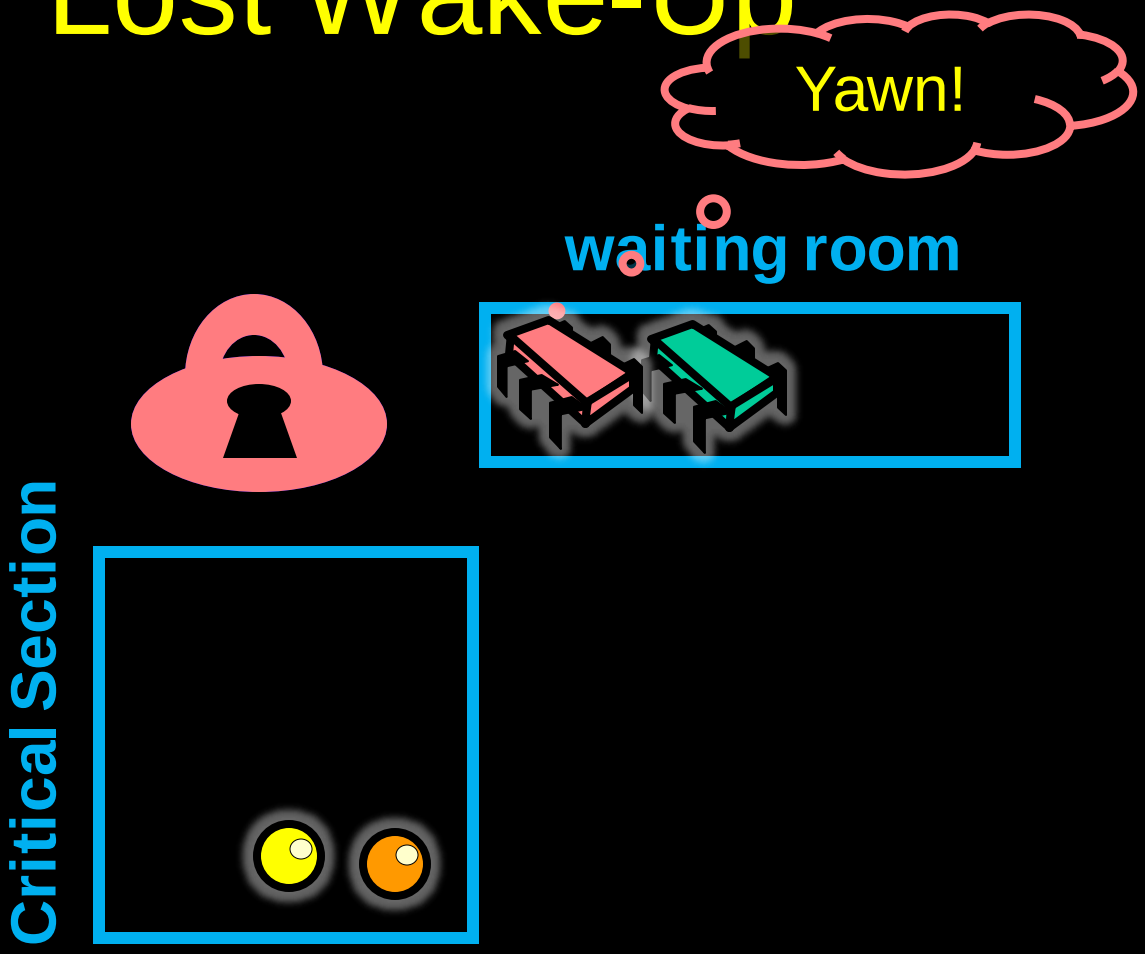




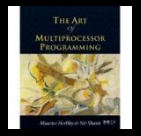
Lost Wake-Up


Yawn!
waiting room
enq()
Critical Section
Queue not empty so no need to signal
unlock()

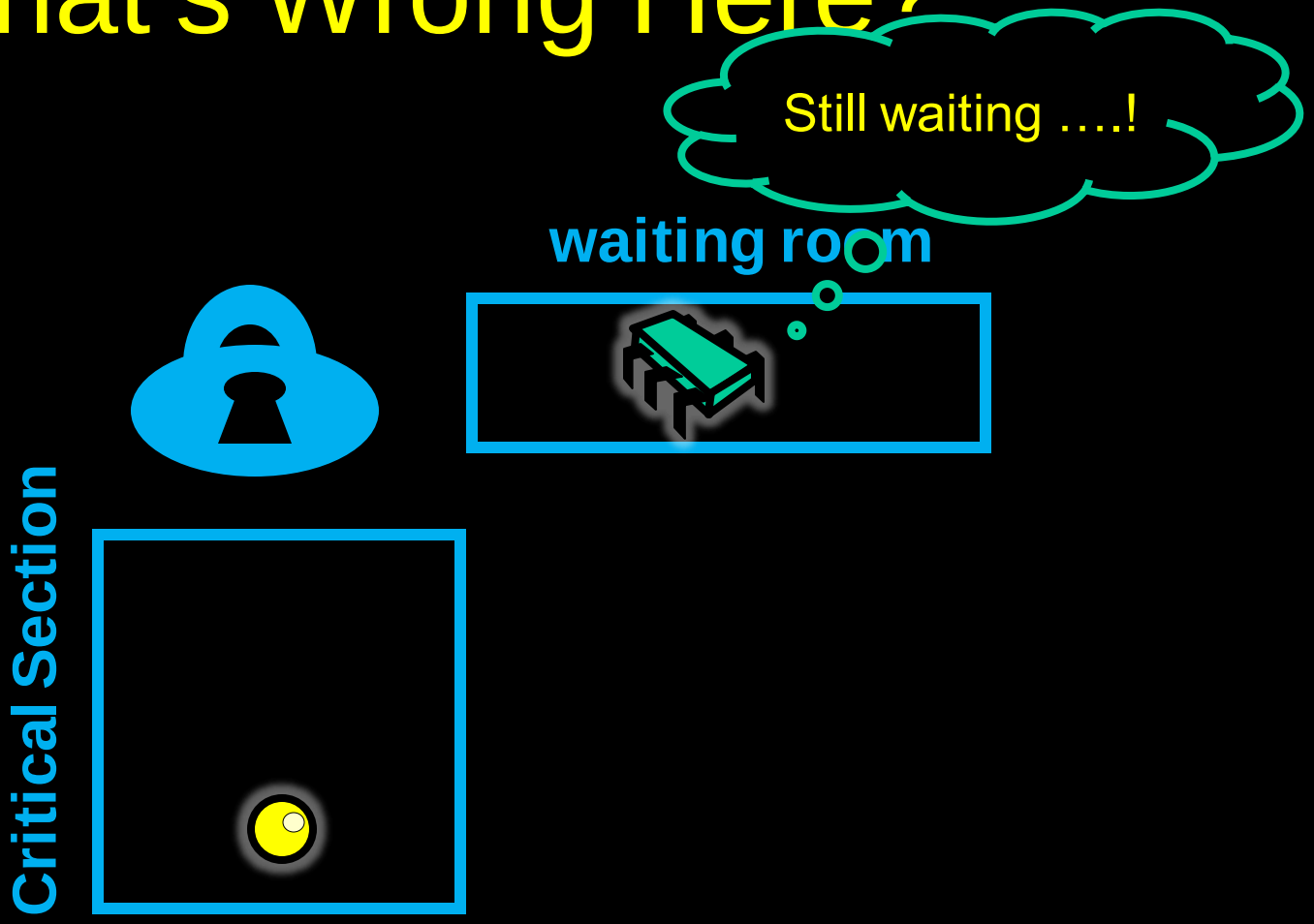
Lost Wake-Up

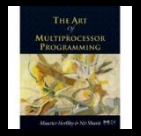


Lost Wake-Up



What's Wrong Here?





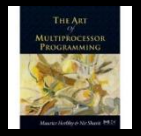
Solution to Lost Wakeup

Always use

`signalAll()` and `notifyAll()`

Instead of

`signal()` and `notify()`



WAKE-UP

Hard to detect!

No crashes, SEGFAULTs, etc, …

Just slowly drains resources

Beware!