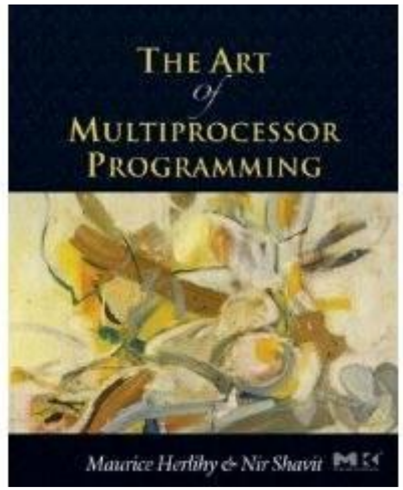
Programming Paradigms for Concurrency Lecture 5 – Monitors and Blocking Synchronization



Based on

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Thomas Wies

New York University

What Should you do if you can't get a lock?

- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

What Should you do if you can’t get a lock?

- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

today’s focus

Producer/consumer based on a FIFO Queue

```
public produce(Object x) {
 mutex.lock();
 try {
   queue.enq(x);
 } finally {
   mutex.unlock();
 }
}
```

The Need for Modular Synchronization

Suppose queue is bounded:

- enq may block until queue has room
- decision whether to block depends on internal state of the queue

Multiple producers/consumers:

- every thread needs to keep track of the lock, the queue state, etc.

The Need for Modular Synchronization

Suppose queue is bounded:

- enq may block until queue has room
- decision whether to block depends on internal state of the queue

Multiple producers/consumers:

- every thread needs to keep track of the lock, the queue state, etc.

not scalable

Modular Synchronization

Let queue handle its own synchronization

- queue has its own lock
  - acquired by each method call
  - released when the call returns
- if thread enqueues on a full queue
  - queue itself detects the problem
  - suspend the caller and resume when the queue has room

Conditions

- a condition object is associated with a lock
- condition objects allow a thread to
  - temporarily release the lock and suspend itself until awoken by another thread
  - awake other threads that are currently suspended

Monitors

The combination of

- an object and its methods
- a mutual exclusion lock
- and the lock's condition objects

is called a **monitor**

Monitors enable modular synchronization.

Java's Lock Interface

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

Java's Condition Interface

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

Java's Condition Interface

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

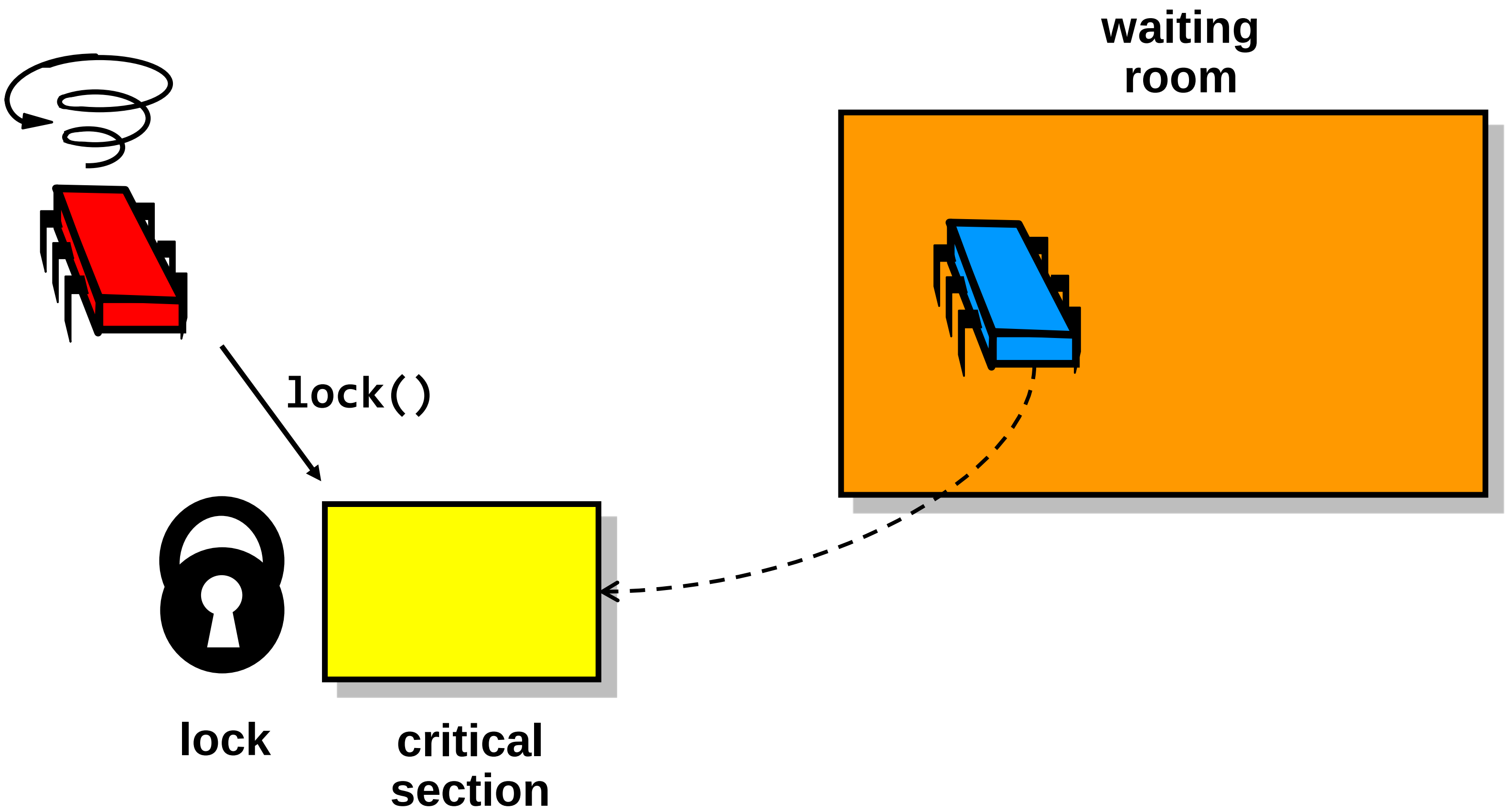
wake up **one** waiting thread

Java's `Condition` Interface

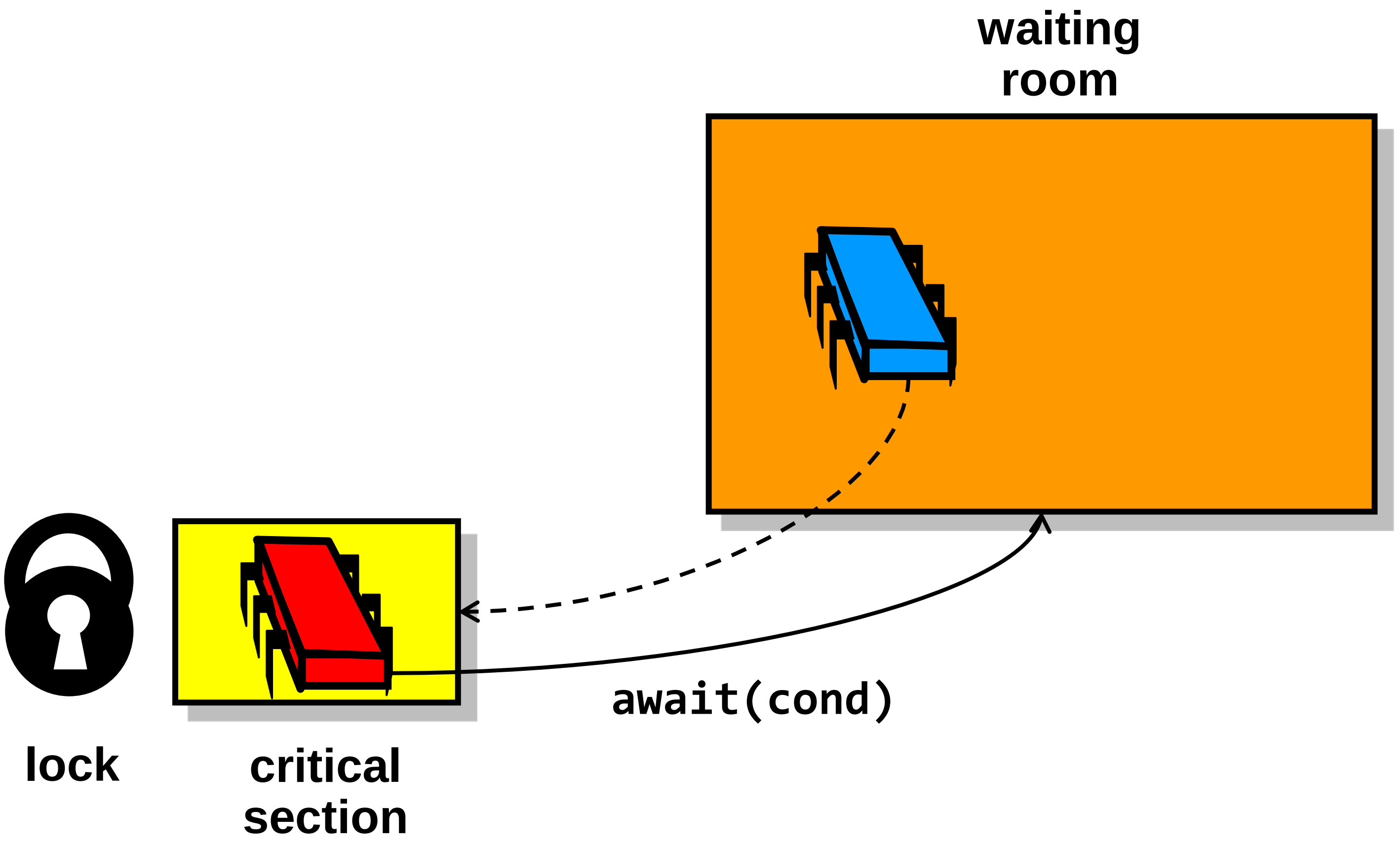
```
public interface Condition {
  void await() throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  ...
  void signal();
  void signalAll();
}
```

wake up **all** waiting threads

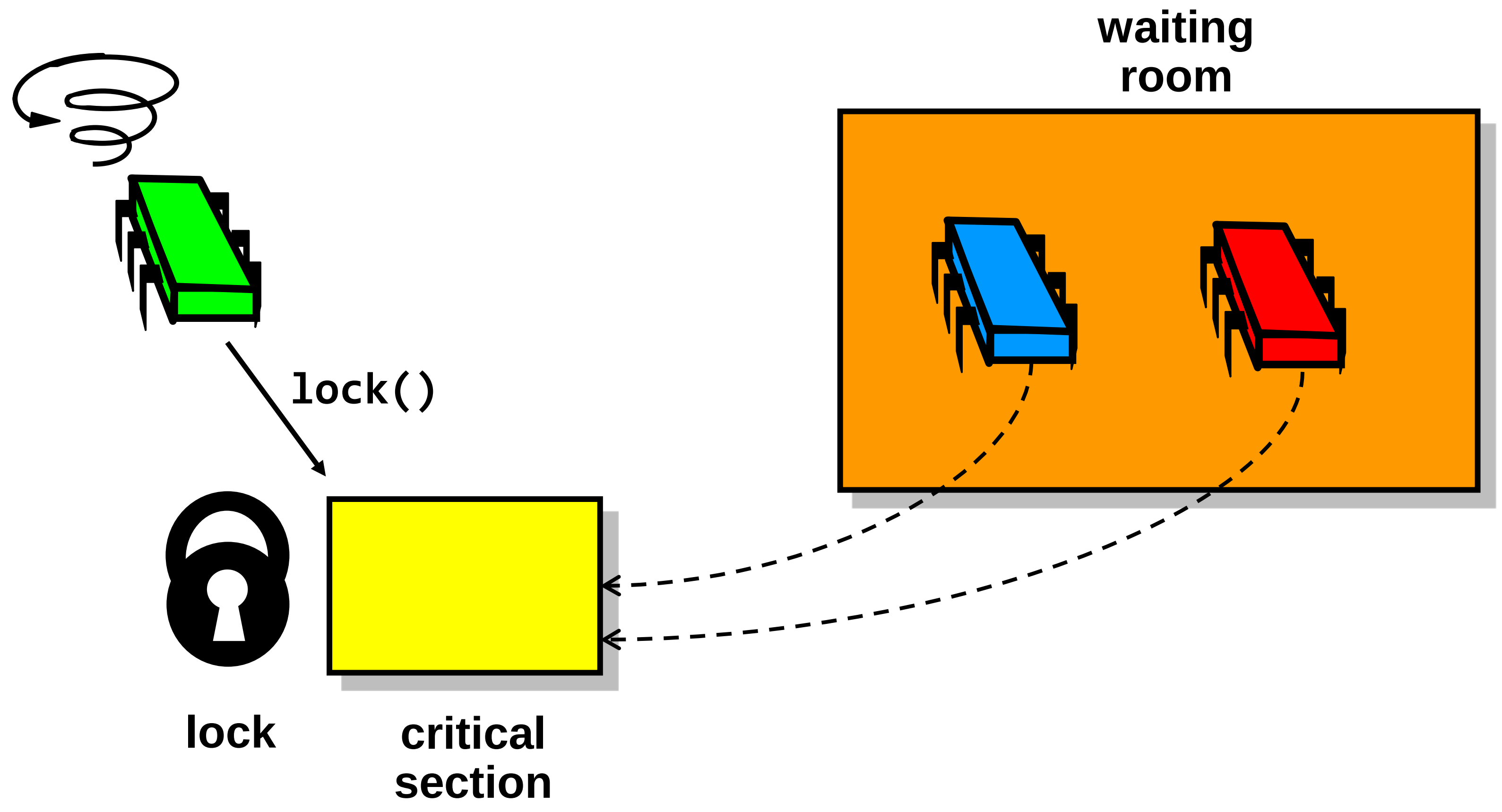
A Typical Monitor Execution



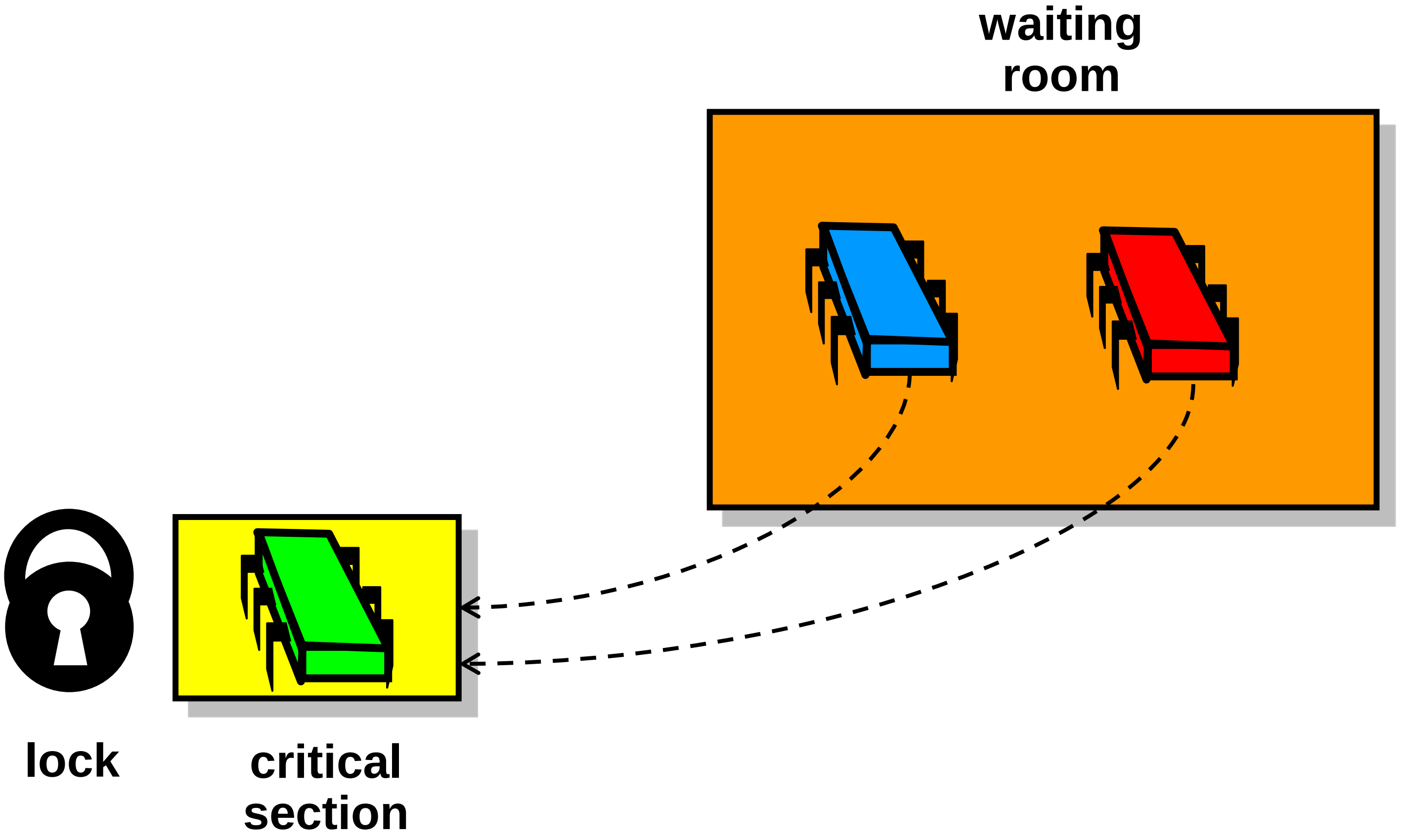
A Typical Monitor Execution



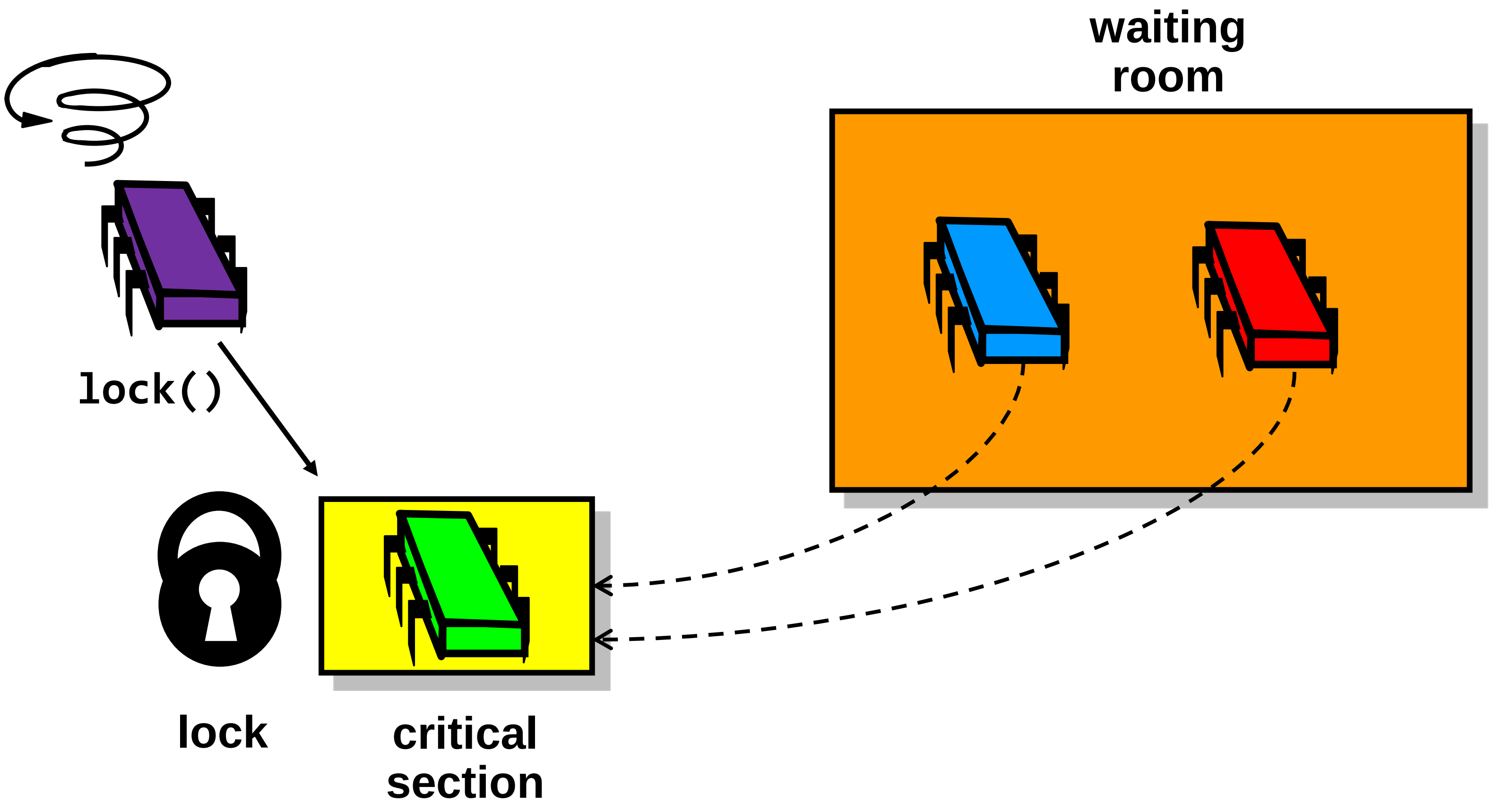
A Typical Monitor Execution



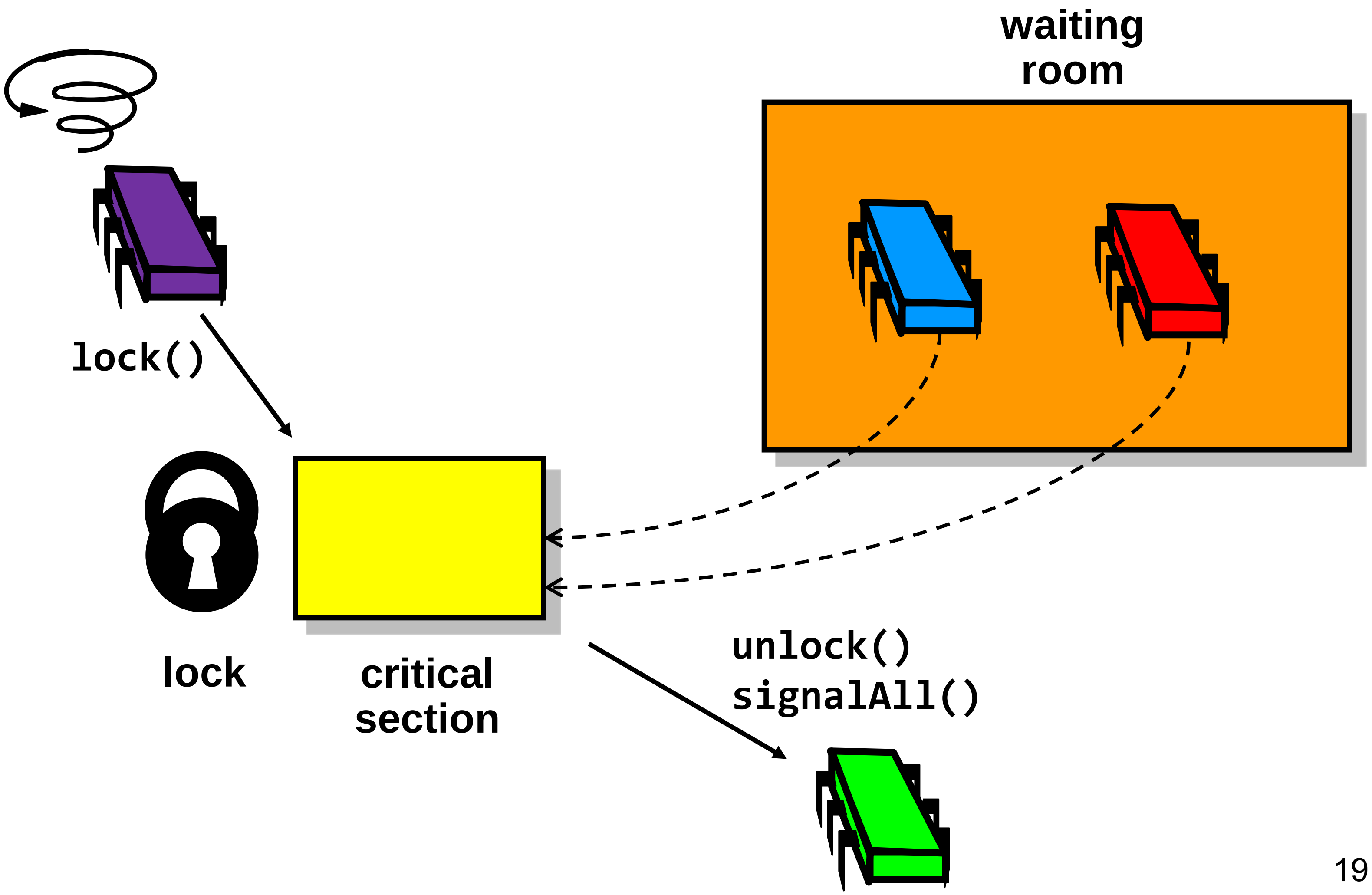
A Typical Monitor Execution



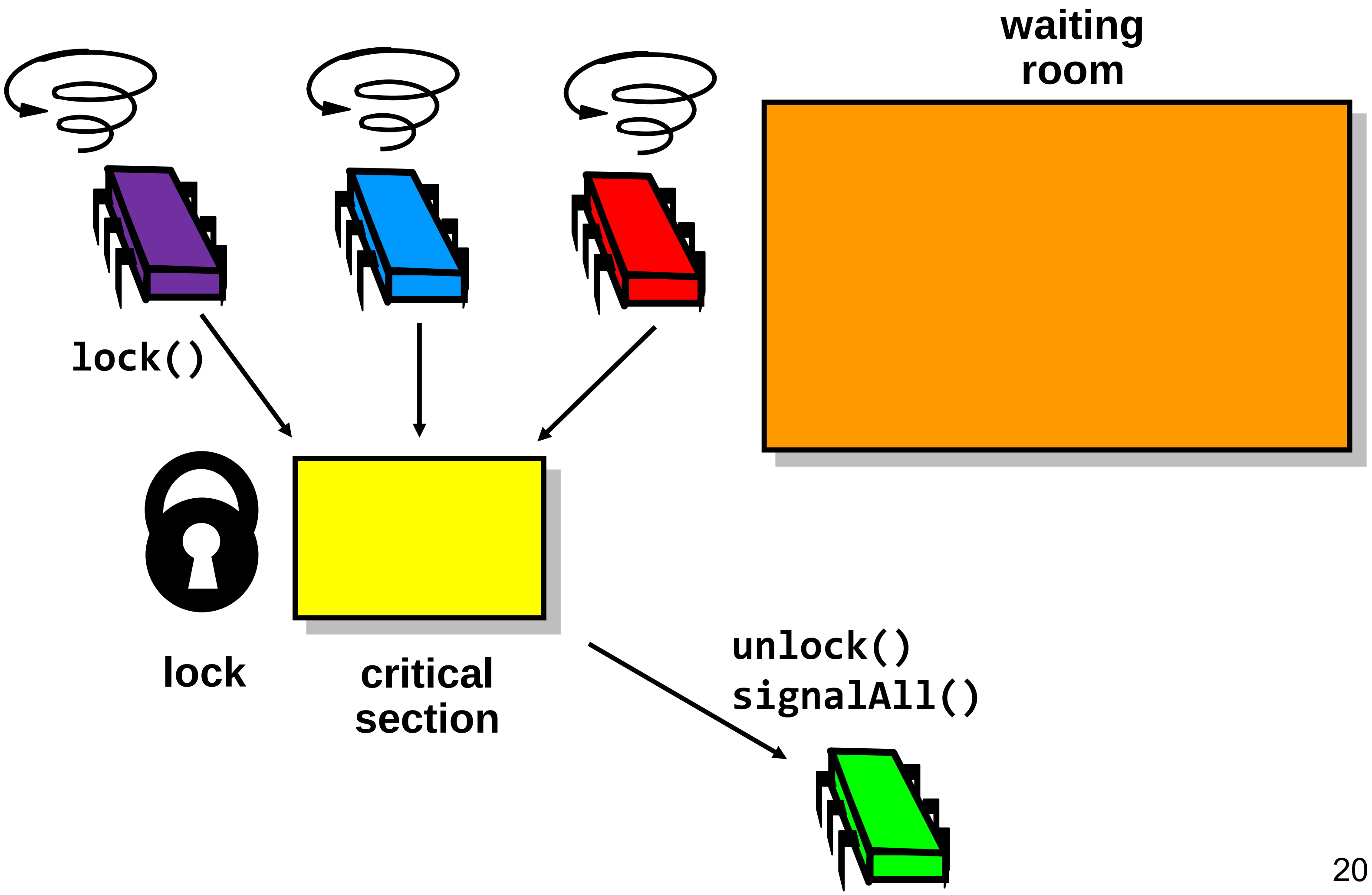
A Typical Monitor Execution



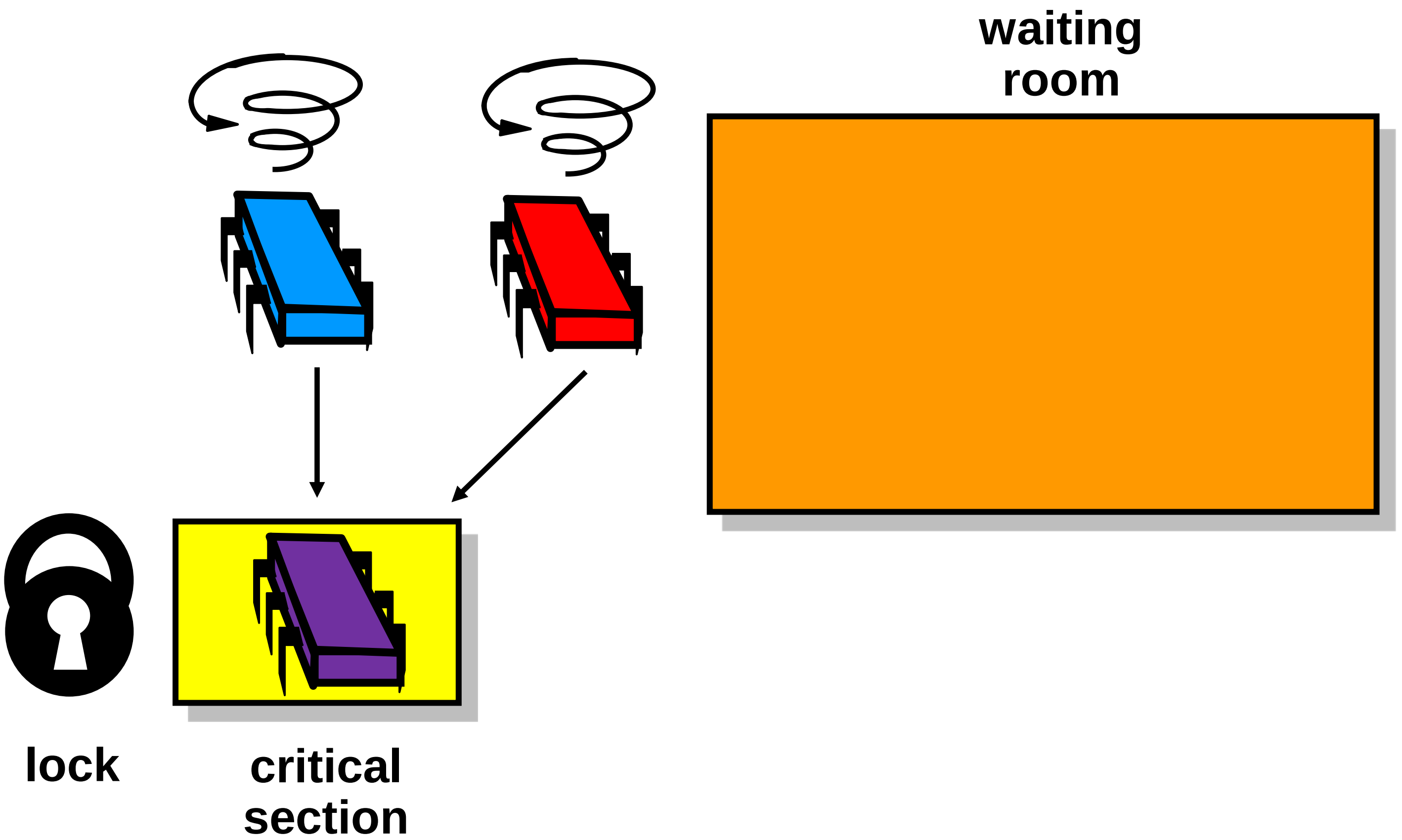
A Typical Monitor Execution



A Typical Monitor Execution



A Typical Monitor Execution



Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterrupedException e) {
  ...
}
...
```

Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterrupedException e) {
  ...
}
...
```

create new condition object

Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterrupedException e) {
  ...
}
...
```

acquire the lock

Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterrupedException e) {
  ...
}
...
```

not happy

Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterrupedException e) {
  ...
}
...
```

release the lock and suspend until notified

Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterruptedException e) {
  ...
}
...
```

application specific response

Using Condition Objects

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
  while (!property)
    condition.await();
} catch (InterrupedException e) {
  ...
}
...
```

happy: **`property`** must hold

Example: Blocking Queue

```
public class BlockingQueue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;

  public BlockingQueue(int capacity) {
    items = new T[capacity];
  }
  ...
}
```

Example: Blocking Queue

```
public class BlockingQueue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;

  public BlockingQueue(int capacity) {
    items = new T[capacity];
  }
  ...
}
```

mutual exclusion lock
for queue object

Example: Blocking Queue

```
public class BlockingQueue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;

  public BlockingQueue(int capacity) {
    items = new T[capacity];
  }
  ...
}
```

condition to wait on
if queue is full

Example: Blocking Queue

```
public class BlockingQueue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;

  public BlockingQueue(int capacity) {
    items = new T[capacity];
  }
  ...
}
```

condition to wait on
if queue is empty

Example: Blocking Queue

```
public class BlockingQueue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;

  public BlockingQueue(int capacity) {
    items = new T[capacity];
  }
  ...
}
```

internal queue state protected by lock

Blocking Queue: enqueue

```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    notEmpty.signal();
  } finally { lock.unlock(); }
}
```

Blocking Queue: enqueue

```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    notEmpty.signal();
  } finally { lock.unlock(); }
}
```

wait until queue has space

Blocking Queue: enqueue

```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    notEmpty.signal();
  } finally { lock.unlock(); }
}
```

queue has space!
insert element

Blocking Queue: enqueue

```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    notEmpty.signal();
  } finally { lock.unlock(); }
}
```

wake up one waiting consumer

Blocking Queue: dequeue

```
public T deq() {
  lock.lock();
  try {
    while (count == 0)
      notEmpty.await();
    T x = items[head];
    if (++head == items.length) head = 0;
    --count;
    notFull.signal();
    return x;
  } finally { lock.unlock(); }
}
```

Blocking Queue: dequeue

```
public T deq() {
  lock.lock();
  try {
    while (count == 0)
      notEmpty.await();
    T x = items[head];
    if (++head == items.length) head = 0;
    --count;
    notFull.signal();
    return x;
  } finally { lock.unlock(); }
}
```

wait until queue is nonempty

Blocking Queue: dequeue

```
public T deq() {
  lock.lock();
  try {
    while (count == 0)
      notEmpty.await();
    T x = items[head];
    if (++head == items.length) head = 0;
    --count;
    notFull.signal();
    return x;
  } finally { lock.unlock(); }
}
```

queue nonempty! retrieve next element

Blocking Queue: dequeue

```
public T deq() {
  lock.lock();
  try {
    while (count == 0)
      notEmpty.await();
    T x = items[head];
    if (++head == items.length) head = 0;
    --count;
    notFull.signal();
    return x;
  } finally { lock.unlock(); }
}
```

wake up one waiting producer

Improved enqueue?

```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    if (count == 1) notEmpty.signal();
  } finally { lock.unlock(); }
}
```

lost wakeups

The Lost-Wakeup Problem

- Condition variables are inherently vulnerable to lost wakeups
  - one thread waits forever without realizing that its waiting condition has become true
- Programming practices
  - if in doubt, signal **all** waiting processes
  - specify a timeout when waiting

Reentrant Locks

- same thread can acquire the lock multiple times without blocking
- commonly used in OOP to handle reentrant calls to locked objects

Using Reentrant Locks

```
public class AtomicArray<T> {
  final Lock lock = new ReentrantLock();
  ...
  public T getAndSet(int i, T v) {
    try { lock.lock();
      T old = get(i);
      set(i, v);
      return old;
    } finally { lock.unlock(); } }
  public T get() {
    try {lock.lock(); return item[i]; }
    finally { lock.unlock(); }
  public void set(int i, T v) { ... } }
```

Using Reentrant Locks

```
public class AtomicArray<T> {
  final Lock lock = new ReentrantLock();
  ...
  public T getAndSet(int i, T v) {
    try { lock.lock();
      T old = get(i);
      set(i, v);
      return old;
    } finally { lock.unlock(); } }
  public T get() {
    try {lock.lock(); return item[i]; }
    finally { lock.unlock(); }
  public void set(int i, T v) { ... } }
```

reacquire **`lock`**

Our Own Reentrant Lock

```
public class SimpleReentrantLock implements Lock{
  final Lock lock = new SimpleLock();
  final Condition cond = lock.newCondition();
  int owner, holdCount;

  public SimpleReentrantLock() {
    owner = holdCount = 0;
  }
  ...
}
```

Our Own Reentrant Lock

```
public class SimpleReentrantLock implements Lock{
  final Lock lock = new SimpleLock();
  final Condition cond = lock.newCondition();
  int owner, holdCount;

  public SimpleReentrantLock() {
    owner = holdCount = 0;
  }
  ...
}
```

nonreentrant lock

Our Own Reentrant Lock

```
public class SimpleReentrantLock implements Lock{
  final Lock lock = new SimpleLock();
  final Condition cond = lock.newCondition();
  int owner, holdCount;

  public SimpleReentrantLock() {
    owner = holdCount = 0;
  }
  ...
}
```

condition to wait on if lock is held by other thread

Our Own Reentrant Lock

```
public class SimpleReentrantLock implements Lock{
  final Lock lock = new SimpleLock();
  final Condition cond = lock.newCondition();
  int owner, holdCount;

  public SimpleReentrantLock() {
    owner = holdCount = 0;
  }
  ...
}
```

thread ID of lock holder

Our Own Reentrant Lock

```
public class SimpleReentrantLock implements Lock{
  final Lock lock = new SimpleLock();
  final Condition cond = lock.newCondition();
  int owner, holdCount;

  public SimpleReentrantLock() {
    owner = holdCount = 0;
  }
  ...
}
```

counts how often lock has been acquired by current owner

Our Own Reentrant Lock

```
public void lock() {
  int me = ThreadID.get();
  lock.lock();
  try {
    if (owner == me) {
      ++holdCount;
      return;
    }
    while (holdCount != 0) condition.await();
    owner = me;
    holdCount = 1;
  } finally { lock.unlock() } }
```

Our Own Reentrant Lock

```
public void lock() {
  int me = ThreadID.get();
  lock.lock();
  try {
    if (owner == me) {
      ++holdCount;
      return;
    }
    while (holdCount != 0) condition.await();
    owner = me;
    holdCount = 1;
  } finally { lock.unlock() } }
```

already holding the lock?
then just increase counter

Our Own Reentrant Lock

```
public void lock() {
  int me = ThreadID.get();
  lock.lock();
  try {
    if (owner == me) {
      ++holdCount;
      return;
    }
    while (holdCount != 0) condition.await();
    owner = me;
    holdCount = 1;
  } finally { lock.unlock() } }
```

otherwise, wait until lock is free and then take ownership

Our Own Reentrant Lock

```
public void unlock() {
  lock.lock();
  try {
    if (holdCount == 0 ||
          owner != ThreadID.get()) {
      throw new IllegalMonitorStateException();
    }
    if (--holdCount == 0) cond.signal();
  } finally { lock.unlock() }
}
```

Our Own Reentrant Lock

```
public void unlock() {
  lock.lock();
  try {
    if (holdCount == 0 ||
        owner != ThreadID.get()) {
      throw new IllegalMonitorStateException();
    }
    if (--holdCount == 0) cond.signal();
  } finally { lock.unlock() }
}
```

fail, if lock is released too often

Our Own Reentrant Lock

```
public void unlock() {
  lock.lock();
  try {
    if (holdCount == 0 ||
        owner != ThreadID.get()) {
      throw new IllegalMonitorStateException();
    }
    if (--holdCount == 0) cond.signal();
  } finally { lock.unlock() }
}
```

otherwise, decrement counter
and wake up one blocked thread
if lock is released

Java's built-in Monitors

- `synchronized` blocks and methods acquire and release an implicit reentrant lock
- access to an implicit condition object is provided via special methods
  - `wait()`
  - `notify()`
  - `notifyAll()`

Simplified Blocking Queue: enqueue

```
public synchronized void enq(T x) {
  while (count == items.length())
    wait();
  items[tail] = x;
  if (++tail == items.length) tail = 0;
  ++count;
  notifyAll();
}
```

Simplified Blocking Queue: dequeue

```
public synchronized T deq() {
  while (count == 0)
    wait();
  T x = items[head];
  if (++head == items.length) head = 0;
  --count;
  notifyAll();
  return x;
}
```

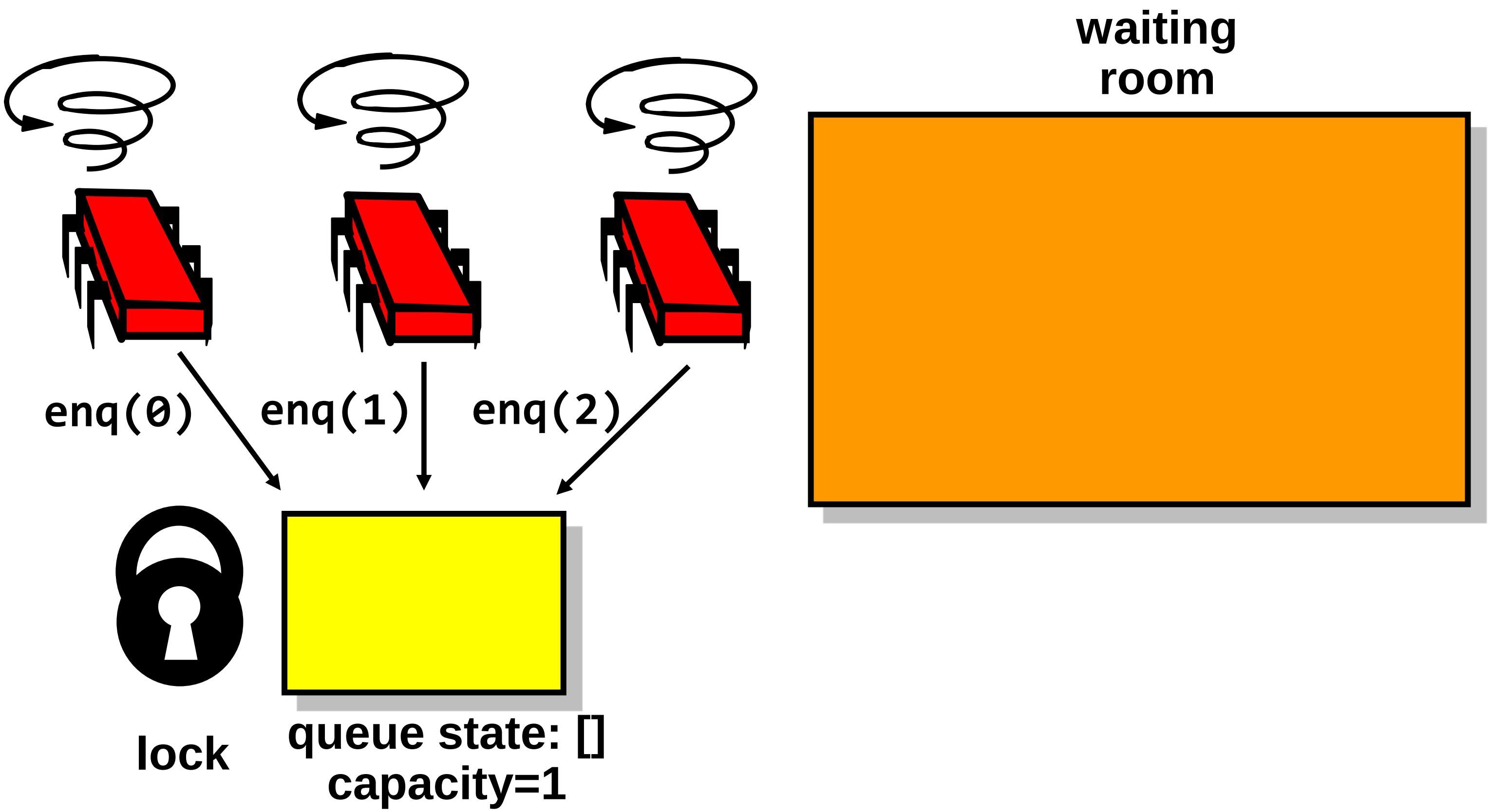
Simplified Blocking Queue: dequeue

```
public synchronized T deq() {
  while (count == 0)
    wait();
  T x = items[head];
  if (++head == items.length) head = 0;
  --count;
  notify();
  return x;
}
```

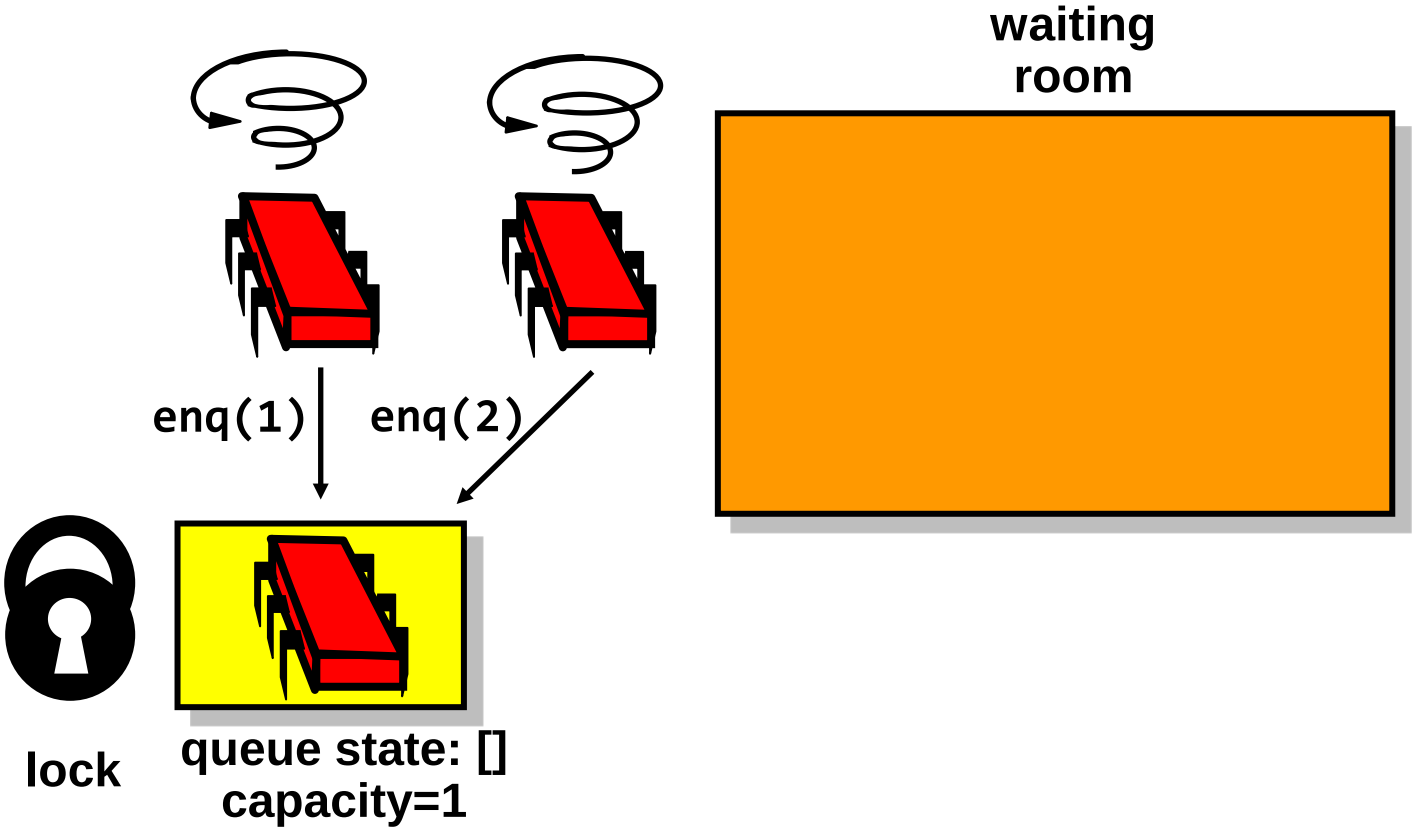
is `notify` enough?

lost wakeups

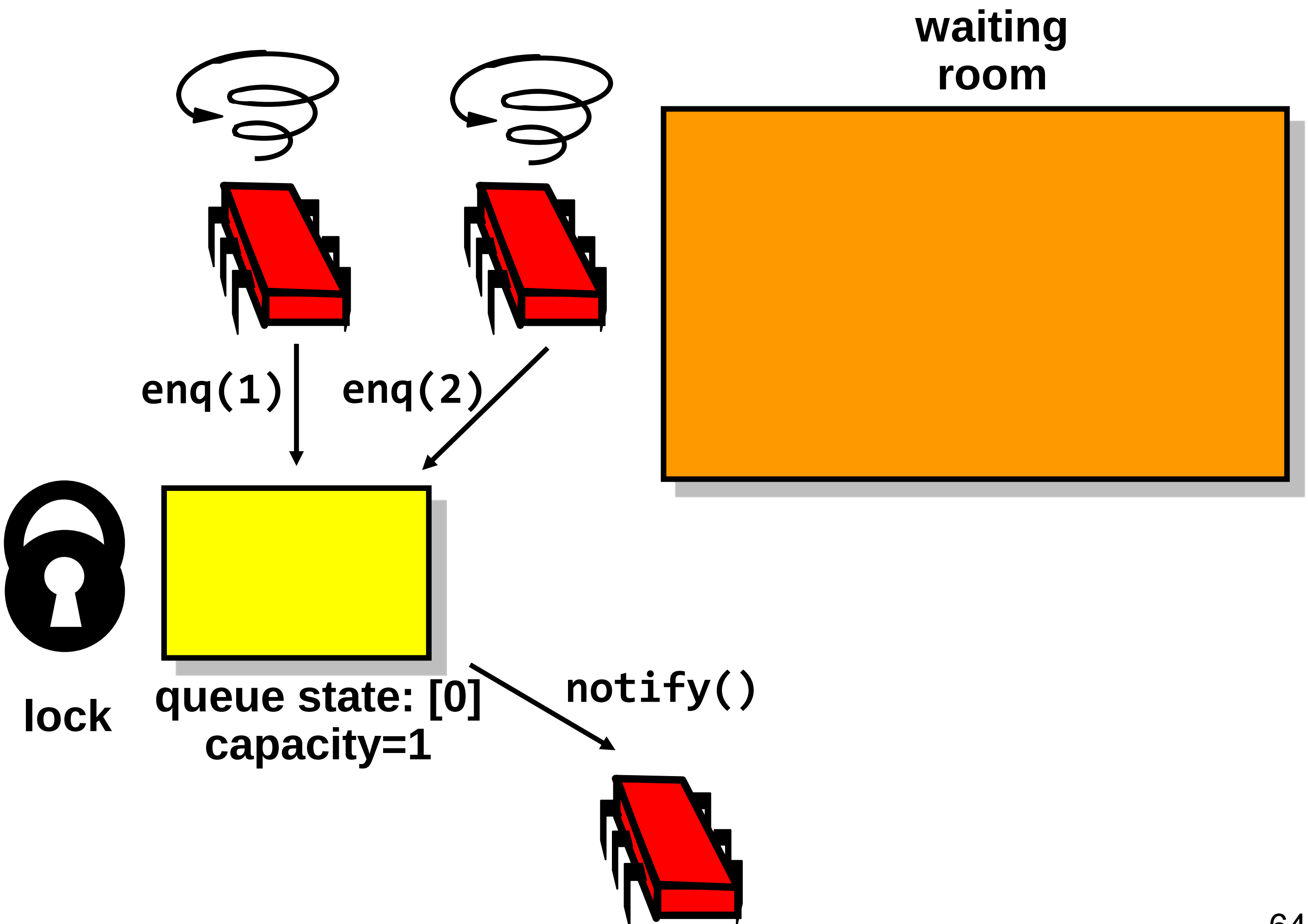
Lost Wakeup in Simplified Queue with `notify()`



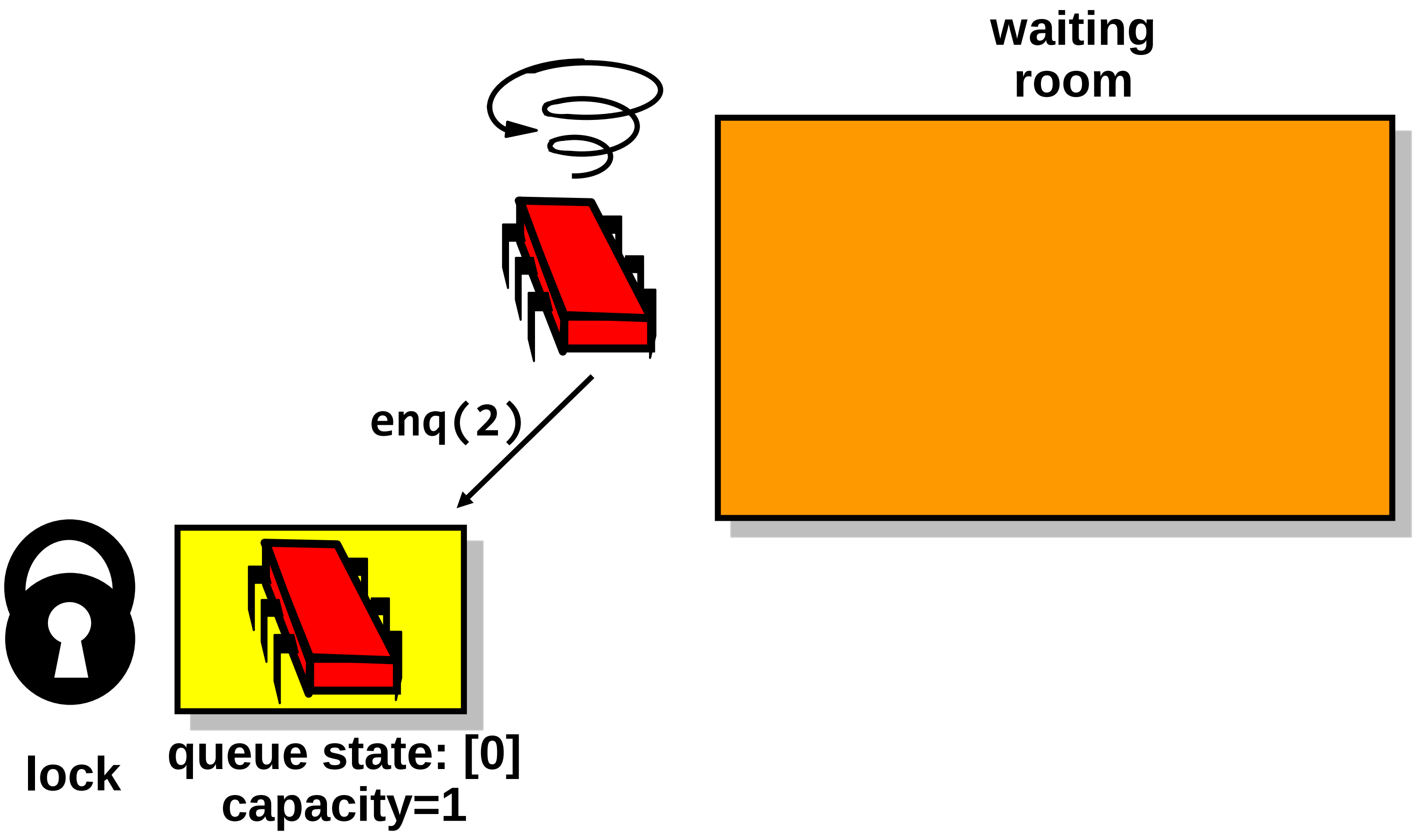
Lost Wakeup in Simplified Queue with `notify()`



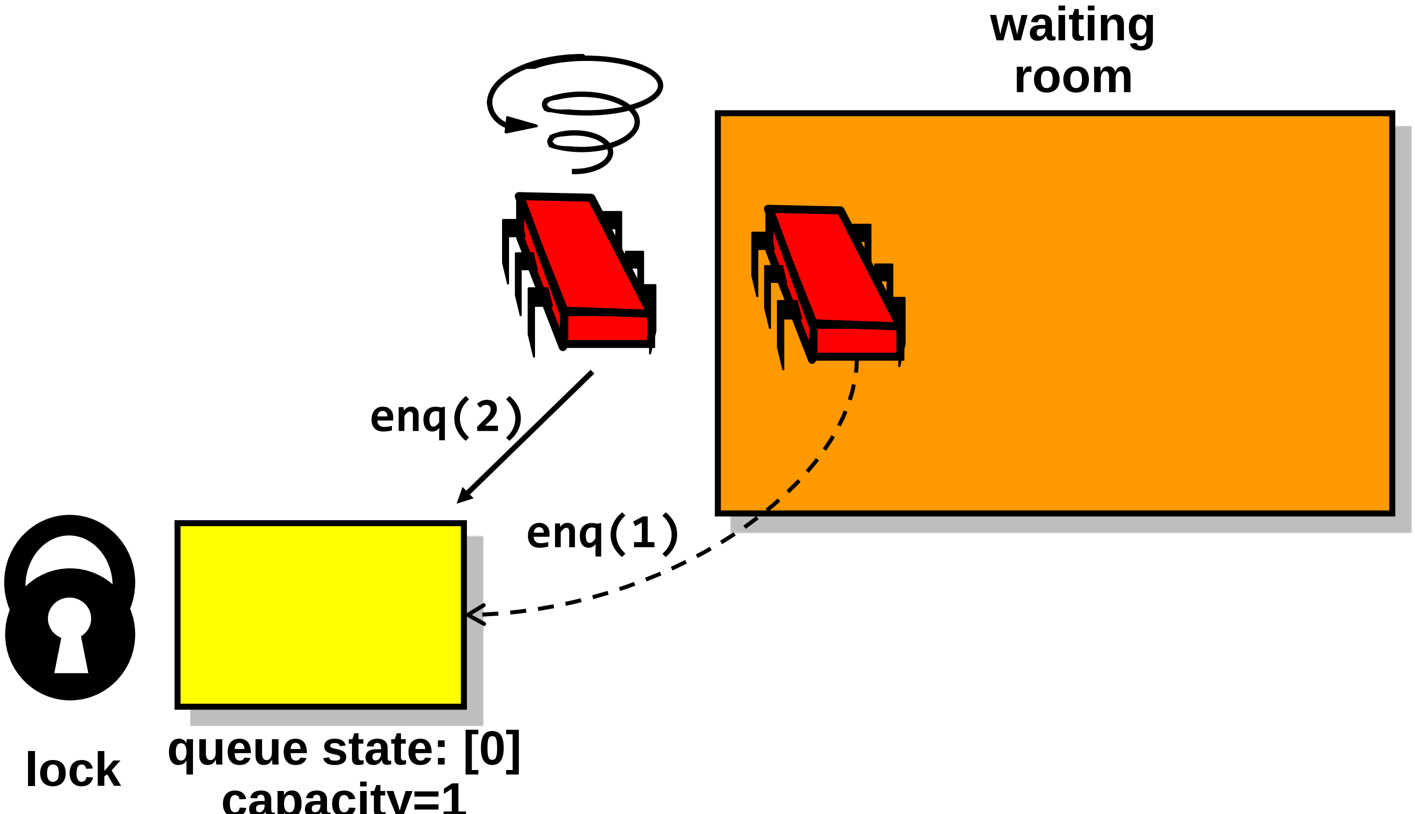
Lost Wakeup in Simplified Queue with `notify()`



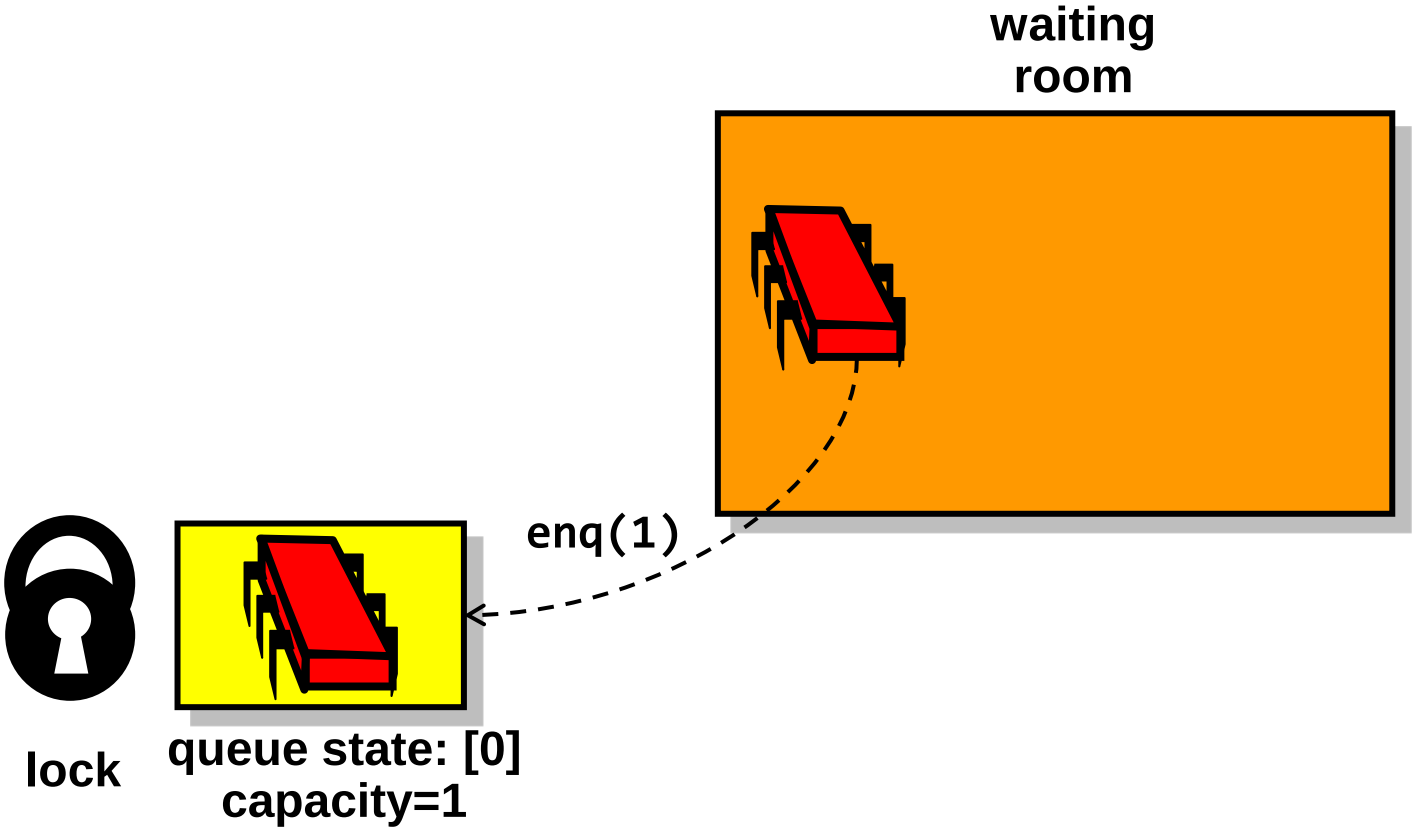
Lost Wakeup in Simplified Queue with `notify()`



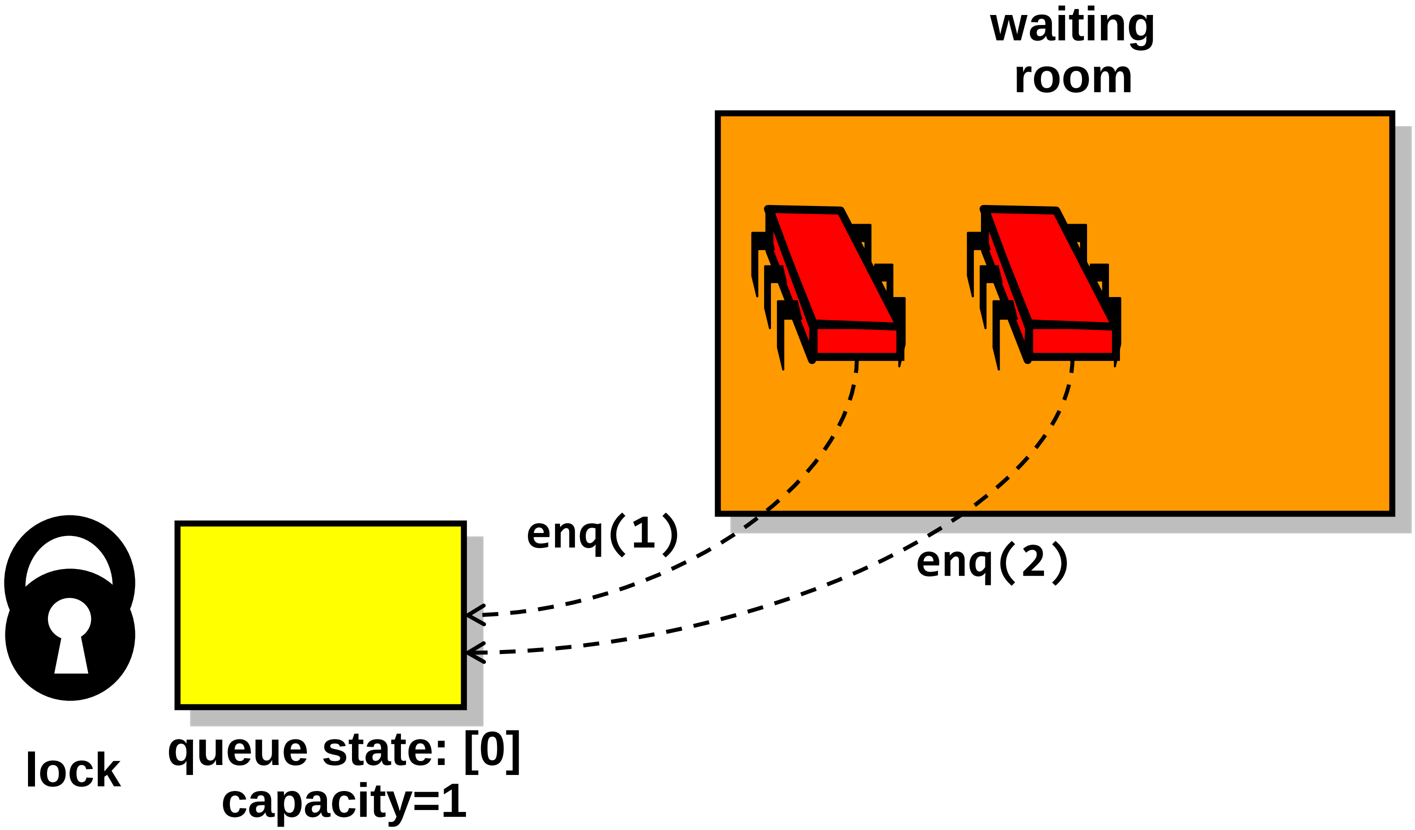
Lost Wakeup in Simplified Queue with `notify()`



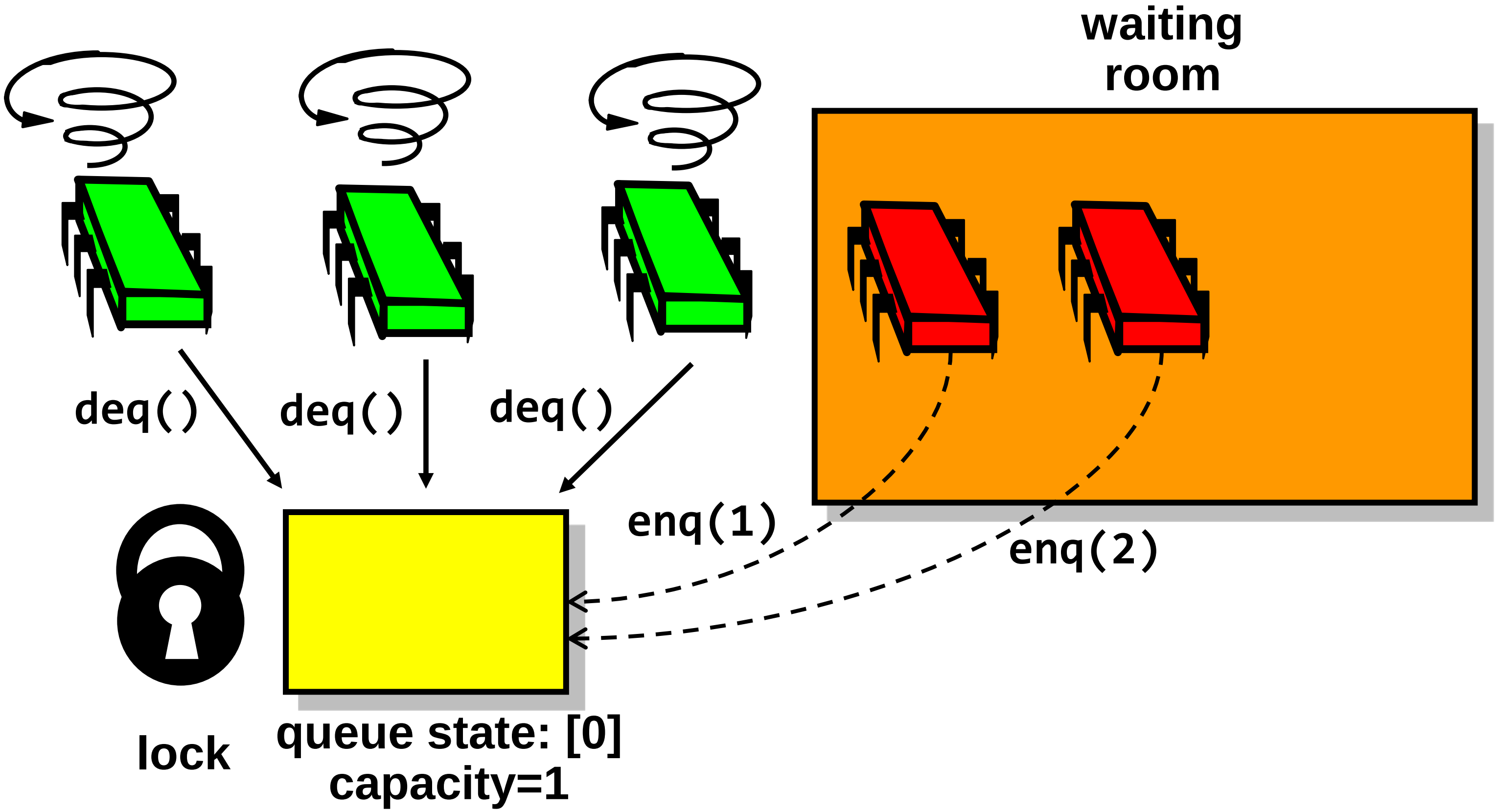
Lost Wakeup in Simplified Queue with `notify()`



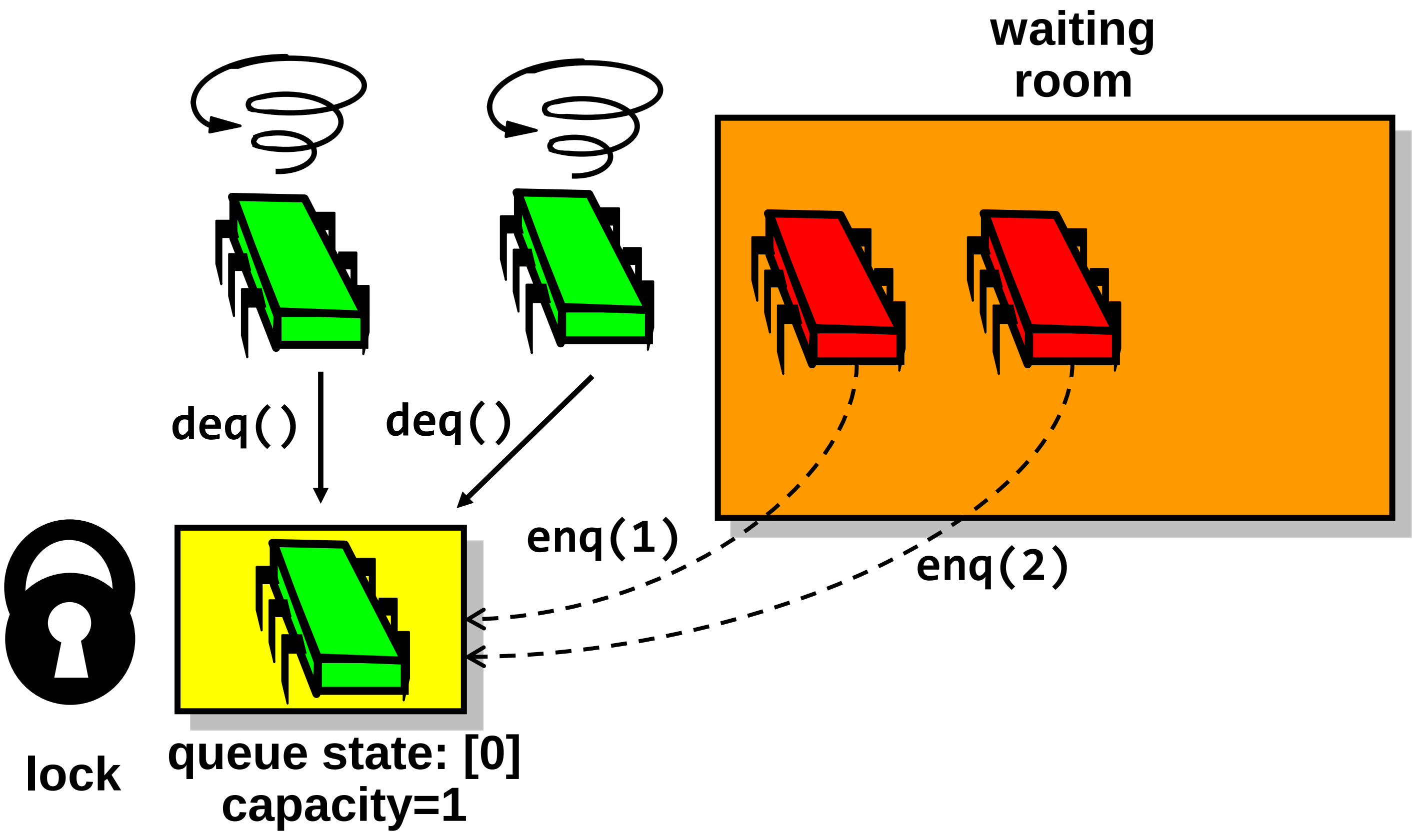
Lost Wakeup in Simplified Queue with `notify()`



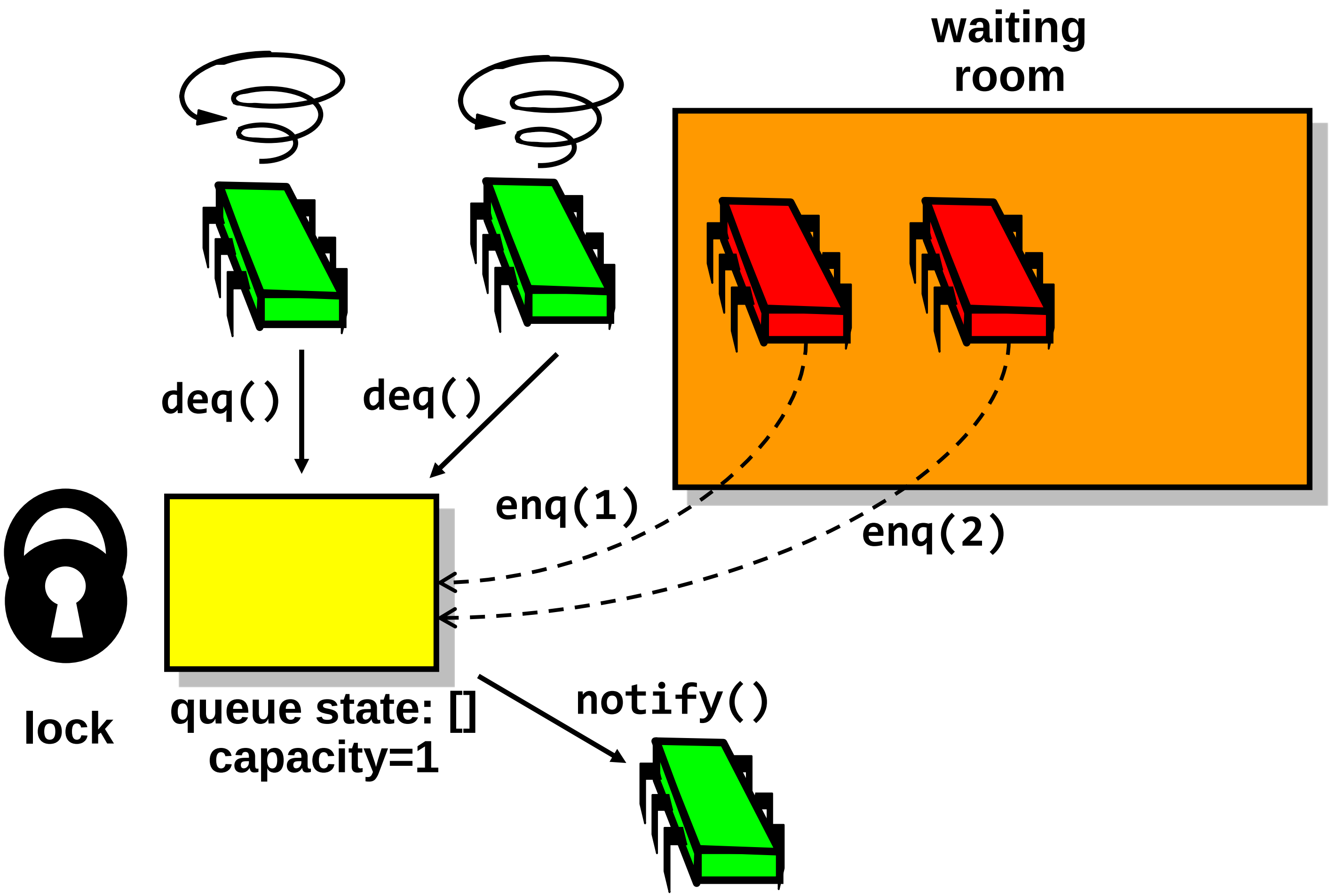
Lost Wakeup in Simplified Queue with `notify()`



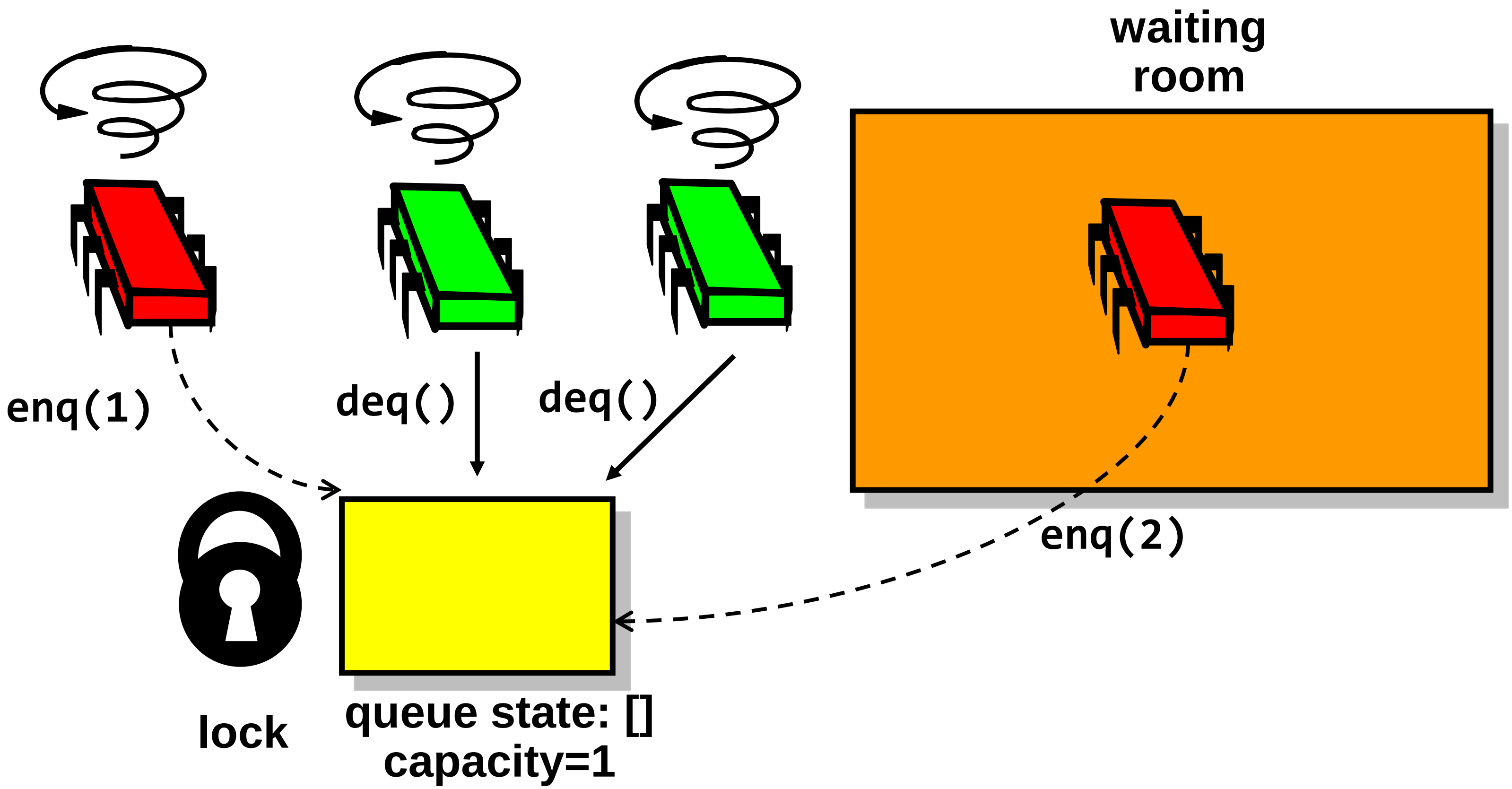
Lost Wakeup in Simplified Queue with `notify()`



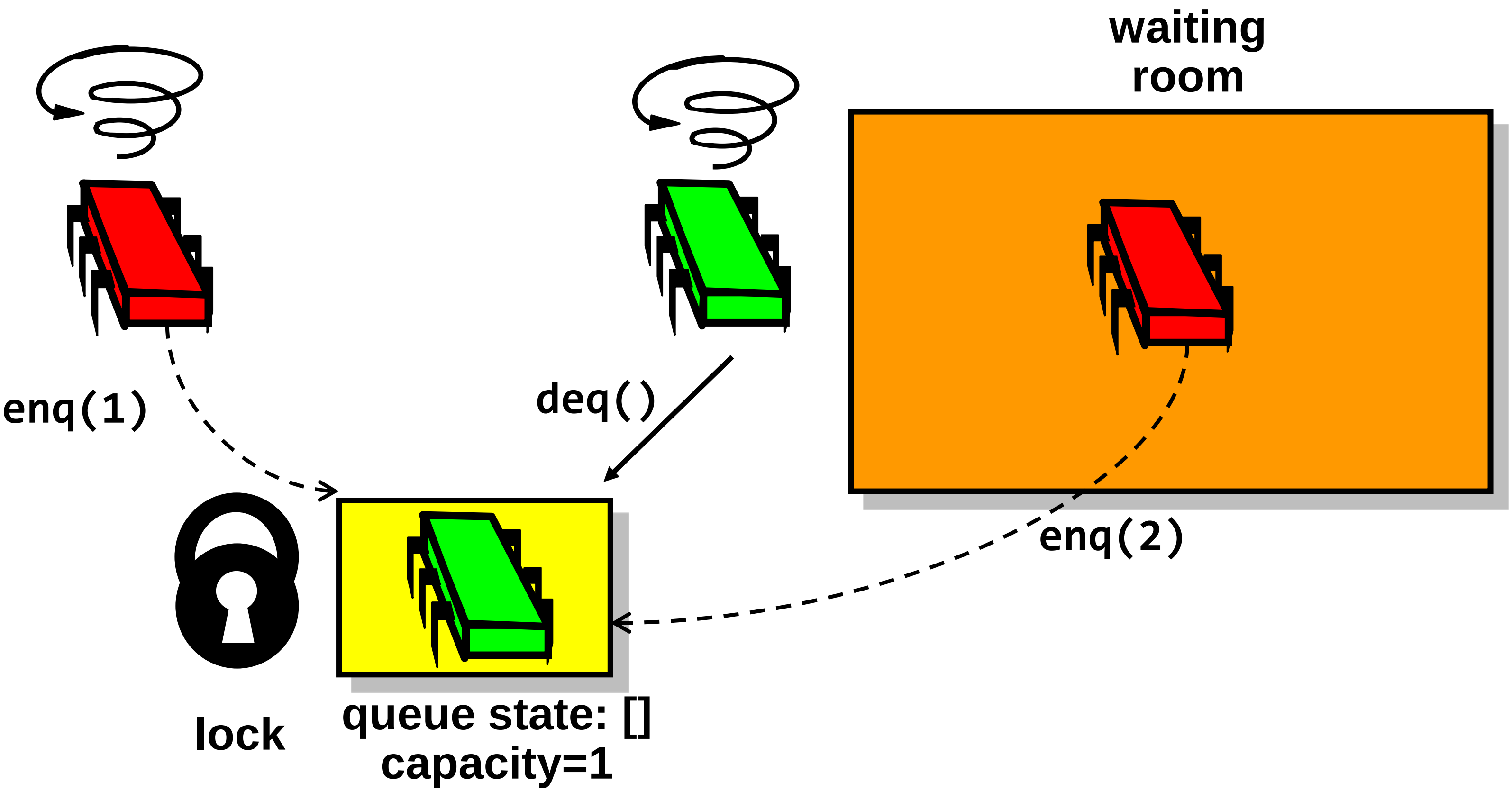
Lost Wakeup in Simplified Queue with `notify()`



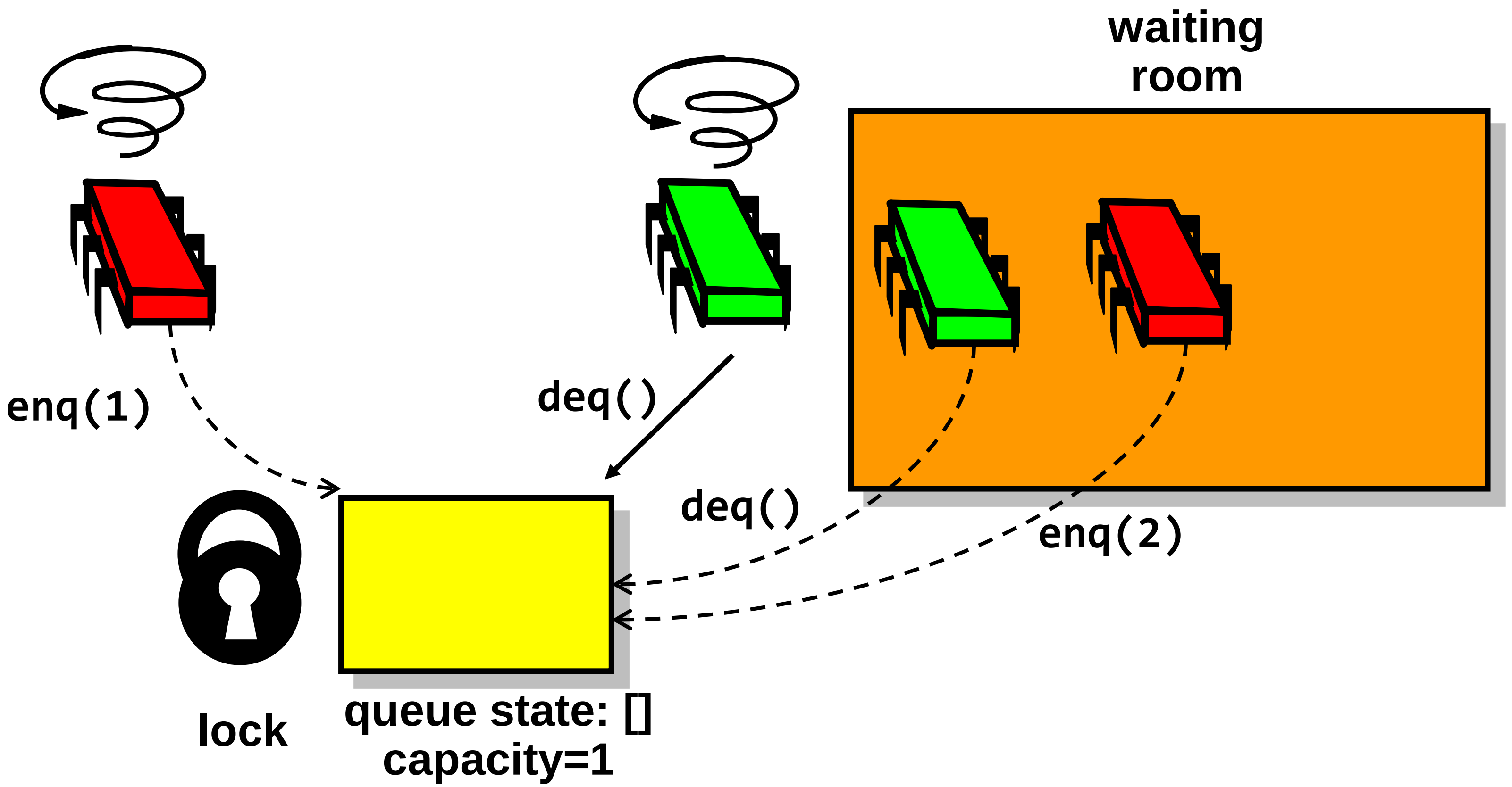
Lost Wakeup in Simplified Queue with `notify()`



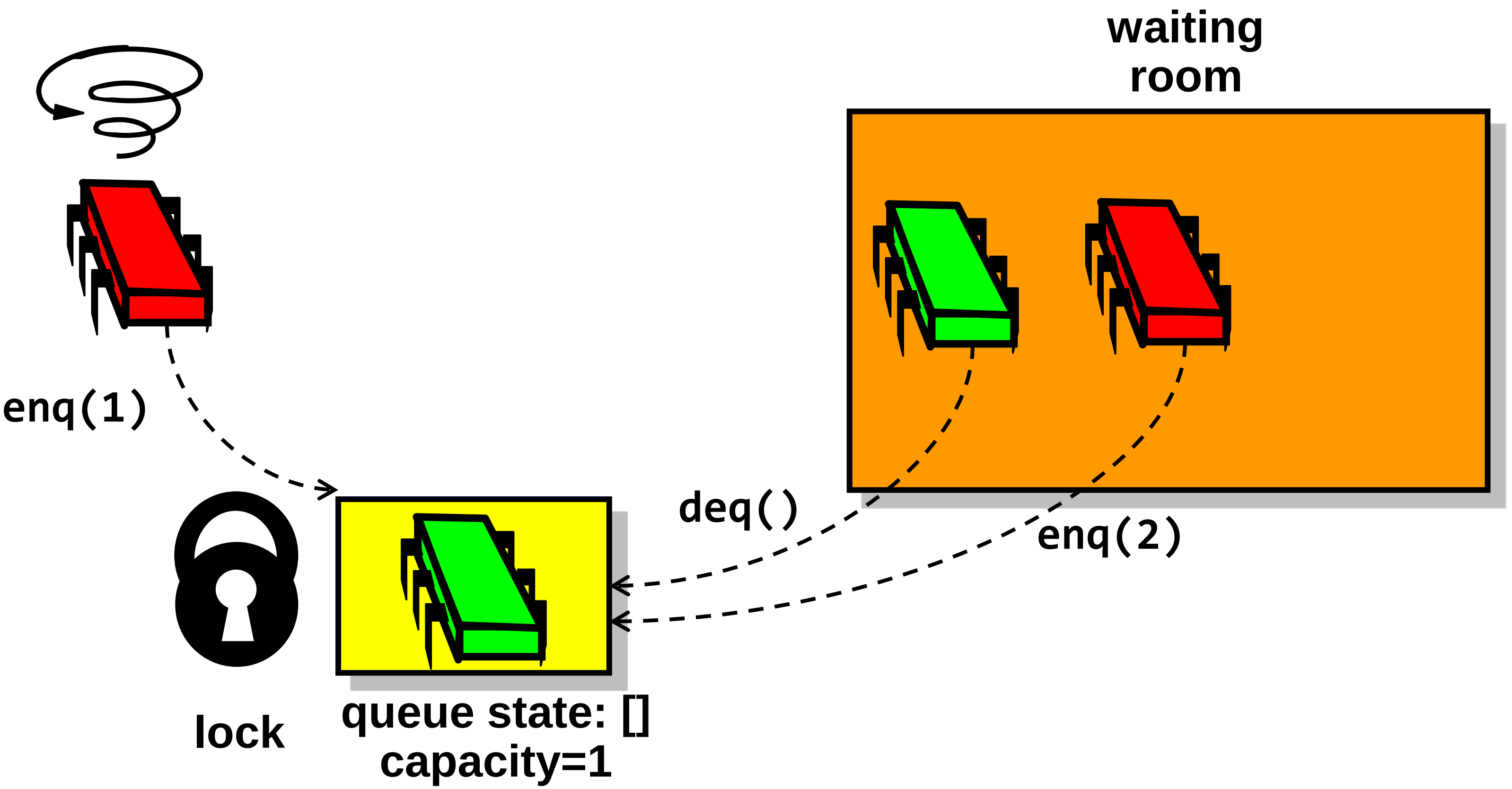
Lost Wakeup in Simplified Queue with `notify()`



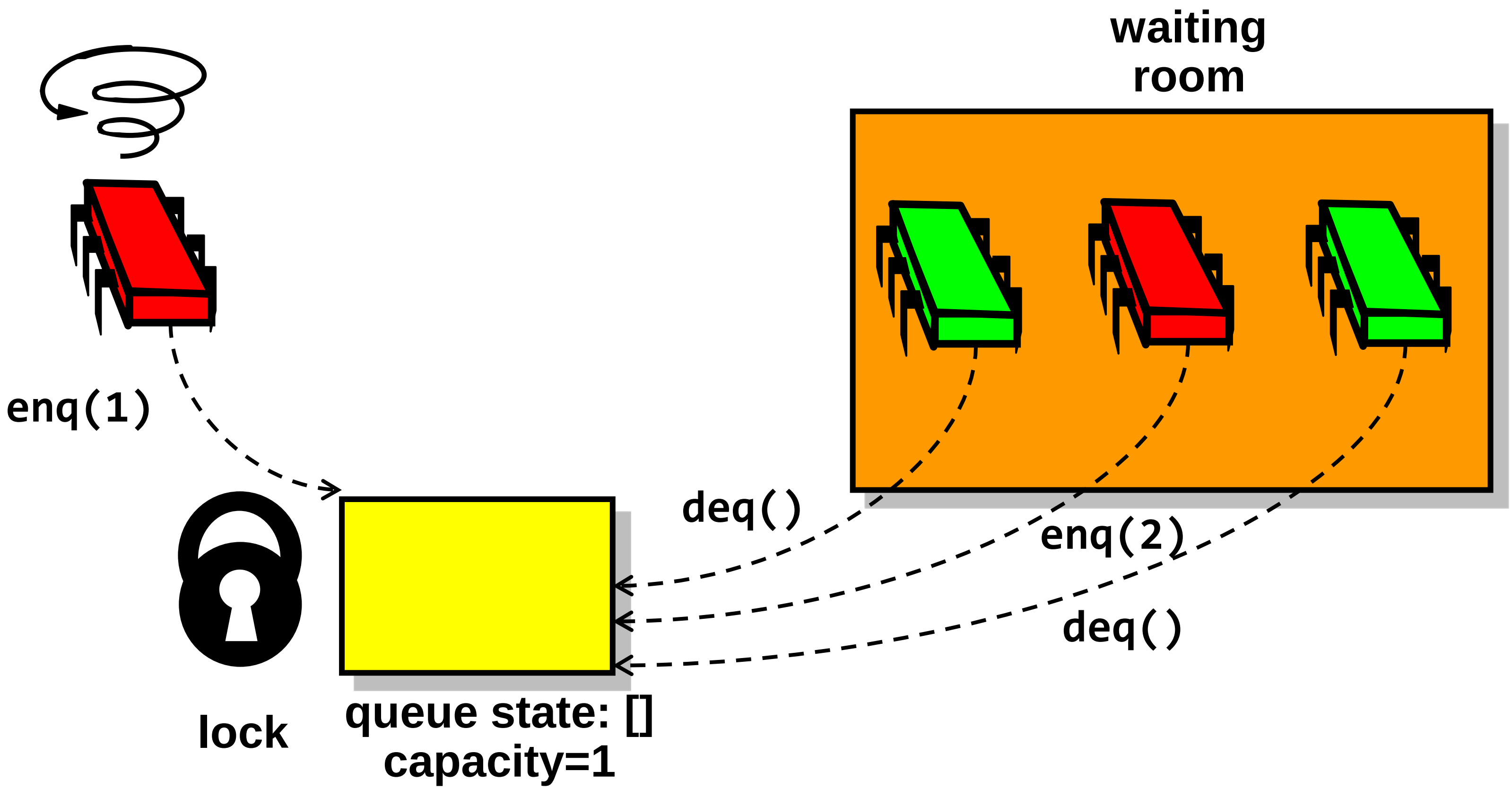
Lost Wakeup in Simplified Queue with `notify()`



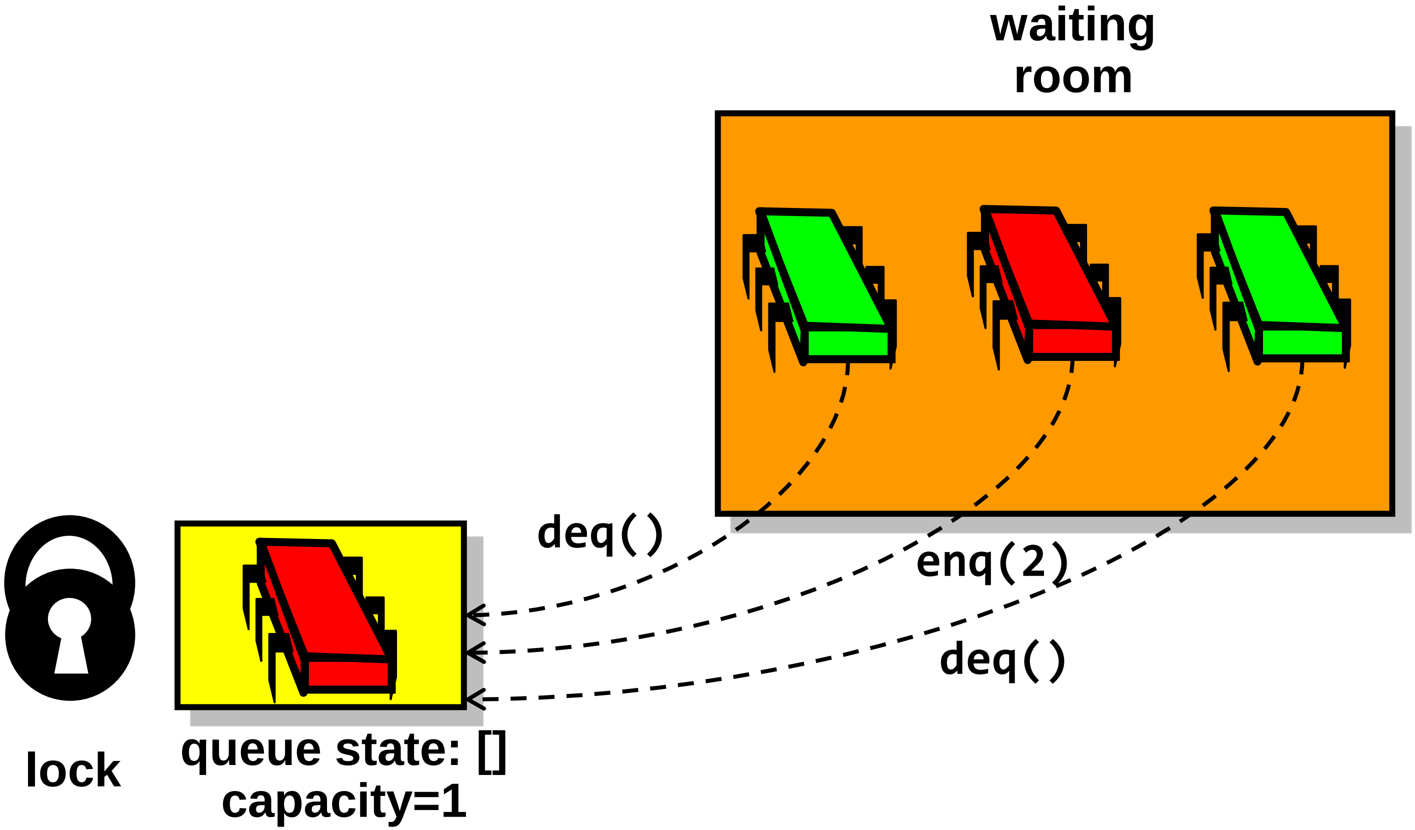
Lost Wakeup in Simplified Queue with `notify()`



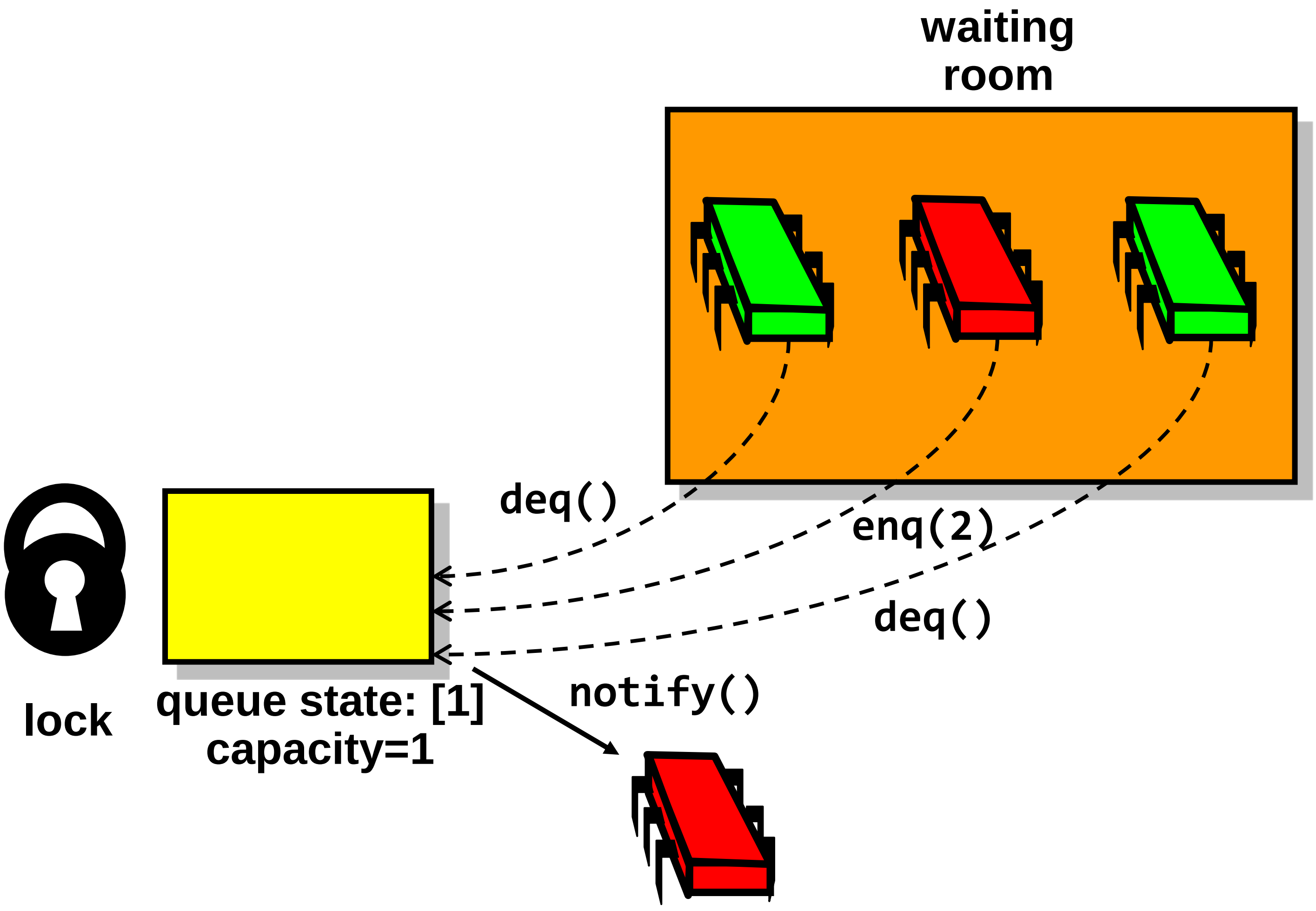
Lost Wakeup in Simplified Queue with `notify()`



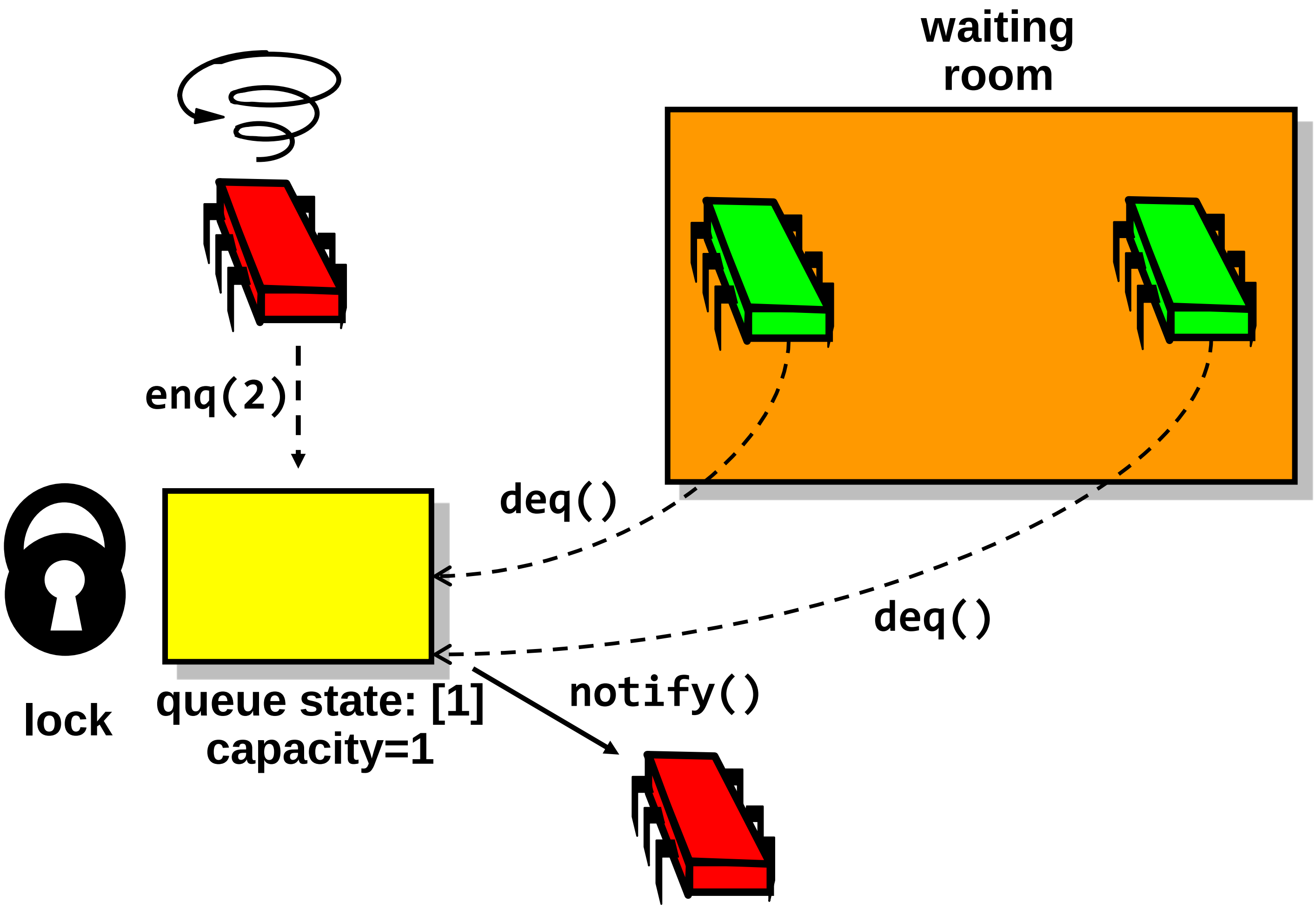
Lost Wakeup in Simplified Queue with `notify()`



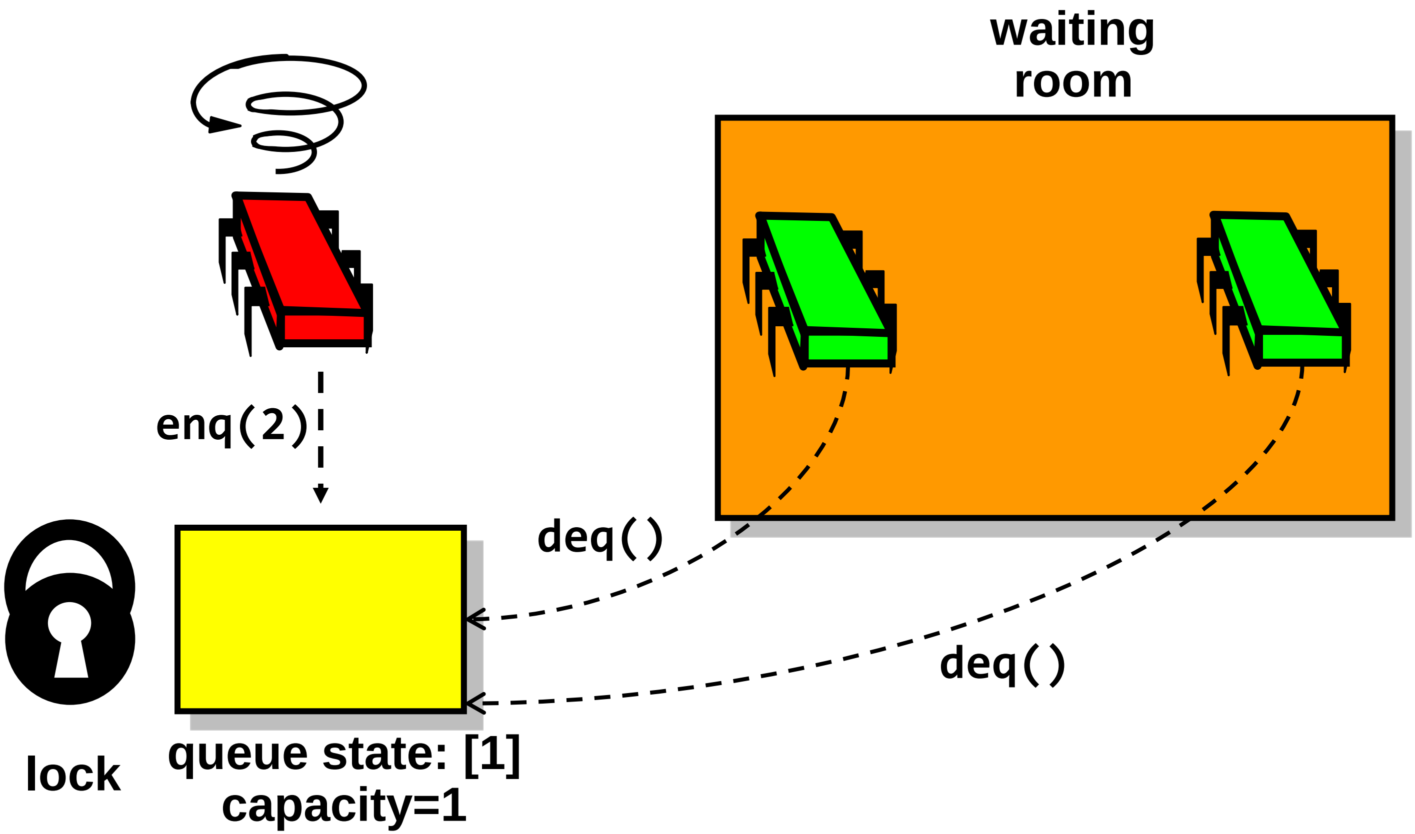
Lost Wakeup in Simplified Queue with `notify()`



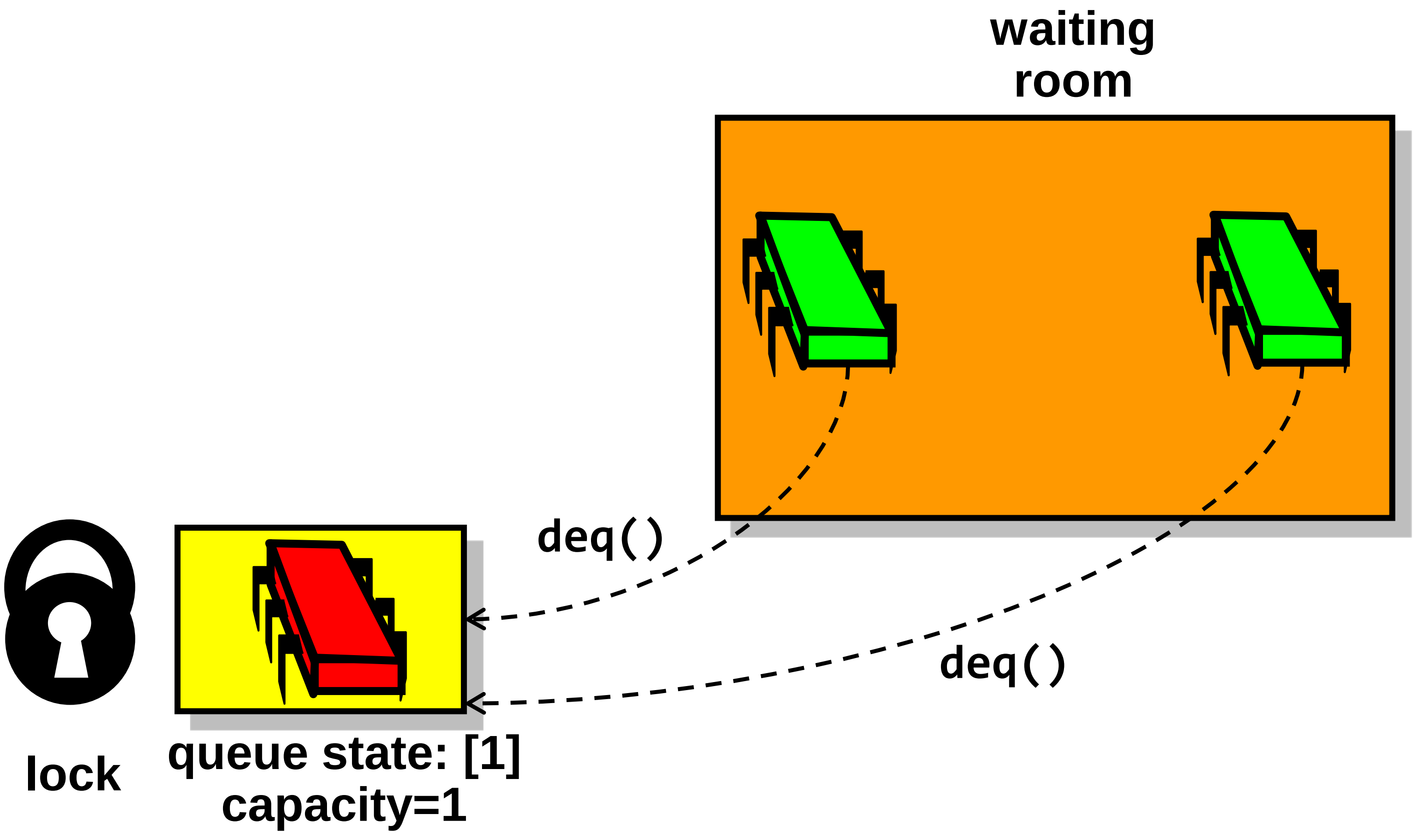
Lost Wakeup in Simplified Queue with `notify()`



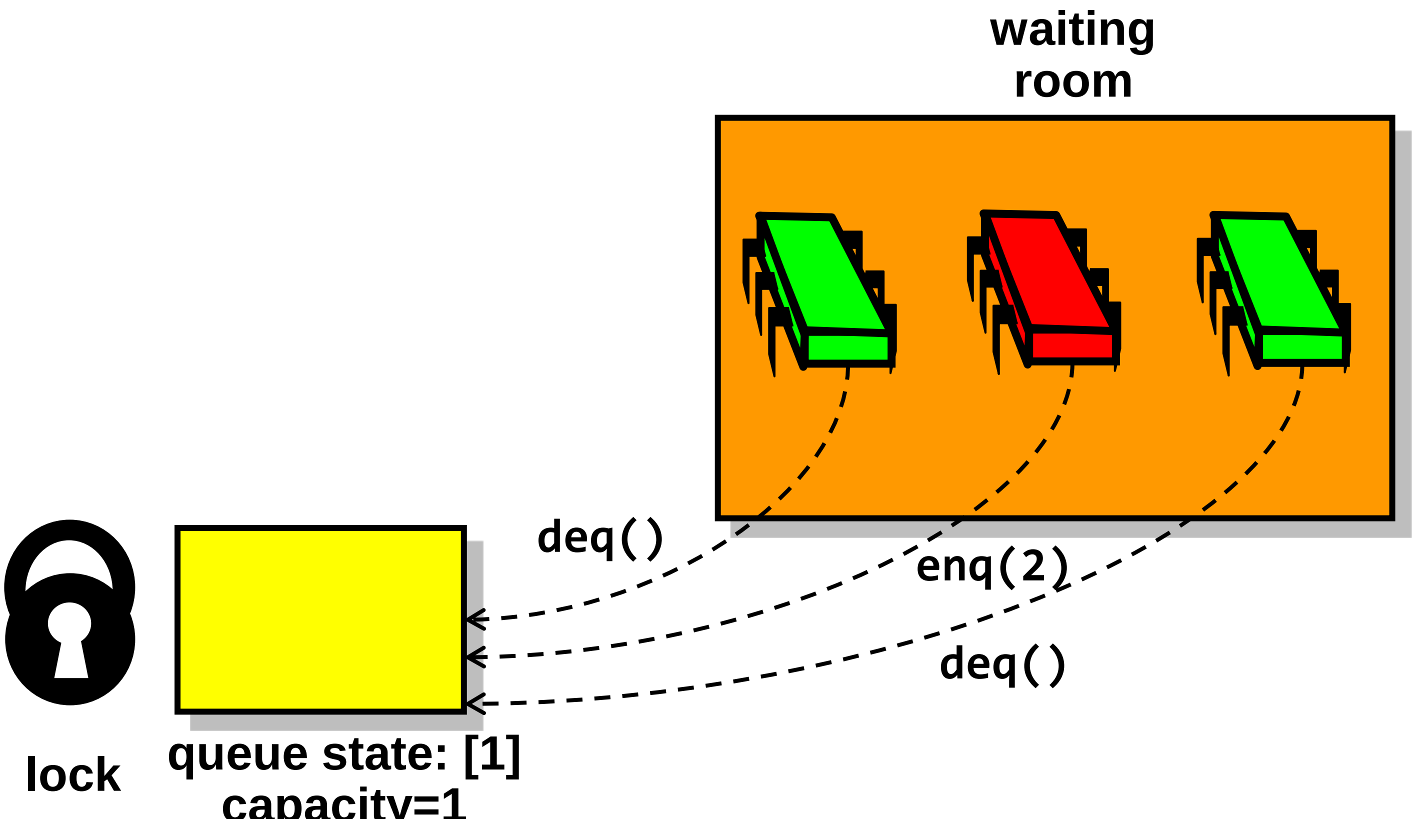
Lost Wakeup in Simplified Queue with `notify()`



Lost Wakeup in Simplified Queue with `notify()`



Lost Wakeup in Simplified Queue with `notify()`



remaining threads are stuck!

Readers-Writers Lock

- shared objects often have the property that their methods can be partitioned into
  - readers: return information about the object
  - writers: actually modify the object
- no need for readers to synchronize with each other

Readers-Writers Lock

```
public interface ReadWriteLock {
   Lock readLock();
   Lock writeLock();
}
```

Readers-Writers Lock

```
public SimpleReadWriteLock implements
  ReadWriteLock {
  int readers = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

Readers-Writers Lock

```
public SimpleReadWriteLock implements
  ReadWriteLock {
  int readers = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

number of current readers

Readers-Writers Lock

```
public SimpleReadWriteLock implements
  ReadWriteLock {
  int readers = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

is there a writer?

Readers-Writers Lock

```
public SimpleReadWriteLock implements
  ReadWriteLock {
  int readers = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

protects internal state of this lock

Readers-Writers Lock

```
public SimpleReadWriteLock implements
  ReadWriteLock {
  int readers = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

condition to wait on if lock is taken

Readers-Writers Lock

```
public SimpleReadWriteLock implements
  ReadWriteLock {
  int readers = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

the actual read and write locks

(implemented by inner classes)

Inner ReadLock class

```
class ReadLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) {
        condition.await();
      }
      readers++;
    } finally { lock.unlock(); }
  }
  ... }
```

Inner ReadLock class

```
class ReadLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) {
        condition.await();
      }
      readers++;
    } finally { lock.unlock(); }
  }
  ... }
```

wait until no writer holds the lock

Inner ReadLock class

```
class ReadLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) {
        condition.await();
      }
      readers++;
    } finally { lock.unlock(); }
  }
  ... }
```

increase the number of readers

Inner ReadLock class

```
class ReadLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      readers--;
      if (readers == 0)
        condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

Inner ReadLock class

```
class ReadLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      readers--;
      if (readers == 0)
        condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

decrease the number of readers

Inner ReadLock class

```
class ReadLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      readers--;
      if (readers == 0)
        condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

no more readers, then wake up waiting writers

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (readers > 0 || writer) {
        condition.await();
      }
      writer = true;
    } finally { lock.unlock(); }
  }
  ... }
```

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (readers > 0 || writer) {
        condition.await();
      }
      writer = true;
    } finally { lock.unlock(); }
  }
  ... }
```

wait until lock is free

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (readers > 0 || writer) {
        condition.await();
      }
      writer = true;
    } finally { lock.unlock(); }
  }
 ... }
```

take the lock

Inner WriteLock class

```
class WriteLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      writer = false;
      condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

Inner WriteLock class

```
class WriteLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      writer = false;
      condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

release the lock

Inner WriteLock class

```
class WriteLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      writer = false;
      condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

wake up waiting readers and writers

Fair Readers-Writers Lock

- Problem with `SimpleReadWriteLock`
  - usually readers are much more frequent than writers
  - writers may be locked out for a long time
- Idea: give priority to writers

FIFO Readers-Writers Lock

```
public FifoReadWriteLock implements ReadWriteLock {
  int readAcquires = 0;
  int readReleases = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock(true);
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

FIFO Readers-Writers Lock

```
public FifoReadWriteLock implements ReadWriteLock {
  int readAcquires = 0;
  int readReleases = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock(true);
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

count releases and acquires of readers separately

FIFO Readers-Writers Lock

```
public FifoReadWriteLock implements ReadWriteLock {
  int readAcquires = 0;
  int readReleases = 0;
  boolean writer = false;
  Lock lock = new ReentrantLock(true);
  Condition condition = lock.newCondition();
  Lock readLock = new ReadLock();
  Lock writeLock = new WriteLock();
  Lock readLock() { return readLock; }
  Lock writeLock() { return writeLock; }
  ...
}
```

create FIFO lock

Inner ReadLock class

```
class ReadLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) {
        condition.await();
      }
      readAcquires++;
    } finally { lock.unlock(); }
  }
  ... }
```

Inner ReadLock class

```
class ReadLock {
  ...
  public void unlock() {
    lock.lock();
    try {
      readReleases++;
      if (readReleases == ReadAcquires)
        condition.signalAll();
    } finally { lock.unlock(); }
  }
}
```

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) condition.await();
      writer = true;
      while (readAcquires != readReleases)
        condition.await();
    } finally { lock.unlock(); }
  }
  ... }
```

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) condition.await();
      writer = true;
      while (readAcquires != readReleases)
        condition.await();
    } finally { lock.unlock(); }
  }
 ... }
```

first wait for writers to release the lock

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) condition.await();
      writer = true;
      while (readAcquires != readReleases)
        condition.await();
    } finally { lock.unlock(); }
  }
  ... }
```

block writers and readers from acquiring the lock

Inner WriteLock class

```
class WriteLock {
  public void lock() {
    lock.lock();
    try {
      while (writer) condition.await();
      writer = true;
      while (readAcquires != readReleases)
        condition.await();
    } finally { lock.unlock(); }
  }
  ... }
```

wait for all readers who already acquired the lock to release it

Inner WriteLock class

```
class WriteLock {
  ...
  public void unlock() {
    writer = false;
    condition.signalAll();
  }
}
```