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 waiting for that condition

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();
}
```

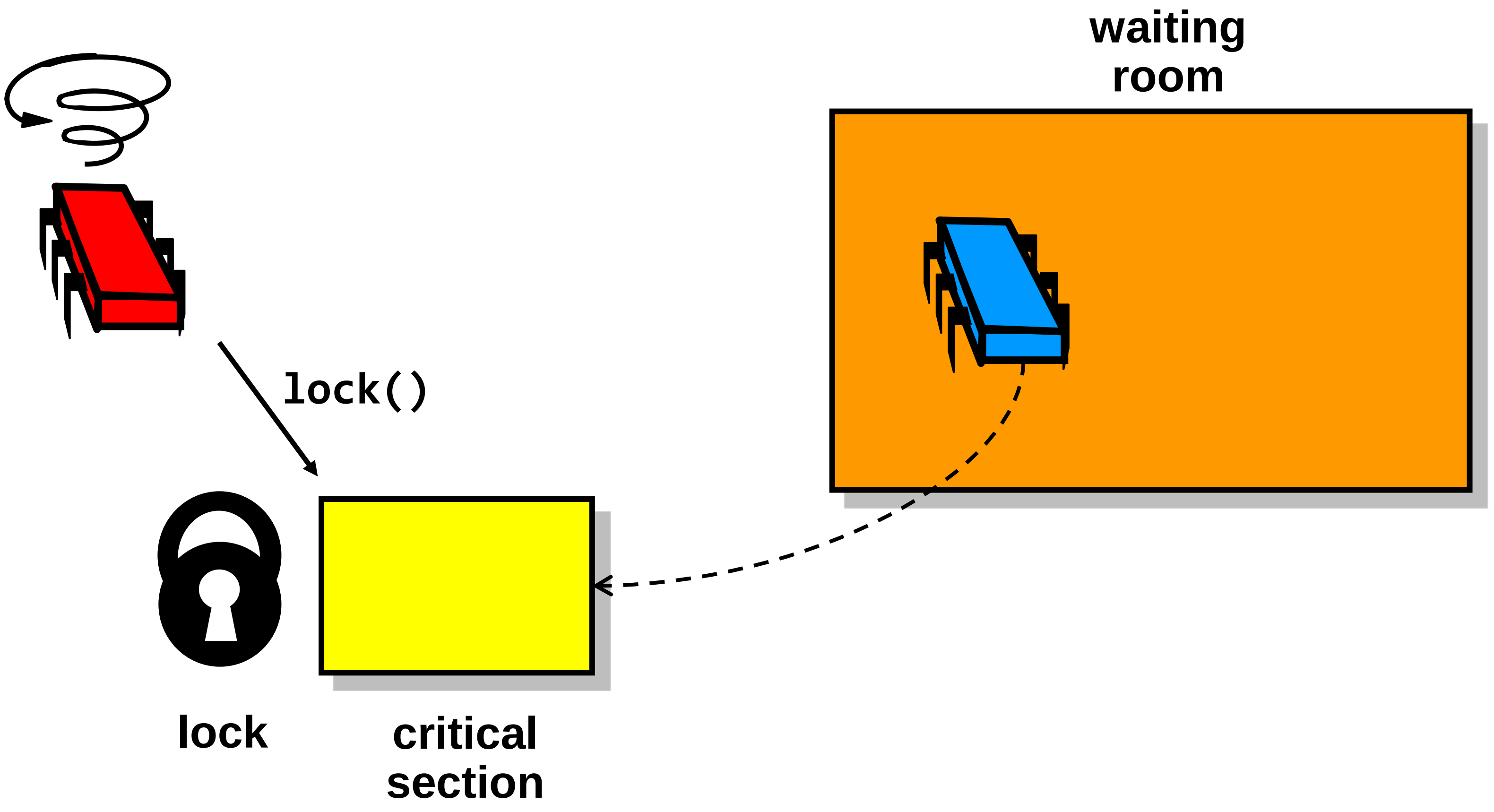
wake up **one** waiting thread

Java's Condition Interface

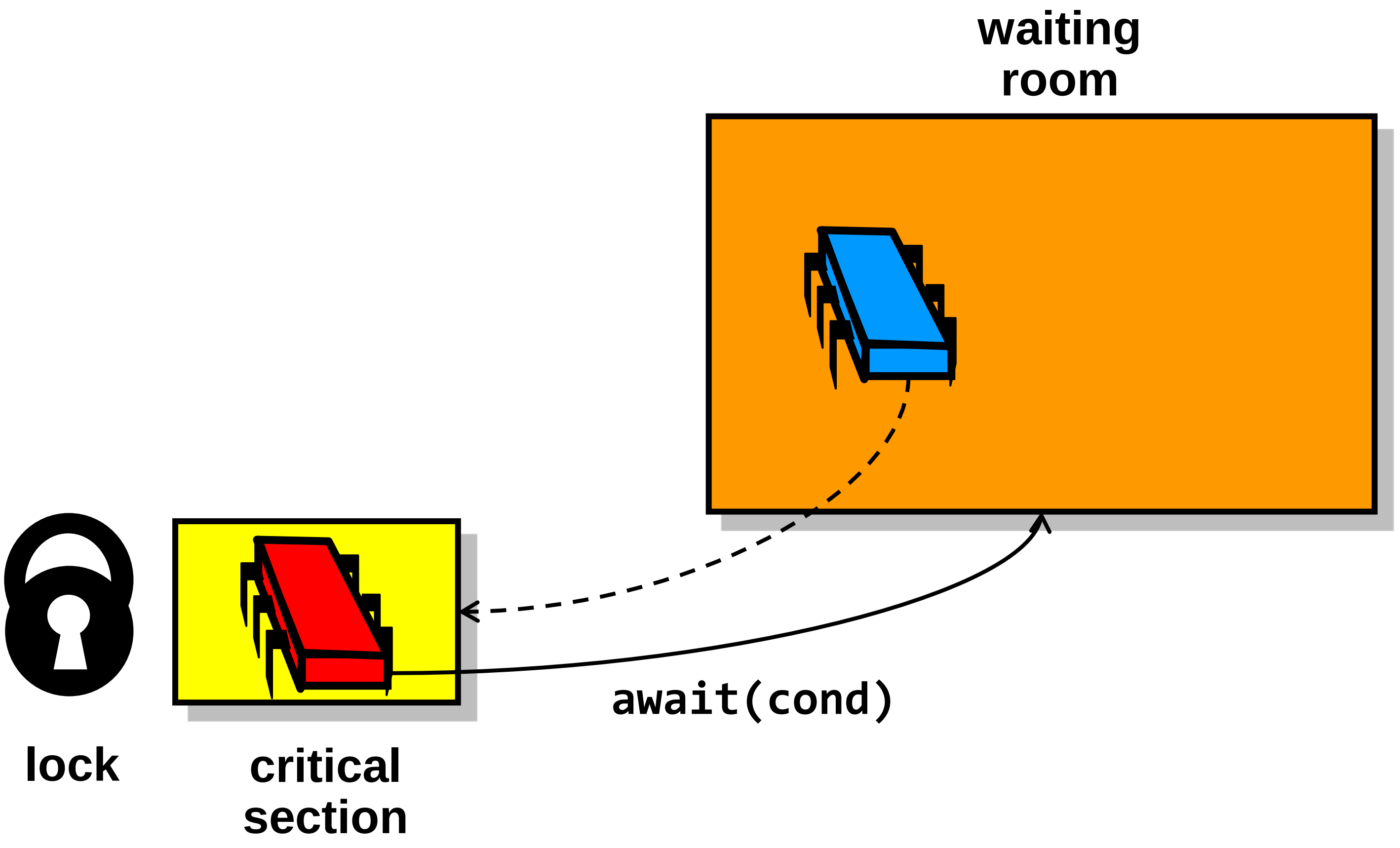
```
public interface Condition {
  void await() throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  ...
  void signal();
  void signalAll();
}
```

wake up **all** waiting threads

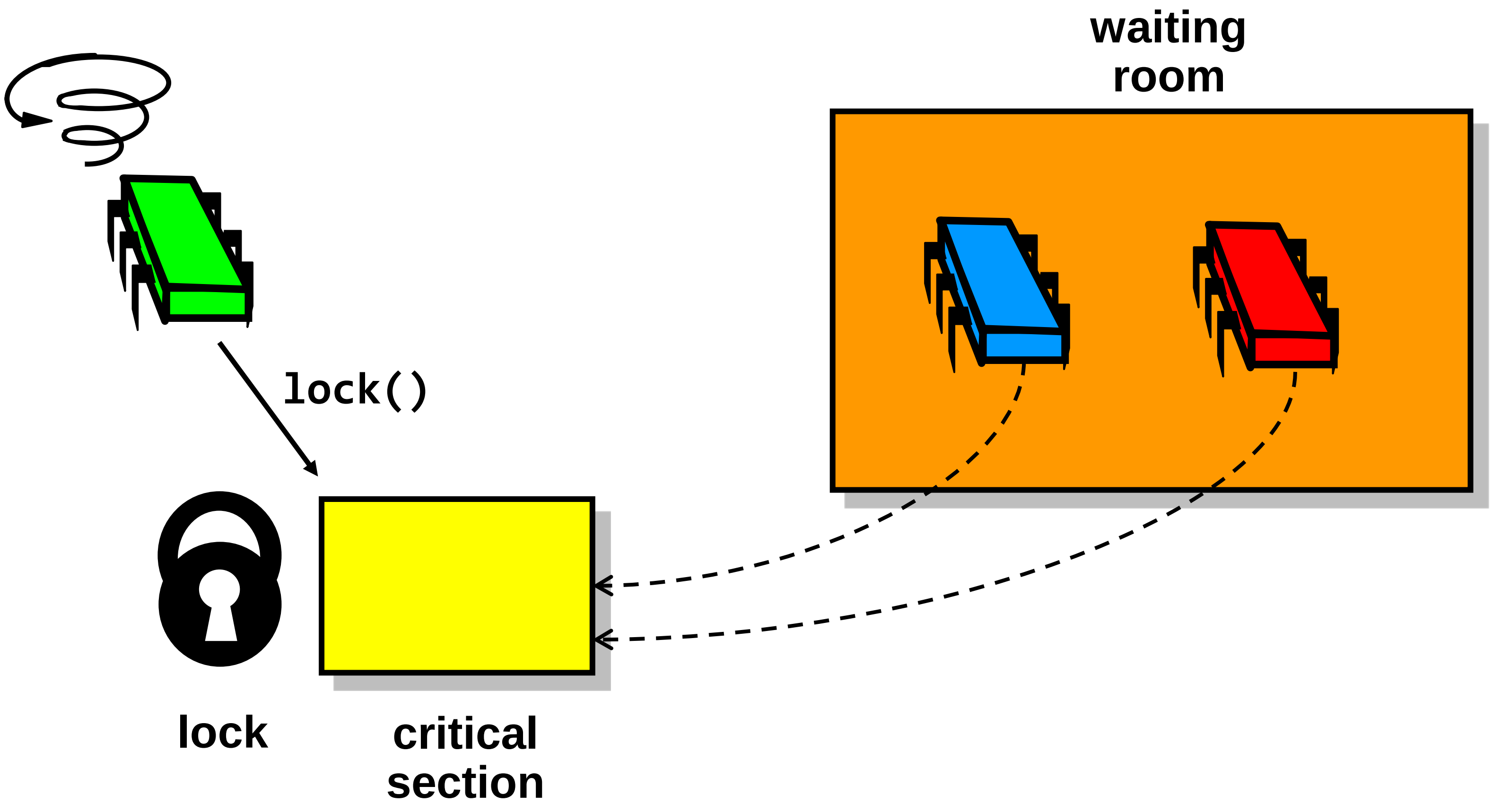
A Typical Monitor Execution



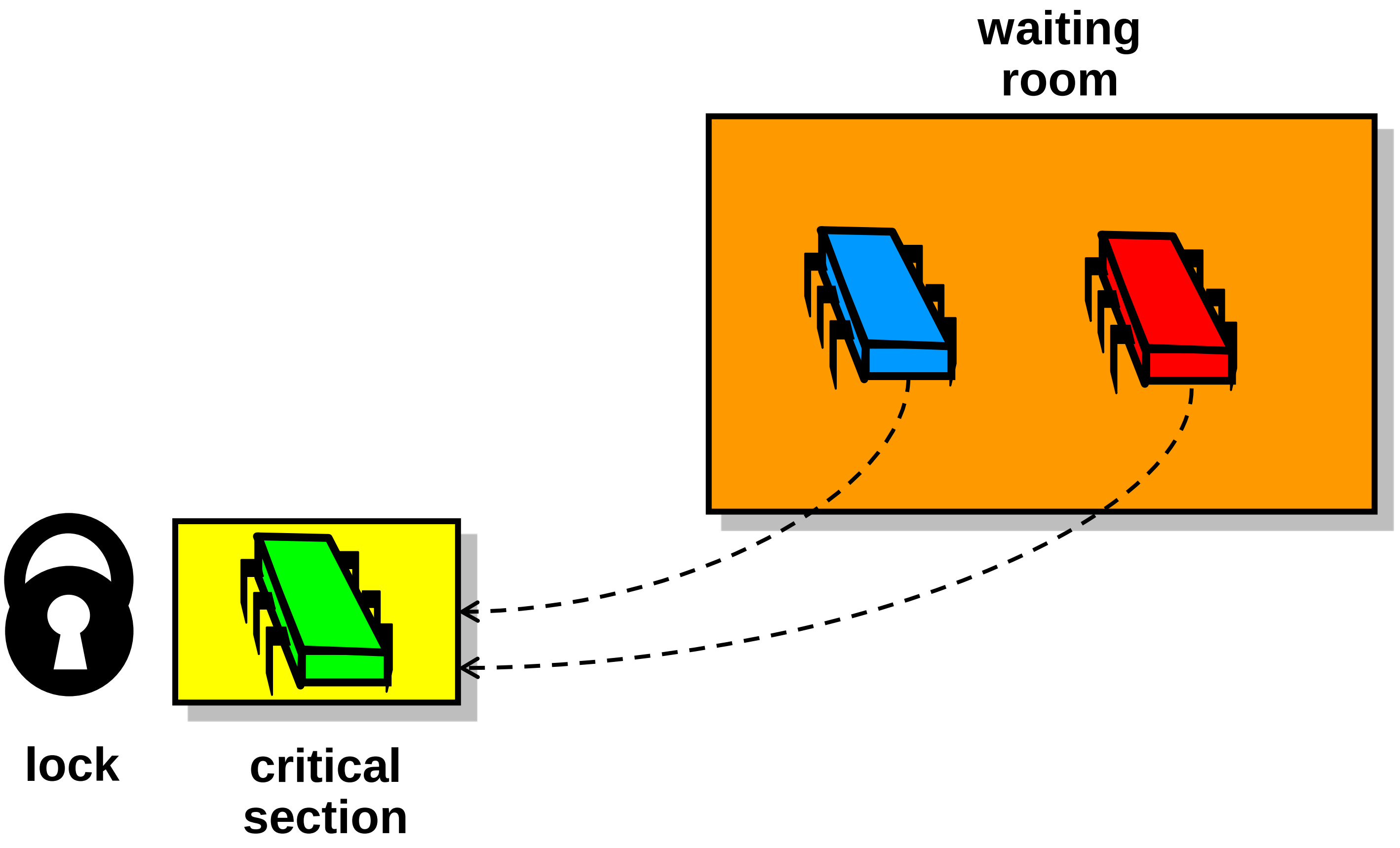
A Typical Monitor Execution



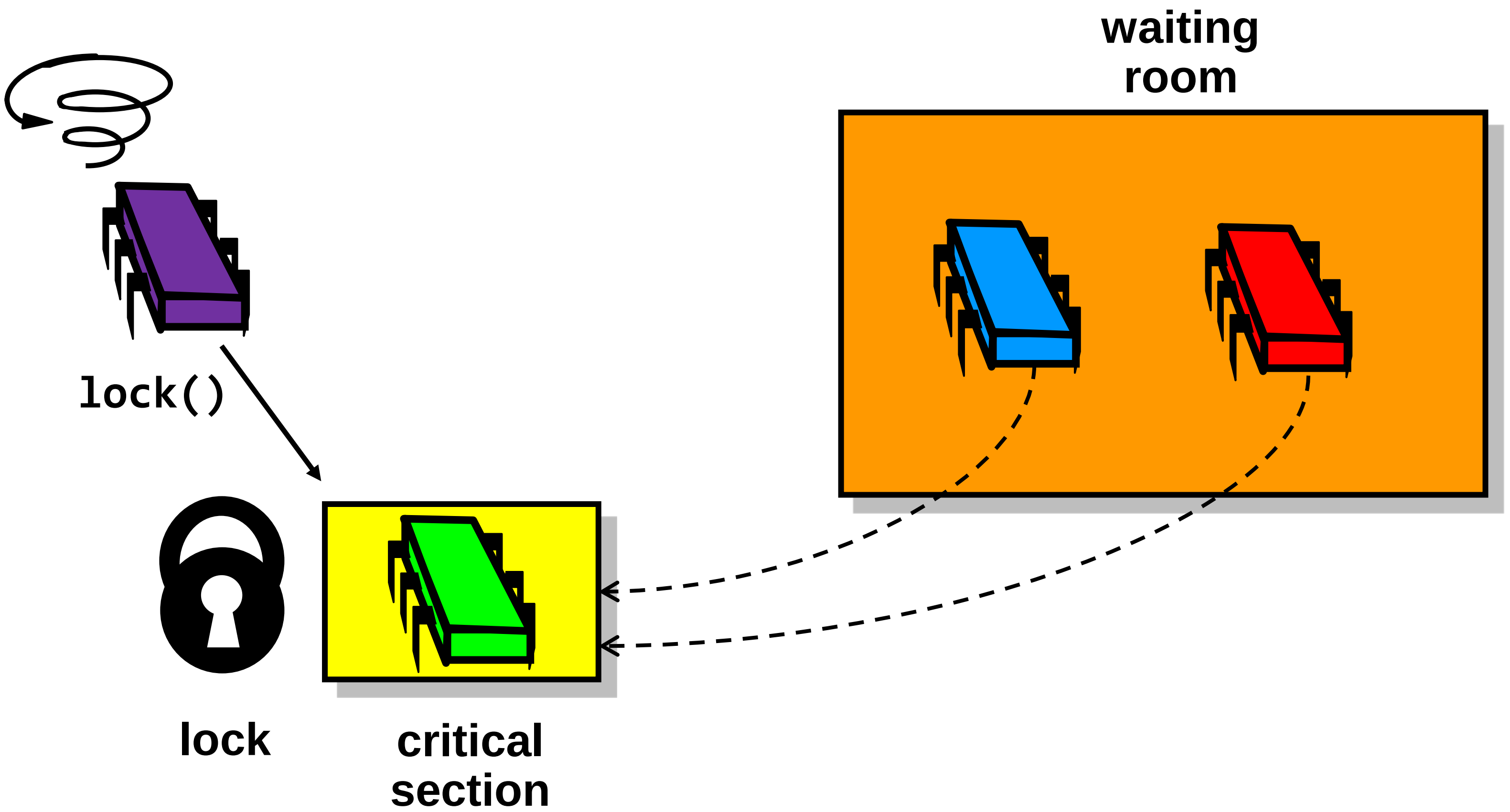
A Typical Monitor Execution



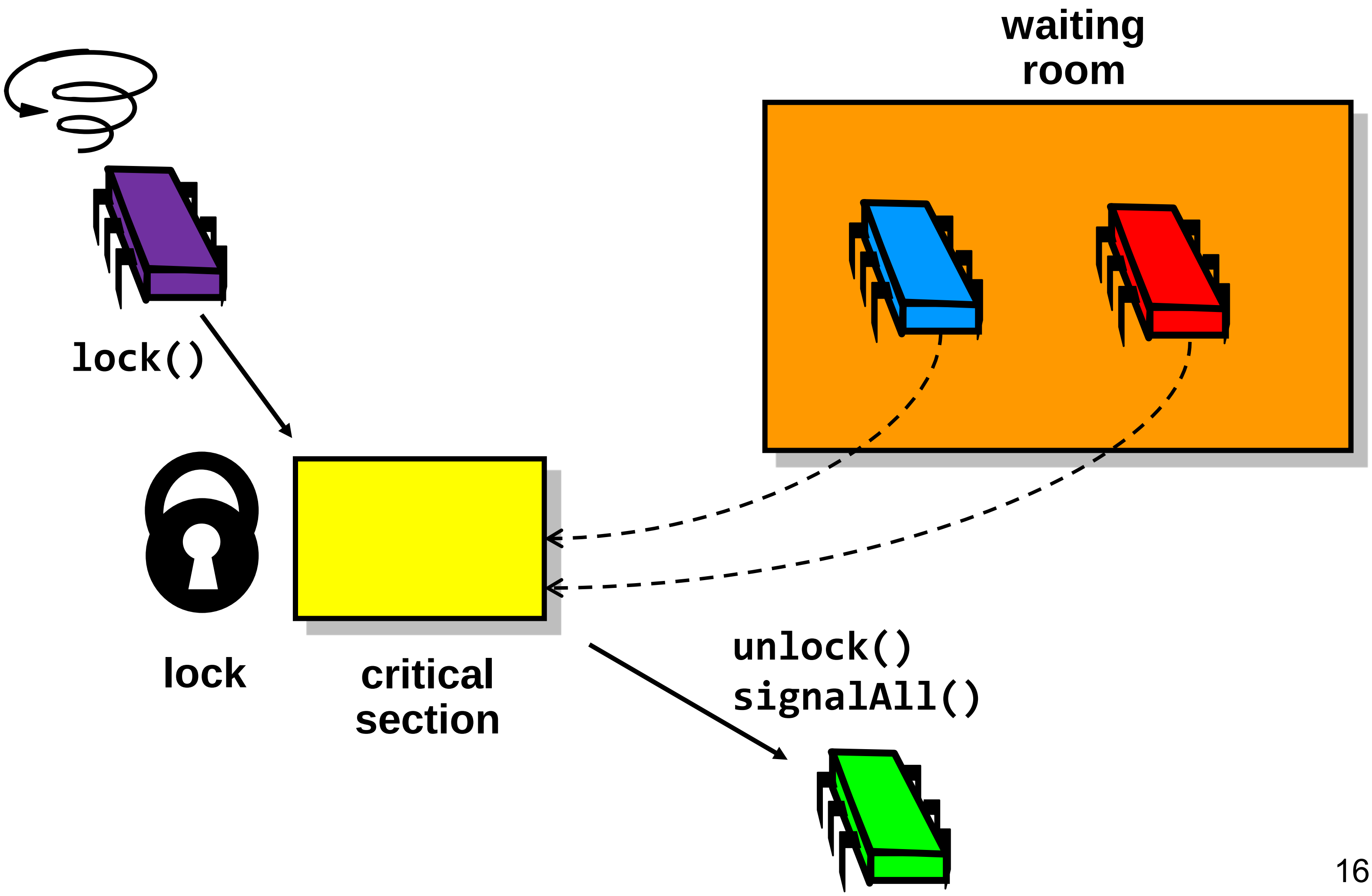
A Typical Monitor Execution



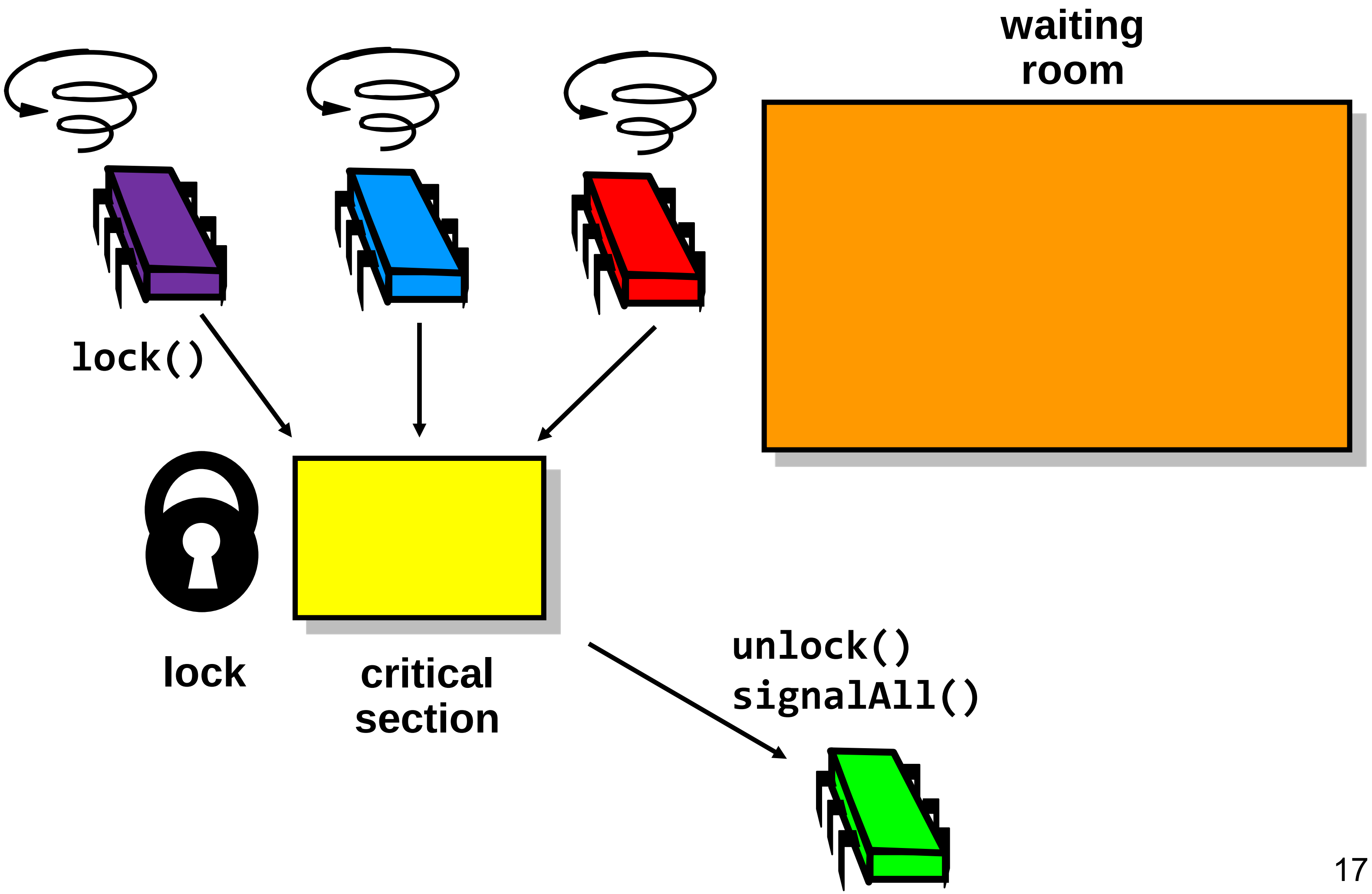
A Typical Monitor Execution



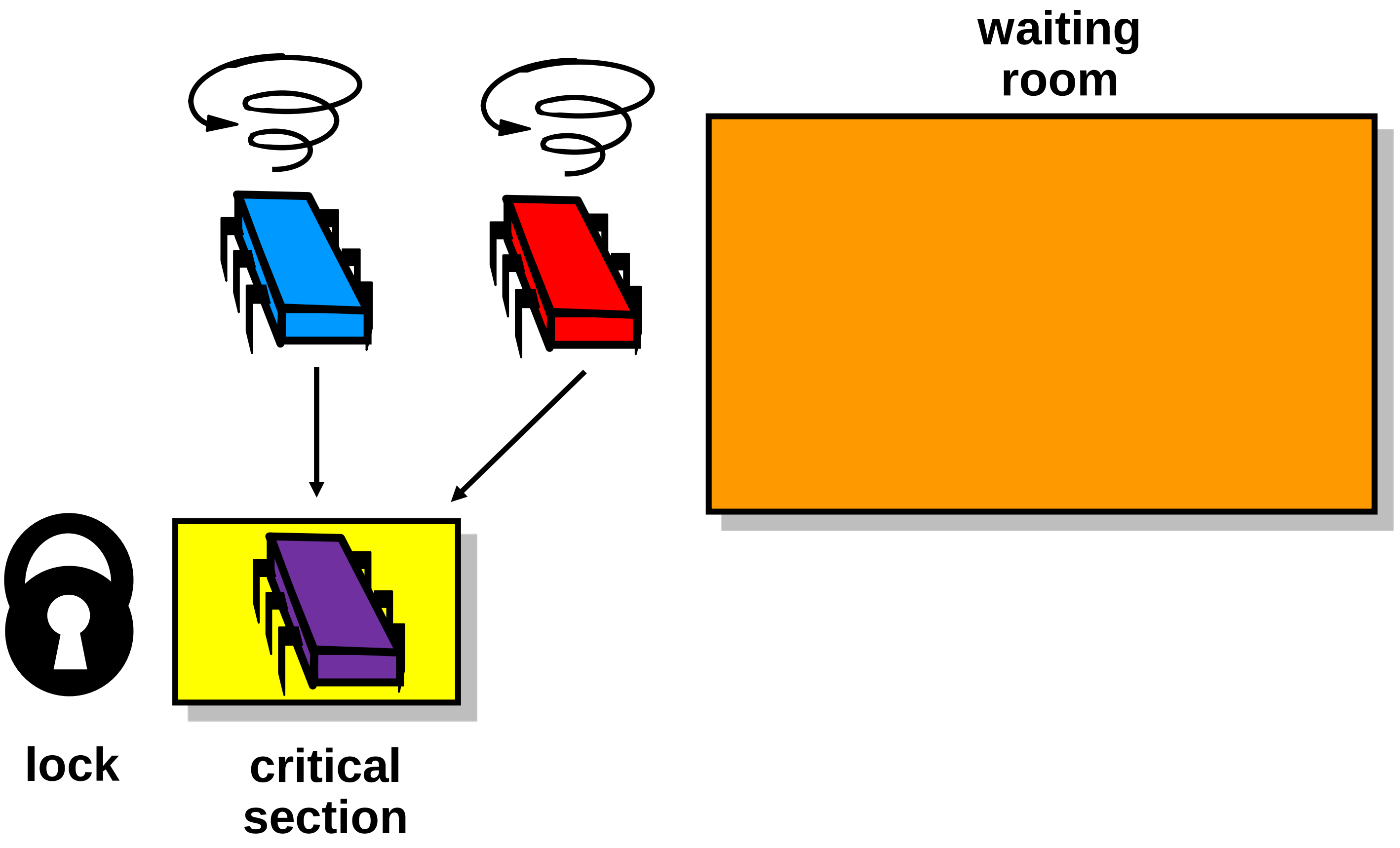
A Typical Monitor Execution



A Typical Monitor Execution



A Typical Monitor Execution



Using Condition Objects

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

Example: Blocking Queue

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

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

beware of weak memory consistency due to caching

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?

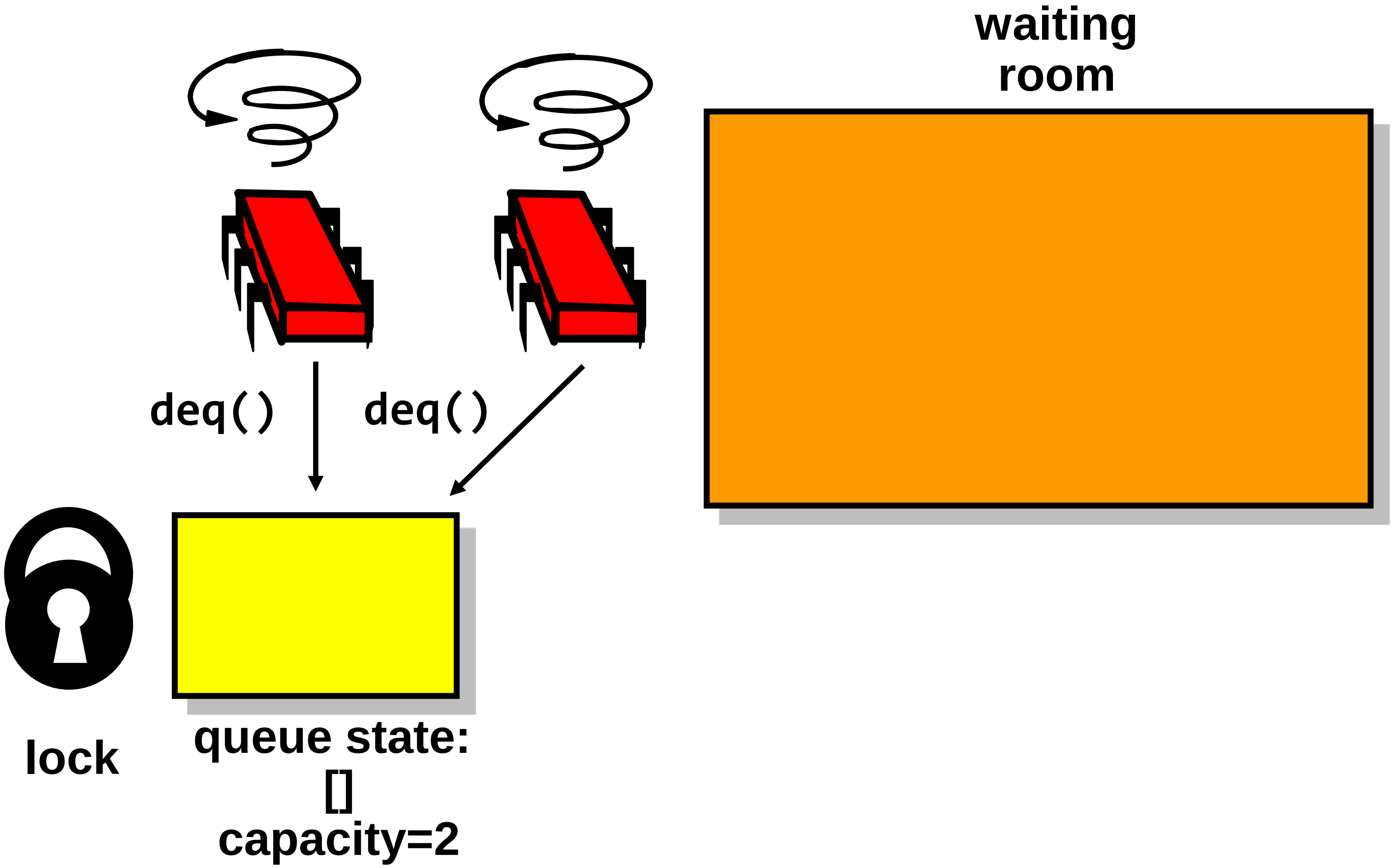
```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    if (count == 1) notEmpty.signal();
  } finally { lock.unlock(); }
}
```

Improved enqueue?

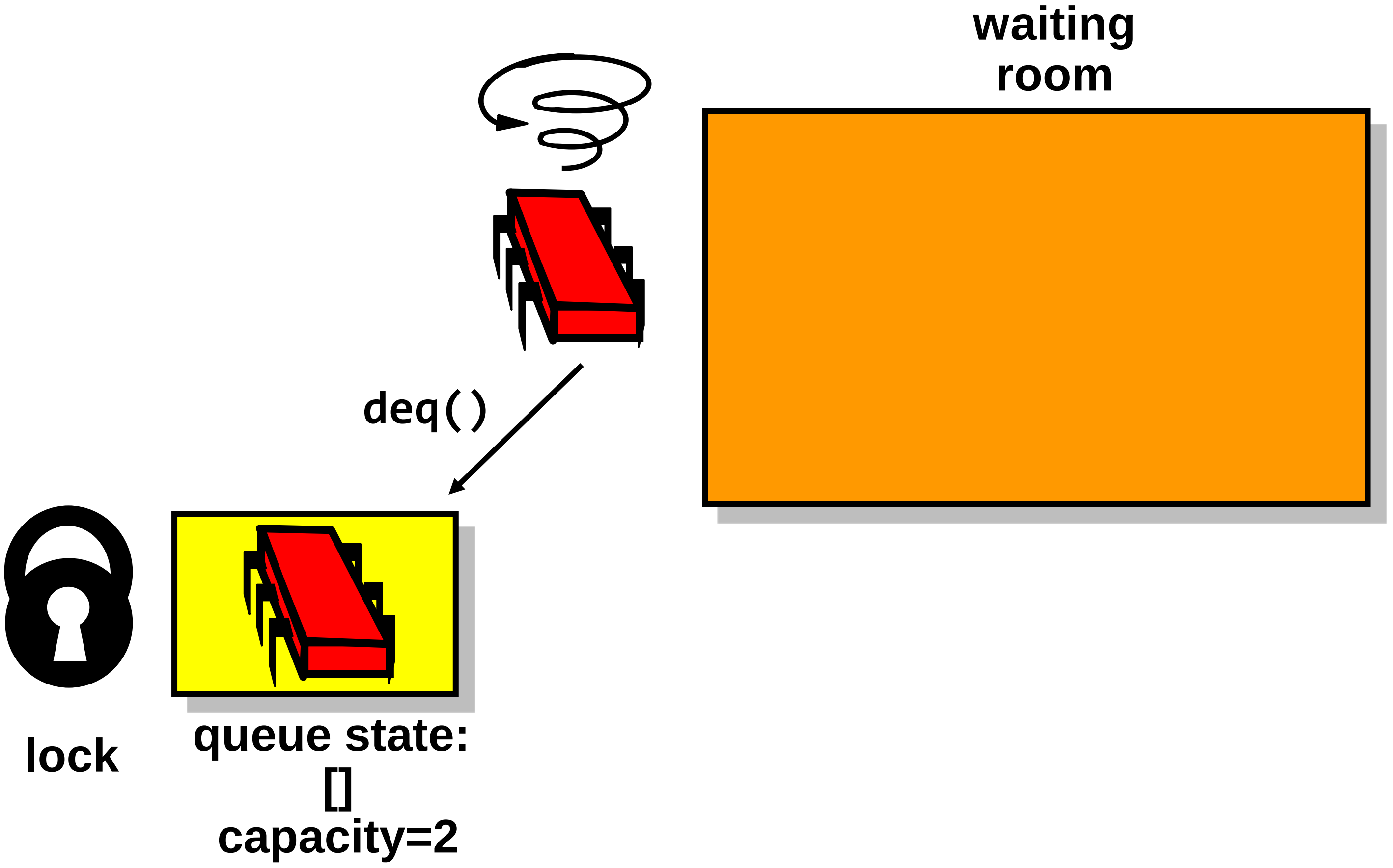
```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length())
      notFull.await();
    items[tail] = x;
    if (++tail == items.length) tail = 0;
    ++count;
    if (count == 1) notEmpty.signal();
  } finally { lock.unlock(); }
}
```

lost wakeups

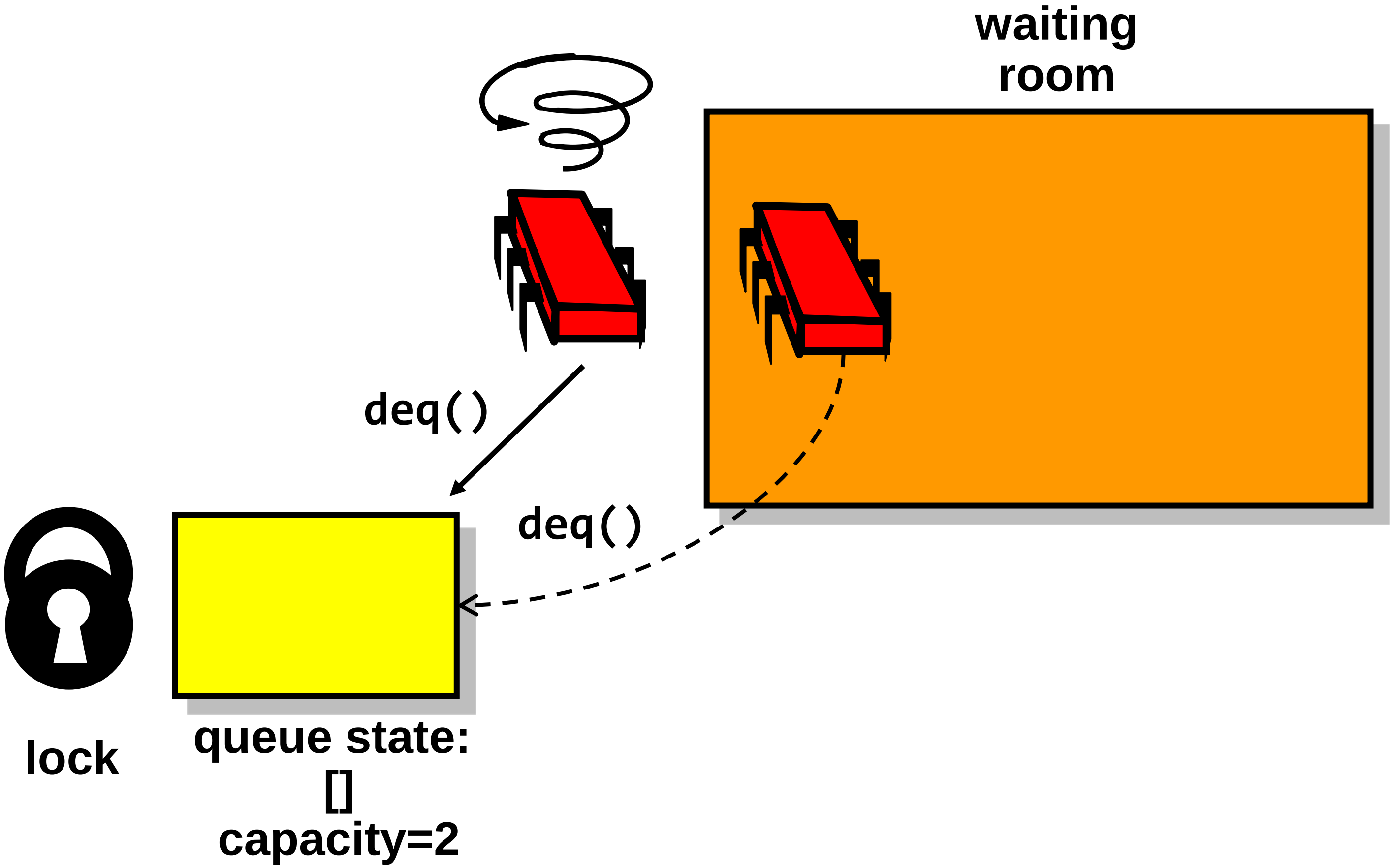
Lost Wakeup



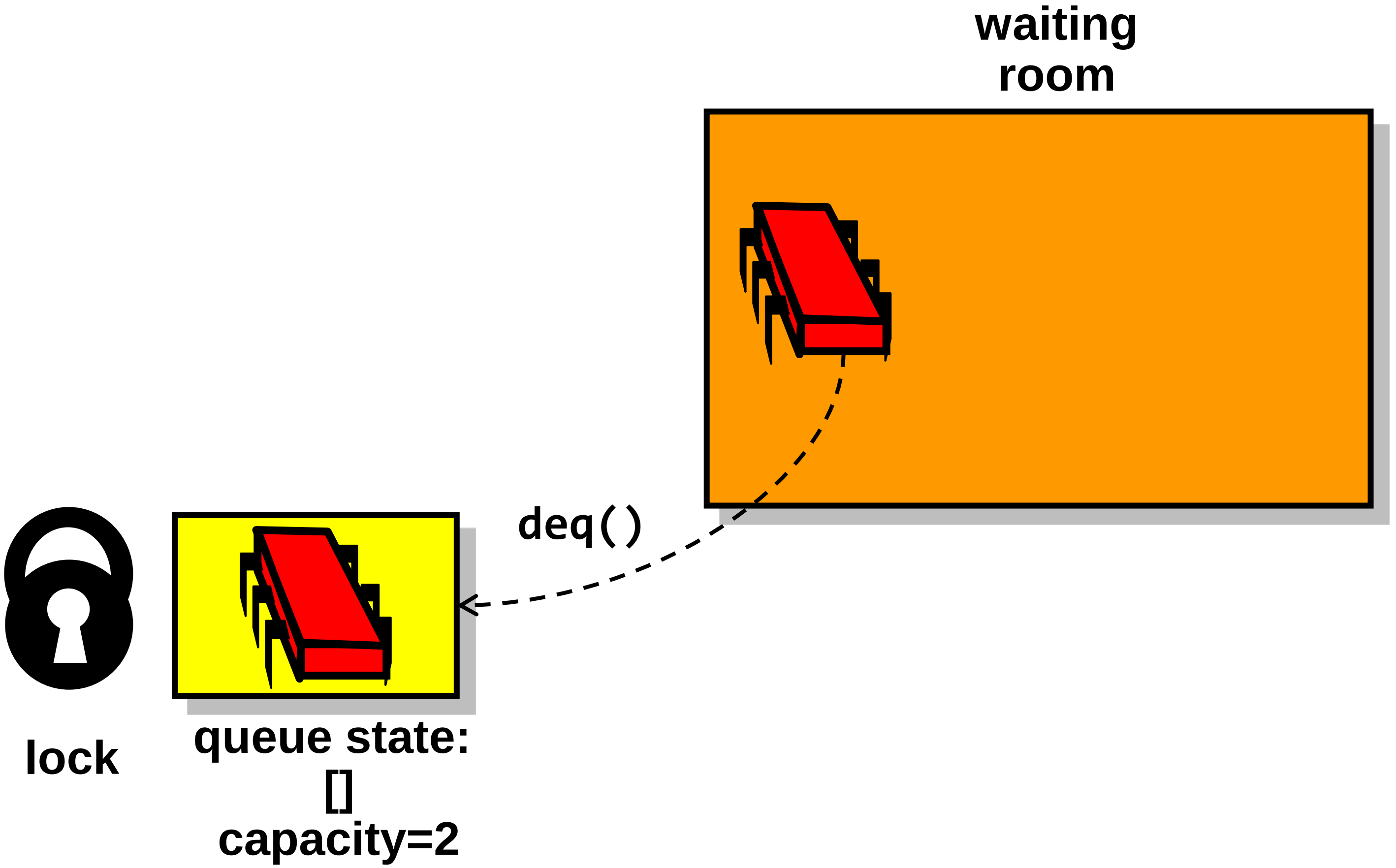
Lost Wakeup



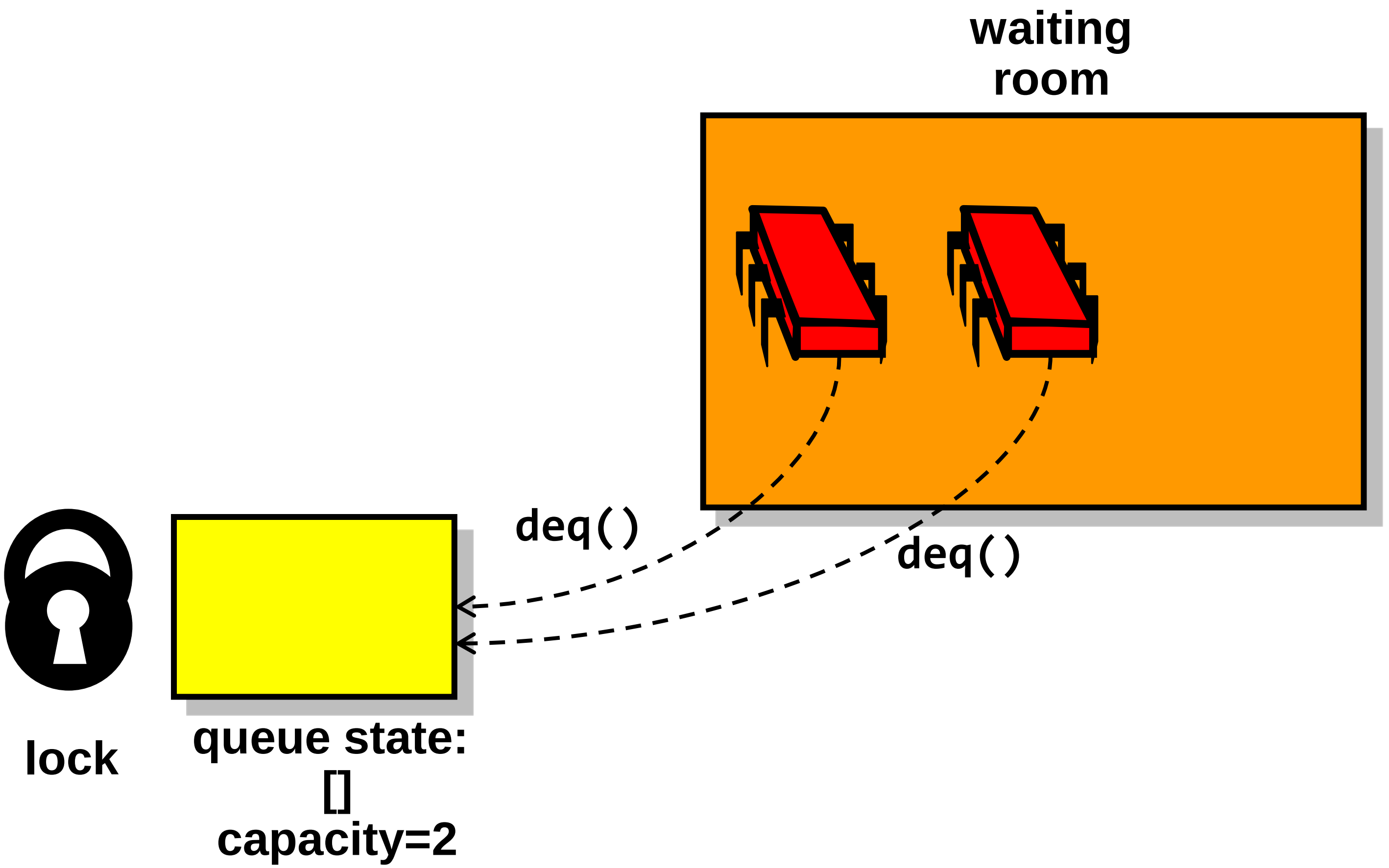
Lost Wakeup



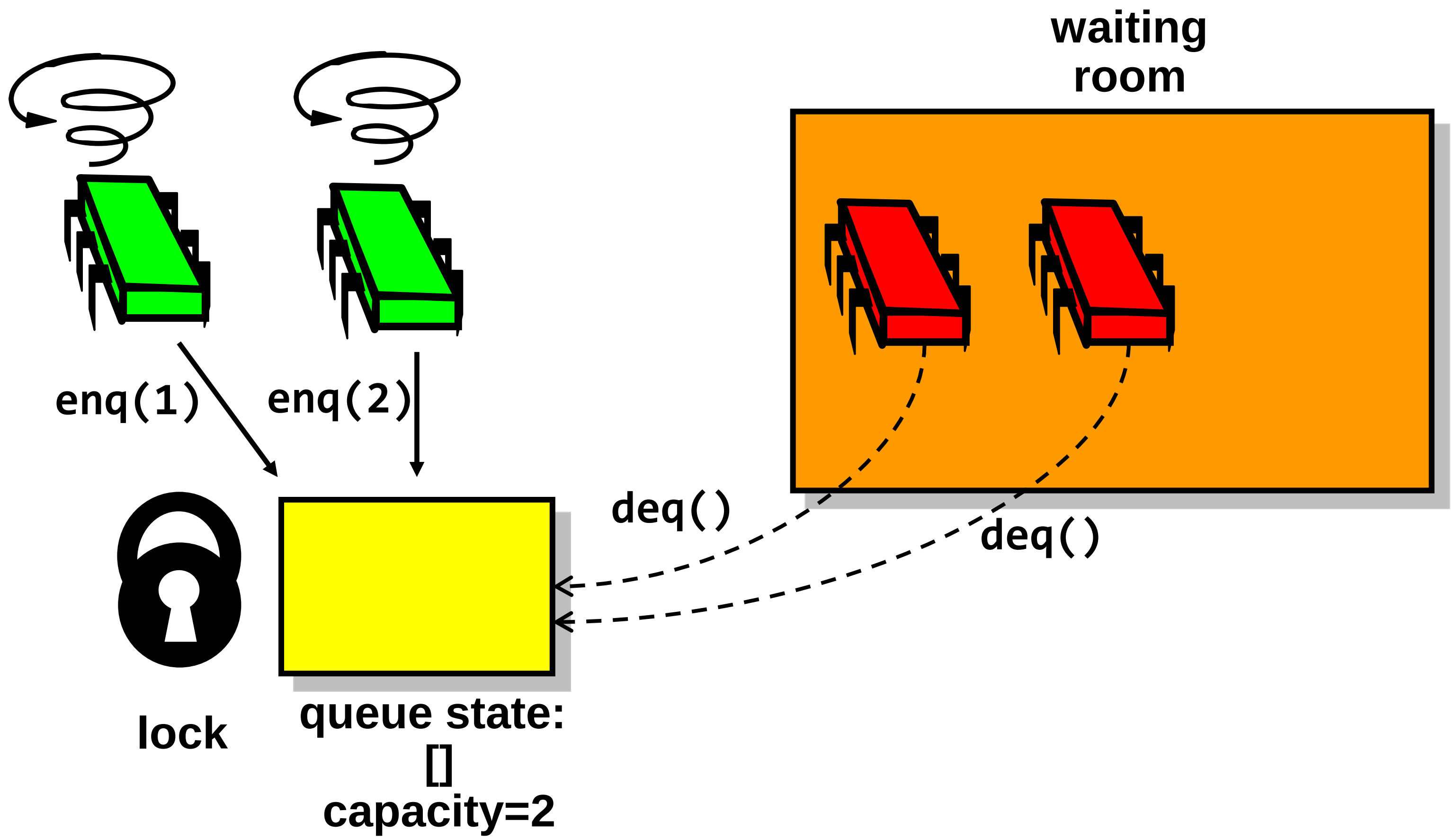
Lost Wakeup



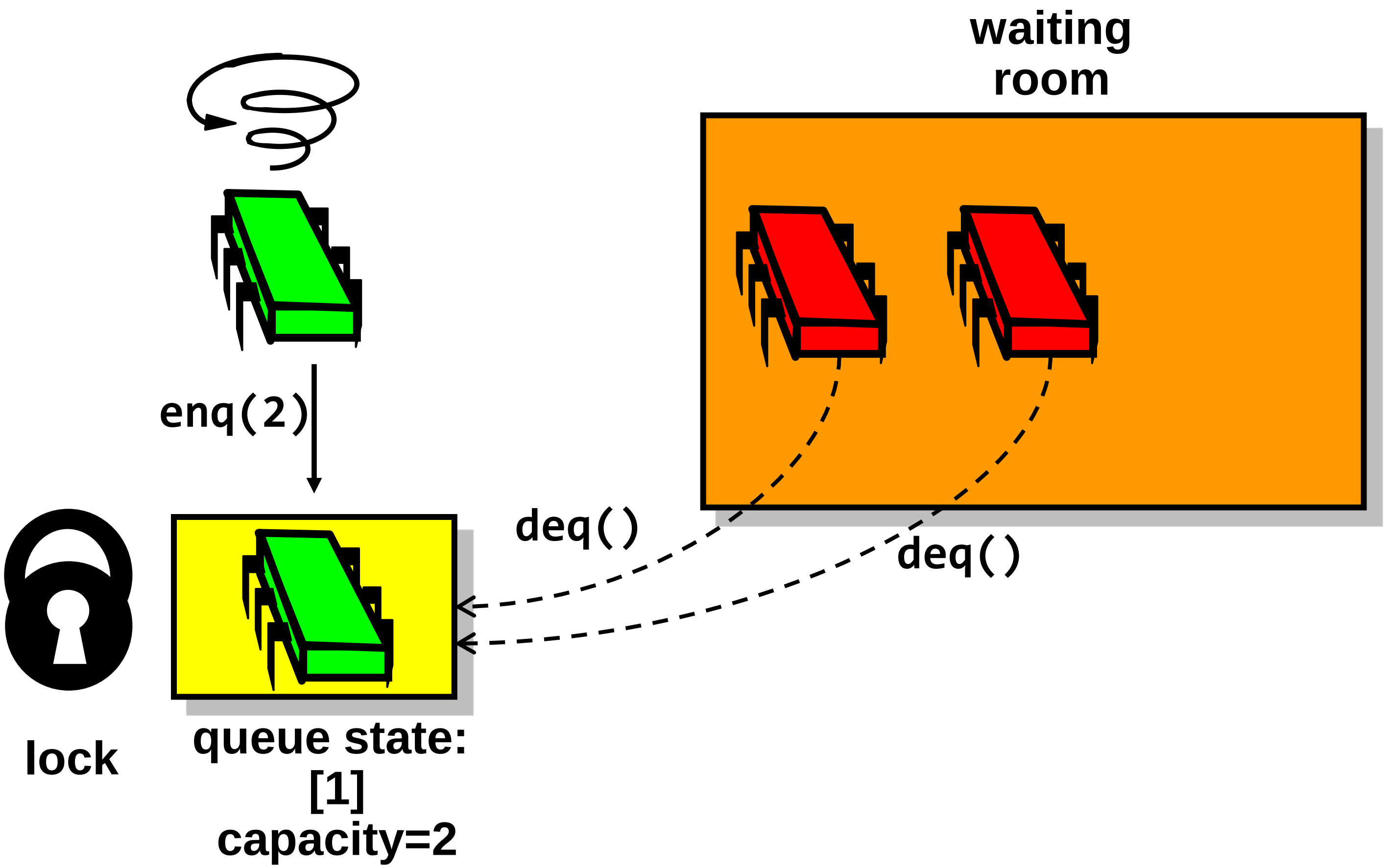
Lost Wakeup



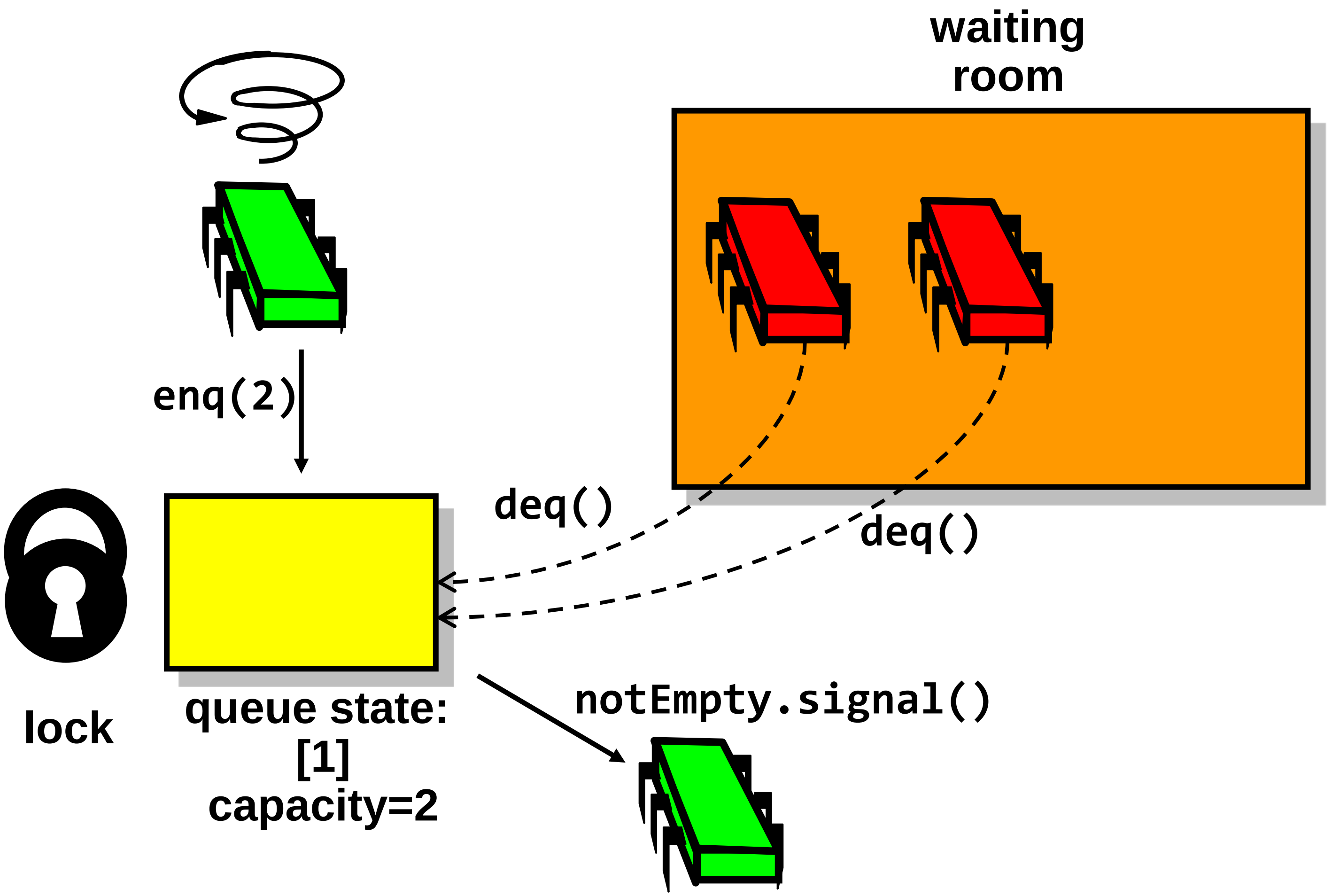
Lost Wakeup



Lost Wakeup

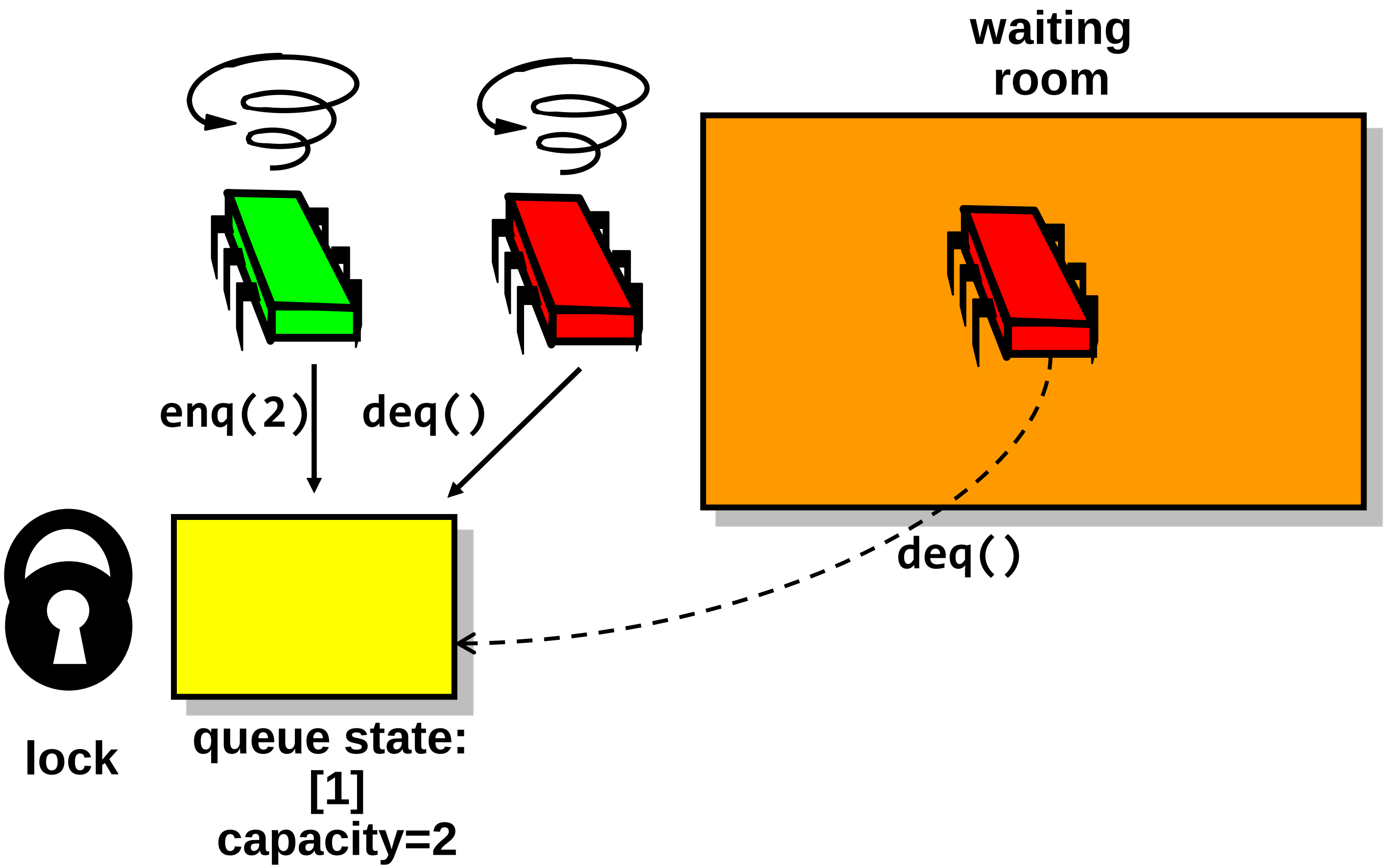


Lost Wakeup

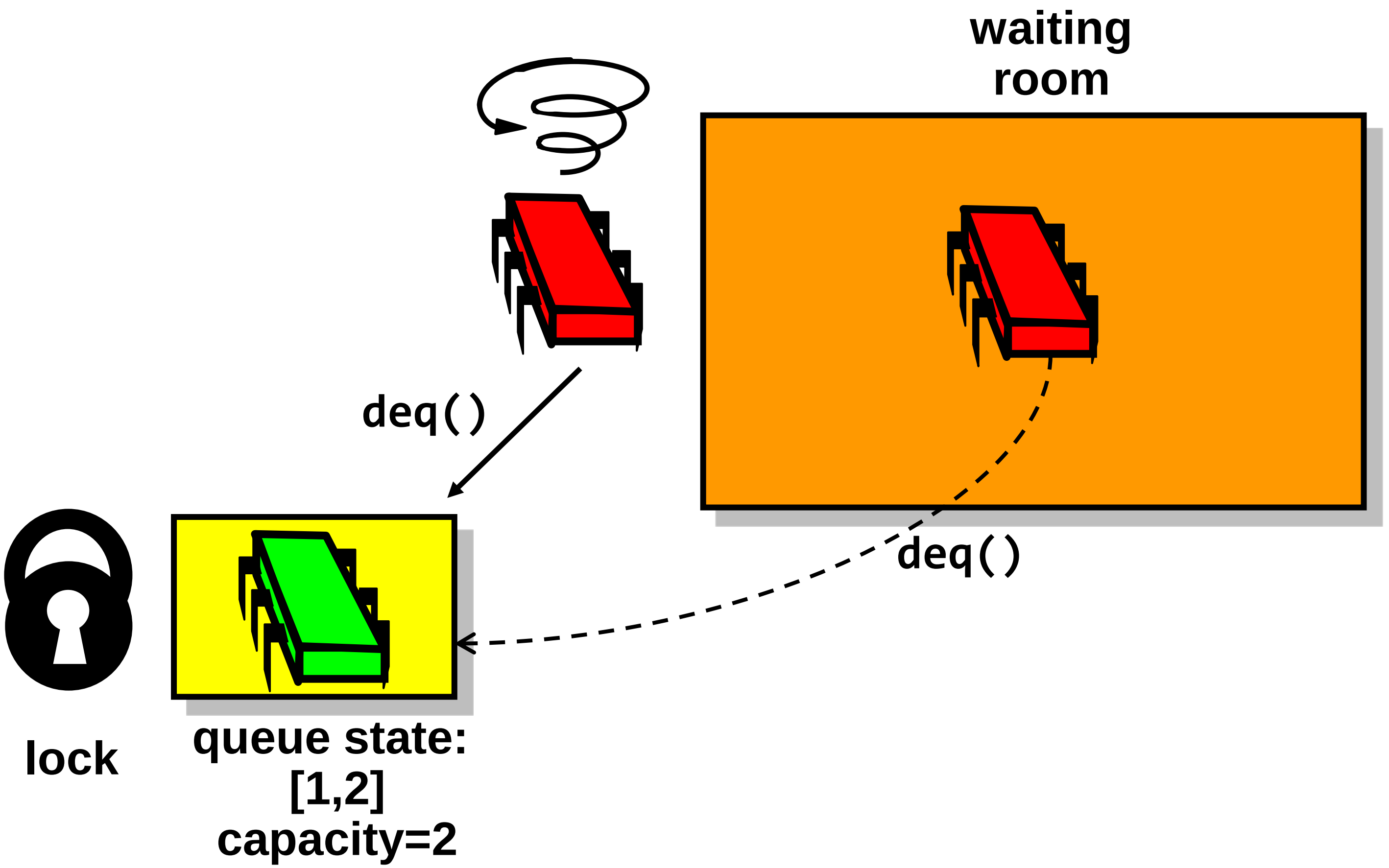


waiting
room
enq(2)
deq()
deq()
lock
queue state:
[1]
capacity=2
notEmpty.signal()

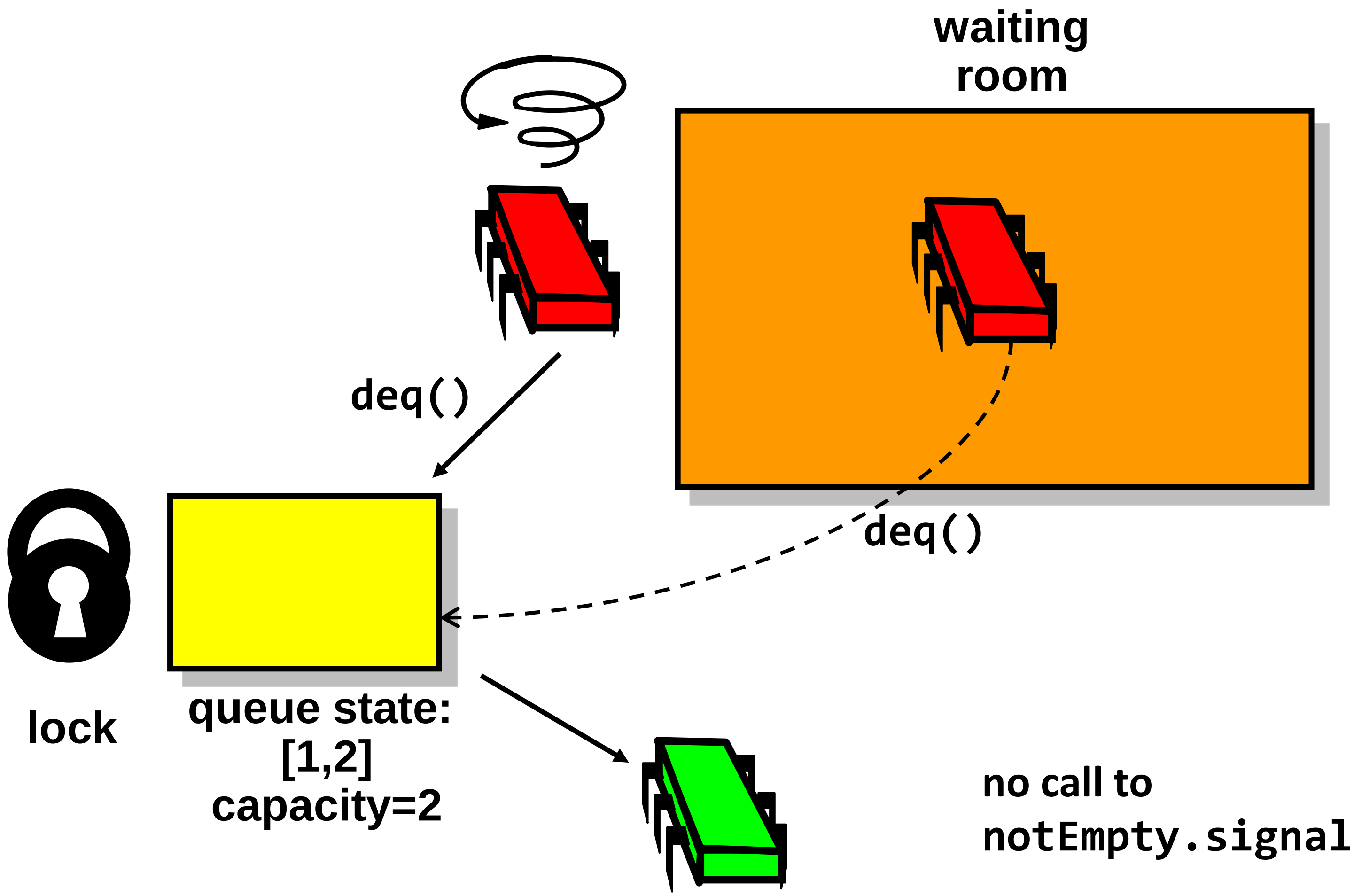
Lost Wakeup



Lost Wakeup

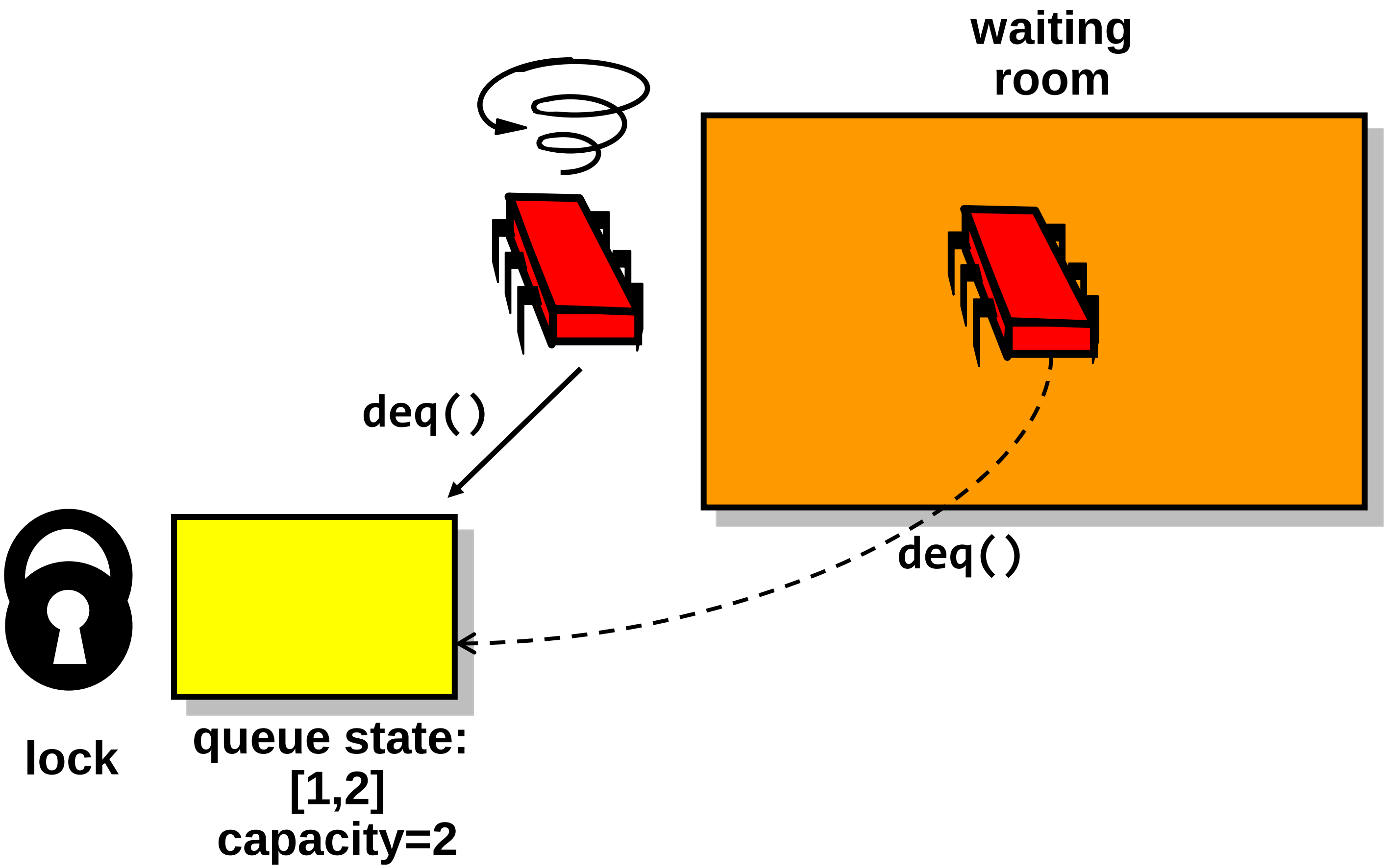


Lost Wakeup

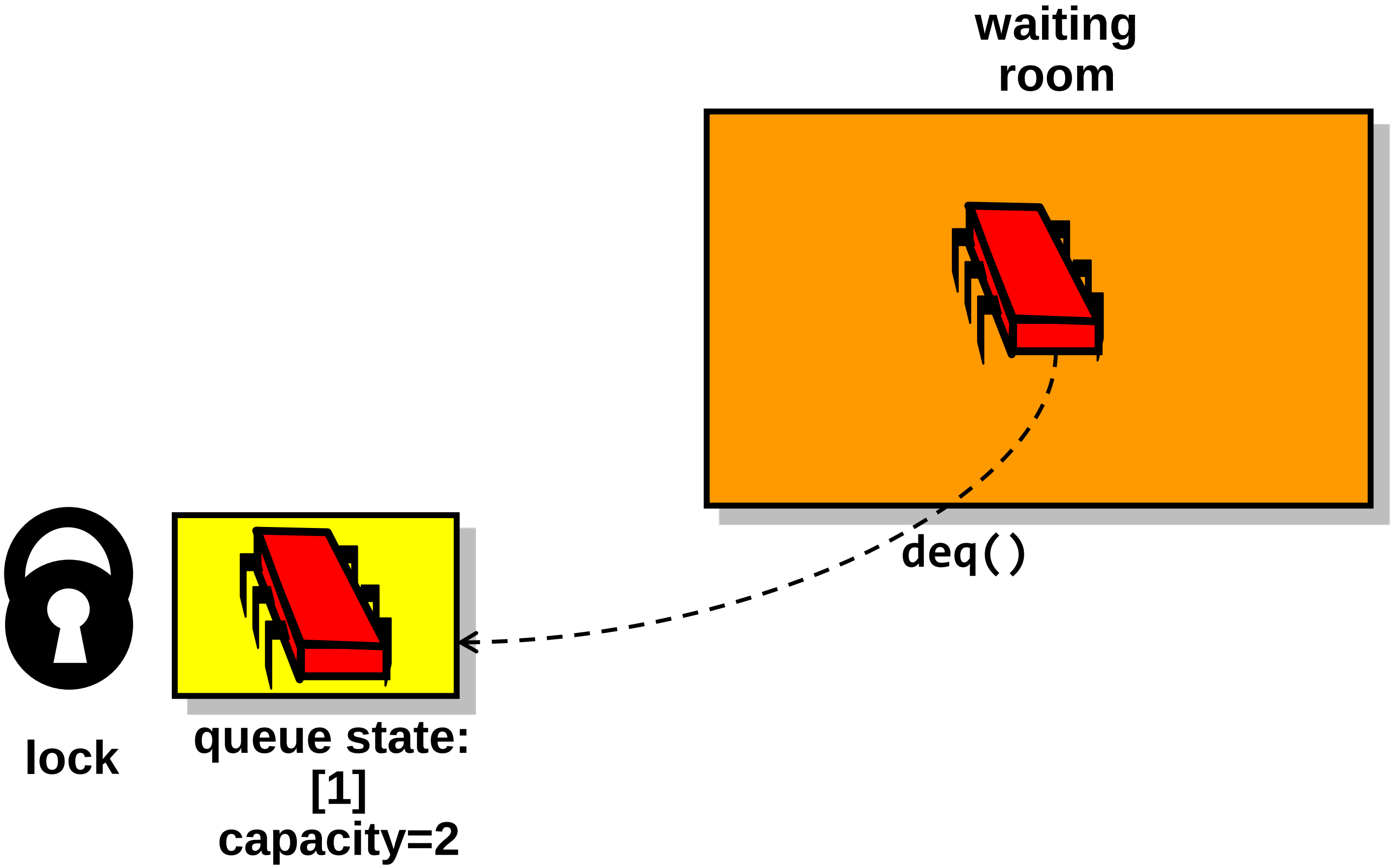


no call to
`notEmpty.signal()!`

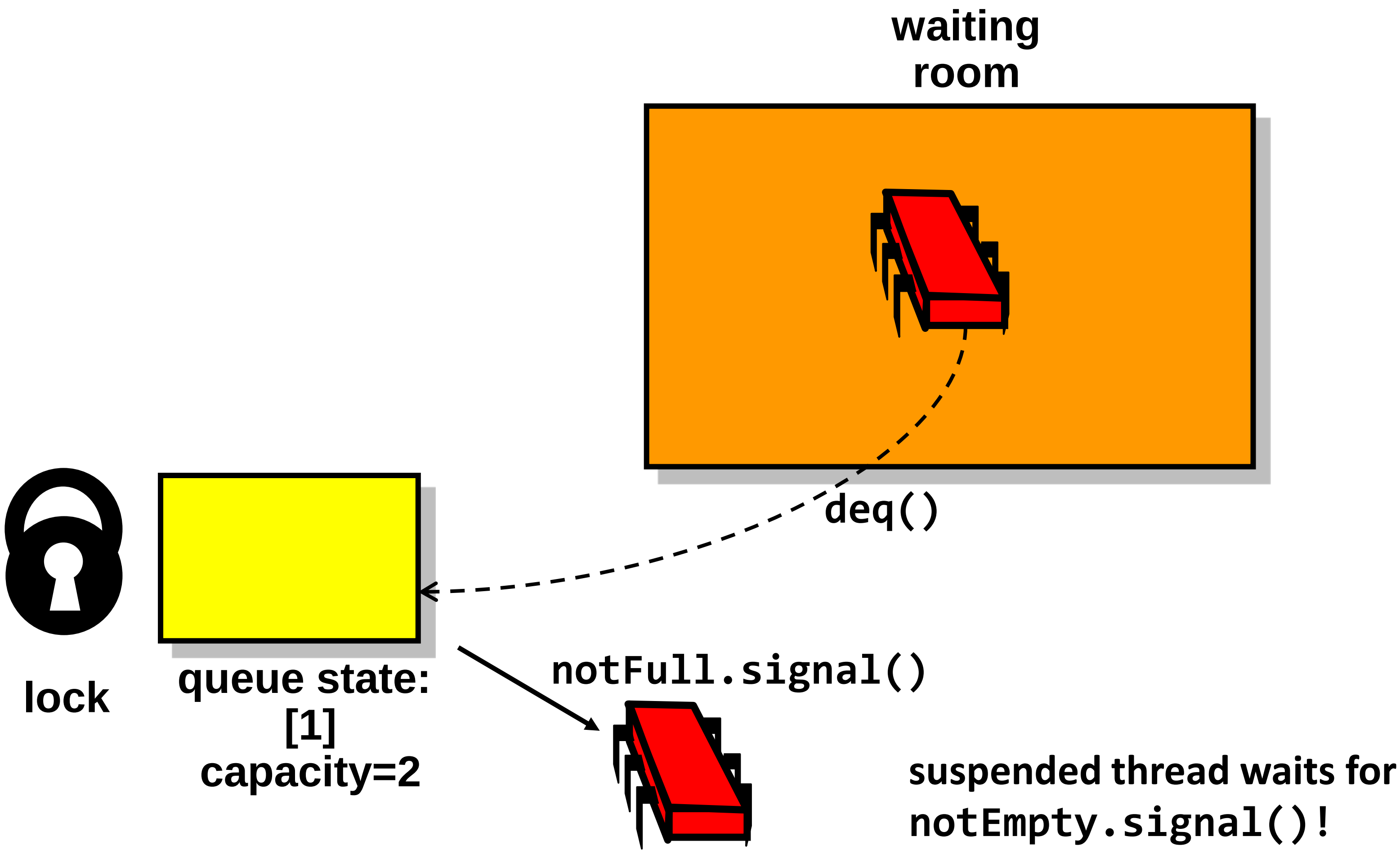
Lost Wakeup



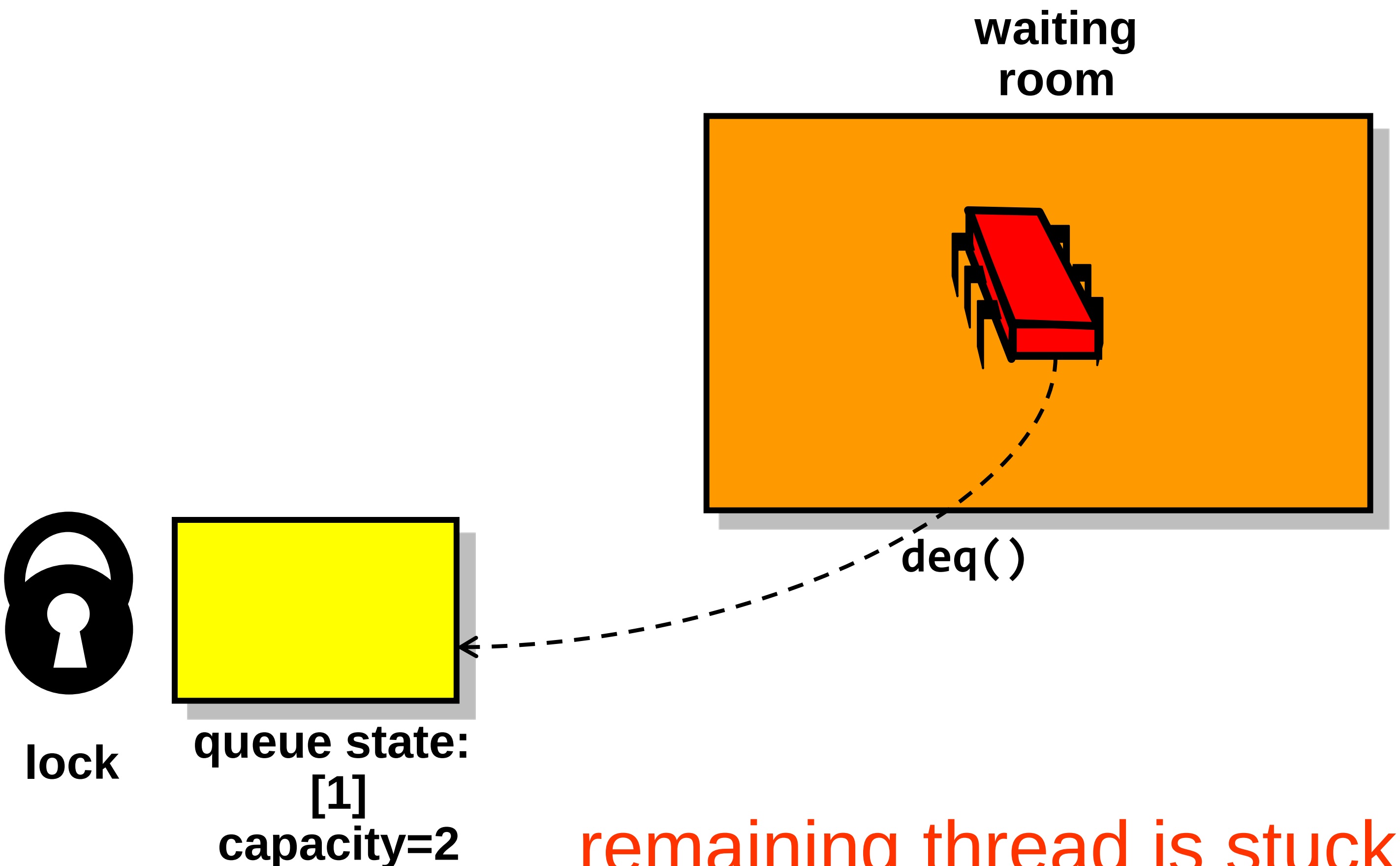
Lost Wakeup



Lost Wakeup


waiting room
deq()
lock
queue state:
[1]
capacity=2
notFull.signal()
suspended thread waits for
notEmpty.signal()!

Lost Wakeup



remaining thread is stuck!

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?

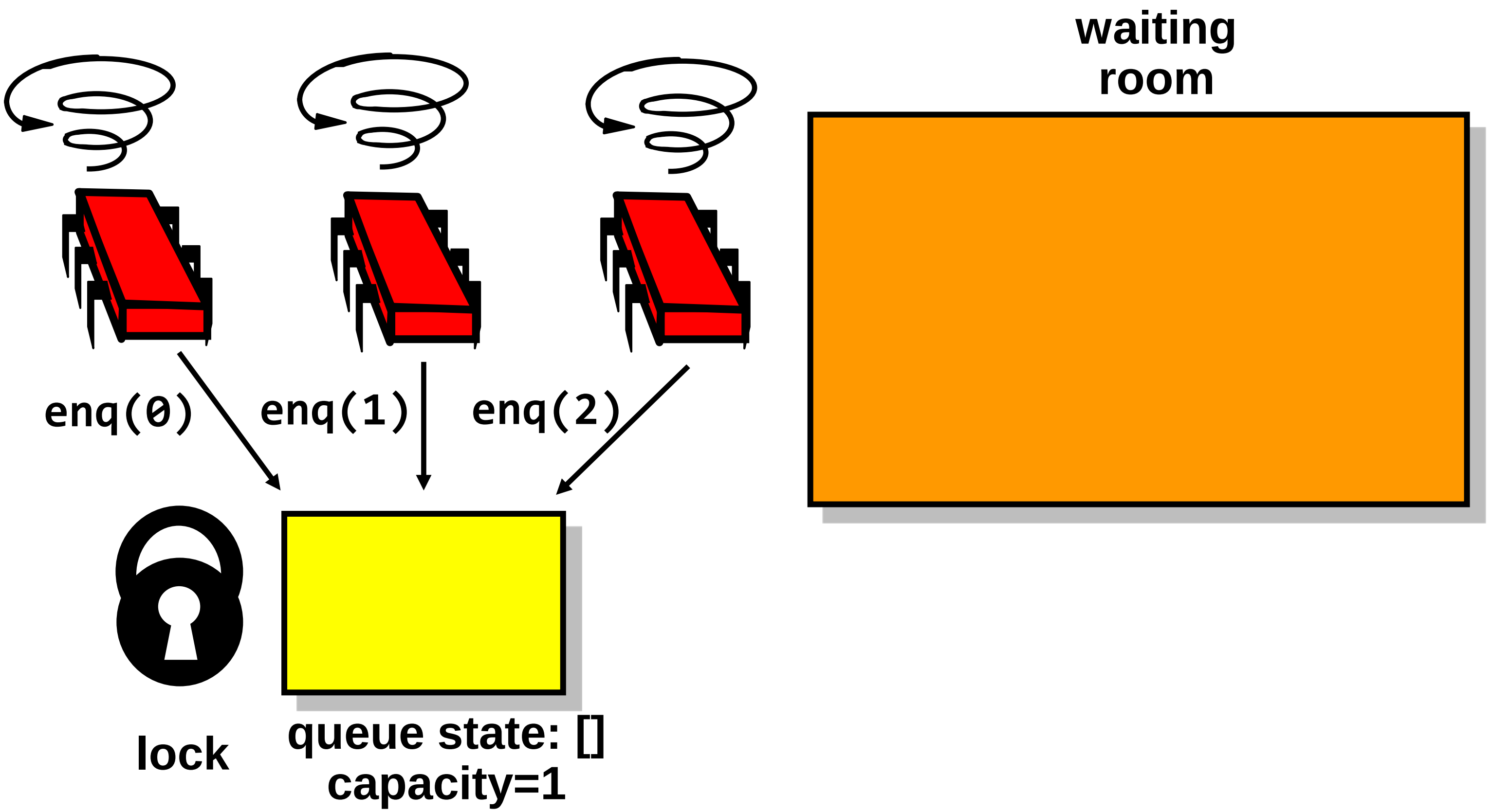
Simplified Blocking Queue: dequeue

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

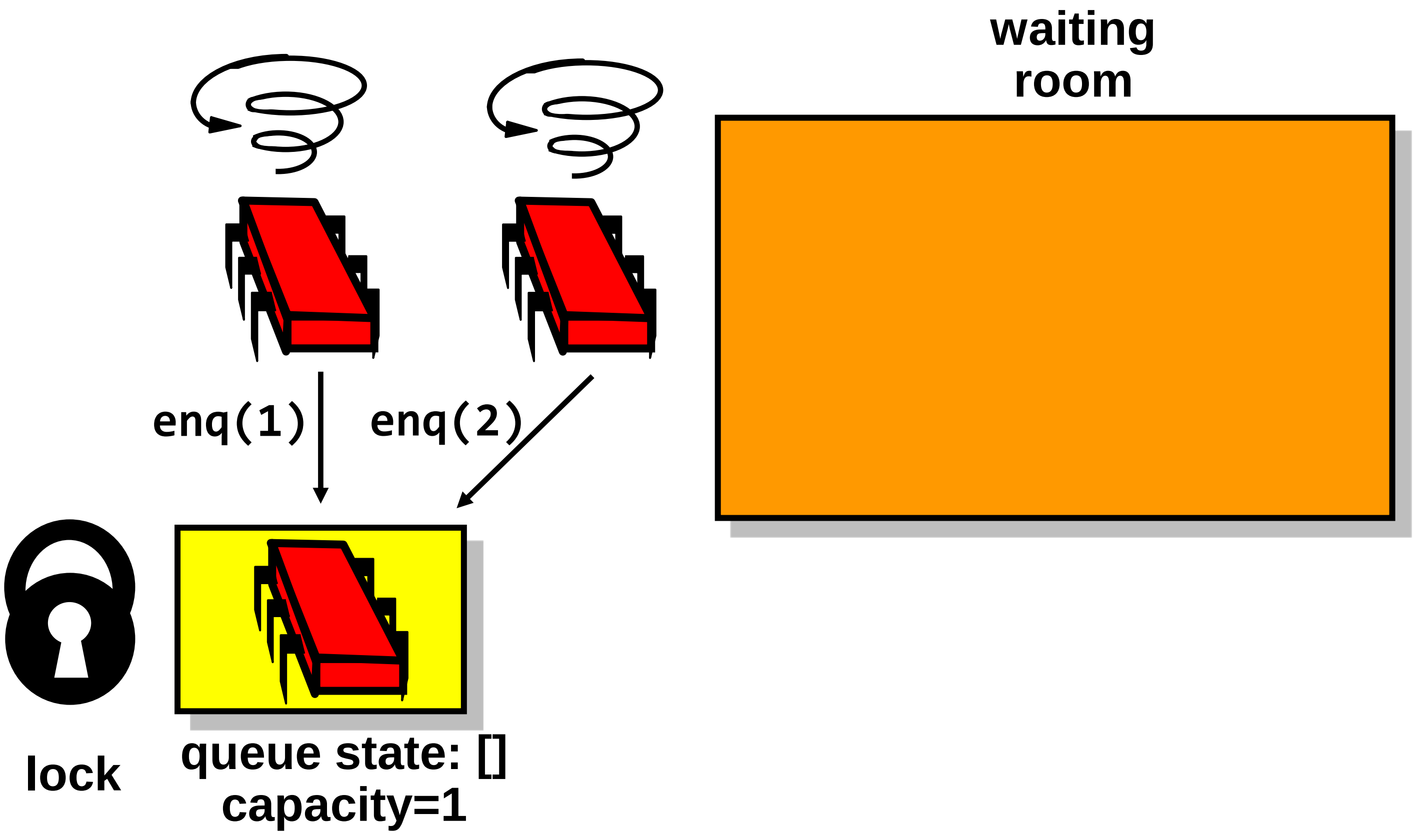
is **`notify`** enough?

lost wakeups

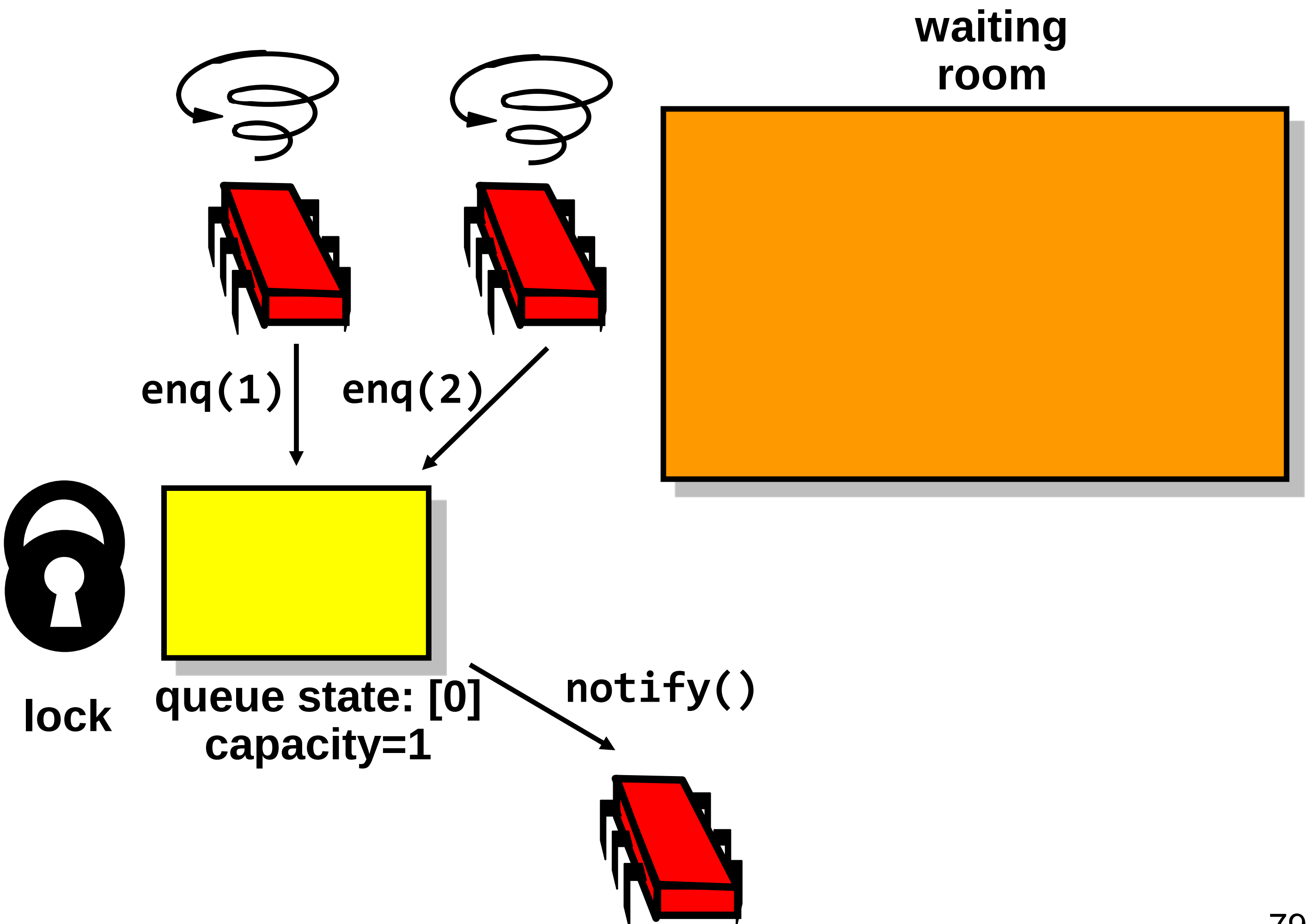
Lost Wakeup in Simplified Queue with `notify()`



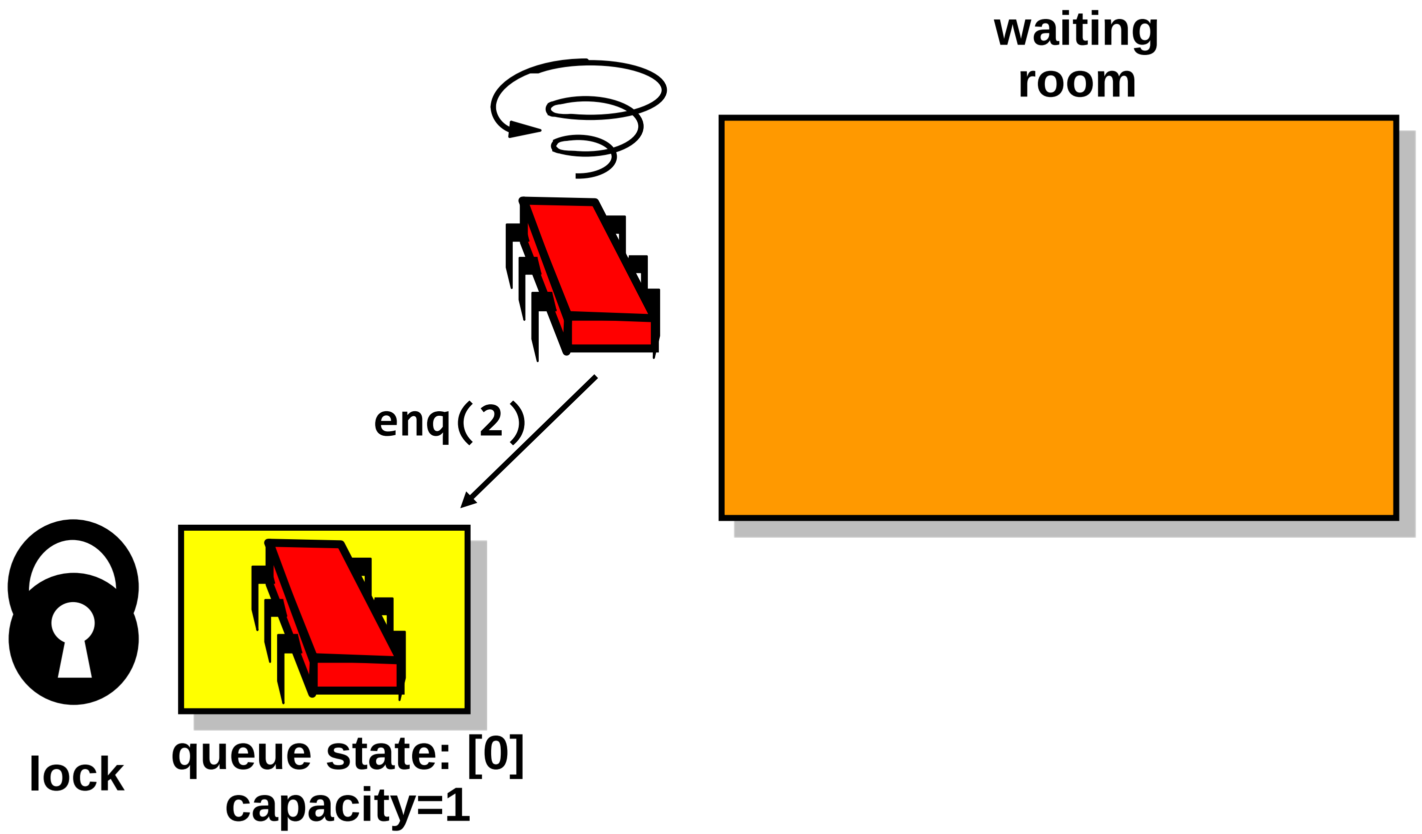
Lost Wakeup in Simplified Queue with `notify()`



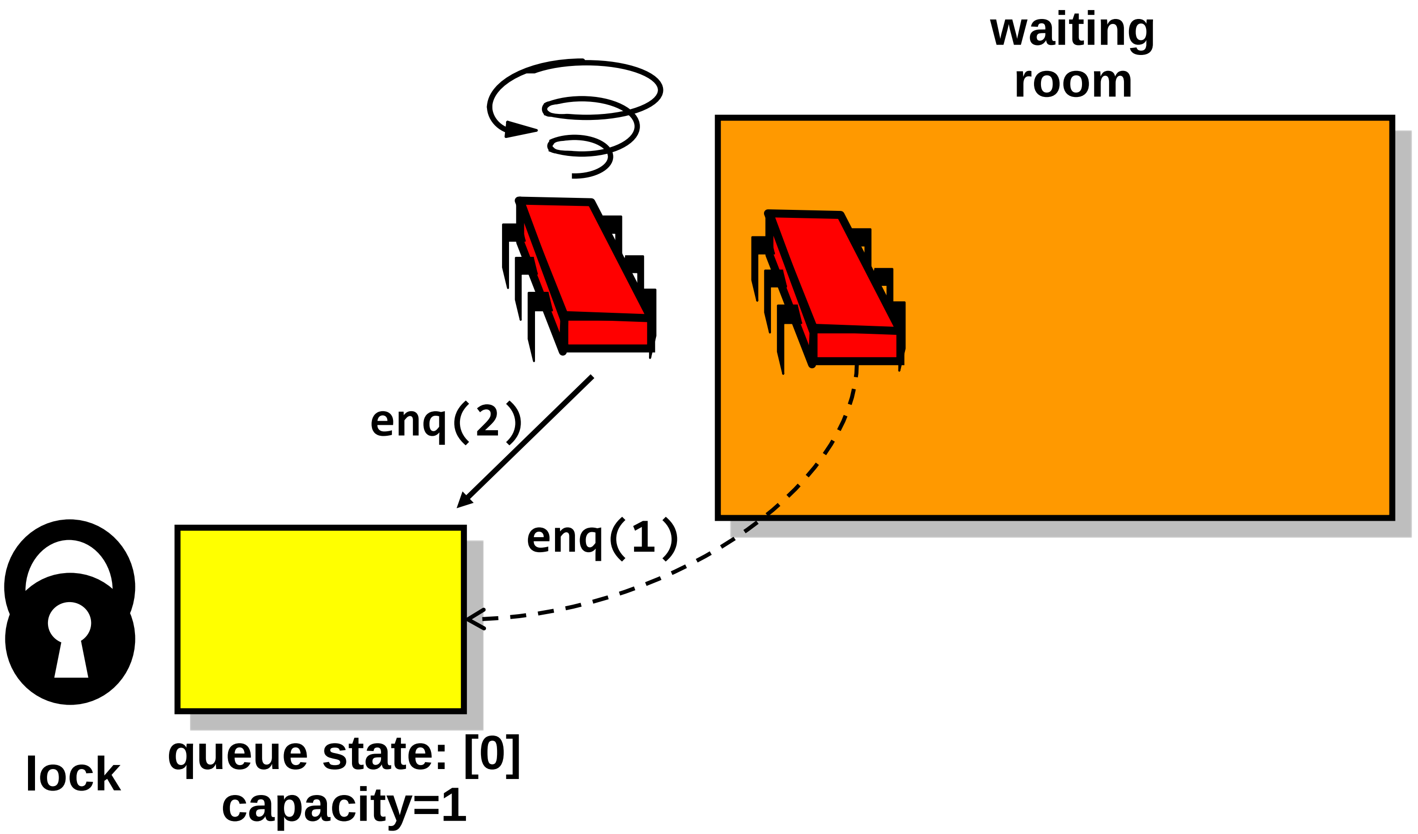
Lost Wakeup in Simplified Queue with `notify()`



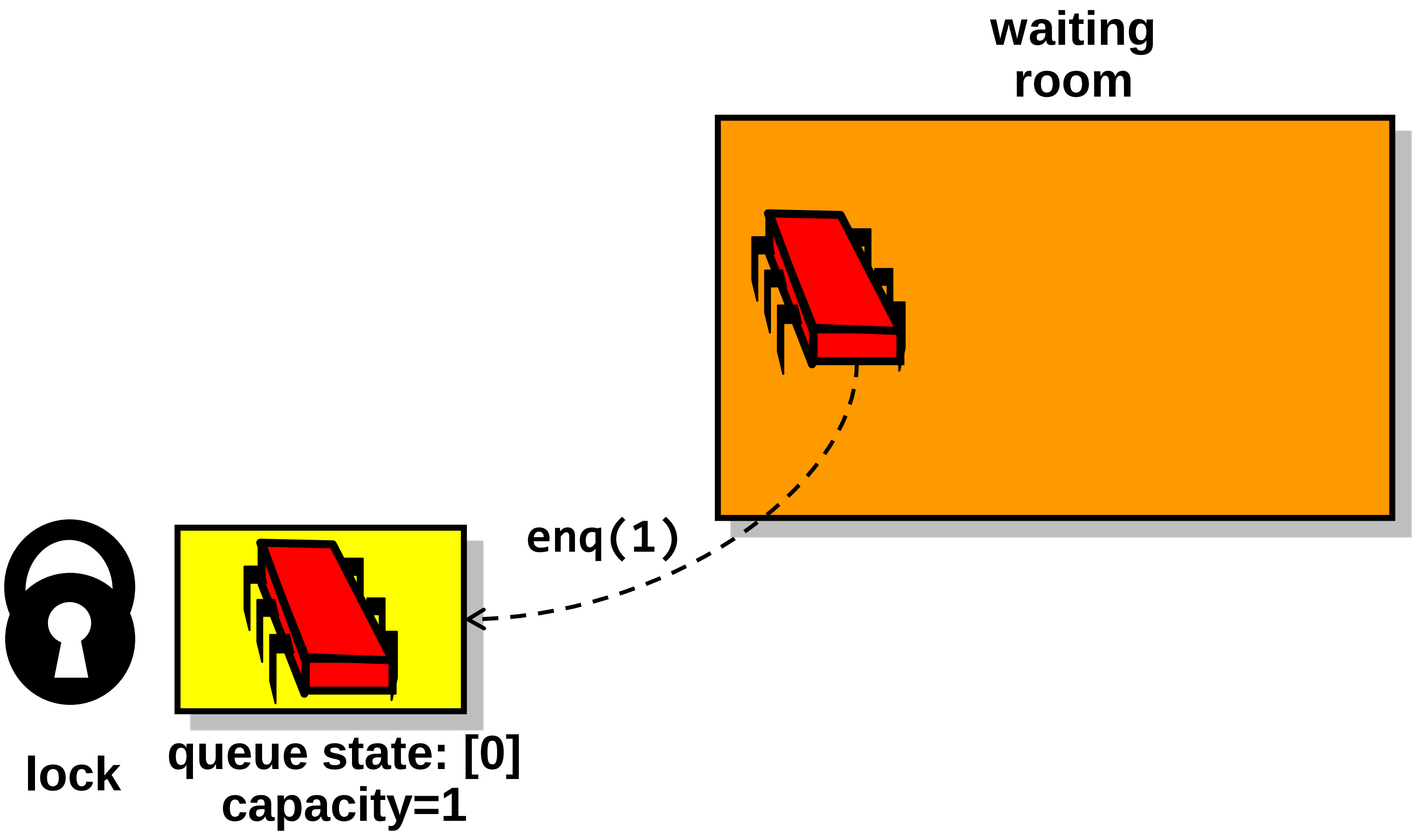
Lost Wakeup in Simplified Queue with `notify()`



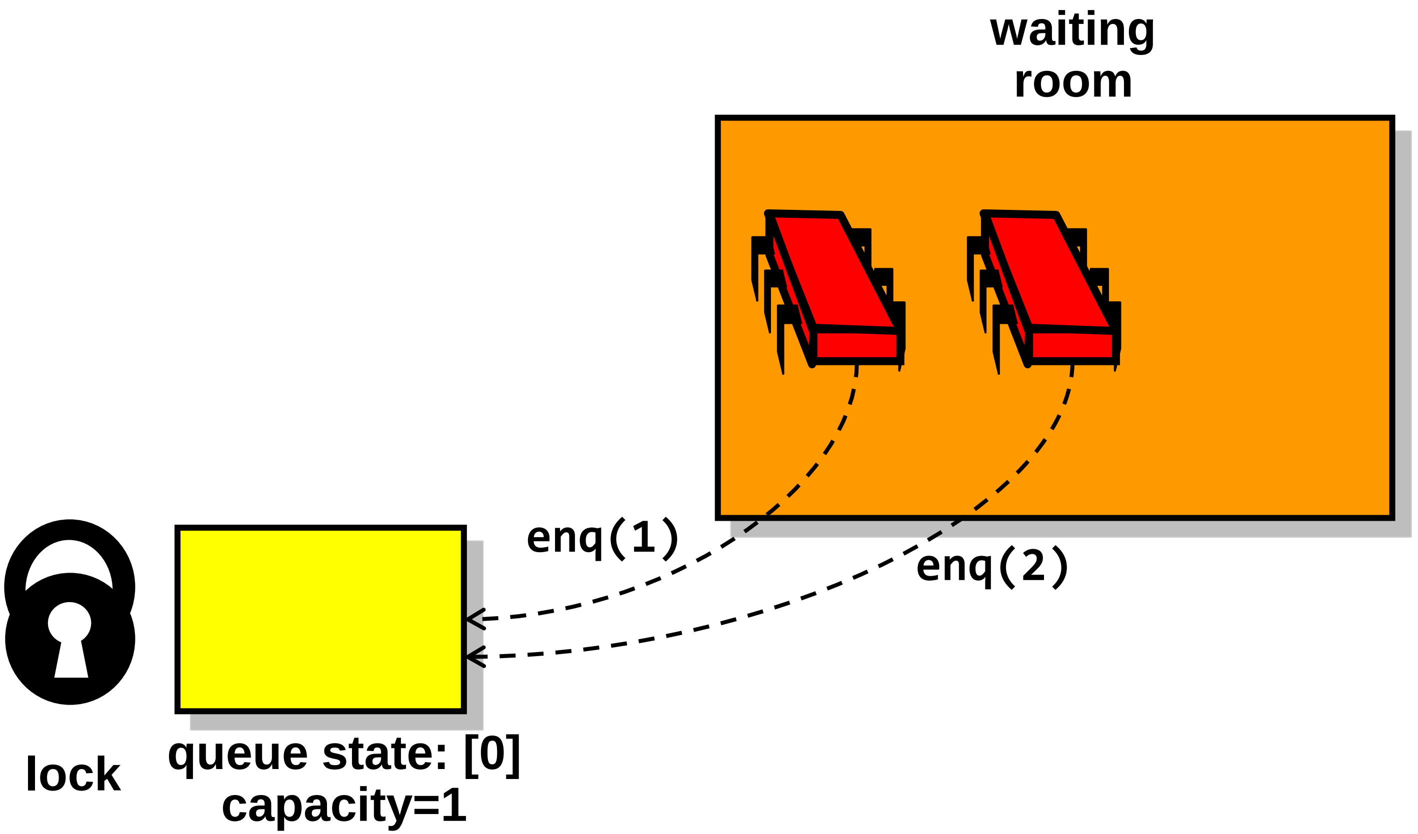
Lost Wakeup in Simplified Queue with `notify()`



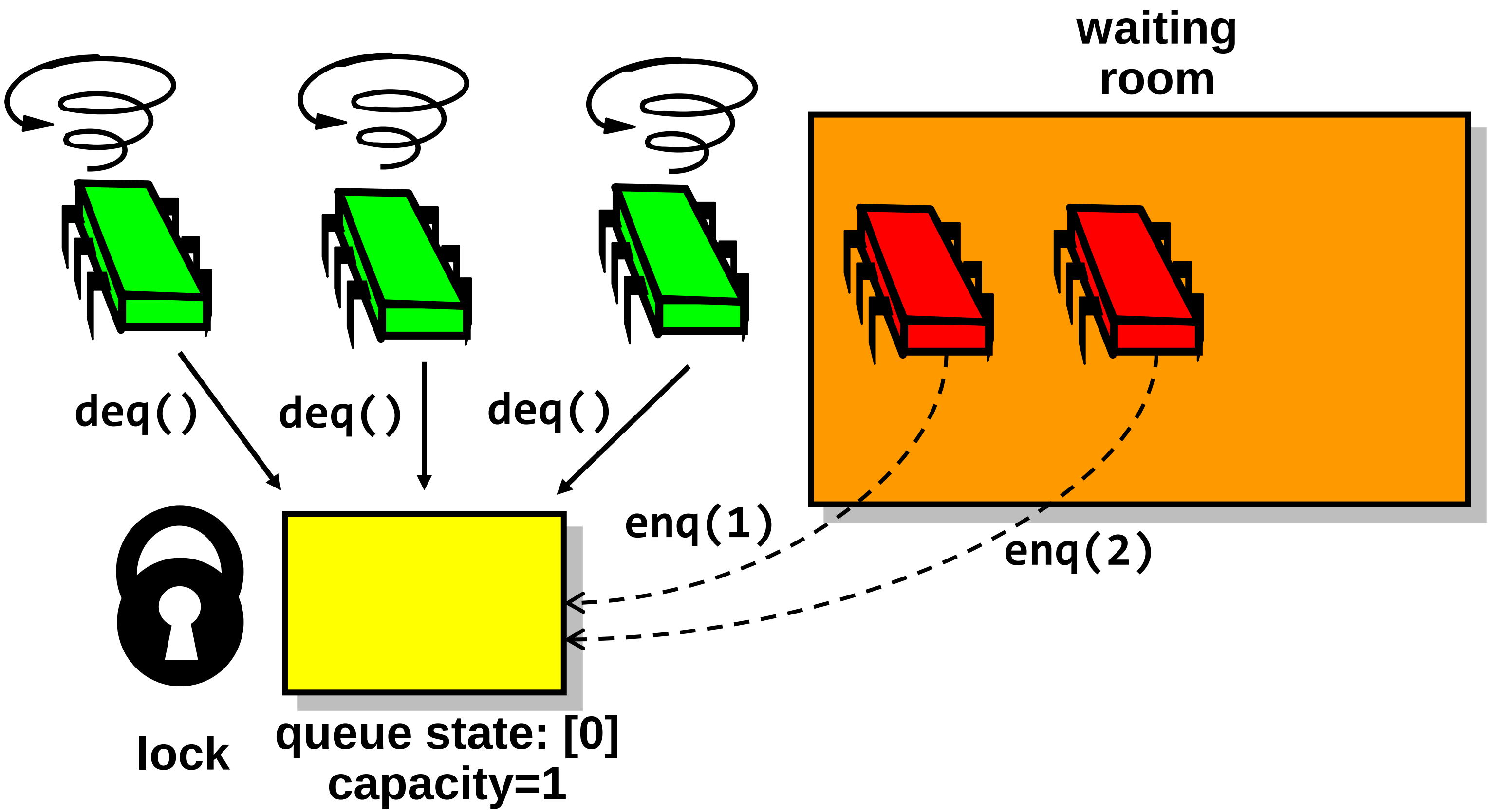
Lost Wakeup in Simplified Queue with `notify()`



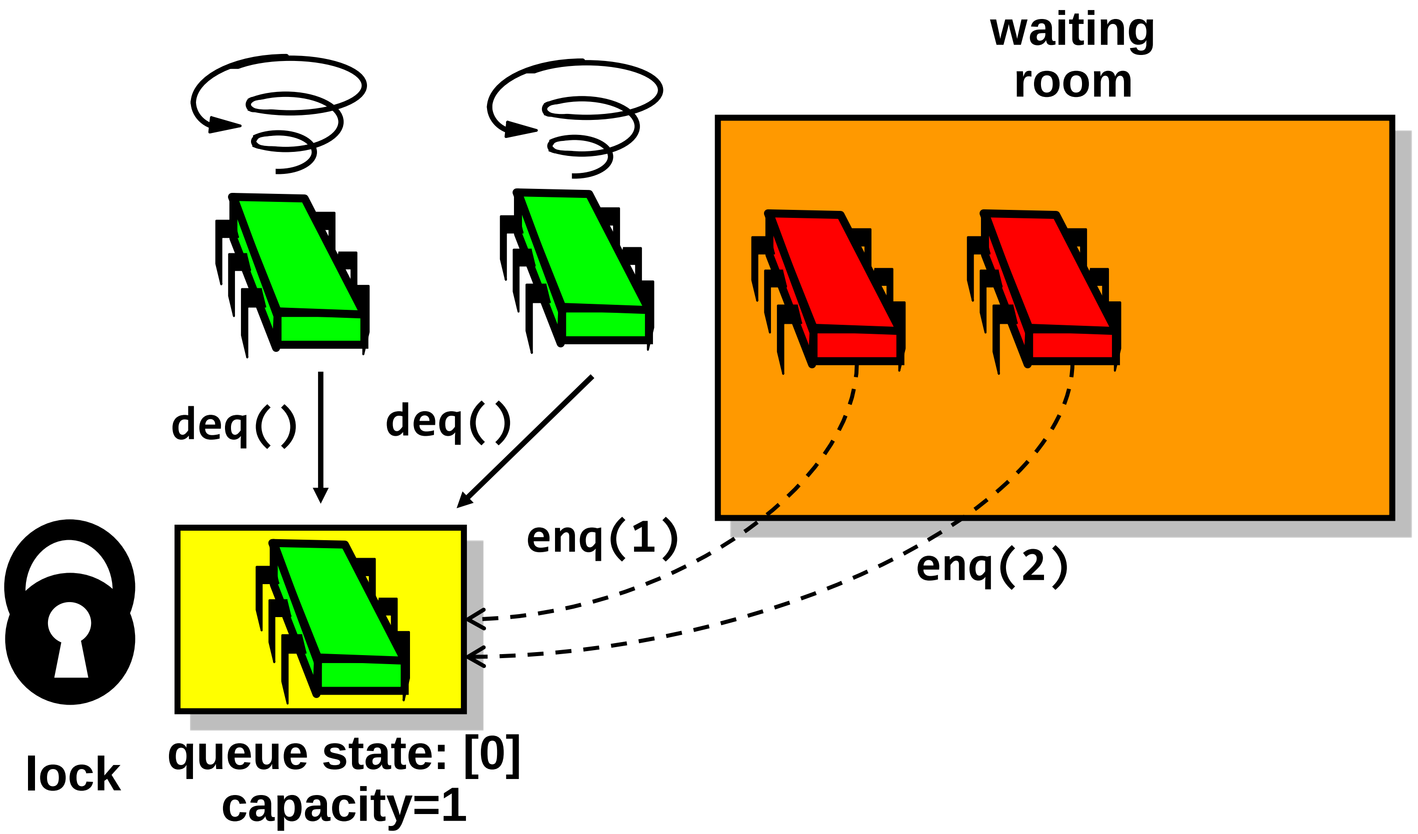
Lost Wakeup in Simplified Queue with `notify()`



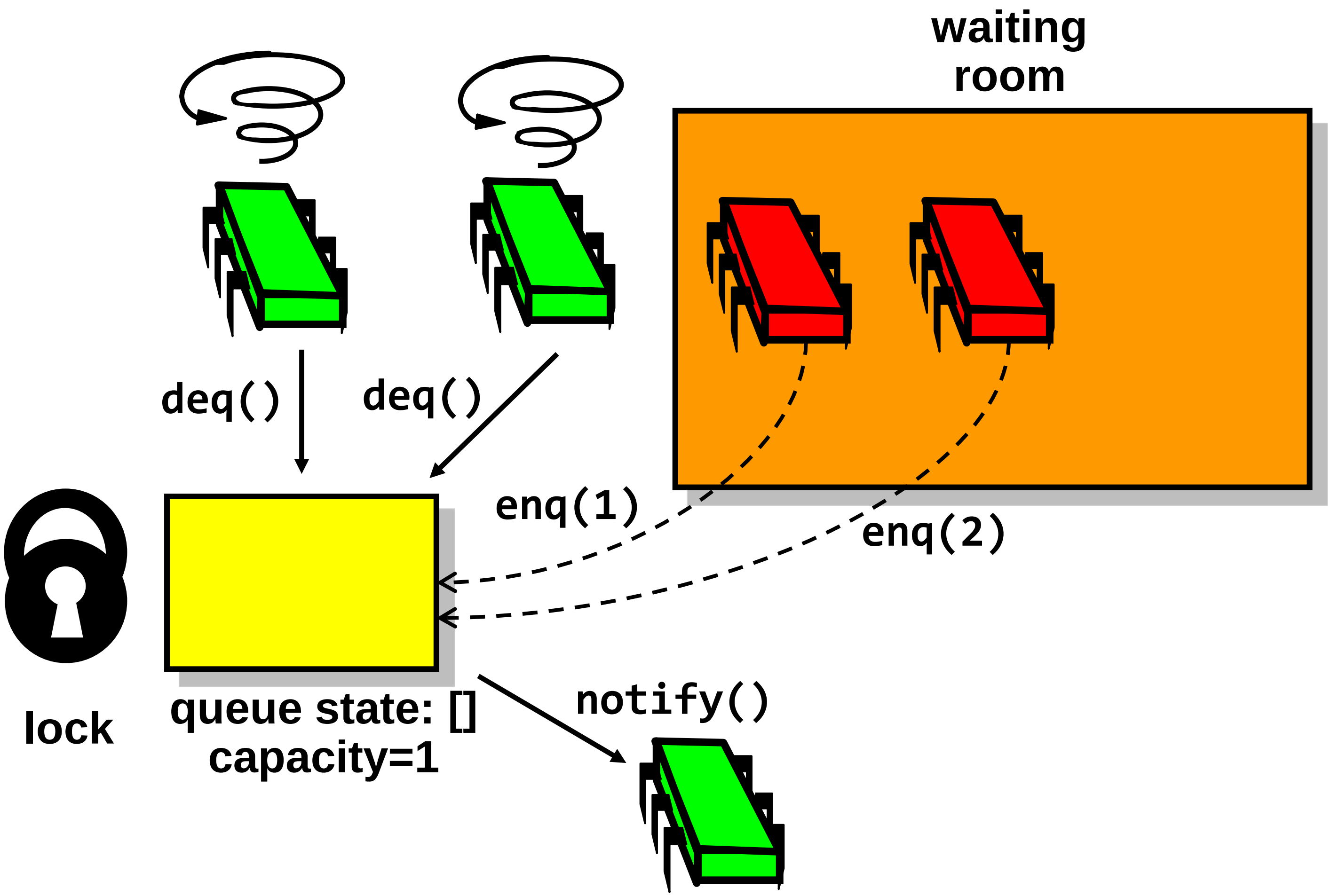
Lost Wakeup in Simplified Queue with `notify()`



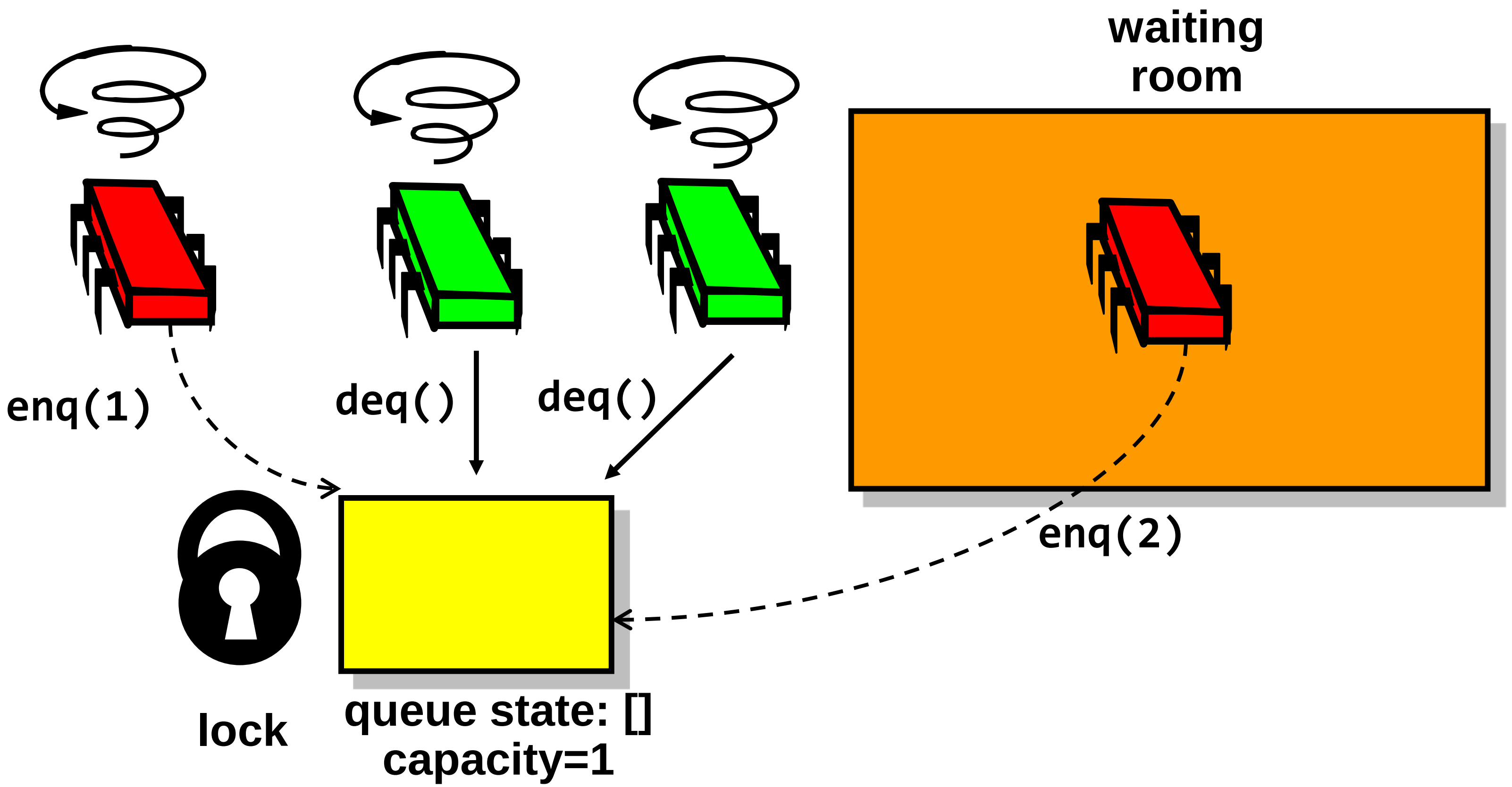
Lost Wakeup in Simplified Queue with `notify()`



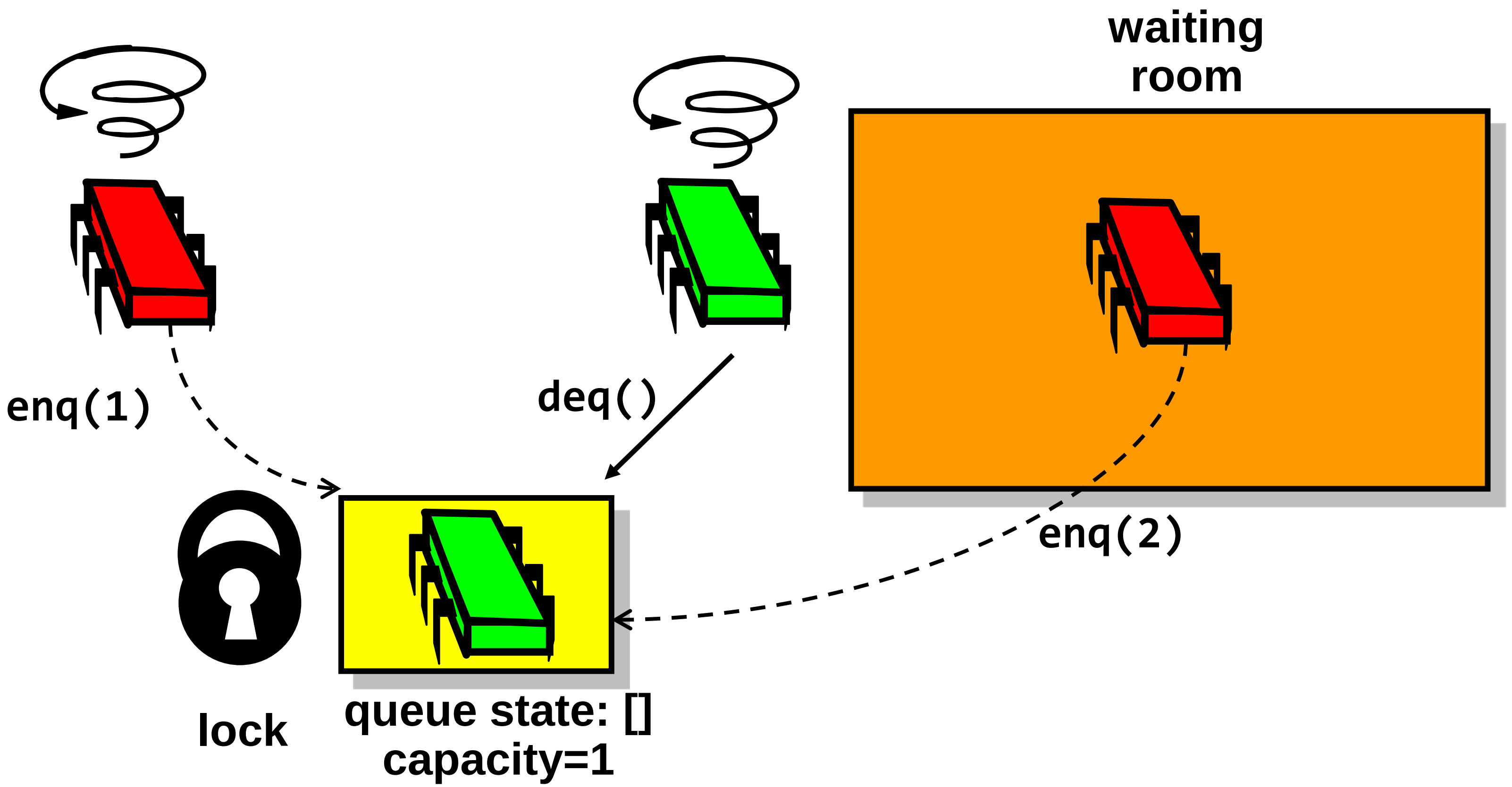
Lost Wakeup in Simplified Queue with `notify()`



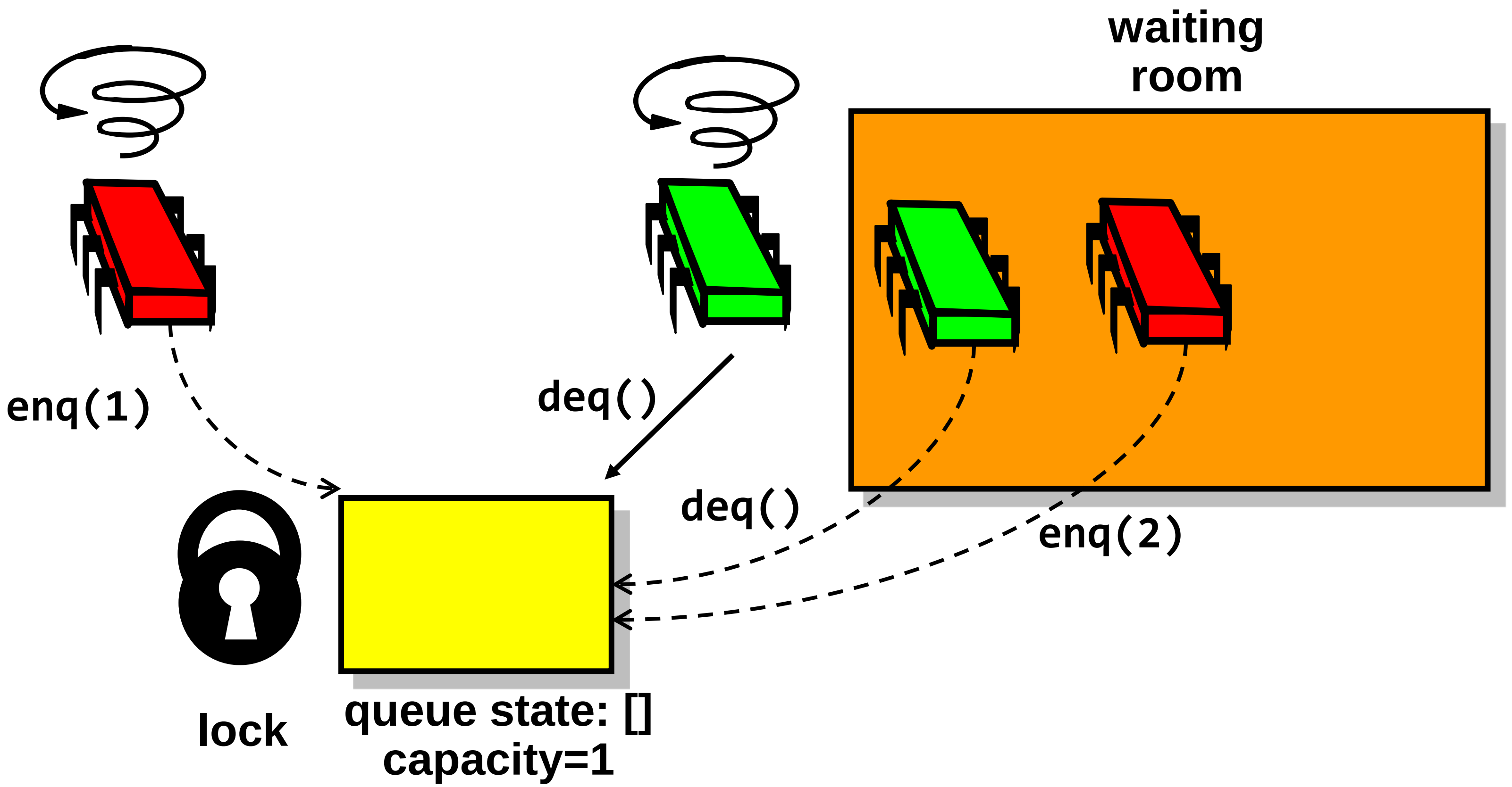
Lost Wakeup in Simplified Queue with `notify()`



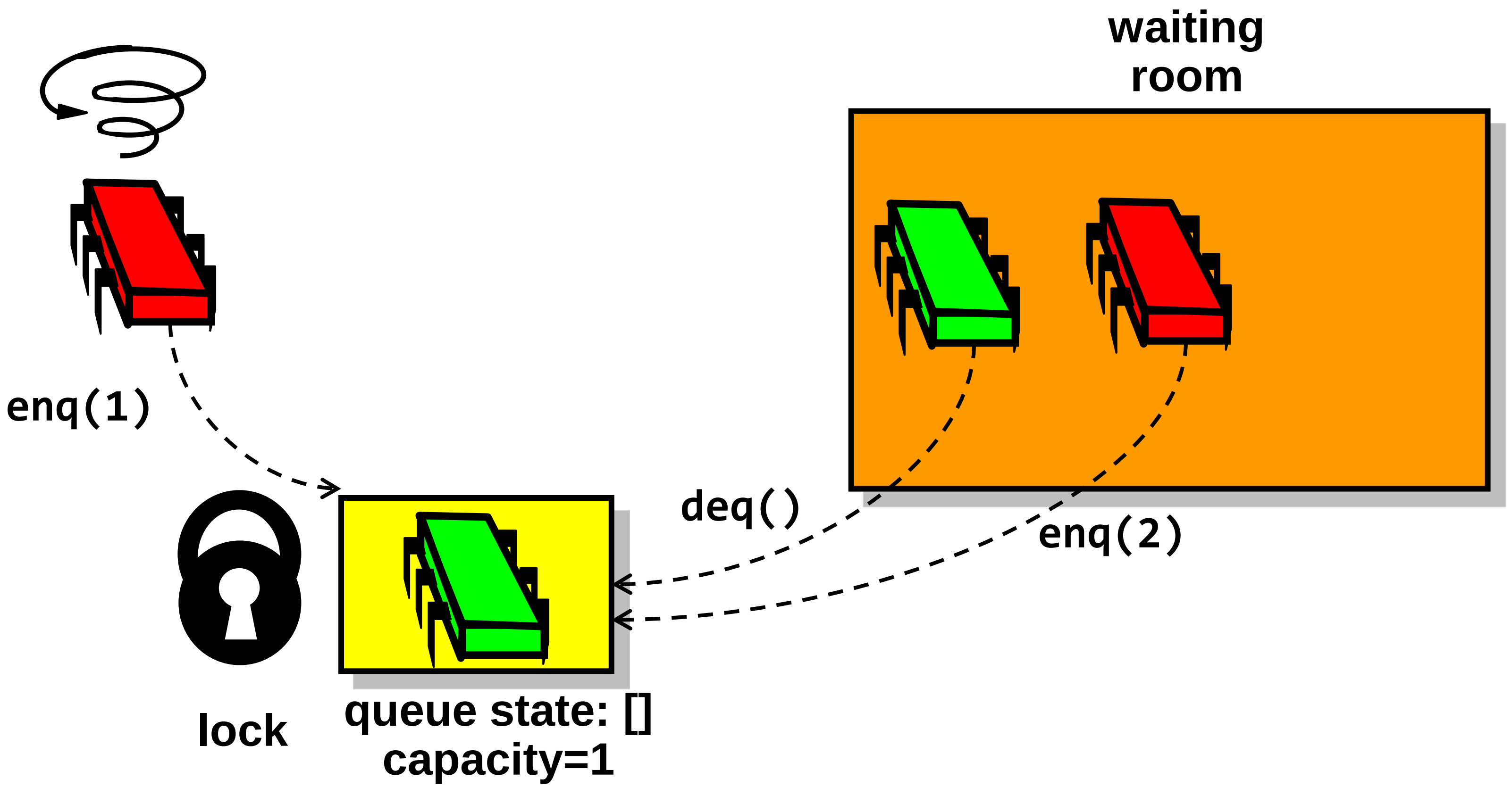
Lost Wakeup in Simplified Queue with `notify()`



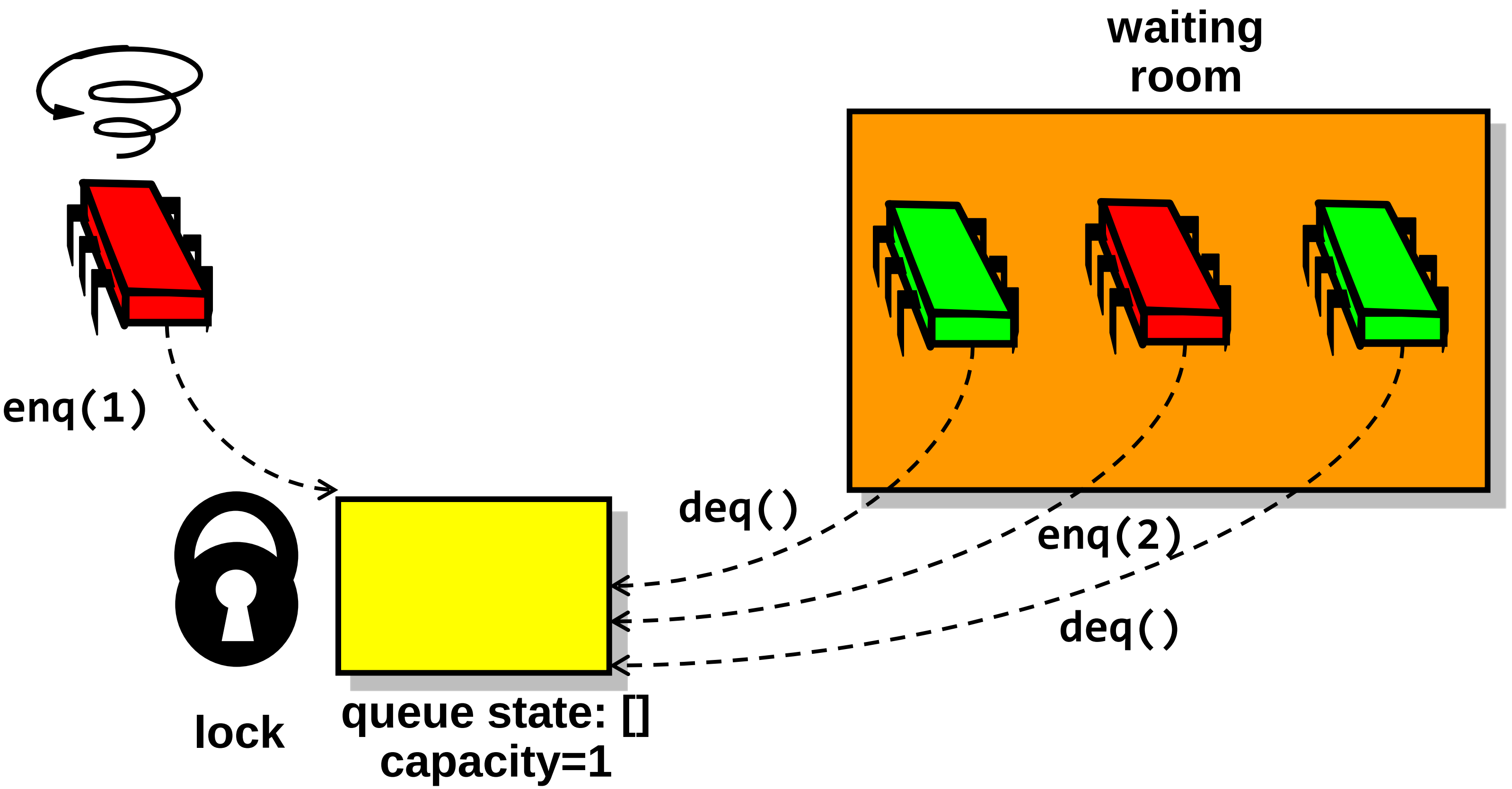
Lost Wakeup in Simplified Queue with `notify()`



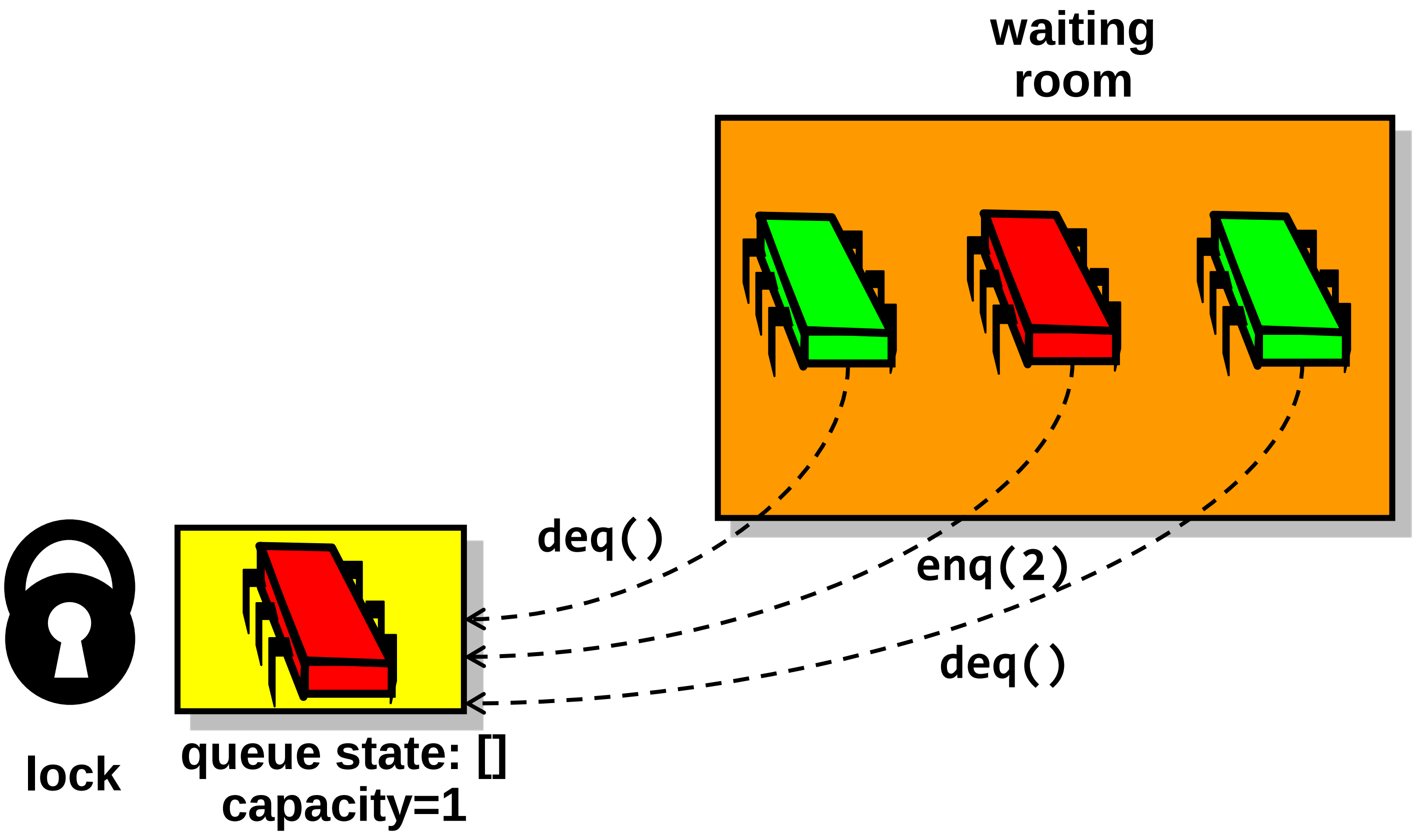
Lost Wakeup in Simplified Queue with `notify()`



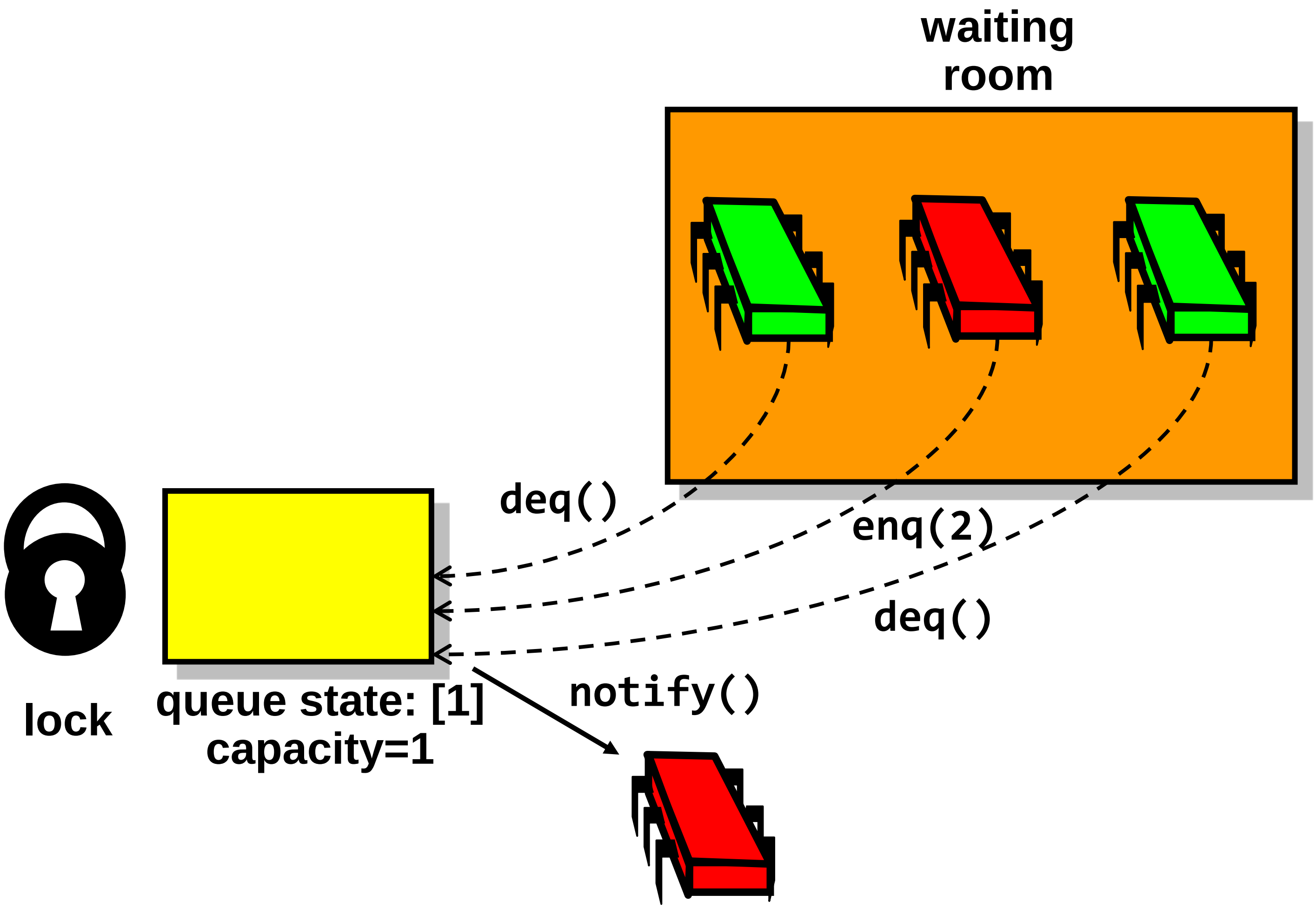
Lost Wakeup in Simplified Queue with `notify()`



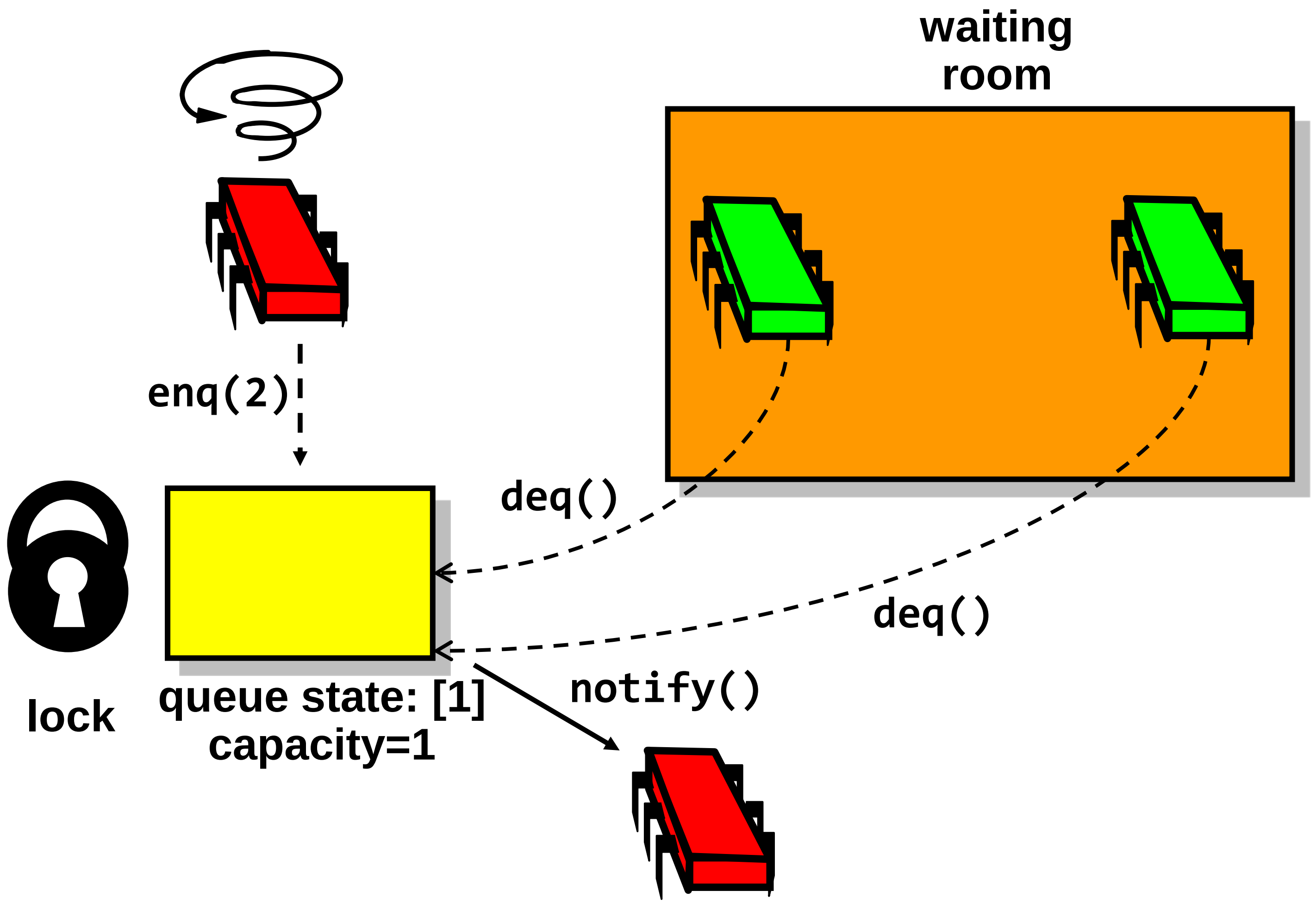
Lost Wakeup in Simplified Queue with `notify()`



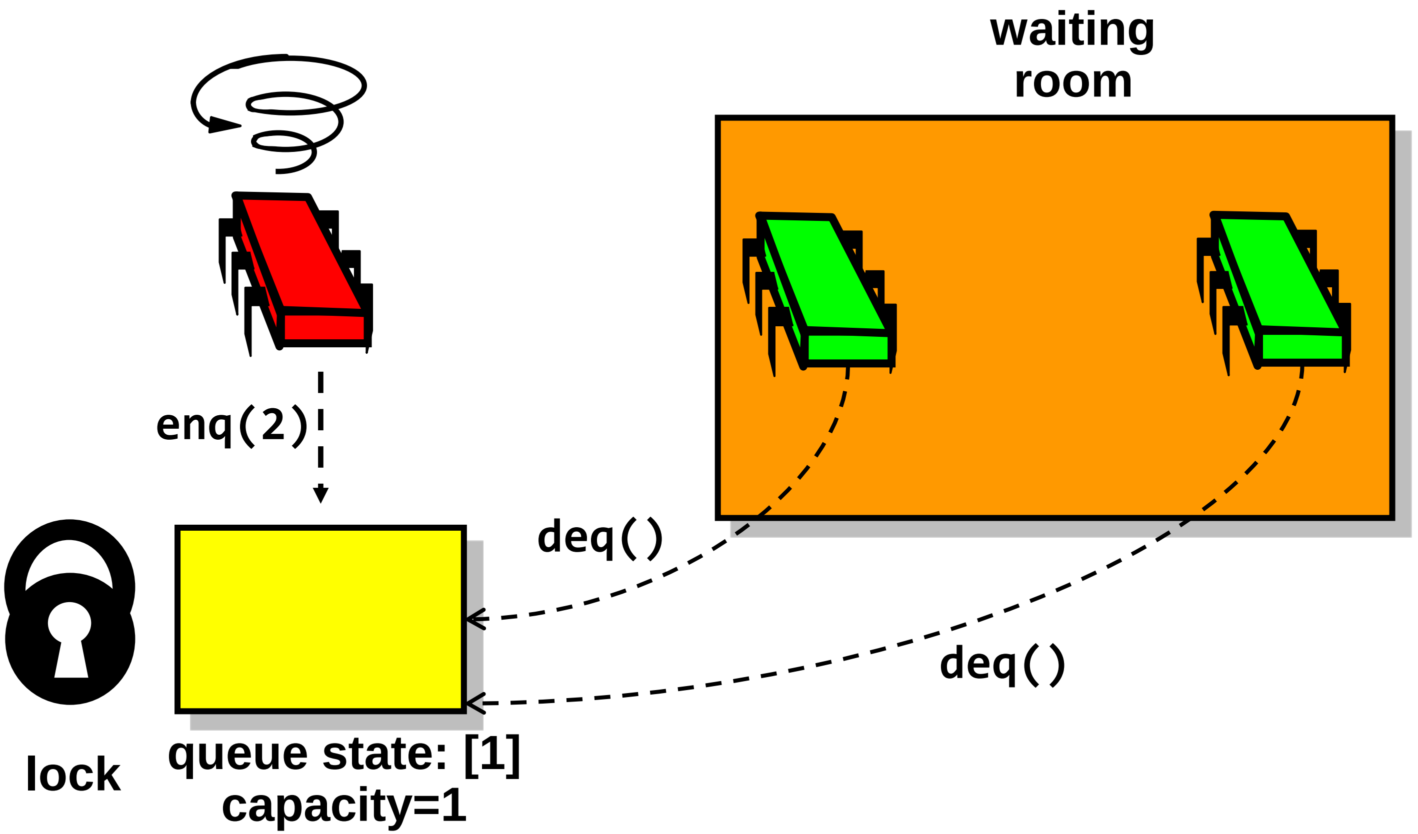
Lost Wakeup in Simplified Queue with `notify()`



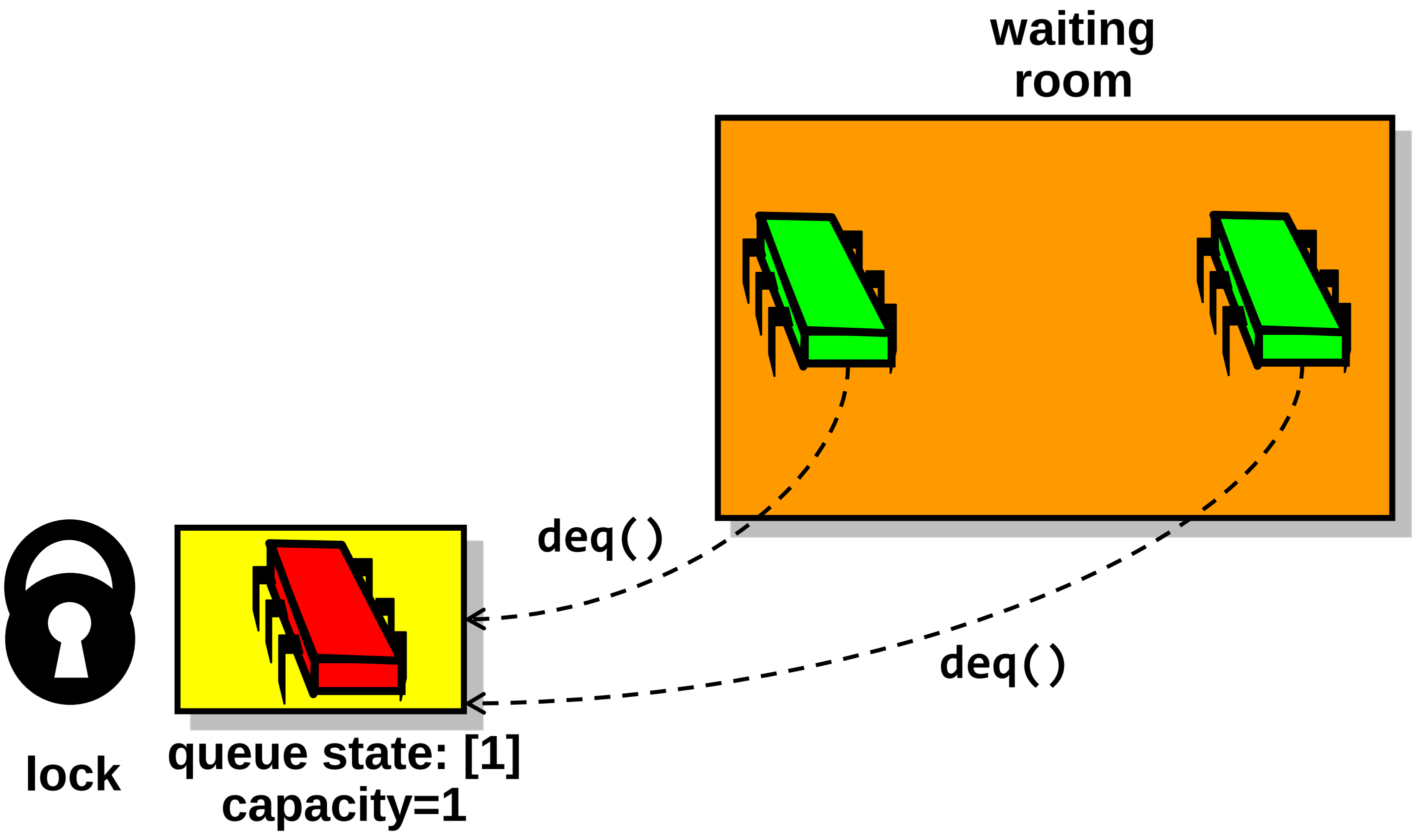
Lost Wakeup in Simplified Queue with notify()



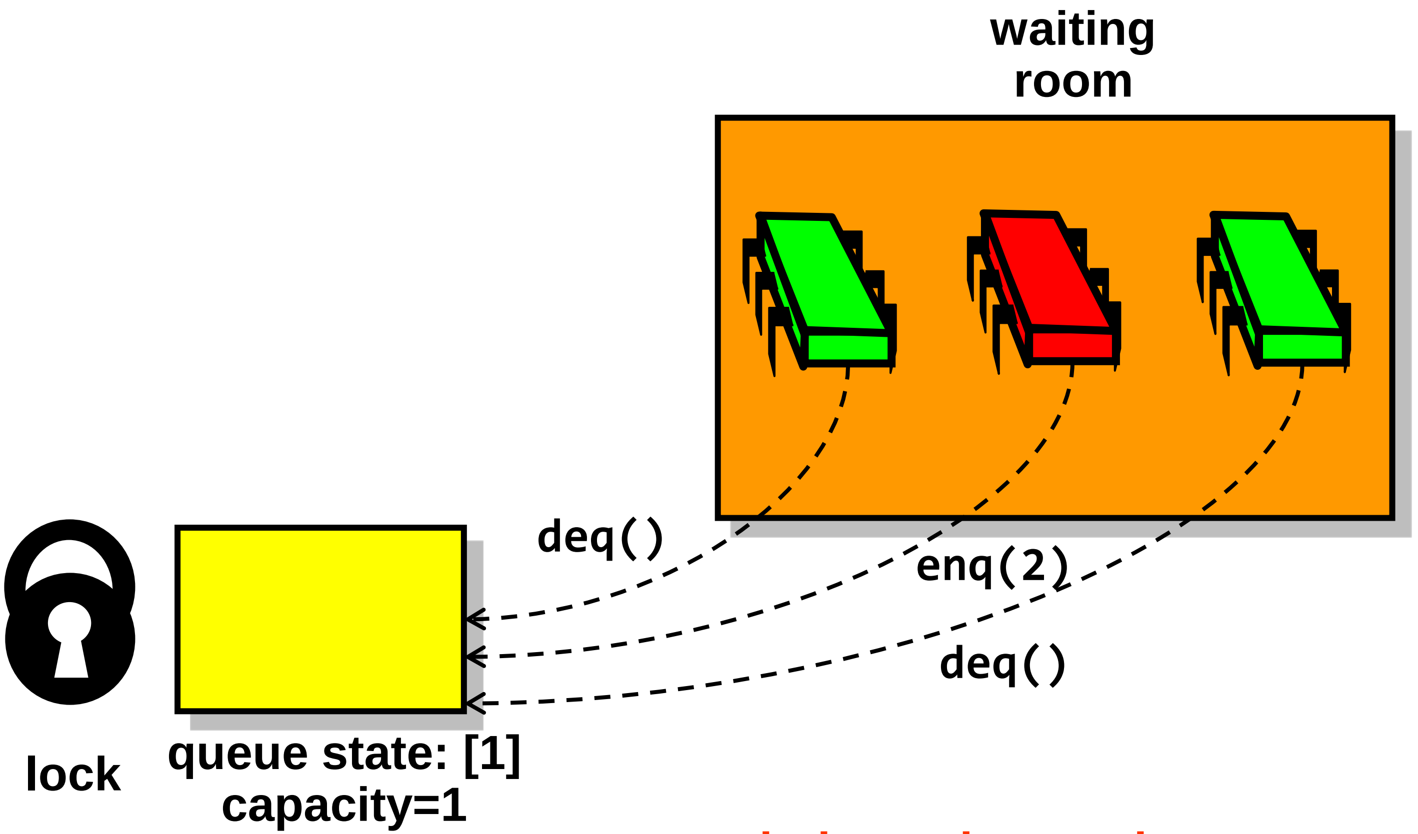
Lost Wakeup in Simplified Queue with `notify()`



Lost Wakeup in Simplified Queue with `notify()`



Lost Wakeup in Simplified Queue with `notify()`



remaining threads are stuck!

Issues with Shared Memory Concurrency

- Complicated locking protocols
  - deadlocks
  - livelocks
  - lost wake-up
  - …
- Weak memory consistency guarantees: need to use
  - `volatile` variables
  - `AtomicReferenceArray<T>` instead of `T[]`
  - …

`java.lang.concurrent` helps but is not a panacea.