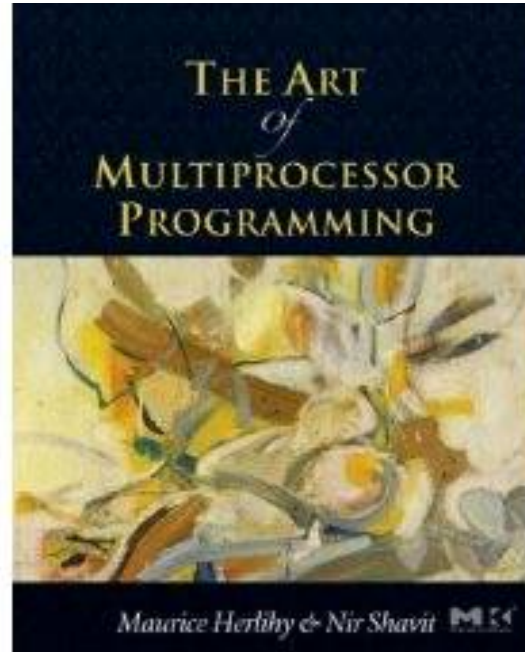


# COS 226

## Chapter 8

## Monitors and Blocking Synchronization

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Monitors

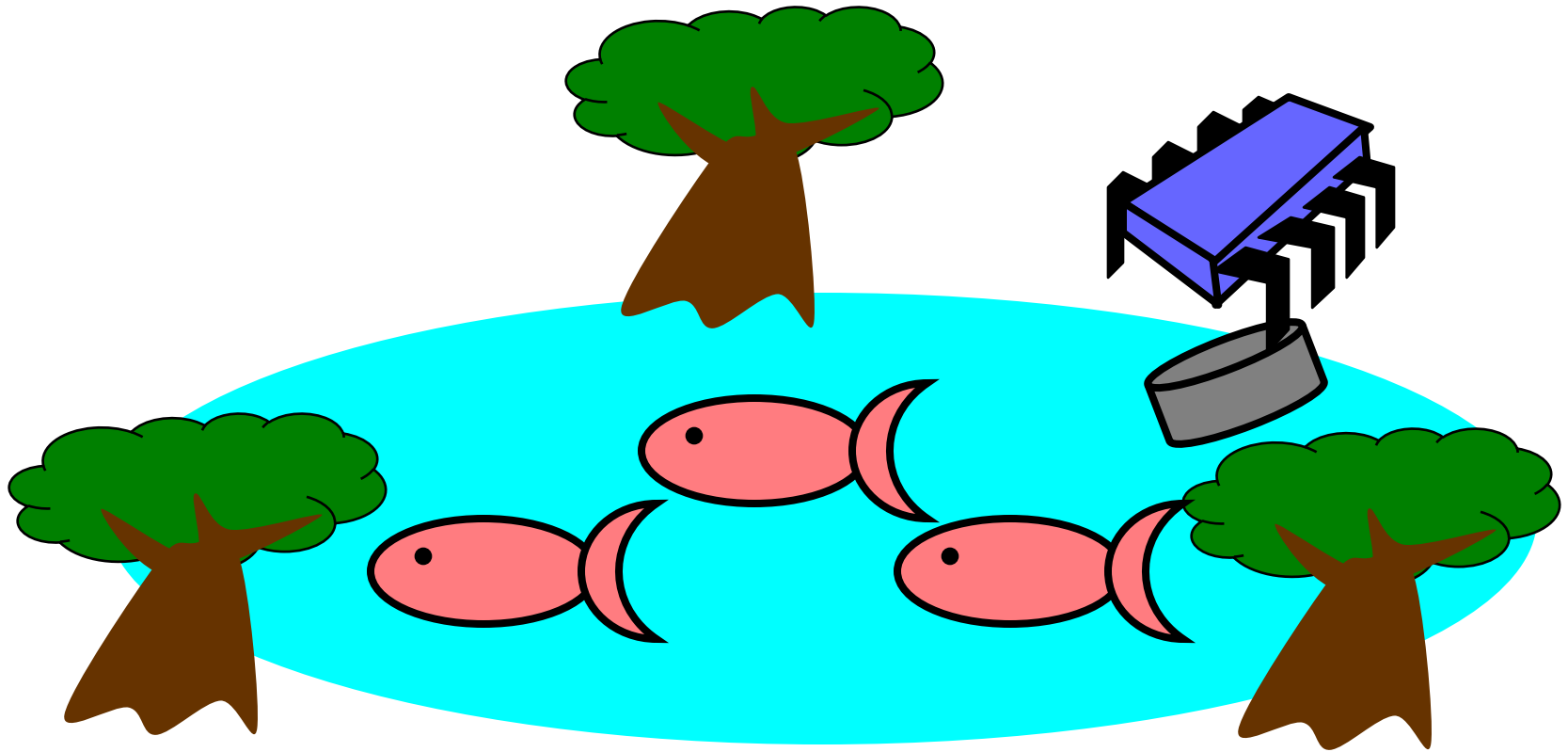
- Structured way of combining synchronization and data
- Combines data, methods and synchronization in a single modular package



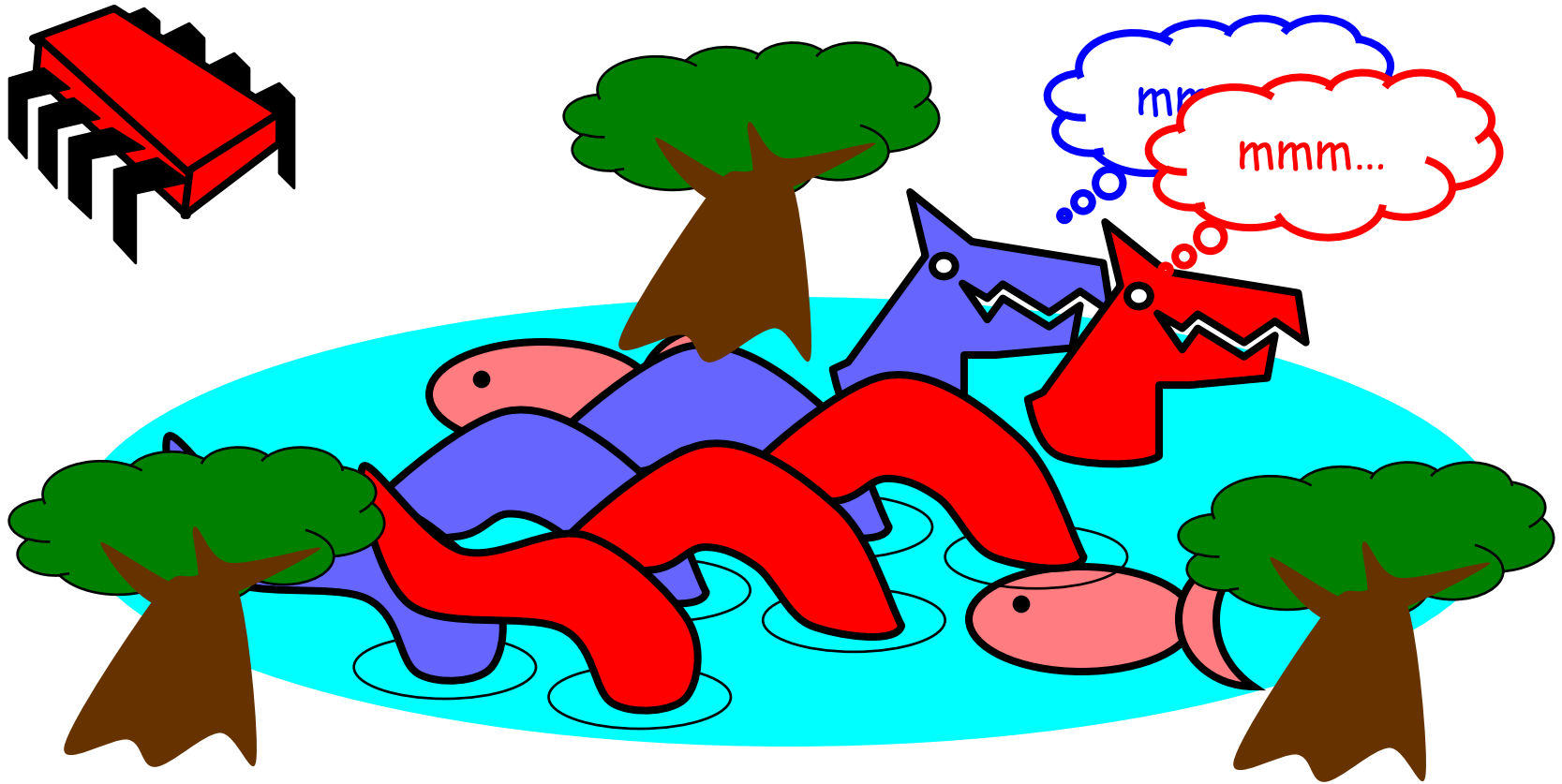
# Producer-Consumer Problem

- Synchronization problem

# Bob Puts Food in the Pond



# Alice releases her pets to Feed





# Producer/Consumer

- Alice and Bob can't meet
  - Each has restraining order on other
  - So he puts food in the pond
  - And later, she releases the pets
- Avoid
  - Releasing pets when there's no food
  - Putting out food if uneaten food remains



# In real life...

- Imagine application with two threads – producer and consumer
- Two threads communicate through a shared FIFO queue
- Principles:
  - The producer generates data, puts it into the queue and start again
  - At the same time the consumer, consumes the data one piece at a time

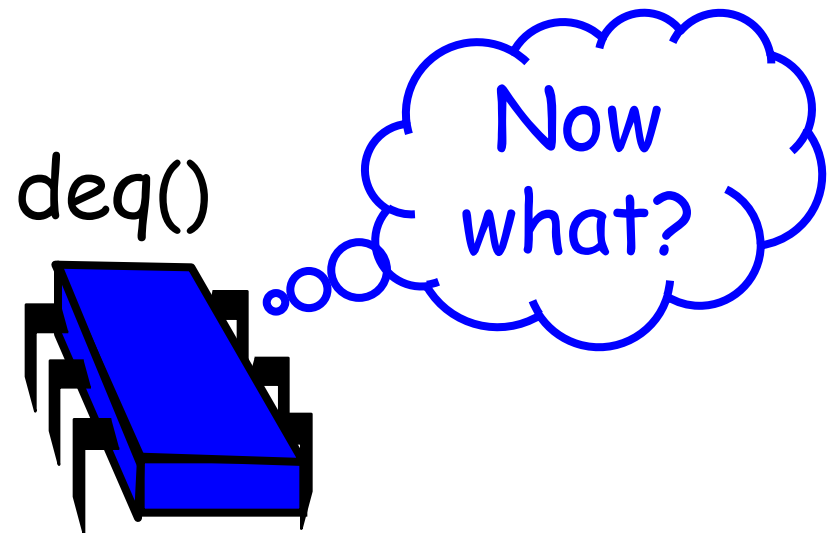
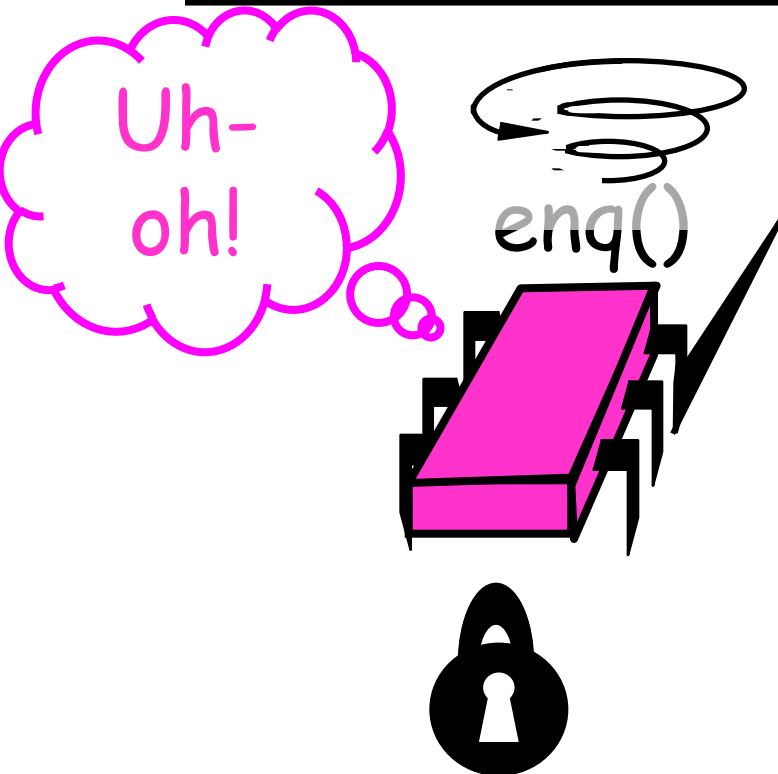
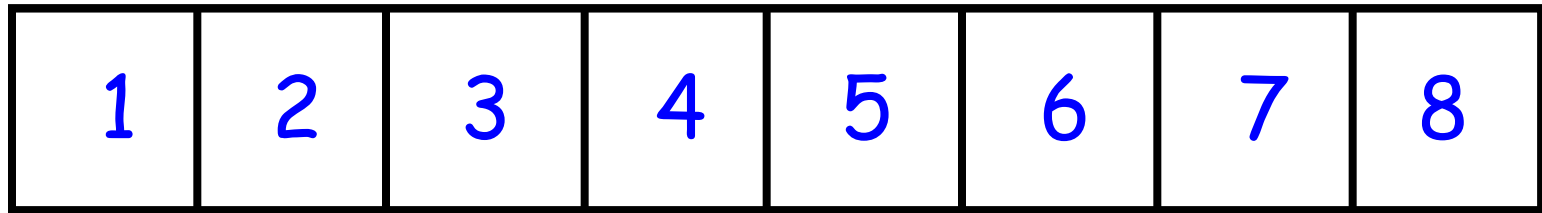




# Producer

```
mutex.lock()
try {
    queue.enq();
} finally {
    mutex.unlock();
}
```

# Producer-Consumer





# Problems

- What happens when the queue is full?
- Problem:
  - ☐ Producer should not try to add data if the buffer is full
  - ☐ Consumer should not try to remove data from an empty buffer
- Where should this be managed?



# Sensible approach

- Allow each queue to manage its own synchronization
- The queue itself has its own internal lock, acquired by methods and released when it returns
  - For example, lock() can be called inside the enq() method
- If a thread then tries to enq() an item onto a full queue, the enq() method itself can detect the problem



# Monitor Locks

- Only one thread at a time can *hold* a lock
- A thread *acquires* a lock when it starts to hold the lock
- A thread *releases* the lock when it stops holding the lock
- A monitor has methods each which acquires the lock when it is called and releases the lock when it returns



# Monitor Locks

- When a thread cannot immediately acquire a lock it either:
  - Spins – repeatedly testing whether it is available
  - Blocks – suspends thread and creates new thread
- Spinning = short time
- Blocking = long time



# Spinning vs. Blocking

- For example:
  - A thread waiting for another thread to release the lock should spin if lock is held briefly
  - A consumer thread waiting to dequeue an item from a empty queue should block
- Often spinning and blocking are combined
- Spinning does not work on uniprocessors



# Producer-Consumer

- Need a way for a thread that has acquired a lock to release the lock for a while and then to reacquire it and try again





# Conditions

- In java concurrency package
- Condition object provides ability to release a lock temporarily
- A Condition is related to a lock and is created by lock's `newCondition()` method



# Conditions

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

# Conditions

```
Condition condition = mutex.newCondition();
```

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

Create Condition object that is related to lock

# Conditions

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

Acquires lock

# Conditions

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

Tests whether property holds

# Conditions

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

Releases the lock and suspends itself

# Conditions

```
Condition condition = mutex.newCondition();  
...  
mutex.lock();  
try {  
    while (!property) {  
        condition.await();  
    } catch (InterruptedException e) {...}  
...  
} Suspension can be interrupted by other thread
```



# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        }  
    }  
}
```



# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        }  
    }  
}
```

Condition to check  
whether queue is  
full

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

```
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        }  
    }  
}
```

Array of elements  
with some max  
amount

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...
```

```
    public void enq(T x) {
```

Attempt to add item  
to queue

```
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...
```

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void eng(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        }  
    }  
}
```

Acquires lock

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        }  
    }  
}
```

Check whether  
queue is full

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

...

```
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();
```

...

Sleep while queue  
is full



# Conditions

- How does the Producer know when the queue is not full anymore?
- Options:
  - Can awaken on time constraints
  - Another thread (Consumer) will have to wake it.



# Conditions

```
public interface Condition {  
    void await()  
    boolean await (long time)  
    boolean awaitUntil (Date)  
    long awaitNanos (long nanoSec)
```

- All of these methods uses time constraints





# Conditions

- Another thread can also use `signal()` to notify threads that it has changed a certain property.

```
void signal()  
void signalAll()
```



# Conditions

- However, there is no guarantee that when the thread awakens after a specified time, the property will hold
- Thus the thread must retest the property when it awakes



# Conditions

- When threads are woken up, they could still have to compete for the lock



# Producer-Consumer queue

- If the Producer thread has been suspended because of a full queue, when should it be woken up again?
- And what about checking whether the queue is empty or not?

```
class LockedQueue {
    Lock lock = new ReentrantLock();
    Condition isFull = lock.newCondition();
    Condition isEmpty = lock.newCondition();
    T[] items;
    ...
    public void enq(T x) {
        lock.lock();
        try {
            while (items.length == max)
                isFull.await();
            items[tail] = x;
            ...
            isEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            items[tail] = x;  
            ...  
            isEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Second condition to  
test whether queue  
is empty

```
class LockedQueue {
    Lock lock = new ReentrantLock();
    Condition isFull = lock.newCondition();
    Condition isEmpty = lock.newCondition();
    T[] items;
    ...
    public void enq(T x) {
        lock.lock();
        try {
            while (items.length == max)
                isFull.await();
            items[tail] = x;
            ...
            isEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

When thread wakes  
up it can access the  
CS – add an item

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

Since an item has  
been added it is not  
empty – alert  
Consumer

```
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            items[tail] = x;  
            ...  
            isEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```



```
class LockedQueue {
    Lock lock = new ReentrantLock();
    Condition isFull = lock.newCondition();
    Condition isEmpty = lock.newCondition();
    T[] items;
    ...
    public T deq() {
        lock.lock();
        try {
            while (items.length == 0)
                isEmpty.await();
            T x = items[head]
            ...
            isFull.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

```
class LockedQueue {
    Lock lock = new ReentrantLock();
    Condition isFull = lock.newCondition();
    Condition isEmpty = lock.newCondition();
    T[] items;
    ...
    public T deq() {
        lock.lock();
        try {
            while (items.length == 0)
                isEmpty.await();
            T x = items[head]
            ...
            isFull.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

First acquire lock

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

```
    ...
```

```
    public T deq() {  
        lock.lock();  
        try {
```

```
            while (items.length == 0)
```

```
                isEmpty.await();
```

```
            T x = items[head]
```

```
            ...
```

```
            isFull.signal();
```

```
        } finally {
```

```
            lock.unlock();
```

Check whether  
queue is empty

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

```
    ...
```

```
    public T deq() {
```

```
        lock.lock();
```

```
        try {
```

```
            while (items.length == 0)
```

```
                isEmpty.await();
```

```
            T x = items[head]
```

```
            ...
```

```
            isFull.signal();
```

```
        } finally {
```

```
            lock.unlock();
```

Release and sleep  
if queue is empty

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

```
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head]  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();
```

Remove item from  
queue

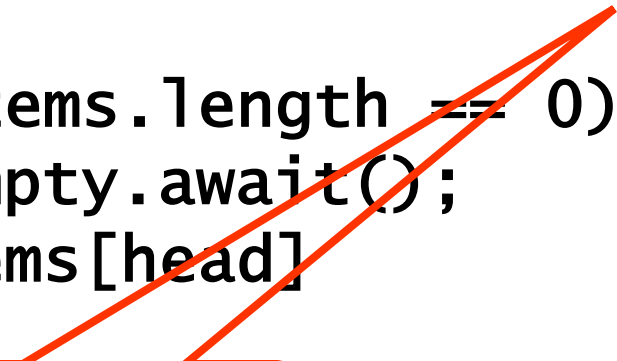


```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;
```

```
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head]
```

```
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Let Producer know  
that queue is not full  
anymore





# Monitor


- The combination of methods, mutual exclusion locks and condition objects is called a **monitor**



# Lost-Wakeup Problem

- Just like locks are vulnerable to deadlock, conditions are vulnerable to lost wakeups
- Lost wakeup:
  - One or more threads wait forever without realizing that the property for which they have been waiting has become true





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



# Lost-Wakeup Problem

- Always make sure that all necessary threads are signalled
- When using multiple consumer and producer threads (for example)
  - Make sure that you signal all processes waiting on a property and
  - Specify a timeout when waiting



# Readers-Writers Locks

- Many shared objects have the property of having readers that only access the object without modifying it and writers that modify the object



# Readers-Writers Locks

- Should readers be synchronized?
- Should writers be synchronized?
- How is synchronization achieved?



# Readers-Writers Locks

- A readers-writers lock should allow multiple readers or a single writer to enter the critical section



# Readers-Writers Locks Interface

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



# Readers-Writers Locks

- The interface should satisfy the following properties:
  - No thread can acquire the write lock while any thread holds either the write lock or the read lock
  - No thread can acquire the read lock while any other thread holds the write lock



# Questions

- How can we make sure that all other readers are finished before acquiring a write lock?
- How can we make sure that a writer is not currently holding a lock?
- When should the threads block?
- When should the threads wake up?





# Solutions

- We need a way of specifying how many readers and writers are currently contending for the lock



# SimpleReadWriteLock

```
public class SimpleReadWriteLock
    implements ReadWriteLock {
    boolean writer;
    int readers;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
```

# SimpleReadWriteLock

```
public SimpleReadWriteLock() {  
    writer = false;  
    readers = 0;  
    lock = new ReentrantLock();  
    readLock = new ReadLock();  
    writeLock = new WriteLock();  
    condition = lock.newCondition();  
}
```

No initial writers

# SimpleReadWriteLock

```
public SimpleReadWriteLock() {  
    writer = false;  
    readers = 0;  
    lock = new ReentrantLock();  
    readLock = new ReadLock();  
    writeLock = new WriteLock();  
    condition = lock.newCondition();  
}
```

No initial readers

# SimpleReadWriteLock

```
public SimpleReadWriteLock() {
```

```
    writer = false;
```

```
    readers = 0;
```

```
    lock = new ReentrantLock();
```

```
    readLock = new ReadLock();
```

```
    writeLock = new WriteLock();
```

```
    condition = lock.newCondition();
```

Separate locks for  
readers and writers



# Reader Lock

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

# Reader Lock

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

Acquire lock

# Reader Lock

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

Suspend while  
there is a writer



# Reader Lock

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

Increase number  
of readers

readers++;

# Reader Lock

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

*Acquire Lock*

# Reader Lock

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

Decrease number  
of readers

# Reader Lock

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

If there are no  
more readers,  
wake the writer



# Writer Lock

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



# Writer Lock

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



# Readers-Writers Lock

- The SimpleReadWriteLock is correct, but is it fair?
- If readers are much more frequent than writers, the writers can be locked out for a long time



# Fair Readers-Writers Lock

- A fair implementation would imply that no readers are allowed to acquire the lock after the writer has acquired it





# FIFOReadWriteLock

```
public class FIFOReadWriteLock
    implements ReadWriteLock {
    boolean writer;
    int readAcquires, readReleases;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
```


# FIFOReadWriteLock

```
public class FIFOReadWriteLock
    implements ReadWriteLock {
    boolean writer;
    int readAcquires, readReleases;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
```

Number of  
readers that  
have acquired  
the lock

# FIFOReadWriteLock

```
public class FIFOReadWriteLock
    implements ReadWriteLock {
    boolean writer;
    int readAcquires, readReleases;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
```



Number of  
readers that  
have released  
the lock



# FIFO Reader Lock

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



# FIFO Reader Lock

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

# FIFO Reader Lock

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

If all the readers that  
have acquired the lock  
released the lock

# FIFO Writer Lock

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



# FIFO Writer Lock

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





# Locks

- Using the locks described in Chapter 2 and 7, a thread that attempts to reacquire a lock it already holds will deadlock with itself
- This situation can arise if a method that acquires a lock makes a nested call to another method that acquires the same lock



# ReentrantLock

- A lock is reentrant if it can be acquired multiple times by the same thread
- `java.util.concurrent.locks` package provides `ReentrantLocks`



# ReentrantLock

- A ReentrantLock is owned by the last thread who successfully locked it, but not yet unlocked it
- A thread will successfully hold the lock when the lock is not owned by another thread
- The lock will return immediately if the current thread already holds the lock



# Our own ReentrantLock

```
public class SimpleReentrantLock
    implements Lock {
    Lock lock;
    Condition condition;
    int owner;
    int holdCount;

    ...
}
```

# Our own ReentrantLock

```
public class SimpleReentrantLock
    implements Lock {
    Lock lock;
    Condition condition;
    int owner;
    int holdCount;
    ...
}
```

The ID of the  
last thread to  
acquire the lock

# Our own ReentrantLock

```
public class SimpleReentrantLock
    implements Lock {
    Lock lock;
    Condition condition;
    int owner;
    int holdCount;
    ...
}
```

Incremented  
each time lock is  
acquired

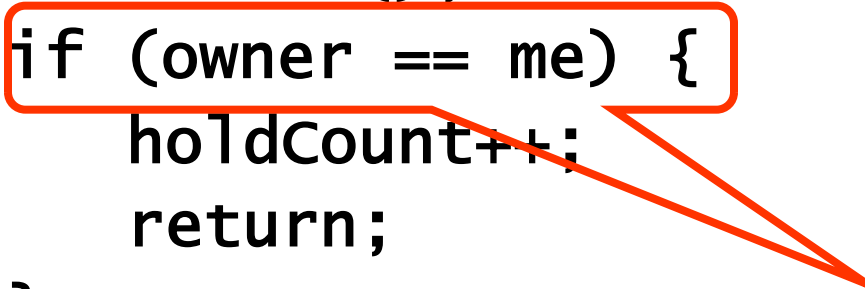


# Our own ReentrantLock

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

# Our own ReentrantLock

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



Do I already hold  
the lock? - then  
I can access the  
lock



# Our own ReentrantLock

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

Otherwise, if the holdCount is not 0 then another thread is holding the lock

# Our own ReentrantLock

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

You cannot  
unlock a lock that  
is not yours



# Mutual exclusion locks

- A mutual exclusion lock guarantees that only one thread can enter the critical section
- If another thread wants to enter the critical section while it is occupied, it suspends itself until the other thread notifies it to try again



# Semaphore

- A generalization of a mutual exclusion lock
- Instead of allowing only one thread into the CS, it allows at most  $c$  – where  $c$  is the capacity

# Semaphore

```
public class Semaphore {  
    Lock lock;  
    Condition condition;  
    int capacity;  
    int state;  
    ...  
}
```

The max amount  
of threads  
allowed in the CS

# Semaphore

```
public class Semaphore {  
    Lock lock;  
    Condition condition;  
    int capacity;  
    int state;  
    ...  
}
```

The current  
number of  
threads in the  
CS



# Semaphores

```
public void lock() {  
    lock.lock();  
    try {  
        while (state == capacity)  
            condition.await;  
        state++;  
    } finally {  
        lock.unlock();  
    }  
}
```



# Semaphores

```
public void unlock() {  
    lock.lock();  
    try {  
        state--;  
        condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```