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

```java
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
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

Never mind what this method does
Lock Conditions

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

Release lock and wait on condition
public interface Condition {
    void await();
    boolean await(long time, TimeUnit unit);
    ...
    void signal();
    void signalAll();
}

Wake up one waiting thread
Lock Conditions

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

Wake up all waiting threads
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

lock()

waiting room

unlock()
Unsuccessful Deq

lock()

deq()

Critical Section

waiting room

await()

Oh no, empty!

Art of Multiprocessor Programming
Another One

lock()

deq()

waiting room

Critical Section

await()

Oh no, empty!
Enqueuer to the Rescue

lock()
enq()

waiting room

signalAll()
unlock()

Yawn!

Critical Section
Monitor Signaling

Awakened thread might still lose lock to outside contender…
Dequeuers Signaled

Critical Section

waiting room

Yawn!

Found it
Dequeuers Signaled

Still empty!
Dollar Short + Day Late

Critical Section

waiting room
Java Synchronized Methods

```java
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;
    }

    ...
}
```
Java Synchronized Methods

```java
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

```java
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
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;
    }

    ...

}}

Signal all threads waiting on condition
Beware Lost Wake-Ups

lock() → Critical Section → enq( ) → Queue empty → so signal ( ) → unlock() → Yawn!

waiting room
Lost Wake-Up

Critical Section

waiting room

Yawn!

queue not empty so no need to signal

unlock()
Lost Wake-Up

Yawn!

waiting room

Art of Multiprocessor Programming
Lost Wake-Up

Critical Section

waiting room

Found it
What’s Wrong Here?

Critical Section

waiting room

Still waiting ....!
Solution to Lost Wakeup

Always use

\texttt{signalAll() and notifyAll()}

Instead of

\texttt{signal() and notify()}
WAKE-UP

Hard to detect!

No crashes, SEGFAULTs, etc, …

Just slowly drains resources

Beware!