Locks (Java 1.5)

interface Lock {
    void lock();
    void unlock();
    ... /* Some more stuff, also */
}
class ReentrantLock implements Lock {
    ... }

- Only one thread can hold a lock at once
  - Other threads that try to acquire it block (or become suspended) until lock becomes available

- Reentrant lock can be reacquired by same thread
  - As many times as desired
  - No other thread may acquire lock until has been released same number of times has been acquired
public class Example extends Thread {
    private static int cnt = 0;
    static Lock lock = new ReentrantLock();
    public void run() {
        lock.lock();
        try {
            int y = cnt;
            cnt = y + 1;
        } finally { lock.unlock(); }
    }
}

Lock, for protecting the shared state

Acquires the lock;
Only returns when not held by another thread

Releases the lock
Suppose we are communicating with a shared variable
   - E.g., some kind of a buffer holding messages

One thread *produces* input to the buffer
One thread *consumes* data from the buffer
How do we implement this?
   - Use condition variables
Conditions (Java 1.5)

```java
interface Lock {
    Condition newCondition(); ...
}
interface Condition {
    void await();
    void signalAll(); ...
}
```

- **Condition created from a Lock**
- **await** called with lock held
  - Releases the lock
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread
- **signalAll** called with lock held
  - Resumes all threads on lock’s wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don’t need to do it explicitly)
Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;

void produce(Object o) {
    lock.lock();
    try {
        while (valueReady)
            ready.await();
        value = o;
        valueReady = true;
        ready.signalAll();
    } finally {
        lock.unlock();
    }
}

Object consume() {
    lock.lock();
    try {
        while (!valueReady)
            ready.await();
        Object o = value;
        valueReady = false;
        ready.signalAll();
    } finally {
        lock.unlock();
    }
```
Use This Design

• This is the right solution to the problem
  – Tempting to try to just use locks directly
  – Very hard to get right
  – Problems with other approaches often very subtle
    • E.g., double-checked locking is broken
Broken Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(Object o) {
    lock.lock();
    while (!valueReady);
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    lock.lock();
    while (!valueReady);
    Object o = value;
    valueReady = false;
    lock.unlock();
}

Threads wait with lock held – no way to make progress
```
Broken Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(object o) {
    while (valueReady);
    lock.lock();
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    while (!valueReady);
    lock.lock();
    Object o = value;
    valueReady = false;
    lock.unlock();
}
```

valueReady accessed without a lock held – race condition
Broken Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;

void produce(object o) {
    lock.lock();
    if (valueReady)
        ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    if (!valueReady)
        ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
}

what if there are multiple producers or consumers?
```
More on the Condition Interface

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

• `await(t, u)` waits for time `t` and then gives up
  – Result indicates whether woken by signal or timeout
• `signal()` wakes up only *one* waiting thread
  – Tricky to use correctly
    • Have all waiters be equal -- use multiple conditions to make this happen
    • handle exceptions correctly
  – Highly recommended to just use `signalAll()`
Await and SignalAll Gotcha’s

• **await** *must* be in a loop
  – Don’t assume that when wait returns conditions are met

• Avoid holding other locks when waiting
  – **await** only gives up locks on the lock that owns the Condition you await on
More locks

- ReadWriteLock
  - holds a pair of coupled locks
    - a read lock, and
    - a write lock
- Write lock is exclusive: if any thread holds it, no thread can hold any other lock
- Read lock is non-exclusive: other threads may simultaneously acquire locks on the read lock
Blocking Queues in Java 1.5

• Interface for producer/consumer pattern

```java
interface Queue<E> extends Collection<E> {
    boolean offer(E x); /* produce */
    /* waits for queue to have capacity */
    E remove();       /* consume */
    /* waits for queue to become non-empty */
    ... 
}
```

• Two handy implementations
  – `LinkedBlockingQueue` (FIFO, may be bounded)
  – `ArrayBlockingQueue` (FIFO, bounded)
  – (plus a couple more)
Wait and NotifyAll (Java 1.4)

- Recall that in Java 1.4, use synchronize on object to get associated lock

- Objects also have an associated wait set
Wait and NotifyAll (cont’d)

• o.wait()
  – Must hold lock associated with o
  – Release that lock
    • And no other locks
  – Adds this thread to wait set for lock
  – Blocks the thread

• o.notifyAll()
  – Must hold lock associated with o
  – Resumes all threads on lock’s wait set
  – Those threads must reacquire lock before continuing
    • (This is part of the function; you don’t need to do it explicitly)
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

    synchronized void produce(Object o) {
        while (valueReady) wait();
        value = o; valueReady = true;
        notifyAll();
    }

    synchronized Object consume() {
        while (!valueReady) wait();
        valueReady = false;
        Object o = value;
        notifyAll();
        return o;
    }
}

Producer/Consumer in Java 1.4
Thread Cancellation

• Example scenarios: want to cancel thread
  – Whose processing the user no longer needs (i.e., she has hit the “cancel” button)
  – That computes a partial result and other threads have encountered errors, … etc.

• Java used to have `Thread.kill()`
  – But it and `Thread.stop()` are deprecated
  – Use `Thread.interrupt()` instead
Thread.interrupt()

- Tries to wake up a thread
  - Sets the thread’s interrupted flag
  - Flag can be tested by calling
    - `interrupted()` method
      - Clears the interrupt flag
    - `isInterrupted()` method
      - Does not clear the interrupt flag

- Won’t disturb the thread if it is working
  - Not asynchronous!
public class CancellableReader extends Thread {
    private FileInputStream dataFile;
    public void run() {
        try {
            while (!Thread.interrupted()) {
                try {
                    int c = dataFile.read();
                    if (c == -1) break;
                    else process(c);
                } catch (IOException ex) { break; }
            }
        } finally { // cleanup here }
    }
}

What if the thread is blocked on a lock or wait set, or sleeping when interrupted?

This could acquire locks, be on a wait set, etc.
InterruptedException

- Exception thrown if interrupted on certain ops
  - `wait`, `await`, `sleep`, `join`, and `lockInterruptibly`
  - Also thrown if call one of these with interrupt flag set
- *Not thrown* when blocked on 1.4 lock or I/O

```java
class Object {
    void wait() throws IE;
    ...
}
interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    ...
}
interface Condition {
    void await() throws IE;
    void signalAll();
    ...
}
```