Condition Variables

• Want access to shared data, but only when some condition holds
  – Implies that threads play different roles in accessing shared data
• Examples
  – Want to read shared variable v, but only when it is non-null
  – Want to insert myself in a data structure, but only if it is not full

CVs: Use Wait and Notify

To wait for a condition to become true:

```java
synchronized (obj) {
    while (condition does not hold)
        obj.wait();
    … perform appropriate actions
}
```

To notify waiters that a condition has changed:

```java
synchronized (obj) {
    … perform actions that change condition
    obj.notifyAll();
}
```

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
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;
    synchronized void produce(Object o) {
        while (valueReady) {
            value = o; valueReady = true;
        }
        synchronized (this) {
            value = o;
            valueReady = true;
        }
    }
    synchronized Object consume() {
        while (!valueReady) {
            synchronized (this) {
                valueReady = false;
            }
        }
        Object o = value;
        value = null;
        return o;
    }
}
```

notify() vs. notifyAll()

- Very tricky to use notify() correctly
  - notifyAll() generally much safer
- To use notify() correctly, should:
  - Have all waiters be equal
    - Each notify only needs to wake up one thread
    - Doesn’t matter which thread it is
  - Handle exceptions correctly
    - Including InterruptedException
- For this course, just use notifyAll()

Wait and Notify Gotcha’s

- wait must be in a loop
  - Don’t assume that when wait returns conditions are met
- Avoid holding other locks when waiting
  - Wait only gives up locks on the object you wait on
Aspects of Synchronization

• Atomicity
  – Locking to obtain mutual exclusion
  – What we most often think about

• Visibility
  – Ensuring that changes to object fields made in one thread are seen in other threads

• Ordering
  – Ensuring that you aren’t surprised by the order in which statements are executed

Quiz Time

• Can this result in i=0 and j=0?

How Can This Happen?

• Compiler can reorder statements
  – Or keep values in registers
• Processor can reorder them
• On multi-processor, values not synchronized in global memory

Doesn’t Seem Possible...

• But this can happen!
When Are Actions Visible?

Forcing Visibility of Actions

- All writes from thread that holds lock M are visible to next thread that acquires lock M
  - Must be the same lock

- Use synchronization to enforce visibility and ordering
  - As well as mutual exclusion

Volatile Fields

- If you are going to access a shared field without using synchronization
  - It needs to be volatile
- Semantics for volatile have been strengthened in JSR-133
  - Many VM’s already compliant
- If you don’t try to be too clever
  - Declaring it volatile just works

Using Volatile

- A one-writer/many-reader value
  - Simple control flags:
    - volatile boolean done = false;
- Keeping track of a “recent value” of something
Misusing Volatile

• Incrementing a volatile field doesn’t work
  – In general, writes to a volatile field that depend on the previous value of that field don’t work
• A volatile reference to an object isn’t the same as having the fields of that object be volatile
  – No way to make elements of an array volatile
• Can’t keep two volatile fields in sync

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!

Cancellation Example

```java
import java.io.FileInputStream;

public class CancellableReader extends Thread {
    private FileInputStream dataFile;
    public void run() {
        try {
            while (!Thread.interrupted()) {
                try {
                    int c = dataFile.read();
                    if (c == -1) break;
                    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?
InterruptedException

- Thrown if interrupted while doing a `wait`, `sleep`, or `join`
  - Also thrown when `interrupt` flag is set and attempt to do a `wait`, `sleep`, or `join`
  - Not thrown when blocked (or blocking on) on a lock or I/O

Responses to Interruption

- Early Return
  - Clean up and exit without producing errors
  - May require rollback or recovery
  - Callers can poll cancellation status to find out why an action was not carried out
- Continuation (i.e., ignore interruption)
  - When it is too dangerous to stop
  - When partial actions cannot be backed out
  - When it doesn’t matter

Responses to Interruption (cont’d)

- Re-throw `InterruptedException`
  - When callers must be alerted on method return
- Throw a general failure exception
  - When interruption is a reason method may fail
- In general
  - Must reset invariants before cancelling
  - E.g., close file descriptors, notify other waiters, etc.

Handling `InterruptedException`

```java
synchronized (this) {
    while (!ready) {
        try { wait(); }
        catch (InterruptedException e) {
            // make shared state acceptable
            notifyAll();
            // cancel processing
            return;
        }
    // do whatever
}
}```
Why No Thread.kill()?

- What if the thread is holding a lock when it is killed? The system could
  - Free the lock, but the data structure it is protecting might be now inconsistent
  - Keep the lock, but this could lead to deadlock
- A thread needs to perform its own cleanup
  - Use InterruptedException and isInterrupted() to discover when it should cancel

Guidelines for Programming with Threads

- Synchronize access to shared data
- Don’t hold multiple locks at a time
  - Could cause deadlock
- Hold a lock for as little time as possible
  - Reduces blocking waiting for locks
- While holding a lock, don’t call a method you don’t understand
  - E.g., a method provided by someone else, especially if you can’t be sure what it locks
  - Corollary: document which locks a method acquires