Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*

```java
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```

Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on A
  - Thread 2 holds lock on B
  - Thread 1 is trying to acquire a lock on B
  - Thread 2 is trying to acquire a lock on A
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - Other than by the fact that nothing is happening
Deadlock: Wait graphs

Deadlock occurs when there is a cycle in the graph

Wait graph example

T1 holds lock on A
T2 holds lock on B
T1 is trying to acquire a lock on B
T2 is trying to acquire a lock on A
Key Ideas

• Multiple threads can run simultaneously
  – Either truly in parallel on a multiprocessor
  – Or can be scheduled on a single processor
    • Assume a running thread can be preempted at any time

• Threads can share data
  – In Java, only fields can be shared
  – Need to prevent interference
    • Synchronization, immutability, and other methods
  – Overuse use of synchronization can create deadlock
    • Violation of liveness

Guaranteeing Safety

• Ensure objects are accessible only when in a **consistent** and appropriate state
  – All invariants are maintained
  – Presents subclass obligations

• Use locks to enforce this
  – Rule of thumb 1: You must hold a lock when accessing shared data
  – Rule of thumb 2: You must not release a lock until shared data is in a valid state
Guaranteeing Liveness

• Ensuring availability of services
  – Called methods eventually execute
• Ensuring progress of activities
  – Managing resource contention
  – Freedom from deadlock
  – Fairness
  – Fault tolerance

(Switch Slides to CPJ)
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 $j=0$?
Doesn’t Seem Possible...

• But this can happen!

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
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 is not atomic
  – 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., he/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
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?
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.

```
synchronized (this) {
  while (!ready) {
    try { wait(); }
    catch (InterruptedException e) {
      // make shared state acceptable
      notifyAll();
      // cancel processing
      return;
    }
    // do whatever
  }
  // 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

Selected 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