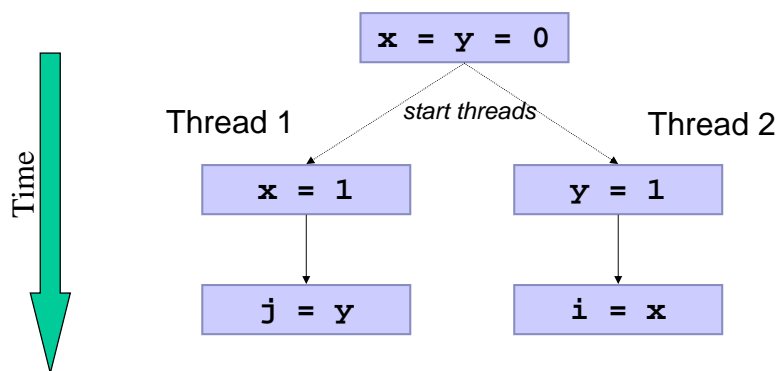


Aspects of Synchronization

- 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

69

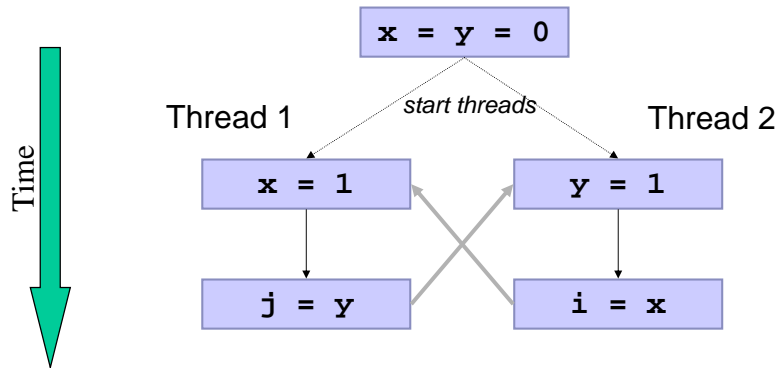
Quiz Time



Can this result in $i = 0$ and $j = 0$?

70

Answer: Yes!



How can $i = 0$ and $j = 0$?

71

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
- Must use synchronization to enforce **visibility** and **ordering**
 - As well as mutual exclusion

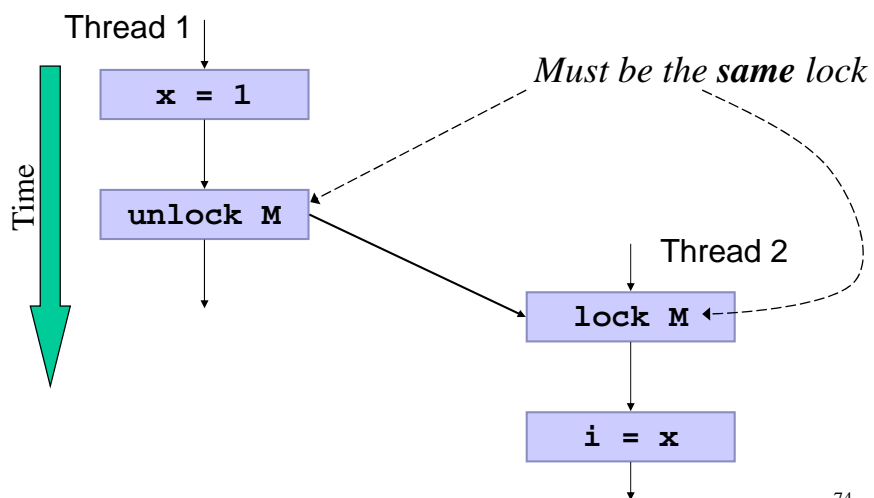
72

Synchronization Actions

```
// block until obtain lock
synchronized(anObject) {
    // get main memory value of field1 and field2
    int x = anObject.field1;
    int y = anotherObject.field2;
    anotherObject.field3 = x+y;
    // commit value of field3 to main memory
}
// release lock
moreCode();
```

73

When Are Actions Visible?



74

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

75

slide

Using Volatile

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

76

slide

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

77

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

78

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!

79

Cancellation Example

```
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 }
    }
}
```

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

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

80

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

81

Responses To Interruption

- Early Return
 - Clean up an exit without producing or signalling 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 cancellation)
 - When it is too dangerous to stop
 - When partial actions cannot be backed out
 - When it doesn't matter

82

Responses To Interruption

- Re-throwing **InterruptedException**
 - When callers must be altered on method return
- Throwing a general failure exception
 - When interruption is a reason a method can fail
- In general:
 - Must reset invariants before cancelling
 - E.g., closing file descriptors, notifying other waiters, etc.

83

Handling InterruptedException

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

84

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

85

Guidelines to simple/safe multi-threaded programming

- Synchronize access to shared data
- Don't hold a lock on more than one object 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

86

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.