

CMSC 433 – Programming Language  
Technologies and Paradigms  
Spring 2003

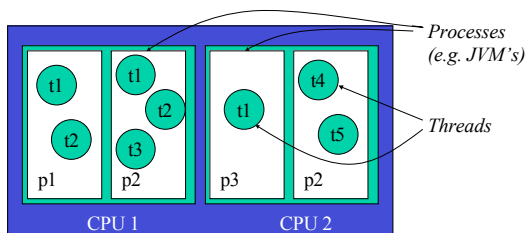
Threads and Synchronization  
April 1, 2003

Overview

- What are threads?
- Thread scheduling, data races, and synchronization
- Thread mechanisms in Java

2

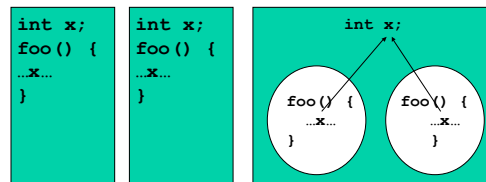
Computation Abstractions



*A computer*

3

Processes vs. Threads



*Processes do not share data*

*Threads share data within a process*

4

## So, what is a thread?

- Conceptually: it is a parallel computation occurring within a process
- Implementation view: it's a program counter and a stack. The heap and static area are shared among all threads
- All programs have at least one thread

5

## Why multiple threads?

- Performance:
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Can easily express some programming paradigms
  - Event processing
  - Simulations
- Keep computations separate, as in an OS
  - Java OS

6

## Programming Threads

- Threads are available in many languages
  - C, C++, Objective Caml, Java, SmallTalk ...
- In many languages (e.g., C and C++), threads are a platform specific add-on
  - Not part of the language specification
- Part of the Java language specification
- The thread API differs with each, but most have the basic features we will now present

7

## Thread Applications

- Web browsers
  - one thread for I/O
  - one thread for each file being downloaded
  - one thread to render web page
- Graphical User Interfaces (GUIs)
  - Have one thread waiting for each important event, like key press, button press, etc.

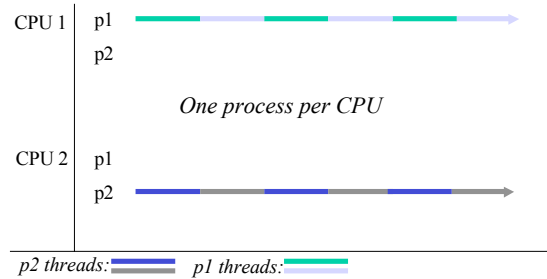
8

## Thread Scheduling

- OS schedules a single-threaded process on a single processor
- Multithreaded process scheduling:
  - One thread per processor
    - Effectively splits a process across CPU's
    - Exploits hardware-level concurrency
  - Many threads per processor
    - Need to share CPU in slices of time

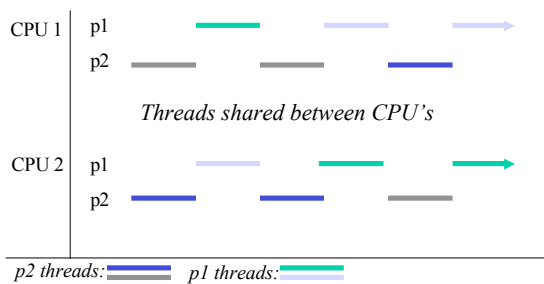
9

## Scheduling Example (1)



10

## Scheduling Example (2)



11

## Scheduling Consequences

- Concurrency
  - Different threads from the same application can be running *at the same time* on different processors
- Interleaving
  - Threads can be **pre-empted** at any time in order to schedule other threads

12

## Data Races

- Data can be shared by threads
  - Scheduler can interleave threads arbitrarily
  - Can lead to unexpected data corruption
  - This is a *data race*
- Need to avoid such data accesses.
  - Use *synchronization*.

13

## Data Race Example

```
int cnt = 0;           Shared state  cnt = 0
void thread1() {
    int y = cnt;
    cnt = y + 1;
}
void thread2() {
    int y = cnt;
    cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

14

## Data Race Example

```
int cnt = 0;           Shared state  cnt = 0
void thread1() {
    int y = cnt;   y=0
    cnt = y + 1;
}
void thread2() {
    int y = cnt;
    cnt = y + 1;
}
```

Thread1 executes, grabbing the global counter value into y.

15

## Data Race Example

```
int cnt = 0;           Shared state  cnt = 0
void thread1() {
    int y = cnt;   y=0
    cnt = y + 1;
}
void thread2() {
    int y = cnt;   y=0
    cnt = y + 1;
}
```

Thread1 is pre-empted. Thread2 executes, grabbing the global counter value into y.

16

## Data Race Example

```
int cnt = 0;
void thread1() {
  int y = cnt; y=0
  cnt = y + 1;
}
void thread2() {
  int y = cnt; y=0
  cnt = y + 1;
}
```

Shared state cnt = 1

■■■

Thread2 executes, storing the incremented cnt value.

17

## Data Race Example

```
int cnt = 0;
void thread1() {
  int y = cnt; y=0
  cnt = y + 1;
}
void thread2() {
  int y = cnt; y=0
  cnt = y + 1;
}
```

Shared state cnt = 1

■■■■

Thread2 completes. Thread1 Executes again, storing the old counter value (1) rather than the new one (2)!

18

## What happened?

- Thread1 was preempted after it read the counter but before it stored the new value.
- A particular way in which the execution of two threads is interleaved is called a *schedule*. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

19

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
void thread2() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

Lock, for protecting the shared state

Acquires the lock; only succeeds if not held by another thread

Releases the lock

20

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 0



Thread1 acquires lock l

21

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 0



Thread1 reads cnt into y

22

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 0



Thread1 is pre-empted.  
Thread2 attempts to  
acquire lock l but fails  
because it's held by  
Thread1, so it blocks

23

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 1



Thread1 runs, assigning  
to cnt

24

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 1



Thread1 releases the lock  
and terminates

25

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 1



Thread2 now can acquire  
lock l.

26

## Applying synchronization

```
int cnt = 0;
Object lock = new
Object();
void thread1() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt;
cnt = y + 1;
}
}
```

Shared state cnt = 1



Thread2 reads cnt into y.

27

## Applying synchronization

```
int cnt = 0;
Object lock
void thread1() {
synchronized(lock) {
int y = cnt; y=0
cnt = y + 1;
}
}
void thread2() {
synchronized(lock) {
int y = cnt; y=1
cnt = y + 1;
}
}
```

Shared state cnt = 2



Thread2 assigns cnt,  
then releases the lock

28

## Synchronization not a Panacea

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

```
Object lock = new Object();
Object lock2 = new Object();
void thread1() {
    synchronized (lock) {
        synchronized (lock2) {
            ...
        }
    }
}
void thread2() {
    synchronized (lock2) {
        synchronized (lock) {
            ...
        }
    }
}
```

29

## Other Thread Operations

- **Condition variables: wait and notify**
  - Alternative synchronization mechanism
- **Yield**
  - Voluntarily give up the CPU
- **Sleep**
  - Wait for a certain length of time

30

## Thread Lifecycle

- While a thread executes, it goes through a number of different phases
  - **New**: created but not yet started
  - **Runnable**: either running, or able to run on a free CPU
  - **Blocked**: waiting for I/O or on a lock
  - **Sleeping**: paused for a user-specified interval

31

## Java Threads

- The class `java.lang.Thread`
  - Implements the basic threading abstraction
  - Can extend this class to create your own threads
- The interface `java.lang.Runnable`
  - Can create a thread by passing it a class that implements this interface
  - Favors composition over inheritance; more flexible

32

## Extending class Thread

- Can build a thread class by extending **java.lang.Thread**
- Must supply a public void run() method
- Start a thread by invoking the start() method
- When a thread starts, executes run()
- When run() returns, thread is finished/dead

33

## Example: Synchronous alarms

```
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);

    // wait (in secs)
    try {
        Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println(""+timeout+" "+msg);
}
```

34

## Making it Threaded (1)

```
public class AlarmThread extends Thread {
    private String msg = null;
    private int timeout = 0;

    public AlarmThread(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }

    public void run() {
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) { }
        System.out.println(""+timeout+" "+msg);
    }
}
```

35

## Making it Threaded (2)

```
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);

    // wait (in secs) asynchronously
    if (t != null)
        t.start();
}
```

36

## Runnable interface

- Extending Thread means can't extend any other class
- Instead implement **Runnable**
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)

37

## Thread example revisited

```
public class AlarmRunnable implements Runnable {
    private String msg = null;
    private int timeout = 0;

    public AlarmRunnable(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }

    public void run() {
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) { }
        System.out.println (" "+timeout+" " +msg);
    }
}
```

38

## Change is in parseInput

- Old parseInput does
  - return new AlarmThread(m,t);
- New parseInput does
  - return new Thread(new AlarmRunnable(m,t));
- Code in while loop doesn't change

39

## Another example

```
public class ThreadDemo implements Runnable {
    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println(i);
            try { Thread.sleep(1000); }
            catch (InterruptedException e) { }; }
        System.out.println("exiting " + Thread.currentThread());
    }
    public static void main(String [] args) {
        Thread t = new Thread(new ThreadDemo(), "Demo Thread");
        System.out.println("main thread: " +
            Thread.currentThread());
        System.out.println("Thread created: " + t);
        t.start();
        try { Thread.sleep(3000); }
        catch (InterruptedException e) { };
        System.out.println("exiting " + Thread.currentThread());
    }
}
```

40

## InterruptedException

- A number of thread methods throw it
  - really means: “wake-up call!”
- **interrupt()** tries to wake up a thread
- Won't disturb the thread if it is working
- Will wake up the thread if it is sleeping, or otherwise waiting (or will do so when such a state is entered)
  - Thrown by **sleep()**, **join()**, **wait()**

41

## Thread scheduling

- When multiple threads share a CPU, must decide:
  - When the current thread should stop running
  - What thread to run next
- A thread can voluntarily **yield()** the CPU
  - call to **yield** may be ignored; don't depend on it
- *Preemptive schedulers* can de-schedule the current thread at any time
  - But not all JVM implementations use preemptive scheduling; so a thread stuck in a loop may *never* yield by itself. Therefore, put **yield()** into loops
- Threads are descheduled whenever they block (e.g. on a lock or on I/O) or go to sleep

42

## Which thread to run next?

- The scheduler looks at all of the runnable threads; these will include threads that were unblocked because
  - A lock was released
  - I/O became available
  - They finished sleeping, etc.
- Of these threads, it considers the thread's priority. This can be set with **setPriority()**. Higher priority threads get preference.
  - Oftentimes, threads waiting for I/O are also preferred.

43

## Simple thread methods

- void **start()**
- boolean **isAlive()**
- void **setPriority(int newPriority)**
  - thread scheduler might respect priority
- void **join()** throws **InterruptedException**
  - waits for a thread to die/finish

44

## Example: threaded, sync alarm

```
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);

    // wait (in secs) asynchronously
    if (t != null)
        t.start();
    // wait for the thread to complete
    t.join();
}
```

45

## Simple static thread methods

- void yield()
  - Give up the CPU
- void sleep(long milliseconds)
  - throws InterruptedException
  - Sleep for the given period
- Thread.currentThread()
  - Thread object for currently executing thread
- All apply to thread invoking the method

46

## Daemon threads

- void setDaemon(boolean on)
  - Marks thread as a daemon thread
  - Must be set before thread started
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons

47

## Synchronization Topics

- Locks
- **synchronized** statements and methods
- **wait** and **notify**
- Deadlock

48

## Locks

- *Any* Object subclass has (can act as) a lock
- Only one thread can hold the lock on an object
  - other threads block until they can acquire it
- If your thread already holds the lock on an object
  - can lock many times
  - Lock is released when object unlocked the corresponding number of times
- No way to only attempt to acquire a lock
  - Either succeeds, or blocks the thread

49

## Synchronized statement

- **synchronized (obj) { statements }**
- Obtains the lock on **obj** before executing statements in block
- Releases the lock when the statements block completes

50

## Recasting earlier example

```
public class State {
    public int cnt = 0;
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        int y = s.cnt; ← Unsynchronized access to
        s.cnt = y + 1; shared data!
    }
}
public void main(String args[]) {
    State s = new State();
    MyThread thread1 = new MyThread(s);
    MyThread thread2 = new MyThread(s);
    thread1.start(); thread2.start();
}
}
```

51

## Adding Synchronization

```
public class State {
    public int cnt = 0;
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        synchronized (s) { Uses s as a lock, forces
            int y = s.cnt; exclusive access
            s.cnt = y + 1;
        }
    }
}
public void main(String args[]) {
    State s = new State();
    MyThread thread1 = new MyThread(s);
    MyThread thread2 = new MyThread(s);
    thread1.start(); thread2.start();
}
}
```

52

## Synchronized methods

- A method can be synchronized
  - add **synchronized** modifier before return type
- Obtains the lock on object referenced by **this**, before executing method
  - releases lock when method completes
- For a **static synchronized** method
  - locks the class object

53

## Synchronization example

```
public class State {
    private int cnt = 0;
    public int synchronized incCnt(int x) {
        cnt += x;
    }
    public int synchronized getCnt() { return cnt; }
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        s.incCnt(1)
    }
}
public void main(String args[]) {
    State s = new State();
    MyThread thread1 = new MyThread(s);
    MyThread thread2 = new MyThread(s);
    thread1.start(); thread2.start();
}
}
```

*Synchronization occurs in State object itself, rather than in its caller.*

54

## Synchronization Style

- Design decision
  - Internal synchronization (class is thread-safe)
    - Have a stateful object synchronize itself (e.g. with synchronized methods)
  - External synchronization (class is thread-compatible)
    - Have callers perform synchronization before calling the object
- Can go both ways:
  - Thread-safe: Random
  - Thread-compatible: ArrayList, HashMap, ...

55

## 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

56

## CVs: Use Wait and Notify

To wait for a condition to become true:

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

To notify waiters that a condition has changed:

```
synchronized (obj) {  
    ... perform actions that change condition  
    obj.notifyAll();  
    or obj.notify(); // using notify() is a tricky optimization  
}
```

57

## Broken Producer/Consumer Example

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

58

## Broken Producer/Consumer Example

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

59

## Broken Producer/Consumer Example

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

60

## Producer/Consumer Example

```
public class ProducerConsumer {
    private boolean bufferFull = false;
    private Object value;

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

    synchronized Object consume()
        throws InterruptedException {
        while (!bufferFull) wait();
        bufferFull = false;
        Object o = value;
        value = null; // why do we do this?
        notifyAll();
        return o;
    }
}
```

61

## Wait and Notify

- Must be called while lock is held on a
- **a.wait()**
  - releases the lock on a
    - But not any other locks acquired by this thread
  - adds the thread to the *wait set* for a
  - blocks the thread
- **a.wait(int m)**
  - limits wait time to **m** milliseconds (but see below)

62

## Wait and Notify (cont.)

- **a.notify()** resumes *one* thread from a's wait set
  - no control over which thread
- **a.notifyAll()** resumes *all* threads on a's wait set
- resumed thread(s) must reacquire lock before continuing
  - (Java inserts the reacquires automatically)

63

## 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 1 thread
    - doesn't matter which thread it is
  - handle exceptions correctly
    - including InterruptedException
- For this course, just use notifyAll()

64

## 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 support Thread.kill()
  - But it and Thread.stop() are deprecated
  - Use Thread.interrupt() instead

65

## Why no Thread.kill()?

- What if the thread is holding a lock when it is killed? The system could
  - free the lock, but the datastructure 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

66

## 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 }
    }
}
```

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

67

## InterruptedException

- Will be thrown if interrupted either while doing or attempting to do a wait, sleep, or join
  - But not when blocked (or blocking on) on a lock or I/O
- Must reset invariants before cancelling
  - E.g. closing file descriptors, notifying other waiters, etc.

68

## InterruptedException Example

- Threads t1 and t2 are waiting
- Thread t3 performs a notify
  - thread t1 is selected
- Before t1 can acquire lock, t1 is interrupted
- t1's call to wait throws InterruptedException
  - t1 doesn't process notification
  - t2 doesn't wake up

69

## Handling InterruptedException

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

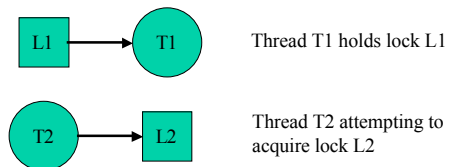
70

## 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

71

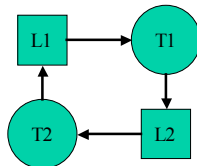
## Deadlock: Wait graphs



Deadlock occurs when there is a cycle in the graph

72

## Wait graph example



T1 holds lock on **L1**  
T2 holds lock on **L2**  
T1 is trying to acquire a lock on **L2**  
T2 is trying to acquire a lock on **L1**

73

## Livelock

- Deadlock arises when *blocked* threads cannot execute
- *Livelock* occurs when threads actually are executing, but no work gets done.
  - Use `notify()` rather than `notifyAll()` and the wrong thread keeps waking up with its condition not met

74

## Field Visibility

- Threads might cache values
- Obtaining a lock forces the thread to get fresh values
- Releasing a lock forces the thread to flush out all pending writes
- **volatile** variables are never cached
- **sleep(...)** doesn't force fresh values
- Many compilers don't perform these optimizations
  - but some do (Hotspot server does)
- Problem might also occur with multiple CPUs

75

## 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

76

## Guidelines (cont.)

- Have to go beyond these guidelines for more complex situations
  - but need to understand threading and synchronization well
- We'll discuss threads more from the textbook *Concurrent Programming in Java* and from a talk at a Java conference by Bill Pugh and Doug Lea

77