

Threads and Concurrency

CSE 7345
Fall 2008

Threads and Concurrency

Part I

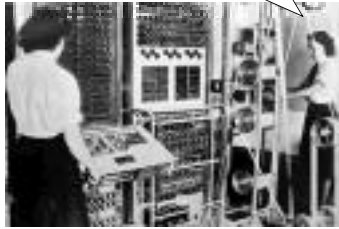
The Process and the Thread

Process

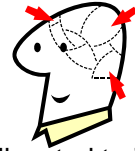
- A program is static (dead)
- A process is dynamic (alive)
- The process is the unit of work in modern operating systems

Origin of the OS Process

She's been hogging the machine all morning and I can't get anything done!



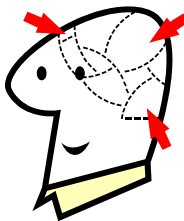
A process must..



- Get some kind of ID
- Be created: have memory allocated to it
 - program, state, resource list
- Be given the CPU for execution
- Be removed: memory reclaimed, resources freed

On a single CPU only ONE process can be running at one time

What does a process know?



- What program it's running
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

What about Threads?

Threads vs Processes

- A process has its own address space
 - In a multitasking operating system, each program is run as a separate process
 - Process switching has overhead
- A thread shares the address space of the the program that created it
 - minimal overhead with thread switching

Why Threads?

- Provides parallel computation with low overhead
- Use threads when a program may need to wait for some resource
 - disk access, network connection
- When one thread is waiting, other can continue processing

Three Categories of Thread Usage

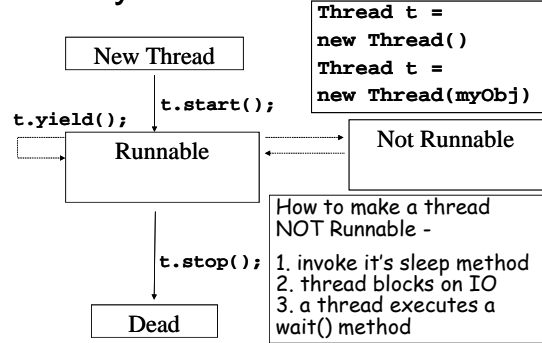
- Independent threads
 - each doing a part of an overall task
 - no interference
 - issue: how wait on all threads to complete
- Data sharing threads
 - each may interfere with data
 - issue: how to guarantee safety & liveness
- Coordinating threads
 - how to structure integrated coordinated action

Key Properties for Concurrent Software

- Safety –
 - the property that nothing bad ever happens
 - bad = random, unexpected behavior
- Liveness
 - the property that the program will make progress (and not stall)

Java Threads

Life Cycle of a Thread



Two ways to create your own thread

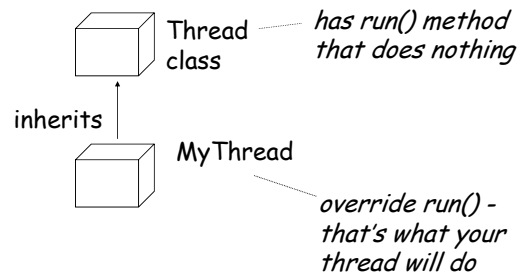
- Extend the Thread class and override the run () method
- Implement the Runnable interface
 - define run ()

either way, there is a thread object

whatever code is in run () will be run as a separate thread



Subclass Thread



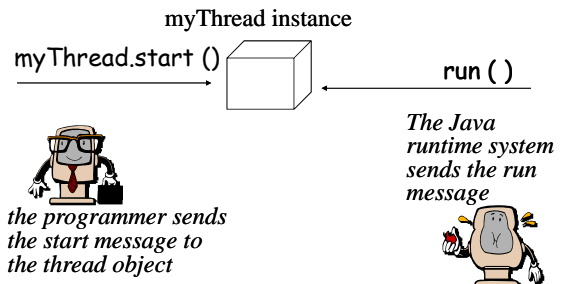
Subclassing Thread

```
class SimpleThread extends Thread {
    private String internalName;

    SimpleThread (String name) {
        internalName = name;
    }

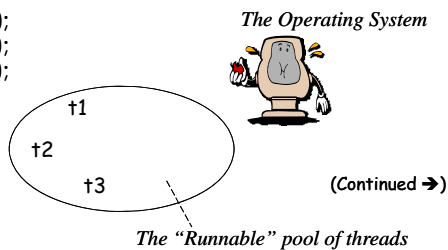
    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(internalName);
        }
    }
}
```

Activating your thread



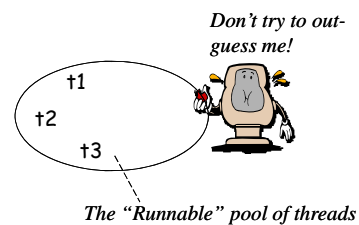
```
public class SimpleThreadTest1 {
    public static void main (String argv[]) {
        SimpleThread t1 = new SimpleThread("sun");
        SimpleThread t2 = new SimpleThread("java");
        SimpleThread t3 = new SimpleThread("beans");

        t1.start();
        t2.start();
        t3.start();
    }
}
```



```
class SimpleThread extends Thread {
    String internalName;

    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(internalName);
        }
    }
}
```



Output:

```
sun
sun
java
java
java
sun
beans
beans
sun
java
java
sun
beans
sun
beans
beans
```

Hog Prevention with yield()

- Use yield to prevent a thread from hogging the CPU

```
public void run() {  
    for (int i=0; i<8; i++) {  
        System.out.println(internalName);  
        Thread.yield();  
    }  
}
```

```
>java SayNameTest  
sun.java..sun..java..sun..java..sun..java..  
sun.java..sun..java..java..sun..java..sun..
```

Threads and Runnable

```
class PrimeRun implements Runnable {  
    long minPrime;  
  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime; }  
  
    public void run() {  
        // compute primes larger than minPrime . . .  
    }  
}
```

Three Categories of Thread Usage

- Independent threads
 - each doing a part of an overall task
 - no interference
 - issue: how wait on all threads to complete
- Data sharing threads
 - each may interfere with data
 - issue: how to guarantee safety & liveness
- Coordinating threads
 - how to structure integrated coordinated action

Partitioning Work with Threads

Three Work Threads

```
t1.start();  ↷ returns immediately
t2.start();  ↷
t3.start();  ↷
System.out.println("done");
```

will print *before* threads
complete their work

Using join() to wait for another Thread

- public final void **join**(long millis)
throws InterruptedException
 - Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.
- public final void **join**()
throws InterruptedException
 - Waits for this thread to die

Three Work Threads

```
t1.start();  ↷ returns immediately
t2.start();  ↷
t3.start();  ↷
t1.join();   ———→ main thread will block until t1
               is no longer alive
t2.join();
t3.join();
System.out.println("done");
```

will print after all threads
complete their work

Thread Priority

- Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.
- Some constants for priorities include
`Thread.MIN_PRIORITY`
`Thread.MAX_PRIORITY`
`Thread.NORM_PRIORITY`

Three Categories of Thread Usage

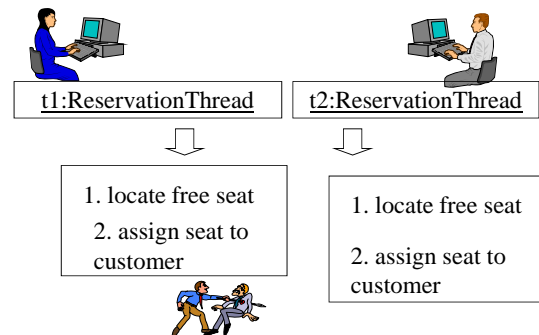
- Independent threads
 - each doing a part of an overall task
 - no interference
 - issue: how wait on all threads to complete
- Data sharing threads
 - each may interfere with data
 - issue: how to guarantee safety & liveness
- Coordinating threads
 - how to structure integrated coordinated action

Threads II Safety, Liveness and Thread Synchronization

Safety

- Similar to the notion of type safety
 - A typed –checked program may not be correct but it never encounters errors due to corruption of representation
- However, types can be checked at compile time
 - safety requires programmer discipline

Need for Mutual Exclusion



Safe Strategy I: Immutable Objects

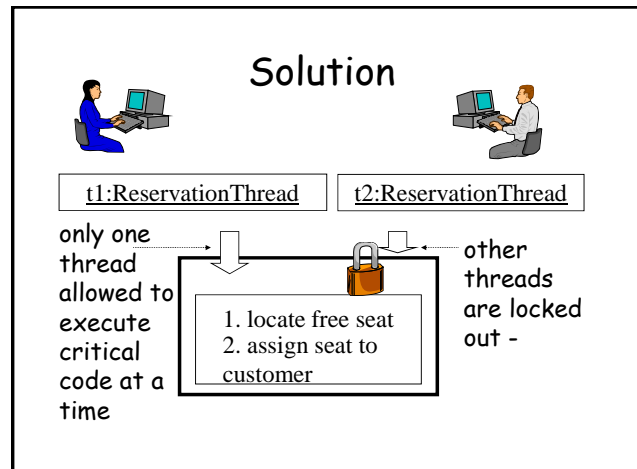
- Bypass the need to worry about synchronizing access from different threads
- No state change – create new when needed
- But, not very useful when there is user interaction but selective use is fine
 - java.lang.String
 - java.lang.Integer
 - java.lang.Color

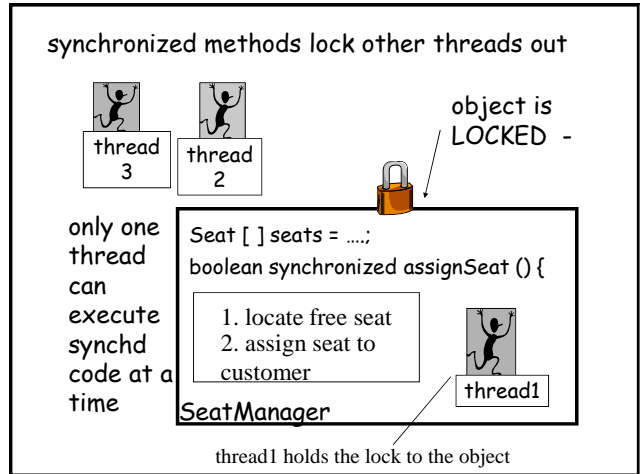
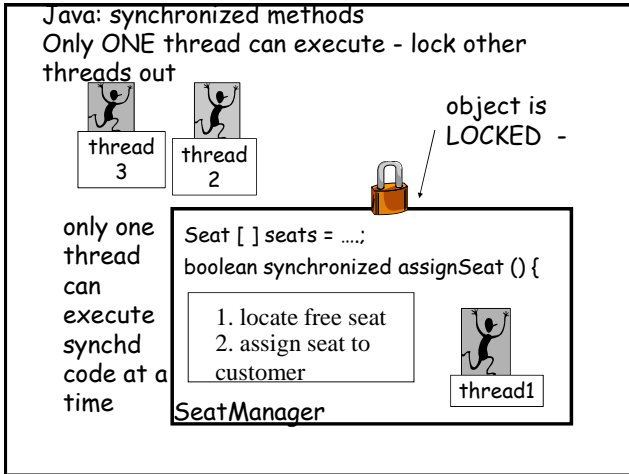
Immutable Object - Example

- a Sorter object that:
 - pass array in as a parameter
 - returns the sorted array
 - no state maintained
 - thread safe

Safe Strategy II: Synchronized Objects

- When visible objects change state, synchronization is necessary to ensure that changes occur only in consistent ways
- Safety preservation requires the avoidance of two kinds of conflicts that might occur
 - read/write conflict
 - reading values that are transient / inconsistent
 - write/write conflict
 - interleaving writes – e.g. ++i
 - start with 0, end result may be 1 or 2





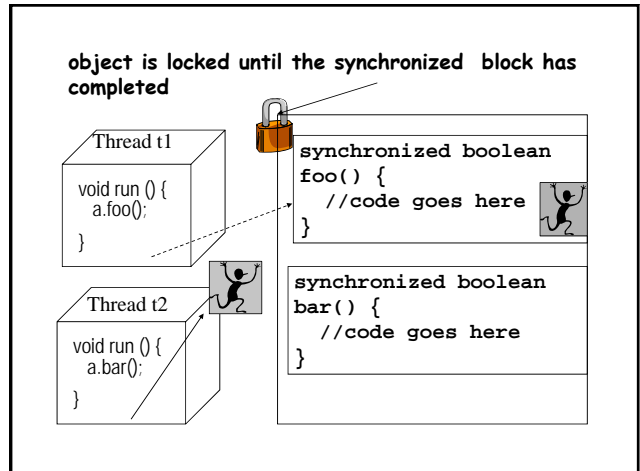
```

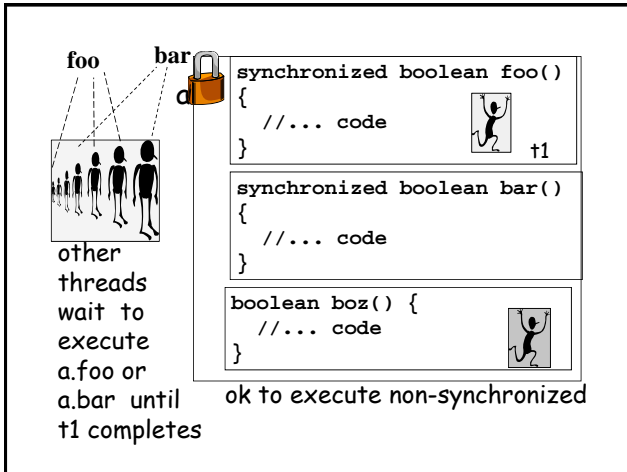
class Account {
    private double balance;

    public Account (double initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized double getBalance () {
        return balance;
    }
    public synchronized void deposit (double amount) {
        balance += amount;
    }
}

```

Multiple synchronized methods per object





Synchronized Block

- Allows locking an object without a synchronized method
- Form:

```

synchronized (<some object>) {
  statements;
}

```

some object to lock

What about variables?

- Variables cannot be synchronized
- However you can control access if you:
 - declare variables private
 - provide accessor methods

```

class Account {
  private double balance;
  synchronized void updateBalance (double amt) {
    balance += amt;
  }
}

```

Threads and Coordinated Action

Part III

Monitor

- A software entity that has:
 - locks (i.e. supports synchronization)
 - wait sets – threads that can be blocked and woken up when some condition is true
- Java objects have:
 - synchronized methods
 - wait sets implemented via `wait()`, `notify()` and `notifyAll()` methods defined as part of class `Object`
- Therefore, every object in Java can serve as a monitor

Thread Coordination with Notify and Wait

Java Object as Monitor

- Each object has a lock that can be obtained and released
 - Java uses an object's synchronized methods to control thread access the object
- Each object also has a waiting area for threads to block wait until some other thread calls `notify()` or `notifyAll()`

`wait()`

- to call `wait()`, a thread must hold the object lock [via synchronized method or block] else exception will be thrown
- lock is released when `wait()` is called and thread is blocked -- put into waiting area for that object

notify()

- Called inside some object's method
- Thread that calls notify() must own lock on the object
- If any threads are in waiting area, ONE thread T is arbitrarily chosen to be unblocked
- T must now re-obtain the LOCK to continue which will always cause it to block until the thread that called notify() releases the lock
- When it gets the lock it will resume from the point where it called wait()

notifyAll()

- All threads waiting in wait area are resumed
- All must contend for the object lock