Threads and Concurrency

Lecture 22 – CS2110 – Fall 2011
Prelim Tonight!
Announcements

• **Prelim 2**
  - Tonight, November 15, 7:30-9pm
  - Uris Auditorium
  - Makeup TBA
  - Topics
    - all material up to and including Lecture 21 (Graphs III)

• **Old exams are available on the course website**
Data Structure Runtime Summary

- Stack [ops = put & get]
  - \( O(1) \) worst-case time
    - Array (but can overflow)
    - Linked list
  - \( O(1) \) time/operation
    - Array with doubling

- Queue [ops = put & get]
  - \( O(1) \) worst-case time
    - Array (but can overflow)
    - Linked list (need to keep track of both head & last)
  - \( O(1) \) time/operation
    - Array with doubling

- Priority Queue [ops = insert & getMin]
  - \( O(1) \) worst-case time if set of priorities is bounded
    - One queue for each priority
  - \( O(\log n) \) worst-case time
    - Heap (but can overflow)
  - \( O(\log n) \) time/operation
    - Heap (with doubling)
  - \( O(n) \) worst-case time
    - Unsorted linked list
    - Sorted linked list (\( O(1) \) for getMin)
    - Unsorted array (but can overflow)
    - Sorted array (\( O(1) \) for getMin, but can overflow)
Data Structure Runtime Summary (Cont’d)

• Set [ops = insert & remove & contains]
  - O(1) worst-case time
    - Bit-vector (can also do union and intersect in O(1) time)
  - O(1) expected time
    - Hash table (with doubling & chaining)
  - O(log n) worst-case time
    - Balanced BST
  - O(n) worst-case time
    - Linked list
    - Unsorted array
    - Sorted array (O(log n) for contains)

• Dictionary [ops = insert(k,v) & get(k) & remove(k)]
  - O(1) expected time
    - Hash table (with doubling & chaining)
  - O(log n) worst-case time
    - Balanced BST
  - O(log n) expected time
    - Unbalanced BST (if data is sufficiently random)
  - O(n) worst-case time
    - Linked list
    - Unsorted array
    - Sorted array (O(log n) for contains)
What is a Thread?

• A separate process that can perform a computational task independently and concurrently with other threads
  – Most programs have only one thread
  – GUIs have a separate thread, the event dispatch thread
  – A program can have many threads
  – You can create new threads in Java
What is a Thread?

• **In reality, threads are an illusion**
  – The processor shares its time among all the active threads
  – Implemented with support from underlying operating system or virtual machine
  – Gives the illusion of several threads running simultaneously
Concurrency (aka Multitasking)

• Refers to situations in which several threads are running simultaneously

• Special problems arise
  – race conditions
  – deadlock
• The operating system provides support for multitasking

• In reality there is one processor doing all this

• But this is an illusion too – at the hardware level, lots of multitasking
  – memory subsystem
  – video controller
  – buses
  – instruction prefetching
Threads in Java

• Threads are instances of the class Thread
  – you can create as many as you like

• The Java Virtual Machine permits multiple concurrent threads
  – initially only one thread (executes main)

• Threads have a priority
  – higher priority threads are executed preferentially
  – a newly created Thread has initial priority equal to the thread that created it (but can change)
Creating a new Thread (Method 1)

class PrimeThread extends Thread {
    long a, b;

    PrimeThread(long a, long b) {
        this.a = a; this.b = b;
    }

    public void run() {
        //compute primes between a and b
        ...
    }
}

PrimeThread p = new PrimeThread(143, 195);

overrides Thread.run()
can call run() directly – the calling thread will run it
or, can call start() – will run run() in new thread
Creating a new Thread (Method 2)

```java
class PrimeRun implements Runnable {
    long a, b;

    PrimeRun(long a, long b) {
        this.a = a; this.b = b;
    }

    public void run() {
        //compute primes between a and b
        ...
    }
}

PrimeRun p = new PrimeRun(143, 195);
new Thread(p).start();
```
public class ThreadTest extends Thread {

    public static void main(String[] args) {
        new ThreadTest().start();
        for (int i = 0; i < 10; i++) {
            System.out.format("%s %d\n",
                Thread.currentThread(), i);
        }
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.format("%s %d\n",
                Thread.currentThread(), i);
        }
    }
}
Example

```java
public class ThreadTest extends Thread {  

    public static void main(String[] args) {
        new ThreadTest().start();
        for (int i = 0; i < 10; i++) {
            System.out.format("%s %d\n", Thread.currentThread(), i);
        }
    }

    public void run() {
        currentThread().setPriority(4);
        for (int i = 0; i < 10; i++) {
            System.out.format("%s %d\n", Thread.currentThread(), i);
        }
    }
}
```
public class ThreadTest extends Thread {

    public static void main(String[] args) {
        new ThreadTest().start();
        for (int i = 0; i < 10; i++) {
            System.out.format("%s %d
", Thread.currentThread(), i);
        }
    }

    public void run() {
        currentThread().setPriority(6);
        for (int i = 0; i < 10; i++) {
            System.out.format("%s %d
", Thread.currentThread(), i);
        }
    }
}
```java
public class ThreadTest extends Thread {
    static boolean ok = true;

    public static void main(String[] args) {
        new ThreadTest().start();
        for (int i = 0; i < 10; i++) {
            System.out.println("waiting...");
            yield();
        }
        ok = false;
    }

    public void run() {
        while (ok) {
            System.out.println("running...");
            yield();
        }
        System.out.println("done");
    }
}
```

allows other waiting threads to run
Stopping Threads

• Threads normally terminate by returning from their run method

• `stop()`, `interrupt()`, `suspend()`, `destroy()`, etc. are all deprecated
  – can leave application in an inconsistent state
  – inherently unsafe
  – don't use them
  – instead, set a variable telling the thread to stop itself
Daemon and Normal Threads

• A thread can be *daemon* or *normal*
  – the initial thread (the one that runs `main`) is normal

• Daemon threads are used for minor or ephemeral tasks (e.g. timers, sounds)

• A thread is initially a daemon iff its creating thread is
  – but this can be changed

• The application halts when either
  – `System.exit(int)` is called, or
  – all normal (non-daemon) threads have terminated
Race Conditions

• A *race condition* can arise when two or more threads try to access data simultaneously

• Thread B may try to read some data while thread A is updating it
  – updating may not be an atomic operation
  – thread B may sneak in at the wrong time and read the data in an inconsistent state

• Results can be unpredictable!
Suppose threads A and B want to call `doSomething()`, and there is one element on the stack

1. thread A tests `stack.isEmpty()` ⇒ false
2. thread A pops ⇒ stack is now empty
3. thread B tests `stack.isEmpty()` ⇒ true
4. thread B just returns – nothing to do
Example – An Unlucky Scenario

```java
private Stack<String> stack = new Stack<String>();

public void doSomething() {
    if (stack.isEmpty()) return;
    String s = stack.pop();
    //do something with s...
}
```

Suppose threads A and B want to call `doSomething()`, and there is one element on the stack

1. thread A tests `stack.isEmpty()` ⇒ false
2. thread B tests `stack.isEmpty()` ⇒ false
3. thread A pops ⇒ stack is now empty
4. thread B pops ⇒ Exception!
private Stack<String> stack = new Stack<String>;

public void doSomething() {
    synchronized (stack) {
        if (stack.isEmpty()) return;
        String s = stack.pop();
    }
    //do something with s...
}

• Put critical operations in a synchronized block
• The stack object acts as a lock
• Only one thread can own the lock at a time
Solution – Locking

• You can lock on any object, including this

```java
public synchronized void doSomething() {
    ...
}
```

is equivalent to

```java
public void doSomething() {
    synchronized (this) {
        ...
    }
}
```
File Locking

• In file systems, if two or more processes could access a file simultaneously, this could result in data corruption
• A process must open a file to use it – gives exclusive access until it is closed
• This is called file locking – enforced by the operating system
• Same concept as synchronized(obj) in Java
Deadlock

• The downside of locking – *deadlock*

• A *deadlock* occurs when two or more competing threads are waiting for the other to relinquish a lock, so neither ever does

• Example:
  – thread A tries to open file X, then file Y
  – thread B tries to open file Y, then file X
  – A gets X, B gets Y
  – Each is waiting for the other forever
wait/notify

• A mechanism for event-driven activation of threads

• Animation threads and the GUI event-dispatching thread in can interact via wait/notify
wait/notify

`animator:`

```java
boolean isRunning = true;

public synchronized void run() {
    while (true) {
        while (isRunning) {
            //do one step of simulation
        }
        try {
            wait();
        } catch (InterruptedException ie) {} 
        isRunning = true;
    }
}
```

public void stopAnimation() {
    animator.isRunning = false;
}

public void restartAnimation() {
    synchronized(animator) {
        animator.notify();
    }
}