

# CS/ENGRD 2110

## Object-Oriented Programming and Data Structures

Fall 2012

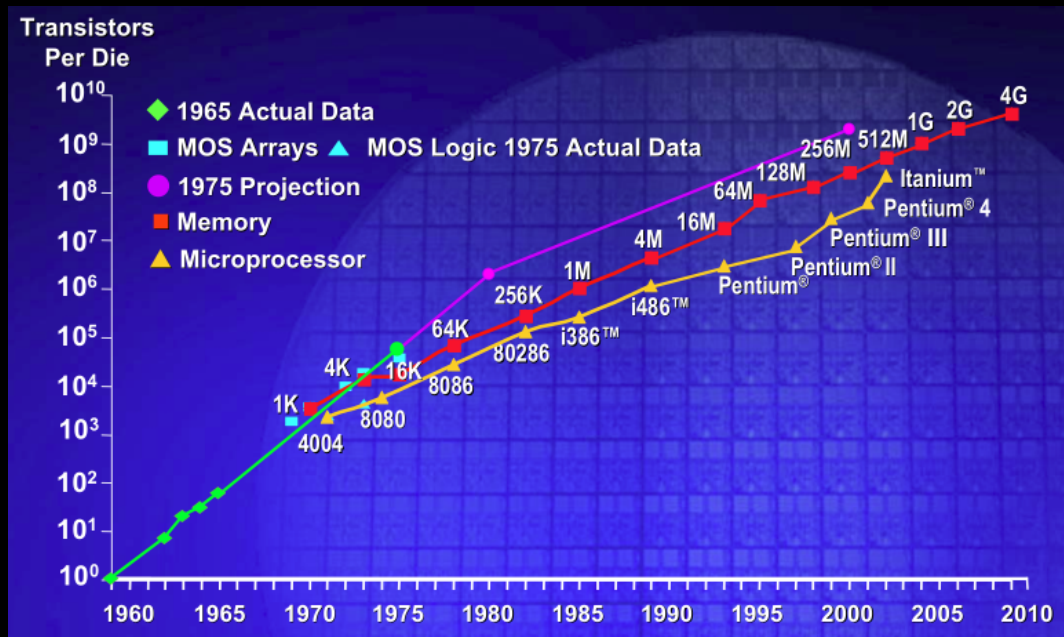
Doug James

### Lecture 21: Threads and Concurrency



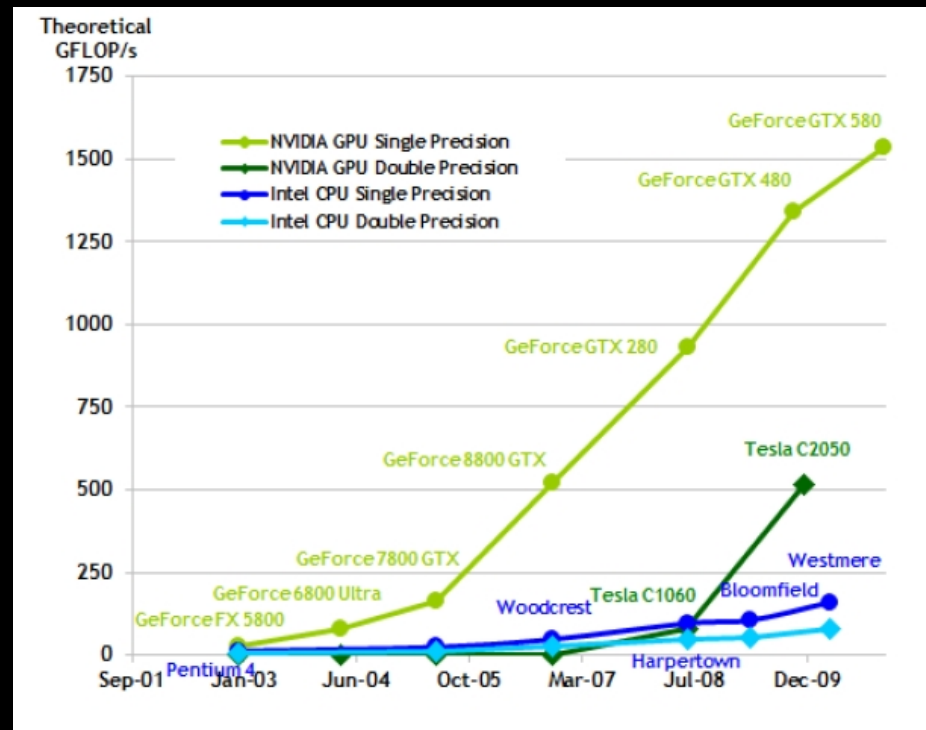
# Computer Processor Trends

- Moore's Law: Computer speeds and memory densities nearly double each year



- Multicore: use additional transistors to put more CPUs (cores) on one chip.

# Computer Processor Trends



- Graphics Processing Units (GPUs)
  - Highly parallel processors
  - Massively multi-threaded programs

# Concurrency (aka Multitasking)

- Multiple processes
  - Multiple independently running programs
- Multiple threads
  - Same program has multiple streams of execution
- Special problems arise
  - race conditions
  - deadlock

Image Name	User Name	CPU	Mem Usage
System Idle Process	SYSTEM	94	28 K
POWERPNT.EXE	tj	02	69,240 K
explorer.exe	tj	02	49,676 K
Skype.exe	tj	01	167,568 K
System	SYSTEM	01	252 K
taskmgr.exe	tj	00	5,784 K
HPBPRO.EXE	SYSTEM	00	2,324 K
HPBPRO.EXE	SYSTEM	00	2,524 K
searchprotocolhost.exe	tj	00	12,024 K
HPBPRO.EXE	tj	00	1,656 K
HPBPRO.EXE	SYSTEM	00	1,528 K
chrome.exe	tj	00	32,476 K
chrome.exe	tj	00	25,188 K
chrome.exe	tj	00	30,972 K
chrome.exe	tj	00	30,596 K
HPBOID.EXE	SYSTEM	00	1,440 K
OUTLOOK.EXE	tj	00	97,100 K
chrome.exe	tj	00	14,420 K
HPBOID.EXE	SYSTEM	00	1,372 K
chrome.exe	tj	00	14,108 K
chrome.exe	tj	00	18,312 K
chrome.exe	tj	00	35,928 K
searchfilterhost.exe	LOCAL SERVICE	00	17,108 K
HPBPRO.EXE	SYSTEM	00	1,536 K
cmd.exe	tj	00	144 K
chrome.exe	tj	00	22,536 K

Show processes from all users End Process

Processes: 116 CPU Usage: 7% Commit Charge: 1901M / 3

# What is a Thread?

- A separate stream of execution 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 dispatching thread
  - A program can have many threads
  - You can create new threads in Java

# What is a Thread?

- # Threads  $\neq$  # Processors  $\neq$  # Cores
  - The processor cores distribute their time over all the active threads
  - Implemented with support from underlying operating system or virtual machine
  - Gives the illusion of many threads running simultaneously, even if more threads than processors / cores

# Threads in Java

- Threads are instances of the class Thread
  - 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  
        ...  
    }  
}
```

overrides  
`Thread.run()`

can call `run()` directly –  
the calling thread will run it

```
PrimeThread p = new PrimeThread(143, 195);  
p.start();
```

or, can call `start()`  
– will run `run()` in new thread



## Creating a new Thread (Method 2)

```
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();
```

# Example

```
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);  
        }  
    }  
}
```

```
Thread[Thread-0,5,main] 0  
Thread[main,5,main] 0  
Thread[main,5,main] 1  
Thread[main,5,main] 2  
Thread[main,5,main] 3  
Thread[main,5,main] 4  
Thread[main,5,main] 5  
Thread[main,5,main] 6  
Thread[main,5,main] 7  
Thread[main,5,main] 8  
Thread[main,5,main] 9  
Thread[Thread-0,5,main] 1  
Thread[Thread-0,5,main] 2  
Thread[Thread-0,5,main] 3  
Thread[Thread-0,5,main] 4  
Thread[Thread-0,5,main] 5  
Thread[Thread-0,5,main] 6  
Thread[Thread-0,5,main] 7  
Thread[Thread-0,5,main] 8  
Thread[Thread-0,5,main] 9
```

# Example

```
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);  
        }  
    }  
}
```

```
Thread[main,5,main] 0  
Thread[main,5,main] 1  
Thread[main,5,main] 2  
Thread[main,5,main] 3  
Thread[main,5,main] 4  
Thread[main,5,main] 5  
Thread[main,5,main] 6  
Thread[main,5,main] 7  
Thread[main,5,main] 8  
Thread[main,5,main] 9  
Thread[Thread-0,4,main] 0  
Thread[Thread-0,4,main] 1  
Thread[Thread-0,4,main] 2  
Thread[Thread-0,4,main] 3  
Thread[Thread-0,4,main] 4  
Thread[Thread-0,4,main] 5  
Thread[Thread-0,4,main] 6  
Thread[Thread-0,4,main] 7  
Thread[Thread-0,4,main] 8  
Thread[Thread-0,4,main] 9
```

# Example

```
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(6);  
        for (int i = 0; i < 10; i++) {  
            System.out.format("%s %d\n",  
                Thread.currentThread(), i);  
        }  
    }  
}
```

```
Thread[main,5,main] 0  
Thread[main,5,main] 1  
Thread[main,5,main] 2  
Thread[main,5,main] 3  
Thread[main,5,main] 4  
Thread[main,5,main] 5  
Thread[Thread-0,6,main] 0  
Thread[Thread-0,6,main] 1  
Thread[Thread-0,6,main] 2  
Thread[Thread-0,6,main] 3  
Thread[Thread-0,6,main] 4  
Thread[Thread-0,6,main] 5  
Thread[Thread-0,6,main] 6  
Thread[Thread-0,6,main] 7  
Thread[Thread-0,6,main] 8  
Thread[Thread-0,6,main] 9  
Thread[main,5,main] 6  
Thread[main,5,main] 7  
Thread[main,5,main] 8  
Thread[main,5,main] 9
```

# Example

```
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

```
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
done
```

# 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 if its creating thread is
  - but this can be changed via `setDaemon(boolean on)`
- 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!



# Example – A Lucky Scenario

```
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()  $\Rightarrow$  false
  2. thread A pops  $\Rightarrow$  stack is now empty
  3. thread B tests stack.isEmpty()  $\Rightarrow$  true
  4. thread B just returns – nothing to do

# Example – An Unlucky Scenario

```
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()  $\Rightarrow$  false
  2. thread B tests stack.isEmpty()  $\Rightarrow$  false
  3. thread A pops  $\Rightarrow$  stack is now empty
  4. thread B pops  $\Rightarrow$  Exception!



# Solution: Locking

- A thread can “lock” an object for exclusive access
  - Only one thread can “hold” a lock at a time
  - If several request the same lock, Java somehow decides which will get it
- The lock is released when the thread leaves the synchronization block
  - `synchronized(someObject) { protected code }`
  - The protected code has a *mutual exclusion* guarantee:  
At most one thread can hold the lock at any time
- When released, some other thread can acquire the lock



# Locking in Java

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

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

**synchronized block**

- 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**

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

is equivalent to

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

# Locks are Associated with Objects

- Every Object has its own built-in lock
  - Just the same, some applications prefer to create special classes of objects to use just for locking
  - This is a stylistic decision and you should agree on it with your teammates or learn the company policy if you work at a company
- Code is “thread safe” if it can handle multiple threads using it... otherwise it is “unsafe”

# 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

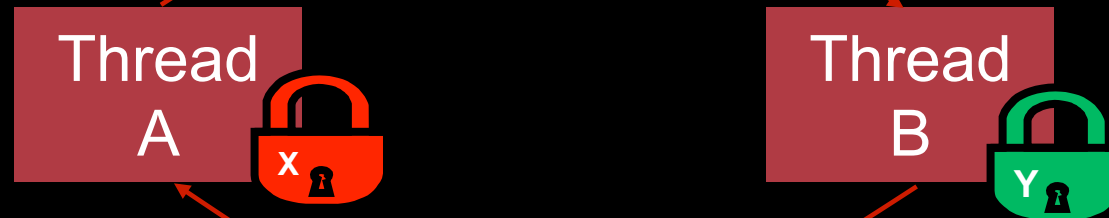
# Deadlock

- 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 lock object X, then object Y
  - thread B tries to lock object Y, then object X
  - A gets X, B gets Y
  - Each is waiting for the other forever



# Visualizing Deadlock

*A has a lock on X  
wants a lock on Y*



*B has a lock on Y  
wants a lock on X*

- Some Strategies for Avoiding Deadlocks
  - If possible, do not acquire more than one lock.
  - If possible, always lock objects in the same order.

# wait/notify

- A mechanism for event-driven activation of threads
  - For example, animation threads and the GUI event-dispatching thread in can interact via wait/notify
- How does it work?
  - A thread that has a lock on an object can call wait() to go to sleep and give up lock.
  - Other thread gets the lock, executes some code, and then calls notify()/notifyAll() to wake other thread
    - notify(): wakes up one of the sleeping threads for this object (roughly according to priority and sleep time)
    - notifyAll(): wakes up all sleeping thread in order (roughly)

# wait/notify

animator:

```
boolean isRunning = true;

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

relinquishes lock on animator –  
awaits notification

notifies processes waiting  
for animator lock

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

public void restartAnimation() {
    synchronized(animator) {
        // do stuff to animator
        animator.notify();
    }
}
```

# A producer/consumer example

- Thread A produces loaves of bread and puts them on a shelf with capacity  $K$ 
  - For example, maybe  $K=10$
- Thread B consumes the loaves by taking them off the shelf
  - Thread A doesn't want to overload the shelf
  - Thread B doesn't wait to leave with empty arms



# Producer/Consumer example

```
class Bakery {
    int nLoaves = 0;    // Current number of waiting loaves
    final int K = 10;  // Shelf capacity

    public synchronized void produce() {
        while(nLoaves == K) this.wait(); // Wait until not full
        ++nLoaves;
        this.notifyall();                // Signal: shelf not empty
    }

    public synchronized void consume() {
        while(nLoaves == 0) this.wait(); // Wait until not empty
        --nLoaves;
        this.notifyall();                // Signal: shelf not full
    }
}
```

# Things to notice

- Wait needs to wait on the same Object that you used for synchronizing (in our example, “this”, which is this instance of the Bakery)
- Notify wakes up just one waiting thread, notifyall wakes all of them up
- We used a while loop because we can’t predict exactly which thread will wake up “next”

# Summary

- Use of multiple processes and multiple threads within each process can exploit concurrency
  - Which may be real (multicore) or “virtual” (an illusion)
- But when using threads, beware!
  - Must lock (synchronize) any shared memory to avoid non-determinism and race conditions
  - Yet synchronization also creates risk of deadlocks
  - Even with proper locking concurrent programs can have other problems such as “livelock”
- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)
  - CS 3420, looks at why the hardware has this issue but not from the perspective of writing concurrent code