Today: Start a new topic

- Modern computers have “multiple cores”
  - Instead of a single CPU on the chip
  - 5-10 common. Intel has prototypes with 80!

- And even with a single core your program may have more than one thing “to do” at a time
  - Argues for having a way to do many things at once

- Finally, we often run many programs all at once

But a fast computer runs hot

- Power dissipation rises as square of the clock rate
- Chips were heading towards melting down!
- Multicore: with four CPUs (cores) on one chip, even if we run each at half speed we can perform more overall computations!

Why Multicore?

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

Challenge

- The operating system provides support for multiple “processes”
- In reality there are usually fewer processors than processes
- Processes are an abstraction: at hardware level, lots of multitasking
  - memory subsystem
  - video controller
  - buses
  - instruction prefetching
- Virtualization allows a single machine to behave like many...
What is a Thread?

- A separate “execution” that runs within a single program and can perform a computational task independently and concurrently with other threads
- Many applications do their work in just a single thread: the one that called main() at startup
  - But there may still be extra threads...
  - Garbage collection runs in a “background” thread
  - GUIs have a separate thread that listens for events and “dispatches” upcalls
- Today: learn to create new threads of our own

Concurrence

- Concurrency refers to a single program in which several threads are running simultaneously
  - Special problems arise
  - They see the same data and hence can interfere with each other, e.g. if one thread is modifying a complex structure like a heap while another is trying to read it
- In this course we will focus on two main issues:
  - Race conditions
  - Deadlock

Thread class in Java

- Threads are instances of the class Thread
  - Can create many, but they do consume space & time
  - The Java Virtual Machine creates the thread that executes your main method.
- Threads have a priority
  - Higher priority threads are executed preferentially
  - By default, newly created Threads have initial priority equal to the thread that created it (but can change)

Creating a new Thread (Method 1)

```java
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);
p.start();
```

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();
```
Terminating Threads is tricky

- Easily done... but only in certain ways
  - The safe way to terminate a thread is to have it return from its run method
  - If a thread throws an uncaught exception, whole program will be halted (but it can take a second or too...)  
  - There are some old APIs but they have issues: stop(), interrupt(), suspend(), destroy(), etc.
  - Issues: they can easily leave the application in a "broken" internal state.
  - Many applications have some kind of variable telling the thread to stop itself.

Threads can pause

- When active, a thread is "runnable".
  - It may not actually be "running". For that, a CPU must schedule it. Higher priority threads could run first.
- A thread can also pause
  - It can call Thread.sleep(k) to sleep for k milliseconds
  - If it tries to do "I/O" (e.g. read a file, wait for mouse input, even open a file) this can cause it to pause
  - Java has a form of locks associated with objects. When threads lock an object, one succeeds at a time.
Background (daemon) Threads

- In many applications we have a notion of "foreground" and "background" (daemon) threads
  - Foreground threads are the ones doing visible work, like interacting with the user or updating the display
  - Background threads do things like maintaining data structures (rebalancing trees, garbage collection, etc)

- On your computer, the same notion of background workers explains why so many things are always running in the task manager.

Race Conditions

- A “race condition” arises if two or more threads access the same variables or objects concurrently and at least one does updates
  - Example: Suppose t1 and t2 simultaneously execute the statement x = x + 1; for some static global x.
    - Internally, this involves loading x, adding 1, storing x
    - If t1 and t2 do this concurrently, we execute the statement twice, but x may only be incremented once
    - t1 and t2 “race” to do the update

Race Conditions

- Suppose x is initially 5

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD X</td>
<td>LOAD X</td>
</tr>
<tr>
<td>ADD 1</td>
<td>ADD 1</td>
</tr>
<tr>
<td>STORE X</td>
<td>STORE X</td>
</tr>
<tr>
<td>... after finishing, X=6! We “lost” an update</td>
<td></td>
</tr>
</tbody>
</table>

Example – A Lucky Scenario

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

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!
Synchronization

- Java has one “primary” tool for preventing these problems, and you must use it by carefully and explicitly – it isn’t automatic.
  - Called a “synchronization barrier”
  - We think of it as a kind of lock
    - Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
    - When it releases the lock, the next thread can acquire it
    - You can’t predict the order in which contending threads will get the lock but it should be “fair” if priorities are the same

Solution – with synchronization

```java
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() {
    ...
}
```

Behaves like

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

Synchronization+priorities

- Combining mundane features can get you in trouble
- Java has priorities... and synchronization
  - But they don’t “mix” nicely
  - High-priority runs before low priority
  - ... The lower priority thread “starves”
  - Even worse...
    - With many threads, you could have a second high priority thread stuck waiting on that starving low priority thread! Now both are starving...

Fancier forms of locking

- Java developers have created various synchronization ADTs
  - Semaphores: a kind of synchronized counter
  - Event-driven synchronization

- The Windows and Linux and Apple O/S all have kernel locking features, like file locking
- But for Java, `synchronized` is the core mechanism

Deadlock

- The downside of locking – deadlock
- A deadlock occurs when two or more competing threads are waiting for one-another... forever

Example:
- Thread t1 calls synchronized b inside synchronized a
- But thread t2 calls synchronized a inside synchronized b
- t1 waits for t2... and t2 waits for t1...
Finer grained synchronization

- Java allows you to do fancier synchronization
- But can only be used inside a synchronization block
- Special primatives called wait/notify

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)

wait/notify

Suppose we put this inside an object called animator:

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

- must be synchronized!
- relinquishes lock on animator – awaits notification
- notifies processes waiting for animator lock

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

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