THREADS & CONCURRENCY
A6 and A7

A6: get 100 for correctness (perhaps minus a late penalty), you can use your A6 in A7. Otherwise, use our solution.
See pinned Piazza note for A7.

Wed morn: A6 grades available. Got 100? No feedback. BUT: Graders are checking adherence to Style guidelines in A6 handout and deducting points here or there. So your grade may be lowered.

Start on A7 soon. Don’t wait till the last minute. Deadline: 3 May. Nothing accepted later. We have to grade quickly and determine tentative course letter grades, so you can decide whether to take the final.
The CPU is the part of the computer that executes instructions.

Java: \(x = x + 2;\)

Suppose variable \(x\) is at Memory location 800, Instructions at 10

**Machine language:**
10: load register 1, 800
11: Add register 1, 2
12: Store register 1, 800

Basic uniprocessor-CPU computer. Black lines indicate data flow, red lines indicate control flow
From wikipedia
>100 processes are competing for time. Here’s some of them:

<table>
<thead>
<tr>
<th>Process Name</th>
<th>% CPU</th>
<th>CPU Time</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grab</td>
<td>4.1</td>
<td>3.33</td>
<td>7</td>
</tr>
<tr>
<td>ReportCrash</td>
<td>2.3</td>
<td>0.78</td>
<td>6</td>
</tr>
<tr>
<td>Eclipse</td>
<td>1.5</td>
<td>1:48:30.07</td>
<td>54</td>
</tr>
<tr>
<td>SuperTab</td>
<td>1.4</td>
<td>1:40:44.59</td>
<td>5</td>
</tr>
<tr>
<td>Activity Monitor</td>
<td>1.4</td>
<td>10.57</td>
<td>10</td>
</tr>
<tr>
<td><a href="https://www.wunderground.c">https://www.wunderground.c</a>...</td>
<td>1.1</td>
<td>1:34.19</td>
<td>23</td>
</tr>
<tr>
<td>Creative Cloud</td>
<td>0.8</td>
<td>58:32.81</td>
<td>27</td>
</tr>
<tr>
<td>Microsoft PowerPoint</td>
<td>0.6</td>
<td>3:24.02</td>
<td>9</td>
</tr>
<tr>
<td>Safari Networking</td>
<td>0.4</td>
<td>26:53.25</td>
<td>10</td>
</tr>
<tr>
<td>loginwindow</td>
<td>0.3</td>
<td>16:14.79</td>
<td>4</td>
</tr>
<tr>
<td>Google Drive</td>
<td>0.3</td>
<td>6.33</td>
<td>22</td>
</tr>
<tr>
<td>Safari</td>
<td>0.3</td>
<td>50:09.48</td>
<td>24</td>
</tr>
</tbody>
</table>
Clock rate

- Clock rate “frequency at which CPU is running”
  Higher the clock rate, the faster instructions are executed.
- First CPUs: 5-10 Hz (cycles per second)
- Now: MacBook Pro 3.5 GHz
- Your OS can control clock rate, slow it down when idle, speed up when more work to do
Why multicore?

- Moore’s Law: Computer speeds and memory densities nearly double each year
But a fast computer runs hot

- Power dissipation rises as square of the clock rate
- Chips were heading toward melting down!
- Put more CPUs on a chip: with four CPUs on one chip, even if we run each at half speed we can perform more overall computations!
Today: Not one CPU but many

Processing Unit is called a core.

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

- We often run many programs/applications at the same time

- Even with a single core (processing unit), your program may have more than one thing “to do” at a time
  - Argues for having a way to do many things at once
Many programs. Each can have several “threads of execution”

We often run many programs at the same time
And each program may have several “threads of execution”

Example, in GUI paint program, when you click the pencil tool, a new thread of execution is started to call the method to process it:

Main GUI thread  Process pencil click
Many processes are executed simultaneously on your computer.

- Operating system provides support for multiple “processes”
- Usually fewer processors than processes
- Processes are an abstraction: at hardware level, lots of multitasking
  - memory subsystem
  - video controller
  - buses
  - instruction prefetching
Concurrency

Concurrency refers to a single program in which several processes, called threads, are running simultaneously.

- Special problems arise.
- They see the same data and hence can interfere with each other, e.g. one process modifies a complex structure like a heap while another is trying to read it.

CS2110: we focus on two main issues:

- Race conditions
- Deadlock
A “race condition” arises if two or more processes access the same variables or objects concurrently and at least one does updates.

Example: Processes t1 and t2

\[ x = x + 1; \]

for some static global x.

Process t1

...  

\[ x = x + 1; \]

Process t2

...  

\[ x = x + 1; \]

But \( x = x + 1; \) is not an “atomic action”: it takes several steps.
Race conditions

Suppose $x$ is initially 5

Thread $t_1$
- LOAD $x$ (register contains 5)
- ADD 1 (register contains 6)
- STORE $x$ ($x$ contains 6)

Thread $t_2$
- ...
- LOAD $x$ (register contains 5)
- ADD 1 (register contains 6)
- STORE $x$ ($x$ contains 6)

... after finishing, $x = 6$! We “lost” an update
Race conditions

- Typical race condition: two processes wanting to change a stack at the same time. Or make conflicting changes to a database at the same time.

- Race conditions are bad news
  - Race conditions can cause many kinds of bugs, not just the example we see here!
  - Common cause for “blue screens”: null pointer exceptions, damaged data structures
  - Concurrency makes proving programs correct much harder!
Deadlock

- To prevent race conditions, one often requires a process to “acquire” resources before accessing them, and only one process can “acquire” a given resource at a time.

- Examples of resources are:
  - A file to be read
  - An object that maintains a stack, a linked list, a hash table, etc.

- But if processes have to acquire two or more resources at the same time in order to do their work, deadlock can occur. This is the subject of the next slides.
Dining philosopher problem

Five philosophers sitting at a table.

Each repeatedly does this:
1. think
2. eat

What do they eat?
spaghetti.

Need TWO forks to eat spaghetti!
Dining philosopher problem

Each does repeatedly:
1. think
2. eat (2 forks)

Eat is then:
- pick up left fork
- pick up right fork
- pick up food, eat
- put down left fork
- put down right fork

At one point, they all pick up their left forks

DEADLOCK!
Dining philosopher problem

Simple solution to deadlock:
Number the forks. Pick up smaller one first
1. think
2. eat (2 forks)
eat is then:
pick up smaller fork
pick up bigger fork
pick up food, eat
put down bigger fork
put down smaller fork
Java: 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” calls to methods to process them.

- Today: learn to create new threads of our own in Java.
A thread is an object that “independently computes”

- Needs to be created, like any object
- Then “started” --causes some method to be called. It runs side by side with other threads in the same program; they see the same global data

- The actual executions could occur on different CPU cores, but but don’t have to

- We can also simulate threads by *multiplexing* a smaller number of cores over a larger number of threads
Java class Thread

- threads are instances of 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 priority can be changed)
Creating a new Thread (Method 1)

class PrimeThread extends Thread {
  long a, b;

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

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

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

Call run() directly? no new thread is used: Calling thread will run it.
overrides Thread.run()
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();
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 name, priority, thread group:
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
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() {
        Thread.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\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);
        }
    }
}
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");
    }
}
Terminating Threads is tricky

- Easily done... but only in certain ways
  - Safe way to terminate a thread: return from method run
  - Thread throws uncaught exception? whole program will be halted (but it can take a second or two ... )
- Some old APIs have issues: stop(), interrupt(), suspend(), destroy(), etc.
  - Issue: Can easily leave 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 pause
  - Call Thread.sleep(k) to sleep for k milliseconds
  - Doing I/O (e.g. read file, wait for mouse input, open file) can cause thread 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 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.
Example: a lucky 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 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!
Java has one primary tool for preventing race conditions. you must use it by carefully and explicitly – it isn’t automatic.

- Called a synchronization barrier
- We study this in the next lecture