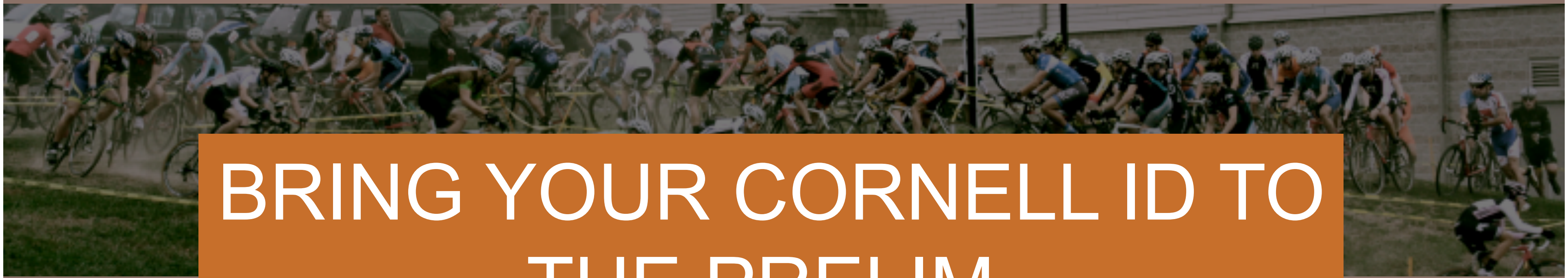


BRING YOUR CORNELL ID TO THE PRELIM.

You need it to get in

THREADS & CONCURRENCY

Lecture 23– CS2110 – Spring 2016

Announcements

- Prelim 2 is next Tonight BRING YOUR CORNELL ID!
- A7 is due Thursday. Our Heap.java: on Piazza (A7 FAQs)
- A8. Firm deadline, no late ones permitted. We have to get it graded and get letter grades calculated immediately so you can determine whether to take the final. Final: 17 May
- For A8. Get the simplest solution correct on both parts before attempting to improve!
- Cheating cases found and we are processing them. Please don't cheat!
- Two versions of dfs (different specs) on Piazza Supplementary material note

Today: New topic: concurrency

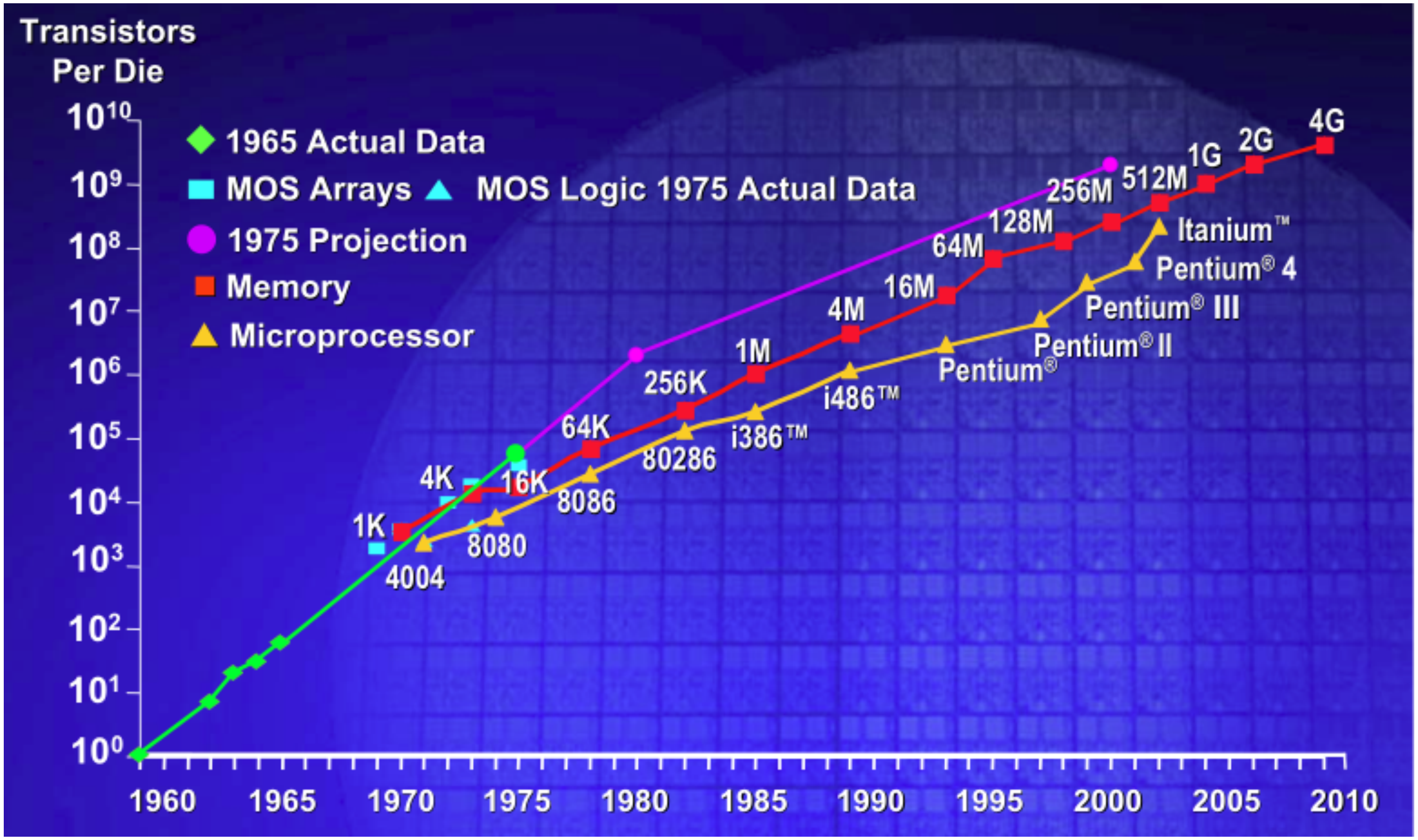
- Modern computers have “multiple cores”
  - Instead of a single CPU (central processing unit) on the chip 5-10 common. Intel has prototypes with 80!

- We often run many programs at the same time

- 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

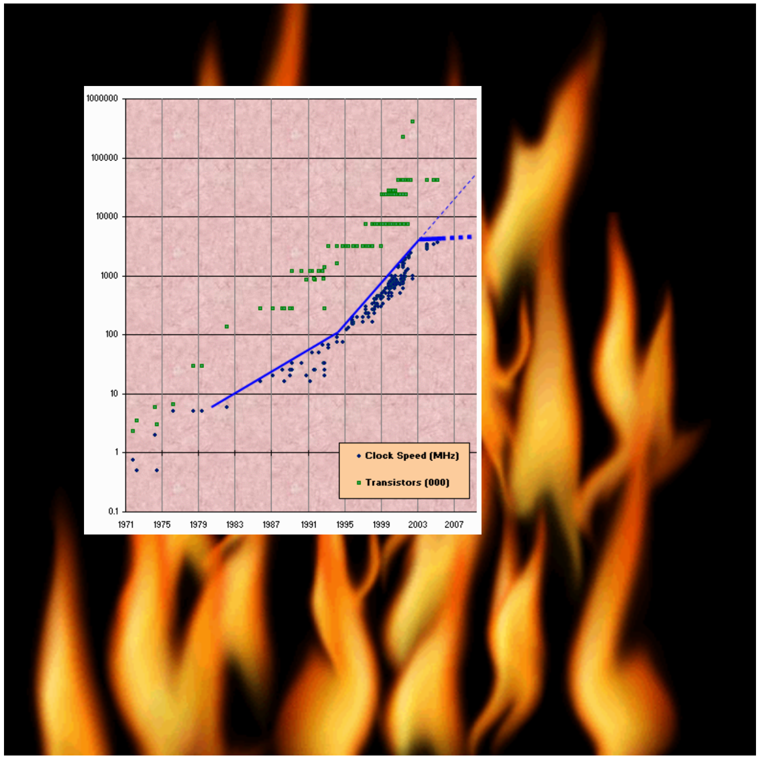
Why multicore?

- 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!
- Multicore: with four CPUs (cores) on one chip, even if we run each at half speed we can perform more overall computations!



Programming a Cluster...

- Sometimes you want to write a program that is executed on many machines!
- Atlas Cluster (at Cornell):
- 768 cores
- 1536 GB RAM
- 24 TB Storage
- 96 NICs (Network Interface Controller)

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

Part of Activity Monitor in Gries's laptop

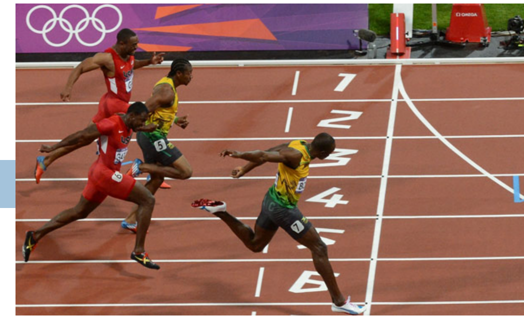
>100 processes are competing for time. Here's some of them:

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

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

Race conditions



- 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 | Process t2
...
x= x + 1;

Process t1	Process t2
...	...
x= x + 1;	x= x + 1;

But x= x+1; is not an "atomic action": it takes several steps

Race conditions

- Suspose x is initially 5

Thread t1	Thread t2
LOAD x	...
	LOAD x
ADD 1	ADD 1
	STORE x
STORE x	

- ... after finishing, x = 6! We "lost" an update

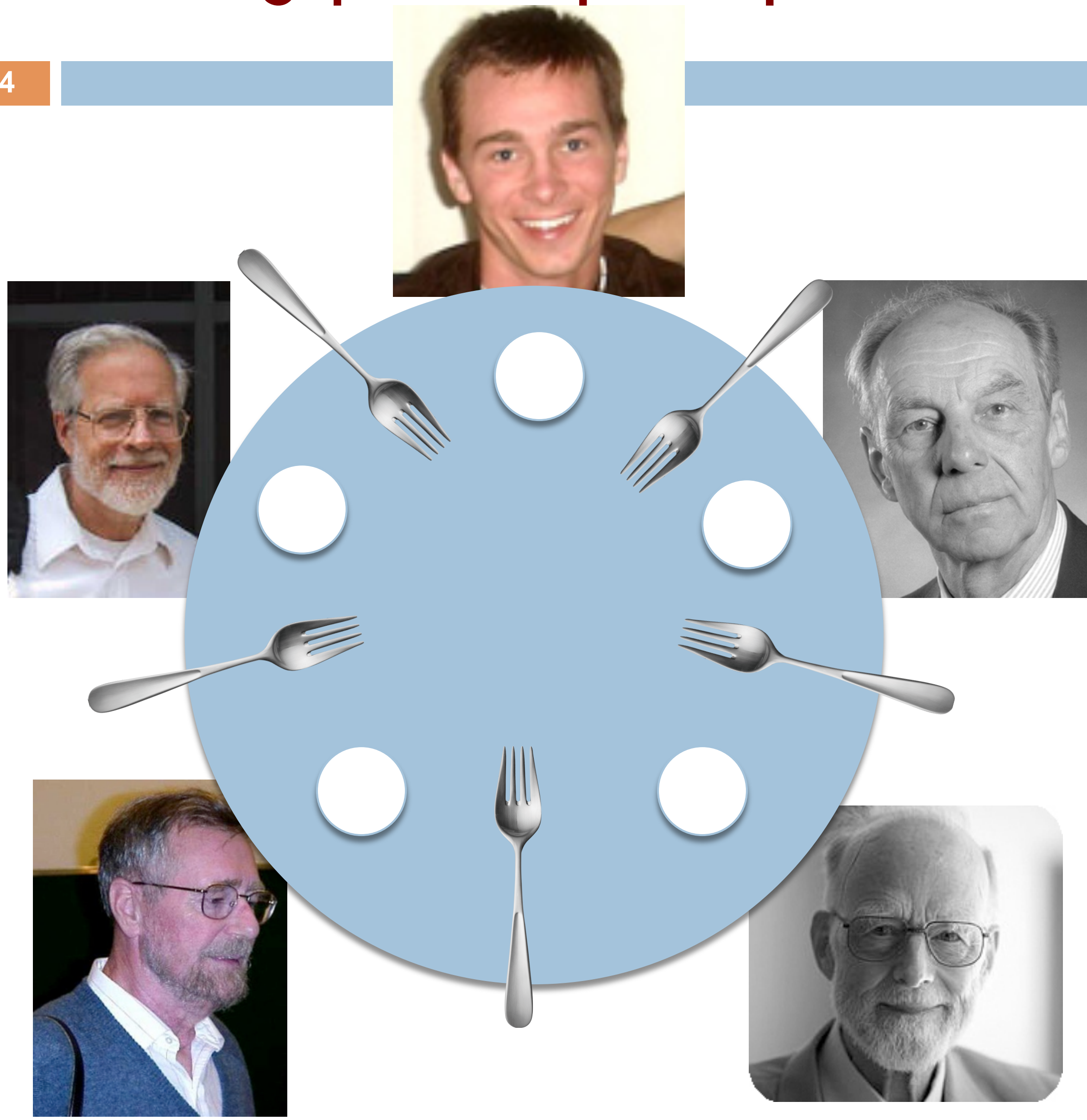
Race conditions

- 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.

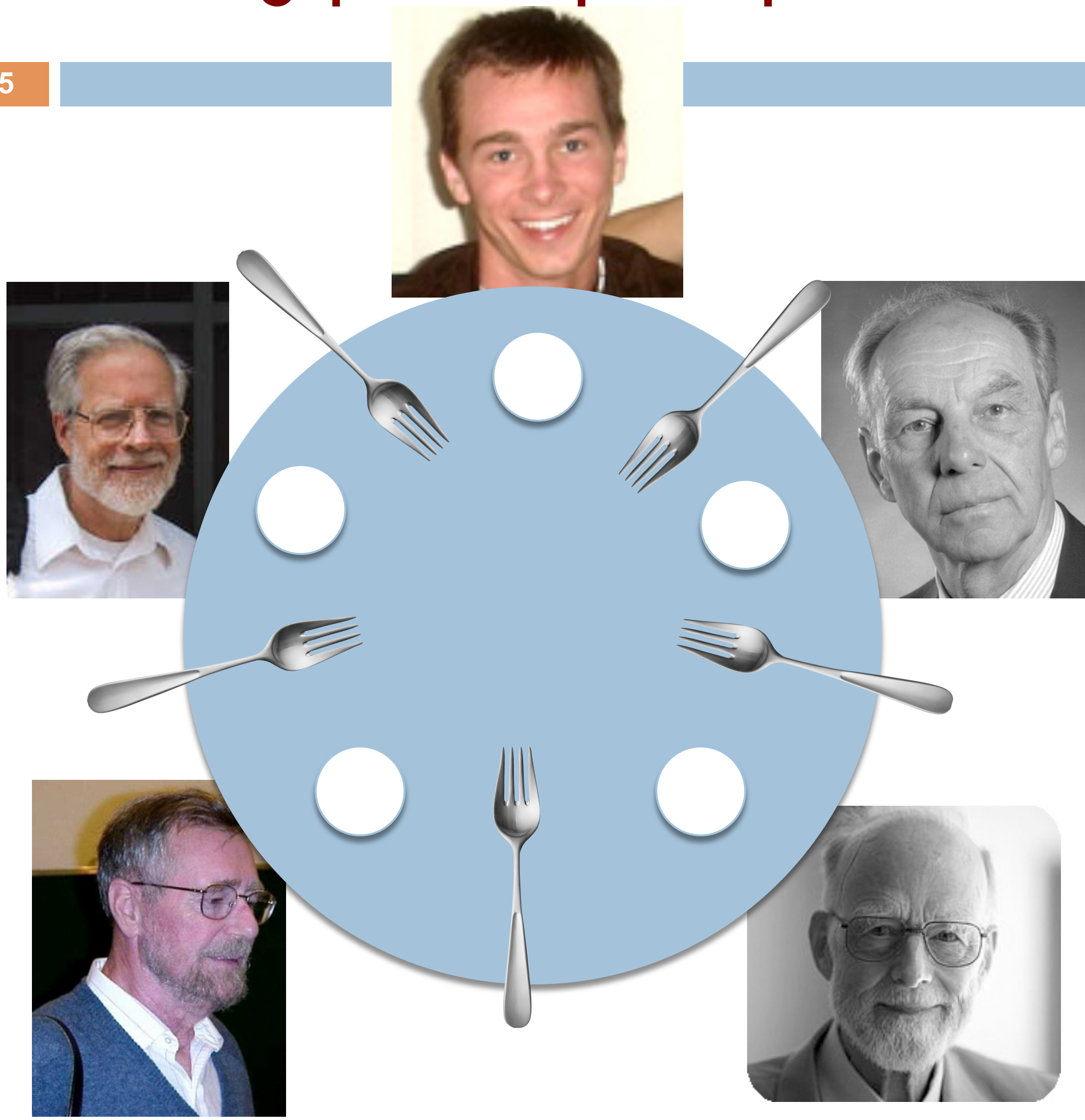
Each repeatedly does this:

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:

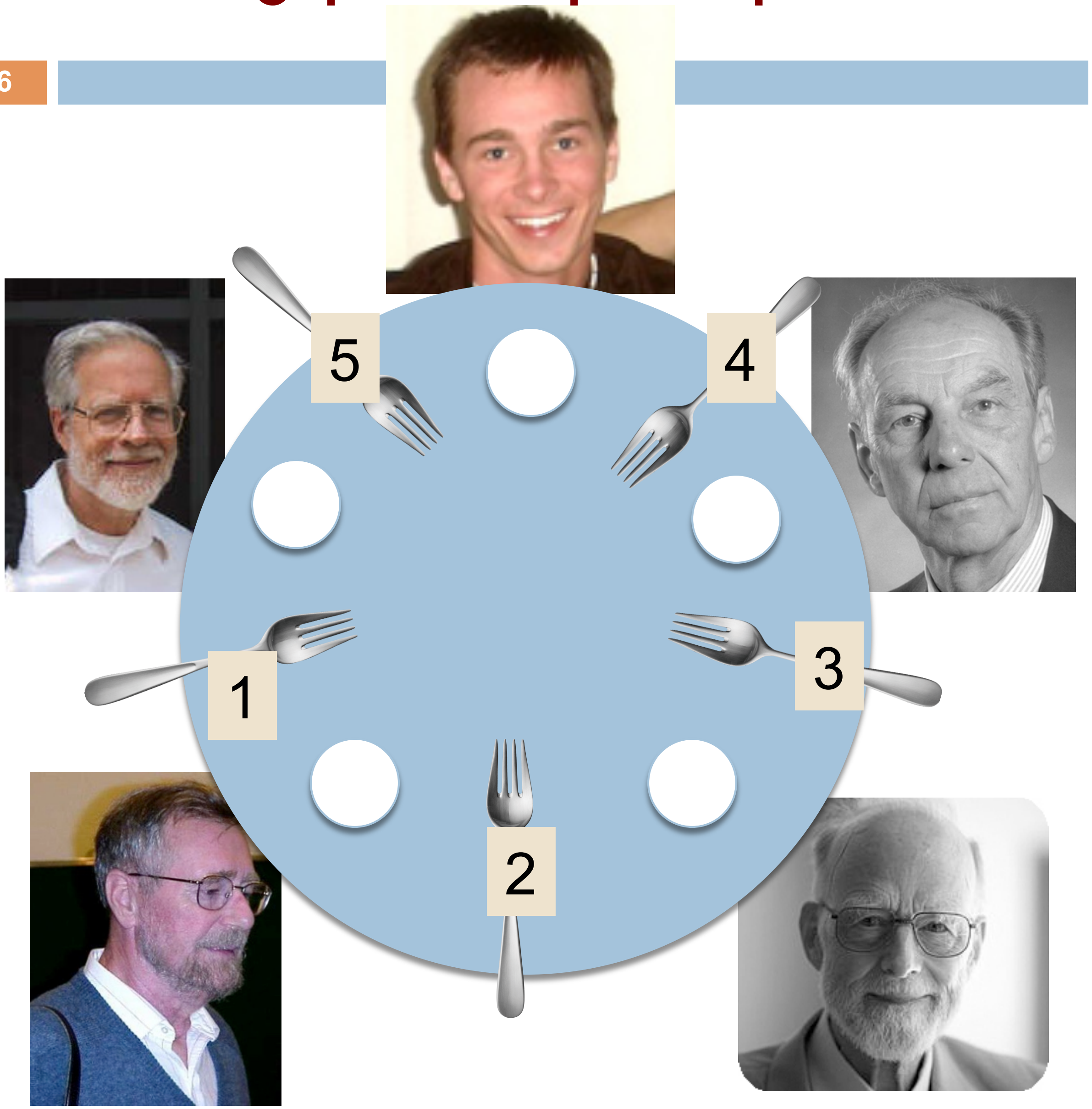
pick up left fork
- pick up right fork

pick up food, eat
put down left fork
put down rght 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 smallerfork

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

Thread

- 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;

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

overrides
Thread.run()

Call **run()** directly?
no new thread is used:
Calling thread will run it

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

Do this and
Java invokes **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

Thread name, priority, thread group

```
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

Thread name, priority, thread group

```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);
      }
   }
}
```

```
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

Thread name, priority, thread group

```
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");
    }
}
```

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

If threads happen to be sharing a CPU, yield allows other waiting threads to run.

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


```
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!

Synchronization

- 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
  - 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, another thread can acquire it
    - Can't predict the order in which contending threads get the lock but it should be "fair" if priorities are the same

Solution: use with synchronization


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

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

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

Syntactic sugar for the above:

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

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 abstract data types
  - Semaphores: a kind of synchronized counter (invented by Dijkstra)
  - Event-driven synchronization

- The Windows and Linux and Apple O/S have kernel locking features, like file locking

- But for Java, **synchronized** is the core mechanism

Finer grained synchronization

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

wait/notify

Suppose we put this inside an object called animator:

```
boolean isRunning = true;

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

must be synchronized!

relinquishes lock on **animator** – awaits notification

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

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

notifies processes waiting for **animator** lock

Summary

- Use of multiple processes and multiple threads within each process can exploit concurrency
  - Which may be real (multicore) or "virtual" (an illusion)
- When using threads, beware!
  - Synchronize any shared memory to avoid race conditions
  - Synchronize objects in certain order to avoid deadlocks
  - Even with proper synchronization, 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)