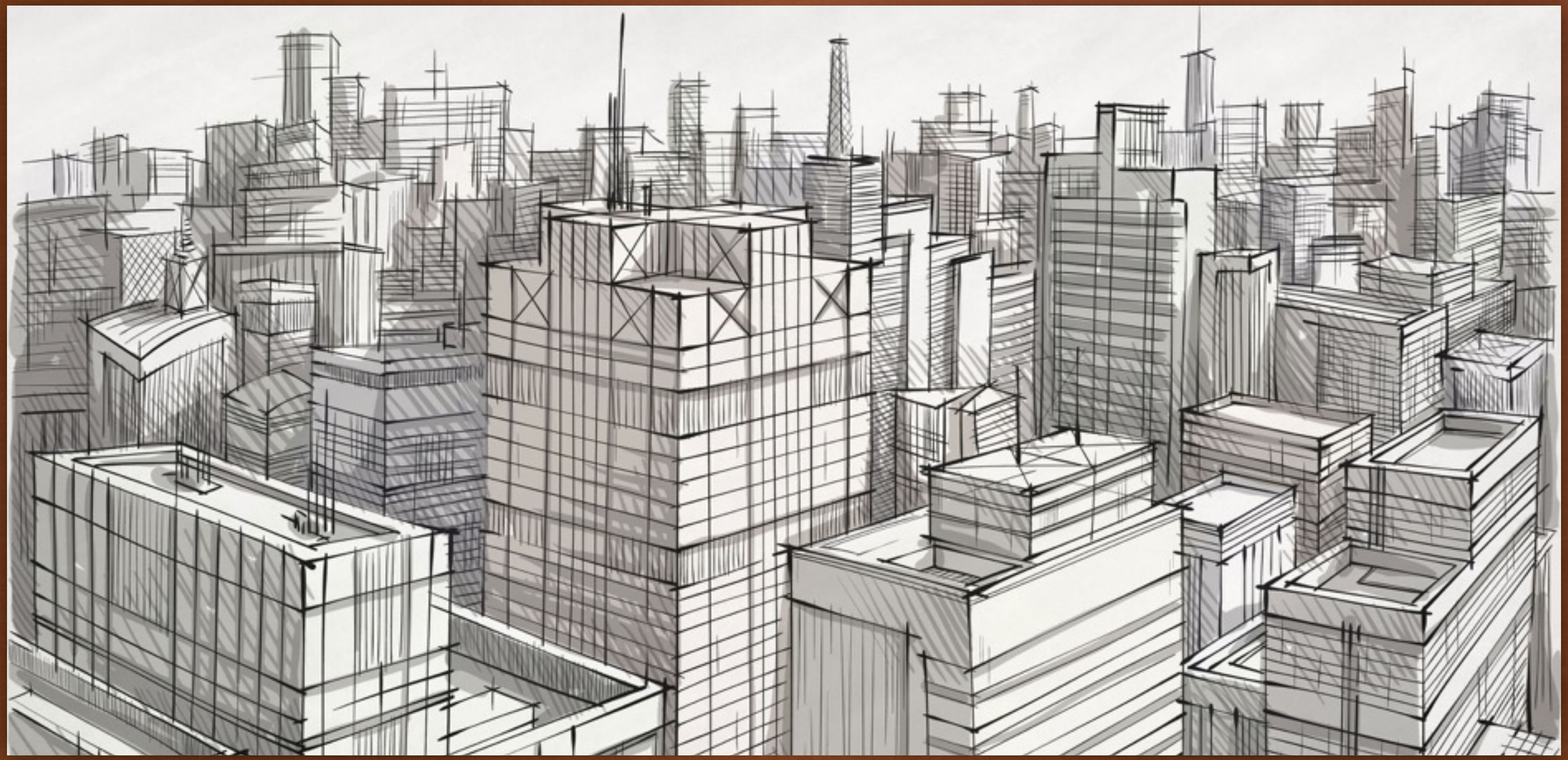
CS1530, LECTURE 15: CONCURRENCY AND MULTI-THREADED PROGRAMMING

BILL LABOON



OLD COMPUTERS WERE NOT SO GREAT AT MULTI-TASKING

- One program could run at a time
- Usually your programming job would run in “batches” - that is, you’d hand the computer operator your punch card deck, the operator would run yours until completion, then someone else’s until completion, then a third person’s, etc.
- As computers became more powerful, and more centralized, “multi-tasking” became common - multiple people could log in and interact with the system at “the same time”

...IN WHICH WE REALIZE THAT THE LAST LINE ON THE LAST SLIDE WAS A TOTAL LIE

- The processor could still only do one thing at a time - single core

- Who had "control" of the processor would rotate rapidly among all existing jobs, creating the ILLUSION of people each job having access to the computer at the

Image courtesy of the Arrested Development wiki
http://arresteddevelopment.wikia.com/wiki/G.O.B.

NOWADAYS, WE DON'T HAVE TO RELY ON ILLUSIONS

- Many processors today are multi-core
- Most in a traditional CPU I could find was
- GPUs have thousands of cores, running truly in parallel
- This is the path of the future - we are hitting fundamental limits on processor speed
    - One CPU cycle on a 3.0 GHz processor takes 1/3rd of a nanosecond
    - In this time, light travels about 3.9 *inches*

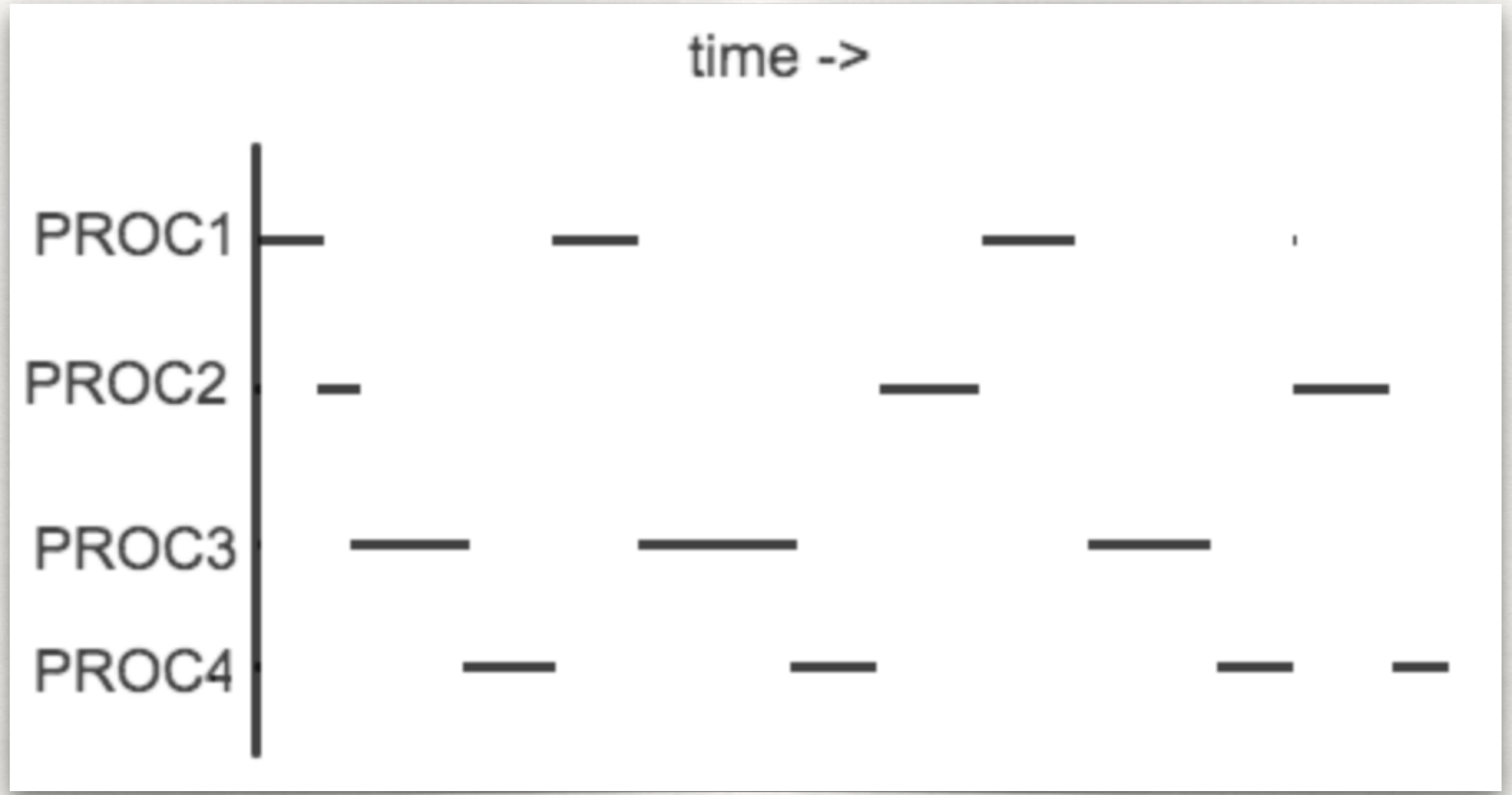
CONCURRENCY VS PARALLELISM

DONEC QUIS NUNC

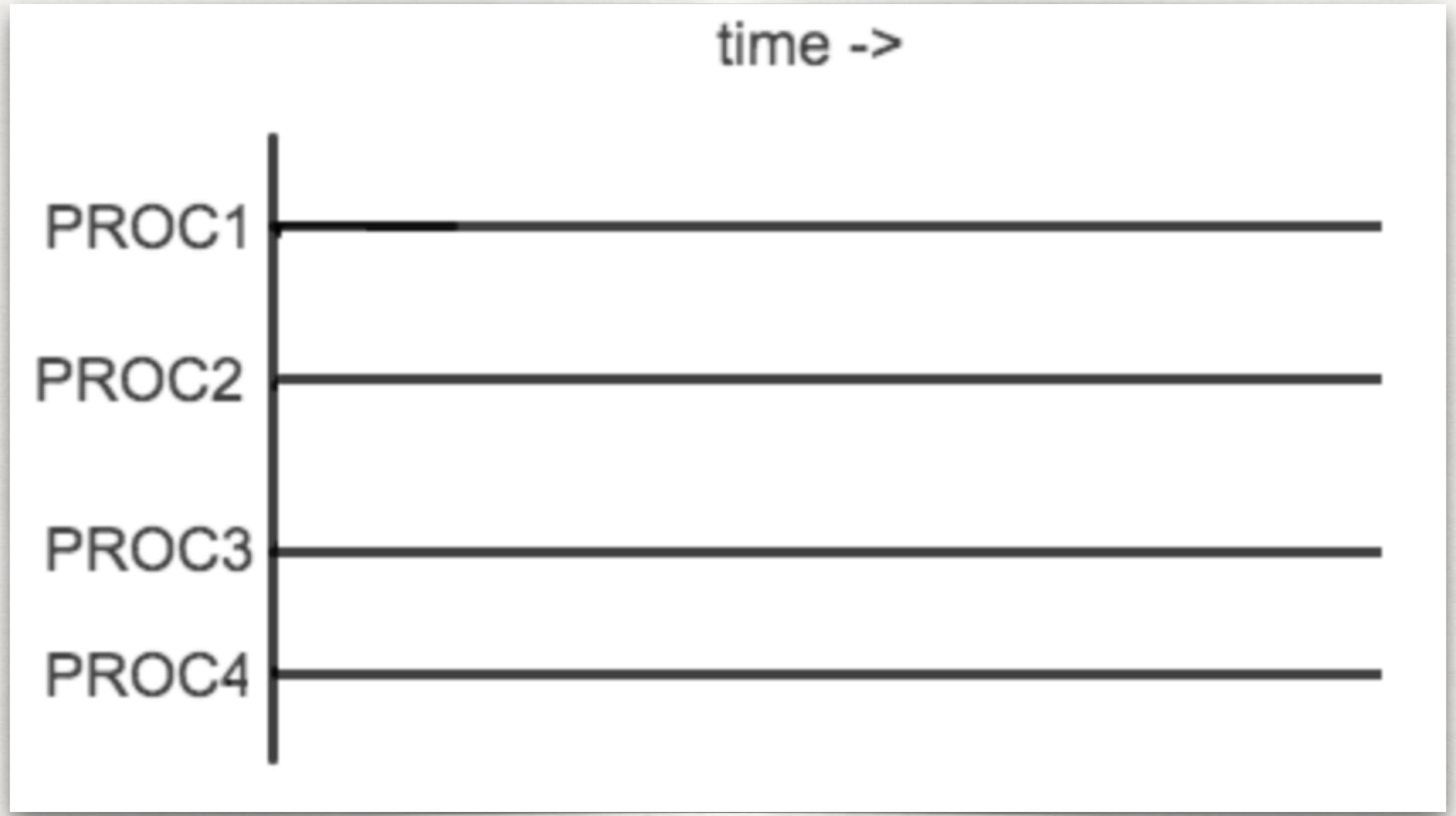
- Concurrency - the order of execution of operations may be changed, and other operations may take place in between proximal code

- Parallelism - multiple operations are occurring at the literal EXACT SAME TIME (not just “pretend” same time)

CONCURRENCY



PARALLELISM



TAKING ADVANTAGE OF CONCURRENCY / PARALLELISM: PROCESSES AND THREADS

- You are probably familiar with processes - they are different "programs". They share information with each only through well-defined interfaces (e.g., STDOUT, STDIN, STDERR)

- If you've taken 449 and/or 1550, or you do research on things other than just what you learn in class, you probably also know about threads (threads of execution)

    - Threads are "mini-processes" inside a single operating system process

    - They are faster (more lightweight) than processes but also share memory

WHY USE THREADS?

- Speed
- Theoretically, you could do anything you can do with a thread via processes - but would be much slower and perhaps more complex
- Let’s take a look at examples -
- SynchronousCalculator.java vs ConcurrentCalculator.java

THREADS ON THE JVM

- All processes have at least one thread - it is the current thread of execution
- A “single-threaded” program is relatively simple to deal with - you don’t have to worry about
- How many threads do you think are running in the following program:
  - HowManyThreads.java

THREADS ON THE JVM

- If you are writing Java code, you are automatically dealing with a multi-threaded environment even if you don’t realize it!

BASIC JAVA THREADING USAGE - CREATING A THREAD

- Look at SimpleThread.java
- First we need to create a Thread object in Java. This accepts a Runnable as a parameter.

```
Thread t = new Thread(SimpleThread::backgroundTask);
```

RUNNABLE

- Can be a:
  - Runnable object
  - Reference to a method
  - Lambda

RUNNABLE OBJECT

```
// Runnable interface only requires a “run” method

public class Foo implements Runnable {

    public void run(){
        while (true) {
            System.out.println(“foo!");
        }
    }
}
```

REFERENCE TO A METHOD

```
Thread t = new Thread(SimpleThread::backgroundTask);
```

LAMBDA

```
// New in Java 8 - think of lambdas as “anonymous
functions”
// This syntax is called the “stabby lambda”

Thread t = new Thread(() -> {
   while (true) {
      System.out.println(“bar");
   }
});
```

STARTING A THREAD

- Once you create a thread, it just sitting out there
- It does NOT start executing until you start it

STARTING A THREAD

```
// Create new thread
Thread t = new Thread(() -> {
    while (true) {
        System.out.println(“bar");
    }
});

// Thread is just sitting out there waiting now

t.start();

// Now another thread has started - it is executing
// and printing out “bar” to STDOUT now
```

RUN VS START

- Run will just run the code in the “run” method!
- 99% of the time, not what you want to do - it will run the method in the current thread of execution

JOINING

- Threads are concurrent - we don't know what order they will be executed in
- So we don't know how long threads will take to do their work, even if we know exactly how long it will take on the processor
- Sometimes we want to wait until a thread is done doing its work before continuing work in another thread
- This is called joining

JOINING

DONEC QUIS NUNC

```
Thread t = new Thread(() -> {
    for (int j = 0; j < 1000000; j++)
        y++;
    }
});
// Start off thread t
t1.start();
// Execution continues on main thread
System.out.println("la la la");
try {
    // main thread will NOT continue until t is done
    t1.join();
} catch (InterruptedException iex) { }

System.out.println("y = " + y);
```

BENEFITS OF THREADS

DONEC QUIS NUNC

- Speed - threads quick to create, quick to destroy
- Allow parallelism and concurrency inside of a process (thus allowing things like GUIs and event-driven programming)
- Efficiency - take advantage of your cores
- Scalability - can distribute amongst more powerful hardware or even different servers / computer systems

PROBLEMS WITH THREADS

- DANGER - have to always worry about some other thread of execution messing up your data WHILE you are dealing with it!
- Data races
- Code will be harder to understand
- Problems may not occur all the time (non-deterministic / intermittent failures) - the hardest kinds of defects to track down
- You may not even REALIZE there's a problem!
- See BadConcurrency.java

DATA RACES

DONEC QUIS NUNC

- Multiple threads modifying the same variable
- Final result depends on ordering of execution of threads (which is controlled by the OS or the process, so for all practical purposes, non-deterministic)
- See BadConcurrency.java

WAYS TO AVOID - MINIMIZING SHARED IMMUTABLE STATE

DONEC QUIS NUNC

- Ensure that any data the thread gets is not shared in a mutable way
- That is, any data the thread gets cannot be modified by some other thread

WAYS TO AVOID - SYNCHRONIZED

- Ensure that certain sections of code are atomic (cannot be interrupted) and that multiple threads cannot access these synchronized blocks at the same time
- Means that multiple threads cannot cause data races
- You still have to deal with logic errors, though!

SYNCHRONIZE ON OBJECTS

- An object “holds” a lock, and only one thread can access it as a time
- See BetterConcurrency.java

SYNCHRONIZED METHODS

- Synchronize on *this*, i.e. the current object - synchronize entire method
- (almost) syntactic sugar on top of synchronized keyword

```
public synchronized void incrementX() {
   x++;
}
==
public void incrementX() {
   synchronized(this) {
      x++;
   }
}
```

DEADLOCK

- Program stops performing work - multiple threads prevent each other from doing anything because they are both waiting for the same or related resources

- Example: I am eating with a friend. We both need a fork and a knife to eat. I grab a fork and my friend grabs a knife. My algorithm is to hold onto whatever I have and wait for the other utensil.

- I hold on to my fork, waiting for my friend to be done with the knife. My friend holds on to the knife, waiting for me to be done with the fork.

- Eventually we both die of starvation.

DEADLOCK EXAMPLE

- Deadlock.java

DEADLOCK - WAYS TO AVOID

- Always perform your locks in a certain, specified order
- Give up after a while - hopefully random (otherwise you may get into a livelock situation - detailed later)
- Analysis of the program

LIVELOCK

- Like deadlock - no useful forward progress is made in program execution - but there is movement, it's just not useful
- For example, two people going down a narrow hallway. It's wide enough to fit two people if they each lean against different sides.
- Both people go to one side, realize the other one is going to that side, so they both go to the other side, realize the other one is going to that side, etc. Repeate ad infinitum
- Lots of movement but no useful forward progress

LIVELOCK EXAMPLE

- Livelock.java

LIVELOCK - WAYS TO AVOID

- Randomization of responses
- External monitors

MULTI-THREADED PROGRAMMING IS HARD

- I, for one, welcome our new multi-core overlords… it's the only way forward
- It's work, though! Humans still have to figure this stuff out.
- We've abstracted away lots of internal details of how a program works (very few non-OS programmers worry about swapping memory to disk, but this was a huge issue years ago) but not this one
- The "sufficiently smart compiler" has been promising to take us into this land of milk and honey for a long time now

HOWEVER, THERE ARE WAYS FORWARD!

- Some languages have tried to work around it
  - Ruby - GIL (Global Interpreter Lock) specifically disallows parallel execution
  - Erlang - Lots of very efficient, small pseudo-processes (no shared mem) running in one system process
  - Rust - Ownership/Borrowing
  - Haskell / Clojure - Make as much state as you can immutable
  - Elixir - Make as much state as you can immutable PLUS lots of very efficient, small pseudo-processes à la Erlang

BUT THIS CLASS IS TAUGHT IN JAVA =(

- Well, let's allow some other smart people to do the work for us
  - Always a good idea when developing software
  - JSR 166 and 133
- Next class: working with threads at a higher level