# Threads

ICS332

**Operating Systems**

Henri Casanova (henric@hawaii.edu)

# Concurrent Programming

- **Concurrency**: the execution of multiple “tasks” at the “same” time
- College students mostly write non-concurrent, or sequential, programs
  - At any point, you could stop the program and say exactly which execution is being executed, what the calling sequence is, what the runtime stack looks like, etc.
  - And there is a single answer to all the above for all execution of your program at the same point in its execution
- In a concurrent program, you design the program in terms of tasks, where each task as a “life of its own”
  - Each task has a specific job to do
  - Tasks may need to “talk” to each other or “wait” for each other
  - Tasks can be in different regions of the code or in the same region of the code a the same time
- A different way of thinking/programming

# Example #1: Make it Fast

- Consider an input array of 10,000 integers: { 23, 56, 7, 68, 68 ...}
- I want to output a boolean array where each element is true if and only if the corresponding integer in the input array is odd
  { true, false, true, false, false ...}
- Assume that it takes one millisecond to test an integer value and update the output array
- **Sequential programming:**
  - Iterate through the array, which would take 10,000 milliseconds.
- **Concurrent programming:**
  - If I create 10 “tasks” that each compute 1000 output values, i.e., 1/10-th of the work, each task takes 1,000 milliseconds
  - Now if I can execute these 10 tasks independently (on a 10-core processor), the whole execution could take only 1,000 milliseconds, i.e., 10 times faster
  - In practice, we can’t go quite 10 times faster due to various overheads and bottlenecks (e.g., memory)
  - But we will go much faster than sequential, provided be have multiple cores (which we all do in this day and age!)

# Example #2: Make it Responsive

- Consider a Photoshop-like app in which a click of a button launches a transformation filter of all images that a user has selected
  - If many images are selected, this can take minutes
- **Sequential programming:**
  - While the transformation is happening, no other code can run, including the code that reacts to button clicks, meaning that the application is “frozen”, including whatever “Cancel” button one may have tried to implement
  - One solution, which is terrible, is to sprinkle “check whether the button is being clicked” code all over the code that performs the transformation
  - And it may not be feasible if that code is, for instance, a 3rd-party library
- **Concurrent programming:**
  - Create a “task” in charge of watching buttons and reacting to clicks, which runs all the time
  - Whenever the user clicks on some “OK” button to perform the image transformations, create a “task” in charge of it
  - Both tasks then run “at the same time”, and thus while the image transformation is being performed, the user can still interact with the app

# Why Concurrency

- The two previous examples illustrate the two main motivations for concurrency
- Make programs faster
  - Because multiple tasks can use different hardware components at the same time
  - e.g., while task #1 uses a core, task #2 uses another core, and task #3 uses the network card
- Make programs more responsive
  - While a task is blocked or doing something time consuming, other tasks can still execute
  - e.g., while a task is stuck waiting for a network packet to arrive, another can display an animated spinning wheel

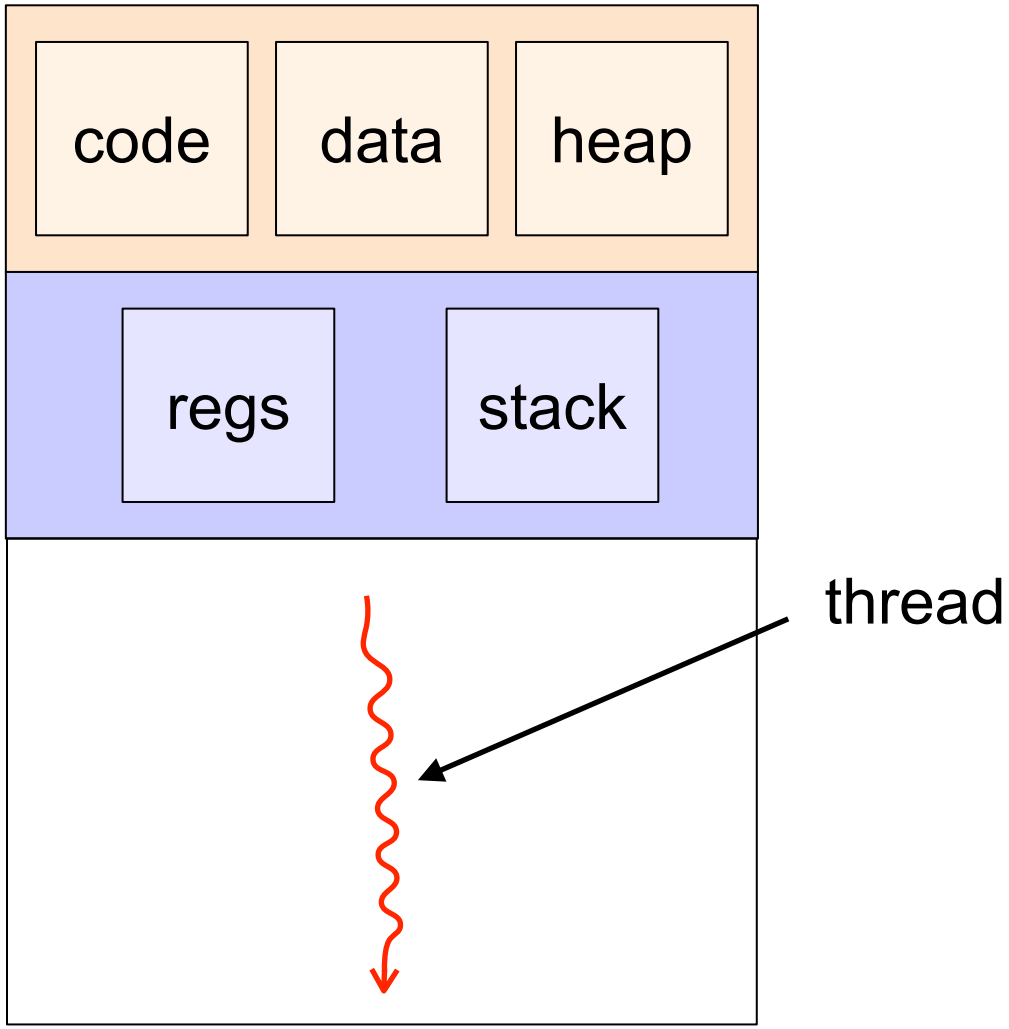
# Concurrency with Processes

- We have already talked about concurrency
  - After all it's the 2nd "easy piece" in our textbook
- Processes run concurrently on the computer
  - They were used for concurrent programming a lot say until the early 90's
  - And still used a lot, e.g., see our programming assignment
- But because the OS virtualizes memory, by default processes don't share memory
- We have seen that processes can communicate with IPC
  - Message passing: often not easy when processes have complicated cooperating behaviors
  - Shared Memory: often simpler, but requires many system calls and cumbersome, up until the arrival of… threads!

# Threads

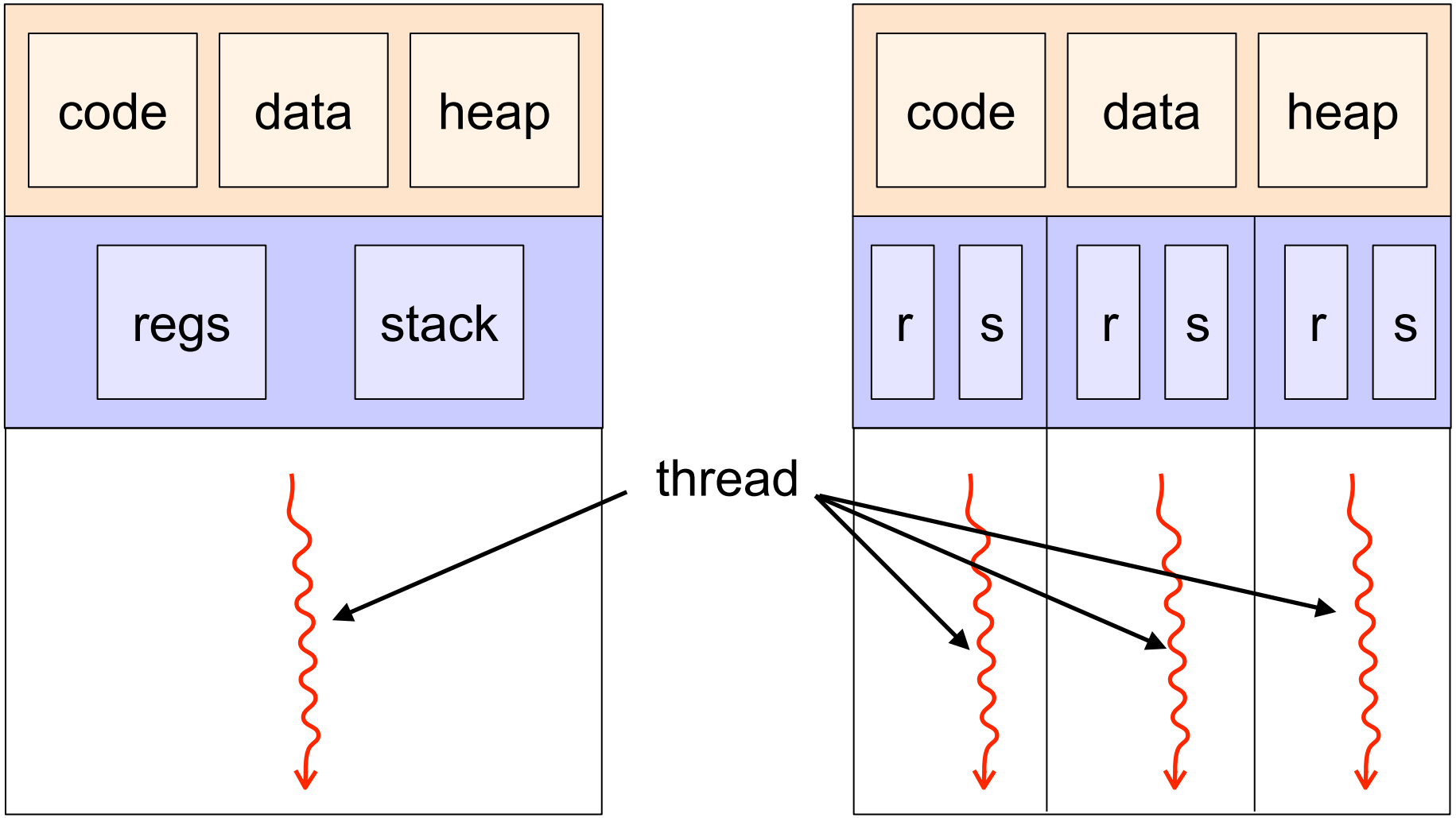
- A thread is a basic unit of CPU utilization within a process (i.e., it's a can be seen as a "task")
- A Multi-threaded process: Concurrent executions of different parts of the same running program, where each execution is a thread
- Each thread has its own:
  - Thread ID (assigned by the OS)
  - Program Counter (which instruction the thread currently executes)
  - Registers Set (which values are stored in registers)
  - Stack (bookkeeping of the thread's function/method invocations)
- The above fully defines "what a thread is doing right now"
- But "within a process" threads share:
  - The code/text section
  - The data segment (global variables)
  - The heap
  - And other things (file descriptors, signal handlers, …)

# Threads: Typical Representation



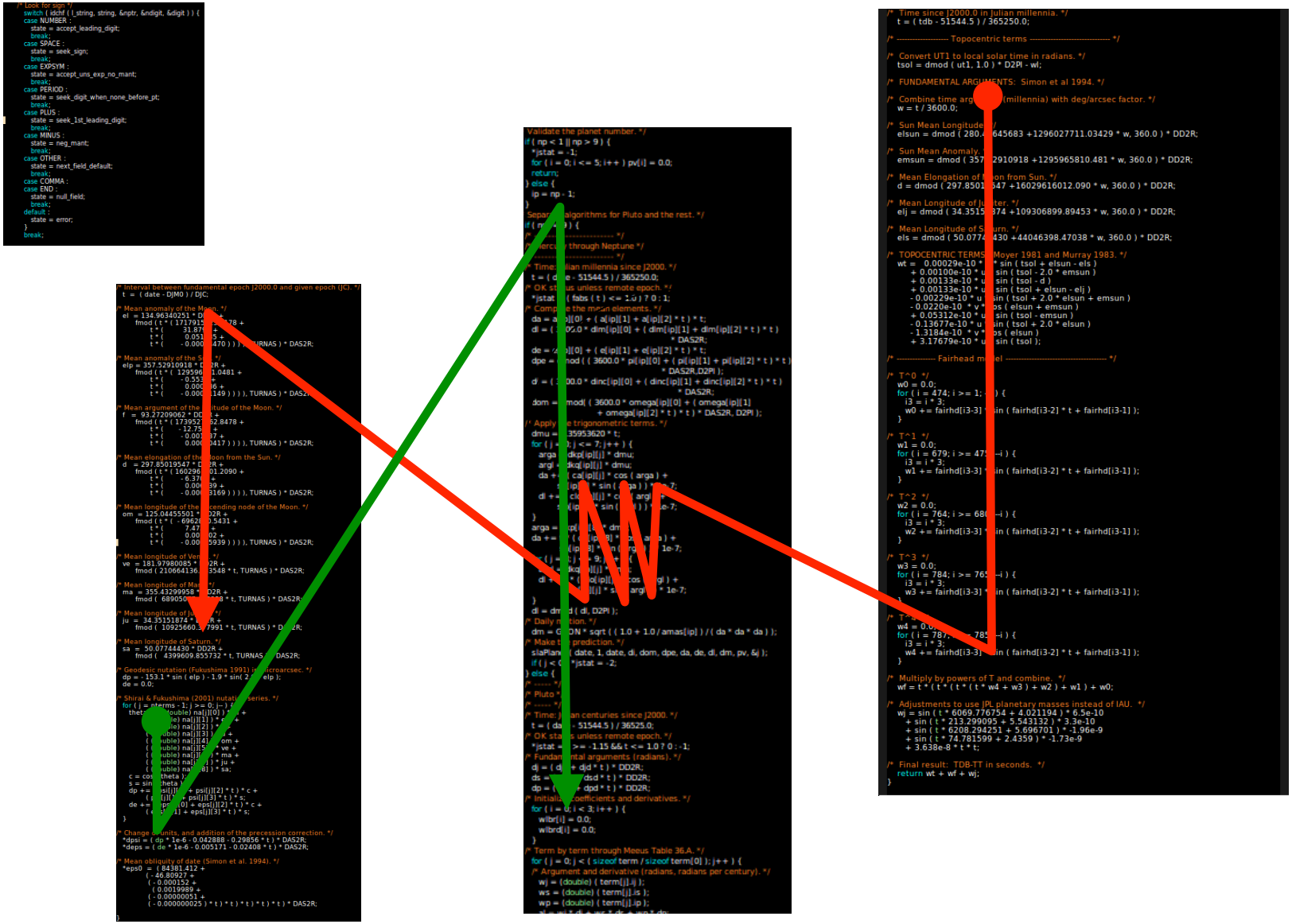

Single-Threaded Process

# Threads: Typical Representation

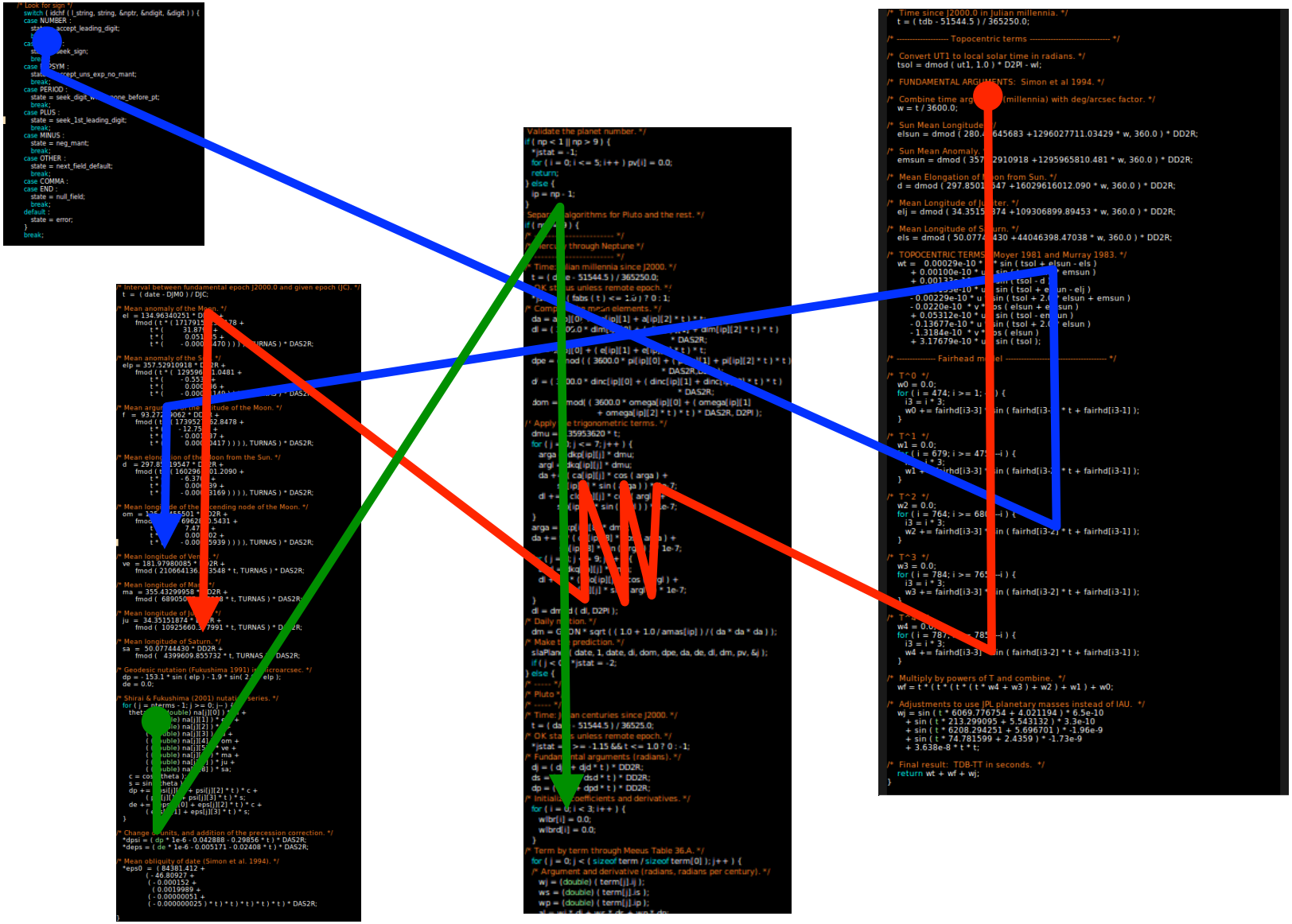


# Multi-Threaded Program in Execution

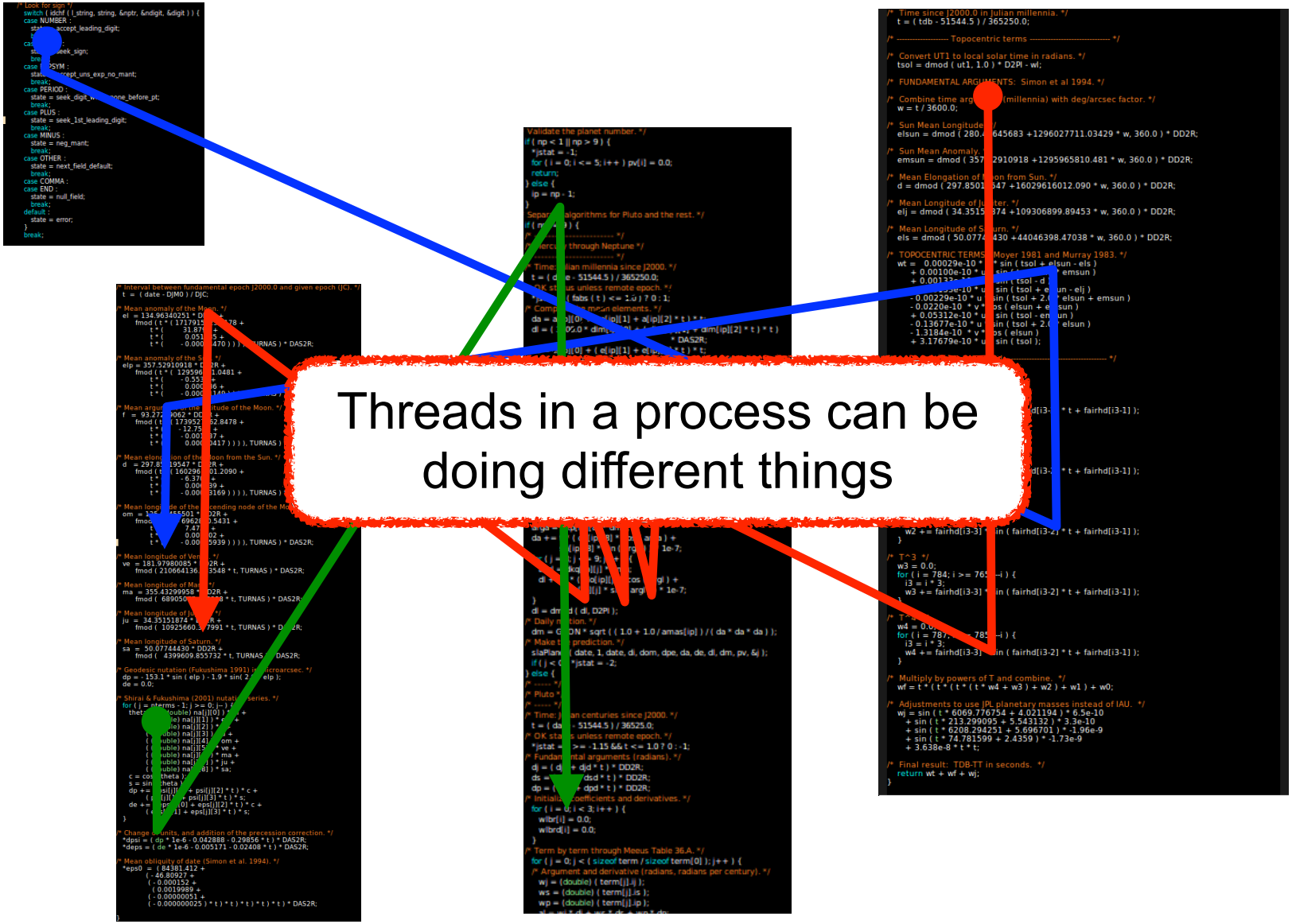
# Multi-Threaded Program in Execution



# Multi-Threaded Program in Execution

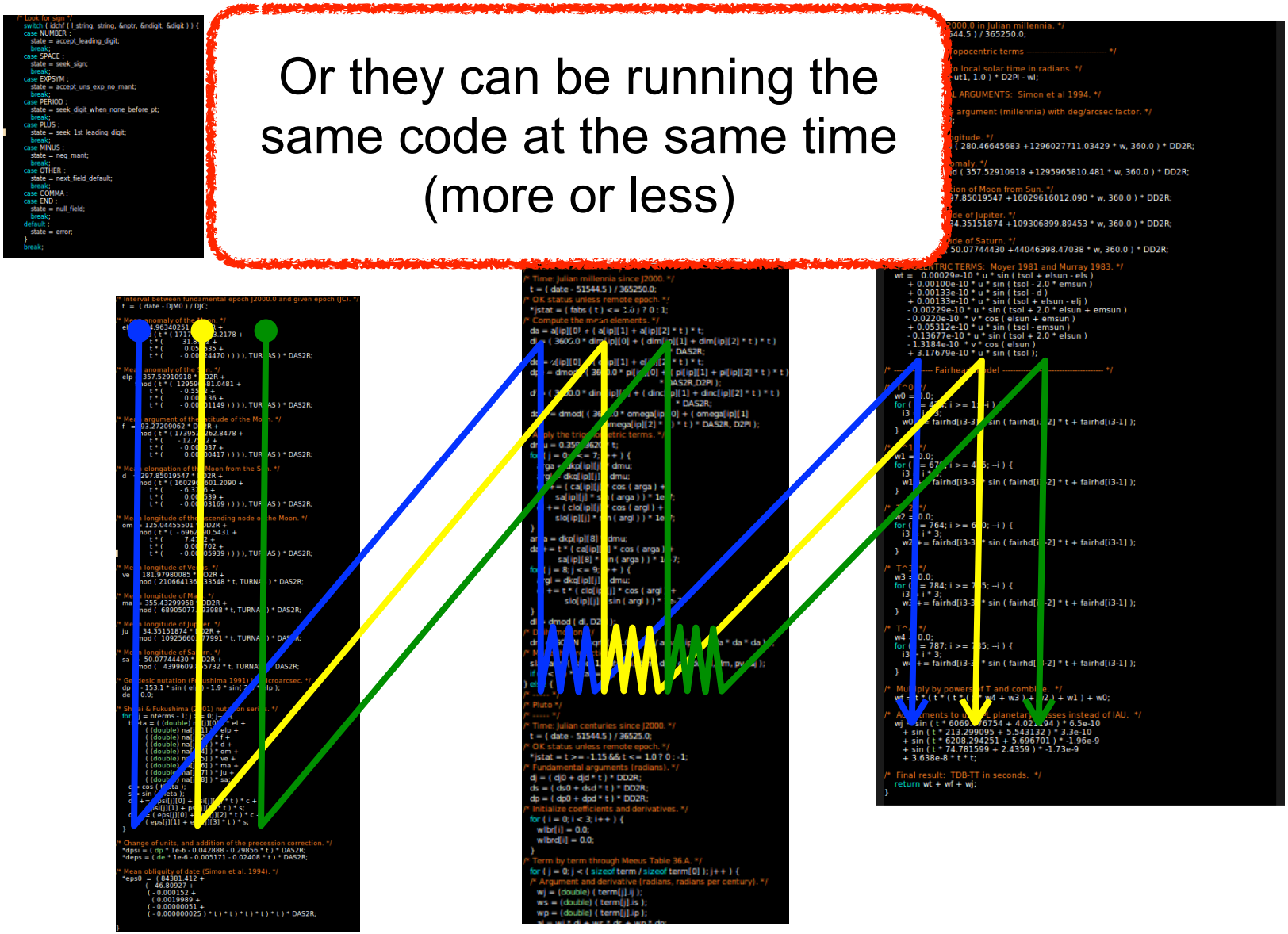


# Multi-Threaded Program in Execution

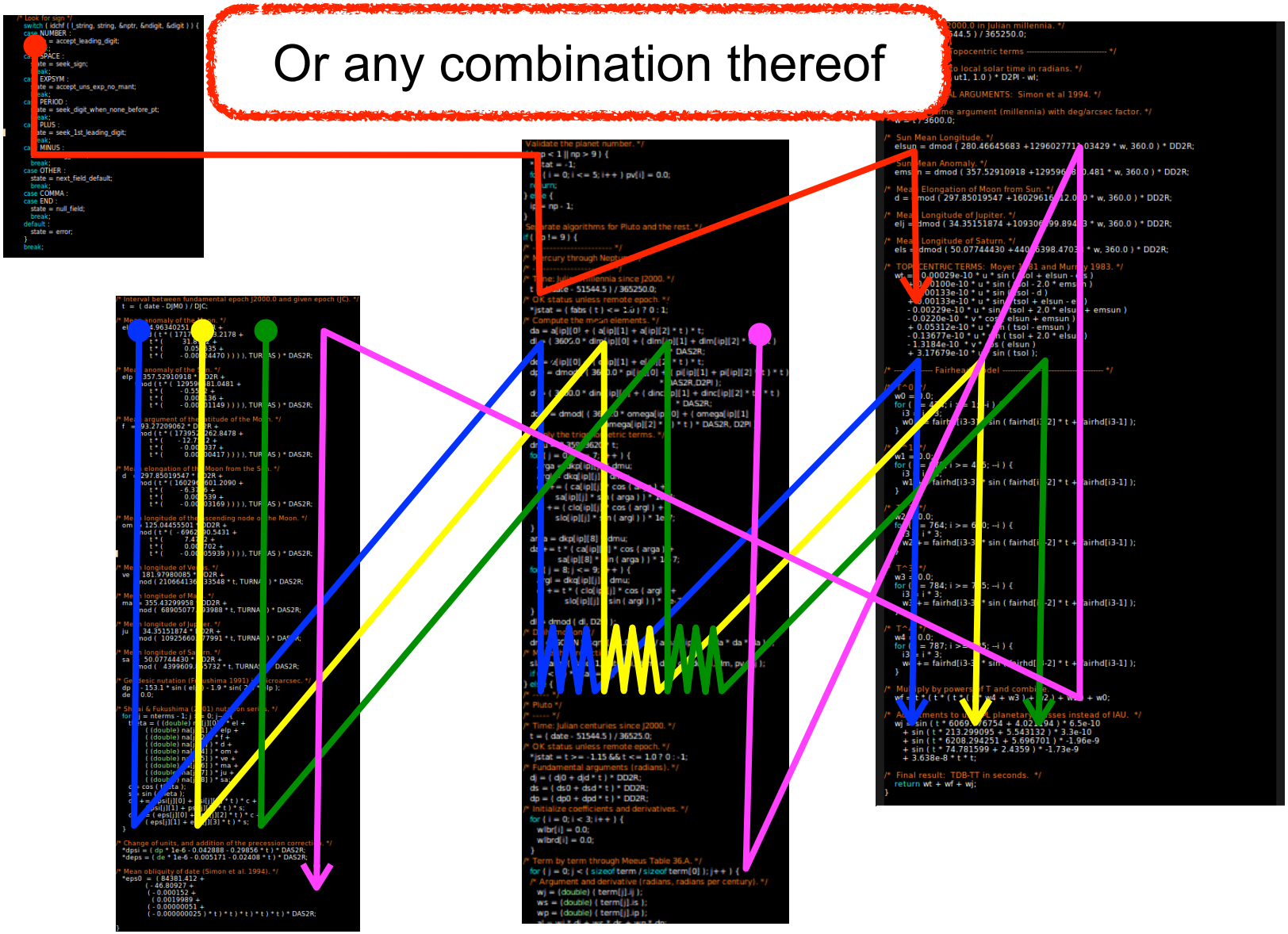


# Multi-Threaded Program in Execution

Or they can be running the same code at the same time (more or less)



# Multi-Threaded Program in Execution



# Threads vs. Processes

- 😀 **Memory sharing**
  - Threads naturally share memory among each other
    - Provides a direct Shared Memory IPC mechanism with no system calls
  - Having concurrent activities in the same address space is very powerful
  - It makes it possible to implement all kinds of concurrency behaviors/patterns
- 😡 **No memory protection**
  - This is a “feature” since we *want* threads to share memory
  - But this can cause really, really difficult bugs
  - More about this in the Synchronization module
- 😀 **Economy**
  - Creating a thread is cheap
    - Slightly cheaper than creating a process in MacOS/Linux
    - Much cheaper than creating a process in Windows
  - Context-switching between threads is cheaper than between processes
  - So if you can do with threads what you can do with processes, then you likely can do it a bit faster
  - In old OSes (Solaris 4), threads were called “lightweight processes”

# Threads vs. Processes

- 😡 **Less fault-tolerance**
  - If a thread fails/crashes, then the whole process fails/crashes, while processes are independent of each other
  - This motivates developers to use both processes and threads (see next slide)
- 😡 **Possibly more memory-constrained**
  - Since threads execute in the same process address space, and an OS can bound the size of a process' address space
  - But that's typically not a big deal (one can configure the OS if need be)
- The advantages here are well worth the drawbacks/limitations
  - The main big drawback is "no memory protection" and we have developed many, many approaches/solutions to deal with it
  - See the Synchronization module
- Natural question: is concurrency with processes obsolete?

# Concurrency with Processes?

- Should we still care about concurrency with processes?
- **YES** because many applications consists of multiple processes (which are often multi-threaded)
- Well-known examples are some popular Web browser (Chrome, Firefox)!
  - They calls fork() each time you open a tab
  - Each tab is a (possibly heavily) multi-threaded process
  - As a result, the code contains processes that do IPC because they don't "see" the same memory naturally
  - But if a tab crashes (due to running bad JavaScript code, for instance) your browser doesn't crash!
  - Google "firefox chrome processes threads" for instance :)
- In real-world settings you often have to put together different software products to make up a whole system
  - Some may just be executables instead of libraries with nice APIs
  - So you have to create processes
  - You interact with them via stdin/stdout/stderr streams for instance (see our programming assignment) or via any supported IPC mechanisms
- **Bottom-line:** don't drink the "I'll only do threads, not processes" Kool-Aid

# User vs. Kernel Threads

- Let's now talk about how the OS implements threads
- Threads can be supported solely in User Space (User Threads)
  - You can write your own thread implementation without help from the OS
  - Often a homework assignment in a graduate OS course
- The main advantage of User Threads is low overhead
  - e.g., because no system calls
- User Threads have several drawbacks:
  - If one thread blocks, all other threads block
  - All threads run on the same core (because the OS doesn't know that there are threads within a process)
- For these reasons User Threads are (no longer) heavily used
- All OSes today provide support for threads (Kernel Threads) that can run on different cores and be truly independent of each other
- We typically just call them "threads"

# Threads in Programming Languages

- C/C++: Pthreads
- C/C++: OpenMP (built on top of Pthreads)
- C++: std::thread
- Java: Java threads (implemented by the JVM, which relies on Pthreads)
- Python: threading / multiprocessing packages
  - WARNING: the threading package implements user threads!!
- Rust: std::thread
- JavaScript: no multithreading in the language, and it won't change, but there are options:
  - Node.js provides worker_threads, but without memory sharing,  a Worker thread implementation
  - There is a standard Web Worker API

- Let's look at Java…

# Java Threads

- Java makes is easy to use threads

- There is a Thread class
- There is a Runnable interface
- There is a Callable interface
- There is an ExecutorService interface

- Let’s see simple examples

# Java Threads

- Java makes is easy to use threads

- There is a Thread class
- There is a Runnable interface
- ~~There is a Callable interface~~
- ~~There is an ExecutorService interface~~

- Let's see simple examples of the **first two**

# Extending the Thread class

- Extend the thread class
- Override the `run()` method with what the thread should do
  - If you forget to override `run()`, your thread won't do anything
- Call the `start()` method to start the thread

**Thread subclass**

```
public class MyThread extends Thread {
  MyThread(…) { … }

  @override
  public void run() { // code for what the thread should do }
}
```

**Main program**

```
public class MyProgram {
  public static void main(…) {
     MyThread myThread = new MyThread(…);
     myThread.start();
     // At this point, 2 threads  are running!
  }
}
```

# run() vs. start()

- You implement the thread's code in **`run()`**
- You start the thread with **`start()`**

- **WARNING**: Calling **`run()`** does **not** create a thread, but it works (it's just a normal method call)
- The **`start()`** method, which you should not override, does all the thread launching
  - It places whatever system calls are needed to start a thread, e.g., **`clone()`**
  - And then makes it so that the newly created thread's fetch-decode-execute cycle begins with the first line of code of the **`run()`** method

# The Runnable Interface

- Using the Runnable interface is preferred because then you can still extend another class
  - Java doesn't have multiple inheritance
  - Typically if you can use an **`implements`** instead of an **`extends`**, you should
    - So that you keep the **`extends`** option open for another purpose
- Let's see an example…

# Using the Runnable Interface

Runnable class

```
public class MyRunnable implements Runnable {
  MyRunnable(…) { … }

  @override
  public void run() { // code for what the thread should do }
}
```

Main program

```
public class MyProgram {

  public static void main(…) {
     //  Create an instance of the runnable class
     MyRunnable myRunnable = new MyRunnable(…);
     // Pass it to the Thread constructor
     Thread thread = new Thread(myRunnable);
     // Start the thread
     thread.start();
    // At this point, 2 threads  are running!
  }
}
```

# In-line Thread Creation

- Sometimes it's cumbersome to create all kinds of Runnable classes, so one can inline everything

Main program

```
public class MyProgram {

  public static void main(…) {

    // Start an anonymous thread with a single statement
    new Thread( new Runnable() {
      @Override
      public void run() {
        …
      }
    }).start();

  }
}
```

# Printing 0's Example

Runnable class

```
public class HelloWorldRunnable implements Runnable {
  private int index;
  public HelloWorldRunnable(int index) {
    this.index = index;
  }
  @Override
  public void run() {
    for (int i=0;i<10000;i++) {
      System.out.print(this.index);
    }
  }
}

public class MyProgram {
  public static void main(String[] args) {
    HelloWorldRunnable helloRunnable = new HelloWorldRunnable(0);
    Thread helloThread = new Thread (helloRunnable);
    helloThread.start();
  }
}
```

# Printing 0's Example

- The previous program runs as a Java process
  - In fact as a thread inside the JVM process
  - We call it the main thread
- When the main thread calls the **`start()`** method it creates a new thread
- We now have two threads that are running:
  - The main threads, who doesn't do anything
  - The newly created thread, who prints a bunch of 0's to the terminal
- In Java, the program terminates only when all your threads terminate (not true in all languages)
  - The main thread terminates when it returns from **`main()`**
  - All others terminate when they return from **`run()`**
- Let's now have the main threads do something as well…

# Printing 0's and 1's Example

Runnable class

```
public class HelloWorldRunnable implements Runnable {
  private int index;
  public HelloWorldRunnable(int index) {
    this.index = index;
  }
  @Override
  public void run() {
    for (int i=0;i<10000;i++) {
      System.out.print(this.index);
    }
  }
}

public class MyProgram {
  public static void main(String[] args) {
    HelloWorldRunnable helloRunnable = new HelloWorldRunnable(0);
    Thread helloThread = new Thread (helloRunnable);
    helloThread.start();
    for (int i=0;i<10000;i++) {
      System.out.print(1);
    }
  }
}
```

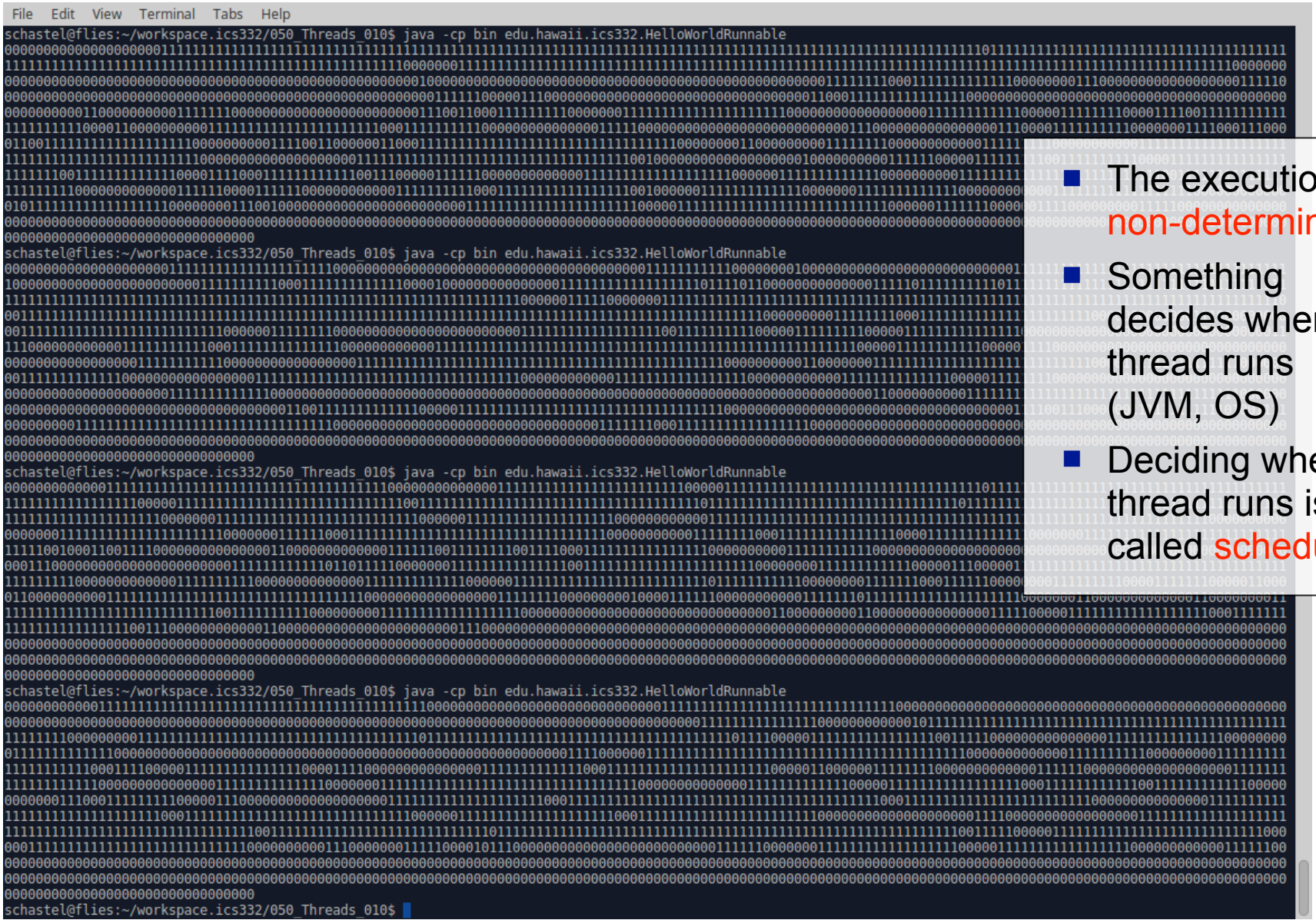
# What to Expect?

- Now we have the main threads printing to the terminal and the new thread printing to the terminal
- What will the output be?

# What to Expect?

- Now we have the main threads printing to the terminal and the new thread printing to the terminal
- What will the output be?
- Answer: Impossible to tell for sure
  - If you know the details of the implementation of the JVM on your host, and you know your OS and hardware well, perhaps you can have some idea of what it might look like
  - ... but it's not very useful because it will look different on a different setup (it's not portable) and different each time you run it
- Let's have a look at a few executions…

# Output Samples




- The execution is non-deterministic
- Something decides when a thread runs (JVM, OS)
  - Deciding when a thread runs is called scheduling

# Multi-Threaded Programming

- Major challenge: You cannot make any assumption about thread scheduling, since the OS is in charge
  - And what the OS does depend on the hardware and on other running processes
- Major difficulty: You may not be able to reproduce a bug because each execution is different!
  - Makes it hard to debug!
  - Worse: you may think your code is working, but that's because you haven't been able to observe the bug yet…
  - If you run your code 10,000 times and don't see the bug, you still cannot be sure that the bug will not happen
  - But, someday, your users will 😱

# Java/Kernel Threads

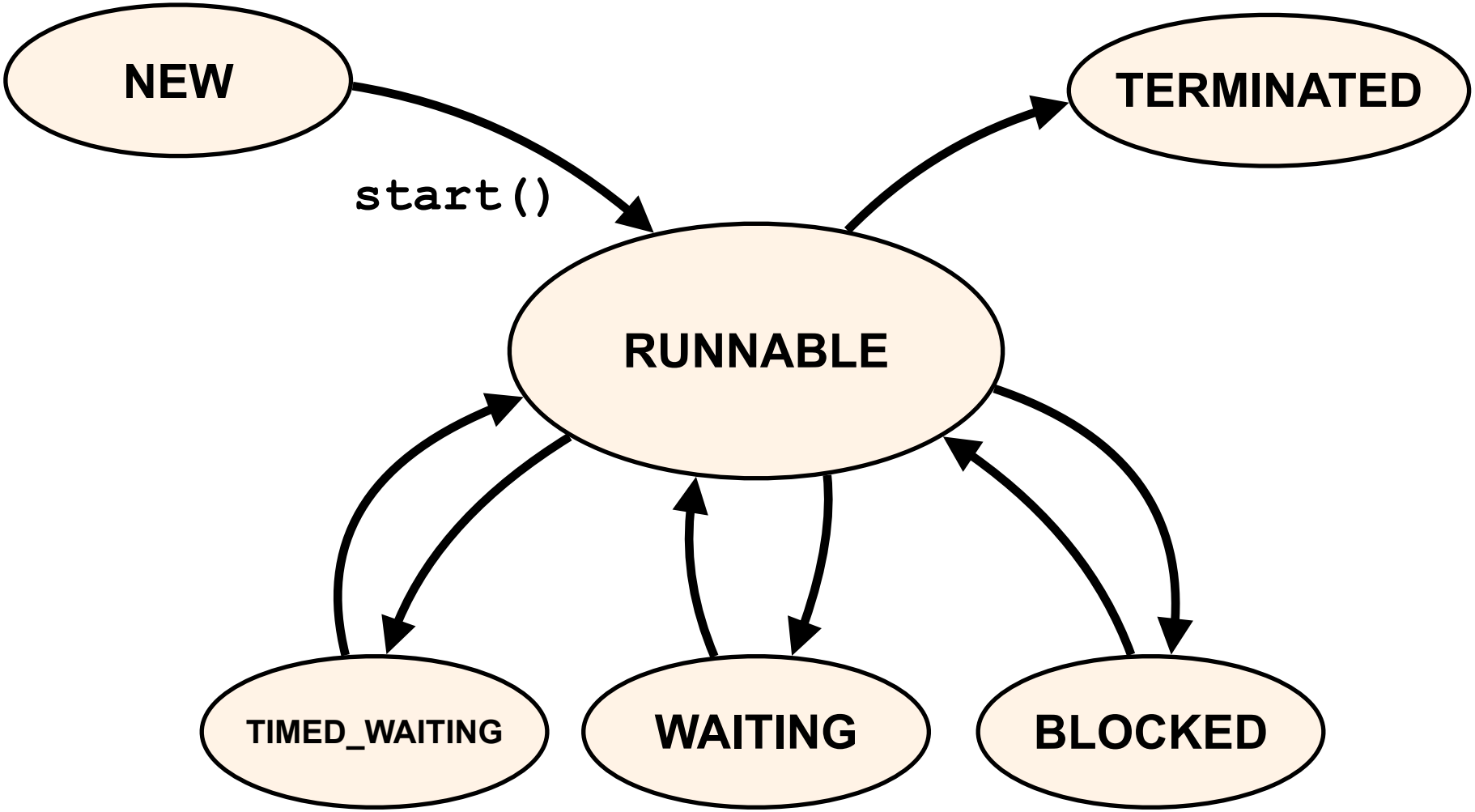
- The JVM is itself multi-threaded!
  - The JVM has a thread scheduler for application threads, which are mapped to kernel threads
    - Several application threads could be mapped to the same kernel thread (they are then “user threads”)
    - That thread scheduler runs itself in a dedicated thread
    - The OS is in charge of scheduling kernel threads
  - But it also runs many threads itself (e.g., the garbage collector)

- In a nutshell: Threads are everywhere
  - Kernel threads that run application threads
  - Kernel threads that do some work for the JVM

# Influencing Threads?

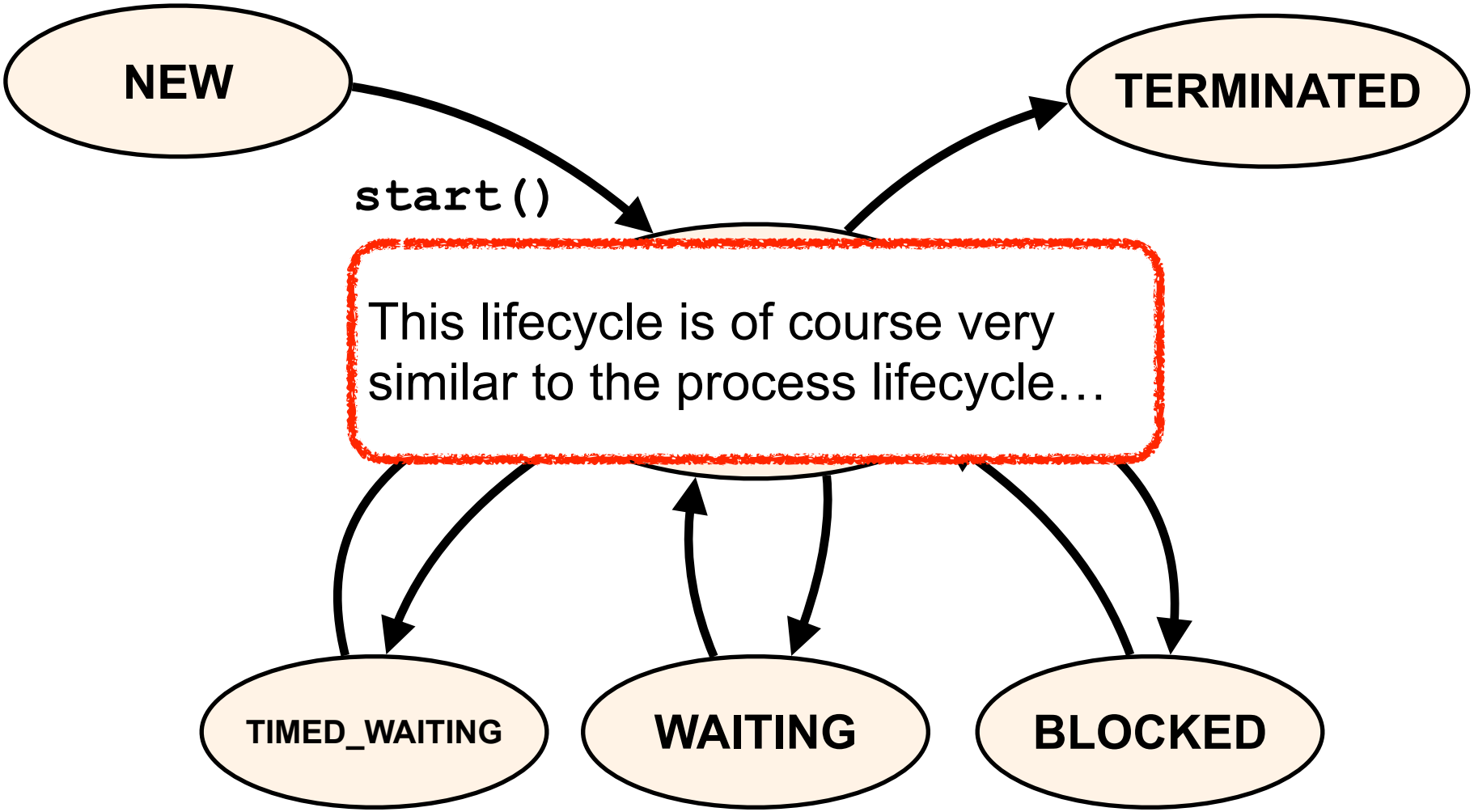
- At this point, it seems that we throw a bunch of threads in, the OS “shakes the bag”, and we don’t really know what happens
- To some extent this is true, but we have ways to influence what happens control
- In Java, a thread can call **`Thread.yield()`**, which says “I am willingly giving up the CPU now”
  - But it is still not deterministic!
  - Programs should NEVER rely on yield() for correctness (it’s more a hint to the JVM, and can help for interactivity)
- In Java, there is a **`Thread.setPriority()`** method
  - Thread priorities are integers ranging between Thread.MIN PRIORITY and Thread.MAX PRIORITY, the greater the integer, the higher the priority
  - Again, these are hints and you can’t rely on them (and they don’t work at all on some JVM implementations!!)
- All the above are basically “hints that may have some effect”, nothing more
  - So they don’t really “solve” anything for certain
- **Bottom Line:** Orchestrating thread executions requires more advanced features (stay tuned…)

# Java Thread LifeCycle



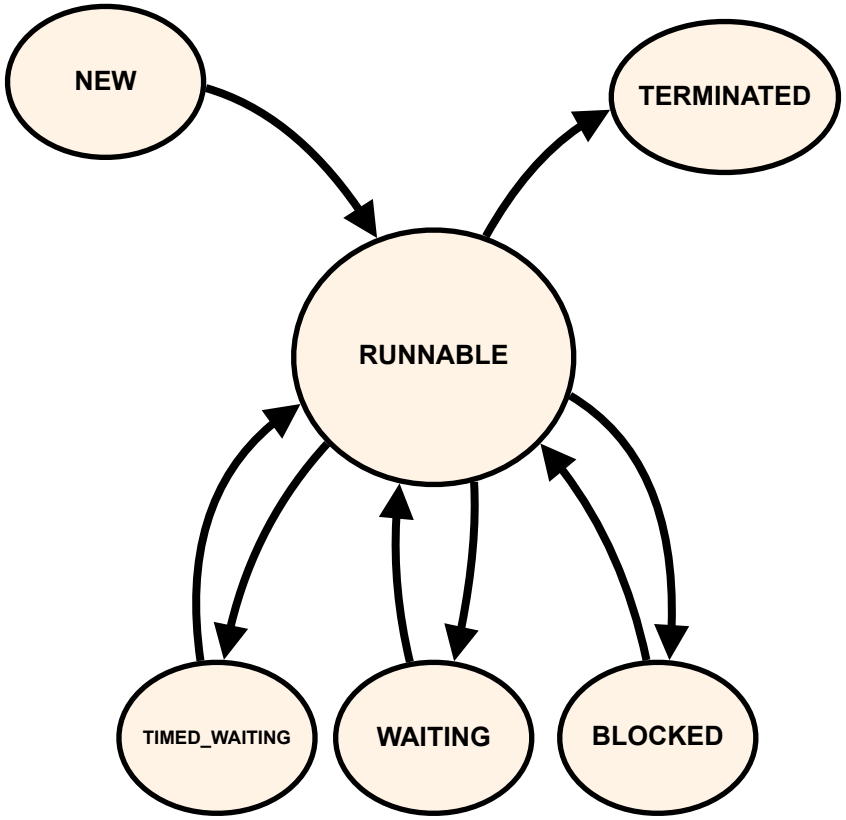

These three states are reached when calling various methods

# Java Thread LifeCycle

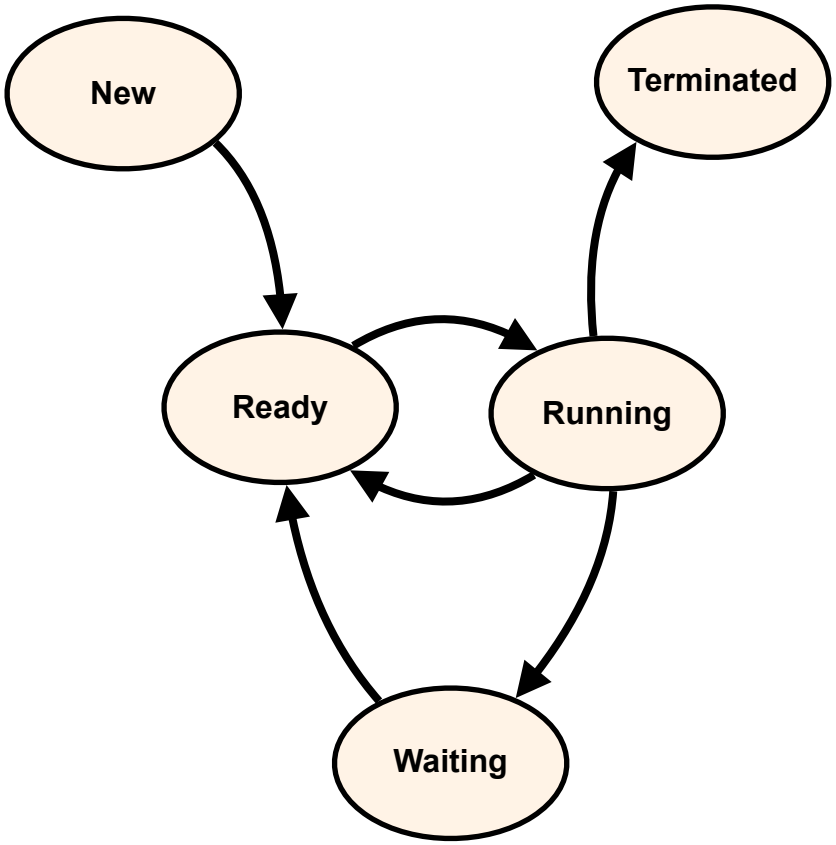



These three states are reached when calling various methods

# Flashback: Process LifeCycle




Java Threads




OS Processes/Threads

# Linux/MacOS Threads

- Processes and Threads are implemented as tasks
  - Kernel data structure: **`task struct`**
  - We already looked at it when we talked about processes
- The **`clone()`** syscall is used to create a task
- It can be invoked with several options, each set or not set
- Each option specifies something the child should share or not share with its parent
- **`fork()`** just calls **`clone()`** with a particular set of options
  - Preserved as a system call for backward compatibility to create processes
- From the man page: “if CLONE_VM is set, the calling process and the child process run in the same memory space”
- To create a process, **`clone()`** is called without the CLONE_VM option
- To create a thread, **`clone()`** is called with, among other things, the CLONE_VM option

# Java Threads: the join() Method

- The `join()` method causes a thread to wait for another thread's termination

**Example program**

```
public class JoinExample {
  public static void main(String args[]) {
    // Create a thread
    Thread t = new Thread (new Runnable() {
     public void run() { . . . }});

    // Spawn it
    t.start();

    // Do some work myself
    . . .

    // Wait for the thread to finish
    try {
      t.join();
    } catch (InterruptedException e) {}
  }
}
```

- Useful to give work to do to a thread
- This is our first example of thread "synchronization"
- Synchronization is a generic word used to denote ways in which one can control the execution of a group of threads
- We'll talk more about this in the Synchronization module

# Conclusion

- Multi-threading today is everywhere, in part due to us having multi-core architectures
- Let's do a **`ps axuM`** on my MacOS laptop and see how many processes are multi-threaded…
  - When I did this while back writing this slides I got 350 processes and 1157 threads. Almost all processes are multithreaded, with up to 60+ threads for a process.
- In this course we focus more on how the OS implements threads than how the user uses threads
- There are many, many more things we could talk about regarding using threads and Java threads
  - We'll talk more about this in the Synchronization module
  - ICS432 is all about that