Threads: Programming

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

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()`** on Linux
  - 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 declare one-shot 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 thread, which doesn't do anything
  - The newly created thread, which prints 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 thread 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);
    }
  }
}
```

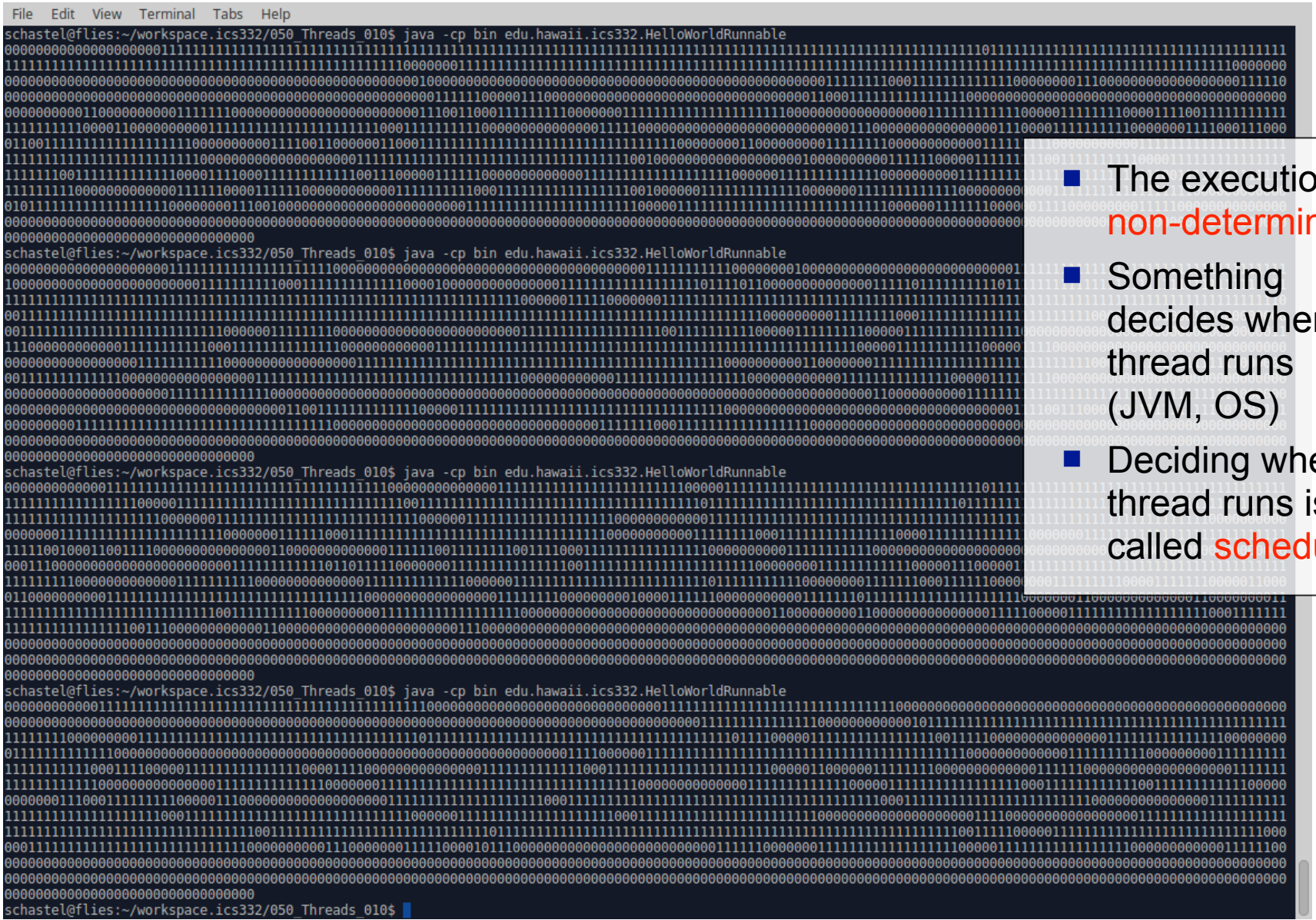
What to Expect?

- 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 at least a bit 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 correct, 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)

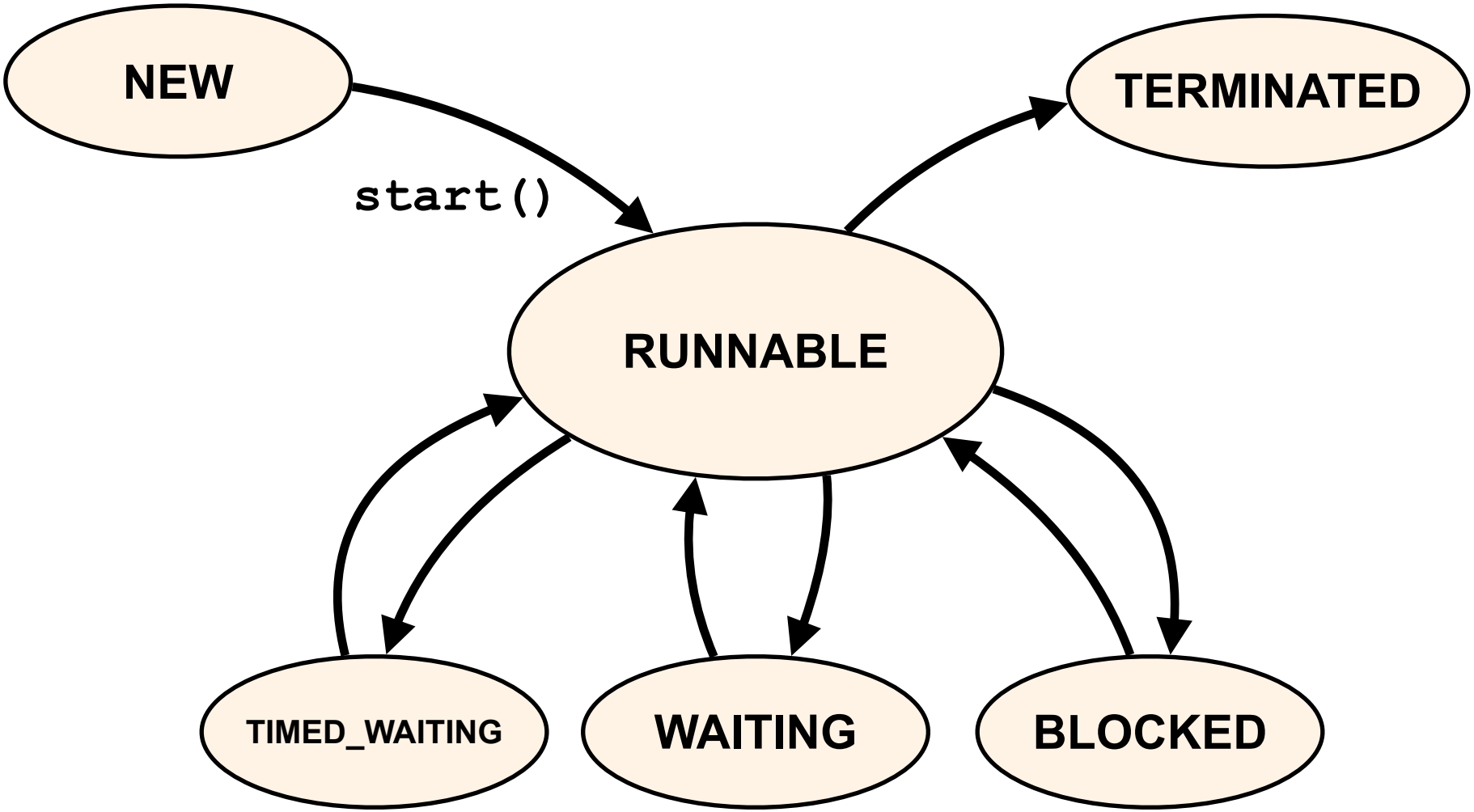
- In a nutshell: Threads are everywhere
  - Kernel threads that run application threads
  - Kernel threads that do some work for the JVM

- Let’s write a Java program that does nothing and count threads (**`ps auxM`**)…

Influencing Threads?

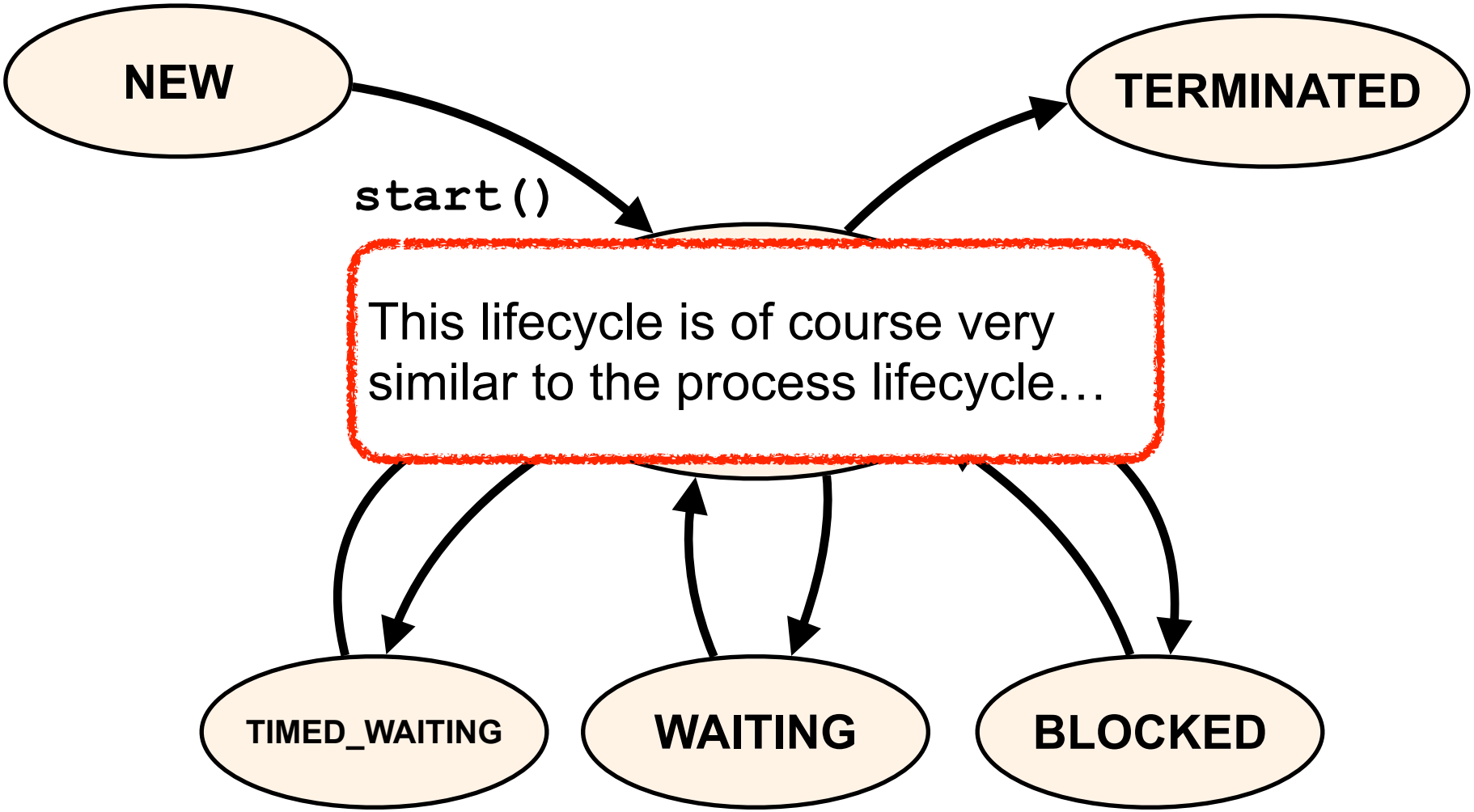
- At this point, it seems that we throw a bunch of threads in, the OS “shakes the bag”, and we don’t really know what happens
- To some extent this is true, but we have ways to influence what happens control
- In Java, a thread can call **`Thread.yield()`**, which says “I am willingly giving up the CPU 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
  - 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 “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



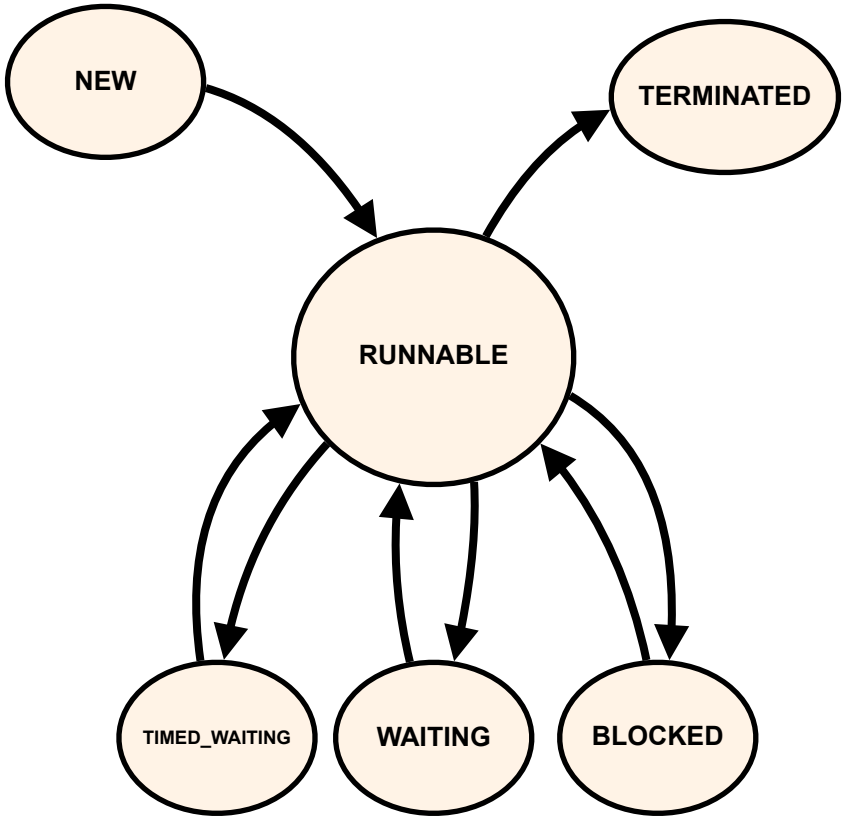
These three states are reached when calling various methods

Java Thread LifeCycle

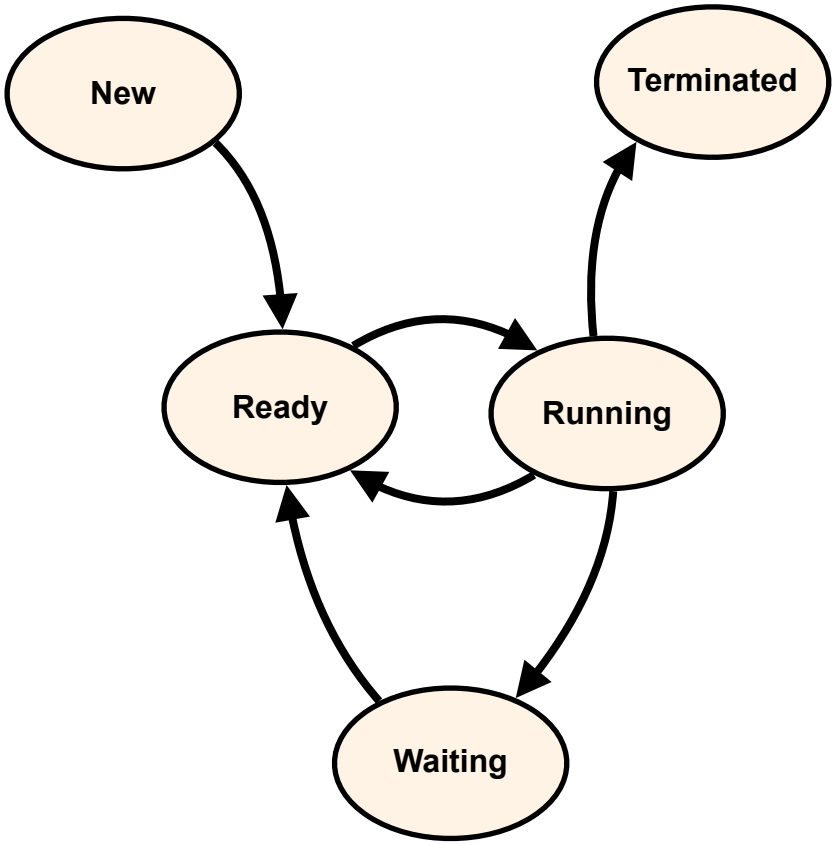


These three states are reached when calling various methods

Flashback: Process LifeCycle



Java Threads



OS Processes/Threads

Java Threads: the join() Method

- The **`Thread::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

Main Takeaways

- Java has several ways to create threads
- The easiest is to extend the Thread class
- A better way (software engineering wise) is to implement the Runnable interface
- Execution can be non-deterministic
- The Thread::join() method causes a thread to wait for the termination of another

Conclusion

- In this course we don't have a large focus on using threads
  - We barely scratched the surface in these lecture notes
  - We'll talk more about programming with threads in the Synchronization module
- The main topic of ICS 432 is using threads
- Quiz next week on this module...
- Optional Homework Assignment #4...
- This concludes all material for Midterm #1!