Threads

ICS332

Operating Systems

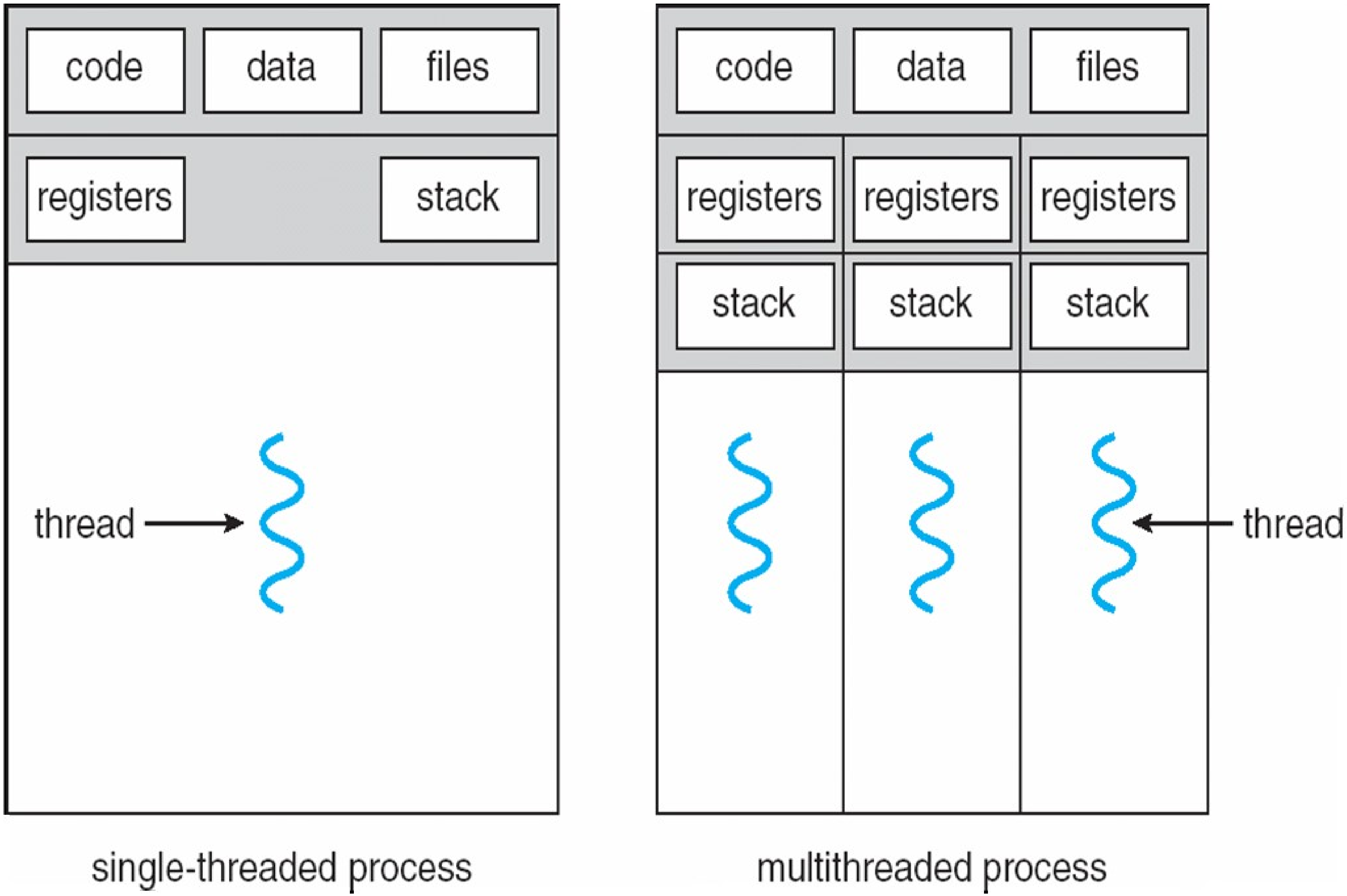
Definition

- Concurrent computing: several computations are performed during overlapping time periods (concurrent instead of sequential)
- Concurrent $\subsetneq$ Parallel
- **Concurrency**: Property of a program that can do multiple things at the same time
- More details? => ICS432

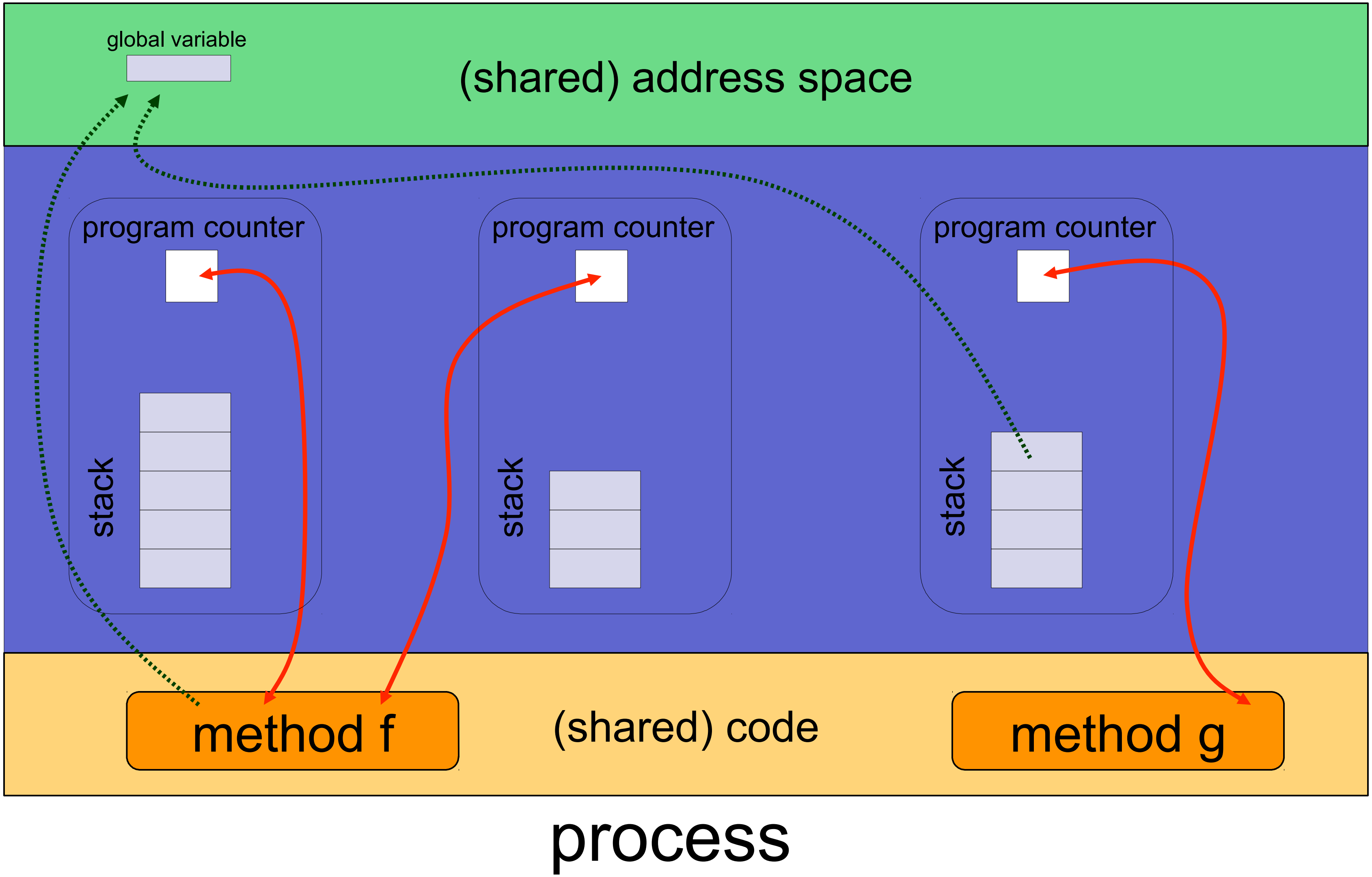
Definition

- A thread is a basic unit of CPU utilization within a process
- Multi-threaded process: Concurrent execution of different parts of the same program
- Each thread has its own
  - thread ID
  - program counter
  - register set
  - stack
- It **shares** the following with other threads within the same process
  - code section
  - data section
  - the heap (dynamically allocated memory)
  - open files and signals

The Typical Figure

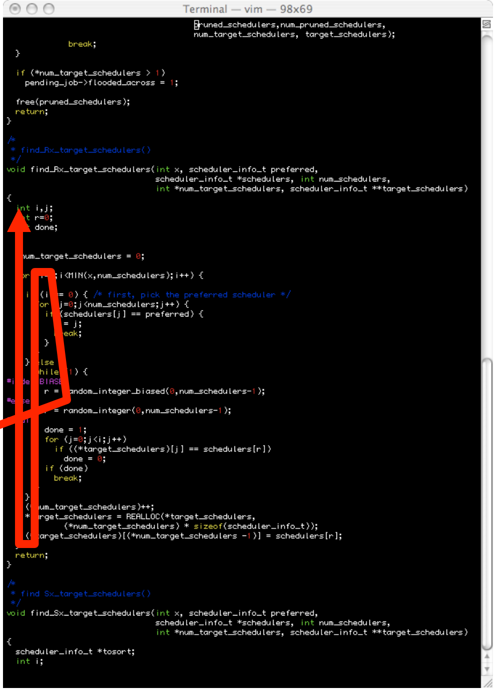


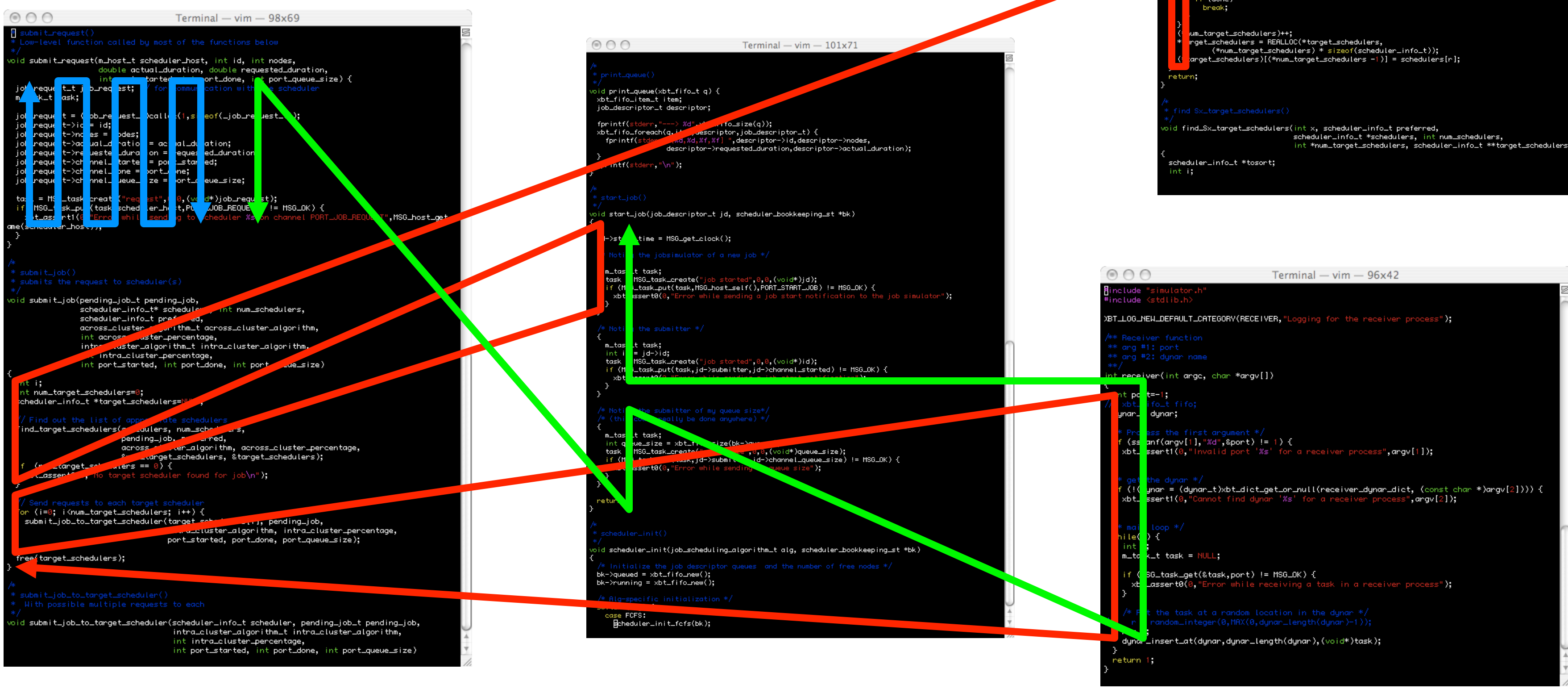
A More Detailed Figure



Multi-Threaded Program

- Source-code view
  - a blue thread
  - a red thread
  - a green thread





Advantages of Threads?

- Economy:
  - Creating a thread is cheap
    - Slightly cheaper than creating a process under MacOSX / Linux
    - Much cheaper than creating a process under Windows (createProcess)
  - Context-switching between threads is cheap
    - Usually cheaper than between processes
- Resource Sharing:
  - Threads naturally share memory
    - With processes you have to use possibly complicated IPC (e.g., Shared Memory Segments)
  - Having concurrent activities in the same address space is very powerful
    - But fraught with danger

Advantages of Threads?

- Responsiveness
  - A program that has concurrent activities is more responsive
    - While one thread blocks waiting for some event, another can do something
    - e.g. Spawn a thread to answer a client request in a client-server implementation
  - This is true of processes as well, but with threads we have better sharing and economy
- Scalability
  - Running multiple “threads” at once uses the machine more effectively
    - e.g., on a multi-core machine
  - This is true of processes as well, but with threads we have better sharing and economy

Drawbacks of Threads

- One drawback of thread-based concurrency compared to process-based concurrency: If one thread fails (e.g., a segfault), then the process fails
  - And therefore the whole program
- This leads to process-based concurrency
  - e.g., The Google Chrome Web browser
  - See http://www.google.com/googlebooks/chrome/
  - Sort of a throwback to the pre-thread era
    - Threads have been available for 20+ years
    - Very trendy recently due to multi-core architectures

Drawbacks of Threads

- Threads may be more memory-constrained than processes
  - Due to OS limitation of the address space size of a single process
- Threads do not benefit from memory protection
  - Concurrent programming with Threads is hard
    - But so is it with Processes and Shared Memory Segments
  - We will see this a bit in this course, and much more in ICS432

Threads on My Machine?

- Let’s run *ps uxM* (or *ps -f -m x*) and look at several applications
  - …
- Let’s compute the thread/process ratio on my machine
  - Parsing the ps output using sed, for instance

Multi-Threading Challenges

- Typical challenges of multi-threaded programming
  - Dividing activities among threads
  - Balancing load among threads
  - Split data among threads
  - Deal with data dependency and synchronization
  - Testing and debugging
- Take ICS432 if you want maximum exposure to these
  - Section 4.2 talks a little bit about this
  - Note that you'll most likely all write multi-threaded code on multi-core architectures

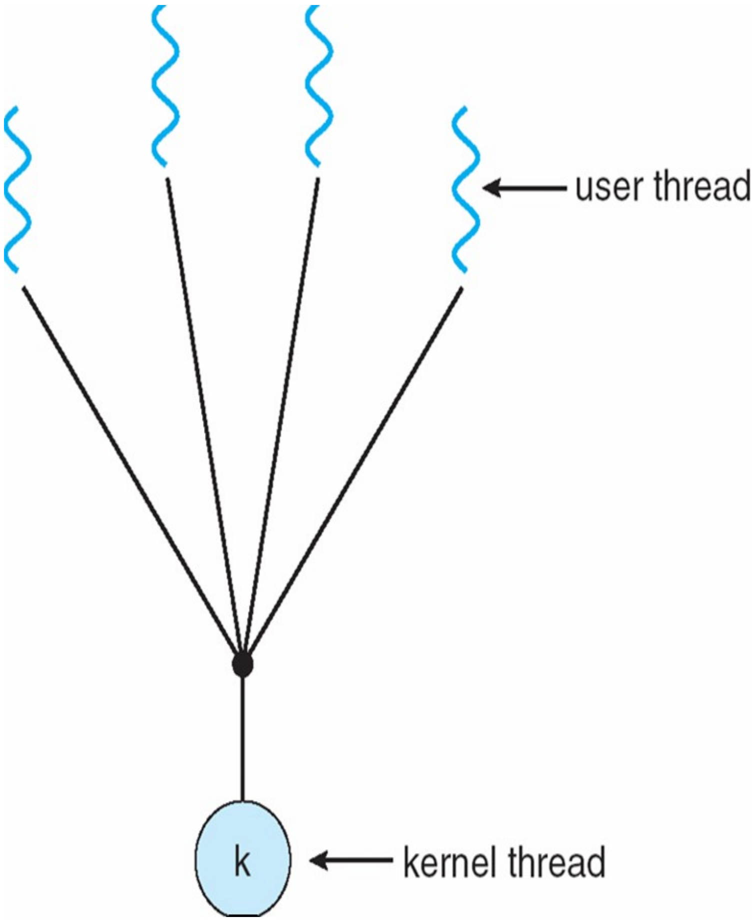
User Threads vs. Kernel Threads

- Threads can be supported solely in User Space
  - Threads are managed by some user-level thread library (e.g., Java Green Threads)

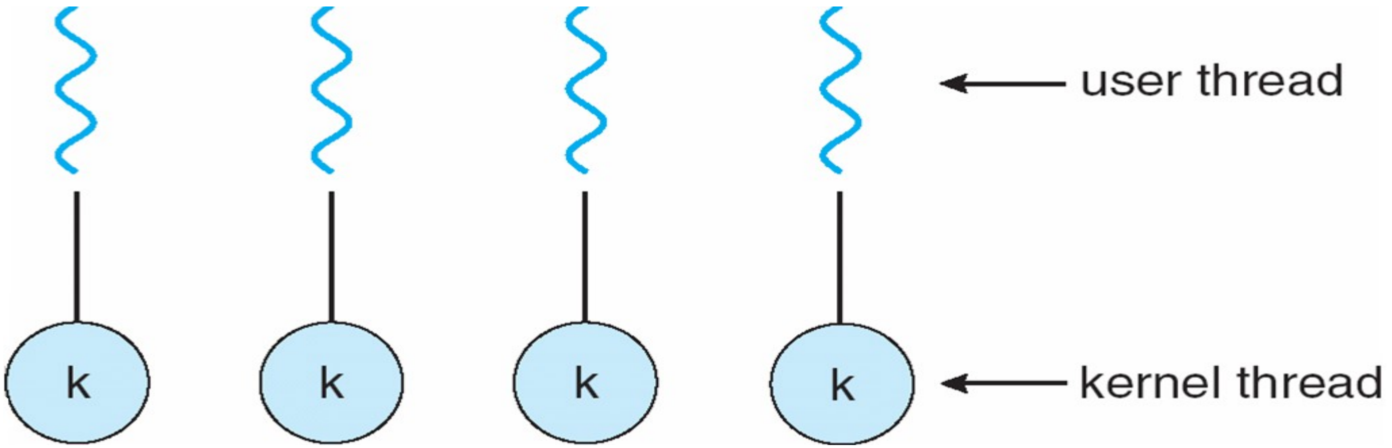
    (i.e.: you can implement your own threads management system and the OS will not know about it)
- Threads can also be supported in Kernel Space
  - The kernel has data structure and functionality to deal with threads
  - Most modern OSes support kernel threads
    - In fact, Linux doesn't really make a difference between processes and threads (same data structure)

Many-to-One Model

- Advantage: multi-threading is efficient and low-overhead
  - No syscalls to the kernel
- Major Drawback #1: cannot take advantage of a multi-core architecture!
- Major Drawback #2: if one threads blocks, then all the others do!
- Examples (User-level Threads):
  - Java Green Threads
  - GNU Portable Threads



One-to-One Model

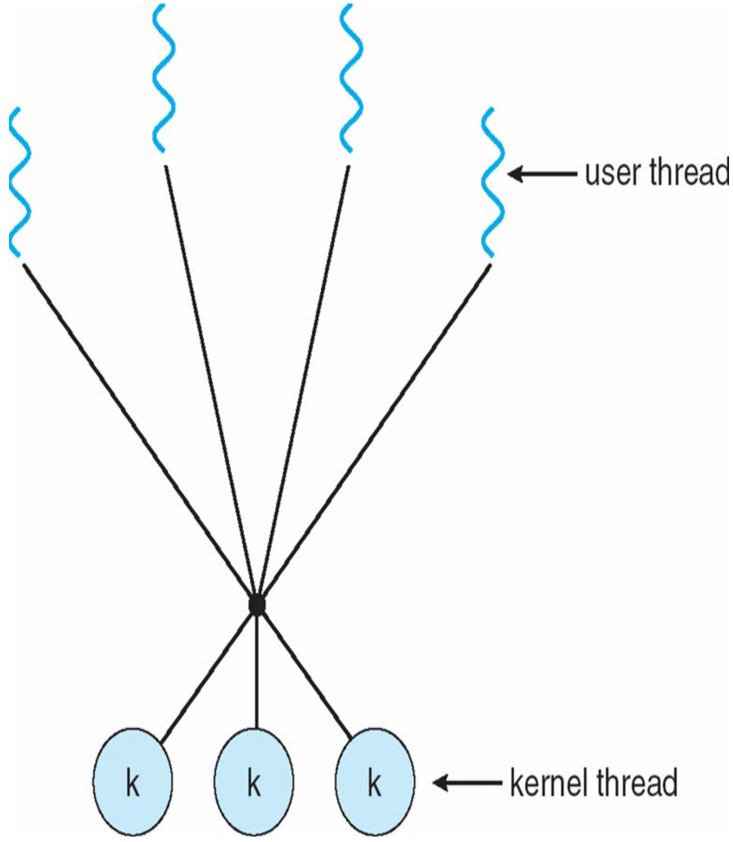


- Removes both drawbacks of the Many-to-One Model
- Creating a new threads requires work by the kernel
  - Not as fast as in the Many-to-One Model

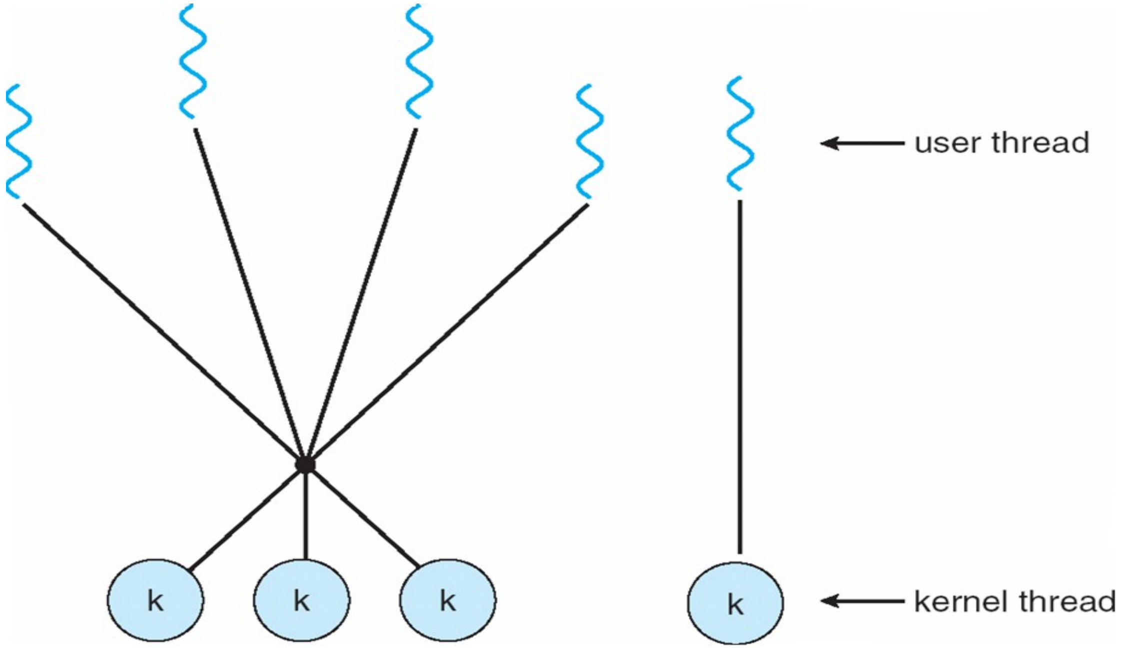
- Example:
  - Linux
  - Windows
  - Solaris 9 and later

Many-to-Many Model

- A compromise
- If a user thread blocks, the kernel can create a new kernel threads to avoid blocking all user threads
- A new user thread doesn't necessarily require the creation of a new kernel thread
- True concurrency can be achieved on a multi-core machine
- Examples:
  - Solaris 9 and earlier
  - Win NT/2000 with the ThreadFiber package



Two-Level Model



- The user can say: “Bind this thread to its own kernel thread”
- Example:
  - IRIX, HP-UX, Tru64 UNIX
  - Solaris 8 and earlier

Thread Libraries

- Thread libraries provide users with ways to create threads in their own programs
  - In C/C++: Pthreads
    - Implemented by the kernel
  - In C/C++: OpenMP
    - A layer above Pthreads for convenient multithreading in “easy” cases
  - In Java: Java Threads
    - Implemented by the JVM, which relies on threads implemented by the kernel

Java Threads

- All memory-management headaches go away with Java Threads
  - In nice Java fashion
- Several programming languages have long provided constructs/abstractions for writing concurrent programs
  - Modula, Ada, etc.
- Java does it like it does everything else, by providing a Thread class
  - You create a thread object
  - Then you can start the thread

Extending the Thread class (All Java)

- To create a thread, you can extend the Thread class and override its “run()” method

```
class MyThread extends Thread {
    public void run() {
        . . .
    }
    . . .
}

MyThread t = new MyThread();
```

Implementing the Runnable interface (All Java)

- To create a thread, you can implement the Runnable interface and its “run()” method

```
class MyStuff implements Runnable {
    public void run() {
        . . .
    }
    . . .
}

MyThread t = new Thread(new MyStuff());
```

Implementing the Callable interface (Java1.5+)

- Implement the Callable interface and its “call()” method
- Adds a return type to call() and checked exceptions!

```
class MyBetterStuff implements Callable<Long> {
    public Long call() throws Exception {
      . . .
        return someLong;
    }
    . . .
}
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(new MyBetterStuff());
```

Example

```
public class MyThread extends Thread {
    public void run() {
        for (int i=0; i<10; i++) {
            System.out.println(“Hello world #“+i);
        }
    }
    . . .
}

myThread t = new MyThread();
```

Spawning a Thread/Runnable

- To launch, or spawn, a thread, you just call the (encapsulating) thread's start() method
- **WARNING**: Don't call the run() method directly to launch a thread
  - If you call the run() method directly, then you just call some method of some object, and the method executes
    - Fine, but probably not what you want
  - The start() method, which you should not override, does all the thread launching
    - It launches a thread that starts its execution by calling the run() method

Example

```
public class MyThread implements Runnable {
    public void run() {
      for (int i=0; i<5; i++) {
          System.out.println(“Hello world #“+i);
      }
    }
}

public class MyProgram {
  public MyProgram() {
    MyThread t = new Thread(new MyThread());
     t.start();
  }
  public static void main(String args[]) {
    MyProgram p = new MyProgram();
  }
}
```

Example

```
public class MyThread implements Callable<Integer> {W
    public Integer call() throws Exception {
       for (int i=0; i<5; i++) {
           System.out.println(“Hello world #“+i);
       }
       Thread.sleep(10000);
       return 42;
    }
}

public class MyProgram {
  public static void main(String args[]) {
     ExecutorService executor = Executors.newFixedThreadPool(4);
     Future<Long> future = executor.submit(new MyThread());
     long value = future.get();
     //... and after 10000 ms, value is 42
  }
}
```

What happens

- The previous program runs as a Java process
  - that is, a thread running inside the JVM
- When the start() method is called, the main thread creates a new thread
- We now have two threads
  - The “main”, “original” thread
  - The newly created thread
- Both threads are running
  - The main thread doesn’t do anything
  - The new thread prints messages to screen and exits
- When both threads terminate, the process terminates
- Let’s have the first thread do something as well...

Example

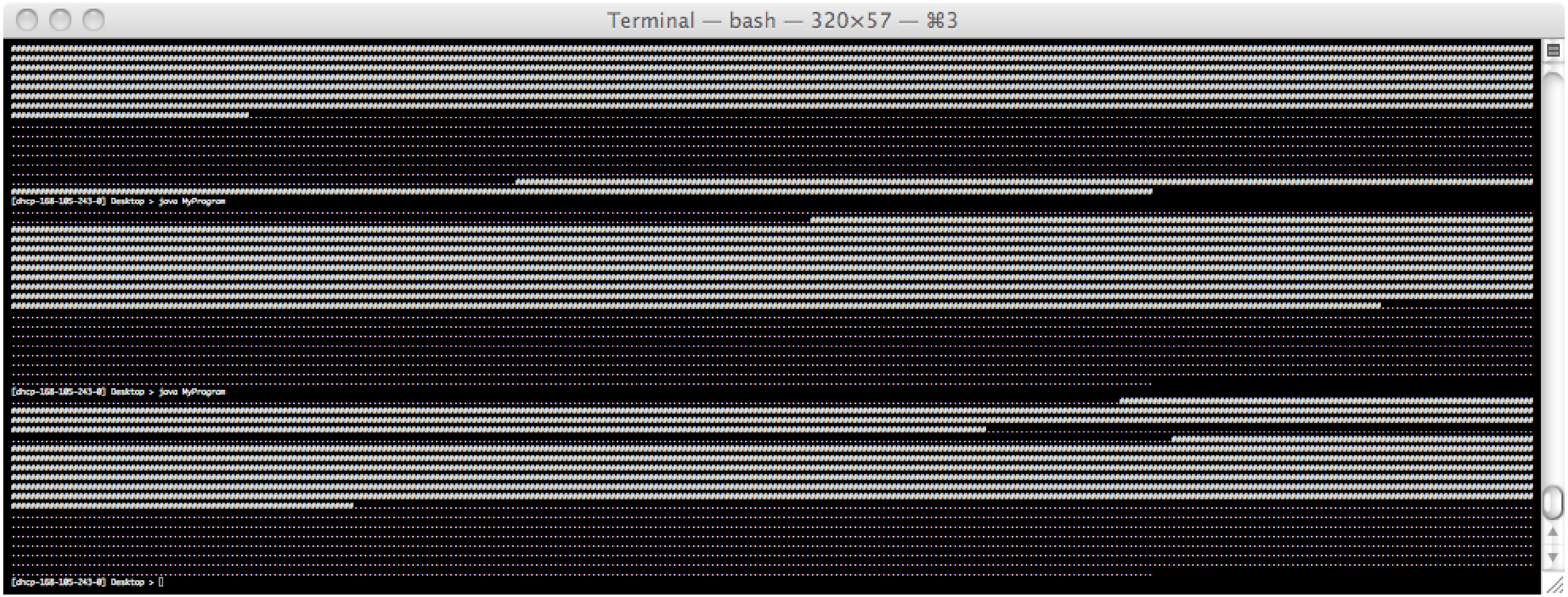
```
public class myThread extends Thread {
    public void run() {
      for (int i=0; i<5; i++)
          System.out.println(“Hello world #“+i);
    }
}

public class MyProgram {
  public MyProgram() {
     MyThread t = new MyThread();
      t.start();
      for (int i=0; i<5; i++)
          System.out.println(“Beep ”+i);
  }
  public static void main(String args[]) {
     MyProgram p = new MyProgram();
  }
}
```

What happens?

- Now we have the main thread printing to the screen **and** the new thread printing to the screen
- Question: what will the output be?
- Answer: Impossible to tell for sure
  - If you know the implementation of the JVM on your particular machine, then you may be able to tell
  - But if you write this code to be run anywhere, then you can't expect to know what happens
- Let's look at what happens on my laptop for a program in which on thread prints "#" and the other prints "." 1000 times each

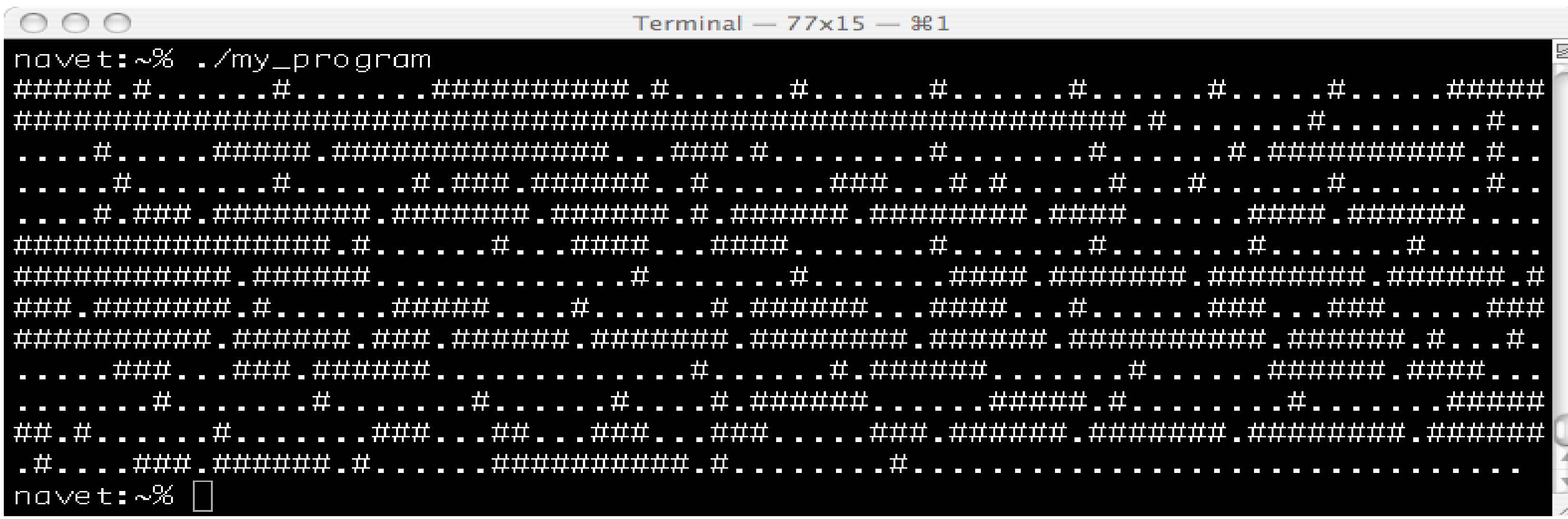
Three Sample Output



- Non-deterministic execution
- Somebody decides when a thread runs
  - You run for a while, now *you* run for a while, ...
- This is called thread scheduling

Thread Programming

- Major Challenge: You cannot make any assumption about thread scheduling
  - Here is an example with C on Linux (no JVM)

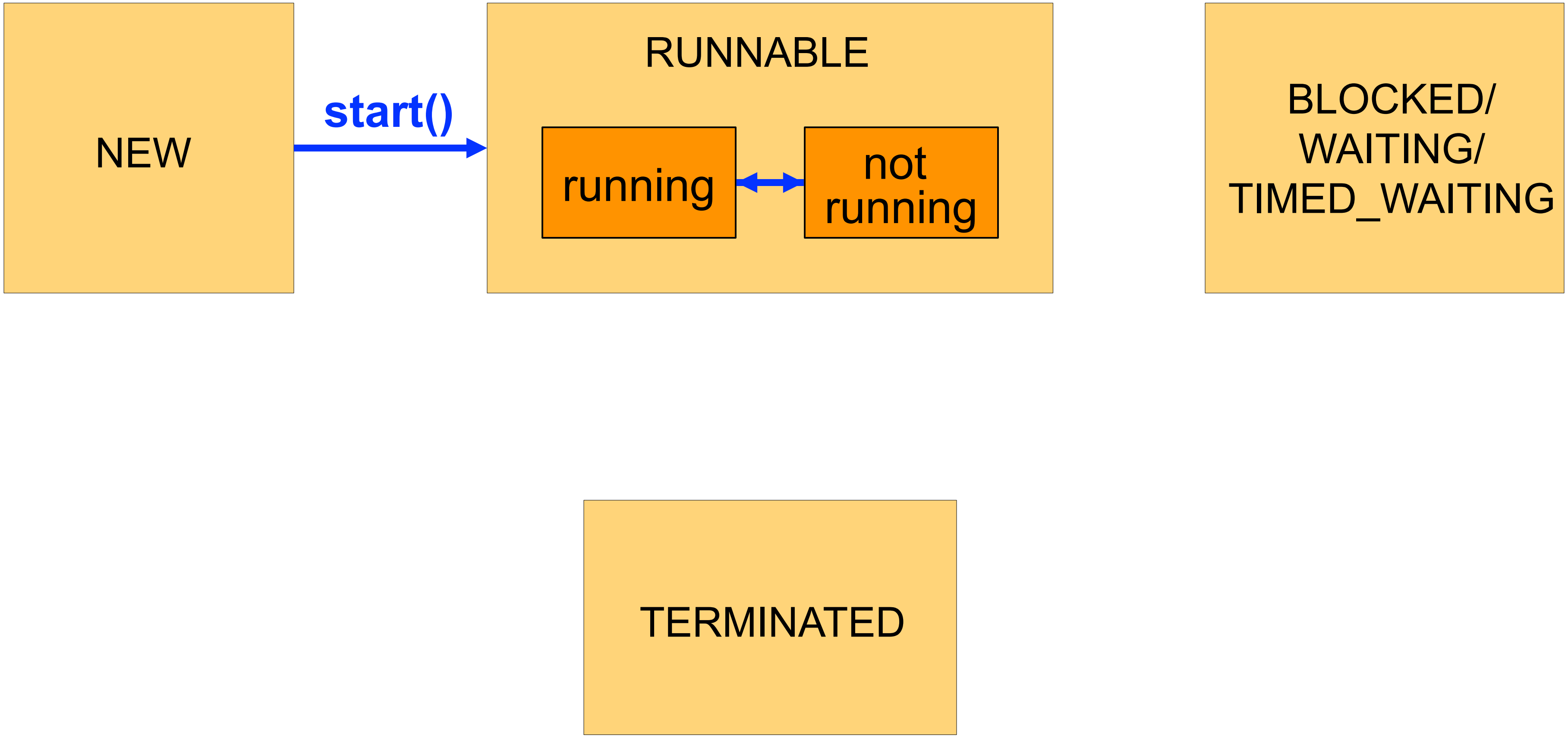


- Major Difficulty: you may not be able to reproduce a bug because each execution is different!

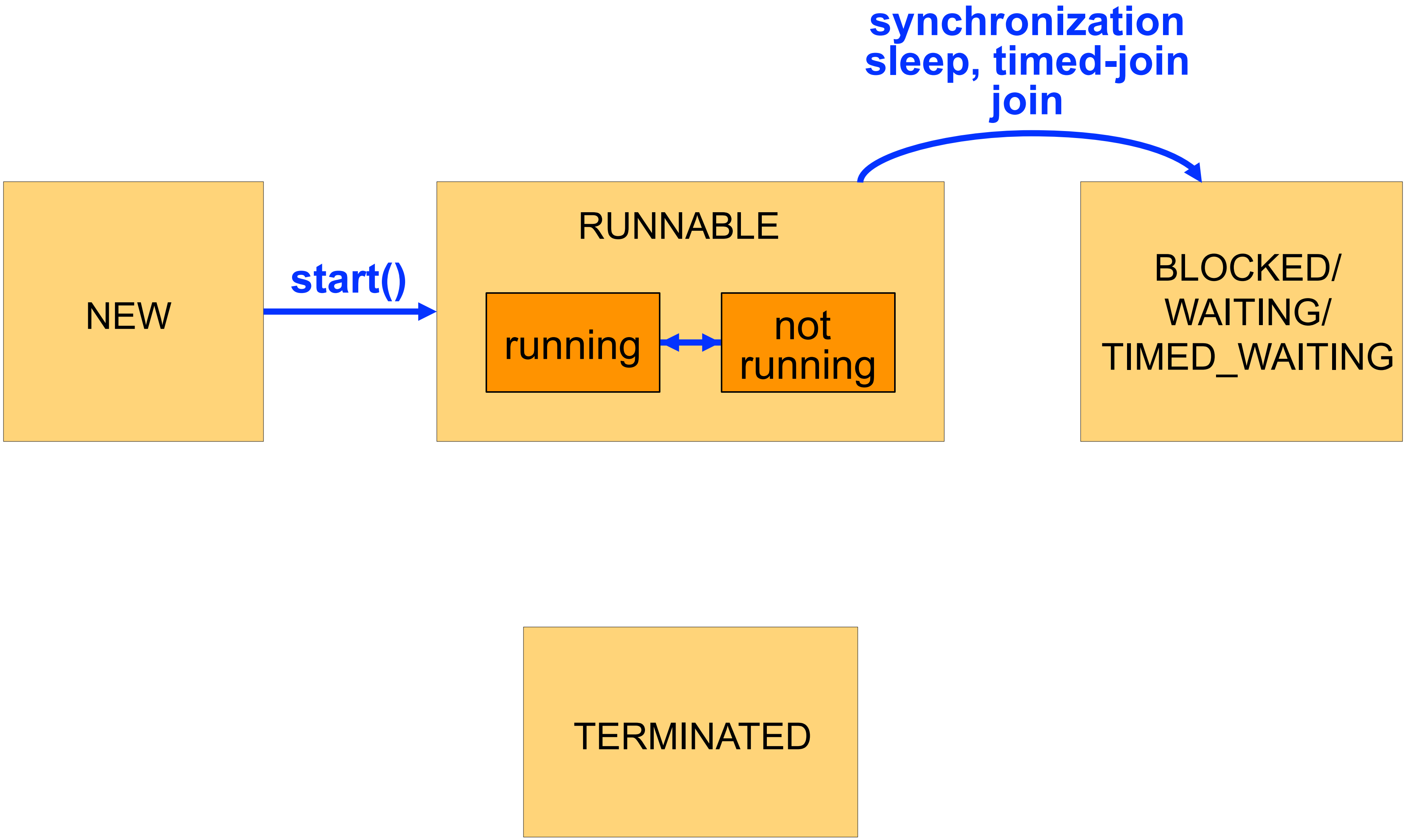
The getState() method

- The possible thread states are
  - NEW: A thread that hasn't been started yet
  - RUNNABLE: The thread can be run, and may be running as we speak
    - It might not because another runnable thread could be running
  - BLOCKED: The thread is blocked on a *monitor*
    - See future lecture
  - WAITING: The thread is waiting for another thread to do something
    - e.g., join()
  - TIMED_WAITING: The thread is waiting for another thread to do something, but will give up after a specified time out
    - e.g., join()
  - TERMINATED: The thread's run method has returned

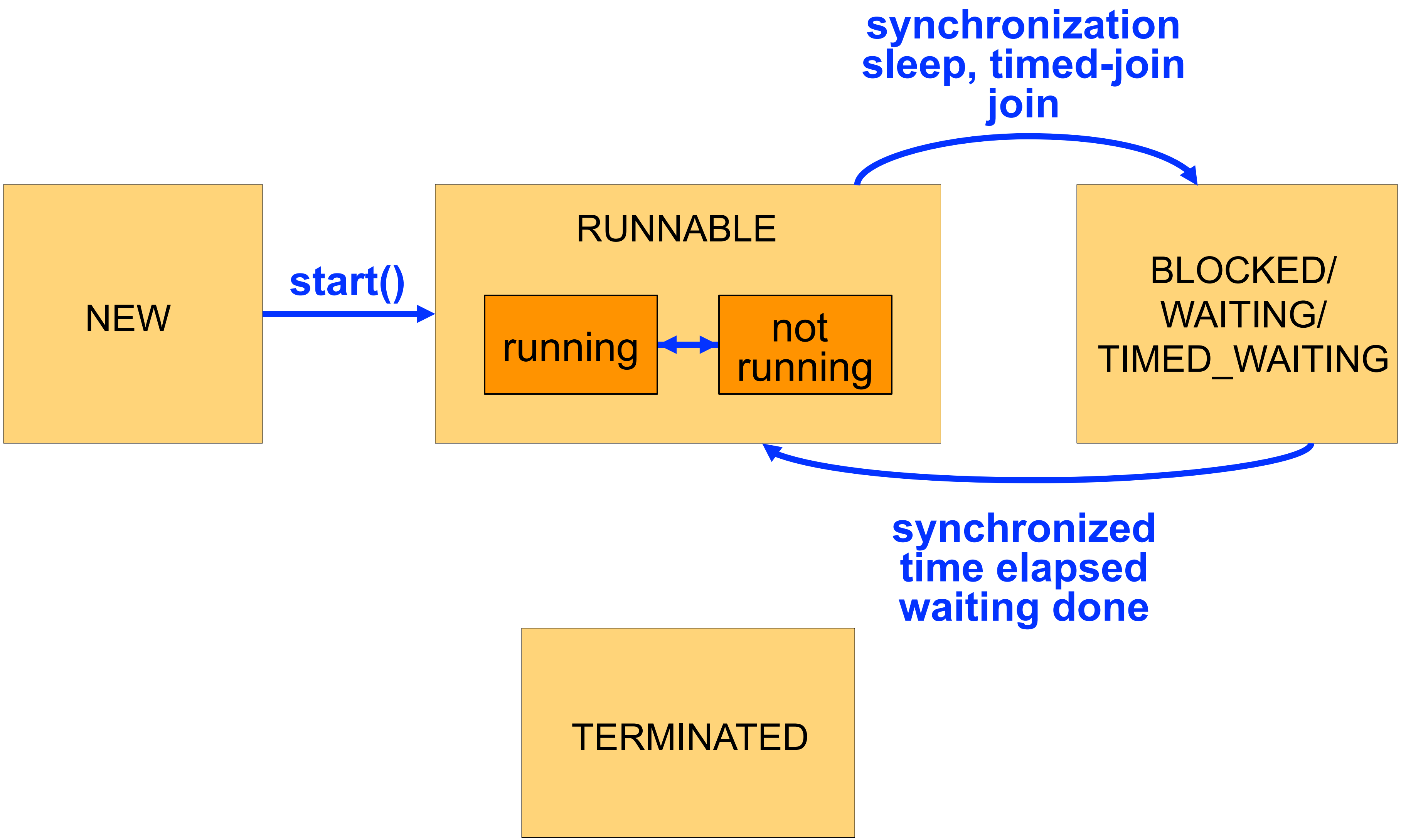
Thread Lifecycle: 4 states



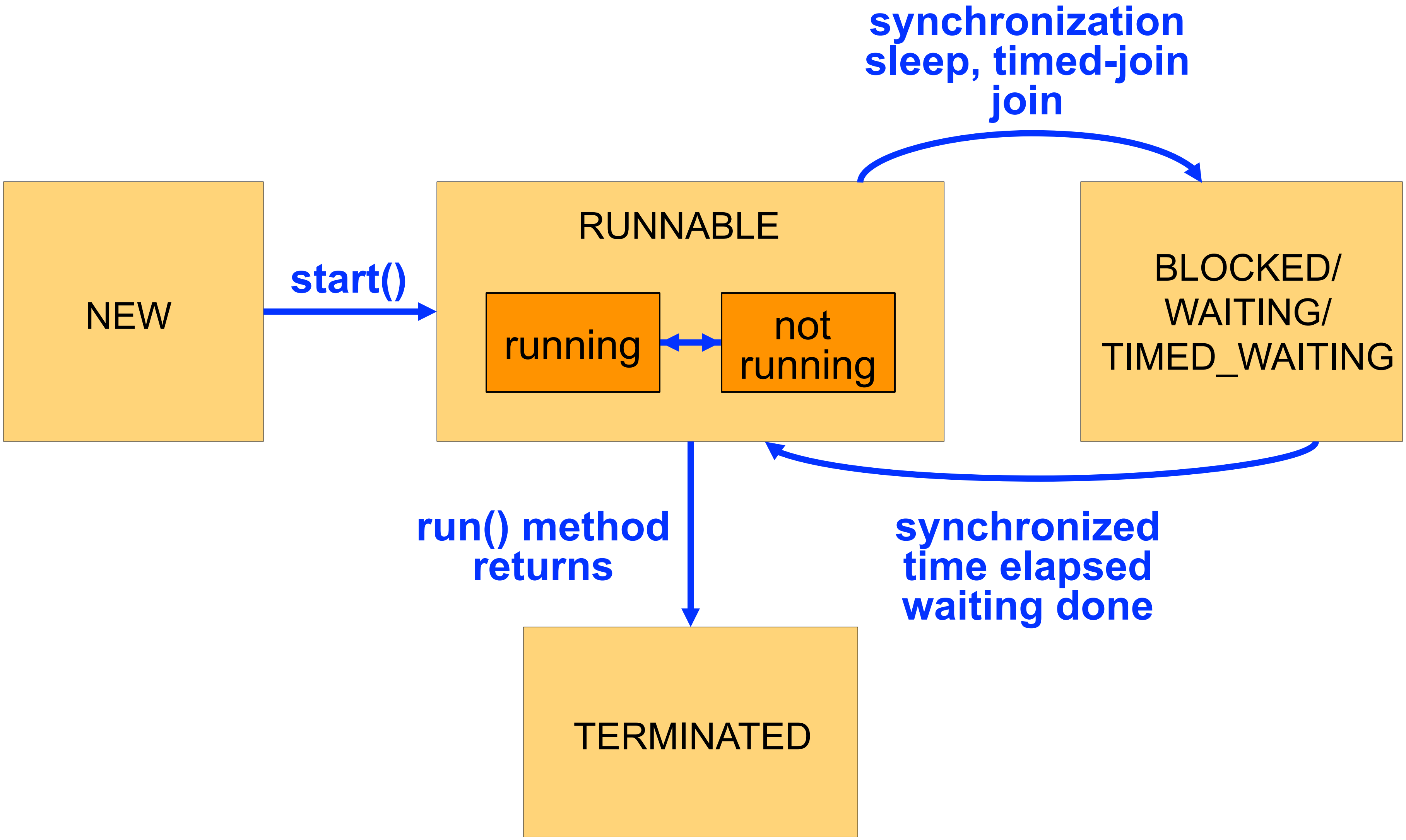
Thread Lifecycle: 4 states



Thread Lifecycle: 4 states

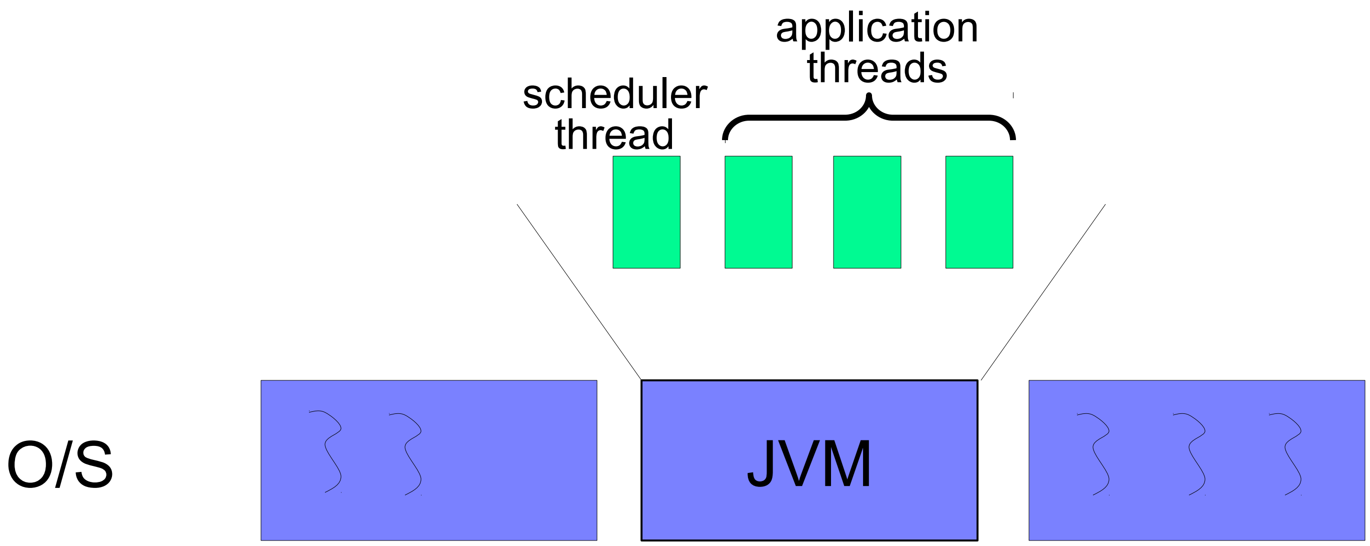


Thread Lifecycle: 4 states



Thread Scheduling

- The JVM keeps track of threads, enacts the thread state transition diagram
- Question: who decides which runnable thread to run?
- Old versions of the JVM used only Green Threads
  - User-level threads implemented by the JVM
  - Invisible to the O/S

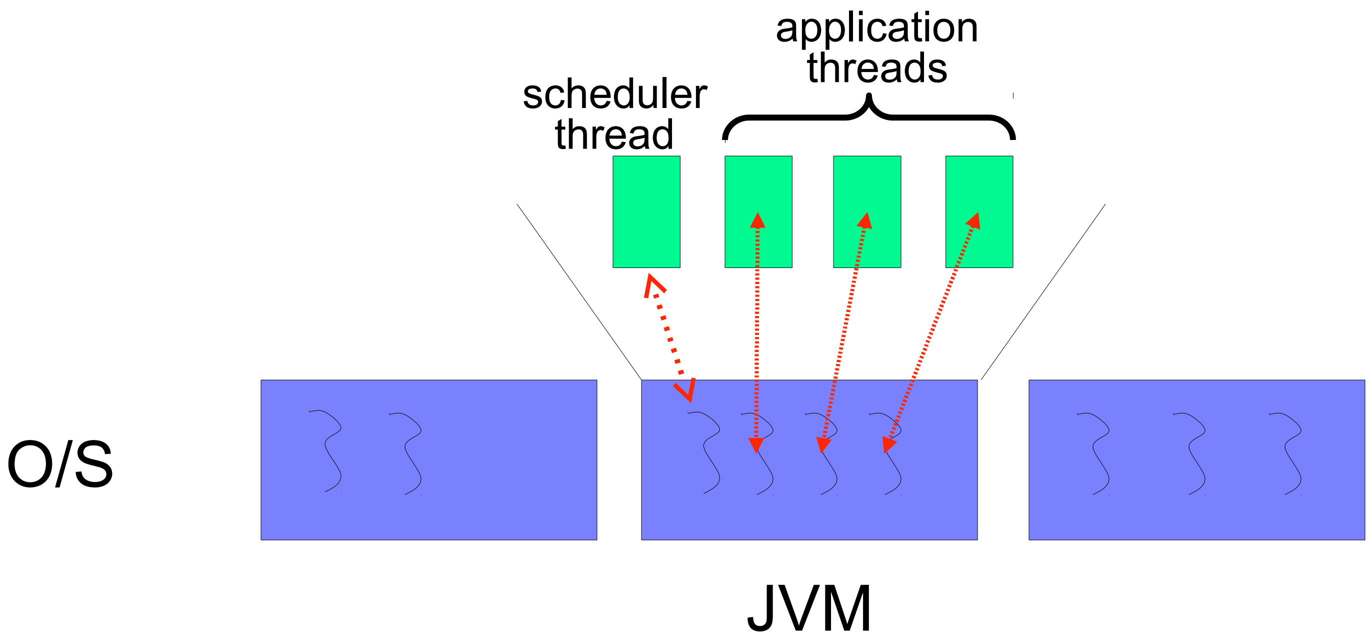


Beyond Green Threads

- Green threads have all the disadvantages of user-level threads (see earlier)
  - Most importantly: Cannot exploit multi-core, multi-processor architectures
- The JVM now provides native threads
  - Green threads are typically not available anymore (in Java)
  - you can try to use “java -green” and see what your system says
- Languages using green threads: Erlang, go...

Java Threads / Kernel Threads

- In modern JVMs, application threads are *mapped* to kernel threads

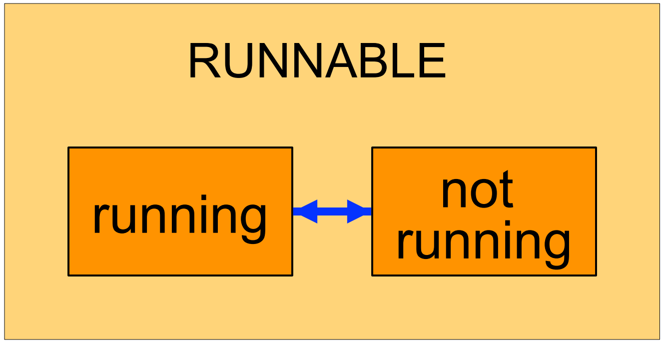


Java Threads / Kernel Threads

- This gets a bit complicated
  - The JVM has a thread scheduler for application threads, which are mapped to kernel threads
  - The O/S also schedules kernel threads
  - Several application threads could be mapped to the same kernel thread!
- The JVM is itself multi-threaded!
- We have threads everywhere
  - Application threads in the JVM
  - Kernel threads that run application threads
  - Threads in the JVM that do some work for the JVM
- Let’s look at a running JVM for a program that runs nothing but an infinite loop...

So what?

- At this point, it seems that we throw a bunch of threads in, and we don't really know what happens
- To some extent it's true, but we have ways to have some control
- In particular, what happens in the RUNNABLE state?



- Can we control how multiple RUNNABLE threads become running or not running?

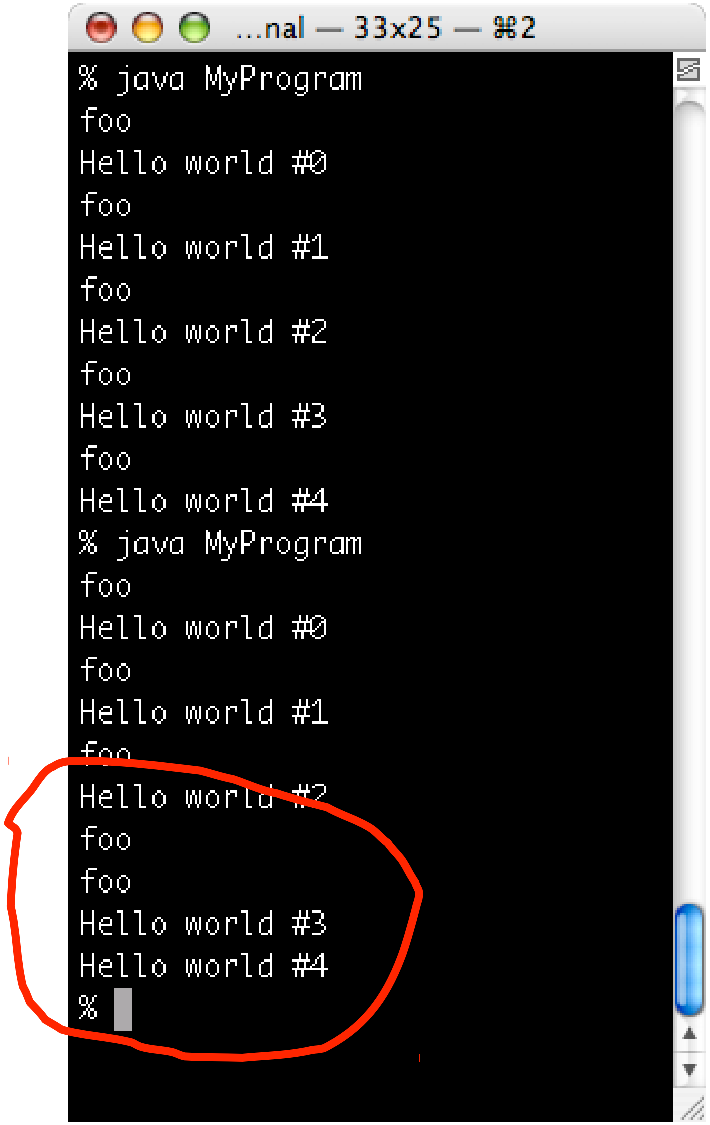
The yield() method: example

- With the yield() method, a thread will pause and give other RUNNABLE threads the opportunity to execute for a while

```
public class MyThread extends Thread {
    public void run() {
      for (int i=0; i<5; i++) {
          System.out.println(“Hello world #“+i);
          Thread.yield();
      }
    }
}


public class MyProgram {
  public MyProgram() {
     MyThread t = new MyThread();
      t.start();
      for (int i=0; i<5; i++) {
          System.out.println(“foo”);
          Thread.yield();
      }
  }
  public static void main(String args[]) {
     MyProgram p = new MyProgram();
  }
}
```

Example Execution



- The use of yield made the threads' executions more interleaved
  - Switching between threads is more frequent
- But it's still not deterministic!
- Programs should NEVER rely on yield() for correctness
  - yield() is really a "hint" to the JVM

Thread Priorities

- The Thread class has a setPriority() and a getPriority() method
  - A new Thread inherits the priority of the thread that created it
- Thread priorities are integers ranging between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY
  - The higher the integer, the higher the priority

What will happen to my threads?

- The Java programmer can give hints to the JVM about what the threads should share CPU resources
- The JVM implements various scheduling policies, that look like those in the Kernel
  - See next set of lecture notes
- The JVM provides hints to the kernel about how the threads should share CPU resources
- The kernel implements possibly complex scheduling policies
- In the end
  - The programmer tries to influence the JVM
  - The JVM tries to influence the kernel
  - The Kernel ultimately decides
- Conclusion: you can never know exactly how your threads will share CPU resources
  - Hence non-deterministic executions

The join() method

- The join() method causes a thread to wait for another thread's termination
- This is useful for "dispatching" work to a worker thread and waiting for it to be done
- Example:

```
Thread t = new MyThread();
 t.start();
 ...
 try { t.join(); } catch (InterruptedException e) { ... }
...
```

The Runnable Interface

- What if you want to create a thread that extends some other class?
  - e.g., a multi-threaded applet is at the same time a Thread and an Applet
- Before Java8, Java did not allow for multiple inheritance
- Which is why it has the concept of interfaces
- So another way to create a thread is to have runnable objects
- It's actually the most common approach
  - Allows to add inheritance in a slightly easier way after the fact
- Let's see this on an example

Runnable Example

```
public class RunnableExample {

  class MyTask implements Runnable {
      public void run() {
         for (int i=0; i<50; i++)
             System.out.print("#");
      }
  }
  public RunnableExample() {
     Thread t = new Thread(new MyTask());
      t.start();
      for (int i=0; i<50; i++)
          System.out.println(".");
  }
  public static void main(String args[]) {
     RunnableExample p = new RunnableExample();
  }
}
```

Extends vs. Implement?

- We have seen two options:
- Option #1: “extends Threads”
- Option #2: “implements Runnable”

- Almost always, option #2 above is preferable since you never know when you'll have to extend a class
- Most Java APIs and documentation talk about “Runnable objects”
- For this class it's up to you, but I suggest sticking to “implements Runnable”
- **2016 update :) BETTER: implements Callable<V>**

Safe Thread Cancellation

- One potentially useful feature would be for a thread to simply terminate another thread
- Two possible approaches:
  - Asynchronous cancellation
    - One thread terminates another immediately
  - Deferred cancellation
    - A thread periodically checks whether it should terminate
- The problem with asynchronous cancellation:
  - may lead to an inconsistent state or to a synchronization problem if the thread was in the middle of “something important”
  - Absolutely terrible bugs lurking in the shadows
- The problem with deferred cancellation: the code is cumbersome due to multiple cancellation points
  - should I die? should I die? should I die?
- In Java, the Thread.stop() method is deprecated, and so cancellation has to be deferred

Java Thread Recap

- Two ways to create threads
  - extends Thread
  - implements Runnable / Callable
- You should never just “kill” a thread
  - Instead have the thread ask “should I die now?” regularly
- The book has a entire Java example you should study (fig. 4.12)

- **Many** more fascinating “features” (ICS432)

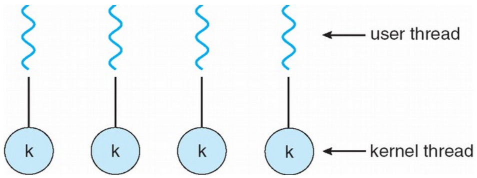
Signals

- We've talked about signals for processes
  - Signal handlers are either default or user-specified
  - signal() and kill() are the system calls
- In a multi-threaded program, what happens?
- Multiple options
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals
- Most UNIX versions: a thread can say which signals it accepts and which signals it doesn't accept
- On Linux: dealing with threads and signals is tricky but well understood with many tutorials on the matter and man pages
  - man pthread_sigmask
  - man sigemptyset
  - man sigaction

Fork()

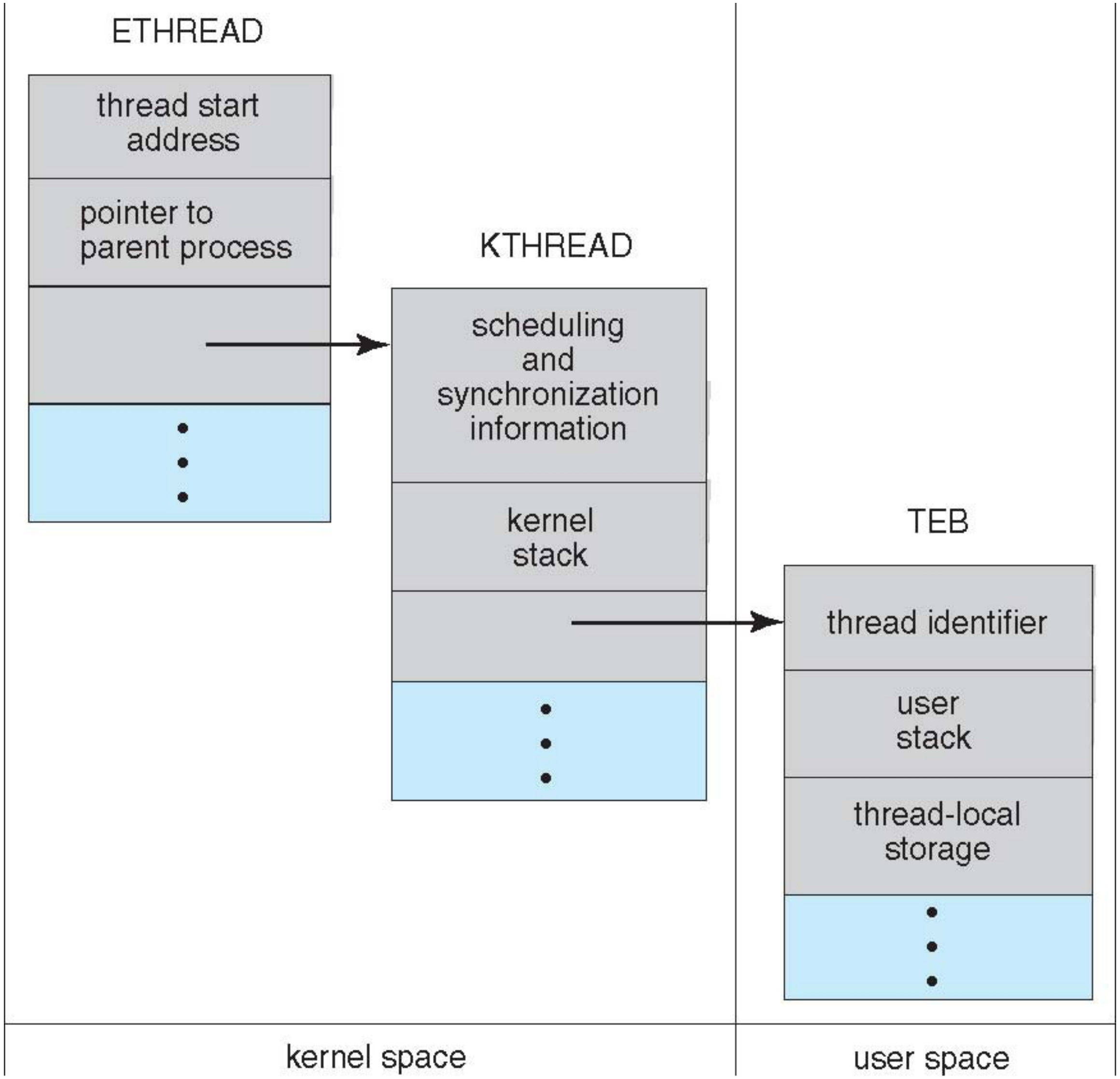
- What happens when a thread calls fork()?
- Two possibilities:
  - A new process is created that has only one thread (the copy of the thread that called fork()), or
  - A new process is created with all threads of the original process (a copy of all the threads, including the one that called fork())
- Some OSes provide both options
  - In Linux the first option above is used
- If one calls exec() after fork(), all threads are “wiped out” anyway

Win XP Threads



- Win XP uses one-to-one mapping
  - Many-to-Many via a separate library
- A thread's defined by its context
  - An ID
  - A register set
  - A user stack and a kernel stack
    - For user mode and kernel mode
  - A private storage area for convenience
- The OS keeps track of threads in data structures, as see in the following figure

Win XP Threads



Linux Threads

- Linux does not distinguish between processes and threads: they're called **tasks**
  - Kernel data structure: task_struct
- The clone() syscall is used to create a task
  - Allows to specify what the new task shares with its parent
  - Different flags lead to something like fork() or like

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Conclusion

- Threads are something you cannot ignore today
  - Multi-core programming
- Programming with threads is known to be difficult, and a lot of techniques/tools are available
- In this course we focus more on how the OS implements threads than how the user uses threads