ECS518U - Operating Systems
Week 9

Concurrency & Threads

Tassos Tombros

Outline

- **Concurrency**
  - What is it / why learn it
- **Threads**
  - Processes vs. threads
- **Interleaving** model of concurrency
  - Problems to solve (links to following weeks)
- **Threads in Java**
  - Thread creation
  - Example based on Lab 8
- **Reading**
  - **Tanenbaum:** Section 2.2
  - **Stallings:** Chapter 4 (but mostly 4.1-4.3)
- Java concurrency tutorial: http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Things you will learn today

- What are **threads** and how they relate to processes
- **Why** do we really want to use threads?
- What we mean by '**concurrency**'
  - Plus a quick preview to some of the programming issues we now face
- Two ways to **create threads in Java**
- Plus a preview of implementation issues for **Lab 8** (Threads in Java, non assessed)

Recap: Processes

- An **instance** of a program running on a computer
- **Why** use processes?
  - Maximise utilisation (I/O slow)
  - Support multi-tasking
  - Protection (process vs. process, process vs. OS)
- Processes are **scheduled** by the OS
  - Time-sharing the CPU
  - Priorities
  - Blocked waiting for I/O
- Processes have **resources**
  - Memory pages, files open, …
- Information that defines the **state** of a process
  - Program counter, registers, stack, …
- Context switch has significant overheads

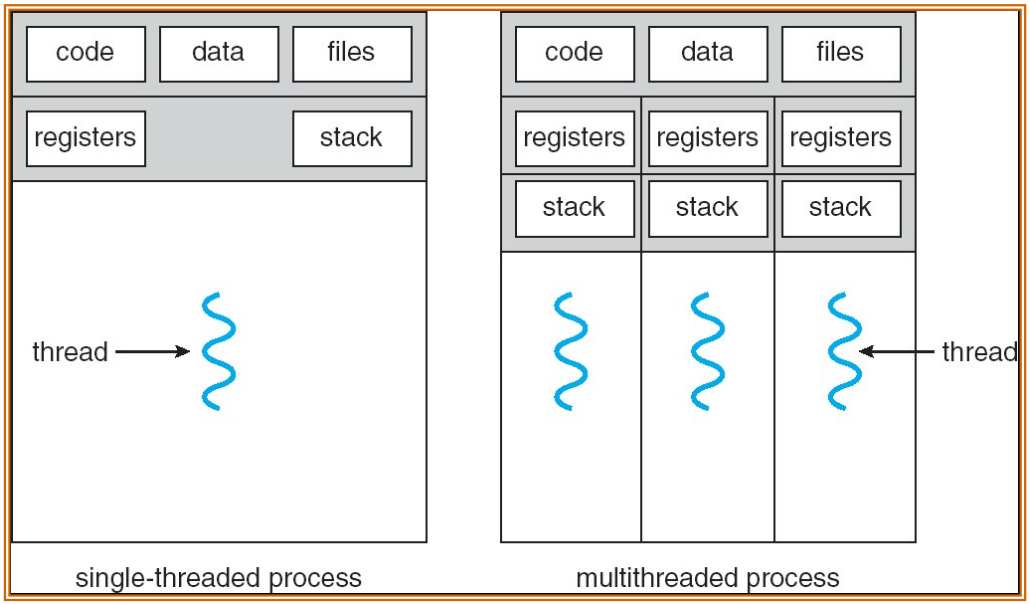
A 'traditional process'

- Process is an OS abstraction to represent what is needed to run a single program
  - No concurrency within a 'traditional' process with a single thread of execution
- Two parts:
  - **Sequential program execution stream**
    - code executed as a sequential stream of execution (i.e. thread)
    - includes state of CPU registers, stack, …
  - **Protected resources:**
    - Main memory state (contents of Address Space)
    - I/O state (i.e. file descriptors)
    - …

A 'modern process' with multiple threads of execution

- **Thread:** a sequential execution stream within a process (also called a "Lightweight process")
  - Process still contains a single Address Space
  - **No protection between threads –** threads share the process's resources
- **Multithreading:** a single program made up of a number of different **concurrent** activities (threads of execution)
- **Why** separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (protection)
  - Heavyweight Process → Process with one thread

Single vs. Multithreaded Processes



State **shared** by all threads in process:

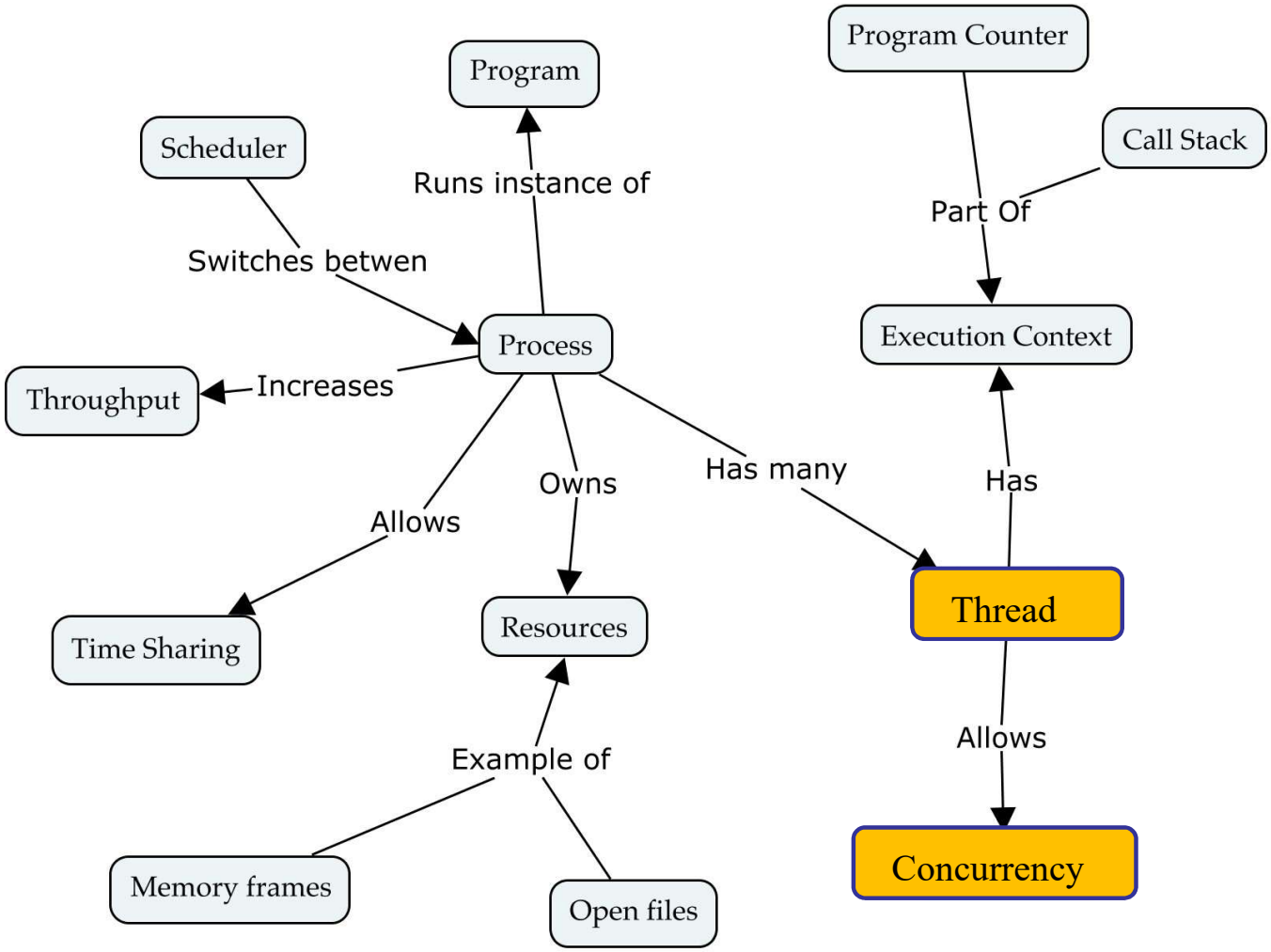
- Content of memory (global variables, heap)
- I/O state (file descriptors, network connections, etc.)

State "**private**" to each thread:

- Kept in **TCB ≡ Thread Control Block**
- CPU registers (including, program counter)
- Execution stack

- Threads encapsulate concurrency
- Address spaces encapsulate protection
  - Keep buggy program from messing up the system (but threads can **VERY** easily mess up each other)

Concept Relationships



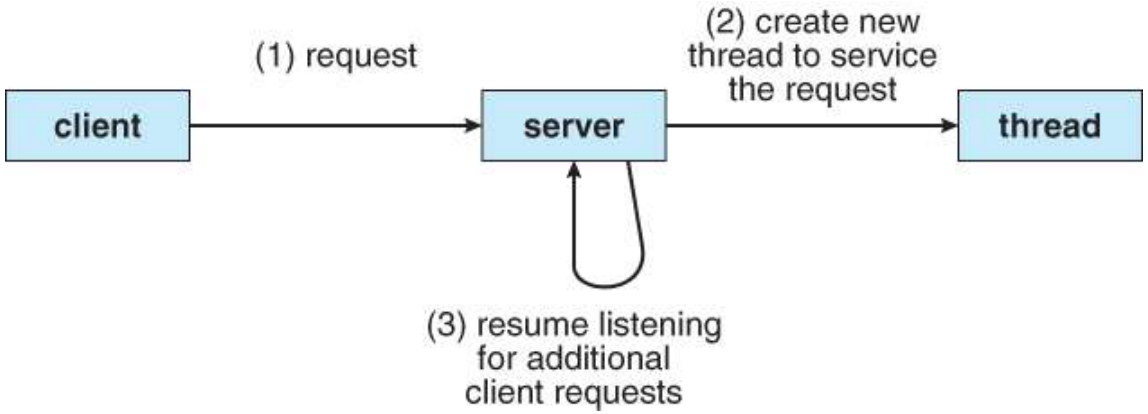
Why Concurrent Programming

- Some problems are best structured as communicating 'tasks' (task → thread)
  - GUIs (Interface must be responsive)
  - Servers (e.g. File servers, Web servers, …)
    - Respond to multiple clients, wait for I/O, etc.
  - Word processors, spreadsheets, etc.
  - **Pretty much everything**
- Exploit multiple processors (cores)
  - Break up a program into multiple tasks to make it go faster
  - e.g. if one task blocks on I/O other tasks in the program should still be able to execute
- **Motivating simple example:**

```
main() {   ComputeLastDigitOfPI();
           PrintECS518UGrades();    }
```

What is the behavior here? Why? (518U grades will never be printed because the method before will never terminate)

More serious example: Web server



Client Browser

- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

Web Server

- thread to get request and issue response
- threads to read data, access DB, etc.

Concurrent Programming and Processes

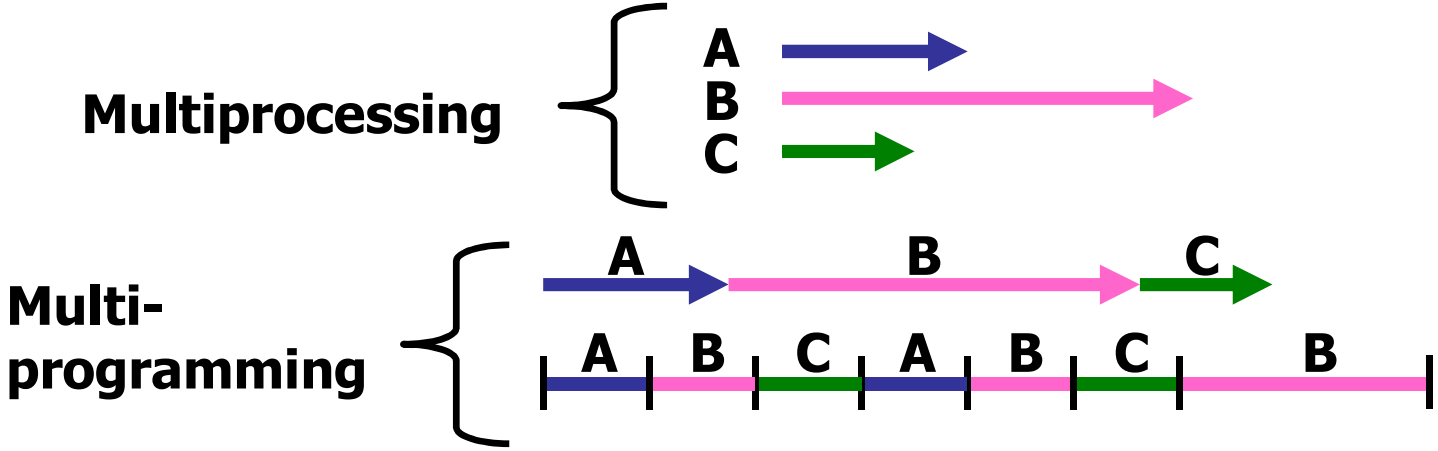
- Why not use processes instead of threads as the 'level of concurrency'?
- Possible but:
  - context switch very **expensive** compared to thread switch (rough figures: context switch: **0.1-1 msecs** Thread switch around **100 ns**)
  - Thread creation much '**lighter**' than process creation
  - Threads naturally **share memory** for communication, for processes we have to make the effort…

User vs. kernel level threads

- **Kernel level threads (KLT)**
  - All modern OSs support KLTs
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
  - **Downside of kernel threads: a bit expensive**
    - Need to make a crossing into kernel mode to schedule
- **User level Threads (ULT)** (Lighter-weight option)
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other
  - **Cheap**
  - **Downside of user threads:**
    - When one thread blocks on I/O, all threads block
    - Kernel cannot adjust scheduling among all threads

Thread interleaving

- Distinguish:
  - **Multiprocessing** → Multiple CPUs
  - **Multiprogramming** → Multiple Jobs or Processes
  - **Multithreading** → Multiple threads per Process
- What does it mean **to run two threads "concurrently"?**
  - Scheduler is free to run threads in any order and **interleaving**: FIFO, Random, …
  - It can choose to run each thread to completion or time-slice in big chunks or small chunks



Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	.....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	.....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	.....	thread is resumed
		y = y + x	.....
		z = x + 5y	z = x + 5y

Threads execute with variable speed

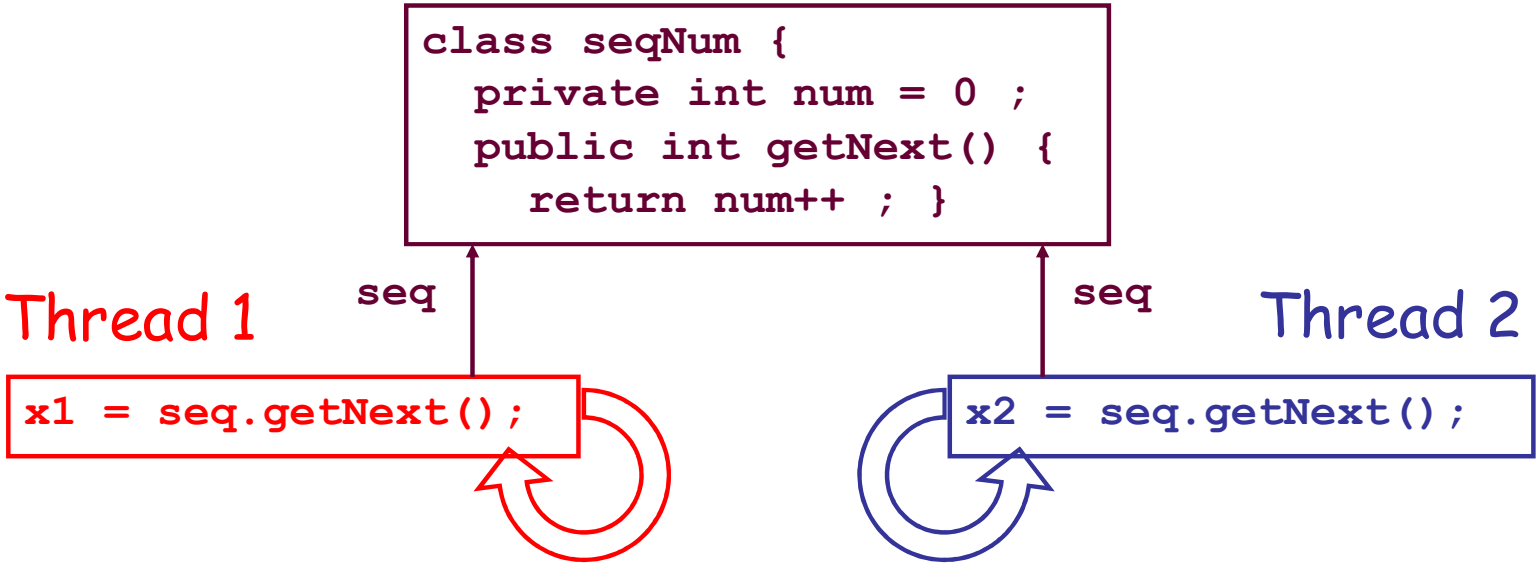
Programs must be designed to work with any schedule

Correctness for systems with concurrent threads

- If threads can be scheduled in any way, programs must work under all circumstances
  - Can we test for this?
  - How can we know if our program works?
- **Independent Threads:**
  - No state shared with other threads
  - Deterministic →Input state determines results
  - Reproducible → Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter
- **Cooperating Threads:**
  - Shared State between multiple threads
  - Non-deterministic, Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called **"Heisenbugs"**
  - **Murphy's law** of concurrent programming for scheduling

An (new type of) Incorrect Program

- Provide a sequence of numbers to different threads
  - no number used more than once



- This program does not work. The two threads could get the same number. **Can you think how?**

OK Interleaving

- Each thread successively reads and writes to seq.num

```
x1 = seq.num
seq.num = seq.num + 1
```

```
x2 = seq.num
seq.num = seq.num + 1
```

OK

Thread	1	x1 = 0
		seq.num = 0 + 1= 1
	2	x2 = 1
		seq.num = 1 + 1= 2
	1	x1 = 2
	2	seq.num = 2 + 1= 3
		x2 = 3
	1	seq.num = 3 + 1= 4
		x1 = 4
		seq.num = 4 + 1= 5

STILL OK

Thread	1	x1 = 0
		seq.num = 0 + 1= 1
	1	x1 = 1
		seq.num = 1 + 1= 2
	1	x1 = 2
	2	seq.num = 2 + 1= 3
		x2 = 3
	1	seq.num = 3 + 1= 4
		x1 = 4
		seq.num = 4 + 1= 5

Problematic Interleaving

- Each thread successively reads and writes to seq.num

```
x1 = seq.num
seq.num = seq.num + 1
```

```
x2 = seq.num
seq.num = seq.num + 1
```

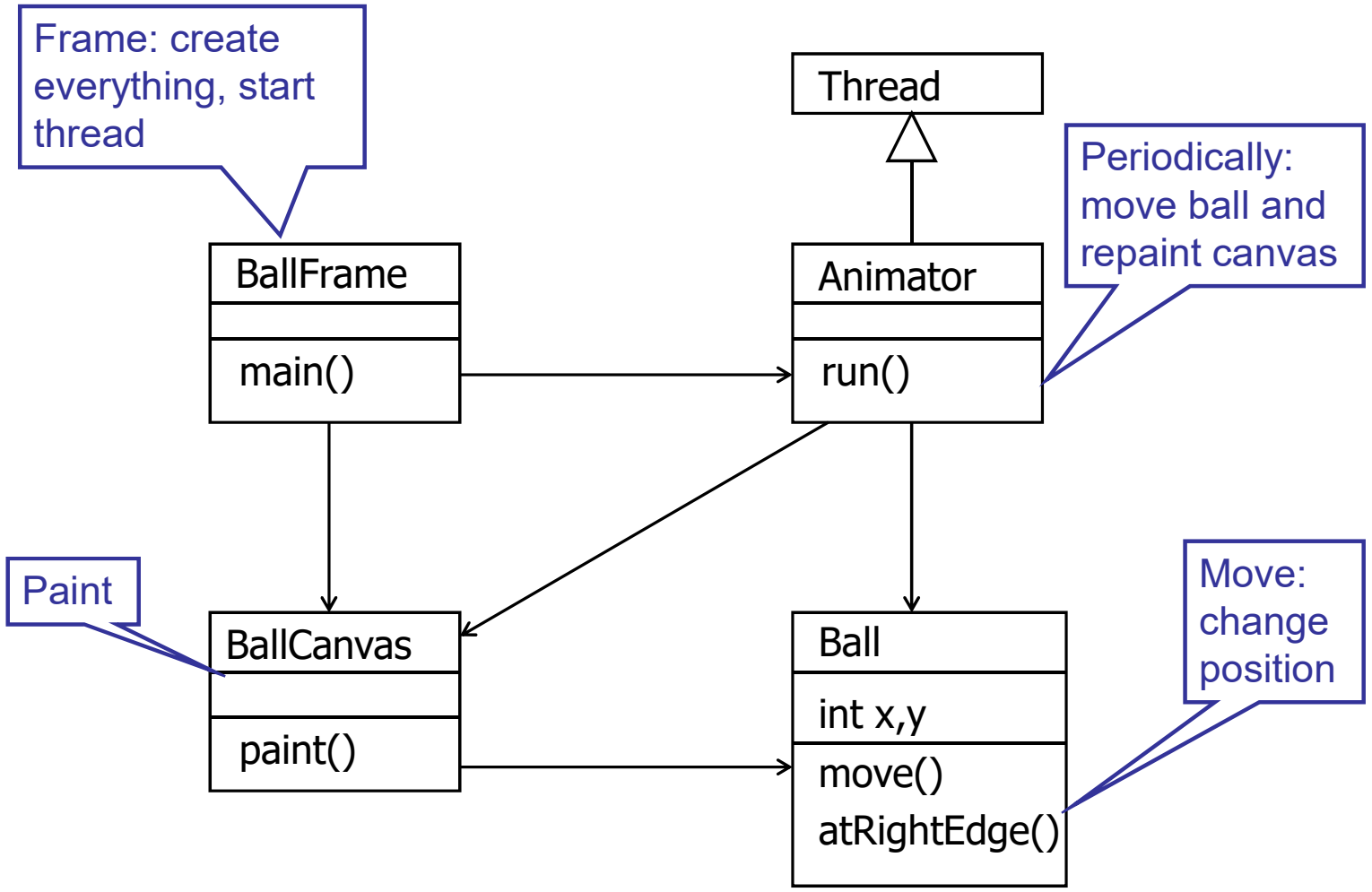
```
x1 = 0
x2 = 0
seq.num = 0 + 1 = 1
seq.num = 0 + 1 = 1
x1 = 1
x2 = 1
seq.num = 1 + 1 = 2
seq.num = 1 + 1 = 2
x1 = 2
seq.num = 2 + 1 = 3
```

NOT OK

Java Threads - Thread Class

- Instances are execution threads
- **Run** method
  - Contains code to be executed in thread
  - **WRITE THIS**
- **Start**
  - Sets thread running
  - **CALL THIS**
- In java.lang

Example: Simple Animation



Example: Simple Animation

- Main – thread 1
  - Create canvas and button, with listener
  - Create ball
  - Create animator
- Animator – thread 2
  - Starts when button pressed
  - Code in run method (call implicitly)
  - Moves the ball
  - Repaints the canvas
  - Sleeps for a bit

Repeatedly, until ball at edge

Creating a Thread Method I: Extend 'Thread'

```
public class Animator extends Thread {
  private Ball ball ;           // reference to ball
  private BallCanvas canvas ;  // reference to canvas

  // Constructor ...

  // Run
  public void run() {
    while (!Thread.interrupted() &&
                                 !ball.atRightEdge()) {
      ball.move() ;
      canvas.repaint() ;
      try {
        Thread.sleep(100) ;
      } catch (InterruptedException e) {}
    }
  }
```

Creating a Thread Method I: Extend 'Thread'

```
public class Animator extends Thread {
  private Ball ball ;           // reference to ball
  private BallCanvas canvas ;   // reference to canvas

  // Constructor ...

  // Run
  public void run() {
    while (!Thread.interrupted() &&
                                !ball.atRightEdge()) {
      ball.move() ;
      canvas.repaint() ;
      try {
        Thread.sleep(100) ;
      } catch (InterruptedException e) {}
    }
  }
```

Thread Operations

- **Start**
  - After a thread has been created, it must be started
  - The 'run' method is invoked
- **Sleep**
  - Suspends the thread for specified number of milliseconds

Methods from Thread

void start()

Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

static void sleep(long millis)

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Starting the Thread:

```
public class BallFrame extends Frame {

  ...

  private Animator animator ; // the animator object
```

- Action listener for the button:

```
public void actionPerformed(ActionEvent e) {
  if (animator == null) {
    animator = new Animator(ball, canvas) ;
    animator.start()
  }
}
```

Creating a Thread - Method II

- **Implement the Runnable interface**
- Convenient when thread class extends something else

```
public class Animator implements Runnable {
  // constructor – as before

  // run method
  public void run() {
    // same code
  }
}
```

Starting the Thread – II

```
public class BallFrame extends Frame {

  private Animator animator ;  // the animator object
  private Thread thread ;      // the animation thread
```

- Step 1 – in the frame constructor

```
  // Create an animator
  animator = new Animator(ball, canvas) ;
```

Starting the Thread – II

```
public class BallFrame extends Frame {

  private Animator animator ;  // the animator object
  private Thread thread ;      // the animation thread
```

- Step 1 – in the frame constructor

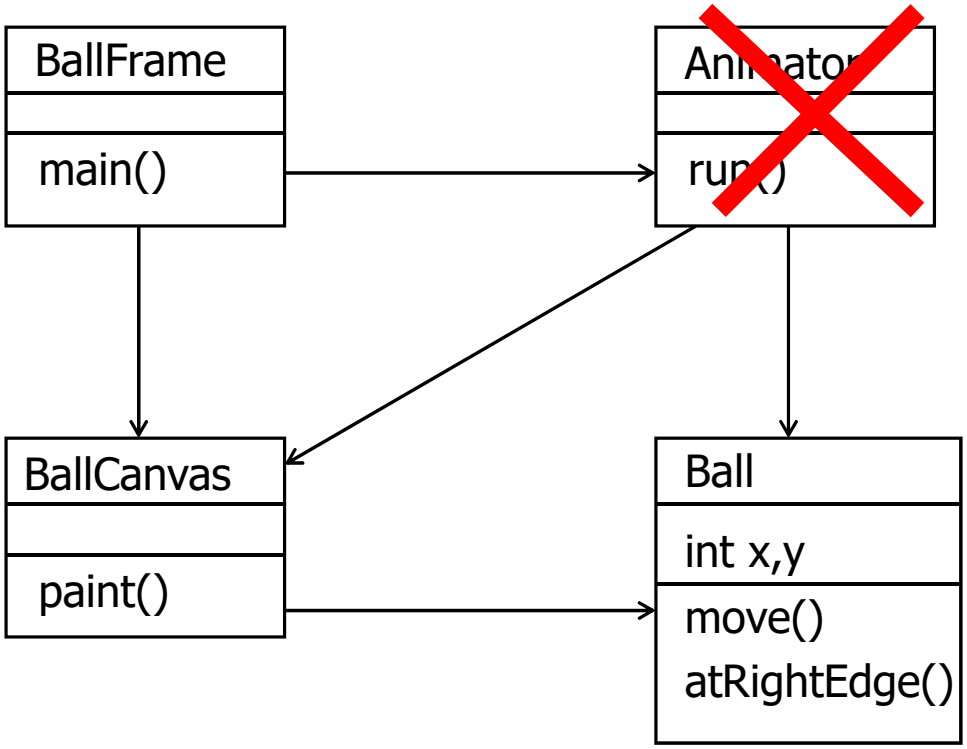
```
  // Create an animator
  animator = new Animator(ball, canvas) ;
```

- Step 2 – in the button's action listener

```
  public void actionPerformed(ActionEvent e) {
    if (thread == null) {
      thread = new Thread(animator) ;
      thread.start() ;
    }
```

Construct the thread

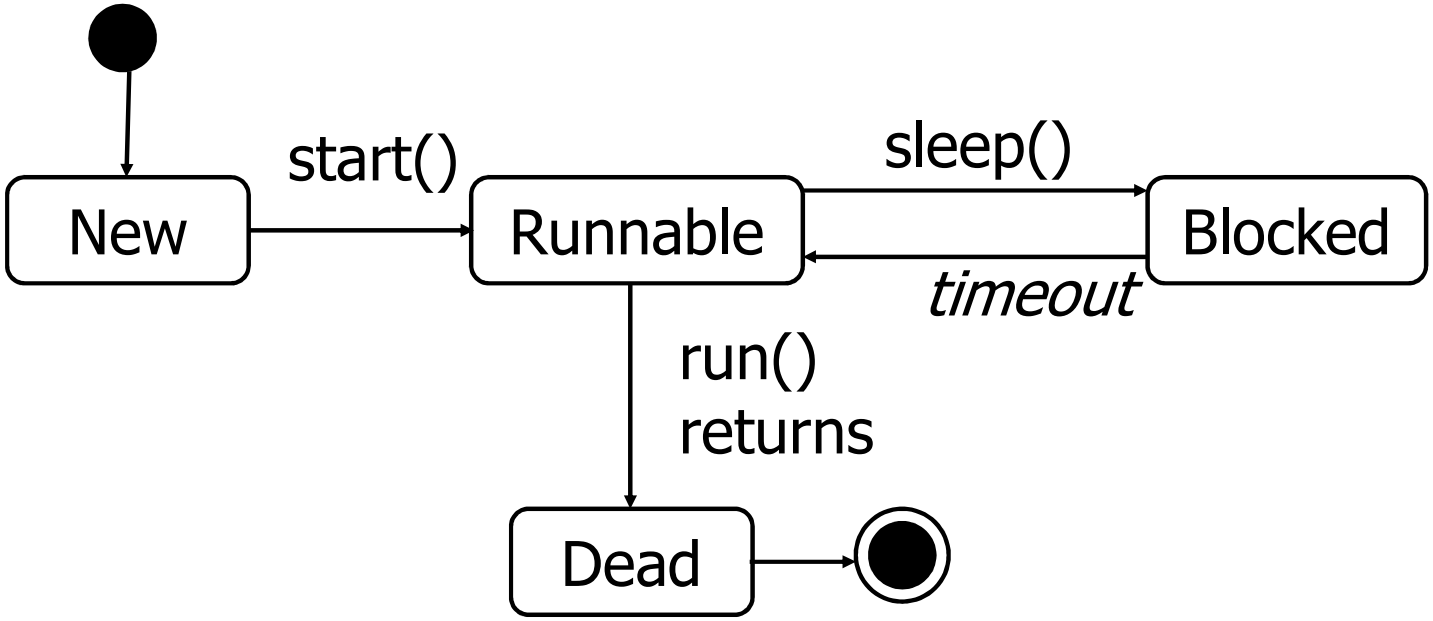
Question: 3 Classes Only?



- Make another class runnable?

Thread State – Simplified

- Lifetime of a thread



- Cannot (re)start a thread once 'dead'
- Multiple runnable and blocked states

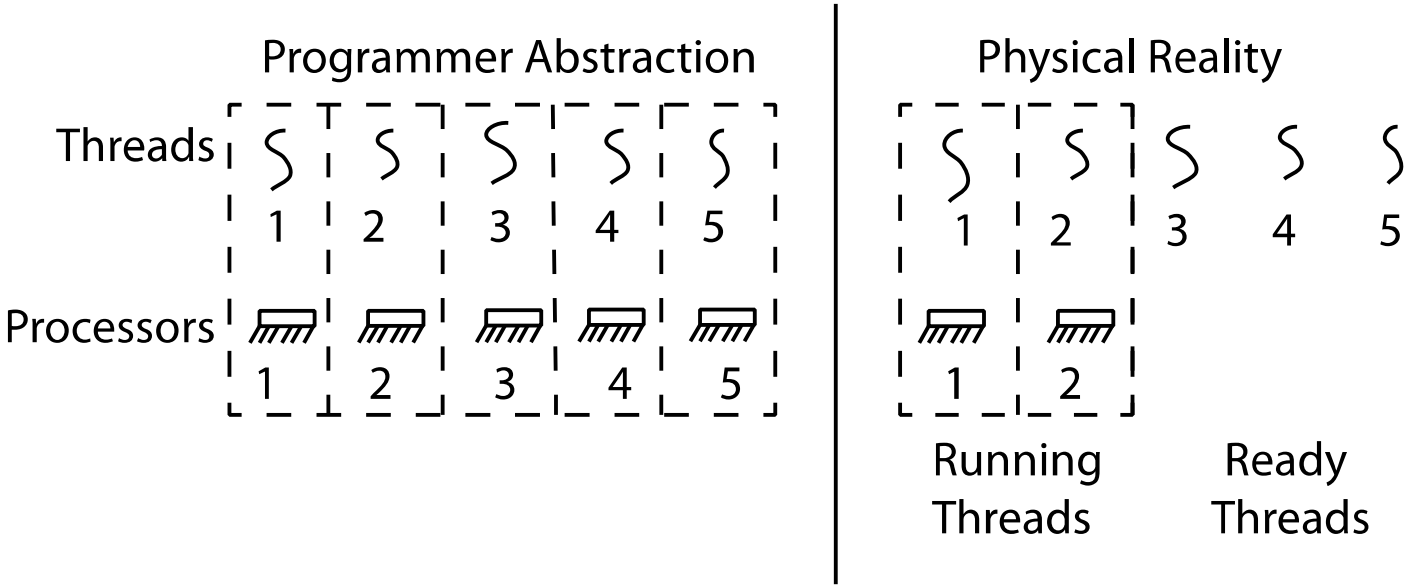
Summary

- **Threads** are light-weight processes
- **Concurrent** programs
  - Necessary in modern computers
  - Introduce new types of errors
- **Interleaving** model of concurrency
  - Real concurrency (multiple cores)
  - Simulated concurrency (time sharing)
- **Java threads** & Lab 8
  - Try the Bouncing Ball example
  - Labs 9 & 10 assume you have done Lab 8 and know the basics of threads in Java
  - Look at: http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Running a thread

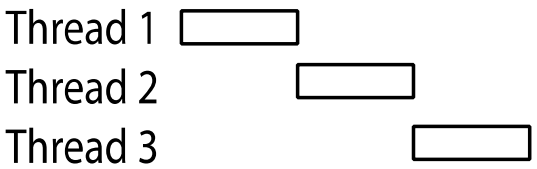
- How **do I run a thread?**
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- **How does the dispatcher get control back?**
  - Blocking on I/O
  - Waiting on a "signal" from other thread
  - Thread executes a yield()
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer interrupt

Thread Abstraction

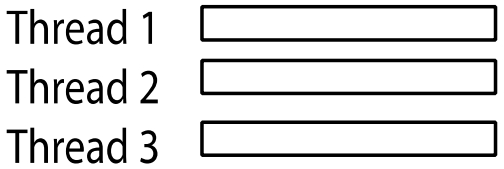


- Infinite number of processors
- Threads execute with variable speed
  - **Programs must be designed to work with any schedule**

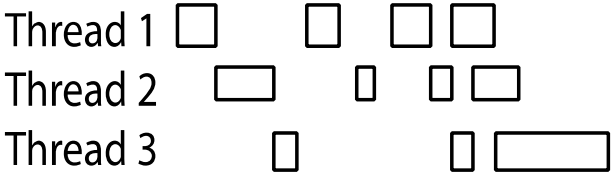
Possible Executions



a) One execution

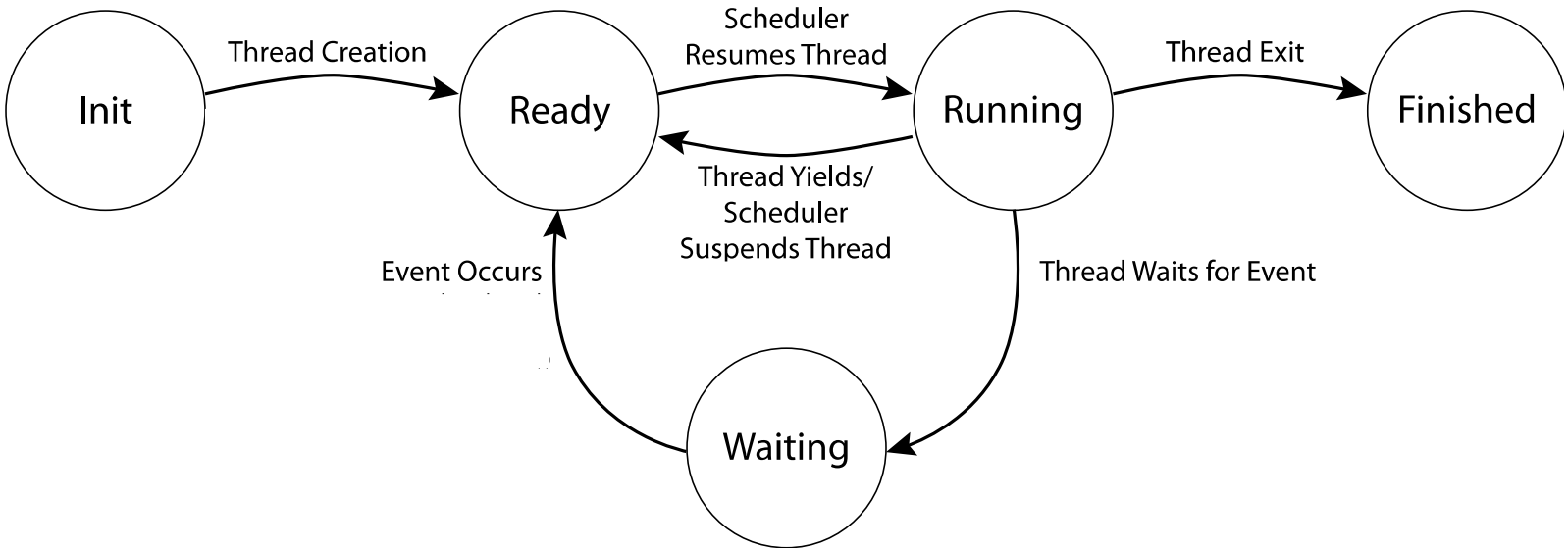


b) Another execution



c) Another execution

Thread Lifecycle

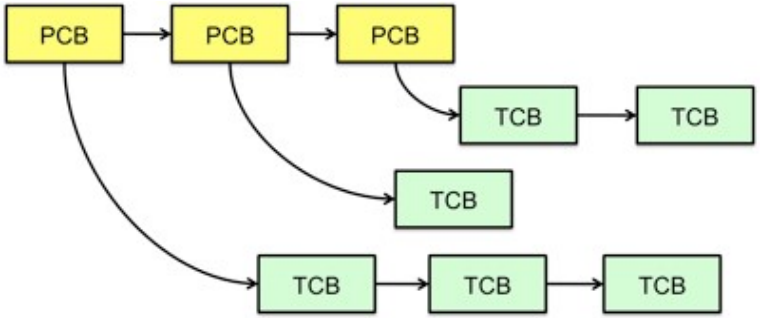


Per Thread State (Kernel Supported Threads)

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Etc.
- OS keeps track of TCBs in "kernel memory"
  - In Array, or Linked List, or …

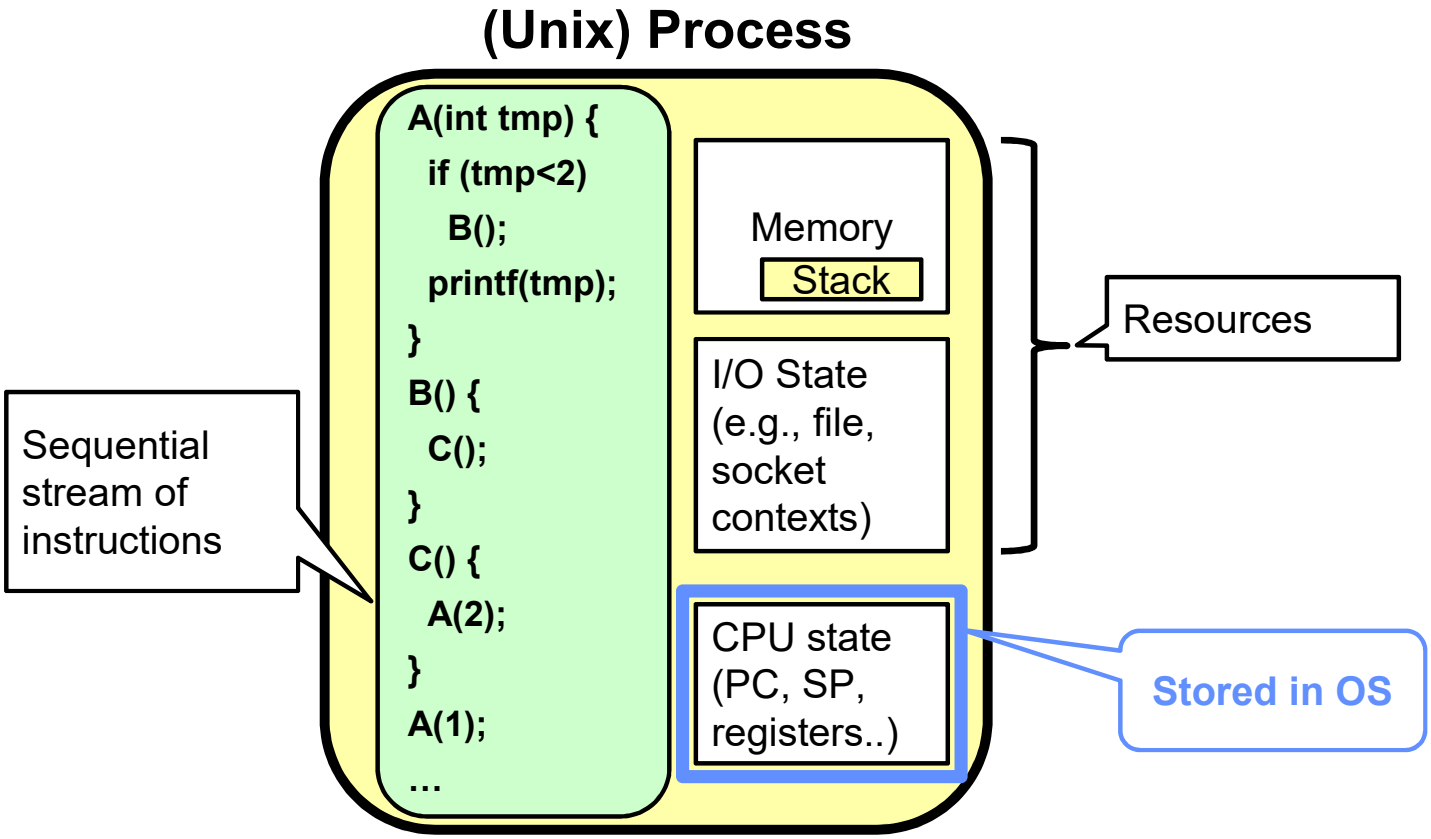
Multithreaded Processes

- PCB points to multiple TCBs:

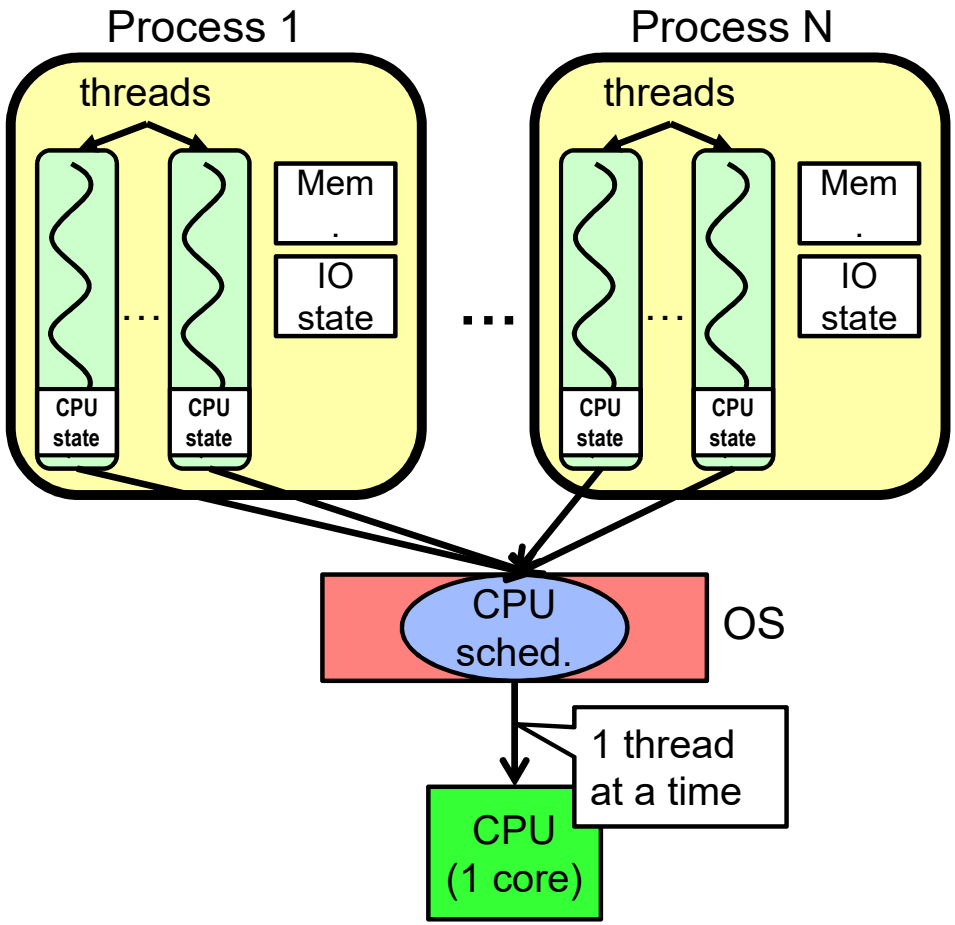


- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables.

Putting it together: Process



Putting it together: Threads



- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory/IO: **No**
- Sharing overhead: **low** (thread switch overhead low)

Putting it together: Multi-Cores