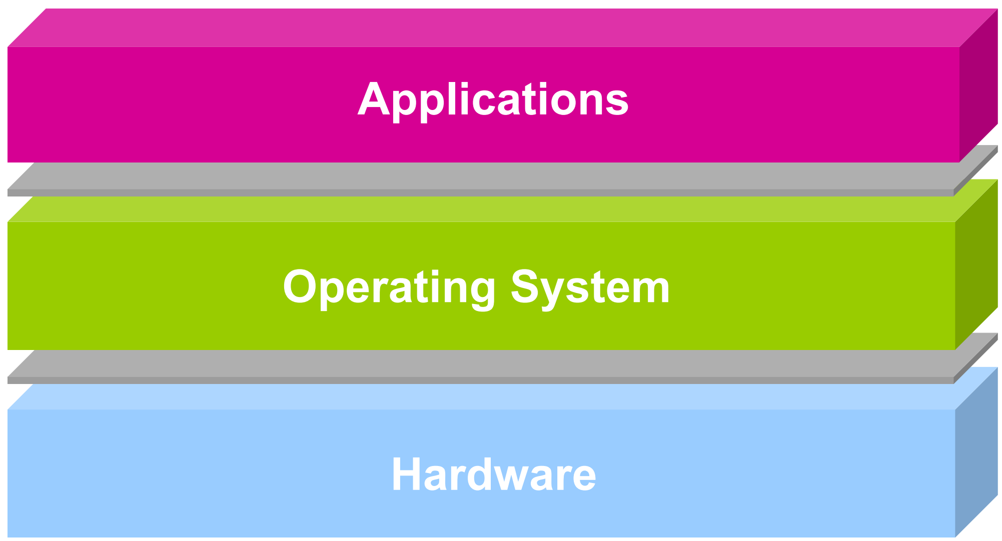
EECS 482
Introduction to Operating Systems

Winter 2018

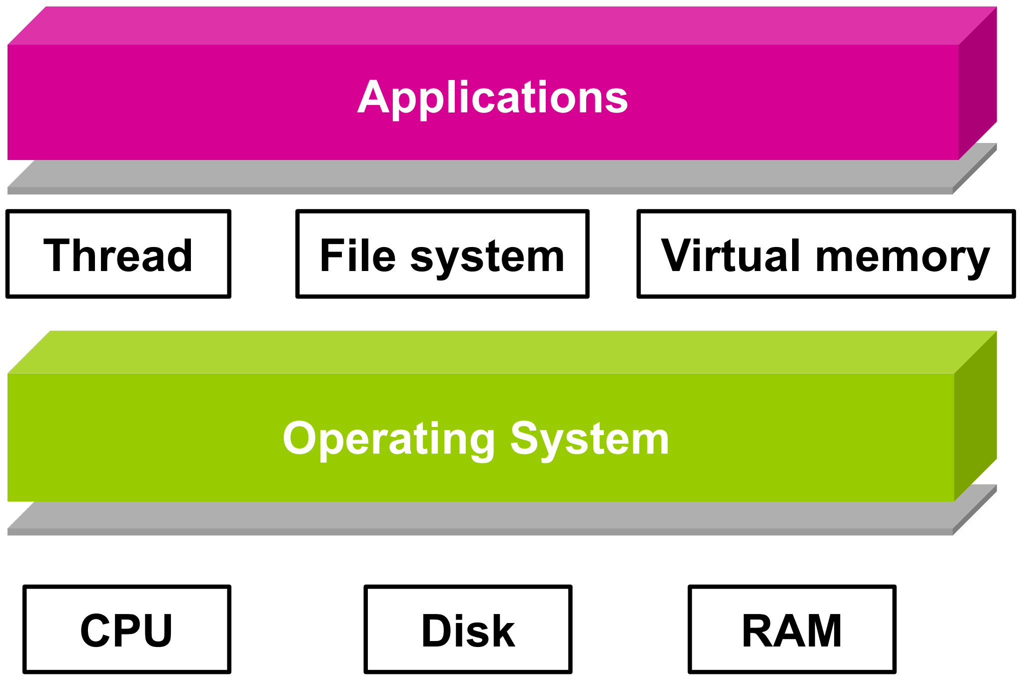
Harsha V. Madhyastha

Recall: What does an OS do?

- Creates abstractions to make hardware easier to use
- Manages shared hardware resources



OS Abstractions



Upcoming Schedule

- This lecture starts a class segment that covers processes, threads, and synchronization
  - Perhaps the most important in this class
  - Basis for Projects 1 and 2

Managing Concurrency

- Recall: Source of OS complexity
  - Multiple users, programs, I/O devices, etc.
  - Originally for efficient use of H/W, but useful even now
- How to manage this complexity?
  - Divide and conquer
  - Modularity and abstraction

```
main() {
    getInput();
    computeResult();
    printOutput();
}
```

The Process

- The process is the OS abstraction for execution
  - Also sometimes called a job or a task
- A process is a program in execution
  - Programs are static entities with potential for execution
- Recall: For each area of OS, ask
  - What interface does hardware provide?
  - What interface does OS provide?

`app1+app2+app3`
`CPU + memory`

`app1` / `CPU + memory`   `app2` / `CPU + memory`   `app3` / `CPU + memory`

Process Components

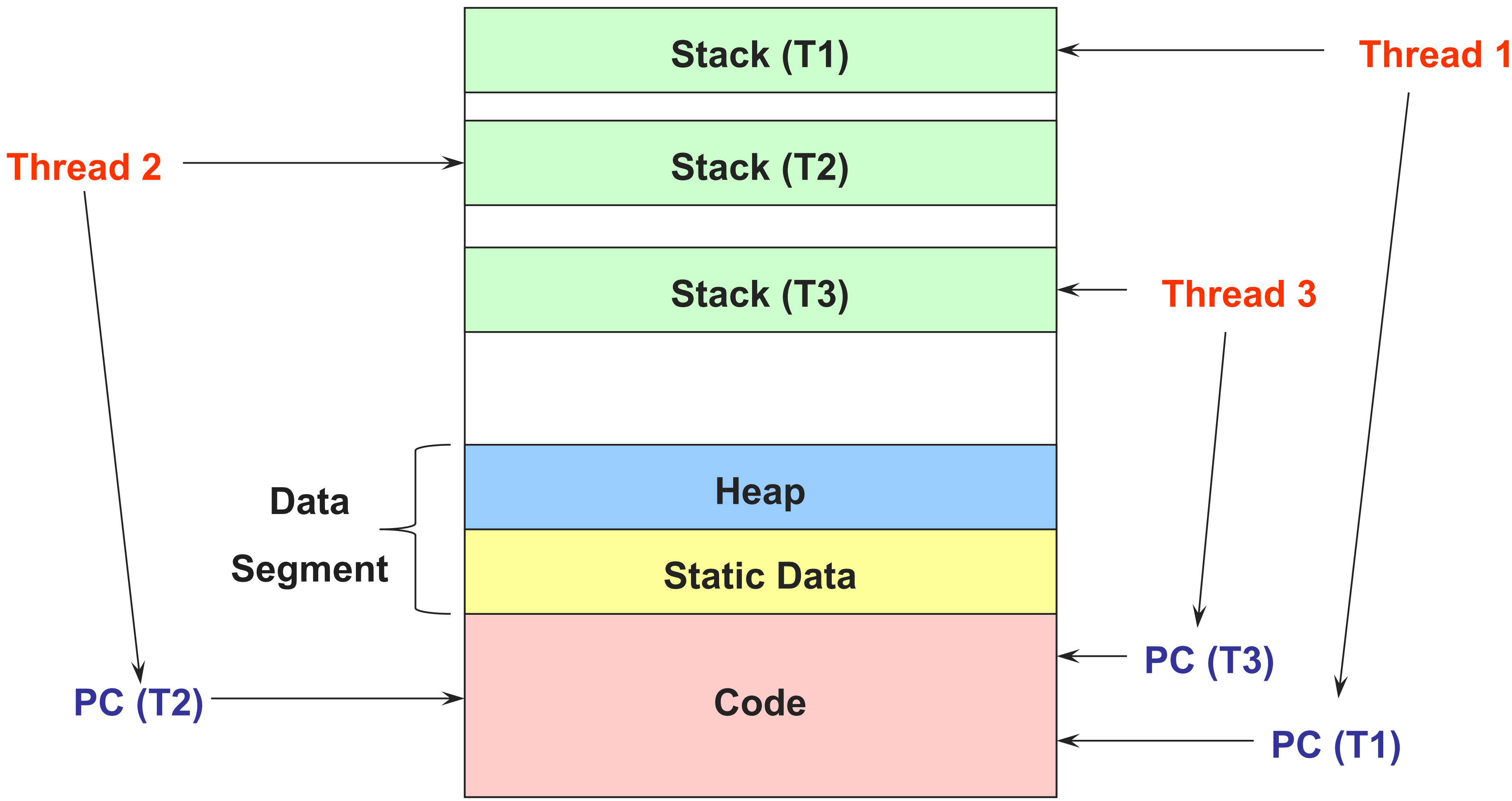
- A process, identified by process ID (PID), is an executing program
  - Set of threads (active)
  - An address space (passive)

- What's in the address space?

Shared across threads
  - » The code for the executing program
  - » The heap memory allocated by the executing program

Private to each thread
  - » An execution stack with local variables, parameters, etc.
  - » The program counter (PC) indicating the next instruction
  - » A set of general-purpose registers with current values

Process Address Space



Review of Stack Frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```

A(tmp = 1)

B(val = 1)

C(foo = 1, bar = 3)

A(tmp = 0)

Multiple Threads

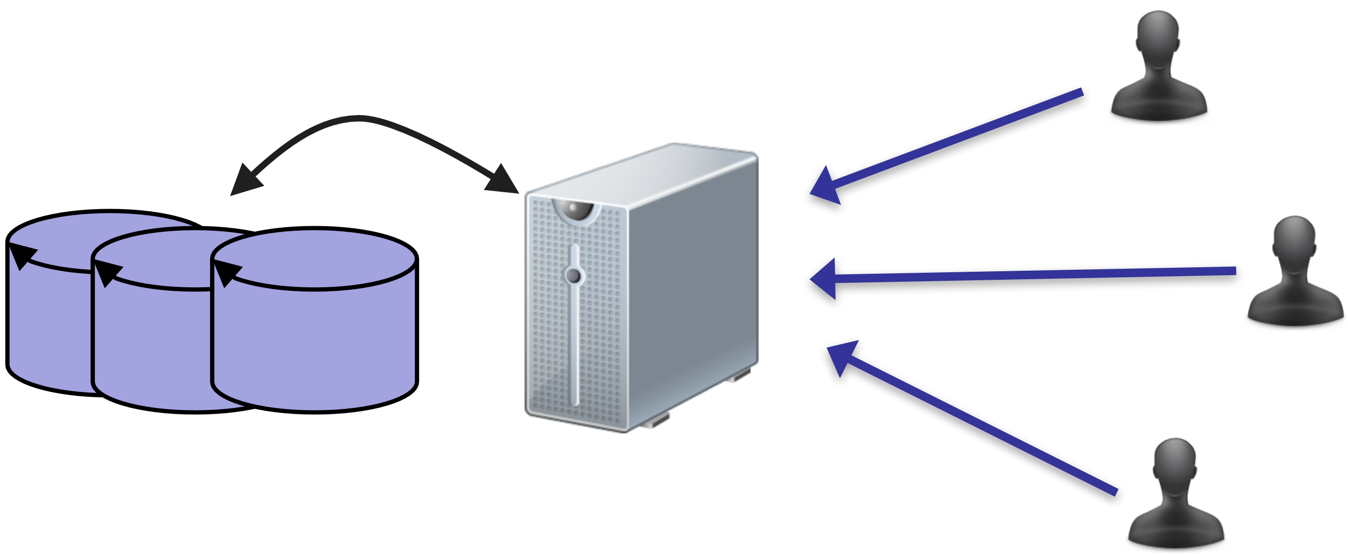
- Which of these is shared between threads?
  - Heap
  - Stack (and SP)
  - PC
  - Code
- Can have several threads in a single address space
  - Sometimes they interact
  - Sometimes they work independently

Upcoming Topics

- Threads: unit of concurrency
  - How multiple threads can cooperate to accomplish a single task?
  - How multiple threads can share limited number of CPUs?
- Address spaces: unit of state partitioning
  - How do address spaces share single physical memory?
    - » Efficiently
    - » Flexibly
    - » Safely

Why do we need threads?

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



Option 1: Handle one request at a time

Request 1 arrives
Server reads in request 1
Server starts disk I/O for request 1
Request 2 arrives
Disk I/O for request 1 finishes
Server responds to request 1
Server reads in request 2

time

- Pros and cons?
- Easy to program, but slow
  - Can't overlap disk requests with computation
  - Can't overlap either with network sends and receives

Option 2: Event-driven web server (asynchronous I/O)

- Issue I/Os, but don't wait for them to complete

  Request 1 arrives

  Server reads in request 1

  Server starts disk I/O for request 1

  Request 2 arrives

  Server reads in request 2

  Server starts disk I/O for request 2

  Disk I/O for request 1 completes

time

- Fast, but hard to program
  - Why?

Option 2: Event-driven web server (asynchronous I/O)

- Issue I/Os, but don't wait for them to complete
  - Request 1 arrives
  - Server reads in request 1
  - Server starts disk I/O for request 1
  - Request 2 arrives
  - Server reads in request 2
  - Server starts disk I/O for request 2
  - Disk I/O for request 1 completes

time

Web server must remember
- **What requests are being served, and what stage they're in**
- **What disk I/Os are outstanding (and which requests they belong to)**

Lots of extra state!

Multi-threaded web server

- One thread per request
  - Thread issues disk (or n/w) I/O, then waits for it to finish
  - Though thread is blocked on I/O, other threads can run
  - Where is the state of each request stored?

Thread 1	Thread 2	Thread 3
Request 1 arrives		
Read in request 1		
Start disk I/O		
	Request 2 arrives	
	Read in request 2	
	Start disk I/O	
		Request 3 arrives
		Read in request 3
Disk I/O finishes		
Respond to request 1		

Benefits of Threads

- Thread manager takes care of CPU sharing
  - Other threads can progress when one thread issues blocking I/Os
  - Private state for each thread
- Applications get a simpler programming model
  - The illusion of a dedicated CPU per thread
- Downsides compared to event-driven model?
  - Efficiency (thread scheduling overhead)

Announcements

- First discussion section this Friday
  - No homework questions
  - Overview of tools and techniques
- Sign up for GitHub and Piazza
- Started putting together project group?
  - Group declaration due in two weeks (Jan 22)
- Bring print out of lecture slides to class
- Speak up when something is unclear

When are threads useful?

- Multiple things happening at once
- Usually some slow resource
  - Network, disk, user, …

- Examples:
  - Controlling a physical system (e.g., airplane controller)
  - Bank ATM server
  - Window system
  - Parallel programming

Ideal Scenario

- Split computation into threads
- Threads run independent of each other
  - Divide and conquer works best if divided parts are independent

How practical is thread independence?

Dependence between threads

- Example 1: Microsoft Word
  - One thread formats document
  - Another thread spell checks document

- Example 2: Desktop computer
  - One thread plays World of Warcraft
  - Another thread compiles EECS 482 project

- Two types of sharing: app resource or H/W
- Example of non-interacting threads?

Cooperating threads

- How can multiple threads cooperate on a single task?
  - Example: Ticketmaster's webserver
  - Assume each thread has a dedicated processor

- Problem:
  - Ordering of events across threads is non-deterministic
  - Speed of each processor is unpredictable

Thread A -->
Thread B -   -   -   -   -   -   -   -   -   >
Thread C ->

- Consequences:
  - Many possible global ordering of events
  - Some may produce incorrect results

Non-deterministic ordering → Non-deterministic results

- Printing example

Thread 1	Thread 2
Print ABC	Print 123

  - Possible outputs?
    - 20 outputs: ABC123, AB1C23, AB12C3, AB123C, A1BC23, A12BC3, A123BC, 1ABC23, 1A2BC3, …
  - Impossible outputs?
    - ABC321

- Ordering within thread is sequential
- Many ways to merge per-thread order into a global order
- What's being shared between these threads?

Non-deterministic ordering → Non-deterministic results

- Arithmetic example (y is initially 10)
  - What's being shared between these threads?
  - Possible results?
    - » If A runs first: x = 11 and y = 20
    - » If B runs first: x = 21 and y = 20

Thread A	Thread B
x = y + 1	y = y * 2

- Another example (x is initially 0)
  - Possible results?
    - » x = 1 or -1
  - Impossible results?
    - » x = 0

Thread A	Thread B
x = 1	x = -1

Thread A	Thread B
x = 0	x = 0
x++	x--

Atomic operations

- Before we can reason at all about cooperating threads, we must know that some operation is **atomic**
  - Indivisible, i.e., happens in its entirety or not at all
  - No events from other threads can occur in between
- Print example:
  - What if each print statement were atomic?
  - What if printing a single character were not atomic?
- Most computers
  - Memory load and store are atomic
  - Many other instructions are not atomic
    - » Example: double-precision floating point
  - Need an atomic operation to build a bigger atomic operation