



Introduction

CS 571: *Operating Systems* (Spring 2021)
Lecture 1

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpacı-Dusseau.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

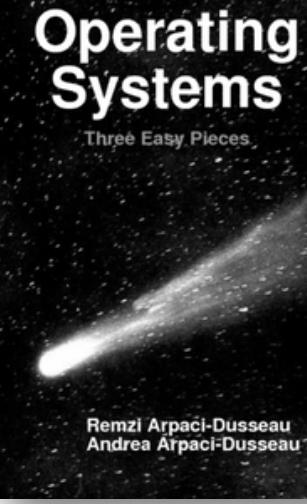
Introduction

- Instructor
 - Dr. Yue Cheng (web: cs.gmu.edu/~yuecheng)
 - Email: yuecheng@gmu.edu
 - Office hours: Wednesday 1:30pm-2:30pm
 - Research interests: Distributed and storage systems, serverless and cloud computing, operating systems

Introduction

- Instructor
 - Dr. Yue Cheng (web: cs.gmu.edu/~yuecheng)
 - Email: yuecheng@gmu.edu
 - Office hours: Wednesday 1:30 – 2:30 pm
 - Research interests: Distributed and storage systems, serverless and cloud computing, operating systems
- Graduate teaching assistant
 - Michael Crawshaw
 - Email: mcrawsha@masonlive.gmu.edu
 - Office hours:
 - Monday 1:30 – 2:30 pm + Thursday 2:30 – 3:30 pm

Administrivia



- Required textbook
 - **Operating Systems: Three Easy Pieces,**
By Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau
- Recommended textbook
 - **Operating Systems Principles & Practices**
By T. Anderson and M. Dahlin
- **Prerequisites are enforced!!**
 - CS 310 Data Structures
 - CS 367 Computer Systems & Programming
 - CS 465 Computer Systems Architecture
 - Be comfortable with C programming language
- Class web page
 - <https://tddg.github.io/cs571-spring21/>
 - Class materials will all be available on the class web page

Administrivia (cont.)

- Syllabus
 - https://cs.gmu.edu/media/syllabi/Spring2021/CS_571ChengY.html
- Grading
 - 50% projects
 - 10% homework
 - 20% midterm exam
 - 20% final exam
- Reminders
 - Honor code
 - Late policy: 15% deducted each day. No credit after 3 days

Course schedule

- Materials, assignments, due dates

CS 571: Operating Systems
George Mason University

Home

Course Information

Course Schedule

Project 0a (Linux utility)

Project 0b (Intro to Go)

Project 1 (Linux shell)

Project 2 (Caching policies)

Project 3 (Green threads)

Project 4 (MapReduce)

Project 5 (MDFS)

GitLab Setup

Announcements

≡

CS 571 Operating Systems (Spring 2021)

Course Schedule

The course schedule is tentative and subject to change*.

Week	Wednesday	Friday
Week 1	Jan 27 Lec 1: Introduction, process abstraction Proj 0a and Proj 0b out	Jan 29
Week 2	Feb 3 Lec 2: CPU scheduling I	Feb 5 Proj 0a due Proj 1 out
Week 3	Feb 10 Lec 3: CPU scheduling II	Feb 12
Week 4	Feb 17 Lec 4: Memory management I	Feb 19
Week 5	Feb 24 Lec 5: Memory management II	Feb 26 Proj 1 due Proj 2 out
Week 6	Mar 3 Lec 6: Thread Midterm review	Mar 5
Week 7	Mar 9 Midterm exam	Mar 12
Week 8	Mar 17	Mar 19

Course format

- (Review) + lecture + (*worksheets* and/or *demos*)
 - A short overview of the previous lecture to make sure the old content is not completely forgotten
 - Worksheet practices to make sure the lecture is well understood
 - Demos to help you gain a better understanding of the materials taught
 - e.g., OSTEP demos/simulators, tools
 - We will also cover a few seminal research papers on the way
 - ARC, MapReduce

Course projects

- Goals:
 1. To gain hands-on systems programming experience with C
 2. To gain experience building practical distributed systems using Go

Course projects

- Goals:
 1. To gain hands-on systems programming experience with C
 2. To gain experience building practical distributed systems using Go
- Five + one coding projects
 - Project 0a (C warm-up): Linux utilities
 - Project 0b: Intro to Go
 - Project 1: Implement a Linux shell
 - Project 2: Implement and analyze a suite of caching policies
 - Project 3: Implement a user-level green thread library
 - Project 4: Implement a MapReduce framework using Go
 - Project 5 (*extra credits*): Implement a Mason Distributed File System (MDFS) using Go

Course projects

- Goals:
 1. To gain hands-on systems programming experience
 2. To gain experience hacking a moderately sized system codebase (OS/161)
- Five + one coding projects (**50%+3%+7%**)
 - Project 0a (C warm-up): Linux utilities – 5%
 - Project 0b: Intro to Go – 5%
 - Project 1: Implement a Linux shell – 10%
 - Project 2: Implement and analyze a suite of caching policies – **10%+3%**
 - Project 3: Implement a user-level green thread library – 10%
 - Project 4: Implement a MapReduce framework using Go – 10%
 - Project 5 (*extra credits*): Implement a Mason Distributed File System (MDFS) using Go – 7%

Homework assignments

- Three written homework assignments
 - Assignment 0 (getting you prepared: 0%)
 - Assignment 1 before the midterm (5%)
 - Assignment 2 after the midterm (5%)

Getting help

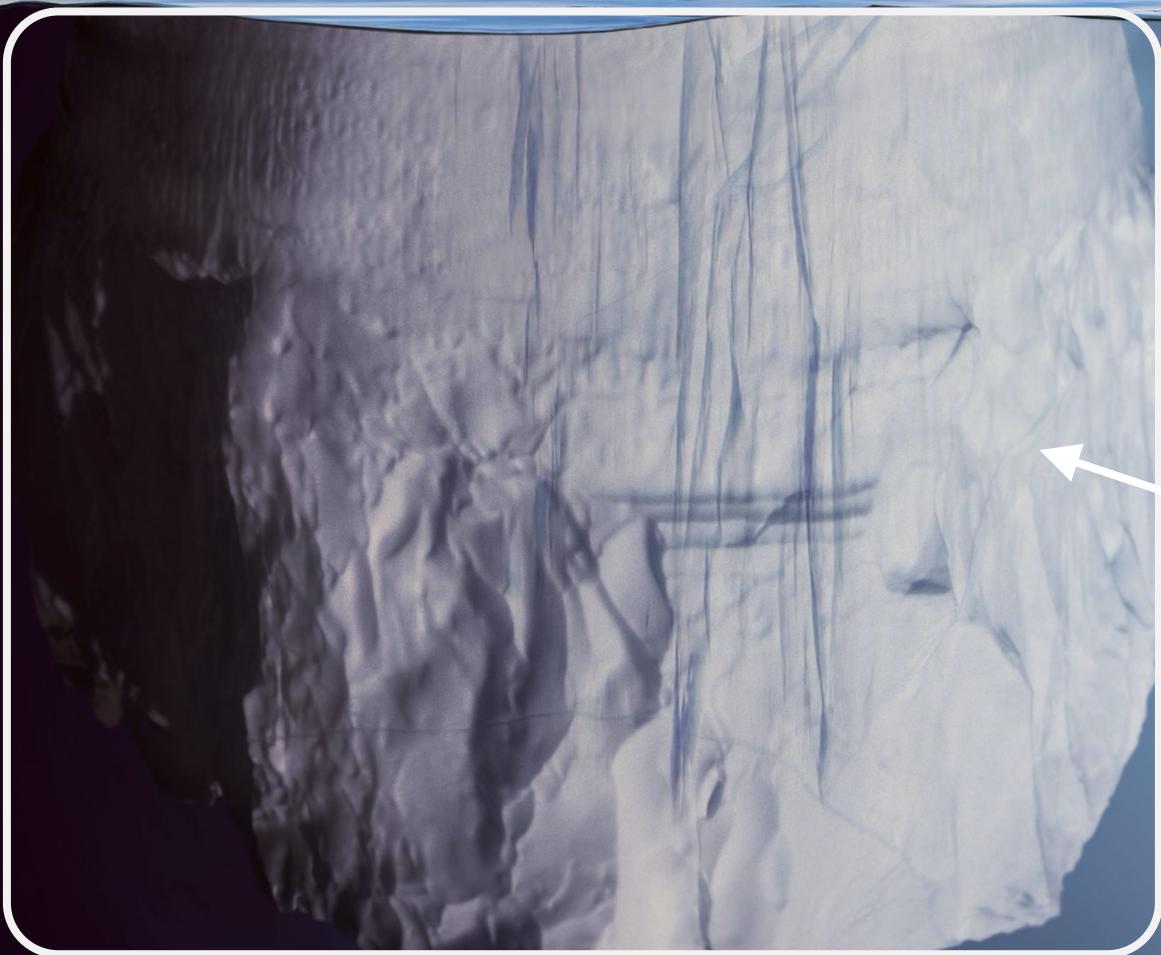
- My office hours
 - Wednesday 1:30 pm – 2:30 pm, on Zoom
- Michael's office hours
 - Monday 1:30 – 2:30 pm + Thursday 2:30 – 3:30 pm
- Piazza
 - Good place to ask and answer questions
 - About project and materials from lectures
 - No anonymous posts or questions
 - You are highly encouraged to answer questions posted by your classmates
 - **Setting expectation:** Michael and I will monitor/respond to Piazza 1-2 times per day in a burst of activity

What is an OS?

What is an OS?

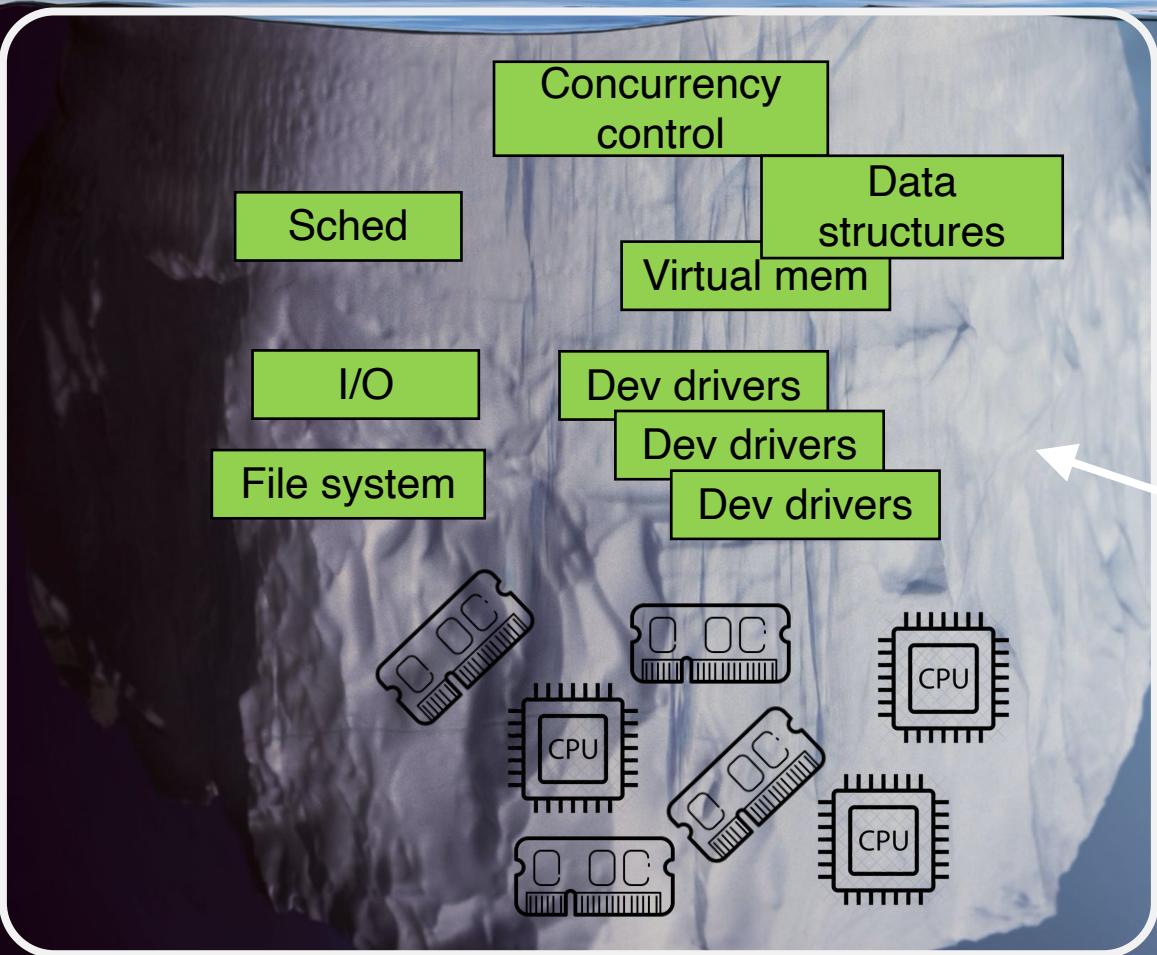
- OS manages resources
 - Memory, CPU, storage, network
 - Data (file systems, I/O)
- Provides low-level abstractions to applications
 - Files
 - Processes, threads
 - Virtual machines (VMs), containers
 - ...

OS abstracts away low-level details



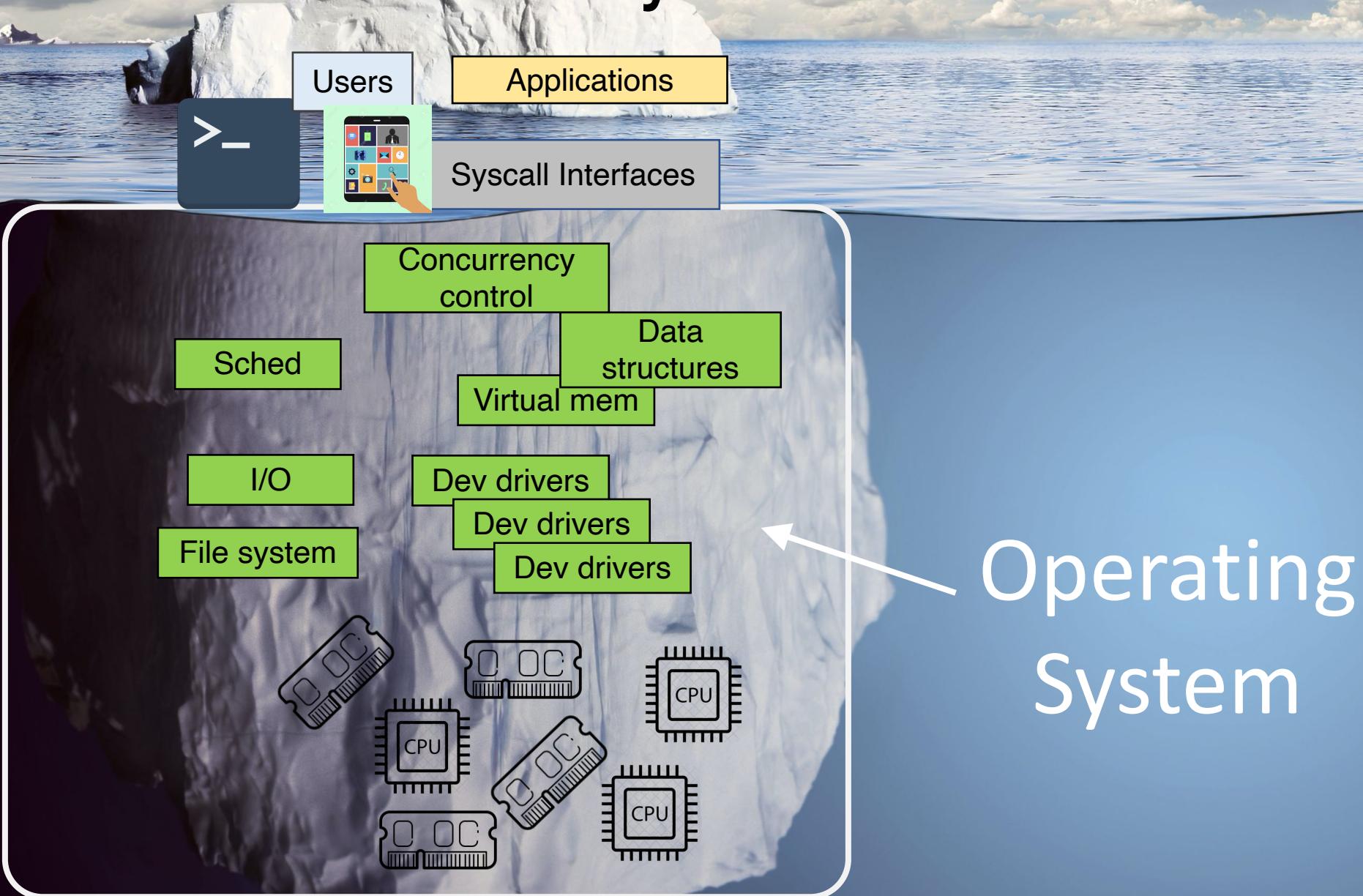
Operating
System

OS abstracts away low-level details



Operating
System

OS abstracts away low-level details



OS abstracts away low-level details

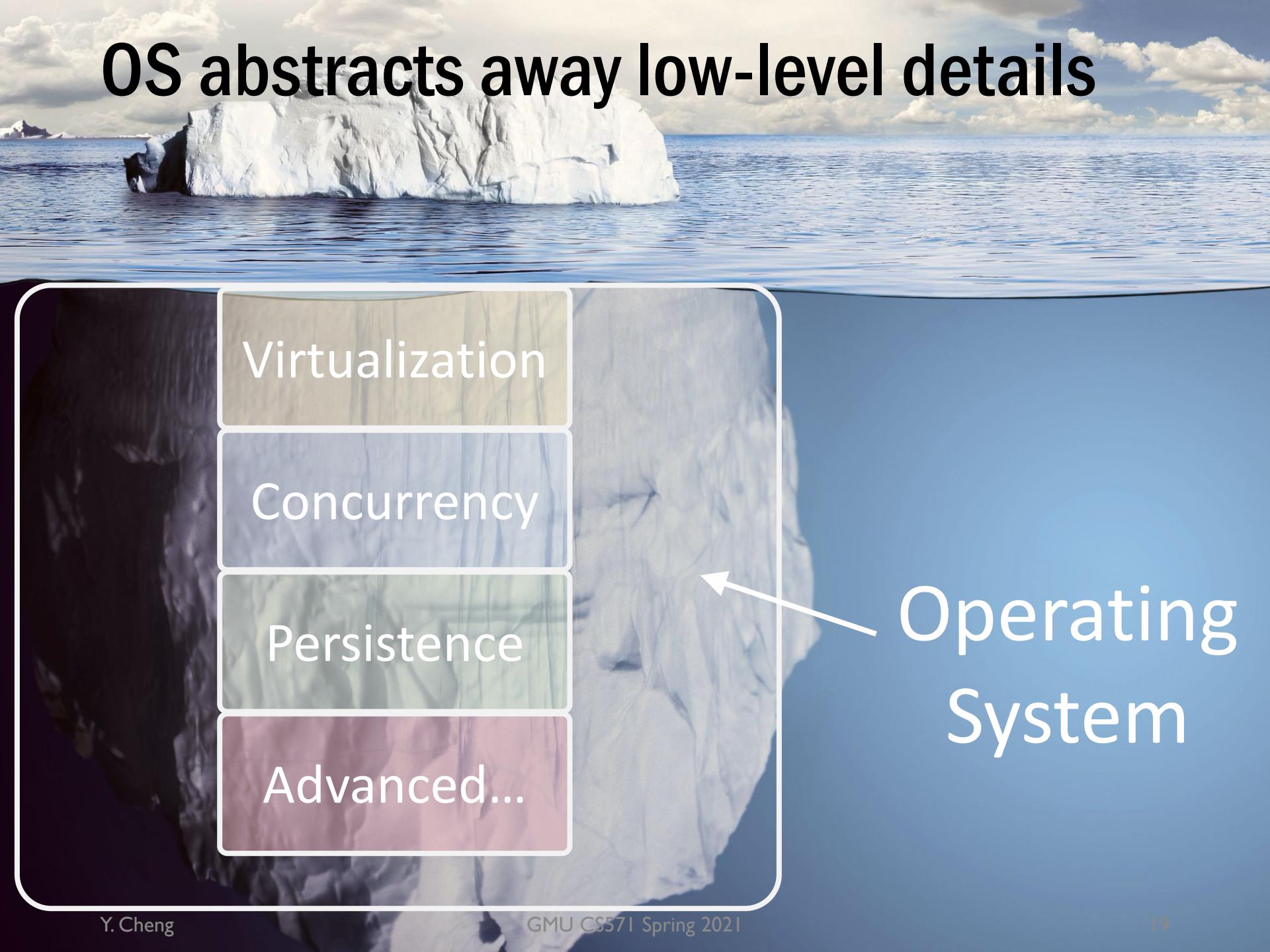
Virtualization

Concurrency

Persistence

Operating
System

OS abstracts away low-level details



A large iceberg is visible in the background, symbolizing the vast amount of low-level details that are abstracted away by the operating system. In the foreground, there is a graphic of four colored rectangles stacked vertically, each containing a system feature: Virtualization, Concurrency, Persistence, and Advanced... An arrow points from the text "Operating System" to the top of the iceberg, indicating that the OS handles these complex features.

- Virtualization
- Concurrency
- Persistence
- Advanced...

Operating
System

What happens when a program runs?

- A running program executes instructions
 1. The processor **fetches** an instruction from memory
 2. **Decode**: Understand which instruction it is
 3. **Execute**
 4. The processor moves on to **the next instruction** and so on

How does a running program interact with the OS?

- System calls allow a user application to tell the OS what to do
 - OS provides interfaces (APIs)
 - Hundreds of system calls (for Linux)
 - Run programs
 - Access memory
 - Access devices

Virtualization

- OS virtualizes physical resources
 - Gives illusion of private resources

Virtualizing the CPU

- OS creates and manages many virtual CPUs
 - Turning a single CPU into **seemingly infinite** number of CPUs
 - Allowing many programs to seemingly run **at once** (**concurrently**)

Demo

Virtualizing memory

- The physical memory is an array of bytes
- A program keeps (most of) its data in memory
 - Read memory (**load**): Access an address to fetch the data
 - Write memory (**store**): Store the data to a given address

Demo

Virtualizing memory (cont.)

- Each process access its own private **virtual address space**
 - OS maps **address space** onto the **physical memory**
 - A memory reference from a running program **does not affect** the address space of other processes
 - Physical memory is a **shared resource** managed by OS

Concurrency

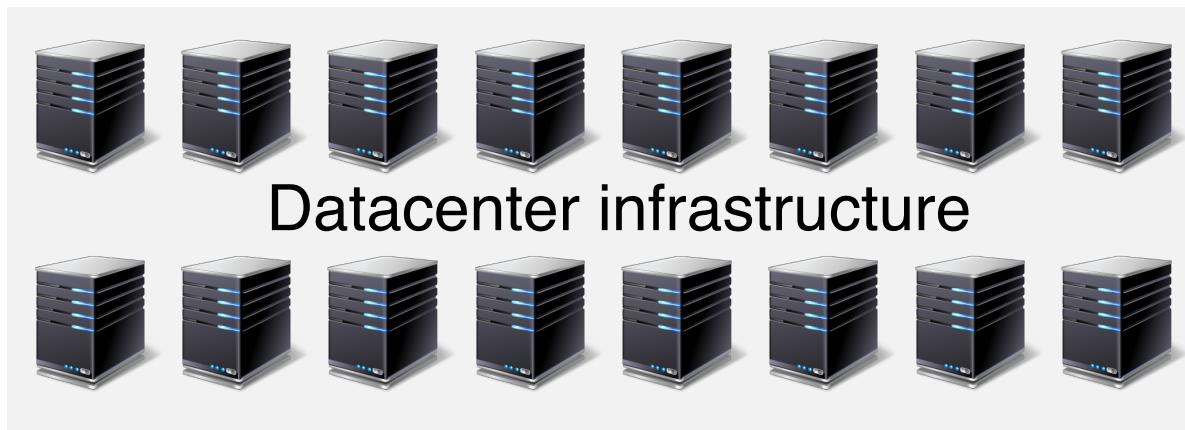
- OS is juggling many things at once
 - First running one process, then another, and so forth
- Multi-threaded programs also have concurrency problem

Demo

Persistence

- Main memory (DRAM) is **volatile**
- How to persist data?
 - **Hardware:** I/O devices such as hard disk drives (HDDs)
 - **Software:** File systems

Advanced topics – Distributed systems



Design goals

- Build up **abstraction**
 - Make the system easy to use
- Provide high **performance**
 - Minimize the overhead of OS
 - Virtualization w/o excessive overhead
- Protection between applications
 - **Isolation:** Bad behavior of one does not harm others and the OS itself

Why do you take this course?

General learning goals

1. Grasp **basic** knowledge about **Operating Systems** and **Computer Systems** software
2. Learn **important systems concepts** in general
 - Multi-processing/threading
 - Concurrency and synchronization
 - Scheduling
 - Caching, memory, storage
 - RPC, MapReduce
 - And more...
3. Gain **hands-on** experience in **writing/hacking/designing** moderately large systems software

Why do you take this course?

- The OS concepts are everywhere
 - Fundamental OS techniques broadly generalize to widely-used systems technique
 - Scheduling
 - Concurrency
 - Memory management
 - **Caching**
 - ...

What is a process?

What is a process?

- **Programs** are code (static entity)
- **Processes** are running programs
- Java analogy
 - class -> “program”
 - object -> “process”

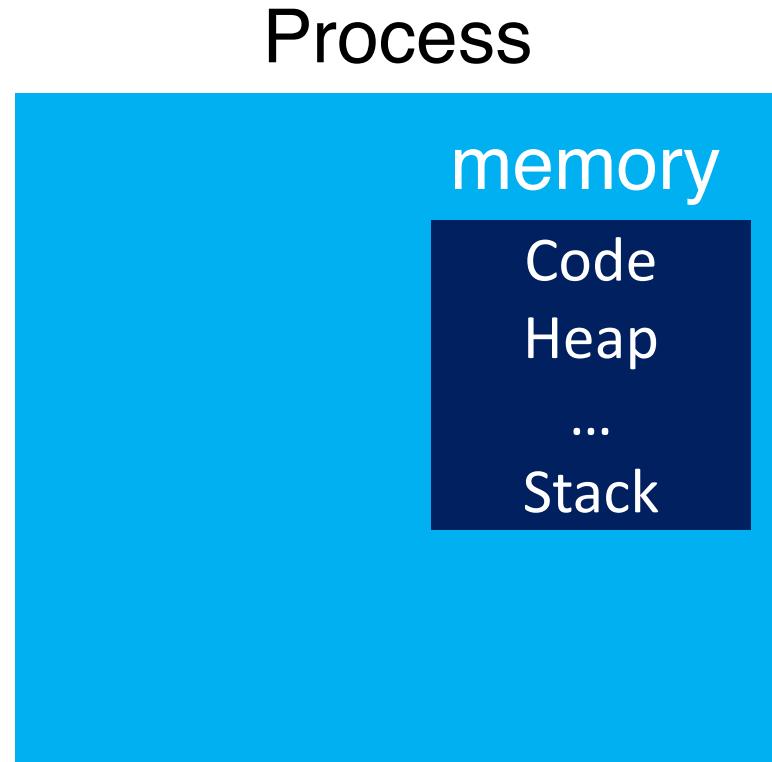
What is in a process?

Process



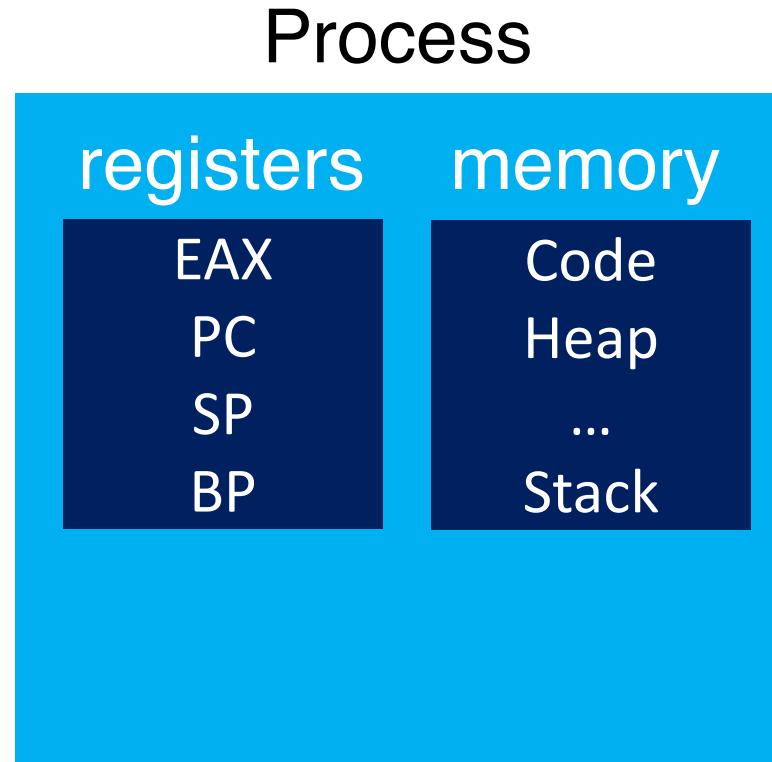
What things change as a program runs?

What is in a process?



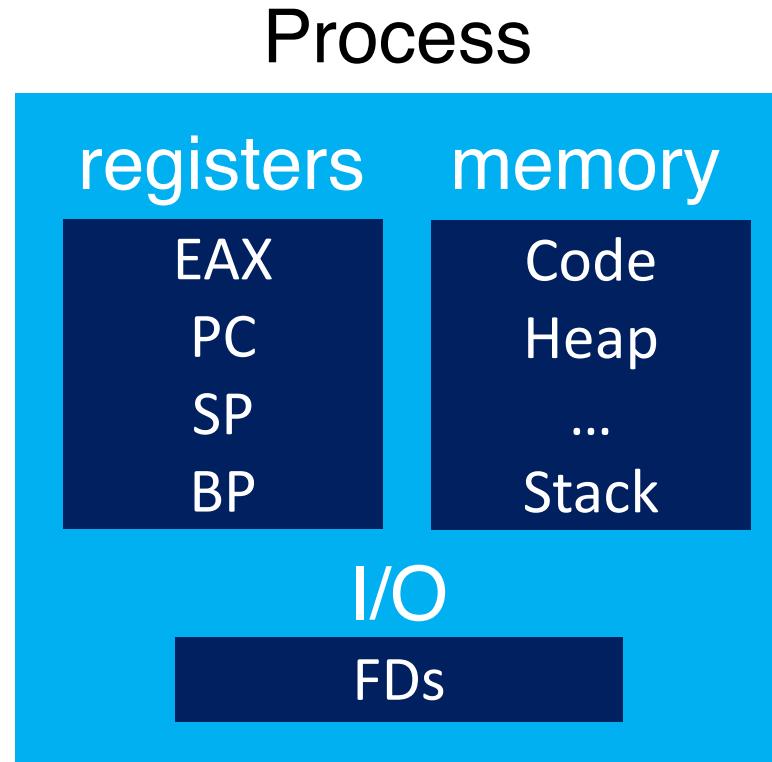
What things change as a program runs?

What is in a process?



What things change as a program runs?

What is in a process?



What things change as a program runs?

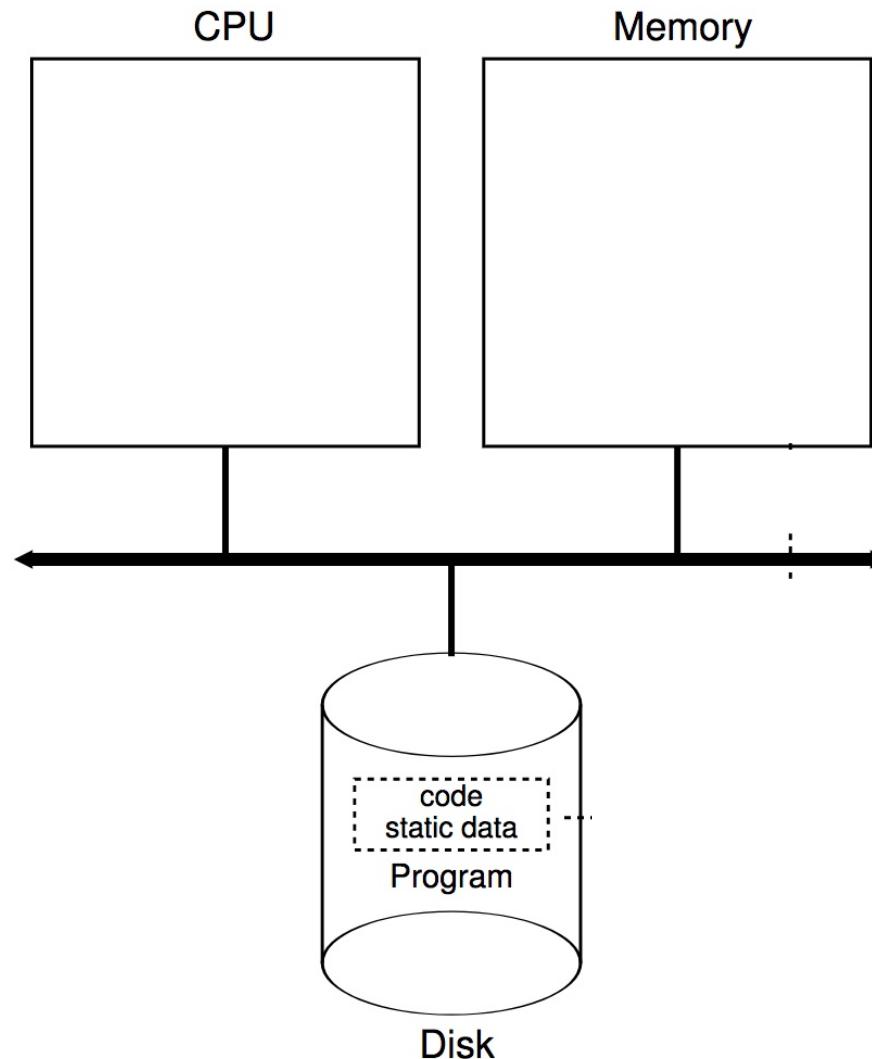
Peeking inside

- Processes share code, but each has its own “context”
- CPU
 - Instruction pointer (Program Counter)
 - Stack pointer
- Memory
 - Set of memory addresses (“address space”)
 - `cat /proc/<PID>/maps`
- Disk
 - Set of file descriptors
 - `cat /proc/<PID>/fdinfo/*`

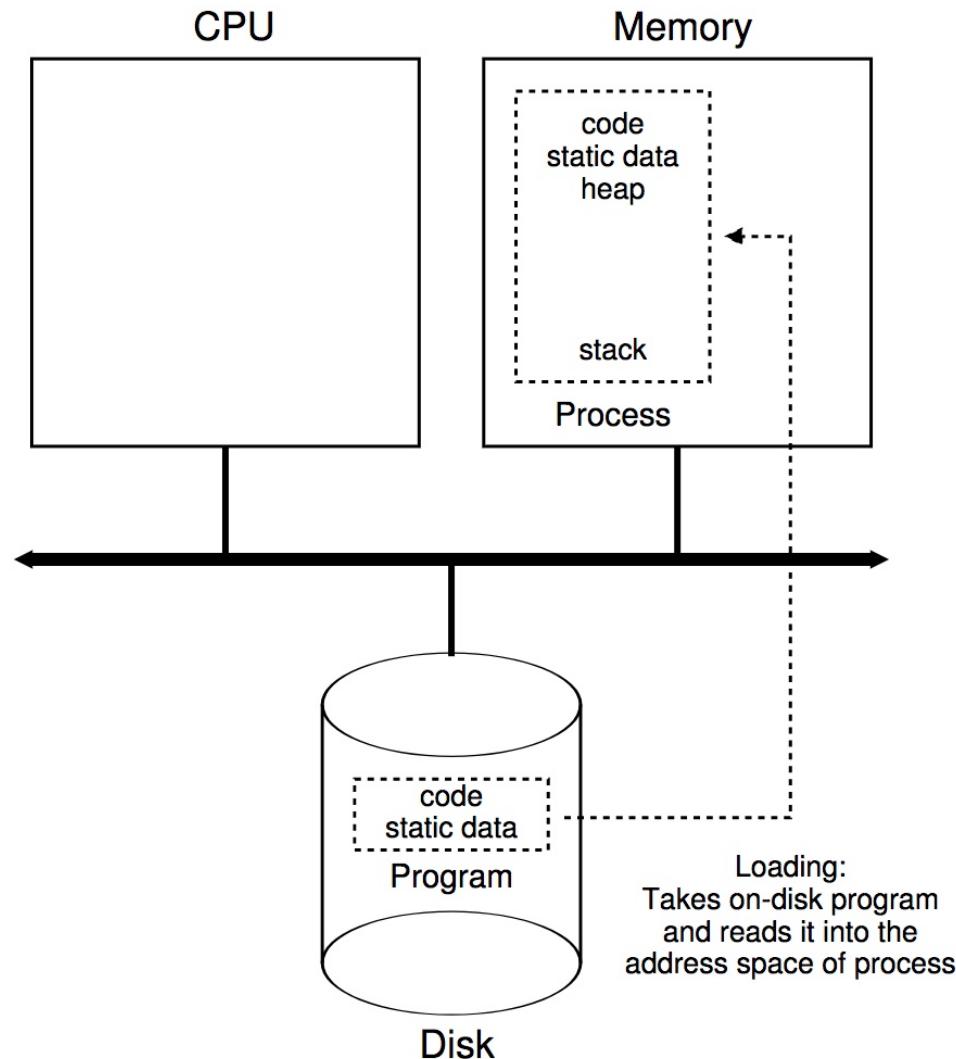
Process creation

- Principal events that cause process creation
 - System initialization
 - Execution of a process creation system call by a running process
 - User request to create a process

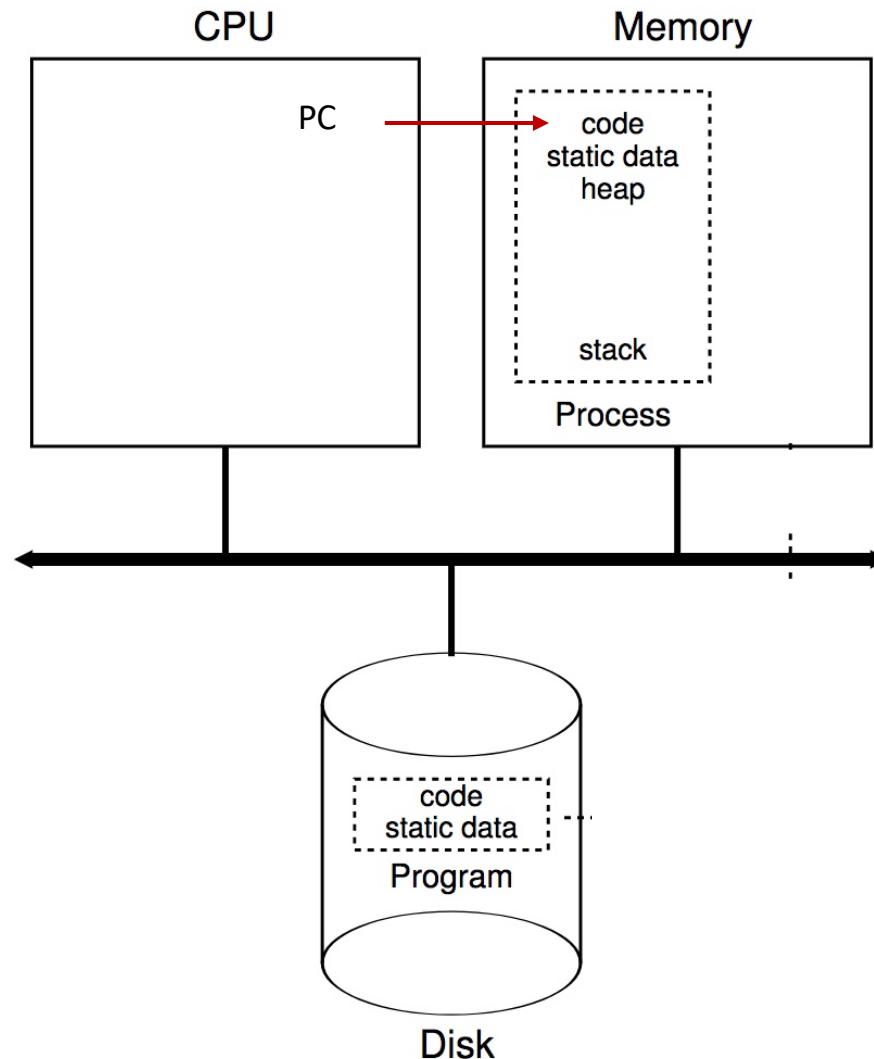
Process creation



Process creation



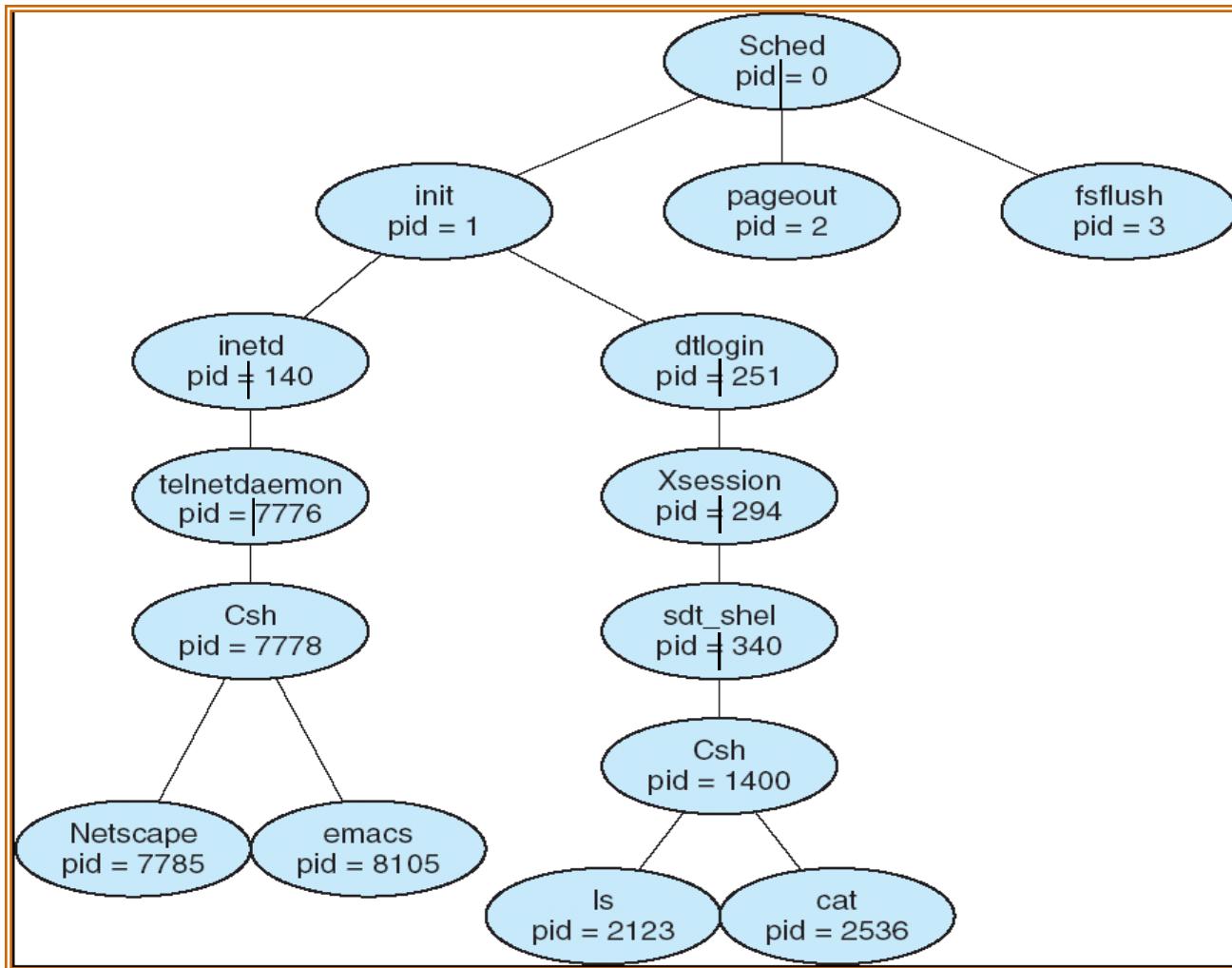
Process creation



Process creation (cont.)

- Parent process creates children processes, which, in turn create other processes, forming a tree (**hierarchy**) of processes

An example process tree



How to view process tree in Linux?

- % `ps auxf`
 - ‘f’ is the option to show the process tree
- % `pstree`

Process creation (cont.)

- Parent process creates children processes, which, in turn create other processes, forming a tree (**hierarchy**) of processes
- **Questions:**
 - Will the parent and child execute **concurrently**?
 - How will the **address space** of the child be related to that of the parent?
 - Will the parent and child **share some resources**?

Process creation in Linux

- Each process has a **process identifier (pid)**
- The parent executes **fork()** system call to spawn a child
- The child process has a **separate copy** of the parent's address space
 - Both the parent and the child continue execution at the instruction following the **fork()** system call
 - The return value for the **fork()** system call is
 - **zero** value for the new (**child**) process
 - **non-zero pid** for the **parent** process
 - Typically, a process can execute a system call like **execvp()** to load a binary file into memory

Process creation in Linux

- Each process has a **process identifier (pid)**
- The parent executes **fork()** system call to spawn a child
- The child process has a **separate copy** of the parent's address space
 - Both the parent and the child continue execution at the instruction following the **fork()** system call
 - The return value for the **fork()** system call is
 - **zero** value for the new (**child**) process
 - **non-zero pid** for the **parent** process
 - Typically, a process can execute a system call like **exec()** to load a binary file into memory

This is the pid of the child process

The man page of fork()

<http://man7.org/linux/man-pages/man2/fork.2.html>

RETURN VALUE

[top](#)

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and `errno` is set appropriately.

ERRORS

[top](#)

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

- * the `RLIMIT_NPROC` soft resource limit (set via `setrlimit(2)`), which limits the number of processes and threads for a real user ID, was reached;
- * the kernel's system-wide limit on the number of processes and threads, `/proc/sys/kernel/threads-max`, was reached (see `proc(5)`);
- * the maximum number of PIDs, `/proc/sys/kernel/pid_max`, was reached (see `proc(5)`); or
- * the PID limit (`pids.max`) imposed by the cgroup "process number" (PIDs) controller was reached.

A new system call: execvp()

- `execvp()` effectively **reboots** a process to run a different program from scratch
- `execvp()` has many variants (`execle`, `execlp`, and so forth. Type `man execvp` to see all of them)
- We generally use `execvp()` in this course

Example program with fork()

```
void main () {
    int pid;

    pid = fork();
    if (pid < 0) {/* error_msg */}
    else if (pid == 0) { /* child process */
        execl("/bin/ls", "ls", NULL); /* execute ls */
    } else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        exit(0);
    }
    return;
}
```

A Very simple shell using fork()

```
while (1) {  
    type_prompt();  
    read_command(cmd);  
    pid = fork();  
    if (pid < 0) {/* error_msg */}  
    else if (pid == 0) { /* child process */  
        execute_command(cmd);  
    } else { /* parent process */  
        wait(NULL);  
    }  
}
```

More example: fork 1

```
forkexample.c *  
1 #include <sys/types.h>  
2 #include <stdio.h>  
3 #include <stdlib.h>  
4 #include <unistd.h>  
5  
6 int number = 7;  
7  
8 int main(void) {  
9     pid_t pid;  
10    printf("\nRunning the fork example\n");  
11    printf("The initial value of number is %d\n", number);  
12  
13    pid = fork();  
14    printf("PID is %d\n", pid);  
15  
16    if (pid == 0) {  
17        number *= number;  
18        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);  
19        return 0;  
20    } else if (pid > 0) {  
21        wait(NULL);  
22        printf("In the parent, the number is %d\n", number);  
23    }  
24  
25    return 0;  
26 }  
27
```

Results

./forkexample1

Running the fork example

The initial value of number is 7

PID is 2137

PID is 0

In the child, the number is 49 -- PID is 0

In the parent, the number is 7

Further more example: fork 2

```
forkexample2.c *  
1 #include <sys/types.h>  
2 #include <stdio.h>  
3 #include <stdlib.h>  
4 #include <unistd.h>  
5  
6 int number = 7;  
7  
8 int main(void) {  
9     pid_t pid;  
10    printf("\nRunning the fork example\n");  
11    printf("The initial value of number is %d\n", number);  
12  
13    pid = fork();  
14    printf("PID is %d\n", pid);  
15  
16    if (pid == 0) {  
17        number *= number;  
18        fork();  
19        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);  
20        return 0;  
21    } else if (pid > 0) {  
22        wait(NULL);  
23        printf("In the parent, the number is %d\n", number);  
24    }  
25  
26    return 0;  
27 }  
28 }
```

Results

./forkexample2

Running the fork example

The initial value of number is 7

PID is 2164

PID is 0

In the child, the number is 49 -- PID is 0

In the child, the number is 49 -- PID is 0

In the parent, the number is 7

execl (or execvp) vs. fork

```
execlexample.c *  
1 #include <sys/types.h>  
2 #include <stdio.h>  
3 #include <stdlib.h>  
4 #include <unistd.h>  
5  
6 int number = 7;  
7  
8 int main(void) {  
9     pid_t pid;  
10    printf("\nRunning the execl example\n");  
11    pid = fork();  
12    printf("PID is %d\n", pid);  
13  
14    if (pid == 0) {  
15        printf("\tIn the execl child, PID is %d\n", pid);  
16        execl("./forkexample2", "forkexample2", NULL);  
17        return 0;  
18    } else if (pid > 0) {  
19        wait(NULL);  
20        printf("In the parent, done waiting\n");  
21    }  
22  
23    return 0;  
24 }
```

Results

./execlexample

Running the execl example

PID is 2179

PID is 0

In the execl child, PID is 0

Running the fork example

The initial value of number is 7

PID is 2180

PID is 0

In the child, the number is 49 -- PID is 0

In the child, the number is 49 -- PID is 0

In the parent, the number is 7

In the parent, done waiting

} forkexample2

Today's demo code

- You can fork it here: <https://github.com/remzi-arpacidusseau/ostep-code>
 - Three easy pieces: under `intro/`
 - Process-related: under `cpu-api/`

Assignment and project

- Assignment 0 (0%):
 - Please sign-up for  aws educate
 - Please sign-up for Piazza
 - Please finish the go programming exercise by Week 11
- Project 0 (10%)
 - Project 0a due next Friday, 02/05, end of day
 - Project 0b due (tentatively) on 04/09 – to familiarize yourself with Go