Evening

Introduction

CS 571: Operating Systems (Spring 2020)

Lecture 1

Yue Cheng

Some material taken/derived from:

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

Introduction

- Instructor
  - Dr. Yue Cheng (web: cs.gmu.edu/~yuecheng)
  - Email: yuecheng@gmu.edu
  - Office: 5324 Engineering
  - Office hours: M 1:30pm-2:30pm
  - Research interests: Distributed and storage systems, serverless and cloud computing, operating systems

HPC

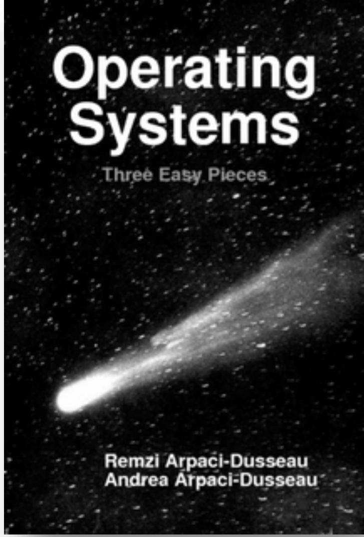
Introduction

- Instructor
  - Dr. Yue Cheng (web: cs.gmu.edu/~yuecheng)
  - Email: yuecheng@gmu.edu
  - Office: 5324 Engineering
  - Office hours: M 1:30pm-2:30pm
  - Research interests: Distributed and storage systems, serverless and cloud computing, operating systems

- Teaching assistant
  - Abhishek Roy
  - Email: aroy6@masonlive.gmu.edu
  - Office hours:
    - TBD

Administrivia

OSTEP



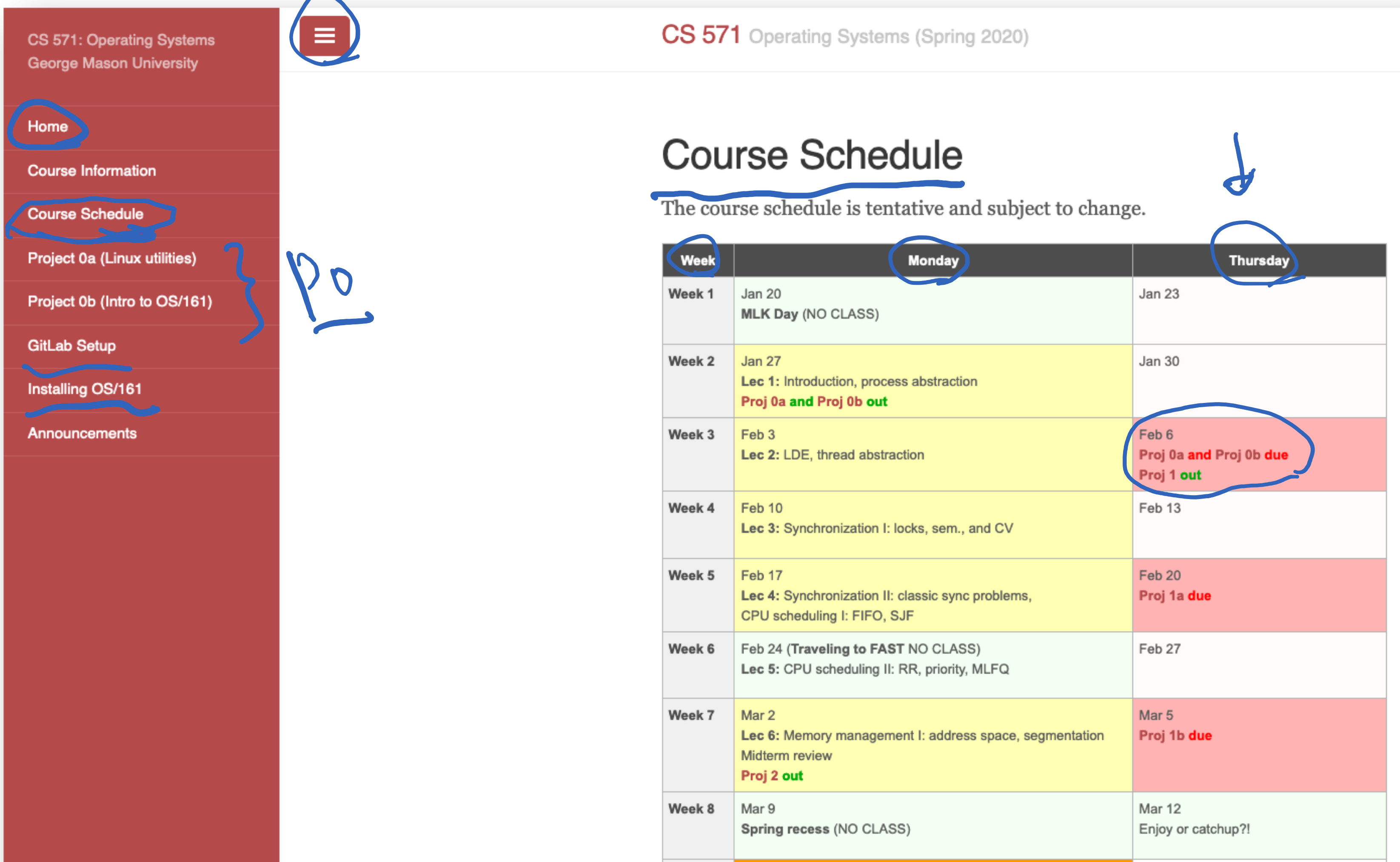
- Required textbook
  - Operating Systems: Three Easy Pieces,
    By Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau
- Recommended textbook
  - Operating Systems Principles & Practices
    By T. Anderson and M. Dahlin
- Prerequisites are enforced!!
  - CS 310 Data Structures
  - CS 367 Computer Systems & Programming
  - CS 465 Computer Systems Architecture
  - Be comfortable with C programming language
- Class web page
  - https://tddg.github.io/cs571-spring20/ ← URL
  - Class materials will all be available on the class web page

Administrivia (cont.)

- Syllabus
  - https://cs.gmu.edu/media/syllabi/Spring2020/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 utilities)
- Project 0b (Intro to OS/161)
- GitLab Setup
- Installing OS/161
- Announcements

CS 571 Operating Systems (Spring 2020)

Course Schedule

The course schedule is tentative and subject to change.

Week	Monday	Thursday
Week 1	Jan 20 **MLK Day** (NO CLASS)	Jan 23
Week 2	Jan 27 **Lec 1:** Introduction, process abstraction **Proj 0a and Proj 0b out**	Jan 30
Week 3	Feb 3 **Lec 2:** LDE, thread abstraction	Feb 6 **Proj 0a and Proj 0b due** **Proj 1 out**
Week 4	Feb 10 **Lec 3:** Synchronization I: locks, sem., and CV	Feb 13
Week 5	Feb 17 **Lec 4:** Synchronization II: classic sync problems, CPU scheduling I: FIFO, SJF	Feb 20 **Proj 1a due**
Week 6	Feb 24 (**Traveling to FAST** NO CLASS) **Lec 5:** CPU scheduling II: RR, priority, MLFQ	Feb 27
Week 7	Mar 2 **Lec 6:** Memory management I: address space, segmentation Midterm review **Proj 2 out**	Mar 5 **Proj 1b due**
Week 8	Mar 9 **Spring recess** (NO CLASS)	Mar 12 Enjoy or catchup?!

Course format

2hr 10-15min break Py Sim

- (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 deeper understanding of the materials taught
        - OSTEP simulators, measurements

Course projects

pthread fork

- Goal:
  1. To gain hands-on systems programming experience
  2. To gain experience hacking a moderately sized system codebase (OS/161)

~20k LoC C

Course projects

- Goal:
  1. To gain hands-on systems programming experience
  2. To gain experience hacking a moderately sized system codebase (OS/161)

- Four coding projects
  - Project 0a (Warm-up): Linux utilities
  - Project 0b: Intro to OS/161
  - Project 1a: Implement a Linux shell
  - Project 1b: OS/161 synchronization
  - Project 2a: OS/161 system calls
  - Project 2b: OS/161 CPU scheduling
  - Project 3: Implement a MapReduce app w/ C

Course projects

- Goal:
  1. To gain hands-on systems programming experience
  2. To gain experience hacking a moderately sized system codebase (OS/161)

- Four coding projects (50%)
  - Project 0a (Warm-up): Linux utilities – **5%**
  - Project 0b: Intro to OS/161 – **5%**
  - Project 1a: Implement a Linux shell – **7%**
  - Project 1b: OS/161 synchronization – **8%**
  - Project 2a: OS/161 system calls – **10%**
  - Project 2b: OS/161 CPU scheduling – **5%**
  - Project 3: Implement a MapReduce app w/ C – **10%**

Homework assignments

- Two written homework assignments
  - One before the midterm 5%
  - One after the midterm 5%

} 10%

Getting help

- Office hours
  - Monday 1:30 pm – 2:30 pm, Engineering 5324

- Piazza
  - Good place to ask and answer questions
    - About project
    - About material from lecture
  - No anonymous posts or questions

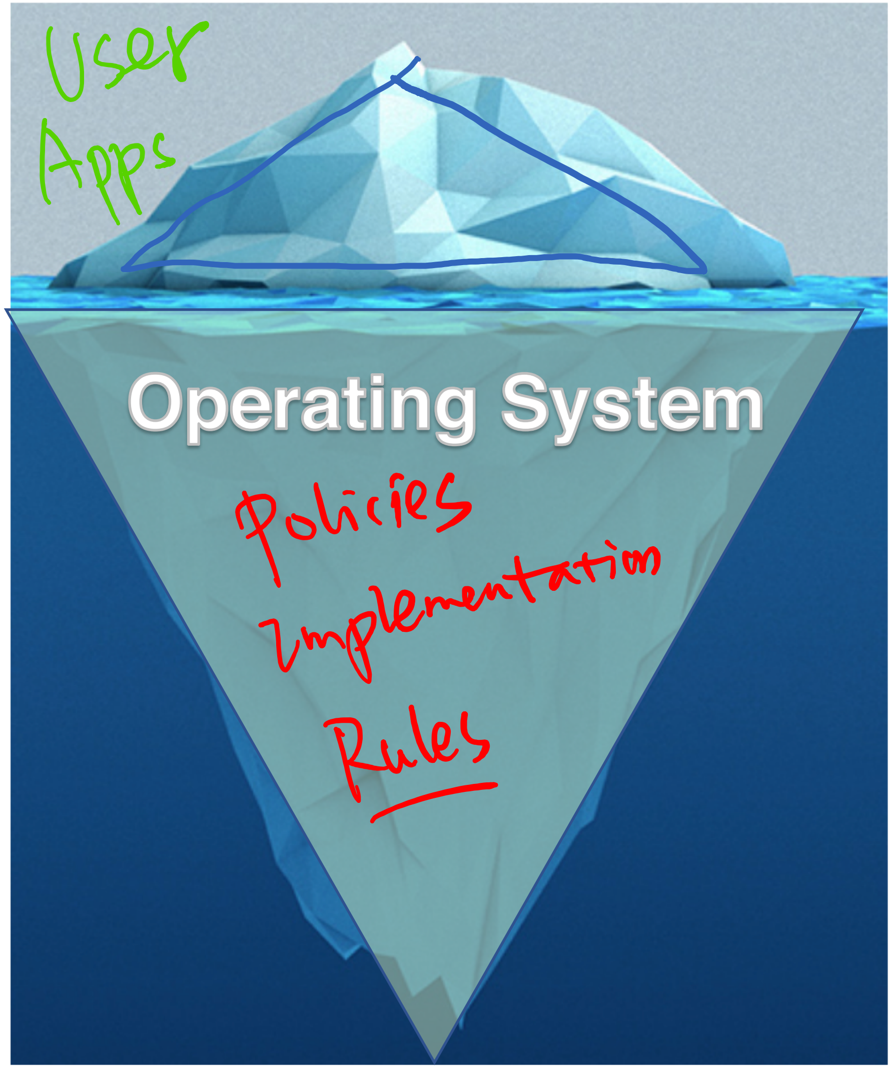
What is an OS?

What is an OS?

- OS manages resources H/W
  - Memory, CPU, storage, network
  - Data (file systems, I/O)

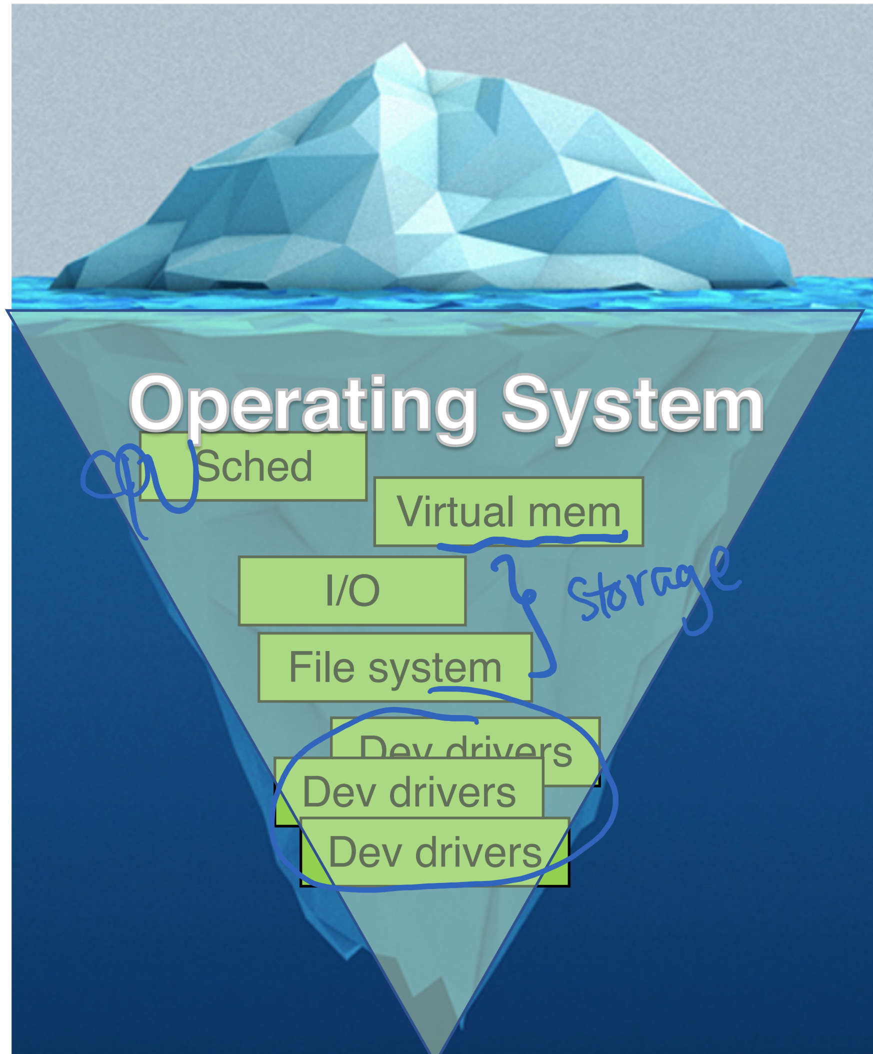
- Provides low-level abstractions to applications
  - Files
  - Processes, threads
  - Virtual machines (VMs), containers
  - …

OS abstracts away low-level details



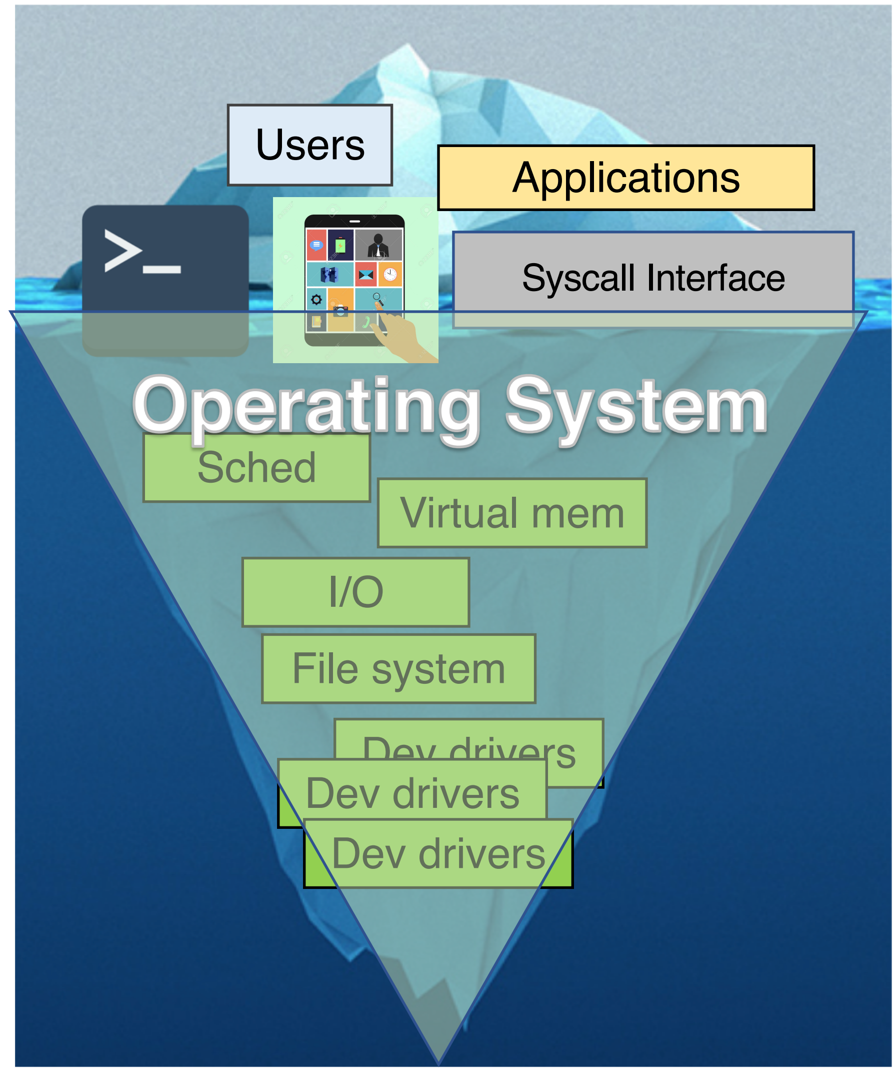
OS abstracts away low-level details

- Under the surface
  - Complex and dirty implementations of abstractions and a lot more…



OS abstracts away low-level details

- User's perspective
  - User interface:
    - Terminal, GUI
  - Application interface:
    - System calls APIs
- Under the surface
  - Complex and dirty implementations of abstractions and a lot more…



The goals of 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
  - …

- Goals
  - Resource efficiency (resource virtualization)
  - Ease-of-use (interfaces)
  - Reliability (user-kernel space separation)

std C

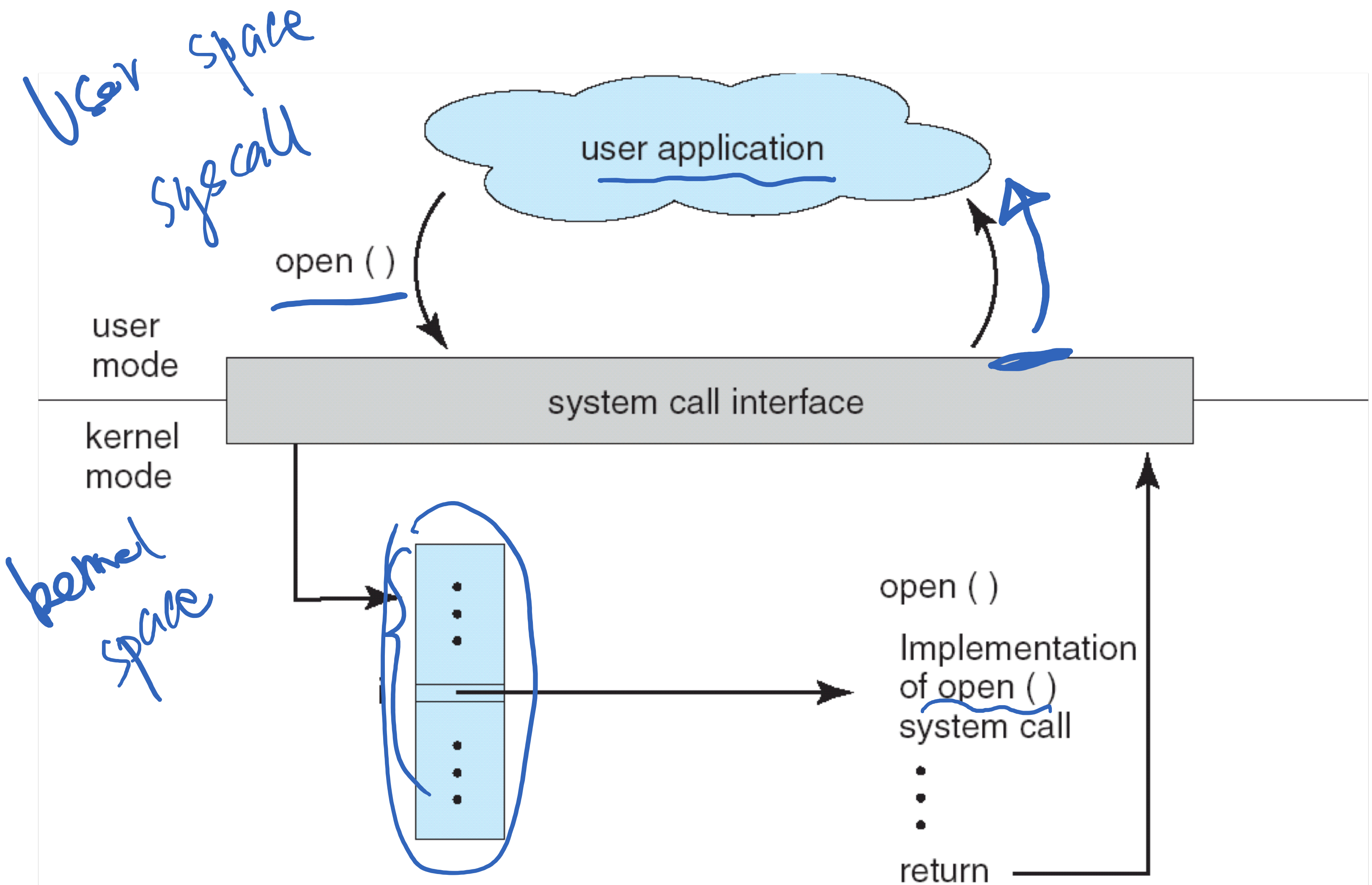
malloc

syscalls

System Calls

- System calls provide the interface between a running program and the operating system
  - Generally available in routines written in C and C++
  - Certain low-level tasks may have to be written using assembly language
- Typically, application programmers design programs using an application programming interface (API)
- The runtime support system (runtime libraries) provides a system-call interface, that intercepts function calls in the API and invokes the necessary system call within the operating system
- Major differences in how they are implemented (e.g., Windows vs. Unix)

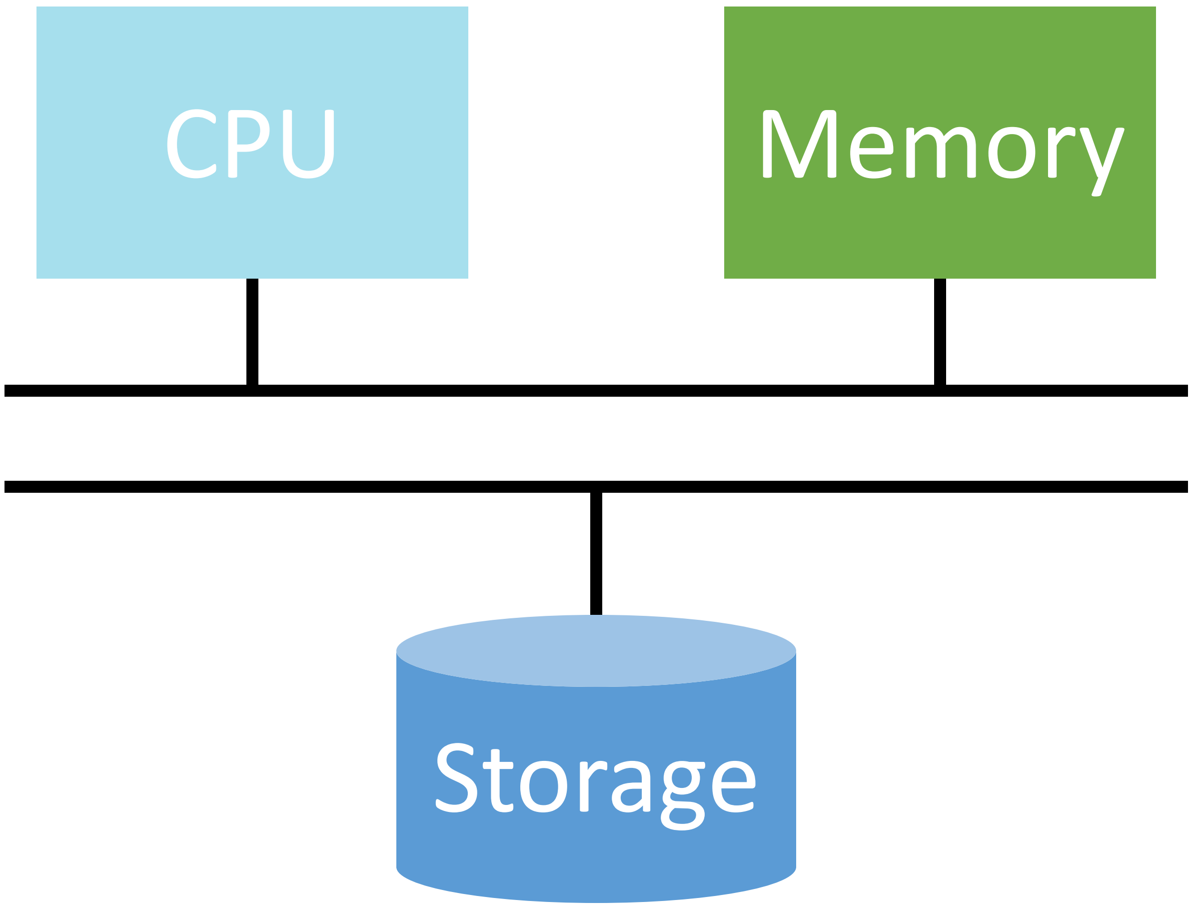
Example System Call Processing



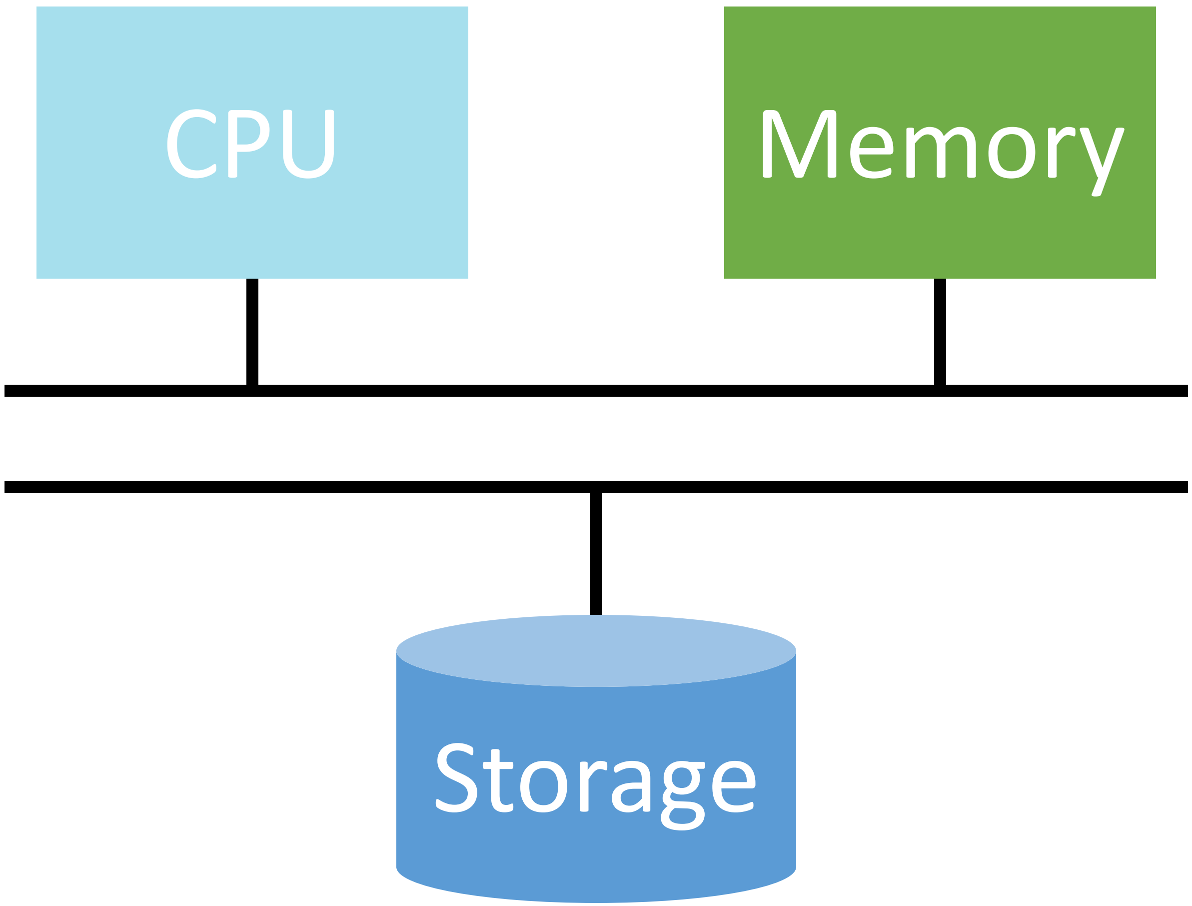
Major System Calls in Linux: File Management

- `fd = open(file, how, …)`
  - Open a file for reading, writing, or both
- `s = close(file)`
  - Close an open file
- `n = read(fd, buf, nbytes)`
  - Read data from a file into a buffer
- `n = write(fd, buf, nbytes)`
  - Write data from a buffer into a file
- `pos = lseek(fd, offset, whence)`
  - Move the file pointer
- `s = stat(name, &buf)`
  - Get a file's status info

3 Major Topics

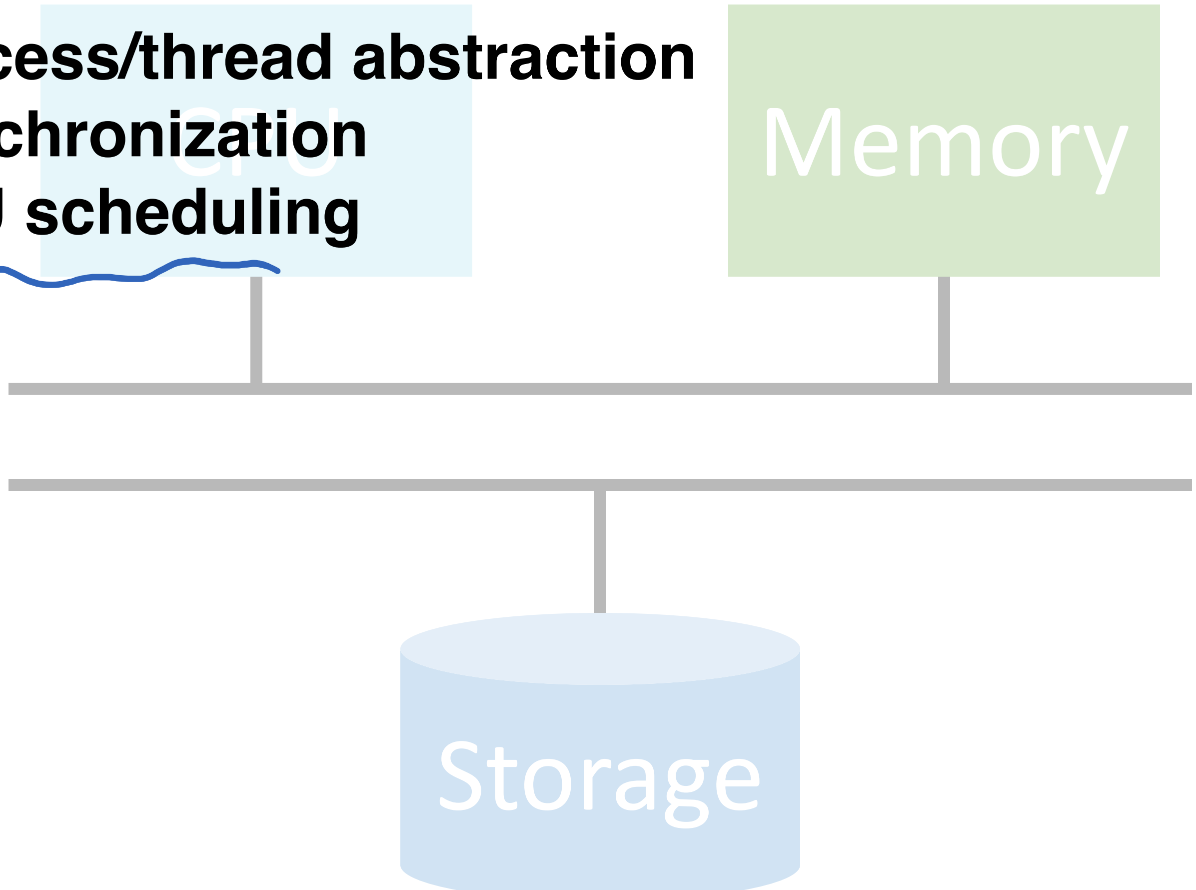


OS Provides Virtualization on Hardware



Topic 1: Concurrency, Synchronization, and CPU Scheduling

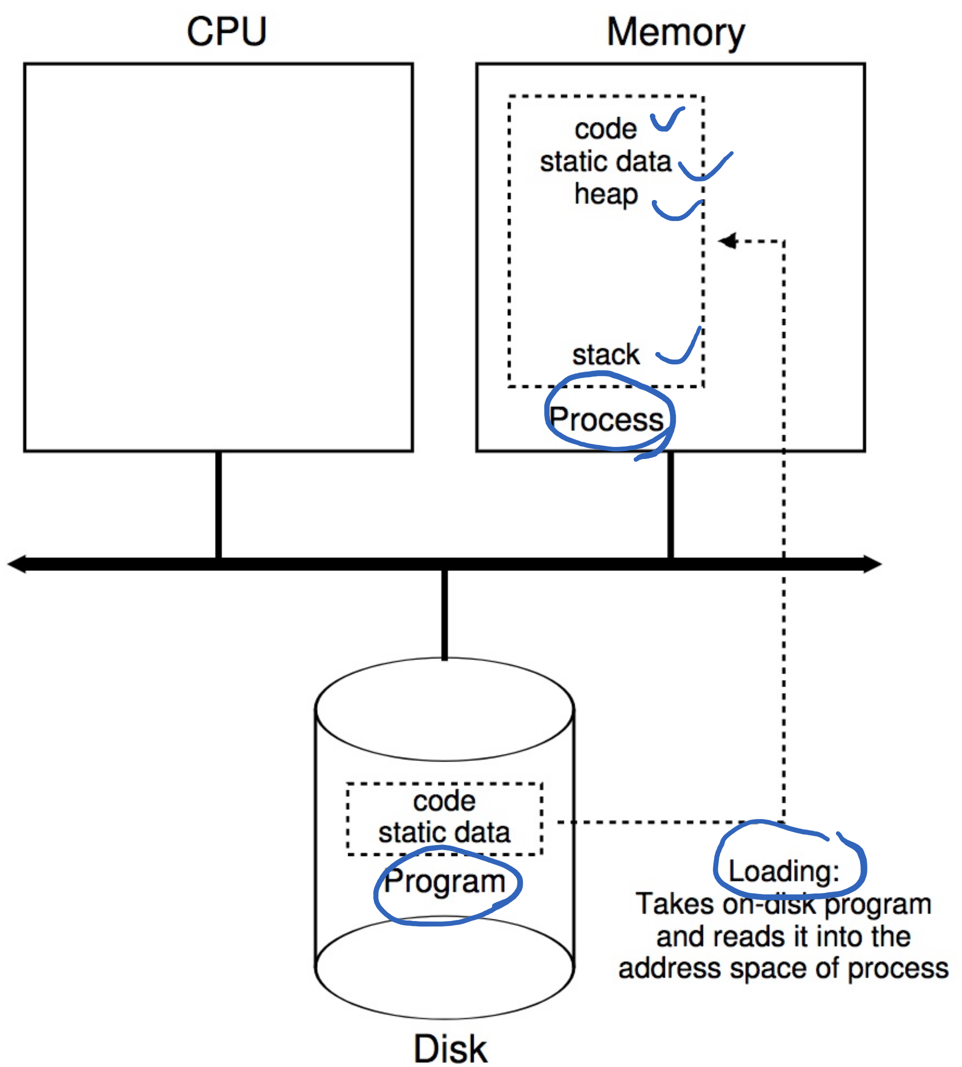
- **Process/thread abstraction**
- **Synchronization**
- **CPU scheduling**



Process Abstraction

- A process is a program in execution
  - It is a unit of work within the system. A program is a **passive entity**, a process is an **active entity**.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one program counter specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- A software system may have many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Loading from Program to Process



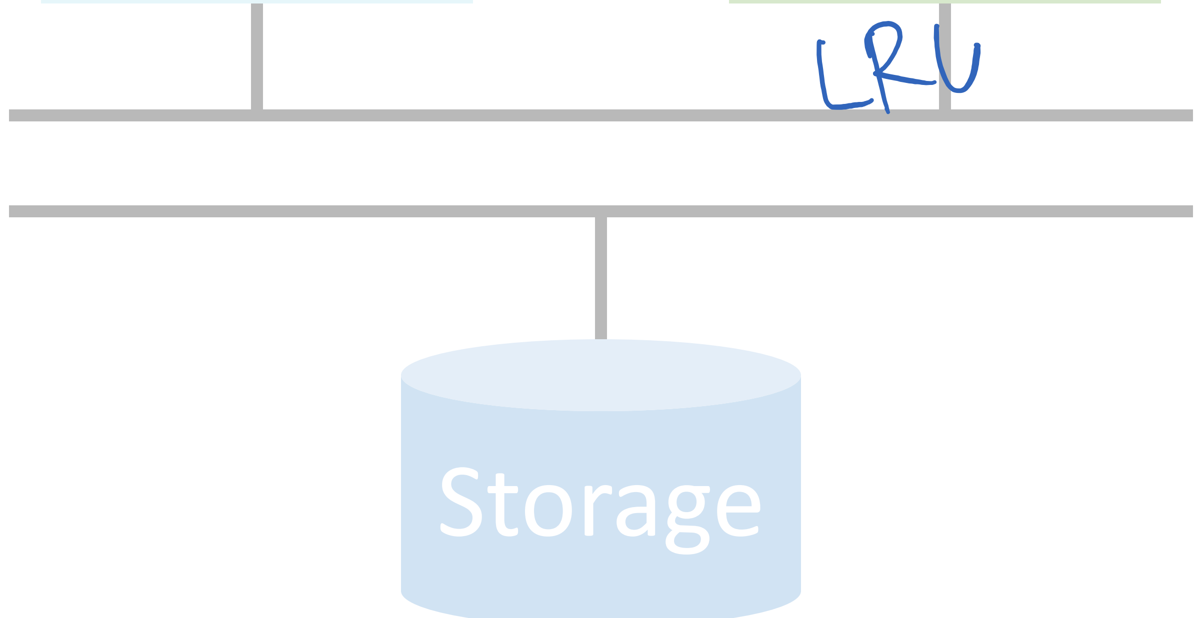
Topic 2: Memory Management and Virtual Memory

time-shared

space-shared

- Process/thread abstraction
- Synchronization
- CPU scheduling

- Memory management
- Virtual memory



Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed
- **Virtual memory** management is an essential part of most operating systems

PT

Topic 3: Storage, I/O, and Filesystems

- Process/thread abstraction
- Synchronization
- CPU scheduling

- Memory management
- Virtual memory

- Hard disk drives *HDDs*
- RAID
- Flash SSDs
- File and I/O systems

NFS

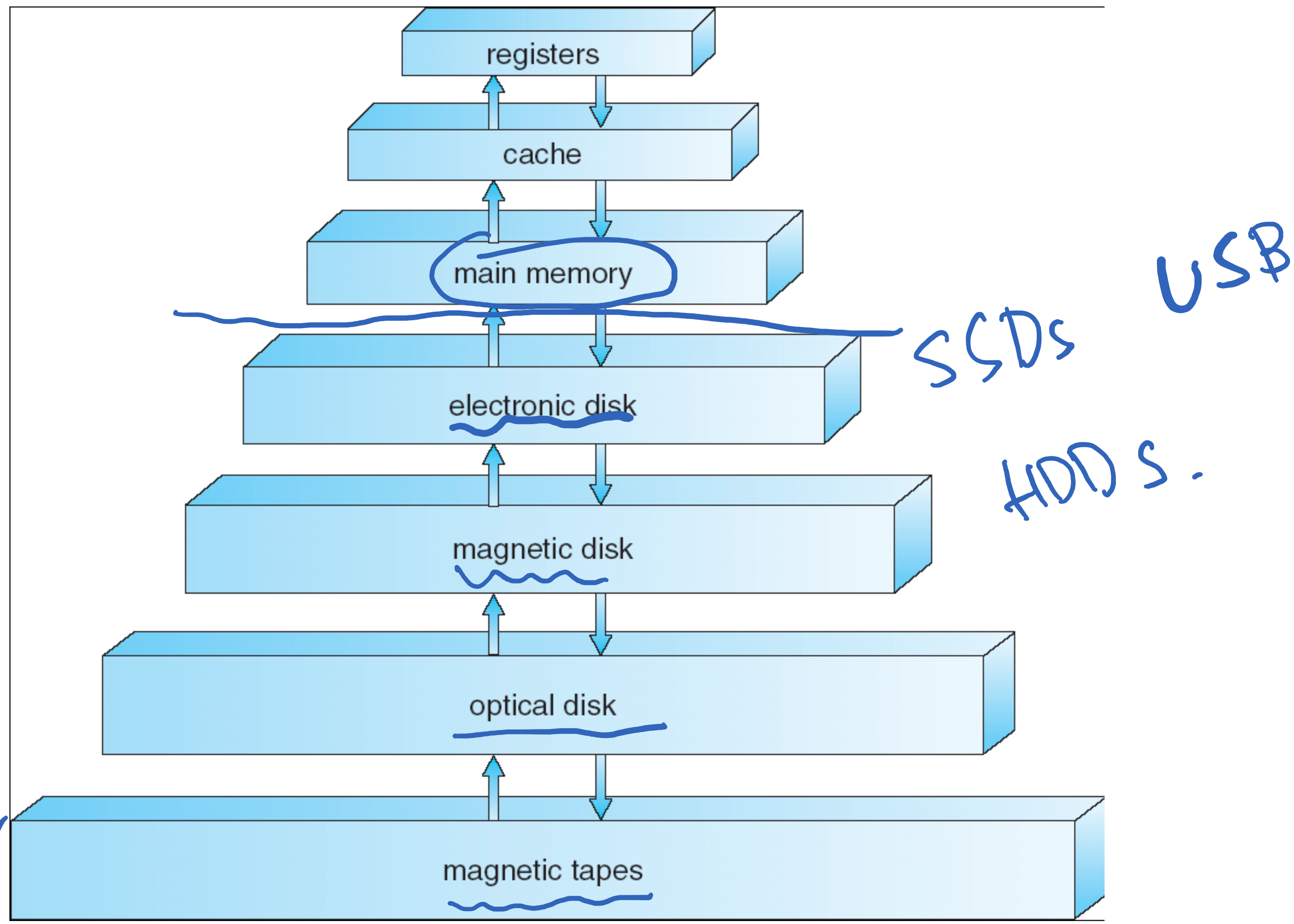
Storage Management

- OS provides a uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - file
  - Each medium is controlled by device type (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- Filesystem management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and dirs
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

Storage hierarchy

tradeoff

Fastest
$$$$



Slowest
$

Storage Structure

- Main memory – relatively large storage media that the CPU can access directly
  - Small CPU cache memories are used to speed up average access time to the main memory at run-time
  - Volatile (data loss at power-off)
  - Byte-addressable

- Secondary storage – extension of main memory that provides large nonvolatile storage capacity.
  - Magnetic disks
  - Electronic disks -- Solid state disks (SSDs)
  - Non-volatile (i.e., persistent)
  - Non byte-addressable

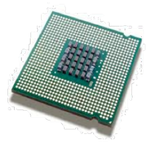
Storage Systems Tradeoffs

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
  - Density

- Faster access time, greater cost per bit
- Greater capacity (density), lower cost per bit
- Greater capacity (density), slower access speed

Increased complexity – Memory

2015



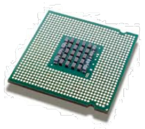
L1/L2 cache	~1 ns
L3 cache	~10 ns
Main memory	~100 ns / ~80 GB/s / ~100GB
NAND SSD	~100 usec / ~10 GB/s / ~1 TB
Fast HDD	~10 msec / ~100 MB/s / ~10 TB

Increased complexity – Memory

2015

L1/L2 cache	~1 ns
L3 cache	~10 ns
Main memory	~100 ns / ~80 GB/s / ~100GB
NAND SSD	~100 usec / ~10 GB/s / ~1 TB
Fast HDD	~10 msec / ~100 MB/s / ~10 TB

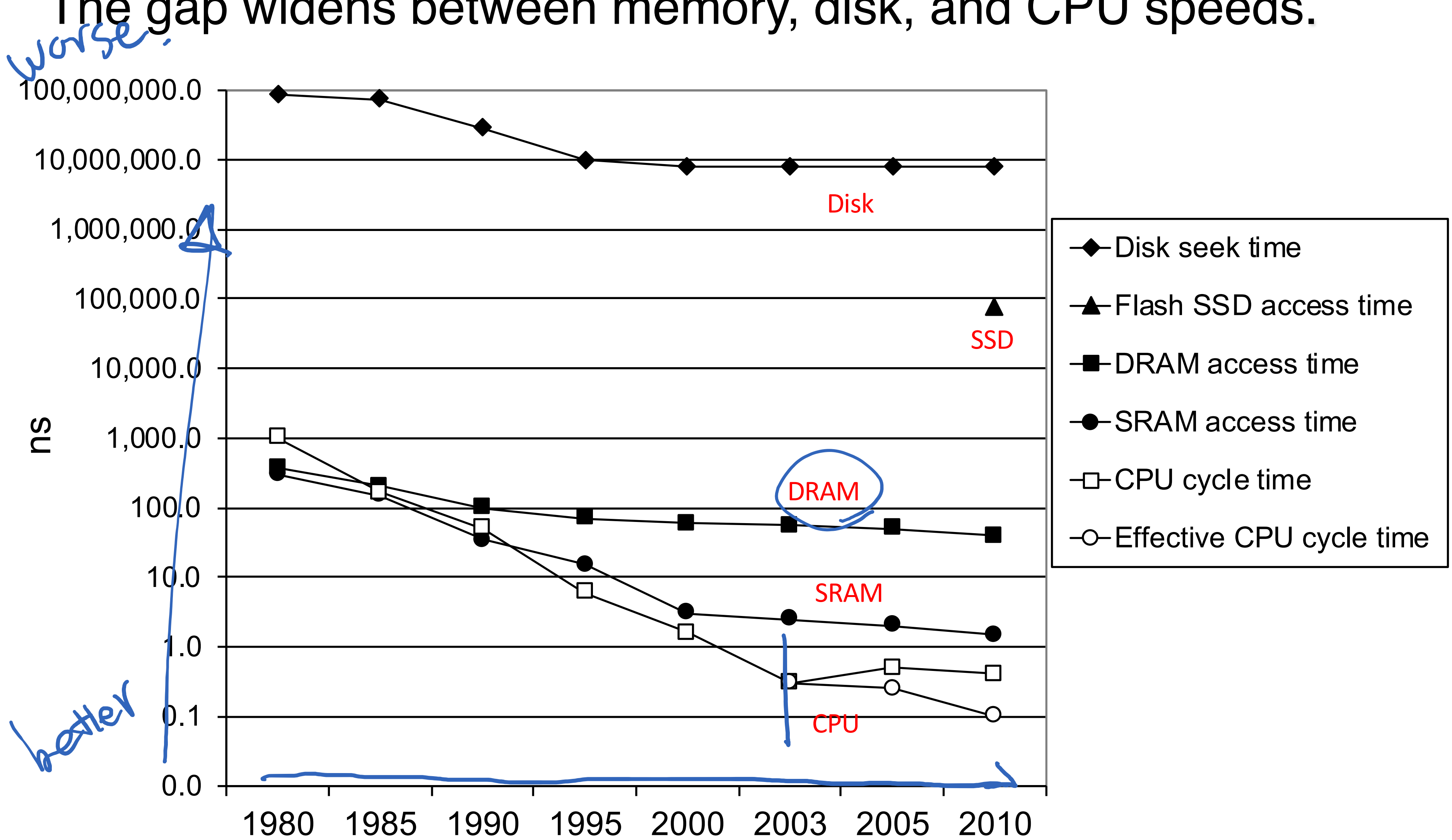
2020



L1/L2 cache	~1 ns
L3 cache	~10 ns BW
HBM	~10 ns / ~1TB/s / ~10GB
Main memory	~100 ns / ~80 GB/s / ~100GB
NVM (Intel Optane DC)	~1 usec / ~10GB/s / ~1TB
NAND SSD	~100 usec / ~10 GB/s / ~10 T
Fast HDD	~10 msec / ~100 MB/s / ~100

The CPU-Memory Gap

The gap widens between memory, disk, and CPU speeds.



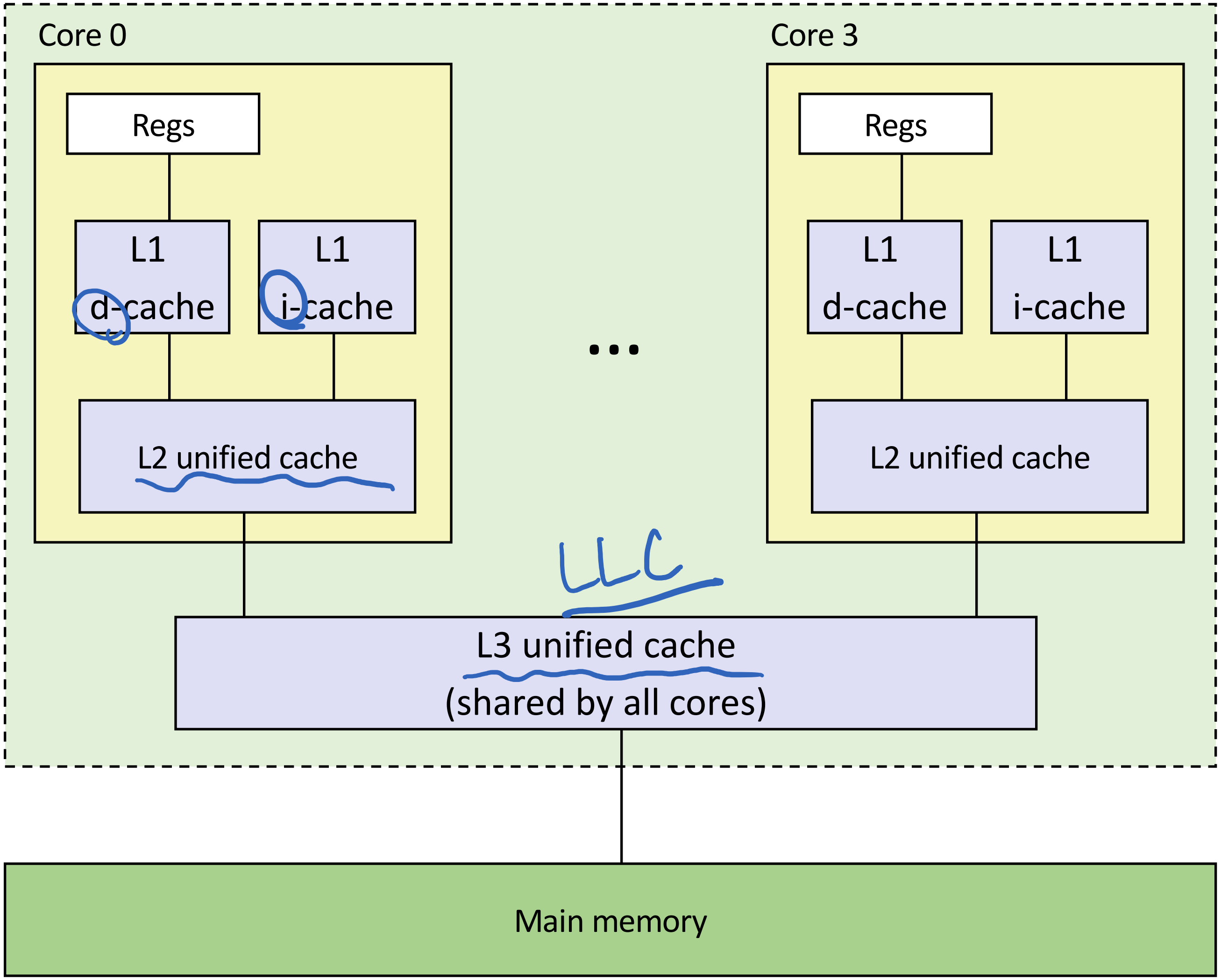
Data decades ago, but trends are the same

Caching

80-20

- Skew rule: 80% requests hit on 20% hottest data
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:

32 KB, 8-way,

Access: 4 cycles

L2 unified cache:

256 KB, 8-way,

Access: 11 cycles

L3 unified cache:

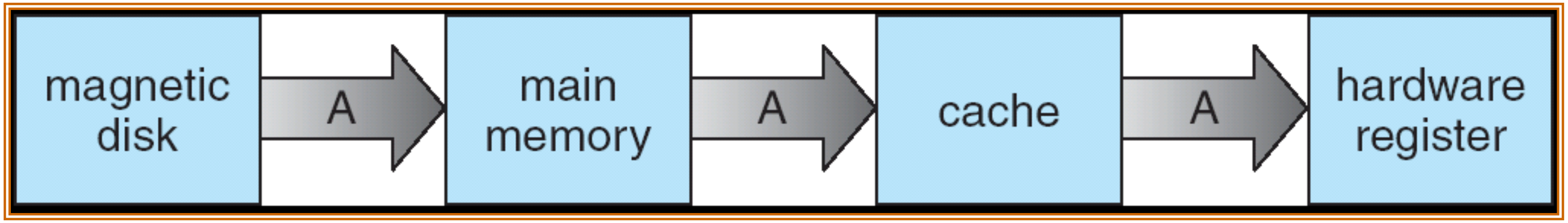
8 MB, 16-way,

Access: 30-40 cycles

Block size: 64 bytes

Migration of Integer A from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a piece of data can exist

Why do you take this course?

General Learning Goals

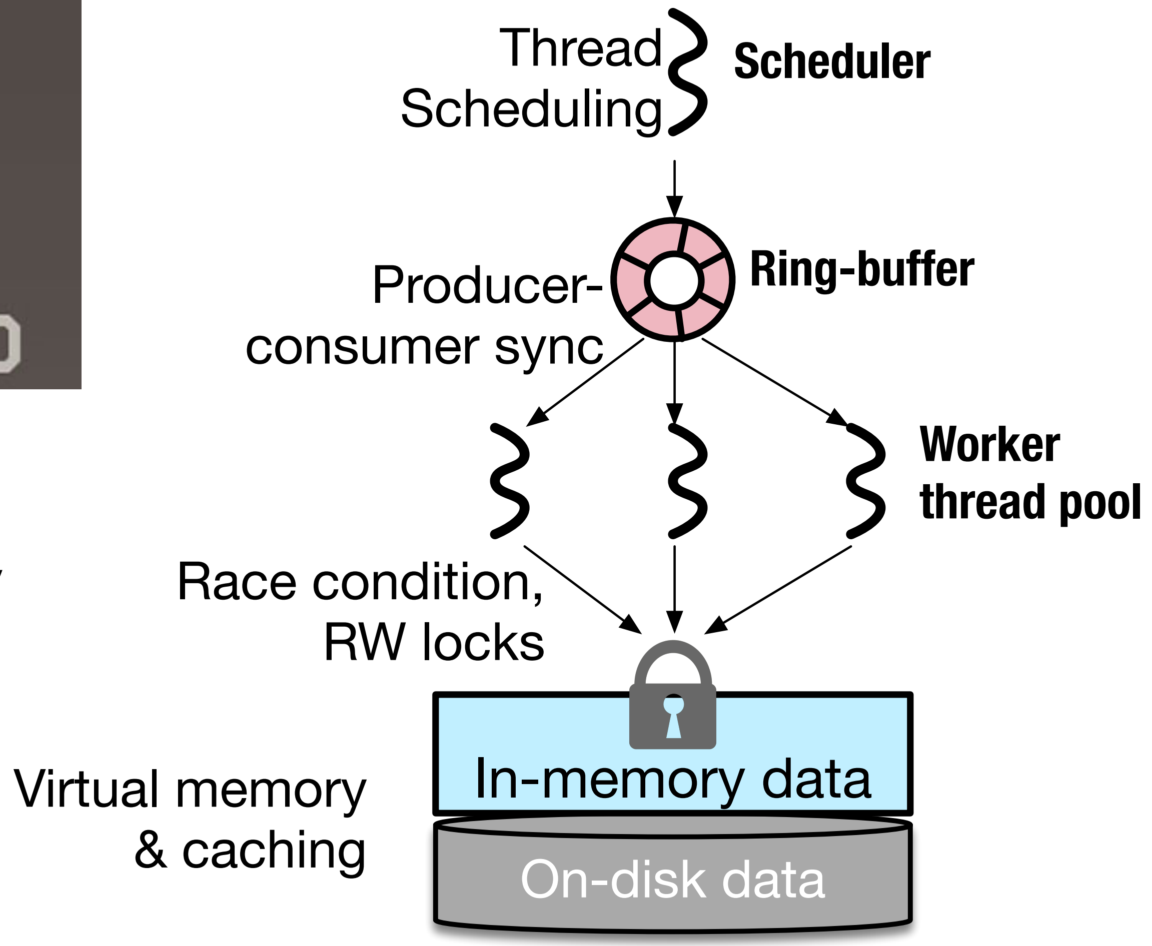
1. Grasp basic knowledge about Operating Systems and Computer Systems software
2. Learn important systems concepts in general
   - Multi-processing/threading, synchronization
   - Scheduling
   - Caching, memory, storage
   - And more…
3. Gain hands-on experience in writing/hacking/designing moderately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

One example: Memcached

- Memcached is a distributed in-memory object cache system
  - Written in C
  - In-memory hash table
  - Multi-threading



Memcached can be treated as a user-space mini-OS

What is a Process?

What is a Process?

- Programs are code (static entity)
- Processes are running programs

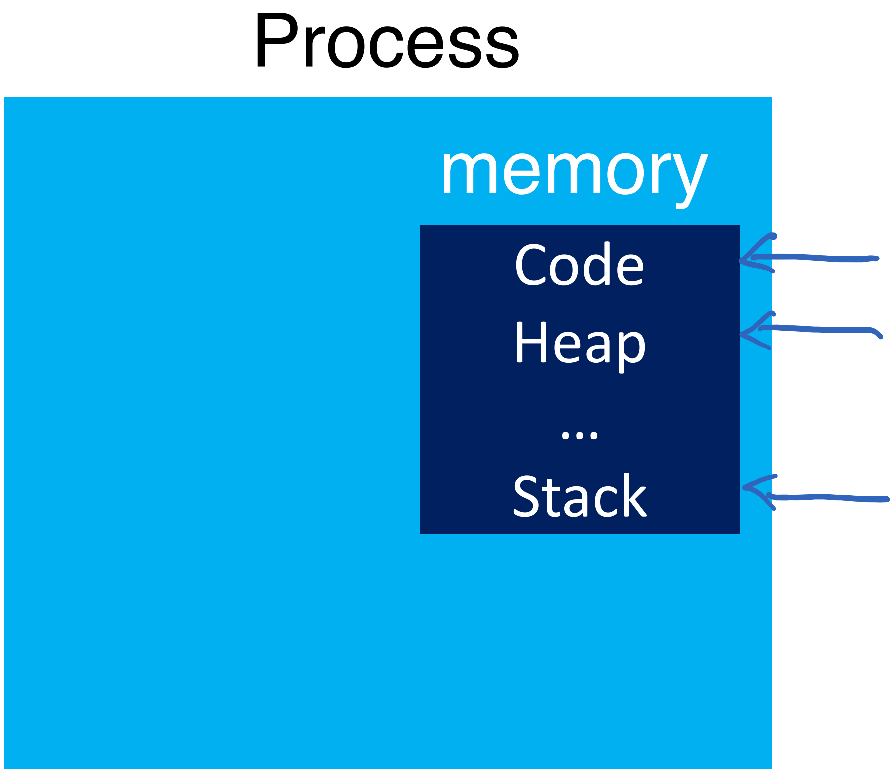
- Java analogy
  - class -> “program”
  - object -> “process”

What is in a Process?

Process

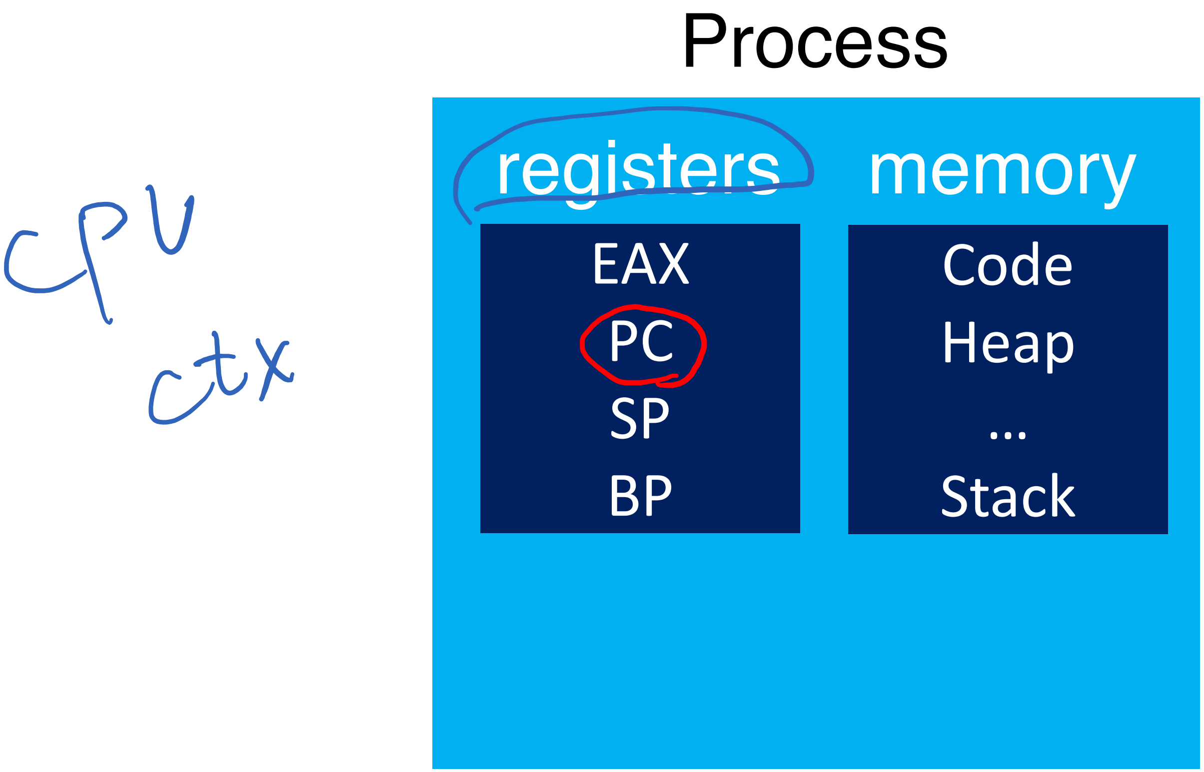
What things change as a program runs?

What is in a Process?



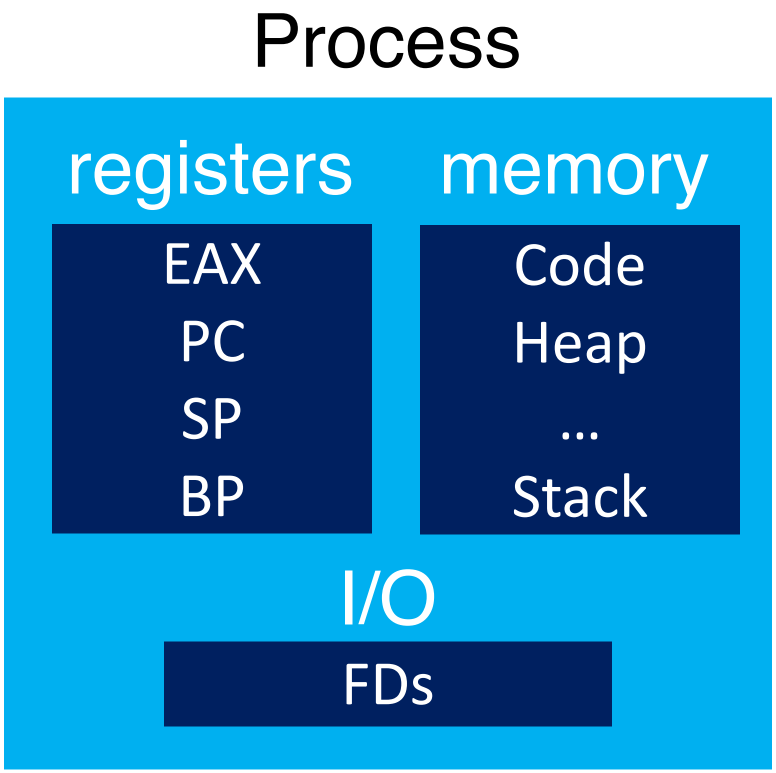
What things change as a program runs?

What is in a Process?



What things change as a program runs?

What is in a Process?



What things change as a program runs?

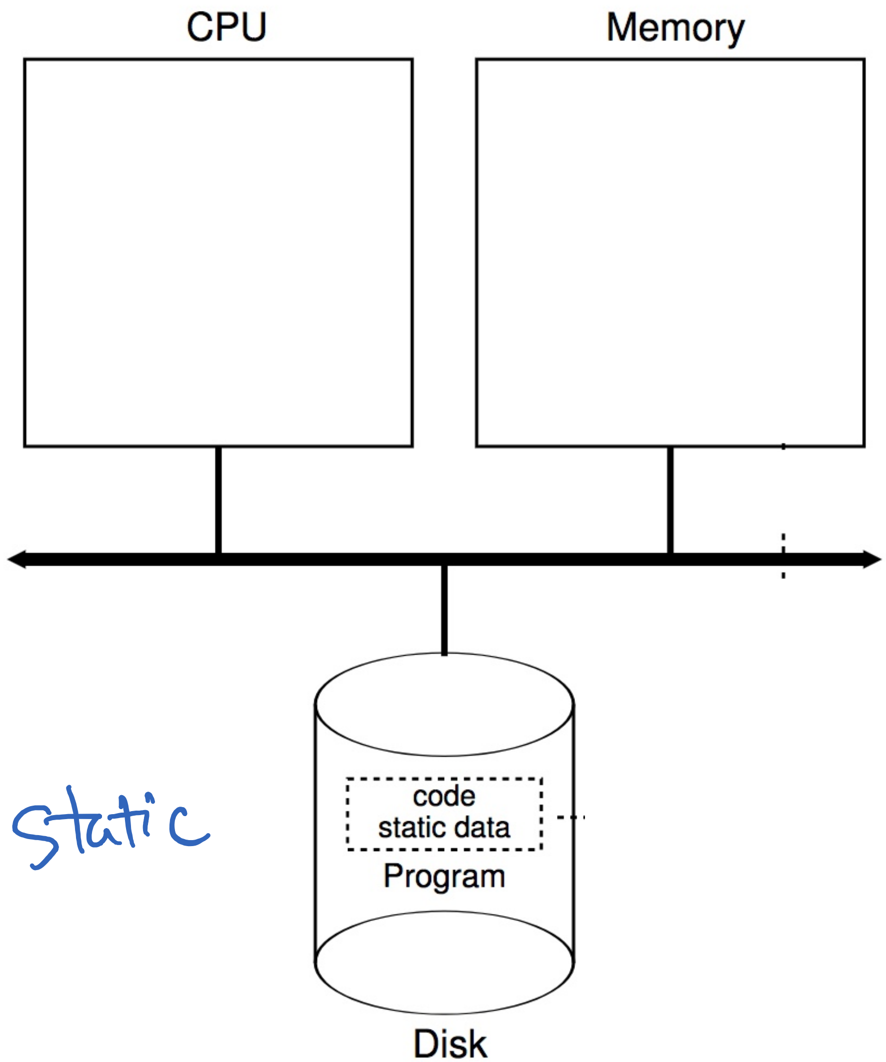
Peeking Inside

- 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/*`

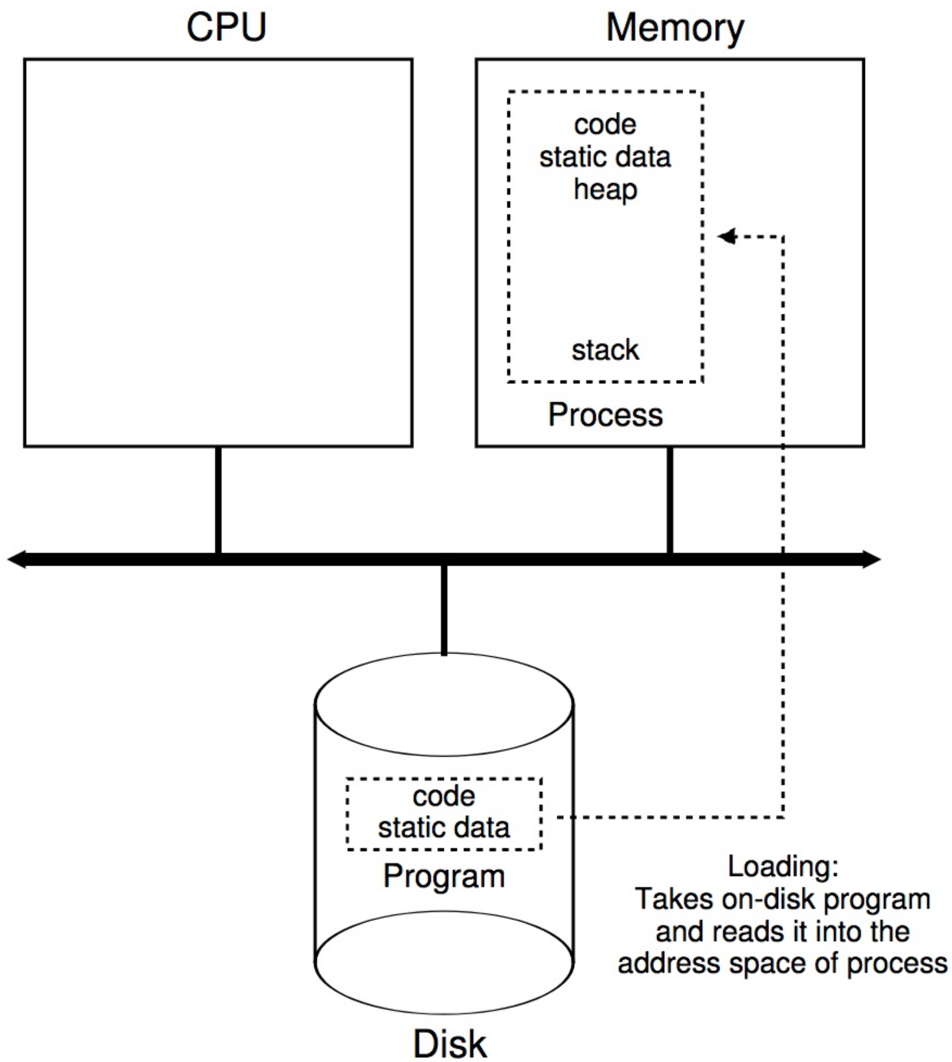
Process Creation

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

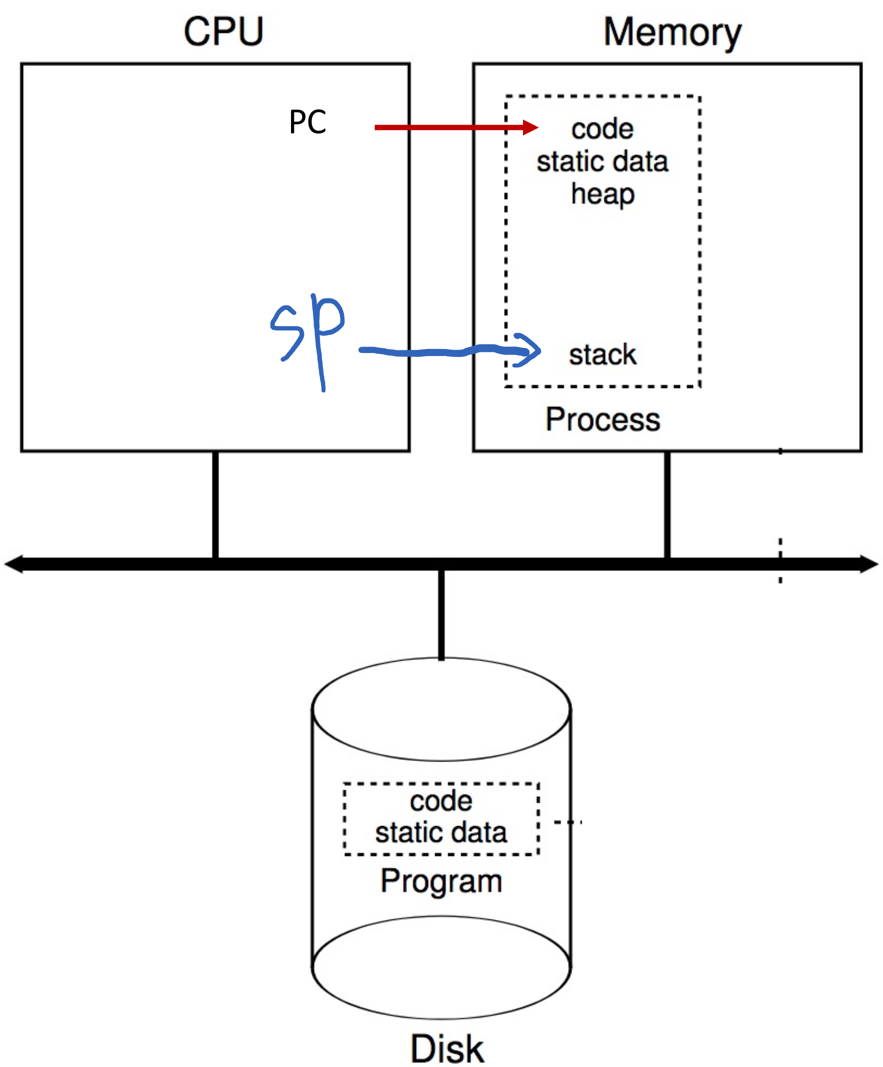
Process Creation



Process Creation



Process Creation

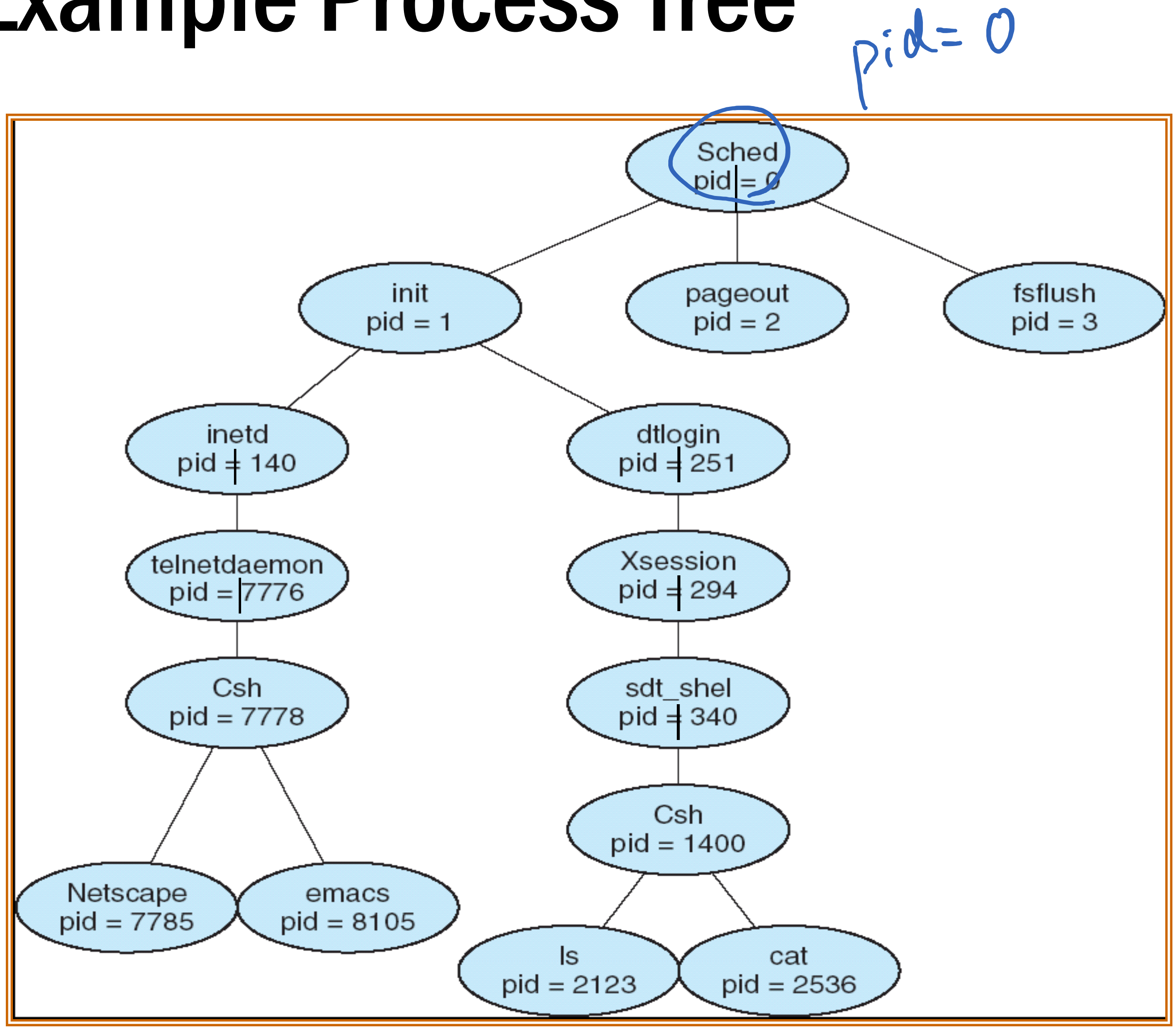


Process Creation (cont.)

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

- 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?

An Example Process Tree

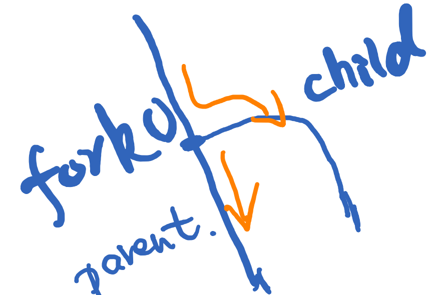


How to View Process Tree in Linux?

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

- `% pstree`

Process Creation 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 `execl()` 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 `execl()` to load a binary file into memory

Simply the return value of fork() in the context of the new child proc

This is really 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.
```

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

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, the number is %d\n", number);
    }

    return 0;
}
```

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

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        fork();
        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, the number is %d\n", number);
    }

    return 0;
}
```

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

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the execl example\n");
    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        printf("\tIn the execl child, PID is %d\n", pid);
        execl("./forkexample2", "forkexample2", NULL);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, done waiting\n");
    }

    return 0;
}
```

Results

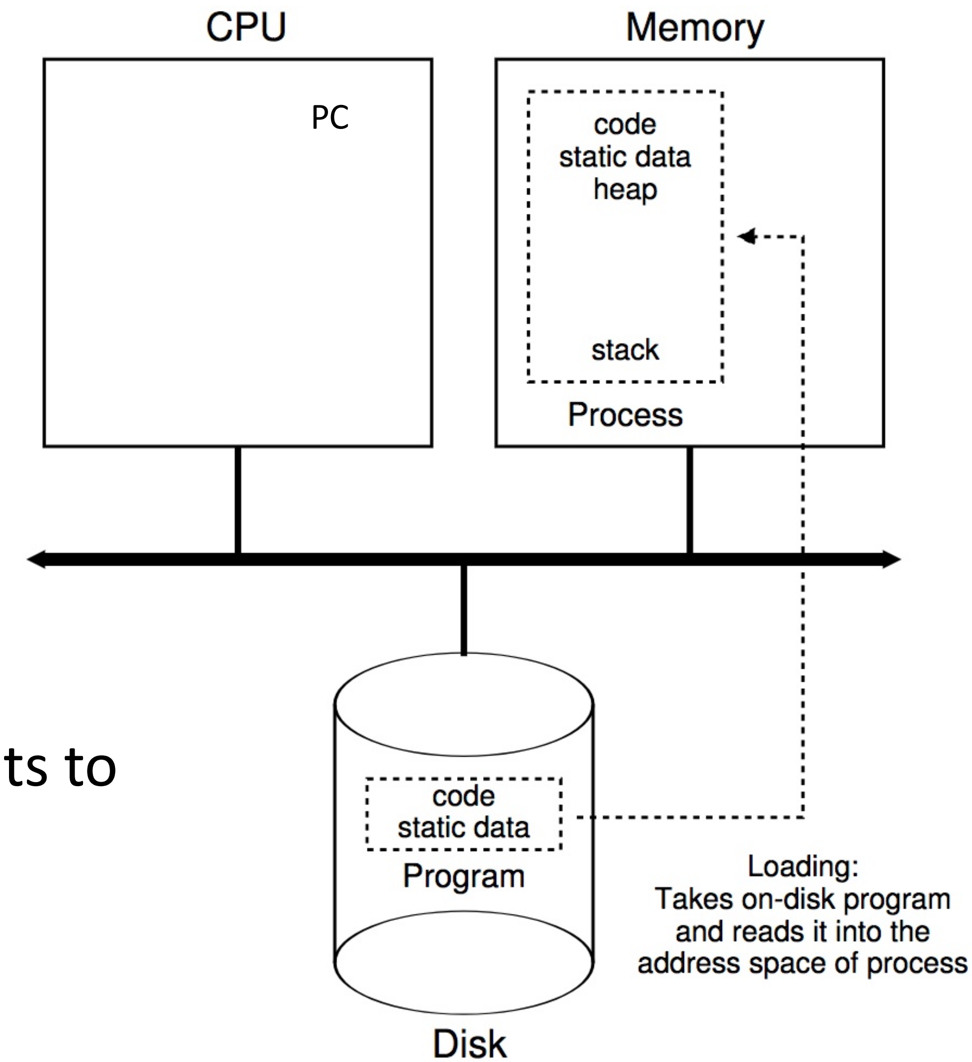
```
./execlexample
Running the execl example
 PID is 2179
 PID is 0

                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

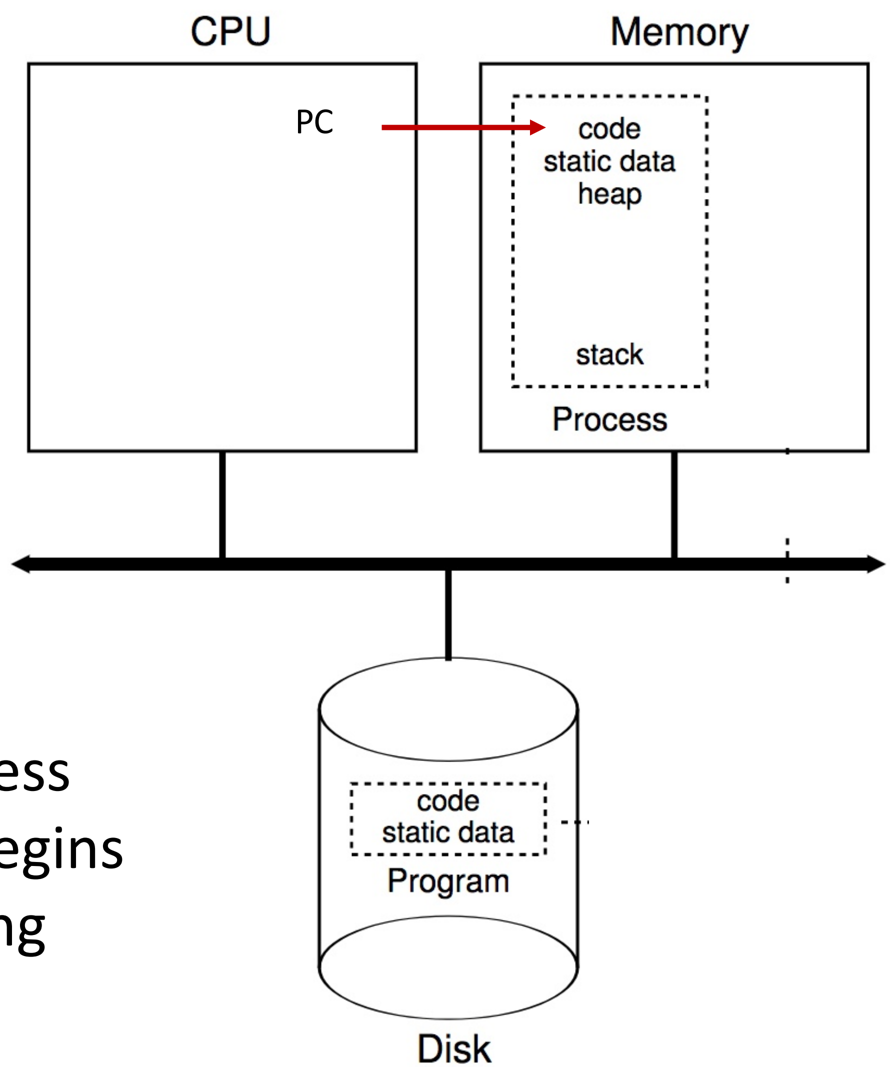
- You can fork it here: https://github.com/tddg/demo-ostep-code
  - under `cpu-api/`

Process Creation



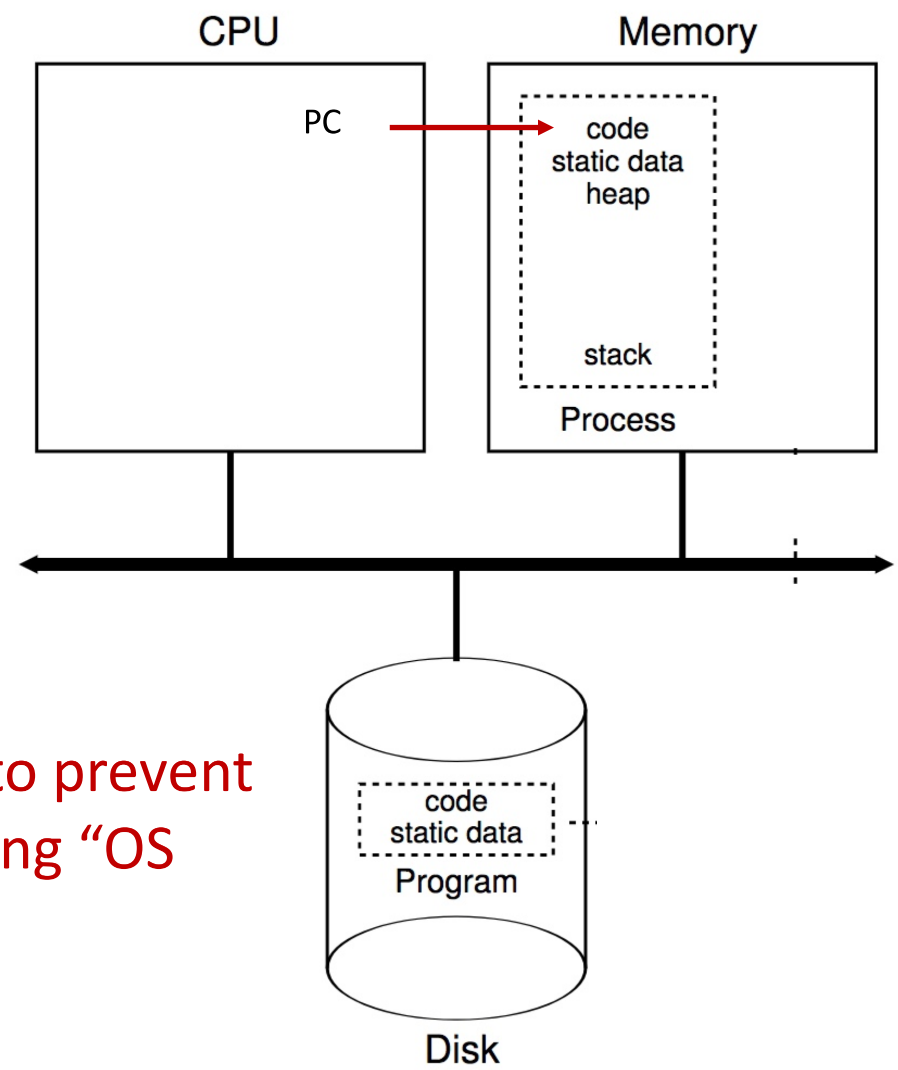
Before, PC points to kernel code

Process Creation



Now, after process creation, CPU begins directly executing process code

Process Creation



Challenge: how to prevent process from doing "OS kernel stuff"?