EECS-343 Operating Systems
Lecture 1:
Introduction

Steve Tarzia

Spring 2019

Northwestern

Today’s lecture

- What will you learn from this class?
- How will this course be operated?
- Are you ready to take this class?

- What is an Operating System?
- History of computers and their operating systems

Why study OS?

- You will probably never write a new OS (please don't!)
  - But you may need to *modify* an OS or write device drivers for new hardware.
- More importantly, to understand *application* software **performance**, you must understand what the OS is doing behind the scenes.
- You'll also get some good software engineering experience.
  - This is likely the first class where you'll have to modify a large, complex, existing codebase.
  - Get practice with C, Linux, and tools like git, gcc, gdb, make, etc.)

Important Topics

- Security
  - How processes (apps) are isolated from each other when running together.
  - Eg., how the recent *Spectre* and *Meltdown* attacks work
- Performance
  - How virtual memory & paging affects read/write latency
  - How processes compete for shared resources
- Concurrency
  - Synchronization (how processes coordinate using locks, semaphores)
  - Non-determinism (how *race conditions* can lead to weird bugs)

Course logistics

- We'll use Canvas for assignments, announcements, lecture slides.
- I will try to post videos of all lectures
- Staff:
  - Steve Tarzia, instructor
  - Teaching Assistants:
    - Yingyi Luo
    - Xutong Chen
    - Kaiyu Hou
  - Peer Mentors:
    - Richie Lee
    - Michael Hsu
    - Peter Bi
    - Rohit Rastogi
    - Ziqin Xu
    - Tianhao Zhang
- TA & PM office hours in Mudd 3303:
  - Mondays 1-7pm, Wednesdays 1-4pm
- Ask all questions on Piazza (not by email)
- Final exam is Monday June 10th at 3pm. Midterm date TBA soon.

Prerequisites

- Data Structures (EECS-214)
- Basic C/Unix programming (EECS-213)
- Basic assembly coding (EECS 213 or EECS-205)

- If you're unsure, then start the first project ASAP
- If you're going to drop, better to drop sooner than later

Collaboration & cheating policy

- Helping each other is OK, but you should not do anything that allows another student skip the learning process.
- You may talk to other students about the homework and projects
- You may **look** at **small parts** of each other's code (on the screen)
- You **may not** send a *copy* of your code to friends
- You **may not** post your code to a public git repository.
- If you copy code from the Internet, you must add a comment explaining the source.
- You must understand what your code does
- I will use *MOSS* to compare your code to everyone else's code and prior years' code.
- If you spent fewer than 6 hours on a project, then you probably cheated!
- I will notify the Dean of blatant cheating and you may be expelled from Northwestern.
- If you're unsure about this policy, please ask me.

Grading

- 50% -- 4 Projects
  - C kernel development.
  - *The most difficult part of the course!*
- 10% -- 4 Homework assignments
  - Short written answers, based on reading & lectures
- 15% -- Midterm exam
- 25% -- Final exam (cumulative)
  - Exams are similar to homework assignments

C language is good for building Operating Systems

- C can interact with hardware directly
  - C code can include some assembly code, when necessary.
  - Assembly code gives access to instructions for low-level stuff like:
    - interrupts, CPU registers, changing CPU modes.
  - C gives direct access to memory to create interrupt tables, etc.
    - A C pointer is a number specifying a location in memory.
- C compiles directly to machine code
  - It does not require any runtime translators and libraries
  - Behavior is reasonably predictable (no weird garbage collection processes)
  - Can inspect the resulting machine code to tweak performance
- C is very efficient

C was wonderful in the 1970s and 80s, but…

- 40 years later, it feels inconvenient and dangerous.
- Like shaving with a straight razor.

C

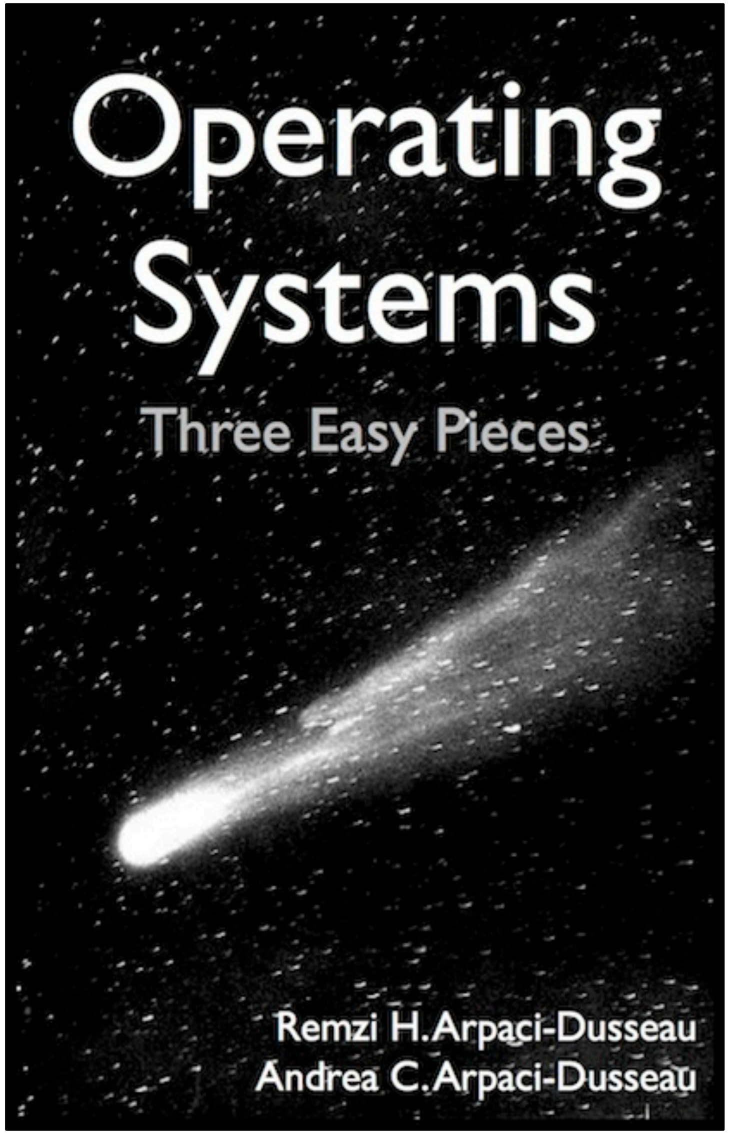
Python

- Lacks standard dynamic collections (lists, dictionaries)
- Must manage memory yourself (malloc & free)
- Cannot easily write generic code
  - Pretty strongly typed, and no inheritance
- Cannot throw/catch exceptions
  - Must check function return status explicitly
- Function parameters often contain return values
  - Have to pass pointer to pre-allocated buffer as a parameter, and choose buffer size
- No good, free IDE. You will use something like emacs, vim, or gedit + make + gdb

Lab hours & Office hours

- TAs and Peer Mentors will hold office hours in Mudd 3303 (to help you with the projects)
  - Mondays 1-7pm
  - Wednesday 1-4pm
- Steve's office hours in Mudd 3225:
  - Mon 1-3pm, Tues 3:30-4:30pm, Wed 3-5pm, Thurs 3:30-4:30pm

Required reading



- 666 page, “open” textbook
  - Chapter PDFs are online
  - $10 for one big PDF
  - $22 softcover
  - $38 hardcover
- Entertaining read
- Repeats & reinforces lectures
- For homework and exams
- **Buy a hard copy** because exams are open book!

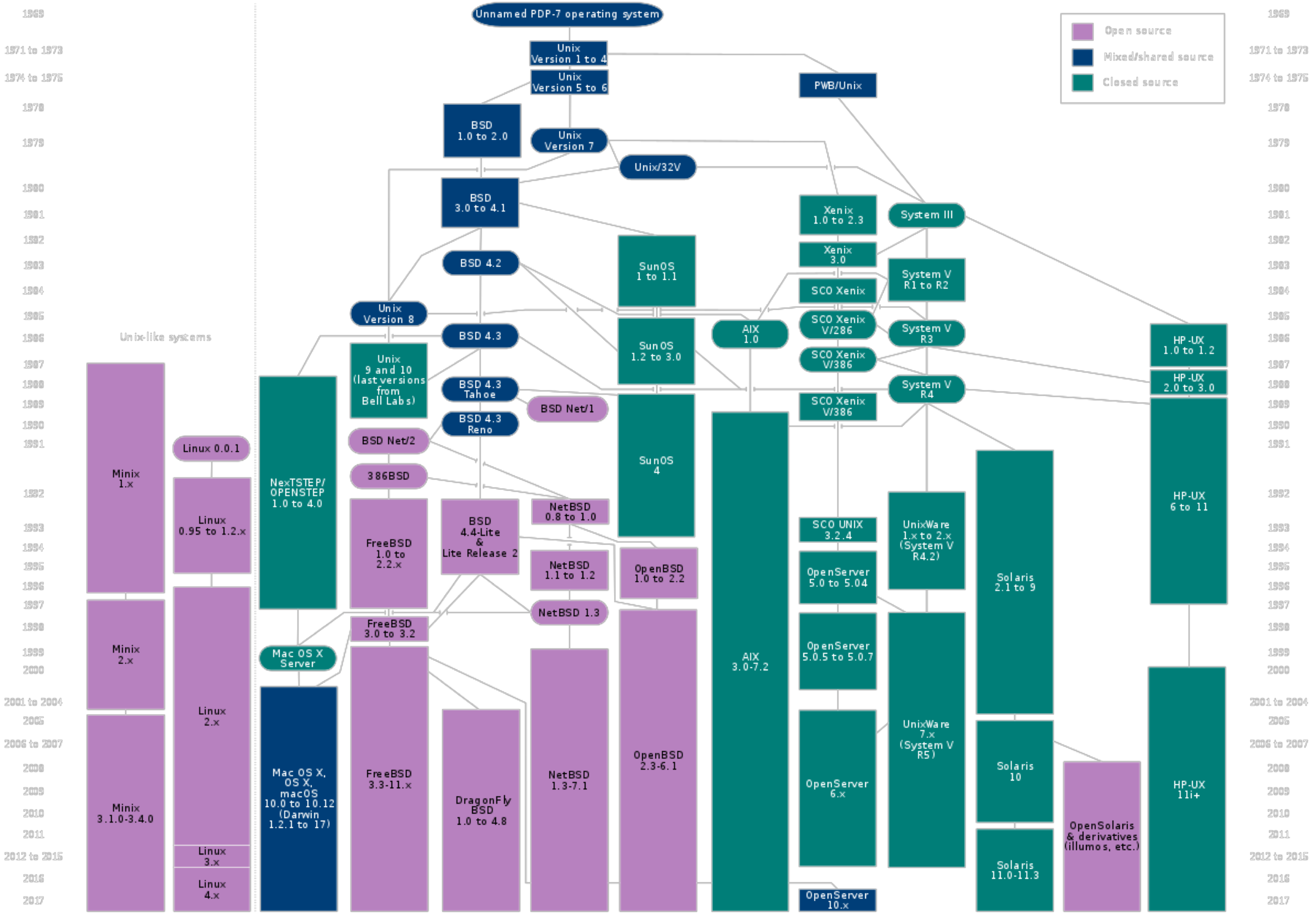
About Steve

- Second year teaching at Northwestern.
- PhD from Northwestern in 2011, BS from Columbia in 2005, both in Computer Engineering
- Research expertise is acoustic sensing on mobile systems.
- Worked at a few Chicago area software startups
  - First as a software engineer, later as VP of Engineering
- Have published about ten iOS apps in the app store
- Fan of Linux, AWS, Java, Python, Objective-C
- Founded the National Gun Violence Memorial (GunMemorial.org).
- Enjoy competitive lap swimming, cooking, repairing things, opera, and playing music (guitar, drums, trumpet).
- Married 14 years & have lived in Evanston for the past 8 years.

Operating Systems

- *From the past:*
  - IBM OS/360, AT&T Unix, DOS, MacOS "classic", Windows 3.1, 95, XP, 2000
- *And in the present:*
  - Windows 7, 8, 10
  - MacOS X
  - Linux
  - BSD
  - Android
  - iOS

Unix family tree



xv6

- A small teaching Operating System, used for the course projects
- Less than 80,000 lines of code
- Instead of about ~15 million for Linux
- Very limited functionality
- It does really run on very basic PC-compatible hardware
- We'll be running it on an *emulated* machine (Qemu)
- Xv6 lacks drivers for most real hardware accessories/peripherals (can't use network cards, most storage devices, graphics)

How is an OS different than a software application?

Pair and share

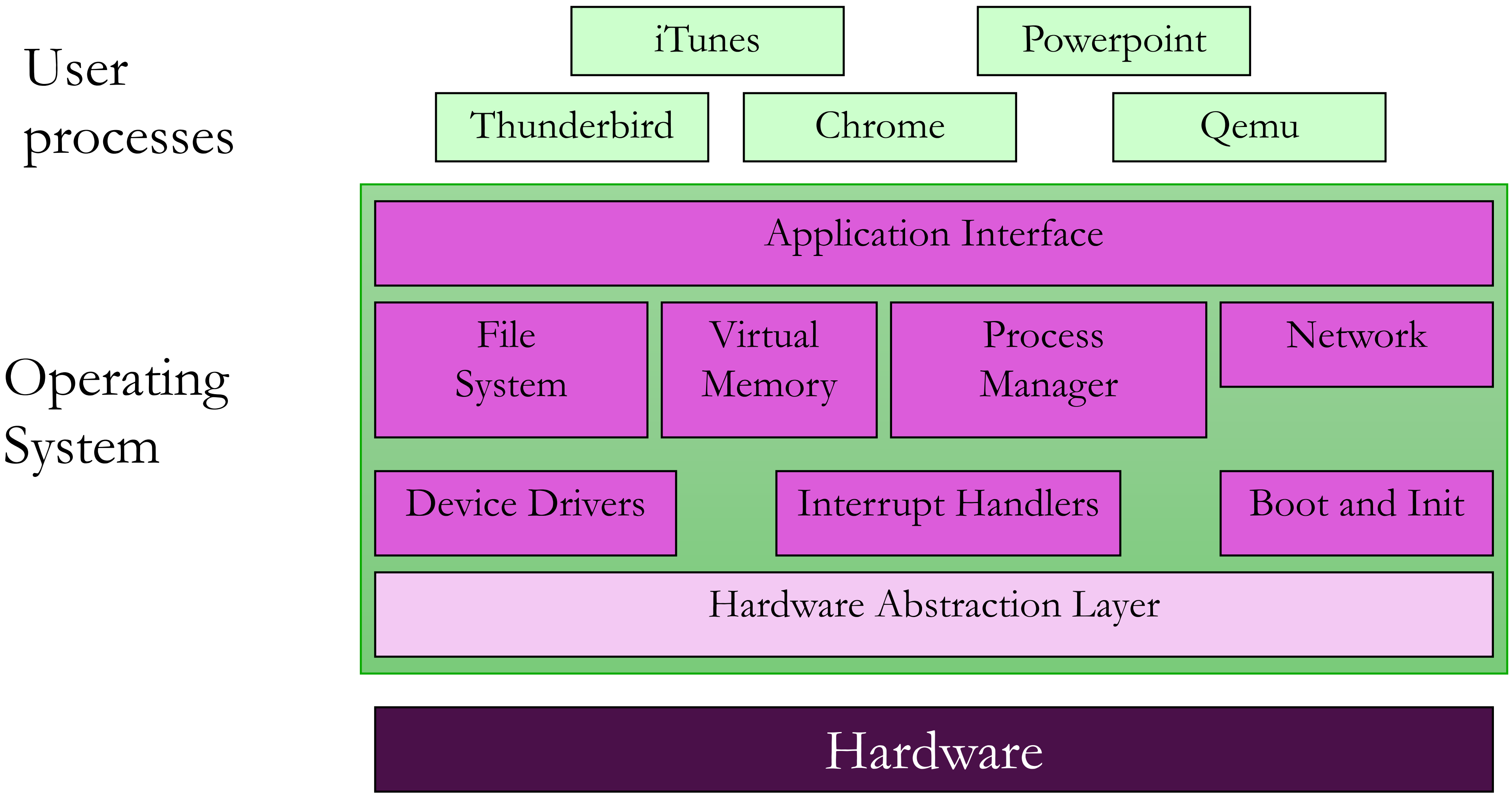
Roles of an OS

- A ***user interface*** for humans to run programs
- A ***resource manager*** allowing multiple programs to share one set of hardware.
- A ***programming interface*** (API) for programs to access the hardware and other services.

Before operating systems

- User could only run one program at a time.
- Had to insert the program disk before booting the machine.
- Program had to control the hardware directly
  - This is a nuisance because hardware is complicated
  - Program will only be compatible with one set of hardware
- For example (at right) 1983 “King’s Quest” game for IBM PC Jr.

OS sits between hardware and your apps



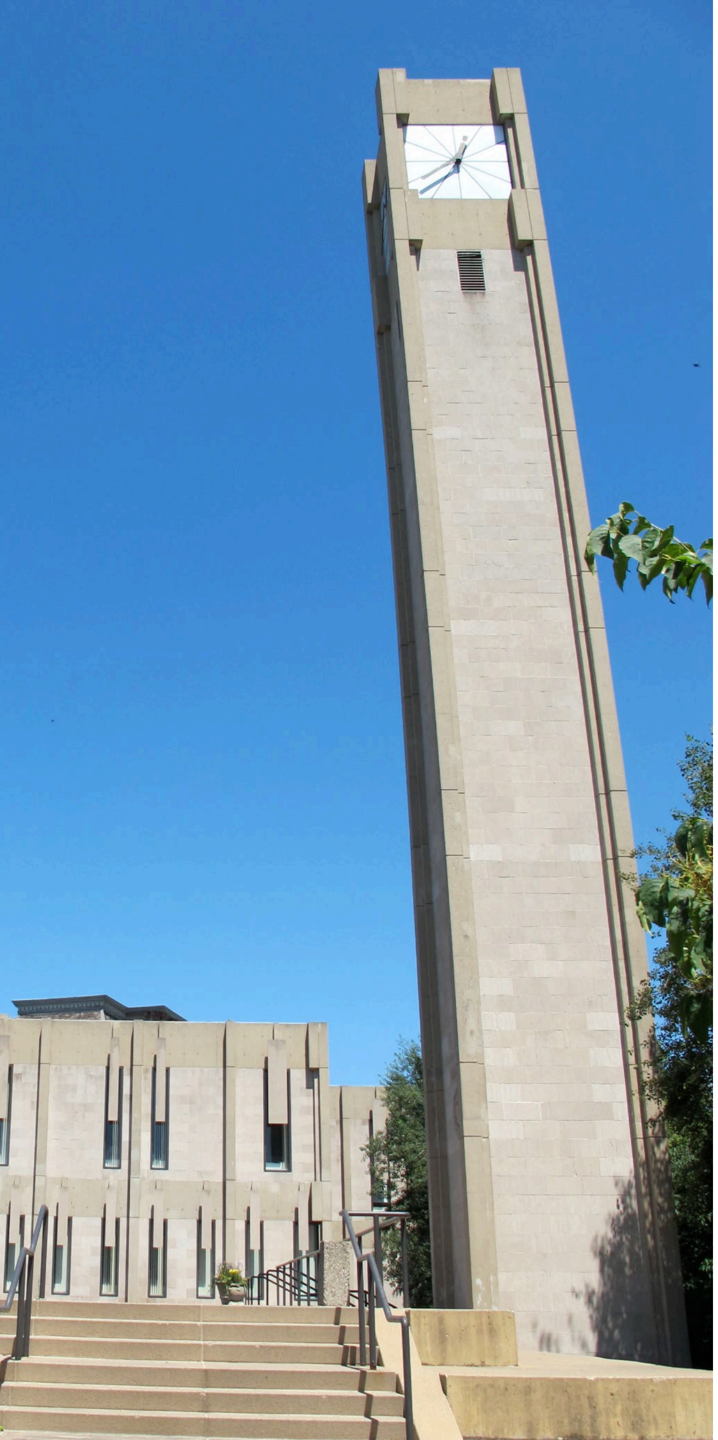
What's part of the OS? – hard to define!

Kernel – the only code without security restrictions

- Process scheduling (who uses CPU)
- Memory allocation (who uses RAM)
- Accesses hardware devices
  - Outputs graphics
  - Reads/writes to network
  - Read/write to disks
  - Handles boot-up and power-down

OS distribution – the kernel + lots of other useful stuff

- GUI / Window manager
- Command shell
- Software package manager
  - "app store", yum, apt, brew
- Common software libraries
- Useful apps:
  - Text editor, compilers, web browser, web server, SSH, anti-virus, file-sharing, media libraries,

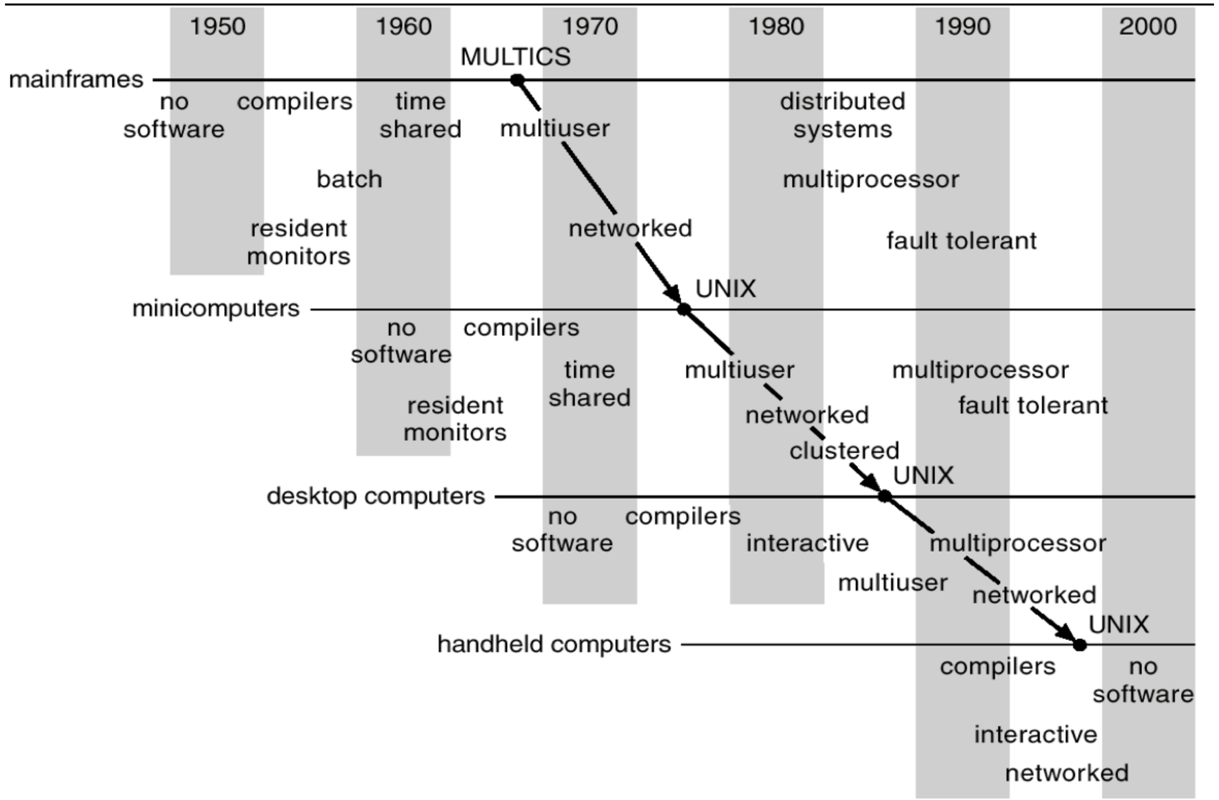


← Ivory tower

In the academic world,

when we say "OS," we usually mean just the OS *kernel*.

Operating systems have evolved with hardware



- Sophisticated operating systems first arose on mainframes.
- OS ideas migrated to smaller machines as those machines became more powerful.
- In 2019, a **smart watch** has 1gb RAM, 16gb SSD storage, two CPU cores, and a real OS.

Early evolution of computing systems

- 1955: Batch systems
  - Collect a bunch of program punch cards and write them all one matgetic tape.
  - Run the tape through the mainframe to execute all the jobs in sequence.
- 1960s: Multiprogramming (IBM OS/360)
  - Keep multiple runnable jobs in memory at once.
  - Allows overlap I/O of one job with computing of another.
    - Uses asynchronous I/O and interrupts or polling to detect I/O completion
- 1960s: Timesharing (MULTICS, Unix)
  - Multiple user terminals connected to one machine
  - Allows *interactive* use of machine to be efficient (because another user's job can run while you're thinking).

Later evolution of computer systems

- 1970s: Parallel systems
  - Processes must communicate to share information.
  - Synchronization is difficult.
- 1980s-90s: Personal Computers (IBM PC, Macintosh)
  - Graphical user interfaces were developed
  - Mainframe OS concepts (like networking) were applied to PCs
  - Magnetic disks become huge, but still slow
- 2000s-10s: Mobile and pervasive computing, Cloud Computing
  - Slow hardware is once again common (phones & wearables)
  - OS manages sensitive information like location and Internet behavior
  - Fast flash storage is common.
  - Server hardware is shared by many different cloud computing customers

Recap: OS is a resource manager

- A computer has many apps (processes) running
- All need access to:
  - CPU (processor)
  - RAM (memory)
  - Storage (disk/filesystem)
  - Other hardware like Network card, display, sound card.
- OS provides safe, fair sharing of the limited hardware resources.

Your first tasks

- Buy the book
- Find a project partner
- Start Project 1 – *as soon as possible!*
    - Project 1 parts 0 and 1 are *due on Monday!*
    - Email root@eecs.northwestern.edu if you have trouble with lab machines
    - Part 2 (the difficult part) will be posted soon (tomorrow).

After that:

- Start reading chapters 1-6
- Watch old timesharing video (1963, Corbató @ MIT): https://www.youtube.com/watch?v=Q07PhW5sCEk