Why Study the OS?

•

What are your thoughts?

- What is interesting?
- What is intimidating?
- What is confusing?
- What feels like magic?

Operating Systems

Why Study the OS?

- The core of all programs
- Virtualization, abstractions – a “magician”
- Interface between hardware (devices, memory, etc.) and software (programs, functions)
- Large, complex system (challenging)

Operating Systems

What is an OS?

- No accepted definition
- Two common pieces:
  - Kernel (protected software)
  - Libraries (shared, unprotected software)
- Together, these create the “world” for your program
  - What it can do, how it thinks

Operating Systems

What does an OS do?

- Coordinator
  - Assigns resources to programs fairly
  - Manages sharing, enforces protection
- Standard abstractions & routines
  - Builds complex things (ex: files, directories)
  - From simple things (ex: blocks on disk)

Operating Systems

What does an OS do?

- Coordinator

What are some things that must be shared and protected?

Operating Systems

What does an OS do?

- Coordinator
- Abstractions & routines

What are some abstractions that we find in a computer (things that don’t literally exist) ?

Operating Systems

What does an OS do?

- Coordinator
- Abstractions & routines

What are some algorithms and routines that an OS or library might implement, and your program can call?

Operating Systems

The Problem:

How to build a system that is

- Safe
- Fast
- Flexible

but makes it easy to write complex programs?

Operating Systems

History

OS - History

- No OS in the first computers
- Ran a single program
  - Originally, programmed by plugging wires!
- Reboot to run another

- Was still common in personal computers until late 80s, early 90s !

OS - History

- Added shared libraries enhanced functionality
  - Drivers (ex: printers)
  - TSRs

- Zero protection
- All code has the same permissions
- Still only using one program at a time

OS - History

- Batch Computing
  - Submit your program to a queue
  - Runs when it’s your turn
  - Complete control of the computer while you run
  - Reboot the computer to start the next job

- Allows users to share hardware
  - Painful to use, long latency on runs

OS - History

- Time Sharing
  - Multiple programs running at the same time
    - One user? Or many at the same time?
  - Shared resources: CPU, memory, disk, I/O
    - Need to protect users from each other
  - Need ways to control resource hogs, infinite loops
    - Enforced sharing? Or polite request?

- Common in business/academic computers in the 70s; common in personal computers in 90s

OS - History

- Devices Gradually Get Smarter
  - Devices start doing non-trivial work on their own
    - Don’t need CPU except occasionally
    - Often have a small on-board CPU (later: GPU)
  - Printers
  - Sound cards
  - Graphics
  - Disks
  - Network

OS - History

- Modern OSes
  - 100s of programs at once
  - Sometimes, many users at once, through networks
  - Programs can die in microseconds, or live for years
  - Pre-emptive multitasking

- Now the norm in all computers except microcontrollers. Even the Raspberry Pi has pre-emptive multitasking and strict user control!

OS - History

- Multi-core, Multi-processor
  - Multiple CPUs running at the same time
  - Share the same memory
  - Hordes of race conditions

- Have existed almost since the beginning
  - Mostly only in servers until late 90s
  - Now, in all computers other than microcontrollers

OS - History

- Distributed Computing
  - Multiple physical computers
  - Each computer has its own OS
  - Libraries make it easy to communicate
  - Some programs think of the entire system as one unit

OS - History

- Virtual Machines
  - A program simulates a computer
    - Including attached hardware (disk, keyboard, etc.)
  - Install & run real OSes inside it
    - It controls its own programs
- Newer: Containers
  - Run programs inside an ordinary OS
  - Lock down what it can see about other programs, files
  - Almost as good as a VM, and **much** cheaper

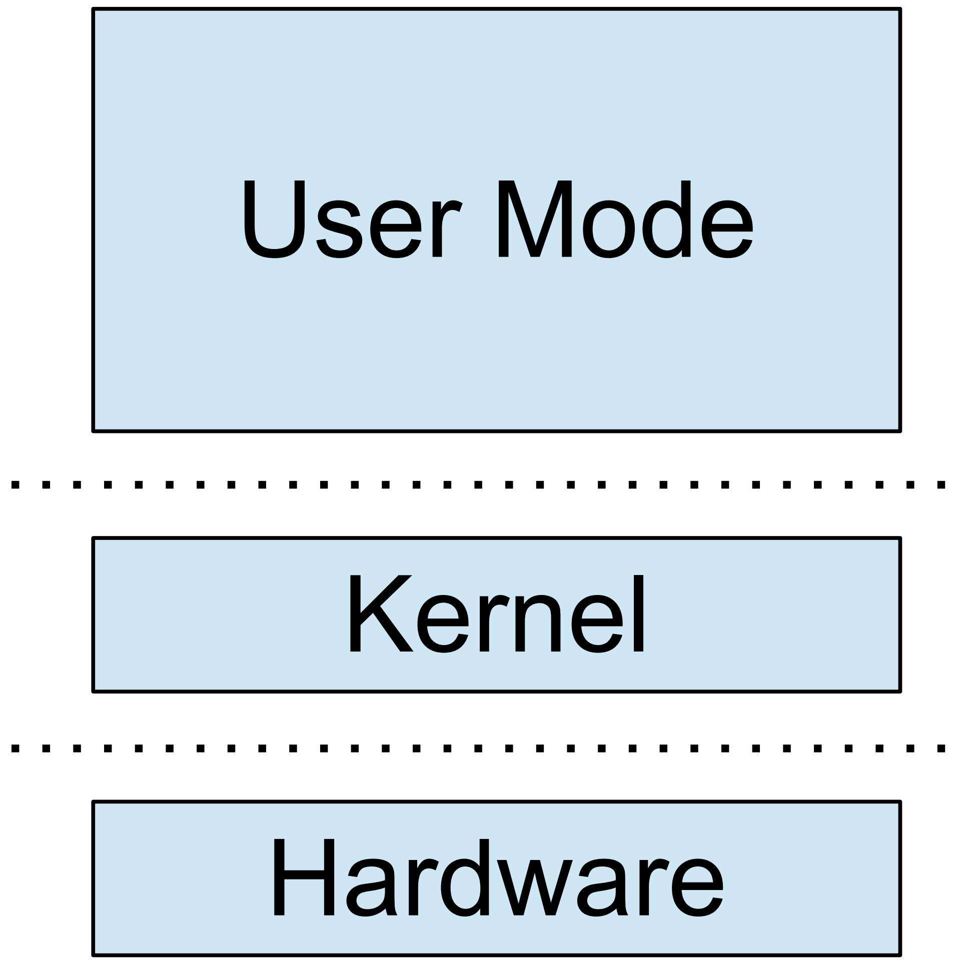
OS - History

- Our simulator (USLOSS)
  - Not even a VM
  - Just a toy for playing around with OS concepts
  - Let’s you simulate a few things about an OS
  - But lots of hacks to make it simple

Operating Systems

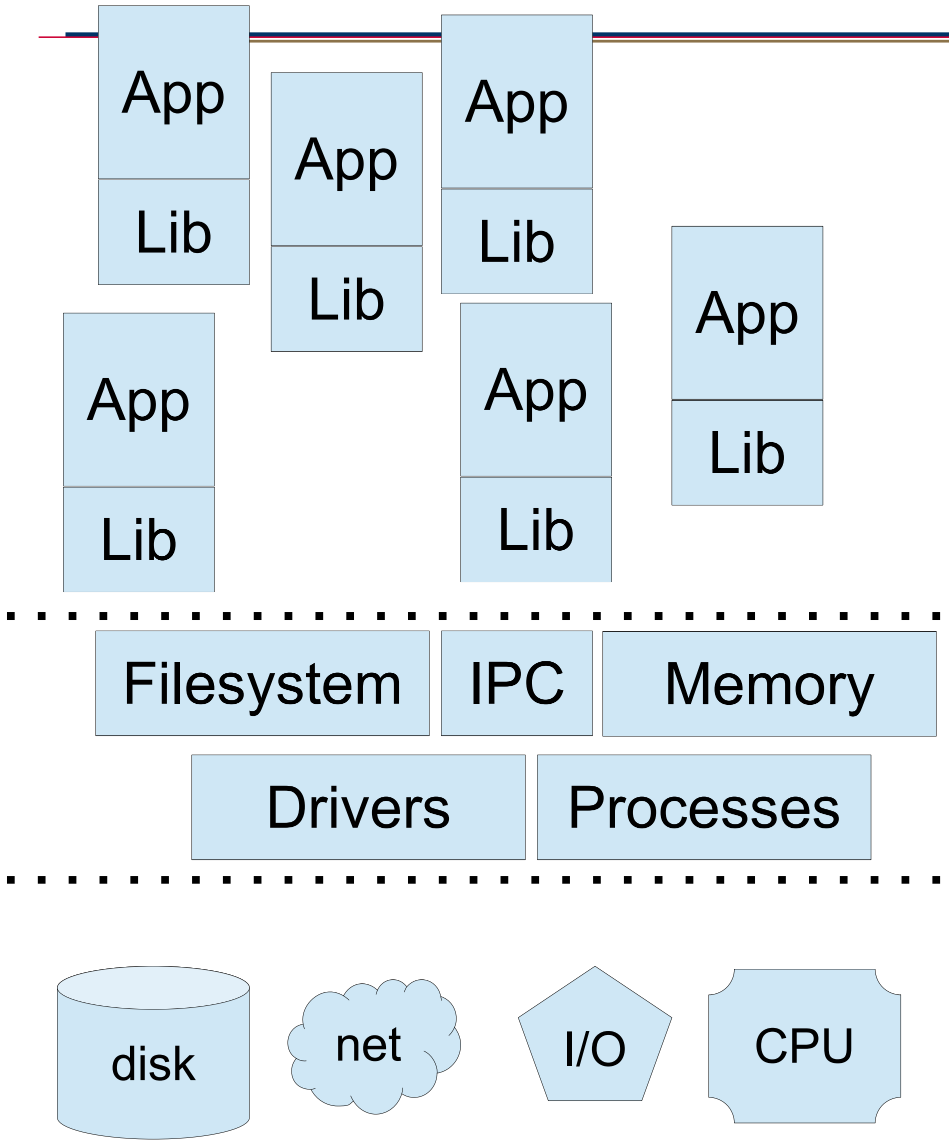
Components

OS Components



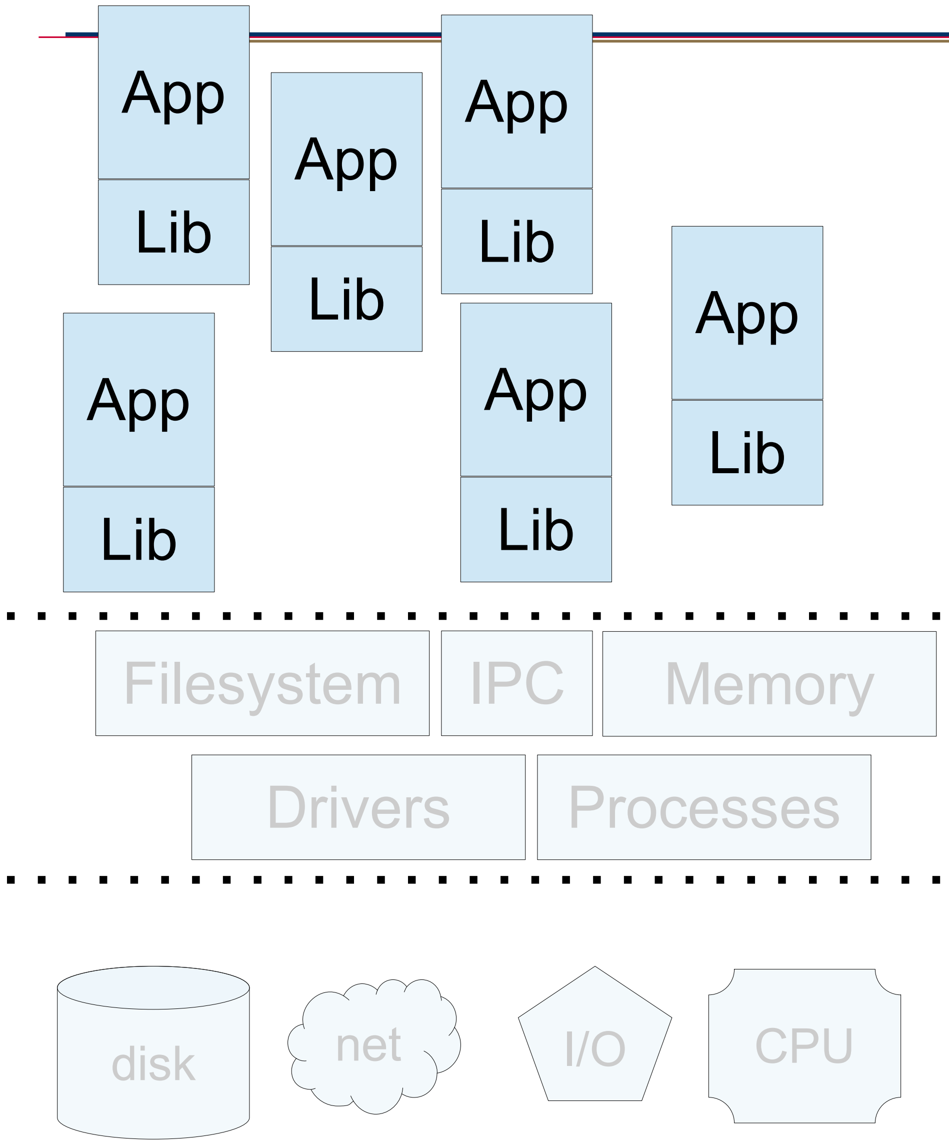
- Extremely simplified picture!

OS Components



- Huge complexity

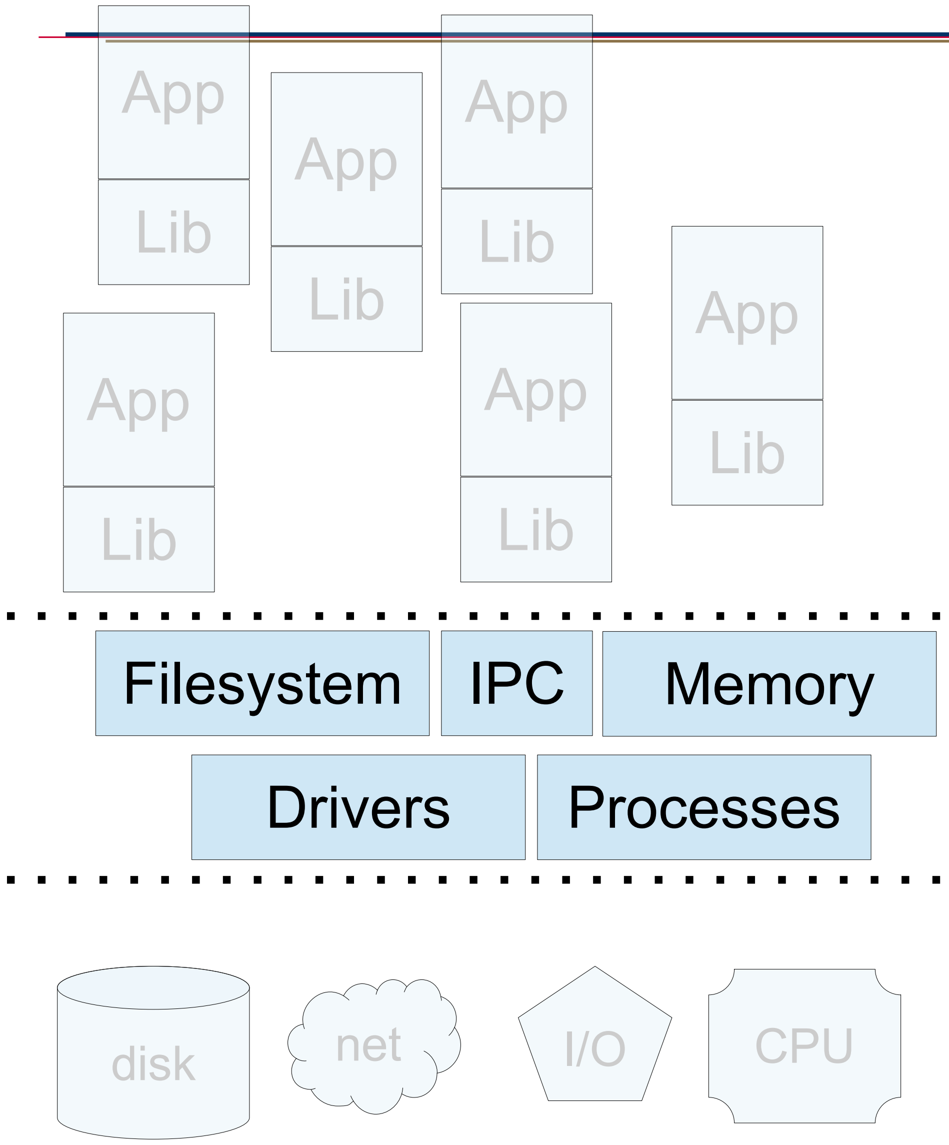
OS Components



User Mode

- Does most work
- Includes most system tools
- Can only access CPU & memory
- Asks OS for anything else (syscalls)

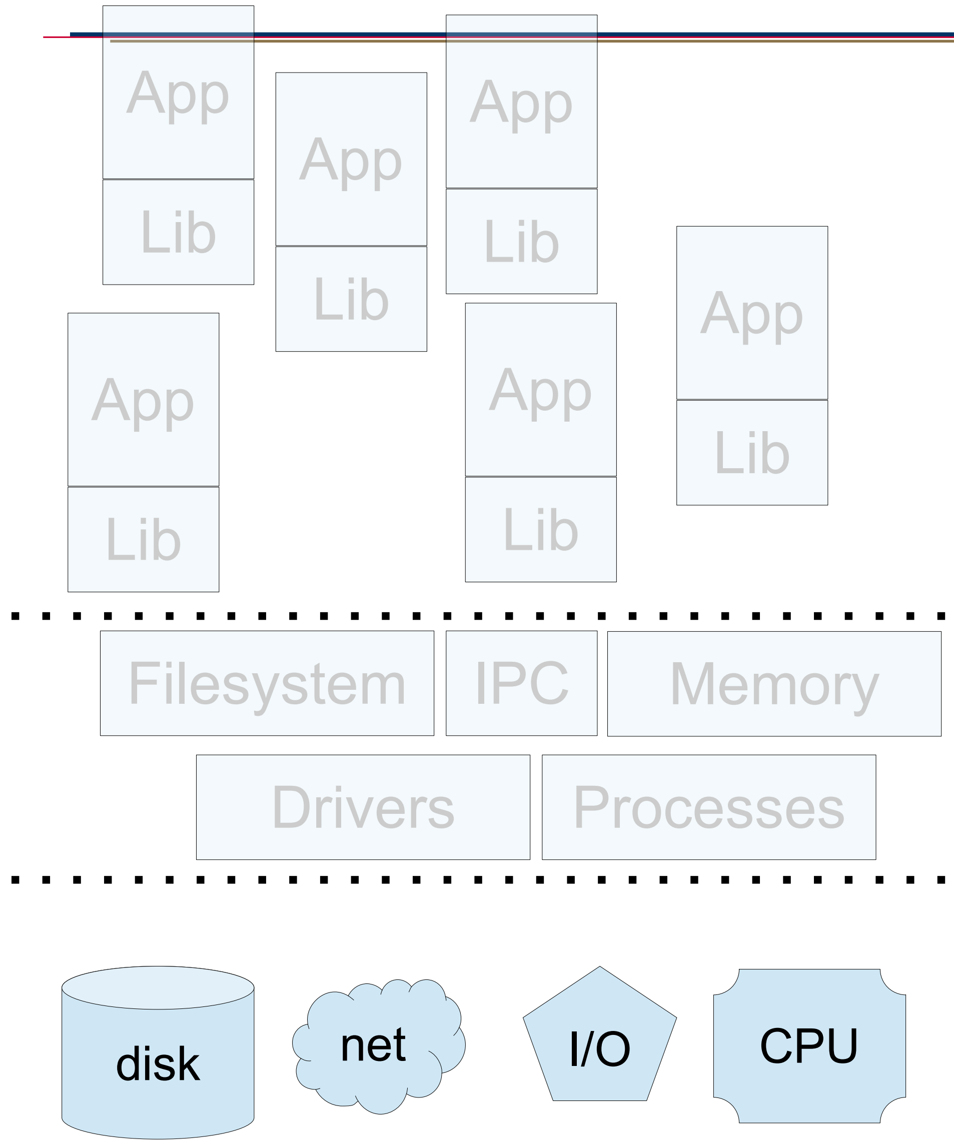
OS Components



Kernel

- Configures hardware
- Chooses which process will run
- Assigns memory
- Handles syscalls
- Answers device requests

OS Components



Hardware

- Mostly stays idle until kernel gives instructions
- Sends interrupts when events occur or operations complete

OS Components

- Example: Compiler
  - Reads one (or many) input files
  - Converts to machine code
  - Writes object file to disk

- Extremely simplified!

OS Components

- Example: Compiler
  - Reads one (or many) input files
  - Converts to machine code
  - Writes object file to disk

How many different small steps can we think of?

Which of these happen in the user mode, which happen in the kernel, and which in hardware?

(And of user mode, which are libraries?)

OS Components

- Compiler reads a file
  - `fopen()` to try to open the file
    - Is in a library
    - Calls `open()` syscall
  - `open()` runs in the kernel
    - Asks filesystem if it’s OK to open the file
  - Filesystem tries to open the file
    - Reads disk: does it exist?
    - Reads disk: who owns it, what are the permissions?
  - Kernel creates “file handle”, and returns

OS Components

- Compiler reads a file (part 2)
  - `fread()` to read data from the open file
    - Is in a library
    - Calls `read()` syscall
  - `read()` runs in the kernel, in the filesystem
    - Checks file handle: where are we in the file, right now?
    - Reads disk: how large is the file?
    - Reads disk: what are its contents?
  - Copies data from disk into the user memory

OS Components

Pause to think…

It seems that almost everything that the compiler does involves lots of OS work. Is this always true?

OS Components

- Most programs mix CPU-heavy work with I/O-heavy work

- CPU-heavy
  - Little OS interaction required
  - May need OS to provide memory as data grows
- I/O-heavy
  - Dominated by syscalls
  - Often blocking

OS Components

- Our Compiler
  - Reads one (or many) input files (I/O)
  - Converts to machine code (CPU)
  - Writes object file to disk (I/O)