COS 318: Operating Systems

Lecture 2:

Continuation of Introduction
Overview of Operating Systems

Jaswinder Pal Singh
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Logistics

- Precepts:
  - Wed: 8:30-9:30pm, 105 CS building
- Please check times for Design review and Assignment 1 due date on the Web site
- Reminder:
  - Register for the cos318 mailing list today!

Today

- Overview of OS structure
- Overview of OS components

Previous Lecture

- Course Staff and Logistics
- What is an operating system?
- Evolution of computing and operating systems
- Why study operating systems?
- What's in COS 318?

Today

- Evolution of computing and operating systems
- Why study operating systems?
- What's in COS 318?
- Overview of Operating Systems

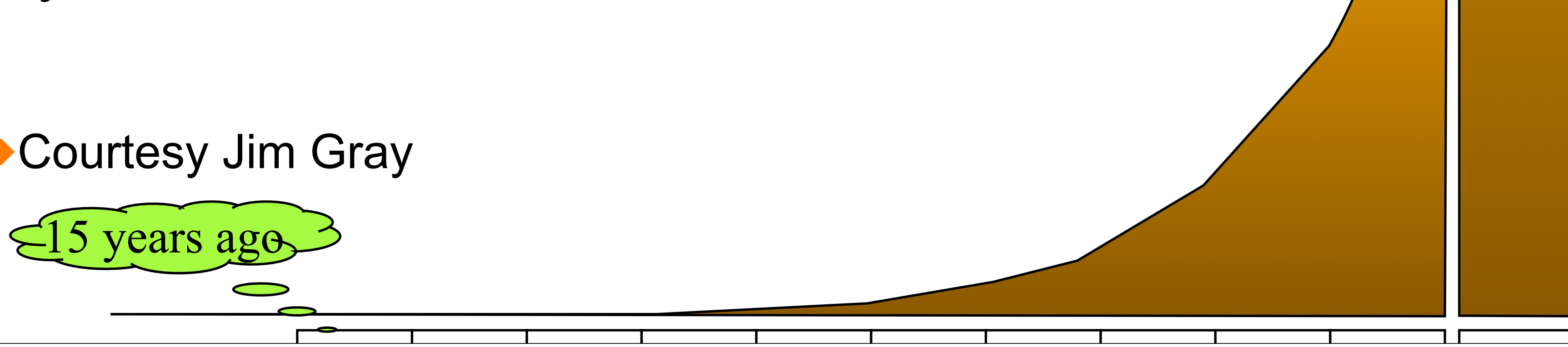
A Typical Academic Computer (1986 v. 2007)

	1986	2007	Ratio
CPU clock	4Mhz	4×3Ghz	3000x
$/machine	$60k	$600	1/100x
DRAM	1MB	2GB	2000x
Disk	50MB	0.5-1TB	10K-20Kx
Network BW	10Mbits/sec	1GBits/sec	100x
Address bits	32	64	2x
Users/machine	10s	< 1	>10x
$/Performance	$60k	$600/3000	1/200,000x

Exponential Growth in Computing, Comm.

- Performance/Price doubles every 18 months
- 100x per decade
- Progress in next 18 months
  = ALL previous progress
  - New storage = sum of all old storage (ever)
  - New processing = sum of all old processing.
- This has led to some broad phases in computing, and correspondingly in the nature of operating systems

- Courtesy Jim Gray



History of Computers and OSes

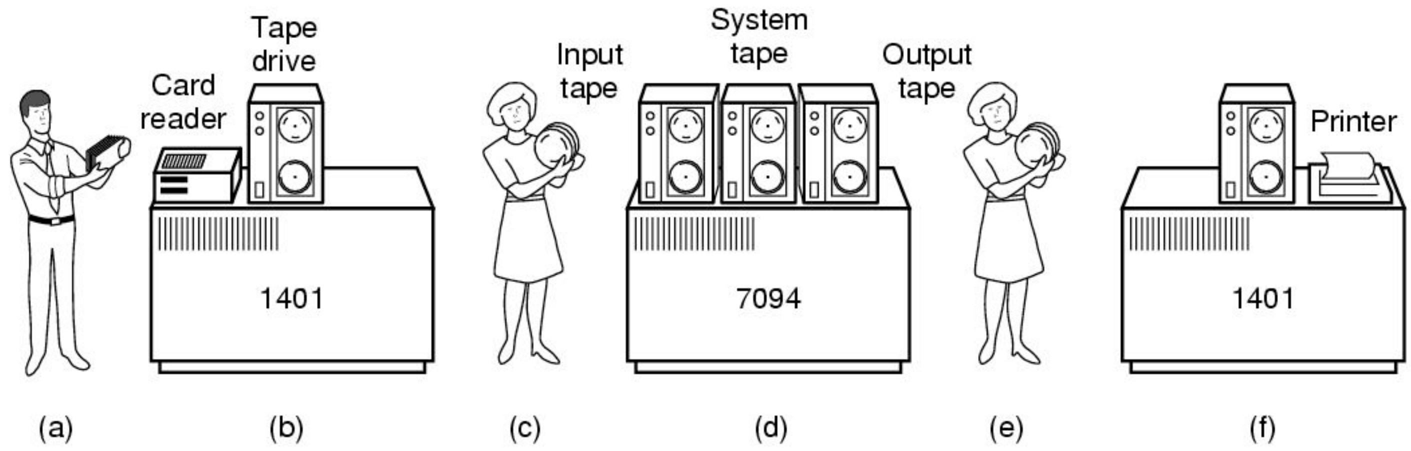
Generations:

- (1945–55) Vacuum Tubes
- (1955–65) Transistors and Batch Systems
- (1965–1980) ICs and Multiprogramming
- (1980–Present) Personal Computers

Phase 1: The Early Days

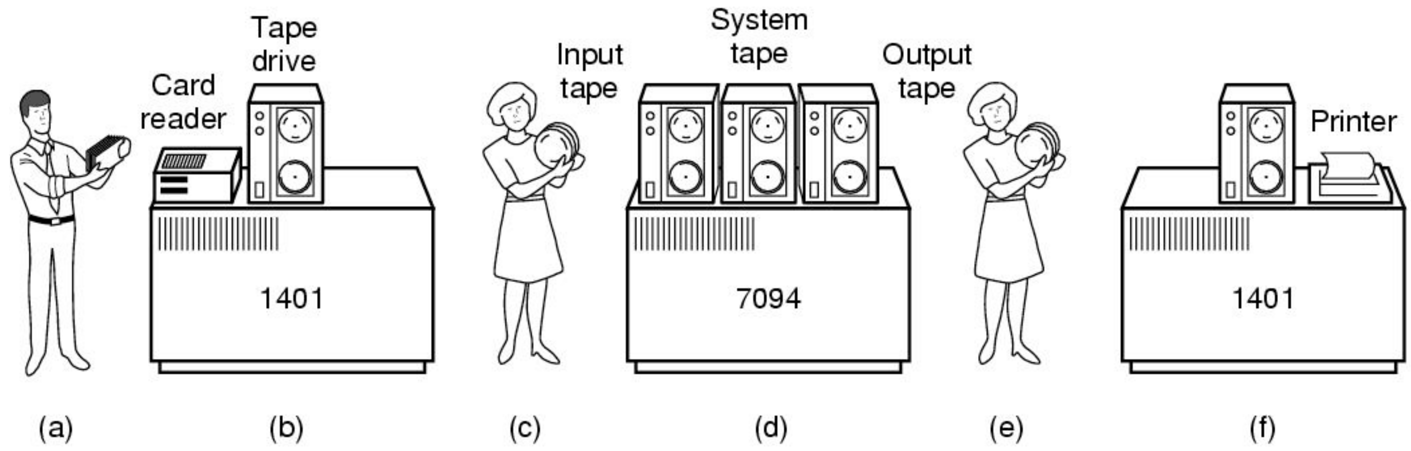
- Hardware very expensive, humans cheap
- When was the first functioning digital computer built?
- What was it built from?
- How was the machine programmed?
- What was the operating system?
- The big innovation: punch cards
- The really big one: the transistor
  - Made computers reliable enough to be sold to and operated by customers

Phase 2: Transistors and Batch Systems



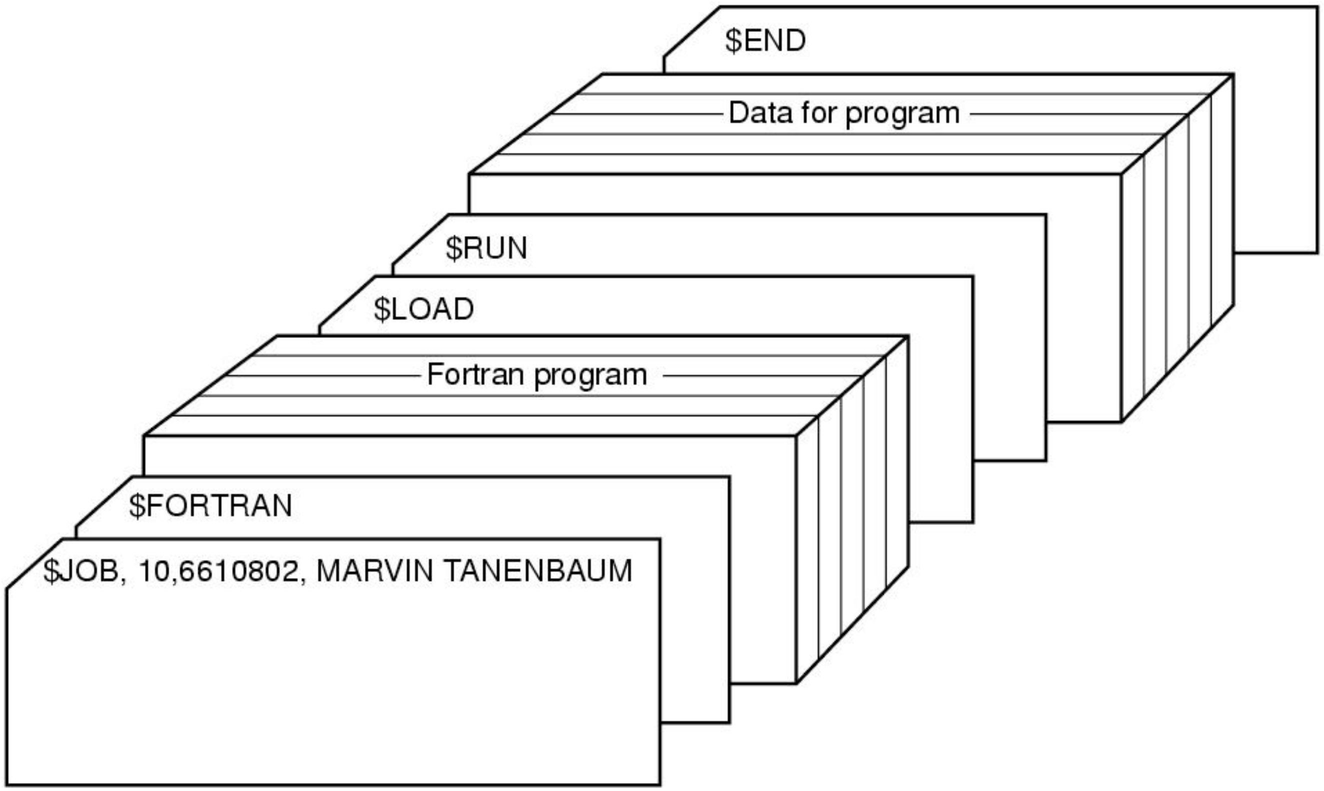
- Hardware still expensive, humans relatively cheap
- An early batch system
  - Programmers bring cards to reader system
  - Reader system puts jobs on tape

Phase 2: Transistors and Batch Systems



- An early batch system
  - Operator carries input tape to main computer
  - Main computer computes and puts output on tape
  - Operator carries output tape to printer system, which prints output

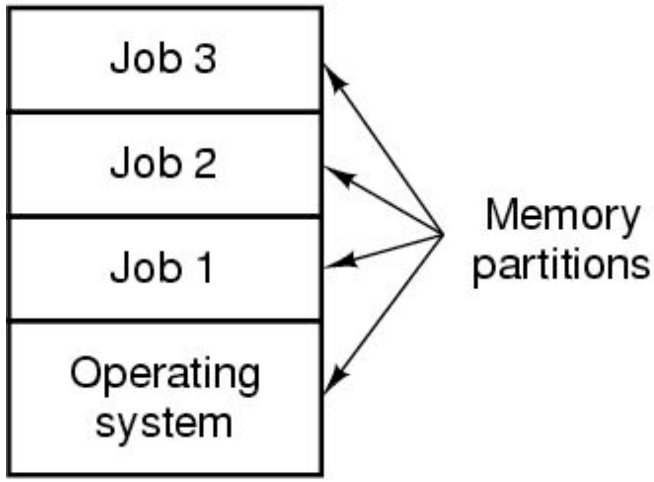
Punch cards and Computer Jobs



Phase 3: ICs and Multiprogramming

- Integrated circuits allowed families of computers to be built that were compatible
- Single OS to run on all (IBM OS/360): big and bloated
- Key innovation: multiprogramming
  - What happens when a job is waiting on I/O
  - What if jobs spend a lot of the time waiting on I/O?

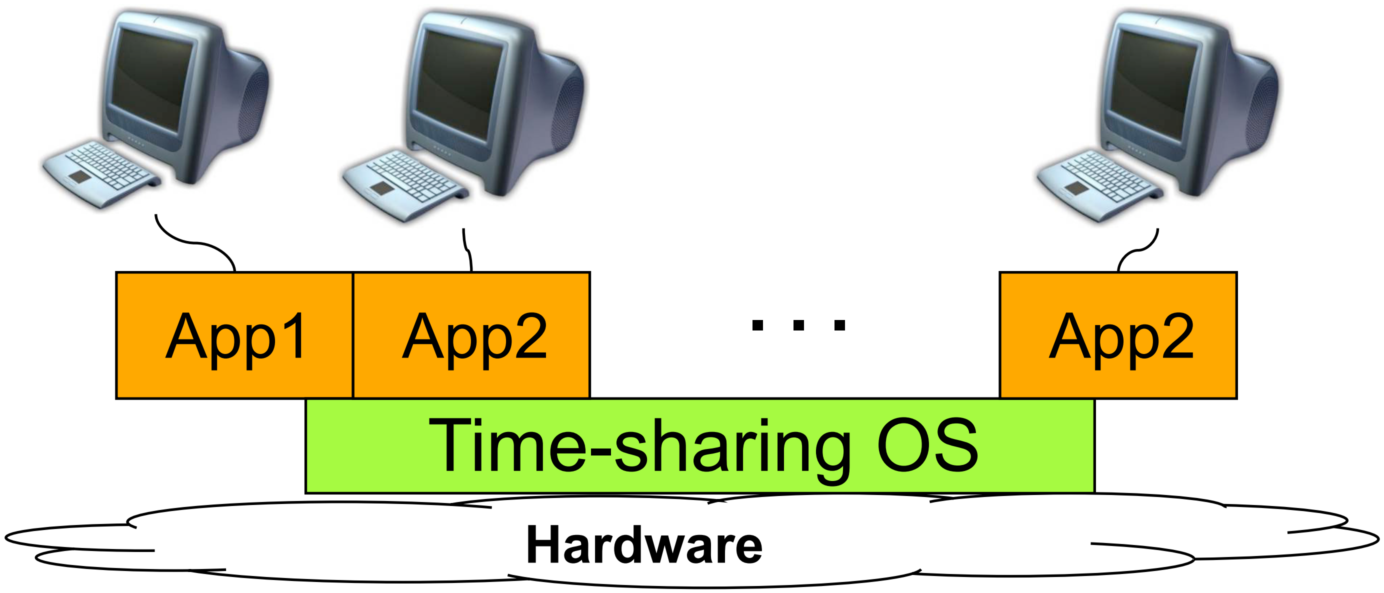
Phase 3: ICs and Multiprogramming



- Multiple jobs resident in computer's memory
- Hardware switches between them (interrupts)
- Hardware protects from one another (mem protection)
- Computer reads jobs from cards as jobs finish (spooling)
- Still batch systems: can't debug online
- Solution: time-sharing

Phase 3: ICs and Multiprogramming

- Time-sharing:
  - Users at terminals simultaneously
  - Computer switches among active ‘jobs’/sessions
  - Shorter, interactive commands serviced faster



Phase 3: ICs and Multiprogramming

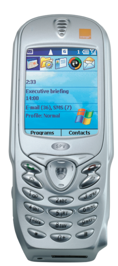
- The extreme: computer as a utility: MULTICS (late 60s)
  - Problem: thrashing as no. of users increases
  - Didn’t work then, but idea may be back
  - Let others administer and manage; I’ll just use
- ICs led to mini-computers: cheap, small, powerful
  - Stripped down version of MULTICS, led to UNIX
  - Two branches (Sys V, BSD), standardized as POSIX
  - Free follow-ups: Minix (education), Linux (production)

Phase 4: HW Cheaper, Human More Costly

- Personal computer
  - Altos OS, Ethernet, Bitmap display, laser printer
  - Pop-menu window interface, email, publishing SW, spreadsheet, FTP, Telnet
  - Eventually >100M units per year
- PC operating system
  - Memory protection
  - Multiprogramming
  - Networking

Now: > 1 Machines per User

- Pervasive computers
  - Wearable computers
  - Communication devices
  - Entertainment equipment
  - Computerized vehicle
- OS are specialized
  - Embedded OS
  - Specially configured general-purpose OS



Now: Multiple Processors per Machine

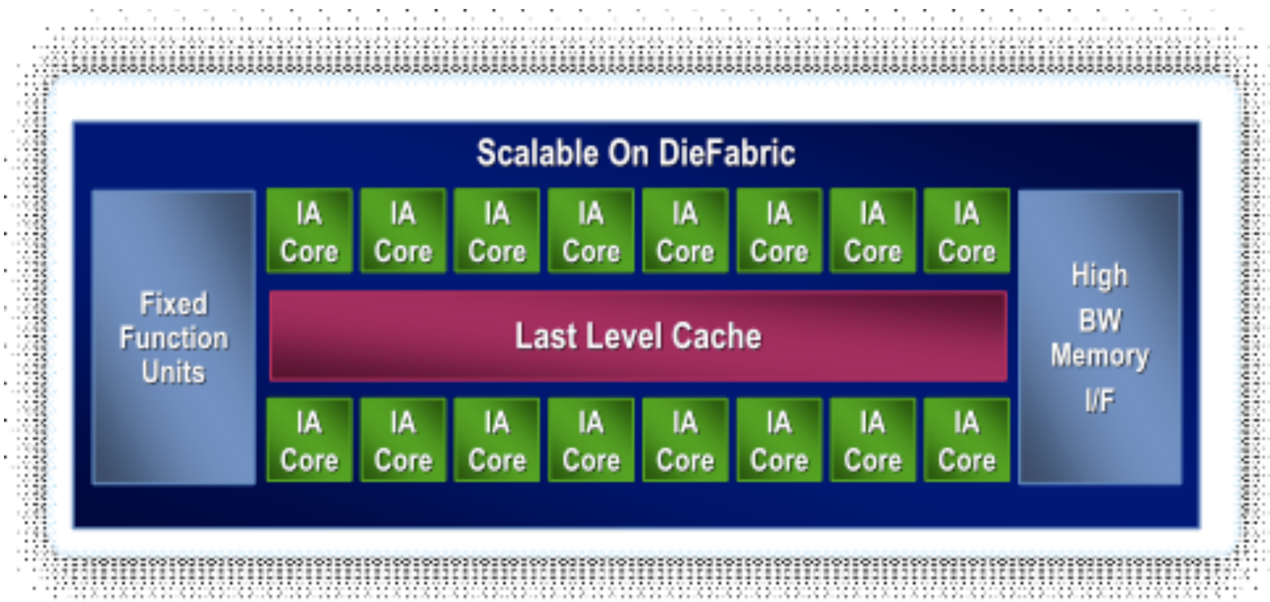
- Multiprocessors
  - SMP: Symmetric MultiProcessor
  - ccNUMA: Cache-Coherent Non-Uniform Memory Access
  - General-purpose, single-image OS with multiproccesor support

- Multicomputers
  - Supercomputer with many CPUs and high-speed communication
  - Specialized OS with special message-passing support

- Clusters
  - A network of PCs
  - Commodity OS

Now: Multiple "Cores" per Processor

- Multicore or Manycore transition
  - Intel and AMD have released 4-core and soon 6-core CPUs
  - SUN's Niagara processor has 8-cores
  - Azul Vega8 now packs 24 cores onto the same chip
  - Intel has a TFlop-chip with 80 cores
  - Ambric Am2045: 336-core Array (embedded, and accelerators)
- Accelerated need for software support
  - OS support for many cores; parallel programming of applications



Summary: Evolution of Computers

60’s-70’s - Mainframes

- Rise of IBM

70’s - 80’s – Minicomputers

- Rise of Digital Equipment Corporation

80’s - 90’s – PCs

- Rise of Intel, Microsoft

Now – Post-PC

- Distributed applications

Summary: Evolution and Implications for OS

	Mainframe	Mini	Micro
System $ / Worker $	10:1 – 100:1	10:1 – 1:1	1:10-1:100
Goal	System utilization	Overall cost	Productivity
Target	Capacity	Features	Ease of Use

Today

- Evolution of computing and operating systems
- Why study operating systems?
- What’s in COS 318?
- Overview of Operating Systems

Why Study OS?

- OS is a key part of a computer system
  - It makes our life better (or worse)
  - It is “magic” to realize what we want
  - It gives us “power”
- Learn about concurrency
  - Parallel programs run on OS
  - OS runs on parallel hardware: all hw becoming parallel
  - OS is great way to learn concurrent programming
- Understand how a system works
  - How many procedures does a key stroke invoke?
  - What happens when program references 0 as a pointer?
  - Real OS is huge and impossible to read everything, but building a small OS will go a long way

Why Study OS?

- Important for studying further areas
  - Networking, distributed systems, …
- Full employment
  - New hardware capabilities and organizations
  - New features
  - New approaches
  - E.g. handheld computers, Java, WWW
  - Engineering tradeoffs, keep changing as the hardware changes from below and the needs of apps from above
- Lots of jargon: sound smart (or super-nerdy)

Today

- Evolution of computing and operating systems
- Why study operating systems?
- What's in COS 318?
- Overview of Operating Systems

What Is in COS 318?

- Methodology
  - Lectures with discussions
  - Readings with topics
  - A lot of design and rationale, some theory, a fair bit of practice
  - Six projects to build key aspects of a basic OS
- Covered concepts
  - Operating system structure
    - Processes, threads, system calls and virtual machine monitor
  - Synchronization
    - Mutex, semaphores and monitors
  - I/O subsystems
    - Device drivers, IPC, and introduction to networking
  - Virtual memory
    - Address spaces and paging
  - Storage system
    - Disks and file system

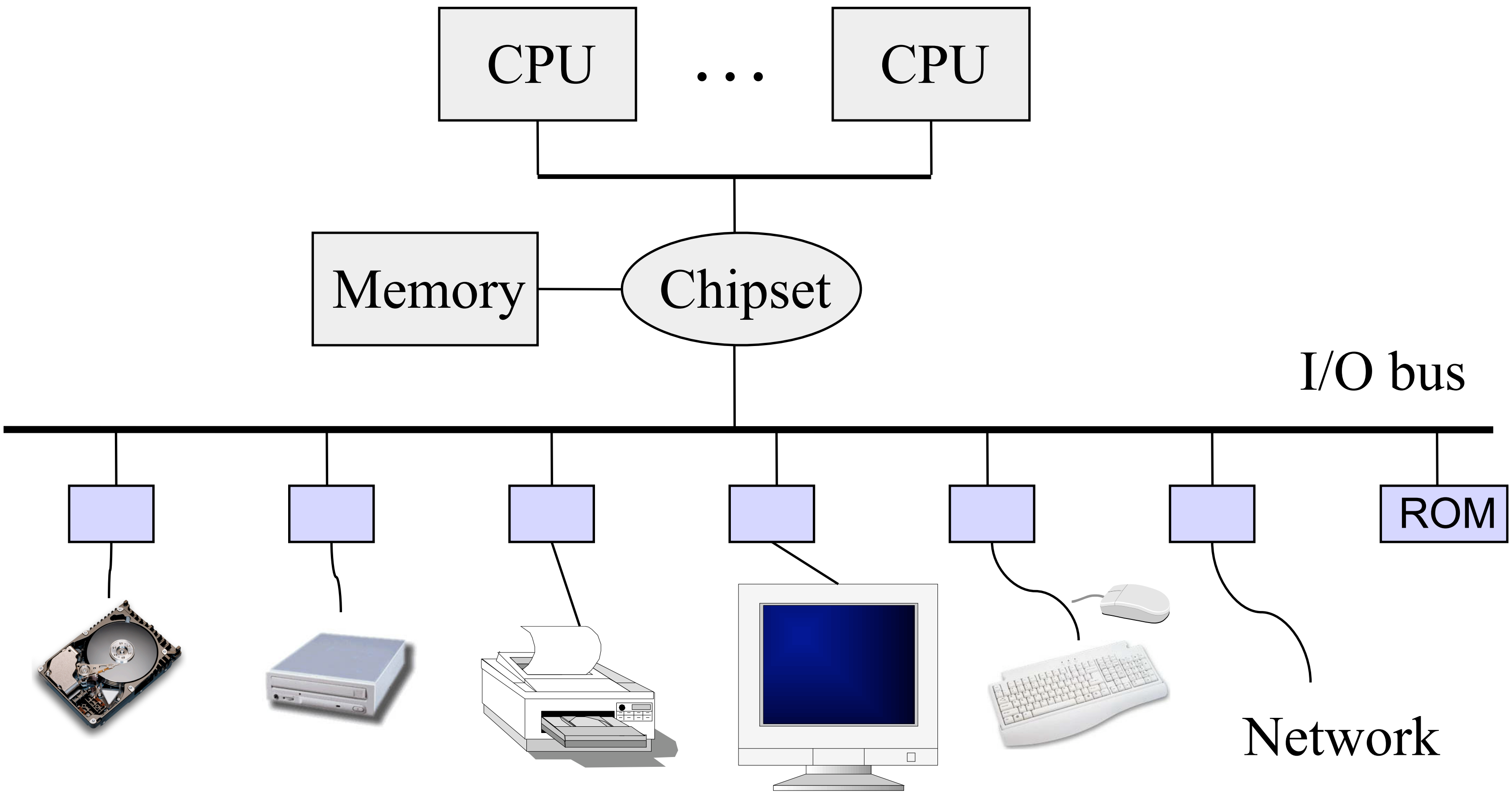
What is COS 318 Like?

- Is tt theoretical or practical?
  - Focus on concepts, but also getting hands dirty in projects
  - More about engineering tradeoffs, constraints, optimization and imperfection than about optimal results and beautiful mathematics
  - High rate of change in the field yet lots of inertia in OSes
- Is it easy?
  - No. Fast paced, hard material, a lot of programming
- What will enable me to succeed?
  - Solid C background, pre-reqs, tradeoff thinking
  - NOT schedule overload

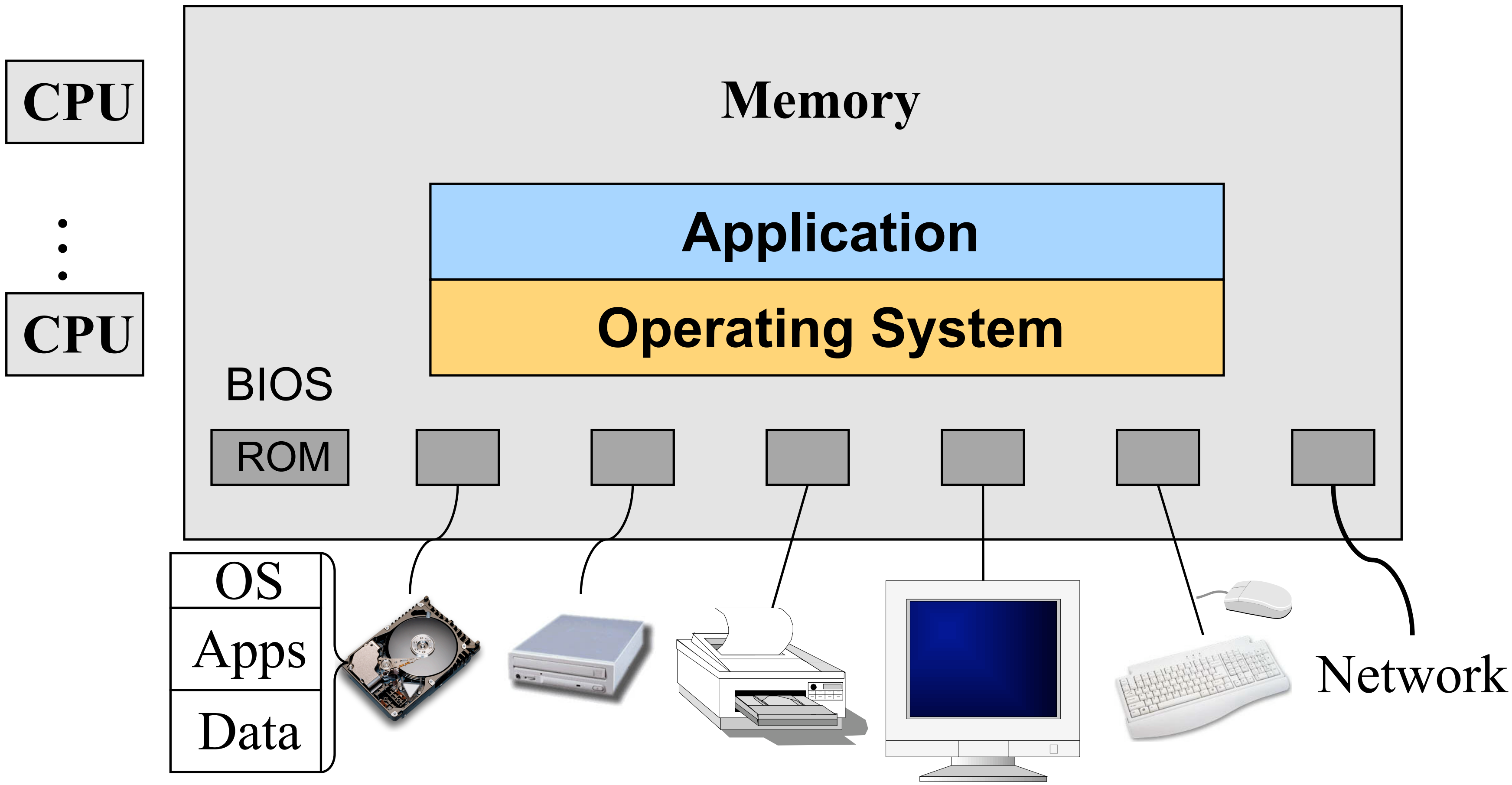
Today

- Evolution of computing and operating systems
- Why study operating systems?
- What's in COS 318?
- Overview of Operating Systems

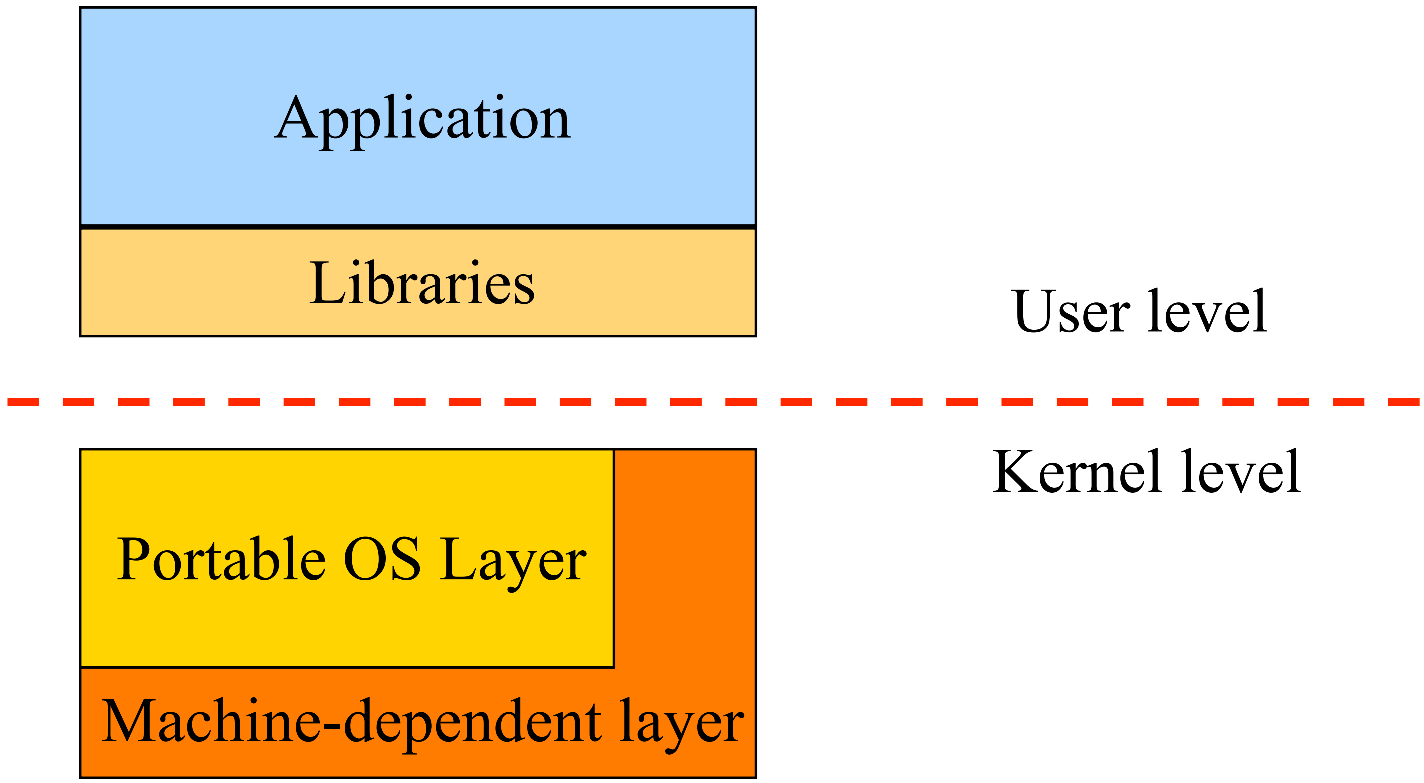
Hardware of A Typical Computer



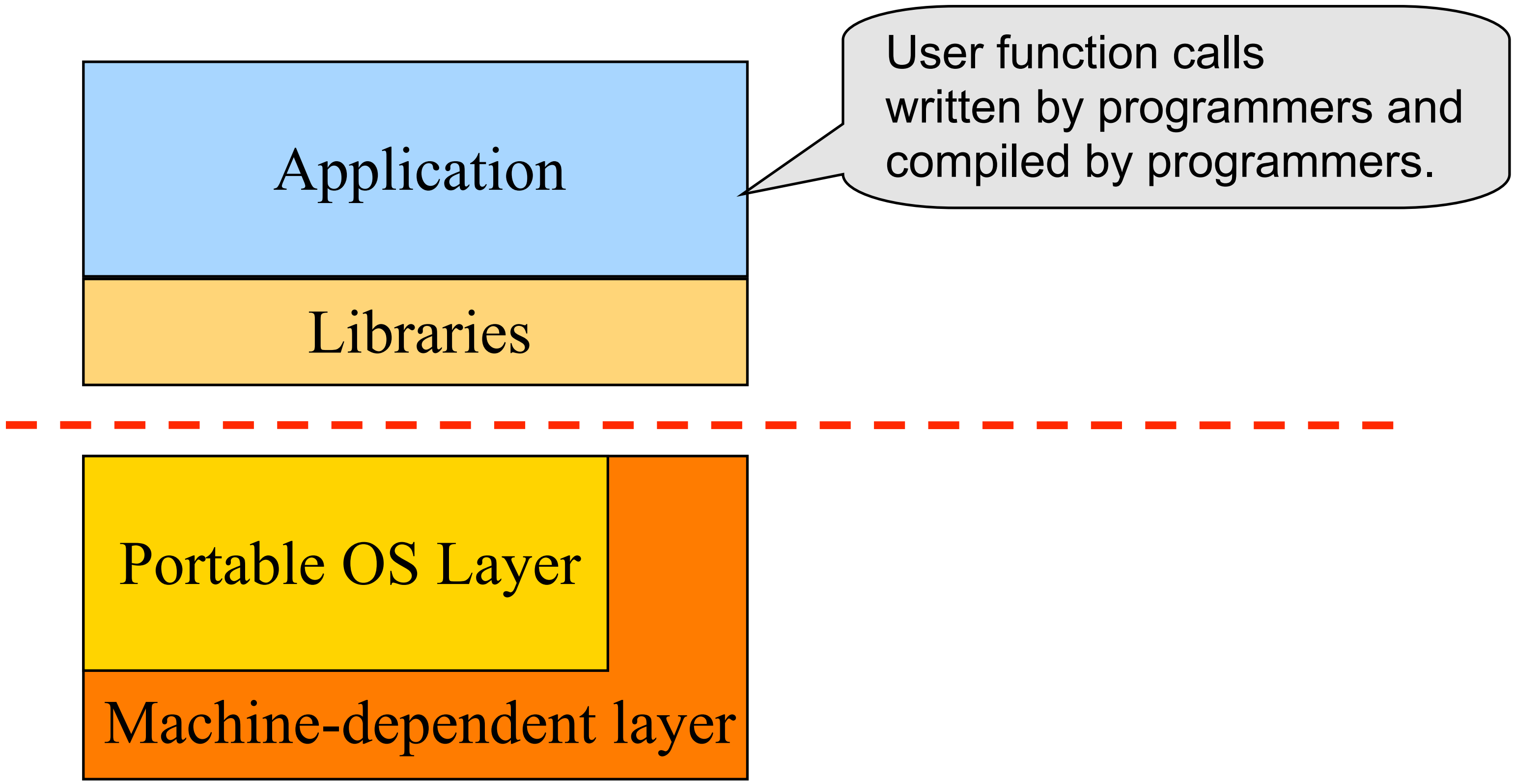
A Typical Computer System



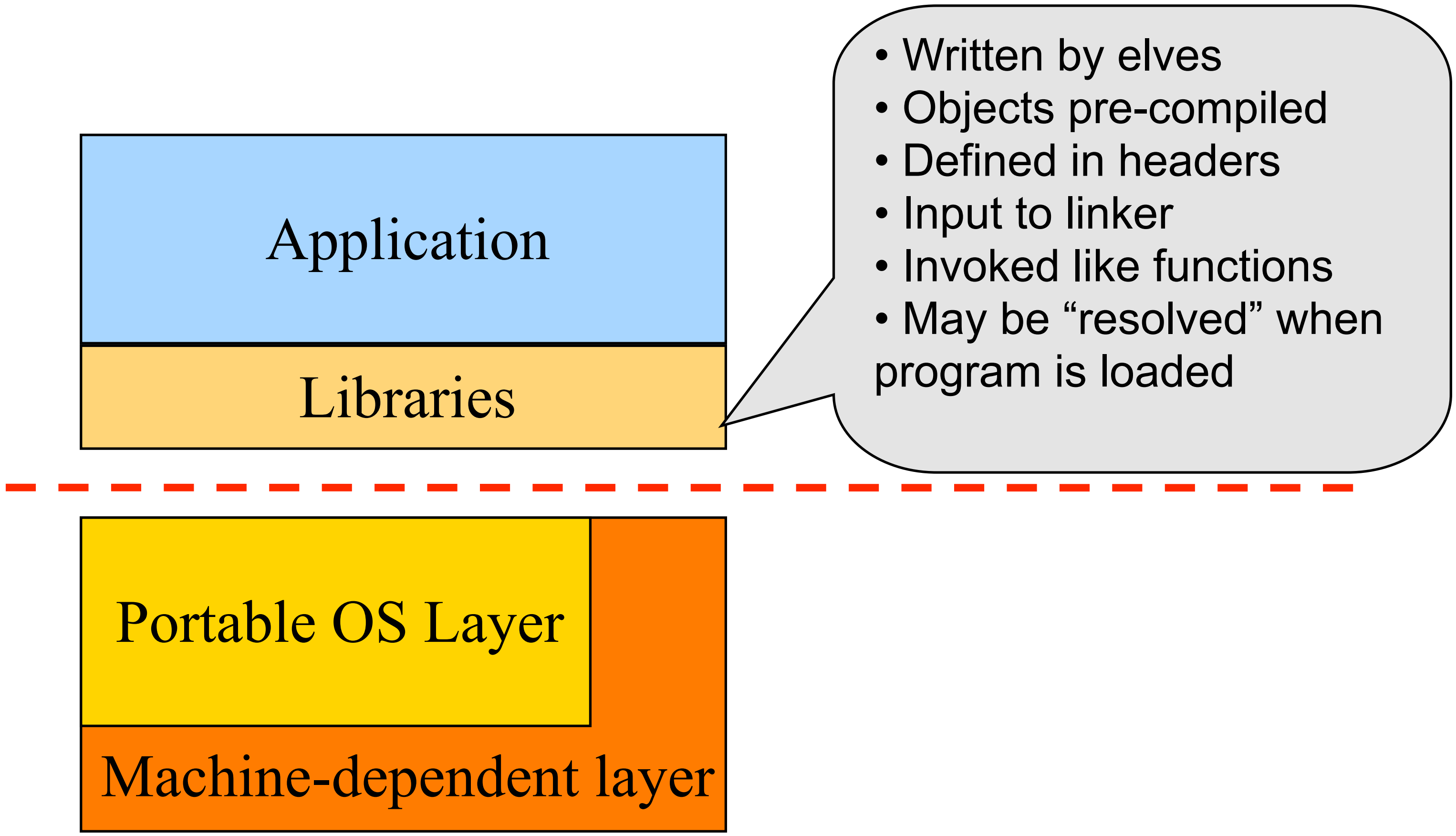
Typical Unix OS Structure



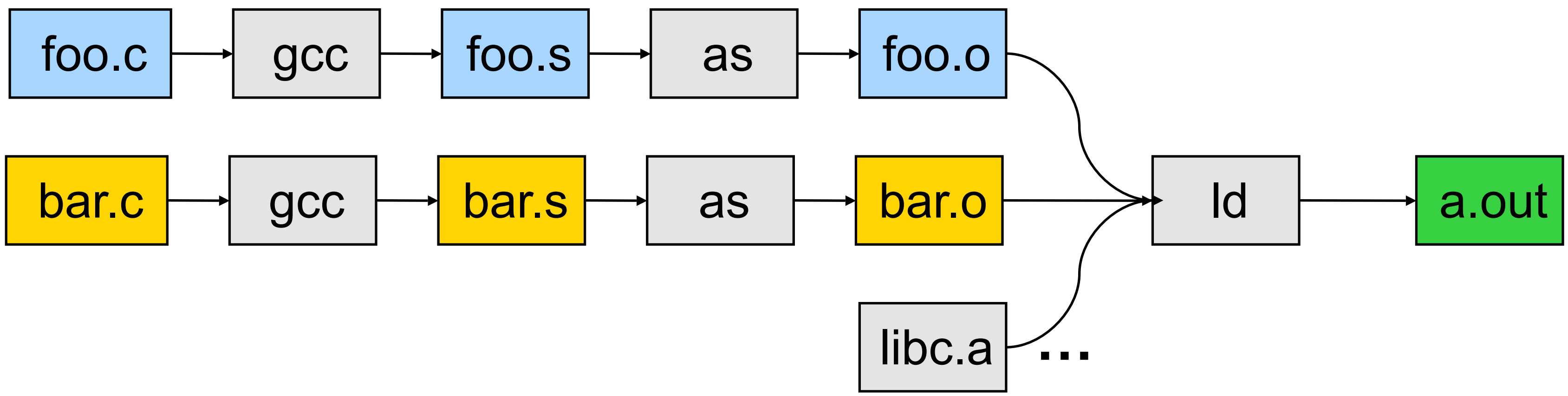
Typical Unix OS Structure



Typical Unix OS Structure



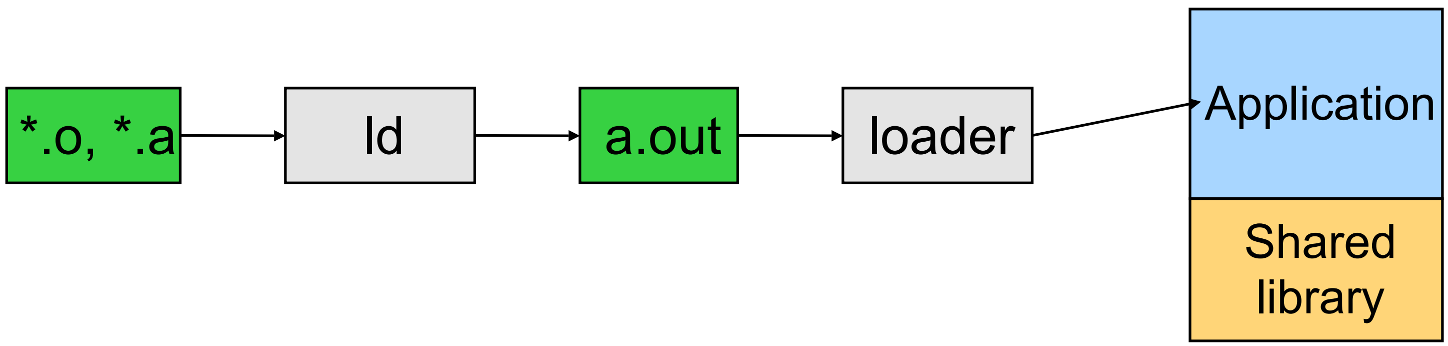
Pipeline of Creating An Executable File



- gcc can compile, assemble, and link together
- Compiler (part of gcc) compiles a program into assembly
- Assembler compiles assembly code into relocatable object file
- Linker links object files into an executable
- For more information:
  - Read man page of a.out, elf, ld, and nm
  - Read the document of ELF

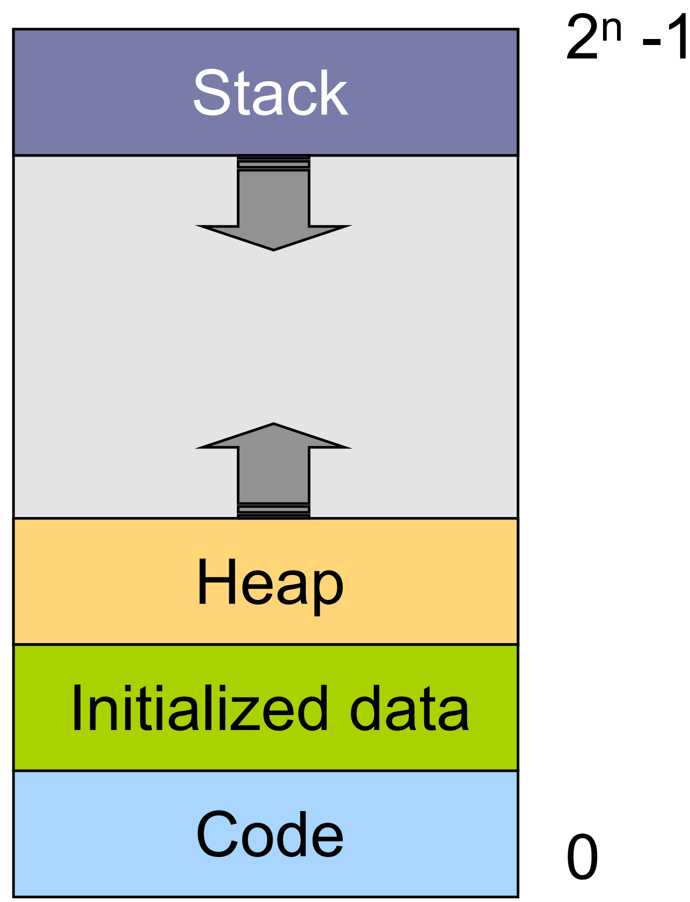
Execution (Run An Application)

- On Unix, “loader” does the job
  - Read an executable file
  - Layout the code, data, heap and stack
  - Dynamically link to shared libraries
  - Prepare for the OS kernel to run the application



What's An Application?

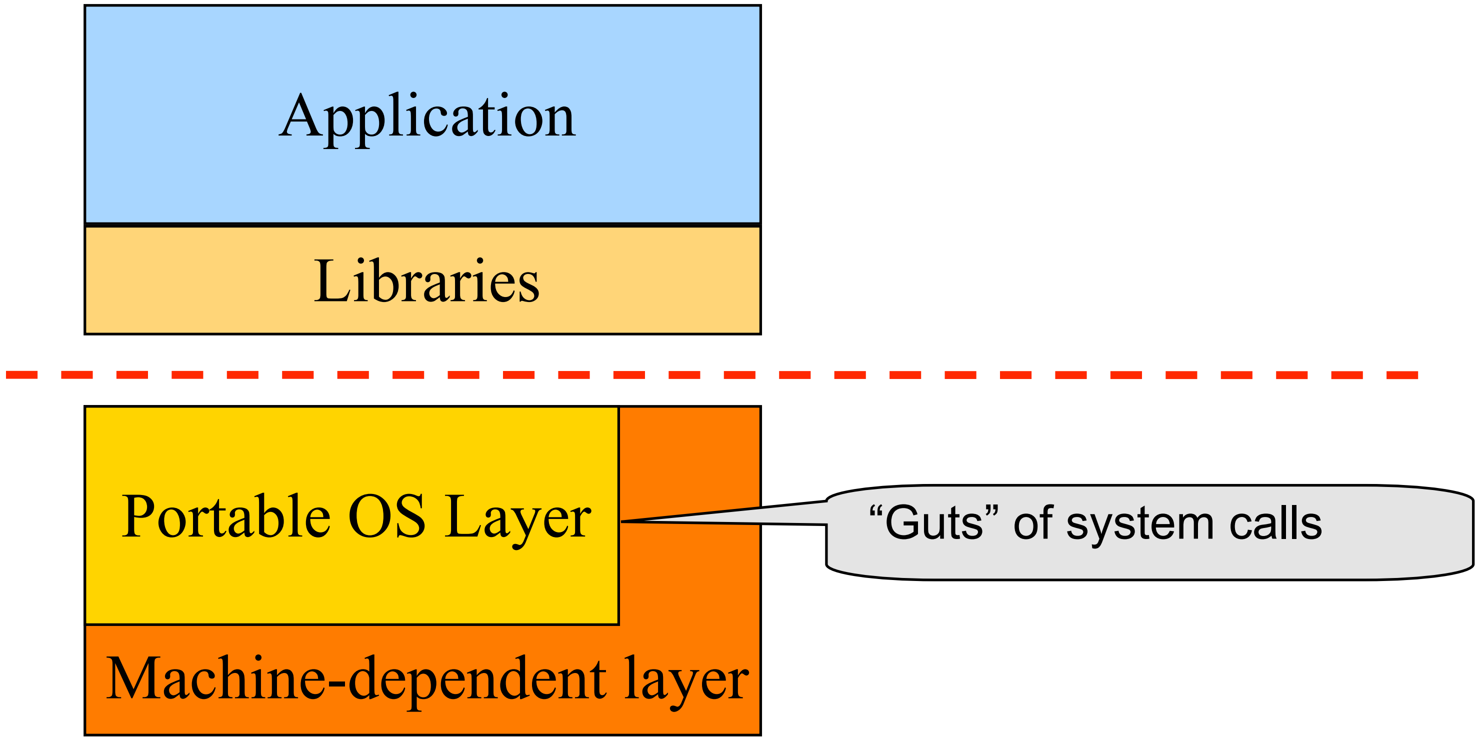
- Four segments
  - Code/Text – instructions
  - Data – initialized global variables
  - Stack
  - Heap
- Why?
  - Separate code and data
  - Stack and heap go towards each other



Responsibilities

- Stack
  - Layout by compiler
  - Allocate/deallocate by process creation (fork) and termination
  - Names are relative off of stack pointer and entirely local
- Heap
  - Linker and loader say the starting address
  - Allocate/deallocate by library calls such as malloc() and free()
  - Application program use the library calls to manage
- Global data/code
  - Compiler allocate statically
  - Compiler emit names and symbolic references
  - Linker translate references and relocate addresses
  - Loader finally lay them out in memory

Typical Unix OS Structure

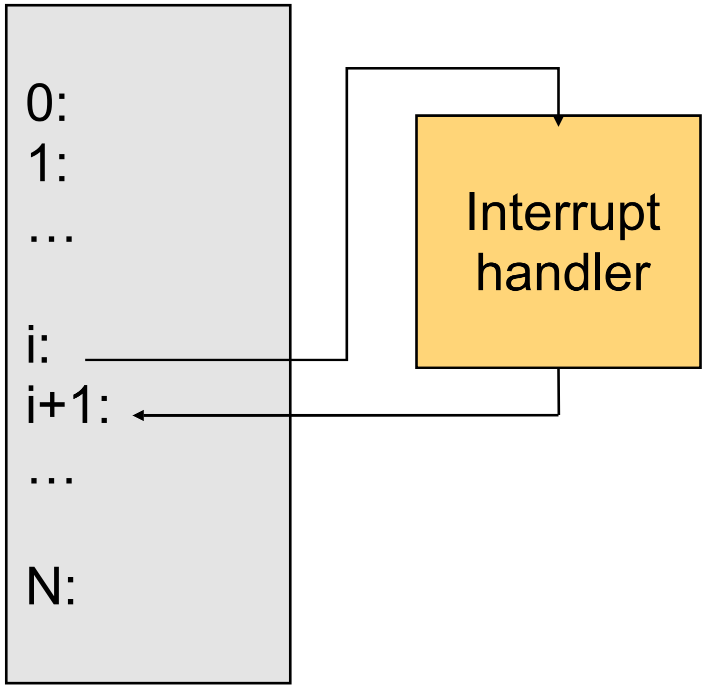


OS Service Examples

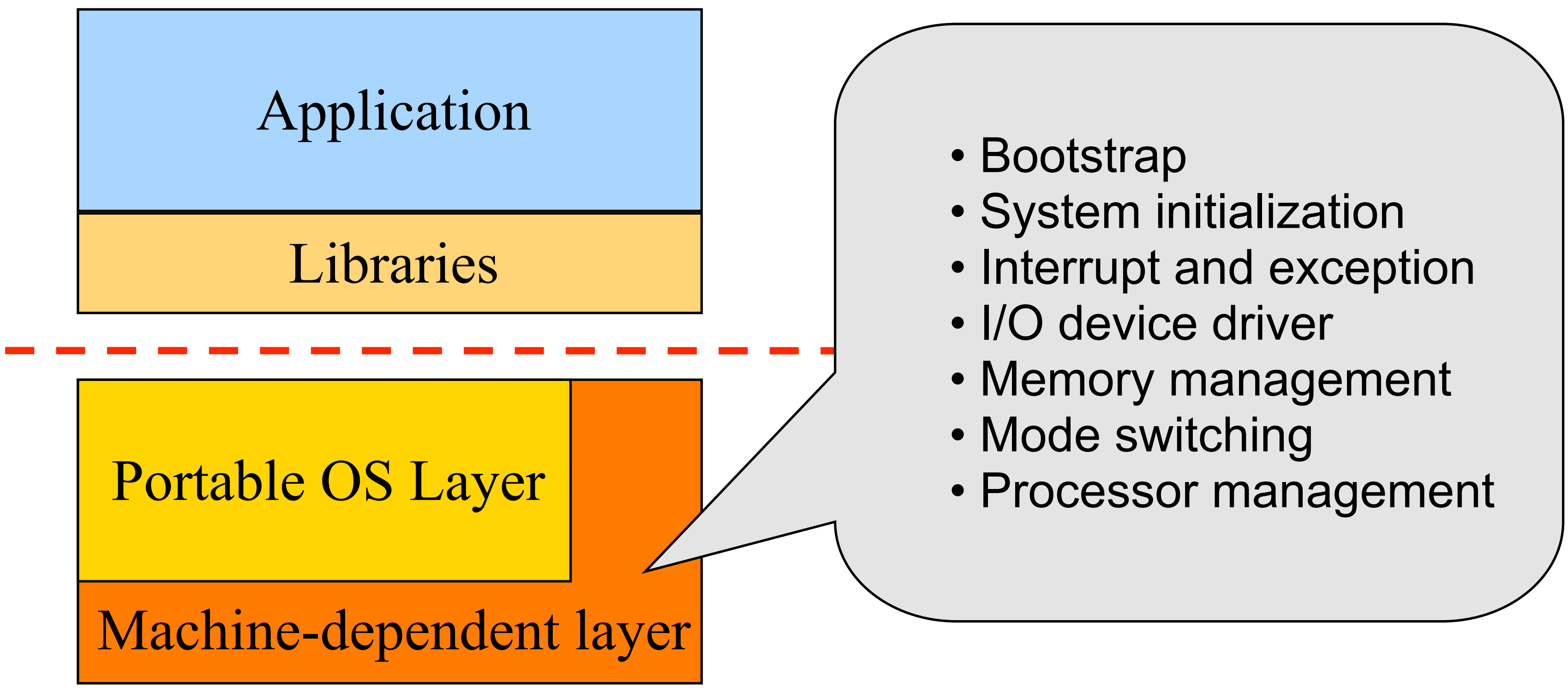
- Examples that are not provided at user level
  - System calls: file open, close, read and write
  - Control the CPU so that users won't stuck by running
    - while (1) ;
  - Protection:
    - Keep user programs from crashing OS
    - Keep user programs from crashing each other
- System calls are typically traps or exceptions
  - System calls are implemented in the kernel
  - Application "traps" to kernel to invoke a system call
  - When finishing the service, a system returns to the user code

Interrupts

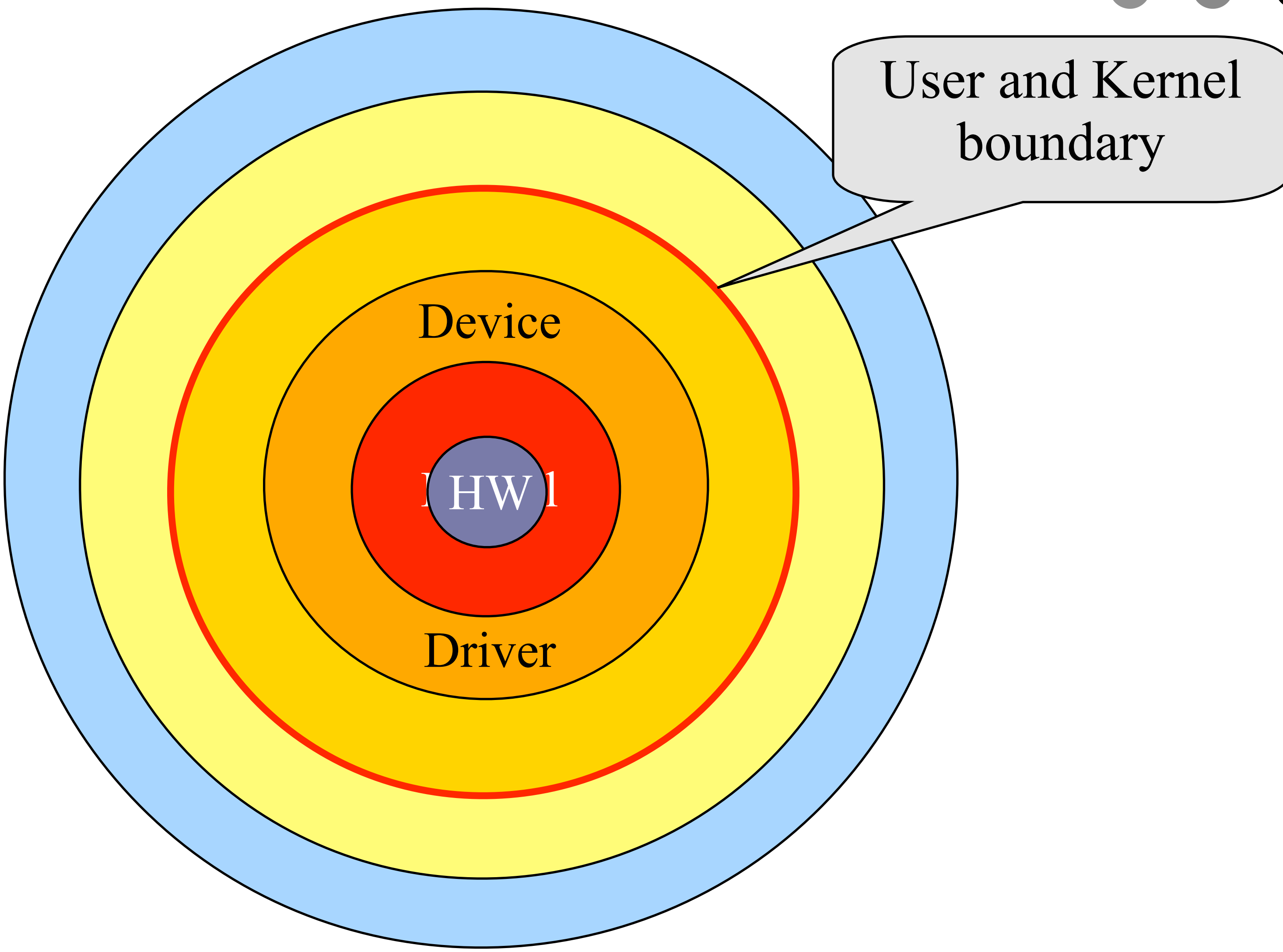
- Raised by external events
- Interrupt handler is in the kernel
  - Switch to another process
  - Overlap I/O with CPU
  - …
- Eventually resume the interrupted process
- A way for CPU to wait for long-latency events (like I/O) to happen



Typical Unix OS Structure

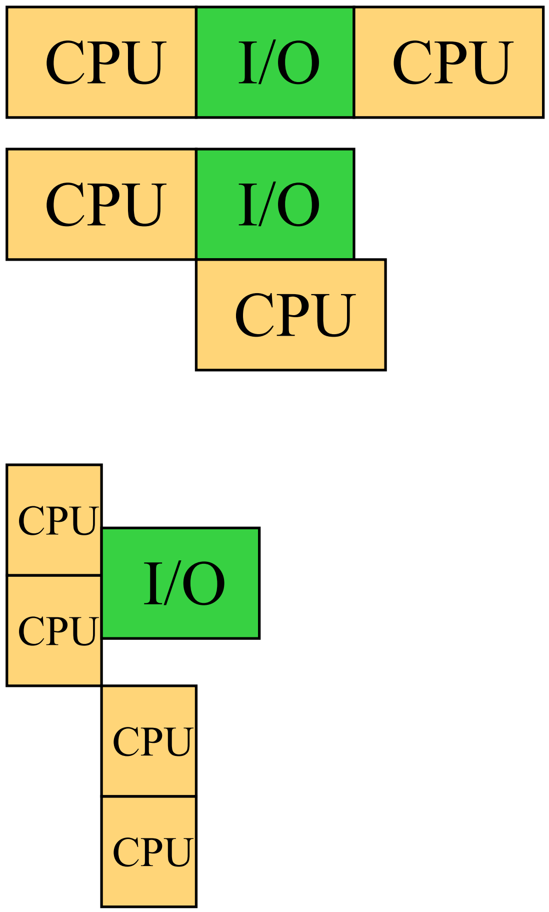


Software “Onion” Layers



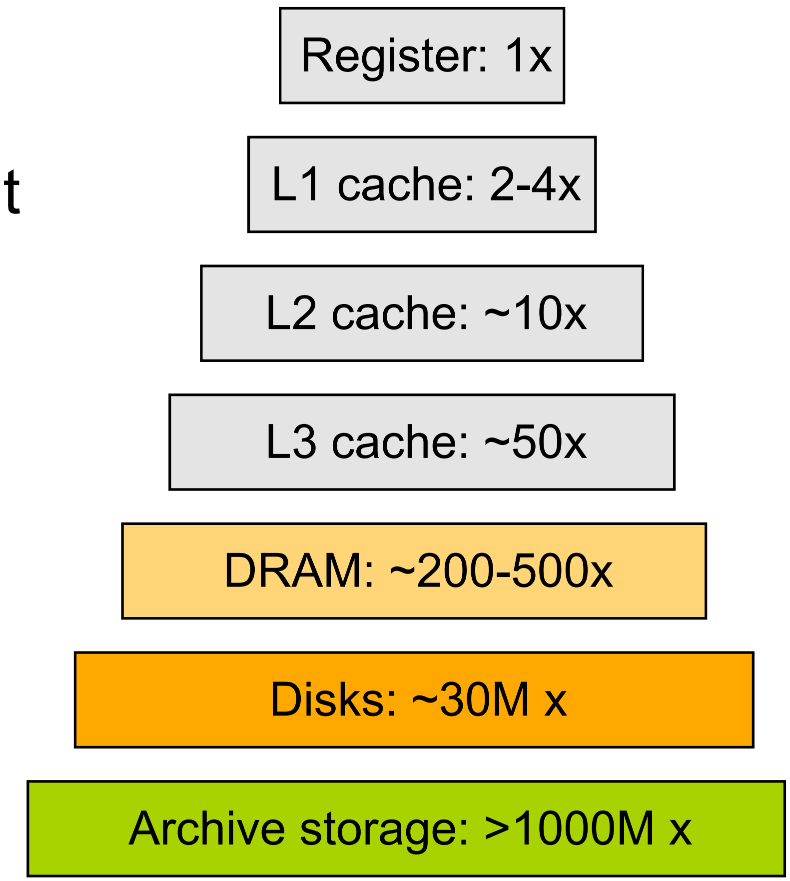
Processor Management

- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free



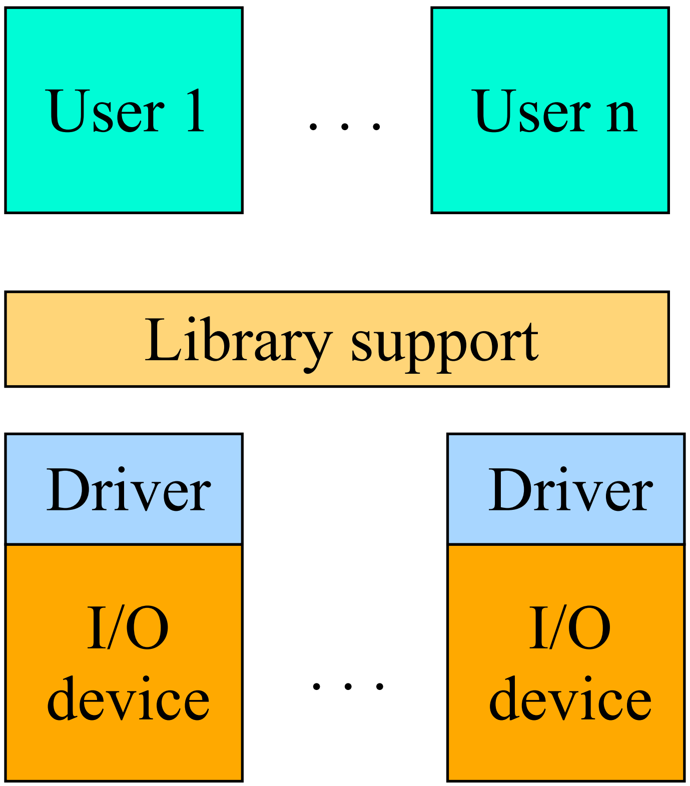
Memory Management

- Goals
  - Support programs to run
  - Allocation and management
  - Transfers from and to secondary storage
- Issues
  - Efficiency & convenience
  - Fairness
  - Protection



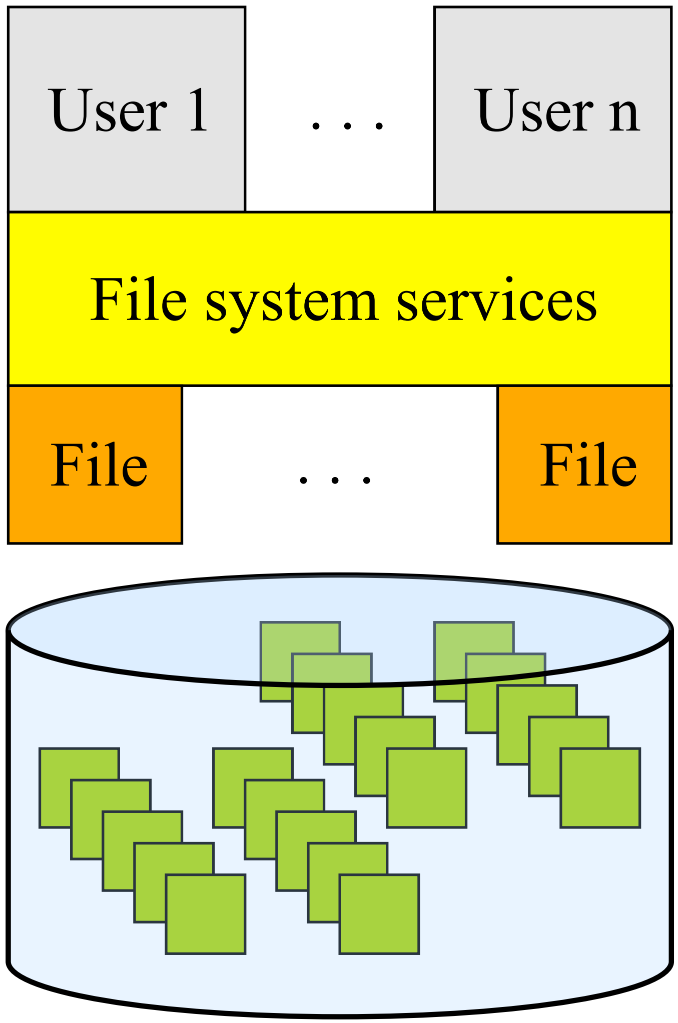
I/O Device Management

- Goals
  - Interactions between devices and applications
  - Ability to plug in new devices
- Issues
  - Efficiency
  - Fairness
  - Protection and sharing



File System

- Goals:
  - Manage disk blocks
  - Map between files and disk blocks
- A typical file system
  - Open a file with authentication
  - Read/write data in files
  - Close a file
- Issues
  - Reliability
  - Safety
  - Efficiency
  - Manageability

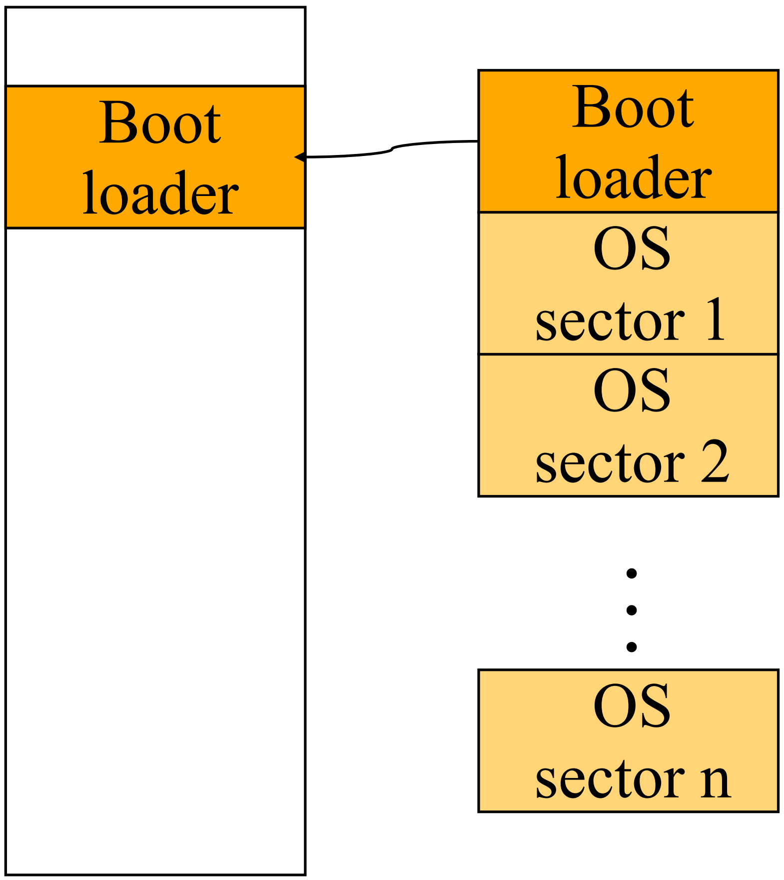


Window Systems

- Goals
  - Interacting with a user
  - Interfaces to examine and manage apps and the system
- Issues
  - Direct inputs from keyboard and mouse
  - Display output from applications and systems
  - Labor of division
    - All in the kernel (Windows)
    - All at user level
    - Split between user and kernel (Unix)

Bootstrap

- Power up a computer
- Processor reset
  - Set to known state
  - Jump to ROM code (BIOS is in ROM)
- Load in the boot loader from stable storage
- Jump to the boot loader
- Load the rest of the operating system
- Initialize and run
- Question: Can BIOS be on disk?



System Boot

- Power on (processor waits until Power Good Signal)
- Processor jumps on a PC to address FFFF0h
  - 1M= 1,048,576= 2^{20} =FFFFFh+1
  - FFFFFh=FFFF0h+16 is the end of the (first 1MB of) system memory
  - The original PC using Intel 8088 had 20 address lines :-)
- (FFFFFFF0h) is a JMP instruction to the ROM BIOS startup program

Maps to FFFFFFF0h= 2^{32}-16

ROM Bios Startup Program (1)

- POST (Power-On Self-Test)
  - If pass then AX:=0; DH:=5 (586: Pentium);
  - Stop booting if fatal errors, and report
- Look for video card and execute built-in ROM BIOS code (normally at C000h)
- Look for other devices ROM BIOS code
  - IDE/ATA disk ROM BIOS at C8000h (=819,200d)
- Display startup screen
  - BIOS information
- Execute more tests
  - memory
  - system inventory

SCSI disks: must often provide their own BIOS

ROM BIOS startup program (2)

- Look for logical devices
  - Label them
    - Serial ports
      - COM 1, 2, 3, 4
    - Parallel ports
      - LPT 1, 2, 3
  - Assign each an I/O address and IRQ
- Detect and configure Plug-and-Play (PnP) devices
- Display configuration information on screen

ROM BIOS startup program (3)

- Search for a drive to BOOT from
  - Floppy or Hard disk
    - Boot at cylinder 0, head 0, sector 1
- Load code in boot sector
- Execute boot loader
- Boot loader loads program to be booted
    - If no OS: "Non-system disk or disk error - Replace and press any key when ready"
- Transfer control to loaded program
- Is it okay to boot at first sector on the floppy or disk?

Ways to Develop An Operating System

- A hardware simulator
- A virtual machine
- A good kernel debugger
  - When OS crashes, always goes to the debugger
  - Debugging over the network