COS 318: Operating Systems

Overview

Jaswinder Pal Singh
Computer Science Department
Princeton University

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

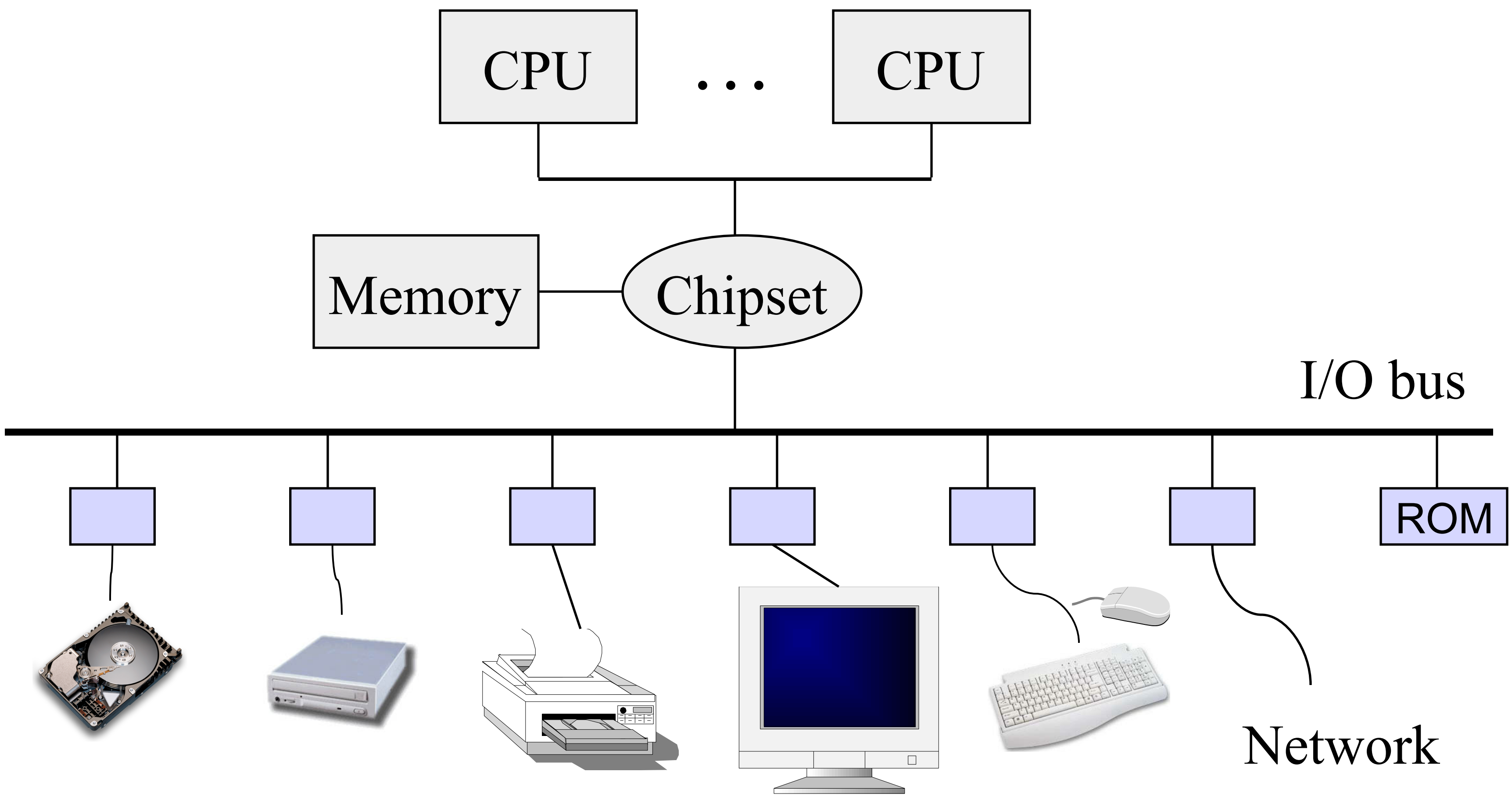
Important Times

- Precepts:
  - Mon: 7:30-8:20pm, 105 CS building
  - This week (9/19: TODAY):
    - Tutorial of Assembly programming and kernel debugging
- Project 1
  - Design review:
    - 9/26: 1:30pm – 6:30pm (**Signup online**), 010 Friend Center
  - Project 1 due: 10/02 at 11:55pm
- To do:
  - Make sure you have your project partner

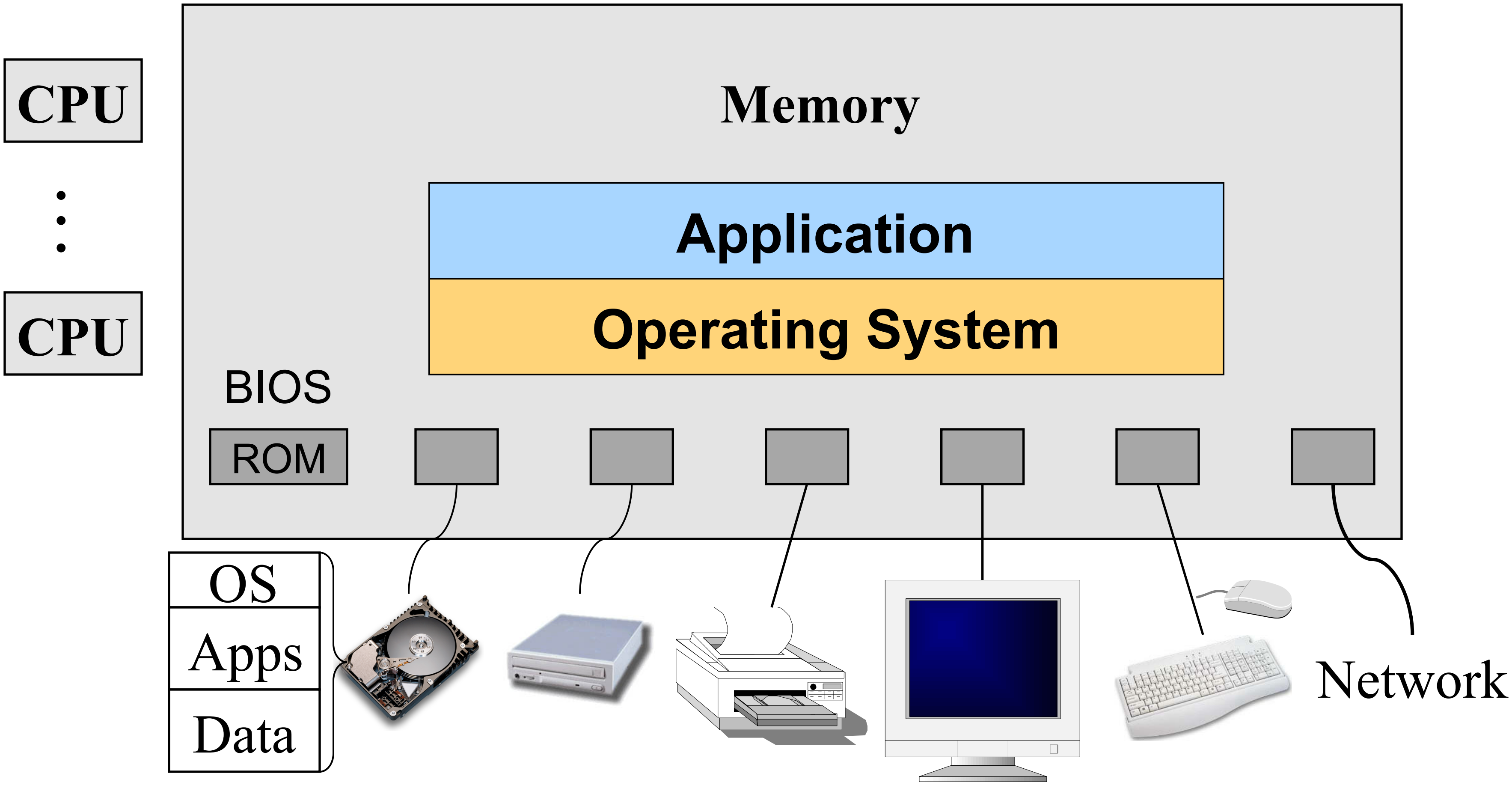
Today

- Overview of OS functionality
- Overview of OS components

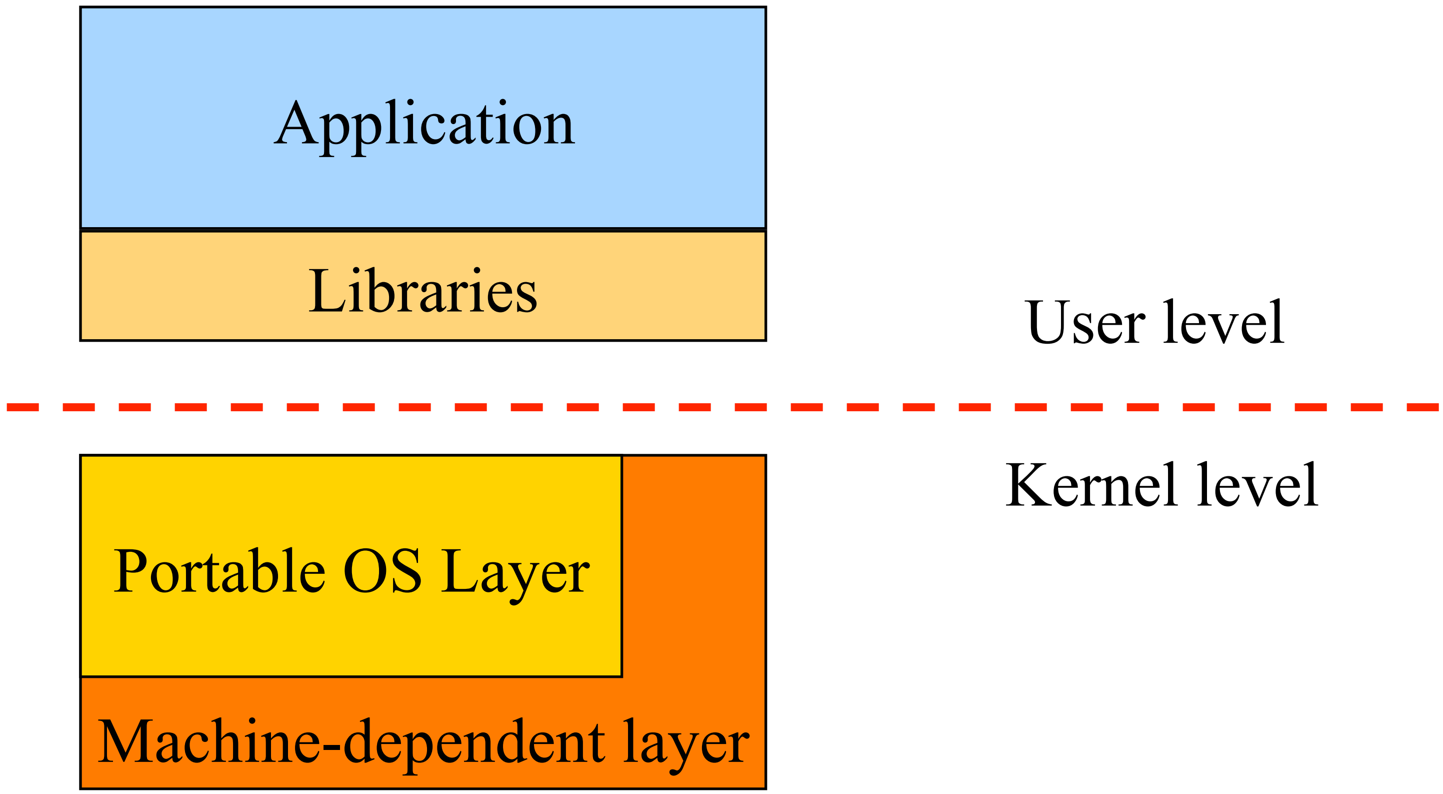
Hardware of A Typical Computer



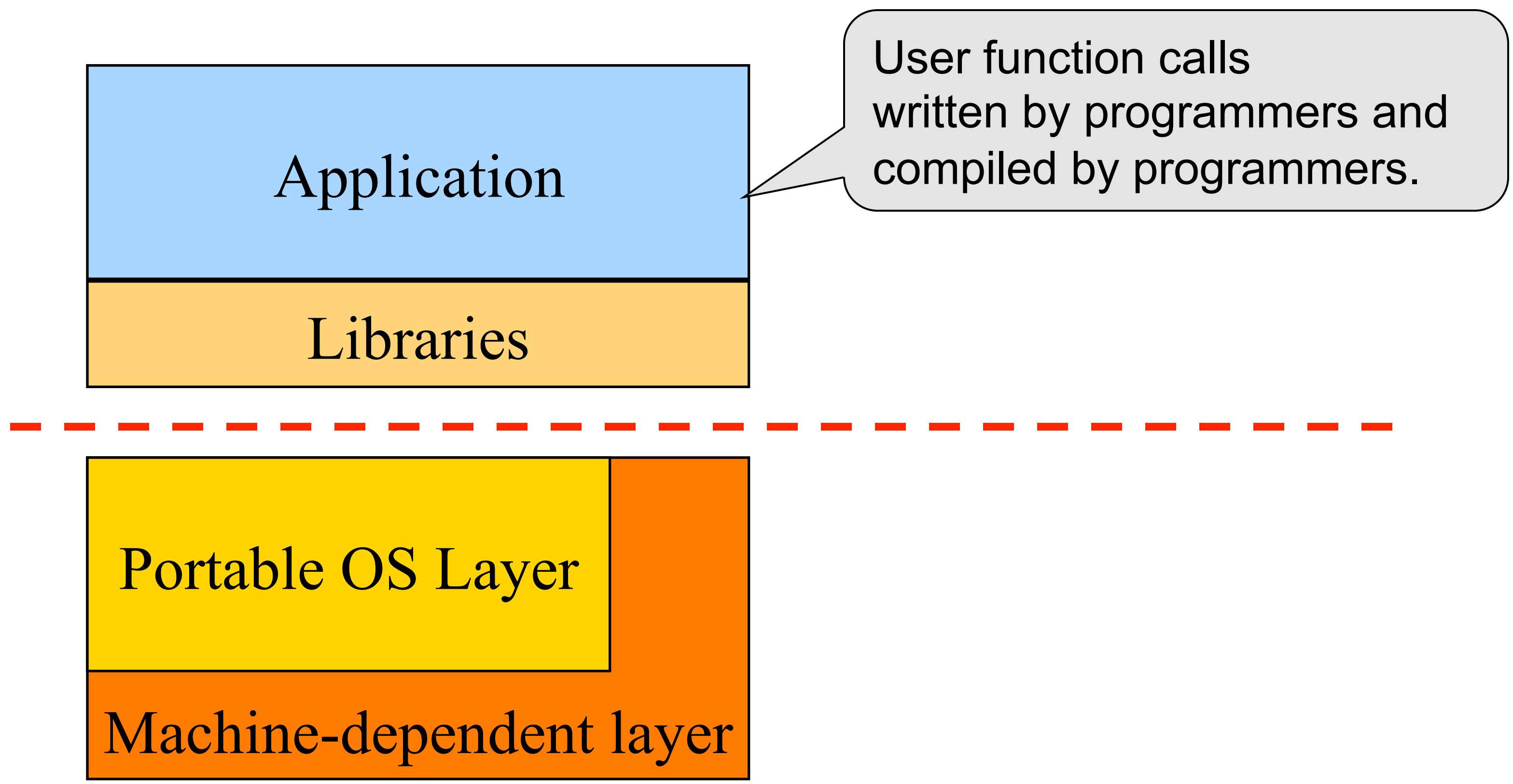
A Typical Computer System



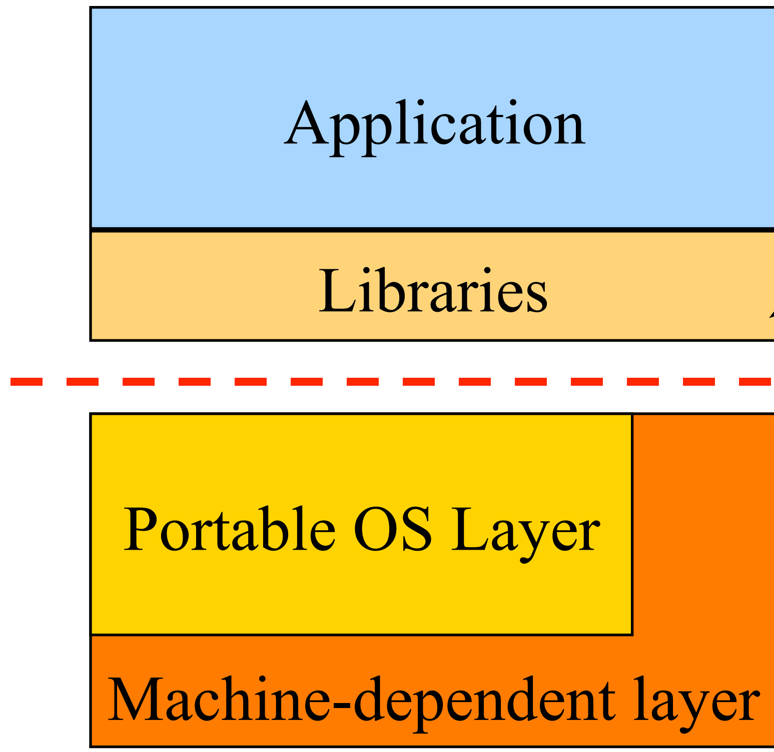
Typical Unix OS Structure



Typical Unix OS Structure

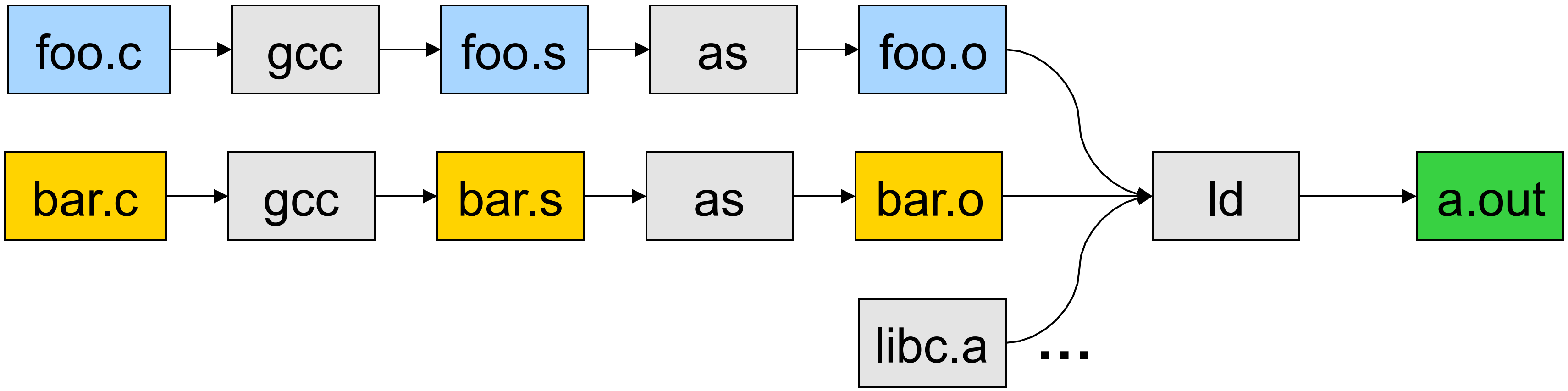


Typical Unix OS Structure



- Written by elves
- Objects pre-compiled
- Defined in headers
- Input to linker
- Invoked like functions
- May be “resolved” when program is loaded

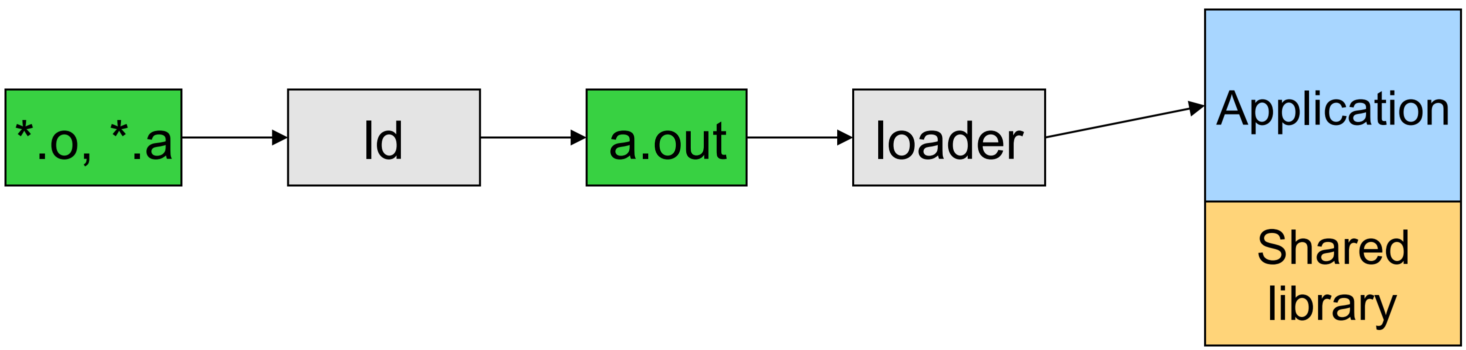
Application: How it's created



- 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

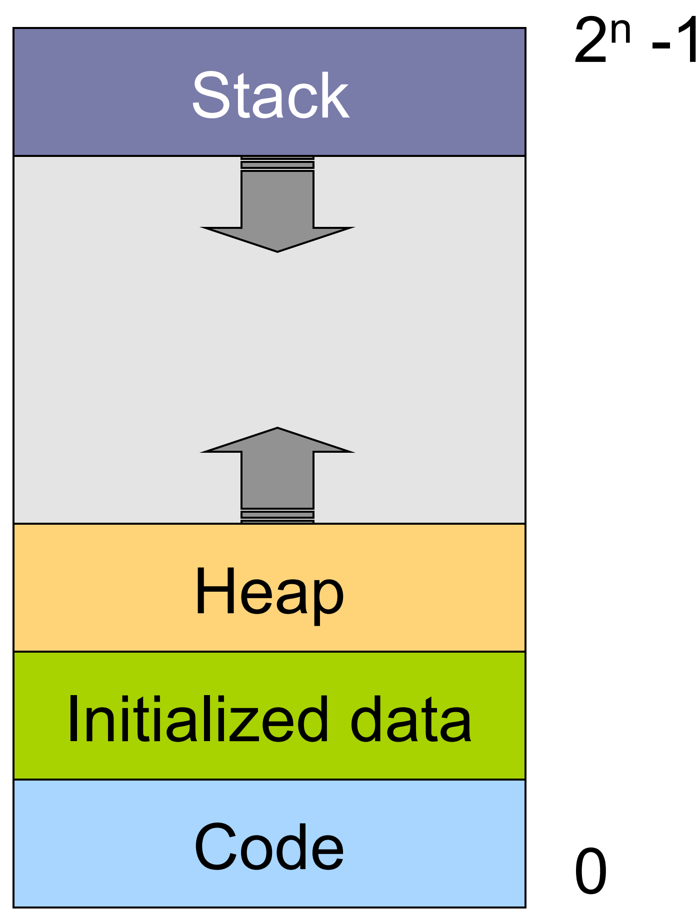
Application: How it's executed

- 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 an executable application looks like

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



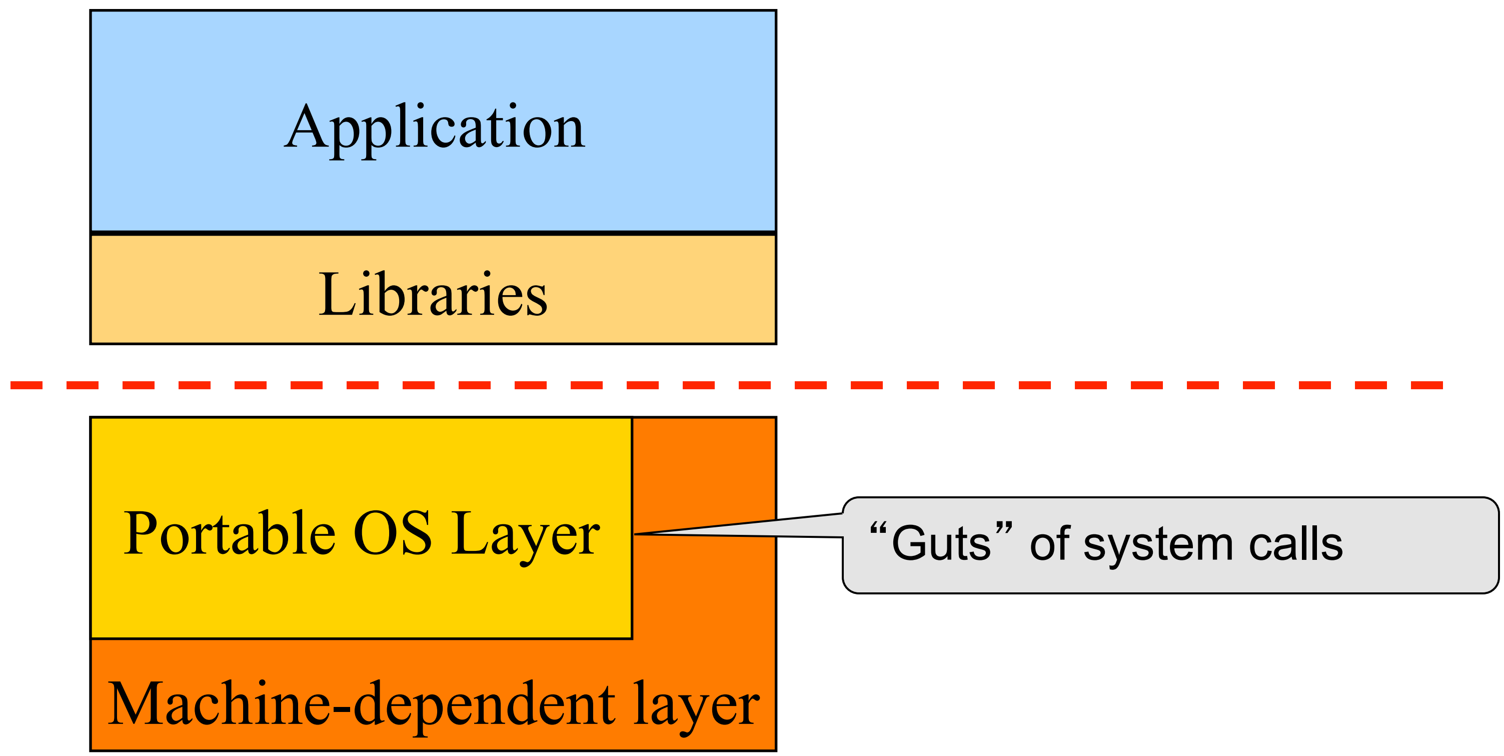
In More Detail

High Address	Args and env vars	Command line arguments and environment variables
	Stack | V	
	Unused memory	
	^ | Heap	
	Uninitialized Data Segment (bss)	Initialized to zero by `exec`.
	Initialized Data Segment	Read from the program file by `exec`.
Low Address	Text Segment	Read from the program file by `exec`.

Responsibilities for the segments

- Stack
  - Layout by ?
  - Allocated/deallocated by ?
  - Names are absolute/relative? Local/global?
- Heap
  - Who sets the starting address?
  - Allocated/deallocated by ?
  - How do application programs manage it?
- Global data/code
  - Who allocates?
  - Who defines names and references?
  - Who translates references?
  - Who relocates addresses?
  - Who lays them out in memory?

Typical Unix OS Structure



Must Support Multiple Applications

- In multiple windows
  - Browser, shell, powerpoint, word, …

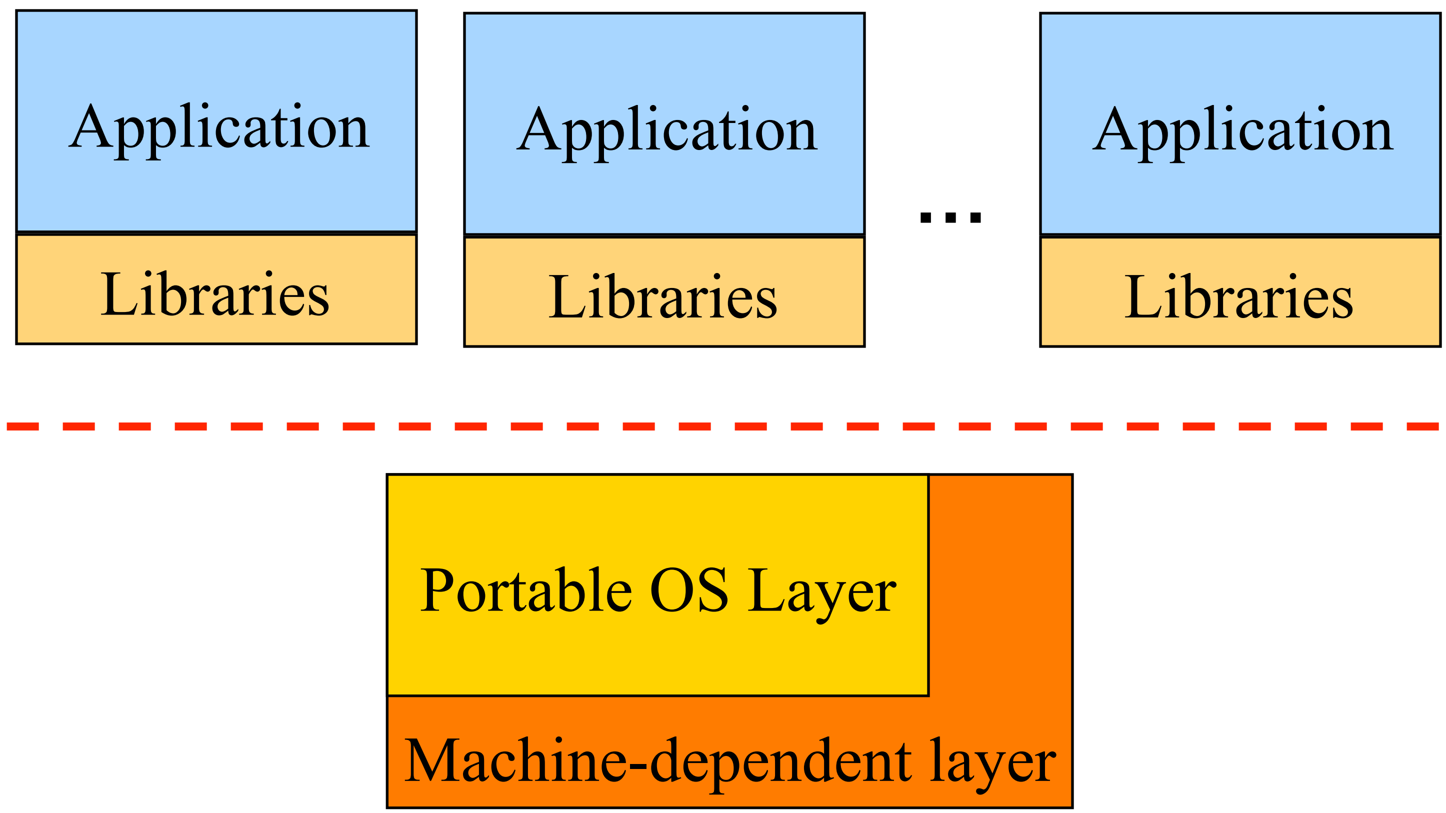
- Use command line to run multiple applications

  % ls –al | grep '^d'

  % foo &

  % bar &

Multiple Application Processes

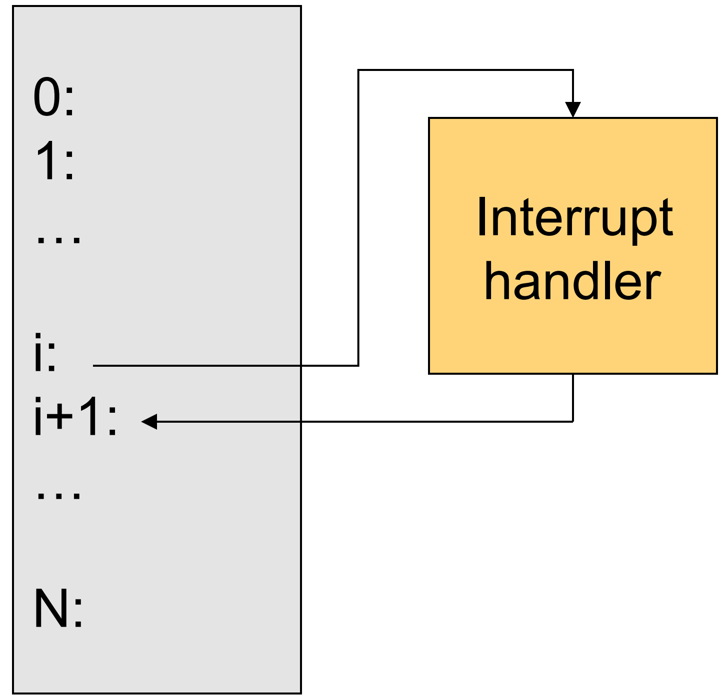


OS Service Examples

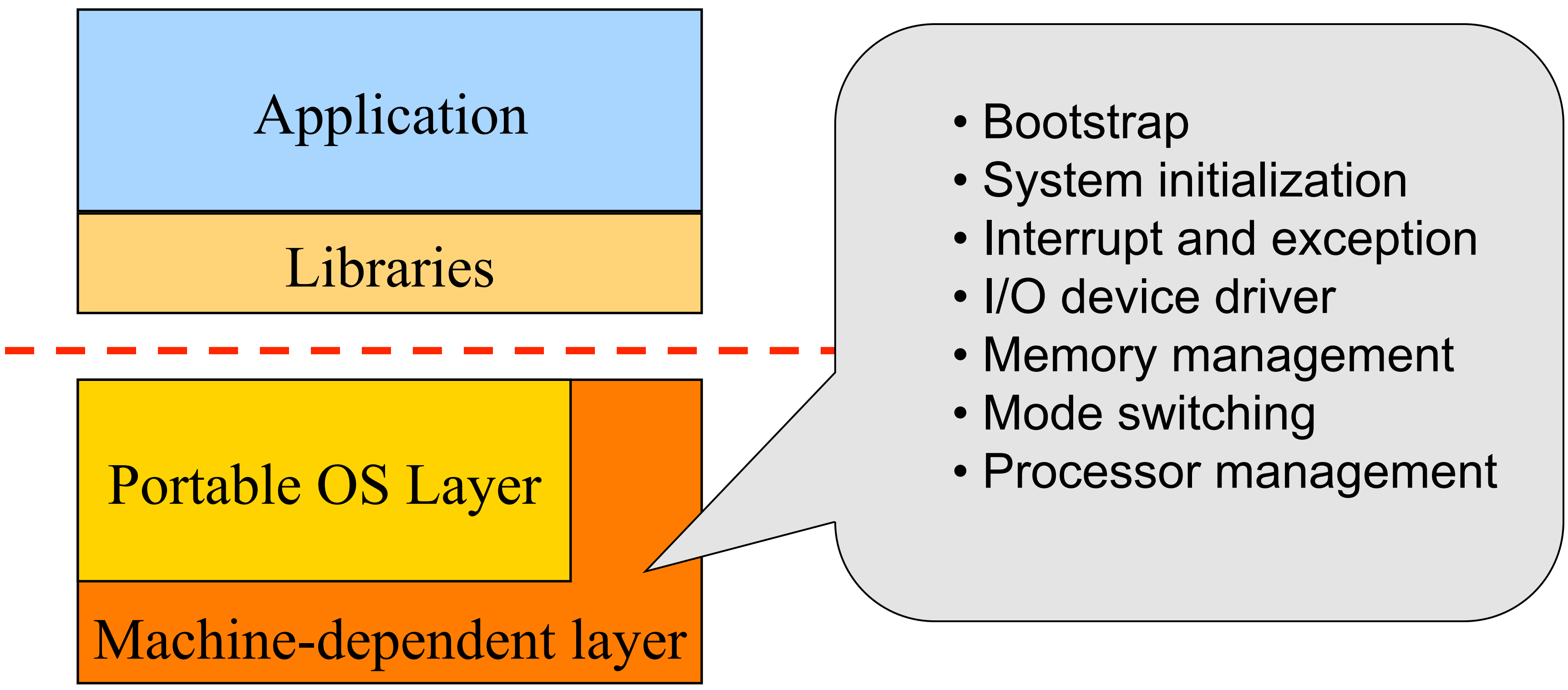
- Examples that are not provided at user level
  - System calls: file open, close, read and write
  - Control the CPU so that users won't cause problems
    - while (1) ;
  - Protection:
    - Keep user programs from crashing OS
    - Keep user programs from crashing each other
- System calls are typically 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

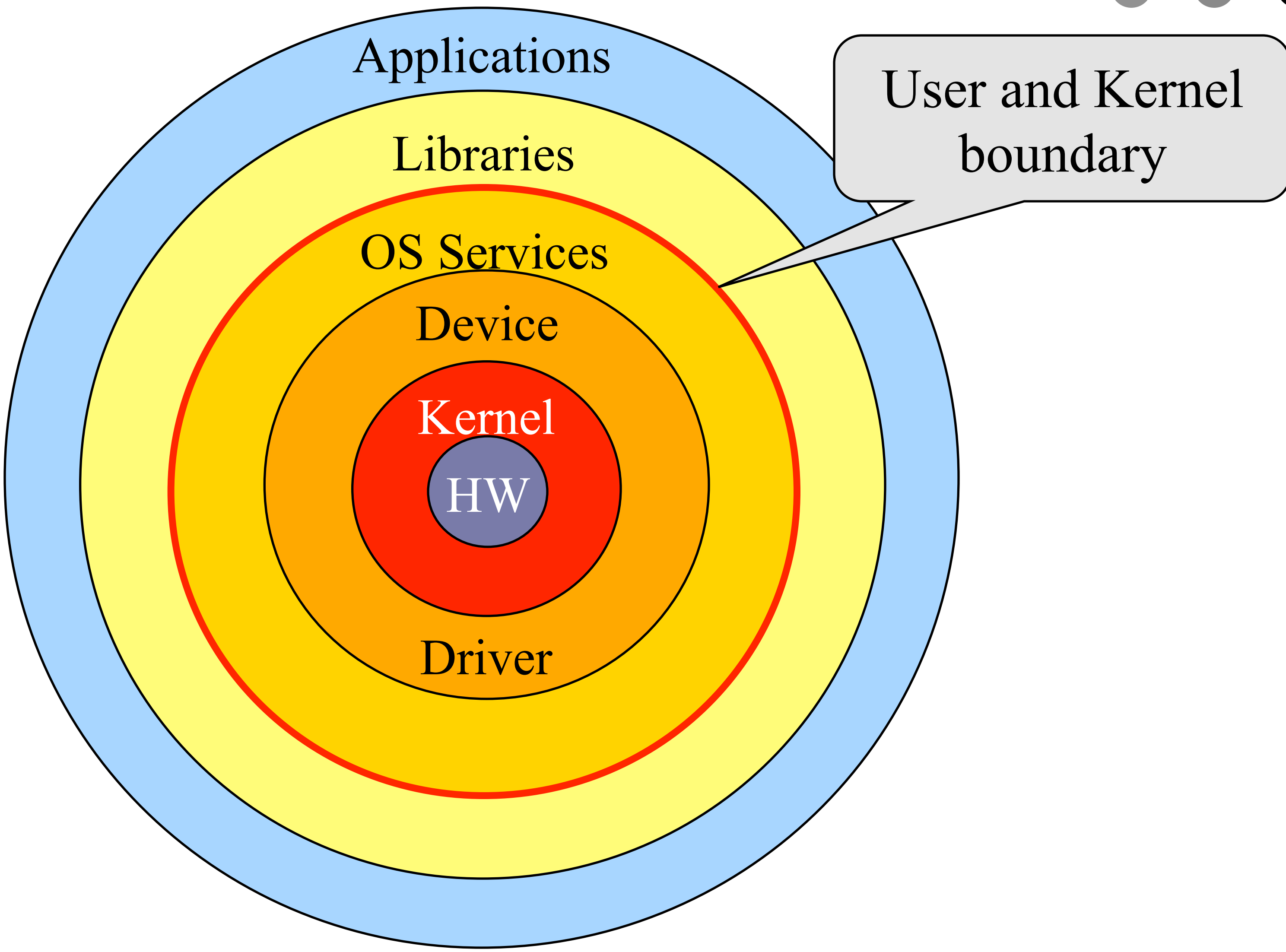
- Raised by external events
- Interrupt handler is in the kernel
  - Switch to another process
  - Overlap I/O with CPU
  - …
- Eventually resume the interrupted process
- A way for CPU to wait for long-latency events (like I/O) to happen



Typical Unix OS Structure



Software "Onion" Layers

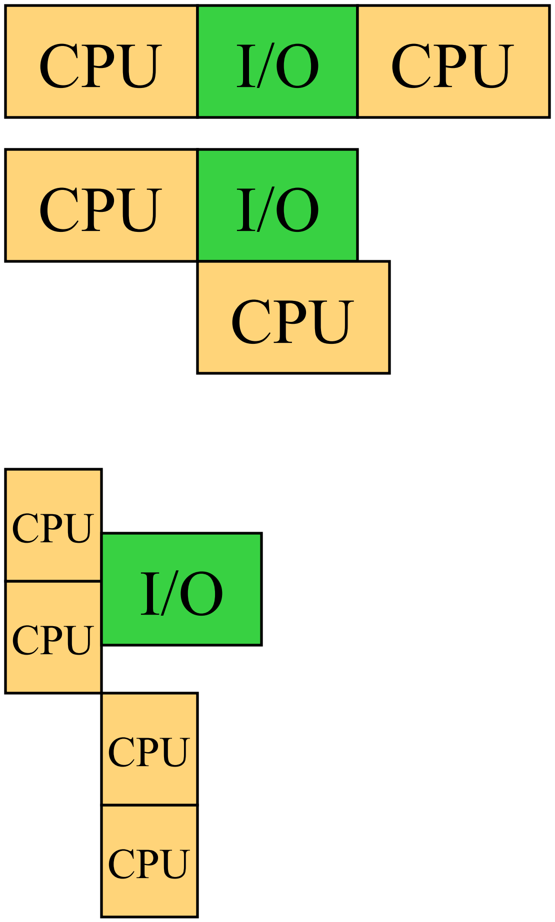


Today

- Overview of OS functionality
- Overview of OS components
  - Process management
  - Memory management
  - I/O device management
  - File System
  - Window System
  - Bootstrap

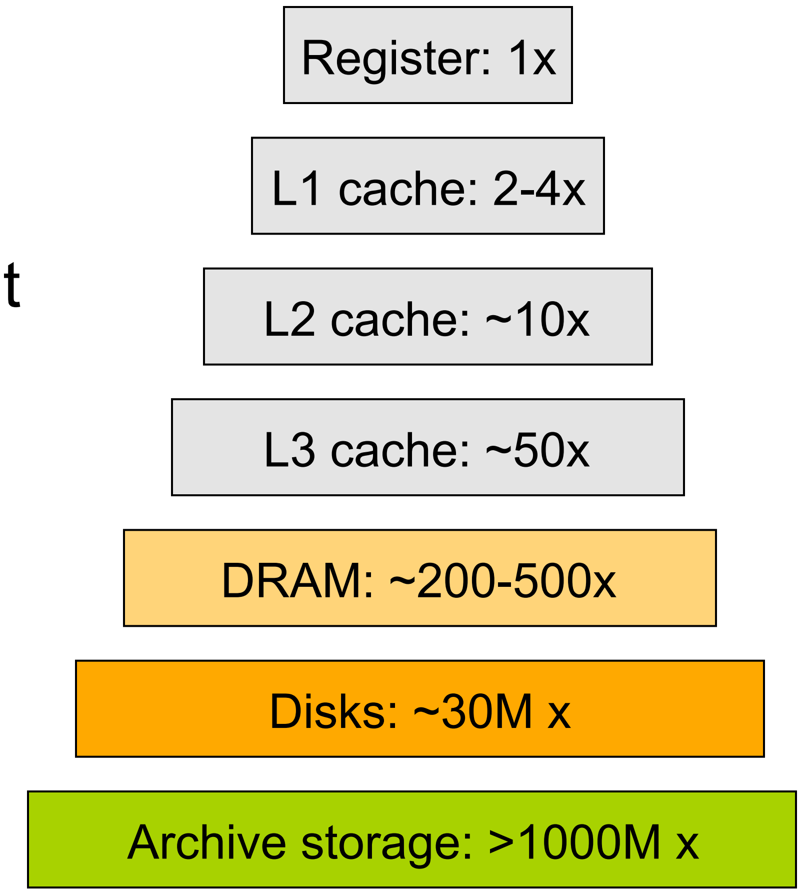
Processor Management

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



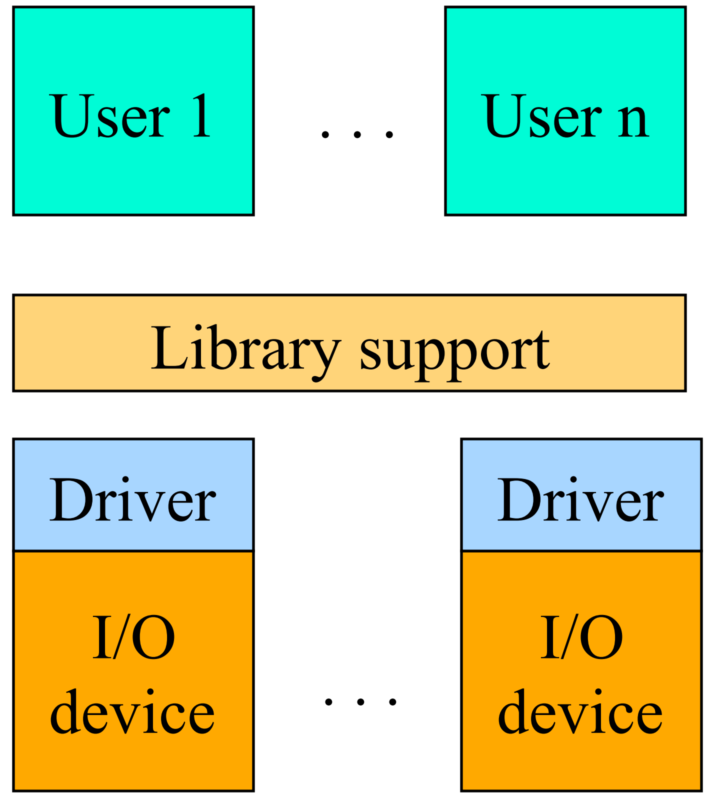
Memory Management

- Goals
  - Support for programs to be written easily
  - Allocation and management
  - Transfers from and to secondary storage
- Issues
  - Efficiency & convenience
  - Fairness
  - Protection



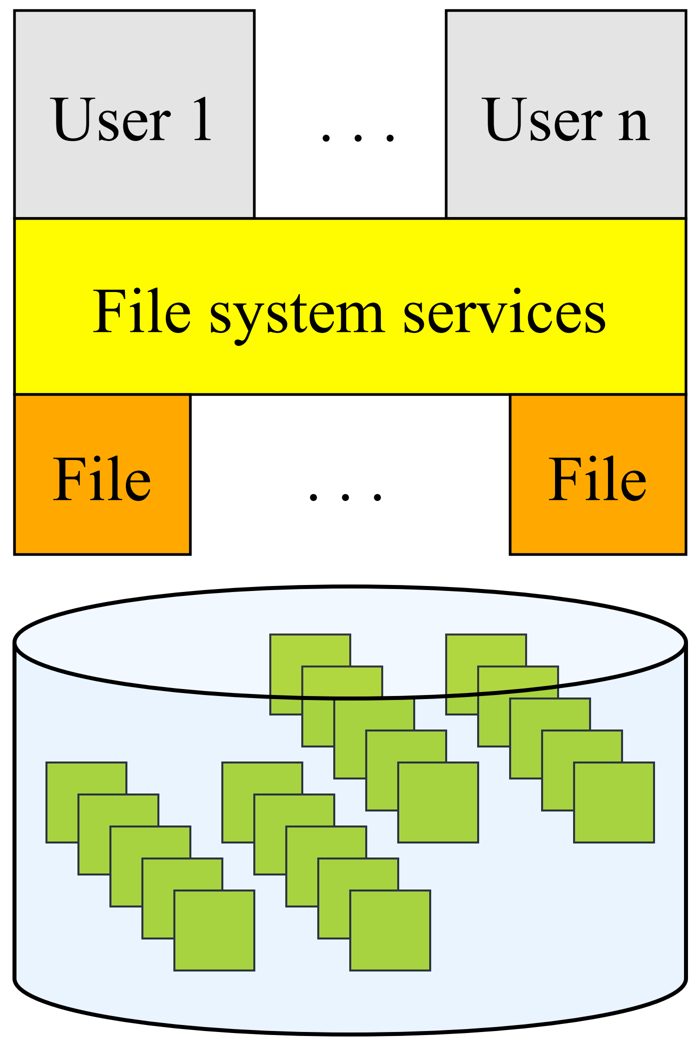
I/O Device Management

- 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
- Typical file system calls
  - Open a file with authentication
  - Read/write data in files
  - Close a file
- Issues
  - Reliability
  - Safety
  - Efficiency
  - Manageability

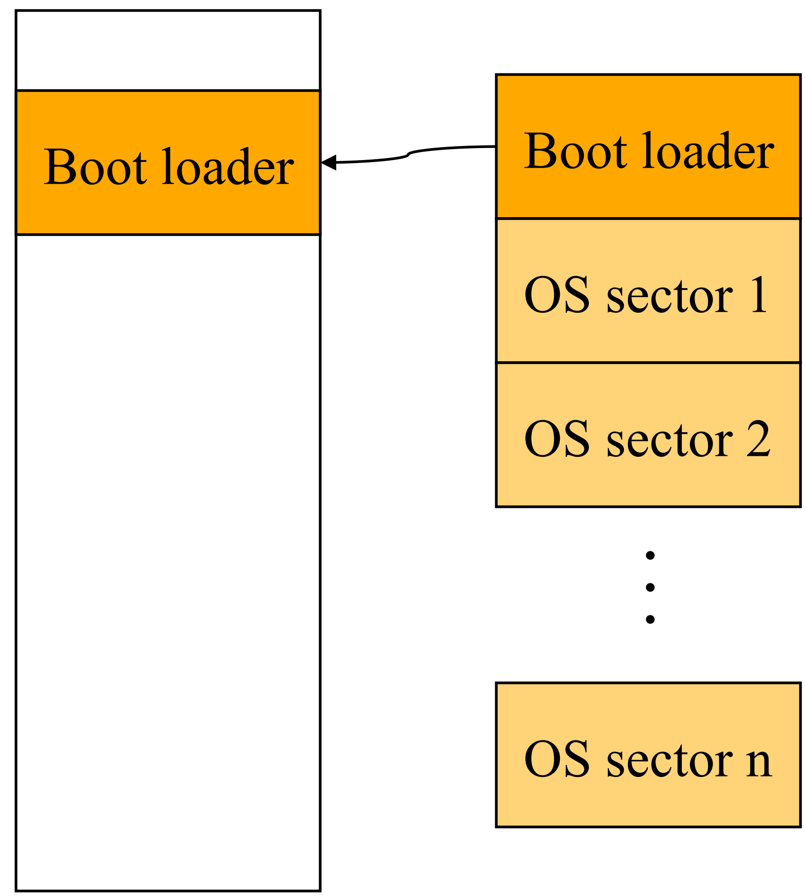


Window Systems

- Goals
  - Interacting with a user
  - Interfaces to examine and manage apps and the system
- Issues
  - Inputs from keyboard, mouse, touch screen, …
  - Display output from applications and systems
  - Where is the Window System?
    - 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



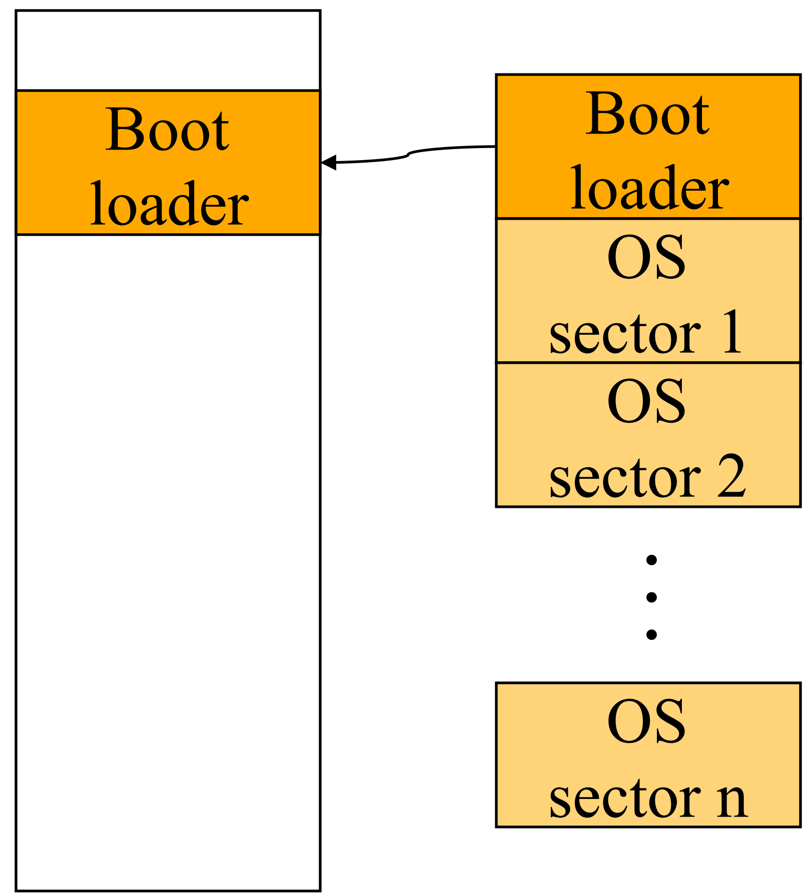
Summary

- Overview of OS functionality
  - Layers of abstraction
  - Services to applications
  - Resource management
- Overview of OS components
  - Processor management
  - Memory management
  - I/O device management
  - File system
  - Window system
  - …

Appendix: Booting a System

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



System Boot

- Power on (processor waits until Power Good Signal)
- Processor jumps to a fixed address, which is the start of the ROM BIOS program

ROM Bios Startup Program (1)

- POST (Power-On Self-Test)
  - 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
- Display startup screen
  - BIOS information
- Execute more tests
  - memory
  - system inventory

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 interrupt numbers
- 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
- 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