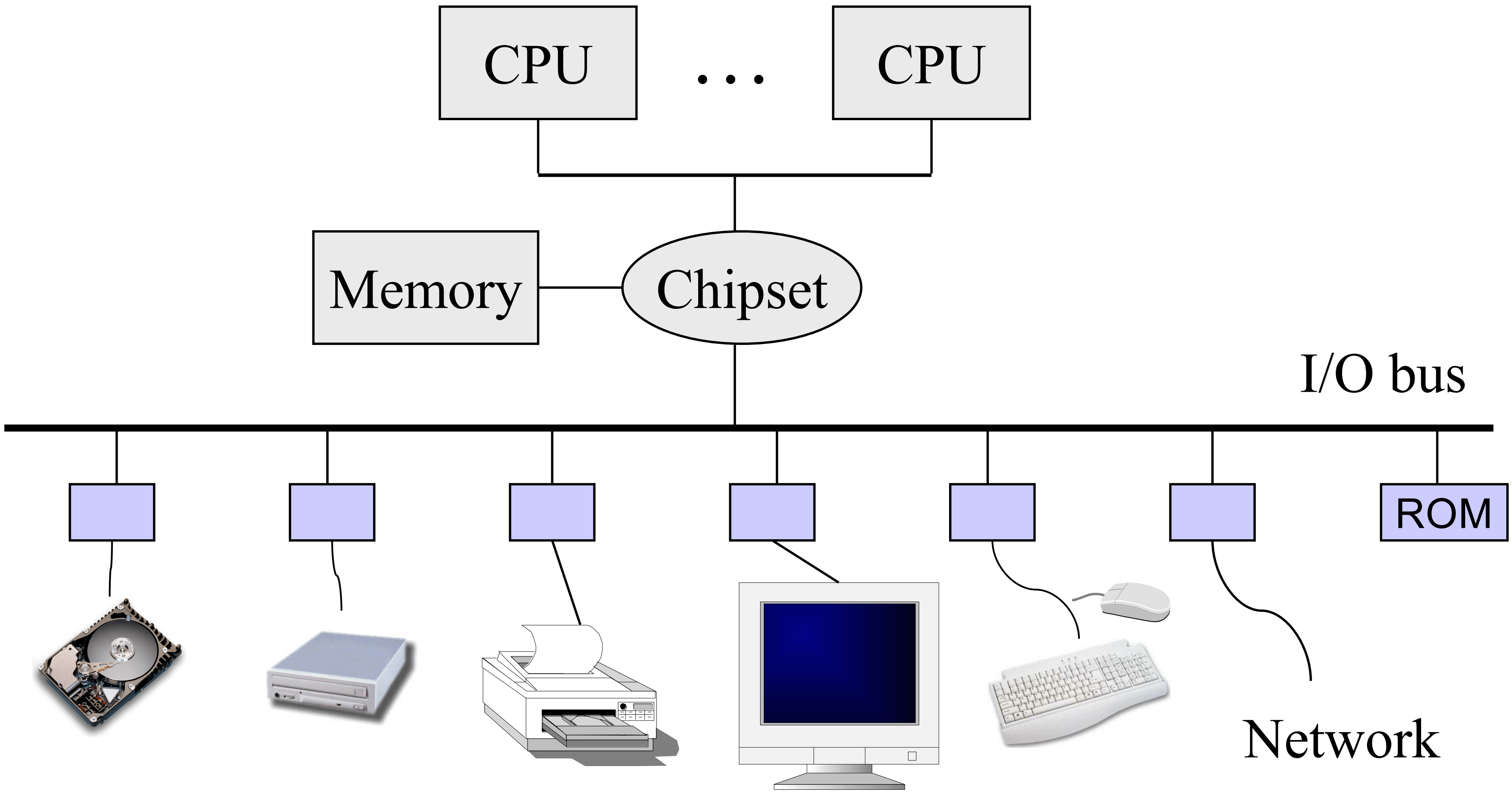
COS 318: Operating Systems

Overview

Today

- Overview of OS functionality
- Overview of OS components
- Interacting with the OS
- Booting a Computer

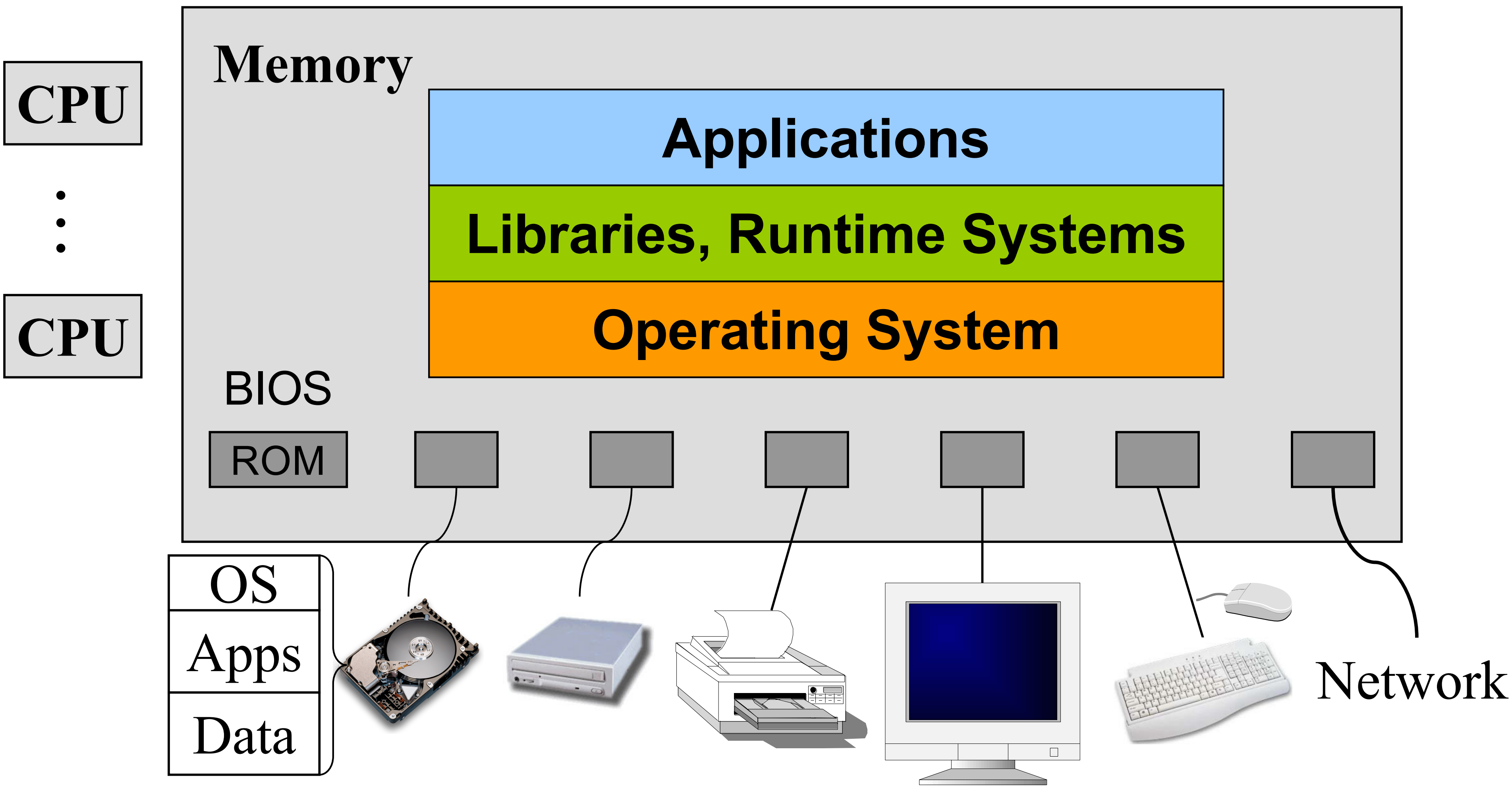
Hardware of A Typical Computer



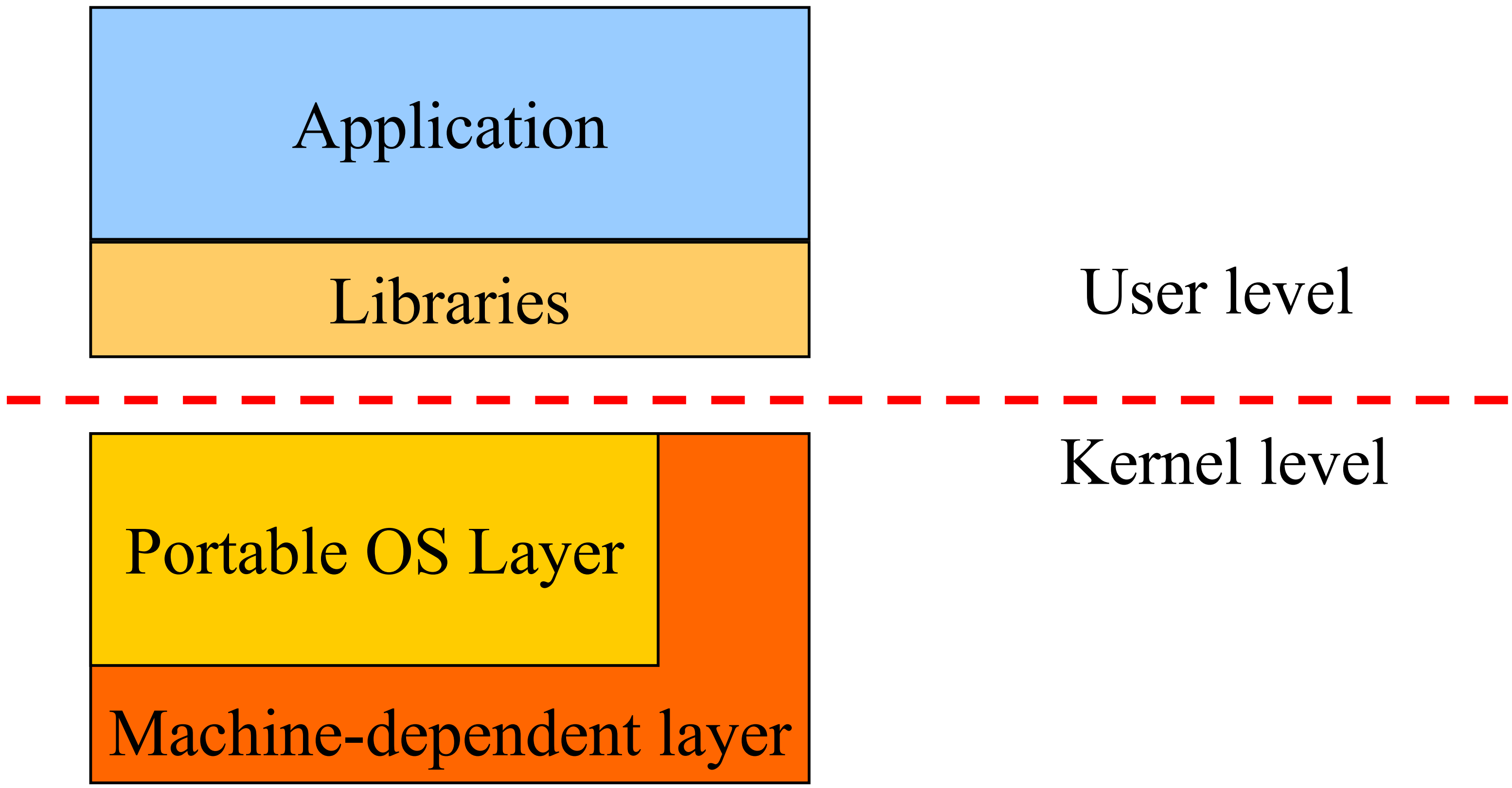
An Overview of HW Functionality

- **Executing machine code** (CPU, cache, memory)
  - Instructions for ALU, branch, memory operations
  - Instructions for communicating with I/O devices
- **Performing I/O operations**
  - I/O devices and the CPU can execute concurrently
  - Every device controller is in charge of one device type
  - Every device controller has a local buffer
  - CPU moves data btwn main memory and local buffers
  - I/O is btwn device and local buffer of device controller
  - Device controller uses **interrupts** to inform CPU it's done
- **Protection**
  - Timer, paging (e.g. TLB), mode bit (e.g. kernel/user)

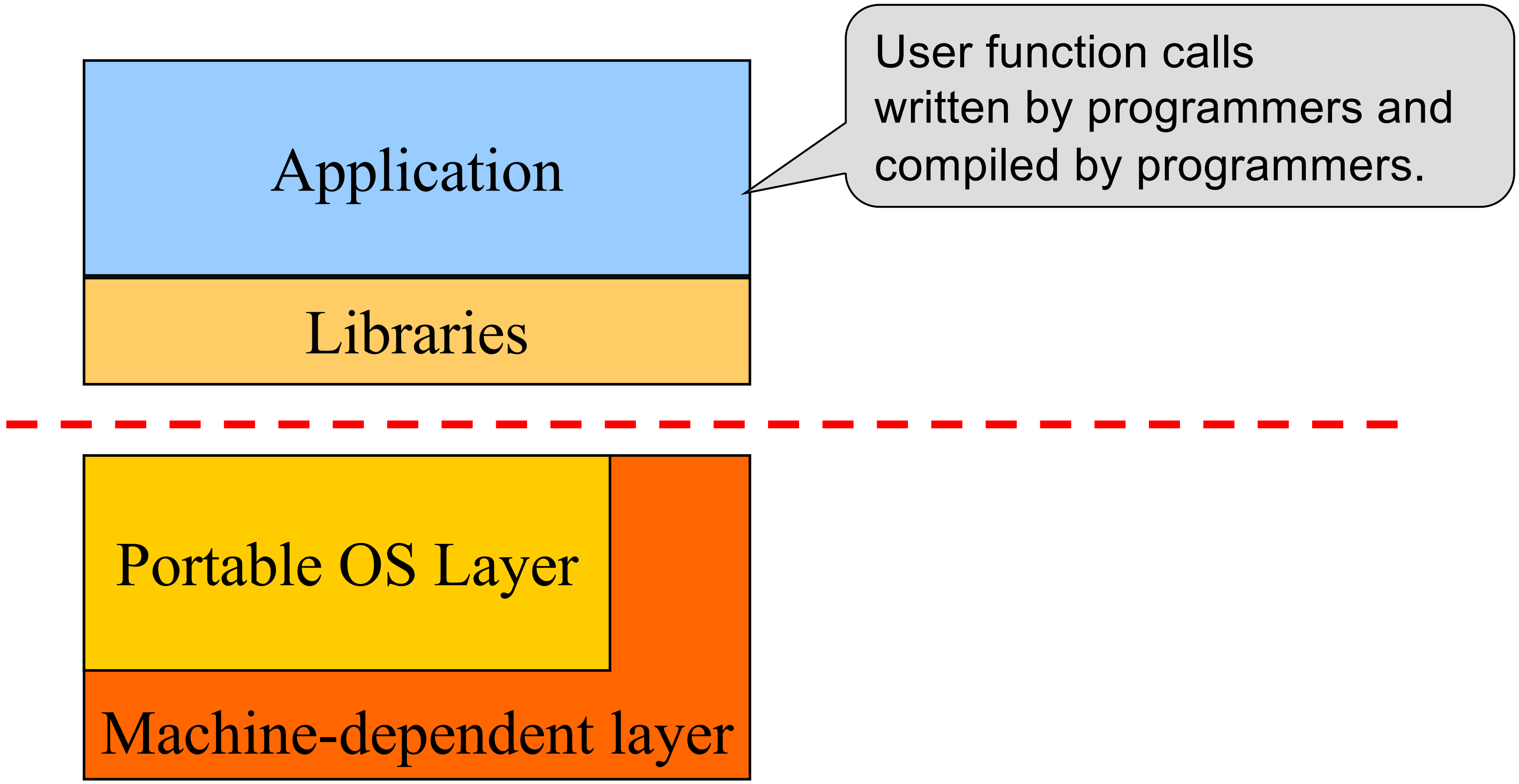
Software in a Typical Computer



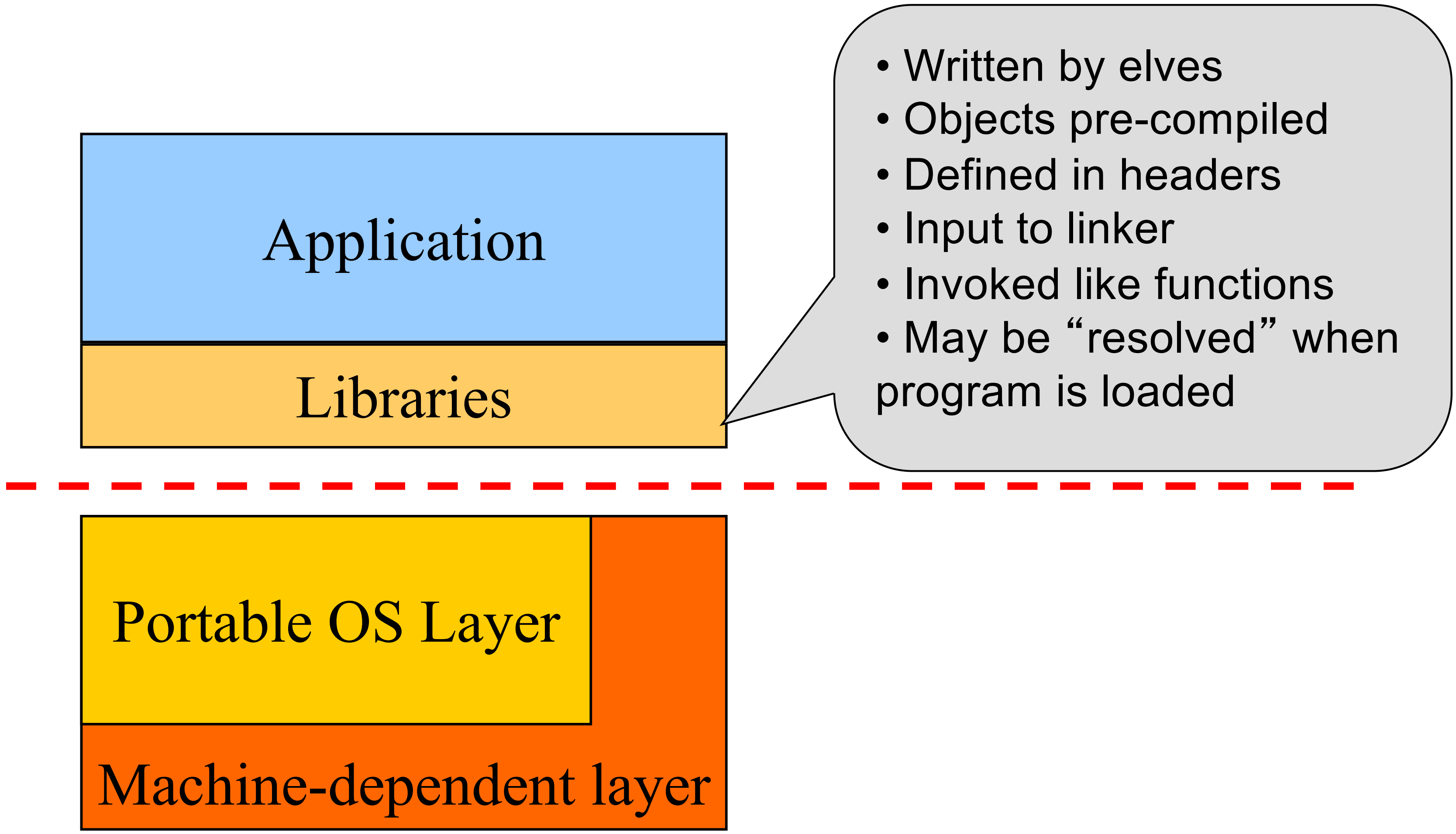
Typical Unix OS Structure



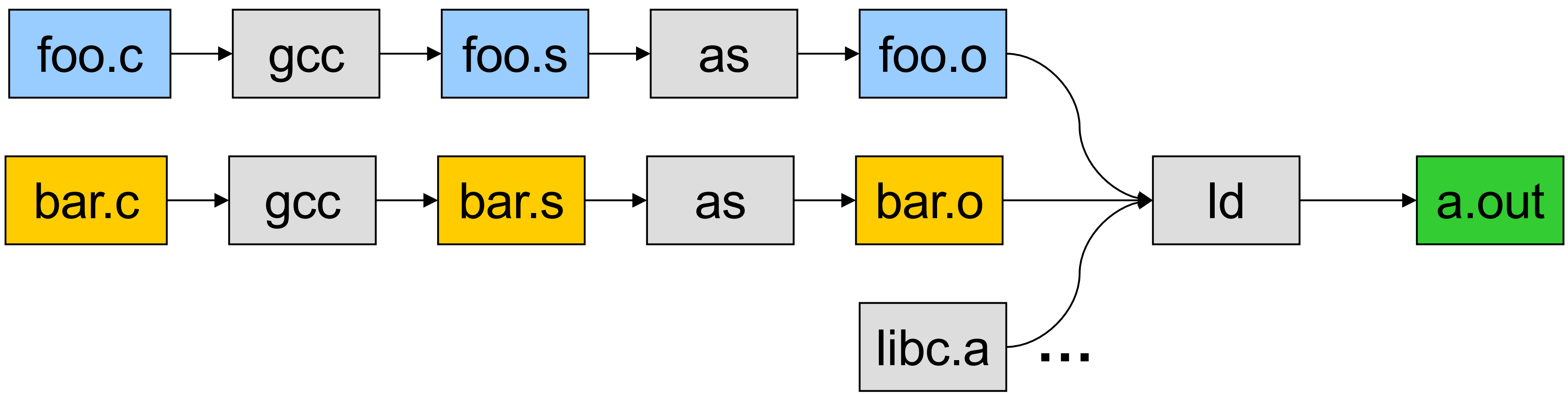
Typical Unix OS Structure



Typical Unix OS Structure



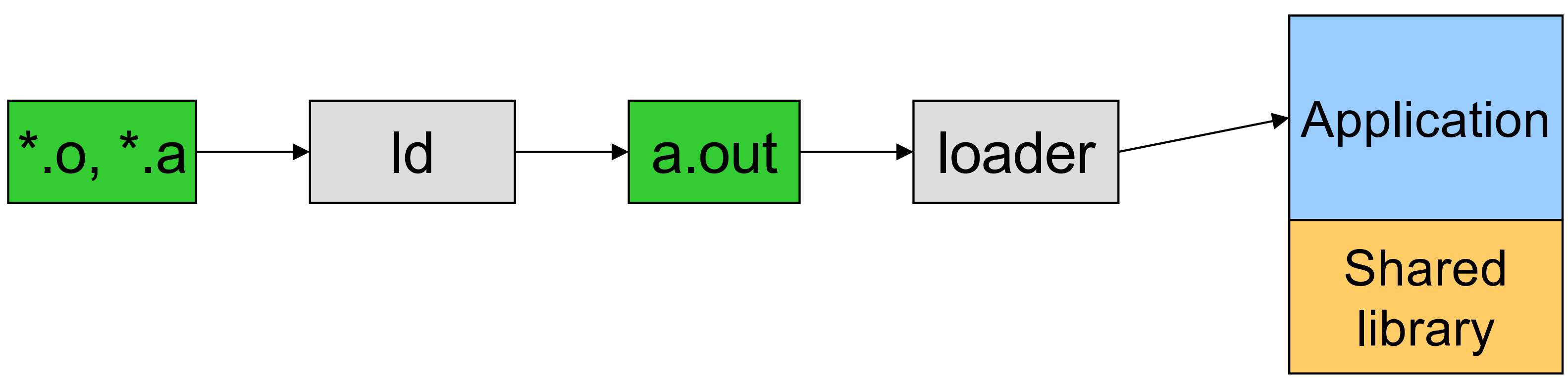
Quick Review: How Application is Created



- 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
- Q: What does the loader do?

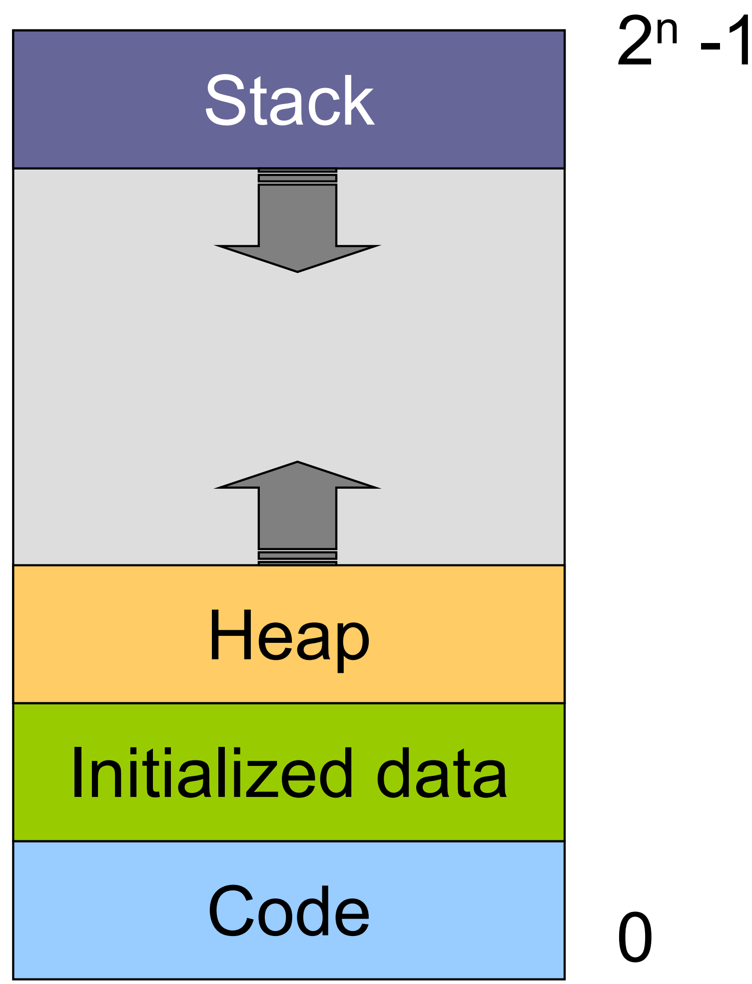
Application: How it's executed

- On Unix, "loader" does the following:
  - 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

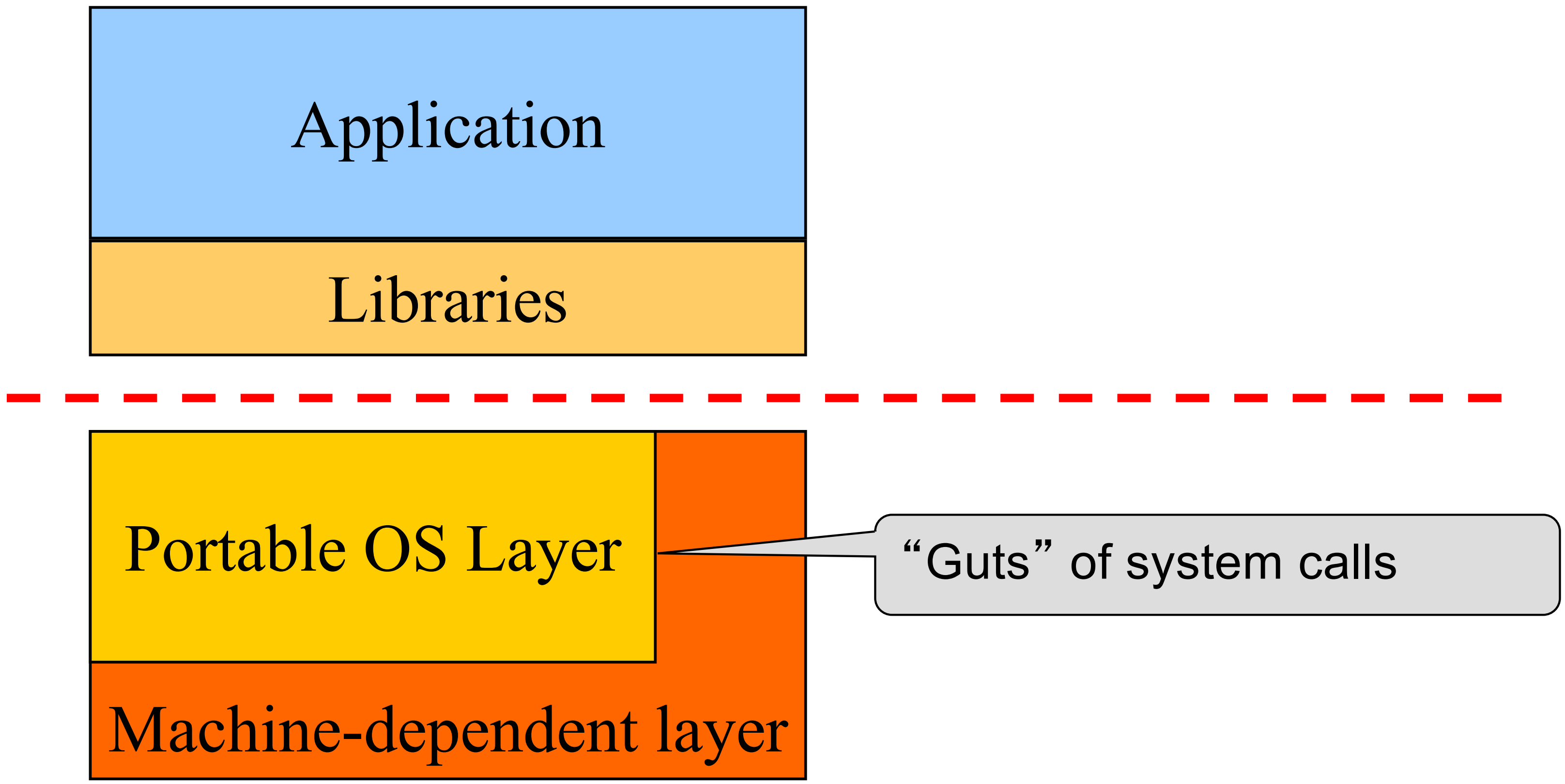
- Four segments
  - Code/Text – instructions
  - Data – global variables
  - Stack
  - Heap
- Why:
  - Separate code and data?
  - Have stack and heap go towards each other?



Responsibilities for the segments

- Stack
  - Layout by ?
  - Allocated/deallocated by ?
  - Local names are absolute/relative?
- 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, Zoom, shell, Powerpoint, Word, …

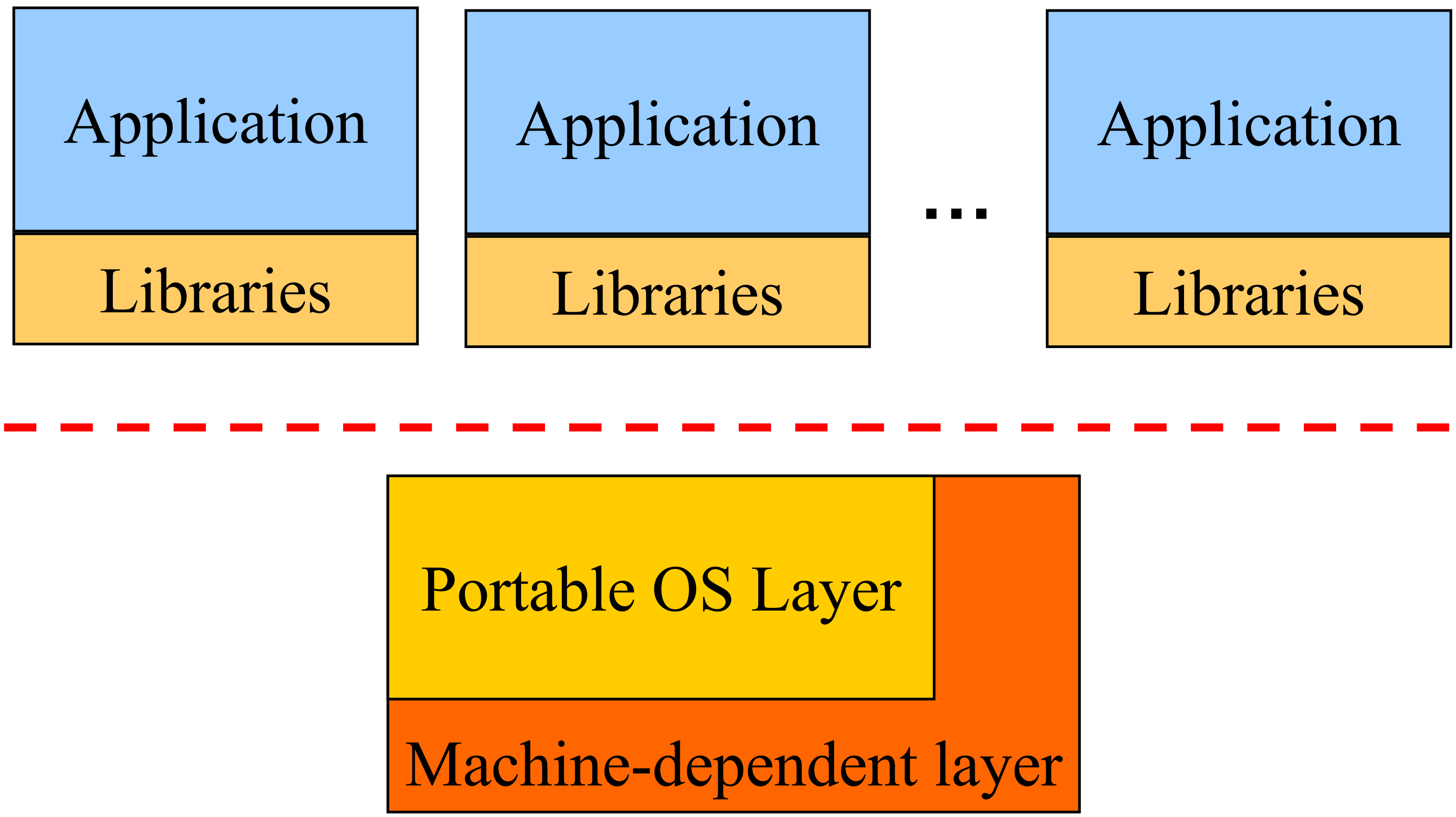
- Command line commands run multiple applications

  % ls –al | grep ‘^d’

  % foo &

  % bar &

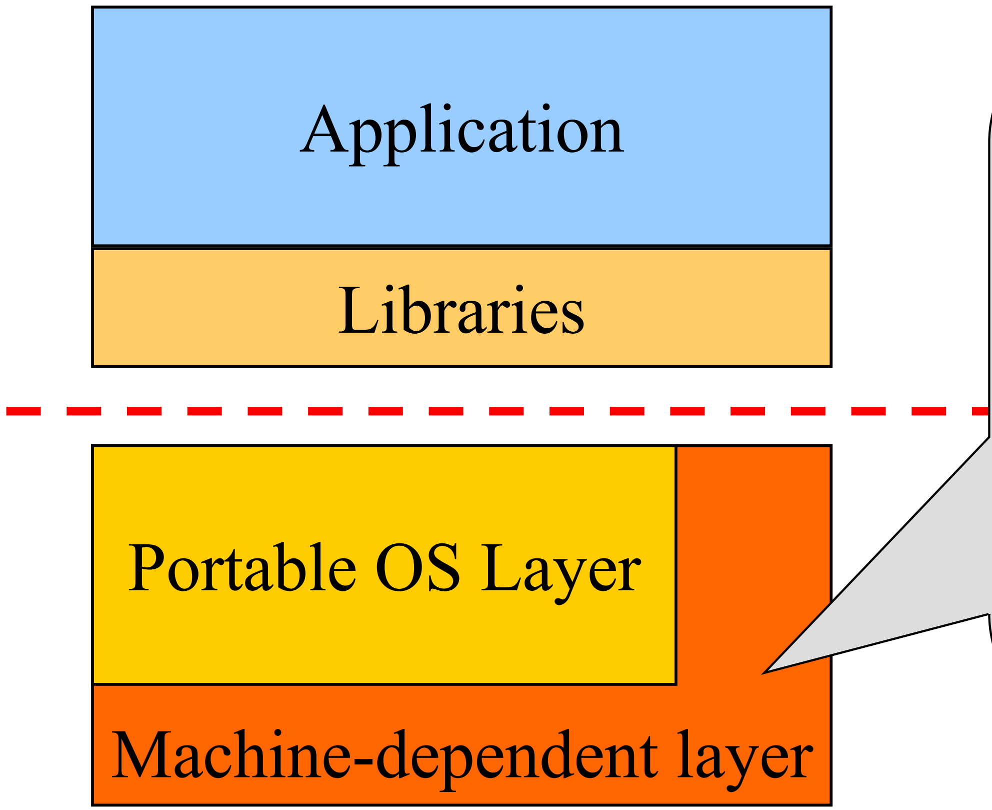
Multiple Application Processes



OS Service Examples

- 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

Typical Unix OS Structure



- Bootstrap
- System initialization
- Interrupt and exception
- I/O device driver
- Memory management
- Mode switching
- Processor management

Today

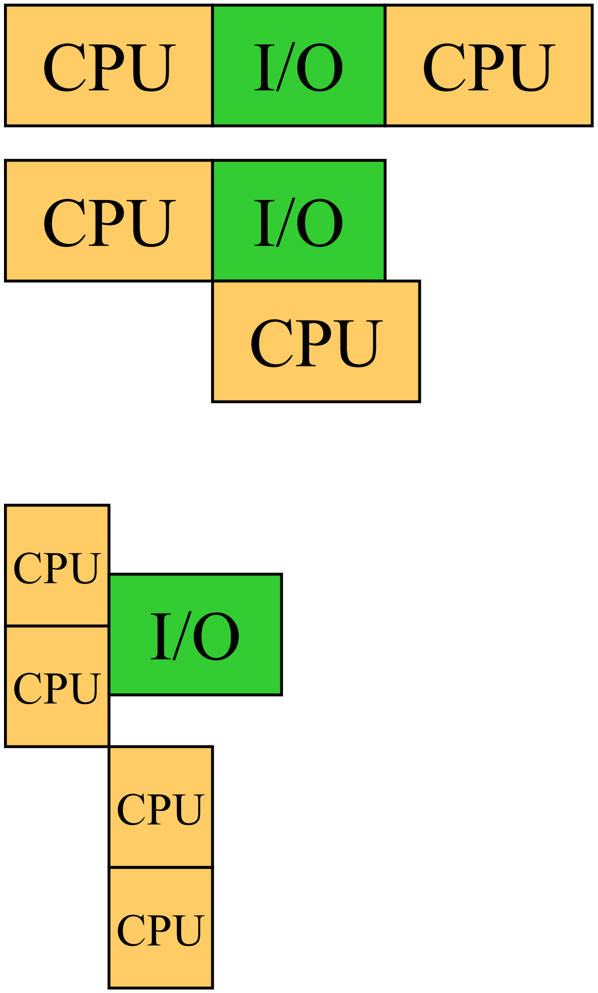
- Overview of OS functionality
- Overview of OS components
- Interacting with the OS
- Booting a Computer

OS components

- Resource manager for each HW resource
  - CPU: processor management
  - RAM: memory management
  - Disk: file system and secondary-storage management
  - I/O device management (keyboards, mouse, network)
- Additional services:
  - window manager (GUI)
  - command-line interpreters (e.g., shell)
  - resource allocation and accounting
  - protection
    - Keep user programs from crashing OS
    - Keep user programs from crashing each other

Processor Management

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



Memory Management

- Memory hierarchy
- Goals
  - Support for programs to run faster without complexity
  - Allocation and management
  - Implicit and explicit transfers among levels of hierarchy
- Issues
  - Efficiency & convenience
  - Fairness
  - Protection
- Q: Who/what manages registers, L1, L2, L3, DRAM?

Register: 1x

L1 cache: 2-4x

L2 cache: ~10x

L3 cache: ~50x

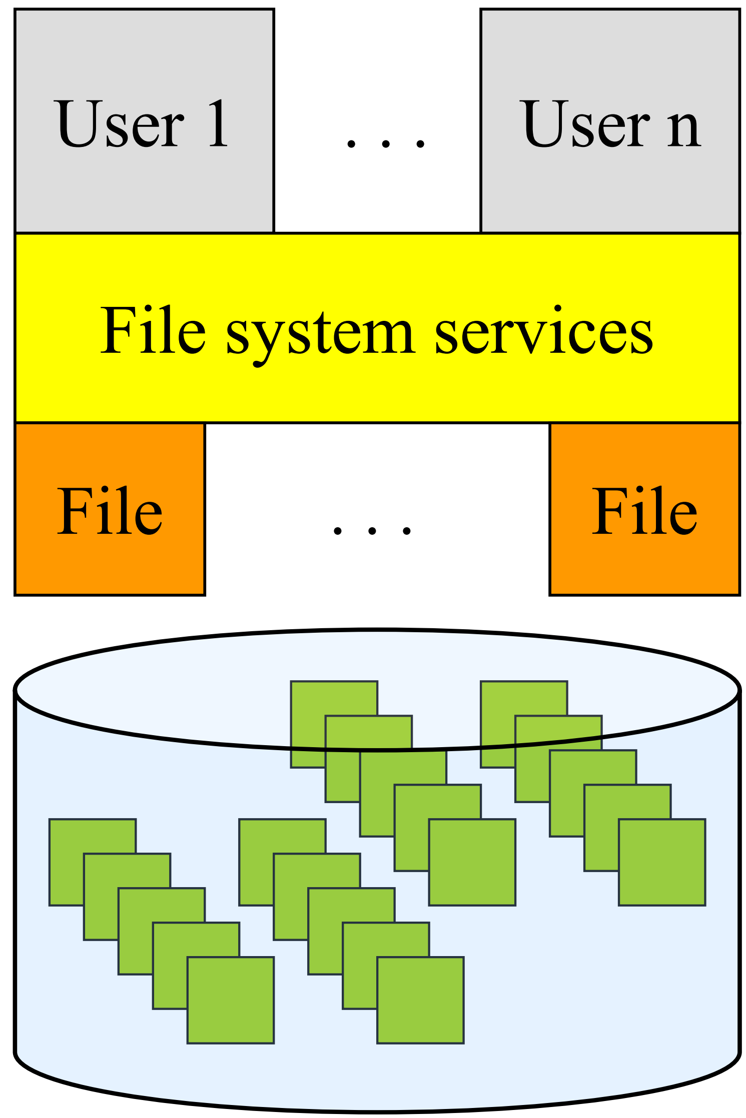
DRAM: ~200-500x

Disks: ~30M x

Archive storage: >1000M x

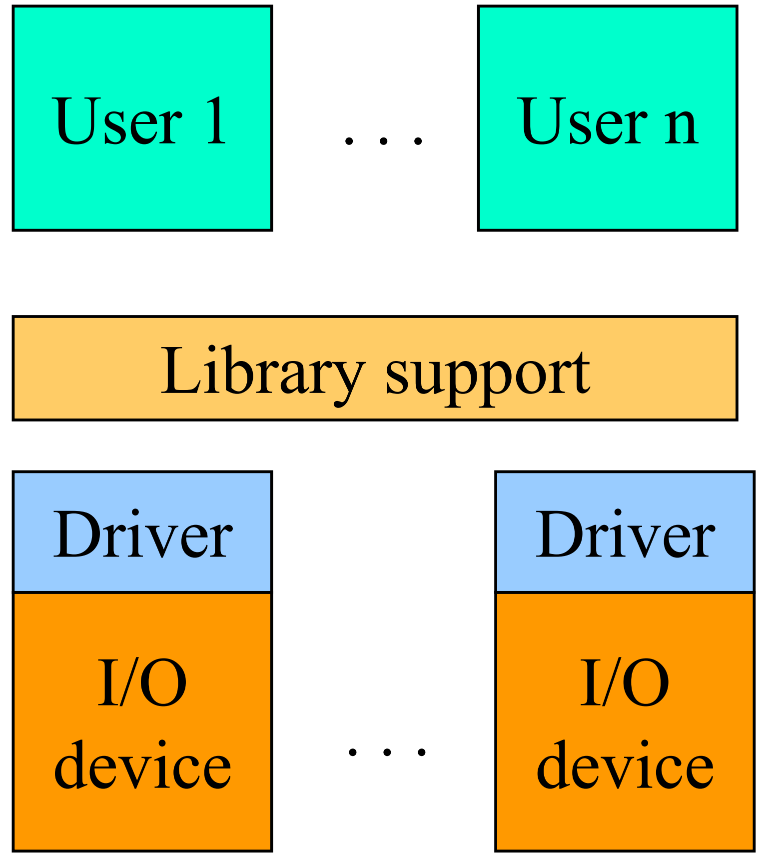
File System

- 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



I/O Device Management

- Goals
  - Interactions between devices and applications
  - Ability to plug in new devices
- Issues
  - Diversity of devices, third-party hardware
  - Efficiency
  - Fairness
  - Protection and sharing



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)

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
  - …

Outline

- Overview of OS functionality
- Overview of OS components
- Interacting with the OS
- Booting a Computer

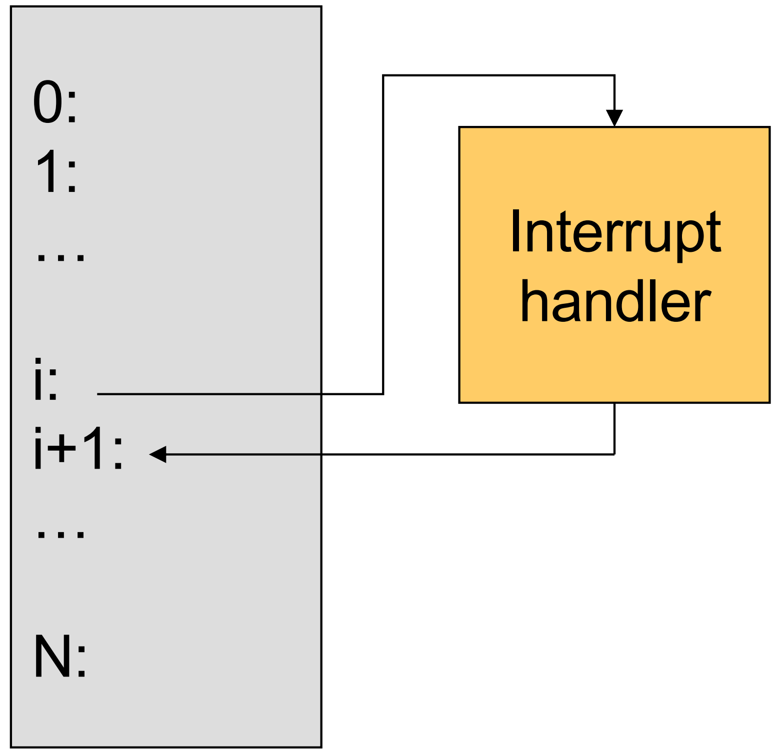
How the OS is Invoked

- Exceptions
  - Normal or program error: traps, faults, aborts
  - Special software generated: INT 3
  - Machine-check exceptions
- Interrupts
  - Hardware (by external devices)
  - Software: INT n
- System calls?
  - Generate a trap

- See Intel document volume 3 for details

Interrupts

- Raised by external events
- Interrupt handler is in kernel
- Eventually resume the interrupted process
- A way to
  - Switch CPU to another process
  - Overlap I/O with CPU
  - Handle other long-latency events



Interrupt and Exceptions (1)

Vector #	Mnemonic	Description	Type
0	#DE	Divide error (by zero)	Fault
1	#DB	Debug	Fault/trap
2		NMI interrupt	Interrupt
3	#BP	Breakpoint	Trap
4	#OF	Overflow	Trap
5	#BR	BOUND range exceeded	Trap
6	#UD	Invalid opcode	Fault
7	#NM	Device not available	Fault
8	#DF	Double fault	Abort
9		Coprocessor segment overrun	Fault
10	#TS	Invalid TSS (Task State Segment). Kernel/HW bug.	

Interrupt and Exceptions (2)

Vector #	Mnemonic	Description	Type
11	#NP	Segment not present	Fault
12	#SS	Stack-segment fault	Fault
13	#GP	General protection	Fault
14	#PF	Page fault	Fault
15		Reserved	Fault
16	#MF	Floating-point error (math fault)	Fault
17	#AC	Alignment check	Fault
18	#MC	Machine check	Abort
19-31		Reserved	
32-255		User defined	Interrupt

System Calls

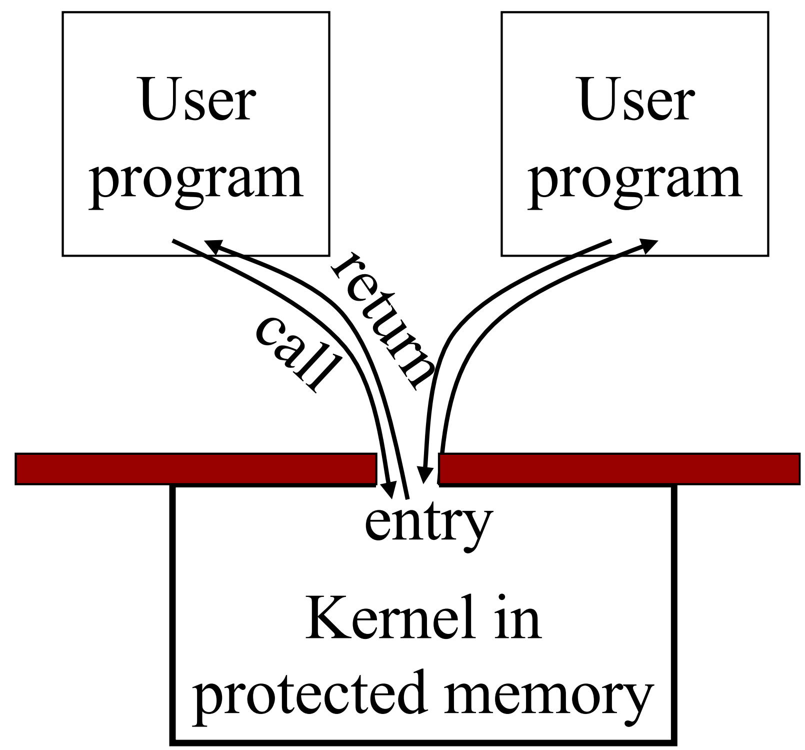
- Operating system API
  - Interface between an application and the operating system kernel
- Categories of system calls
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication

How many system calls?

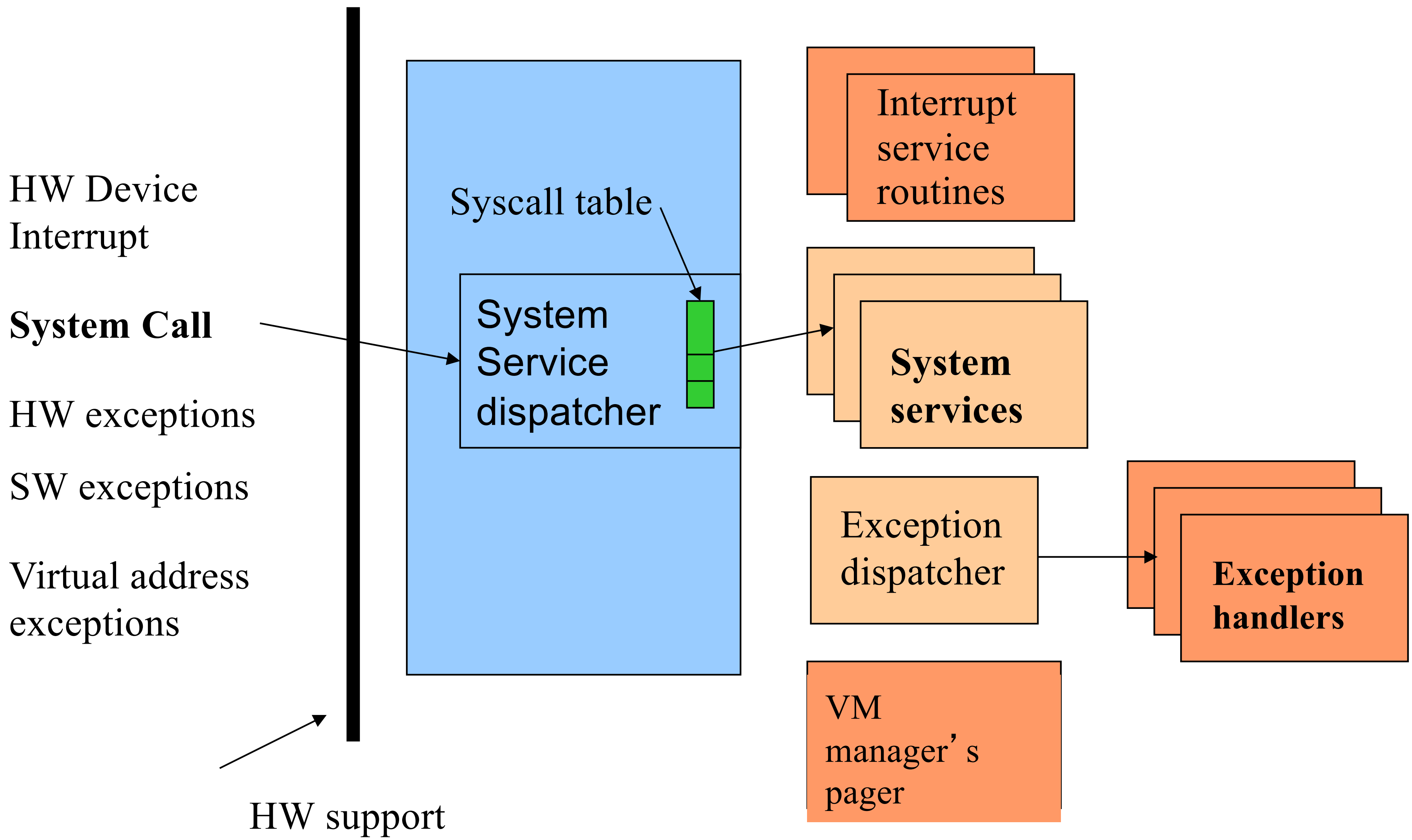
6th Edition Unix:	~45
POSIX:	~130
FreeBSD:	~130
Linux:	~250
Windows 7:	> 900

System Call Mechanism

- Assumptions
  - User code can be arbitrary
  - User code cannot modify kernel memory
- Design Issues
  - User makes a system call with parameters
  - The call mechanism switches code to kernel mode
  - Execute system call
  - Return with results



OS Kernel: Trap Handler



From http://minnie.tuhs.org/UnixTree/V6

V6/usr/sys/ken/sysent.c

Find at most 5 related files. Search

☐ including files from this version of Unix.

```
#
/*
 */

/*
 * This table is the switch used to transfer
 * to the appropriate routine for processing a system call.
 * Each row contains the number of arguments expected
 * and a pointer to the routine.
 */
int	sysent[]
{
	0, &nullsys,			/*  0 = indir */
	0, &rexit,			/*  1 = exit */
	0, &fork,			/*  2 = fork */
	2, &read,			/*  3 = read */
	2, &write,			/*  4 = write */
	2, &open,			/*  5 = open */
	0, &close,			/*  6 = close */
	0, &wait,			/*  7 = wait */
	2, &creat,			/*  8 = creat */
	2, &link,			/*  9 = link */
	1, &unlink,			/* 10 = unlink */
	2, &exec,			/* 11 = exec */
	1, &chdir,			/* 12 = chdir */
	0, &gtime,			/* 13 = time */
	3, &mknod,			/* 14 = mknod */
	2, &chmod,			/* 15 = chmod */
	2, &chown,			/* 16 = chown */
	1, &sbreak,			/* 17 = break */
	2, &stat,			/* 18 = stat */
	2, &seek,			/* 19 = seek */
	0, &getpid,			/* 20 = getpid */
	3, &smount,			/* 21 = mount */
	1, &sumount,			/* 22 = umount */
	0, &setuid,			/* 23 = setuid */
	0, &getuid,			/* 24 = getuid */
	0, &stime,			/* 25 = stime */
	3, &ptrace,			/* 26 = ptrace */
	0, &nosys,			/* 27 = x */
	1, &fstat,			/* 28 = fstat */
	0, &nosys,			/* 29 = x */
	1, &nullsys,			/* 30 = smdate; inoperative */
	1, &stty,			/* 31 = stty */
	1, &gtty,			/* 32 = gtty */
	0, &nosys,			/* 33 = x */
	0, &nice,			/* 34 = nice */
	0, &sslep,			/* 35 = sleep */
	0, &sync,			/* 36 = sync */
	1, &kill,			/* 37 = kill */
	0, &getswit,			/* 38 = switch */
	0, &nosys,			/* 39 = x */
	0, &nosys,			/* 40 = x */
	0, &dup,			/* 41 = dup */
	0, &pipe,			/* 42 = pipe */
	1, &times,			/* 43 = times */
	4, &profil,			/* 44 = prof */
	0, &nosys,			/* 45 = tiu */
	0, &setgid,			/* 46 = setgid */
	0, &getgid,			/* 47 = getgid */
	2, &ssig,			/* 48 = sig */
```

Passing Parameters

- Pass by registers
  - # of registers
  - # of usable registers
  - # of parameters in system call
  - Spill/fill code in compiler
- Pass by a memory vector (list)
  - Single register for starting address
  - Vector in user's memory
- Pass by stack
  - Similar to the memory vector
  - Procedure call convention

Library Stubs for System Calls

- Example:

  int read(int fd, char * buf, int size)

  {

  move fd, buf, size to R_1, R_2, R_3

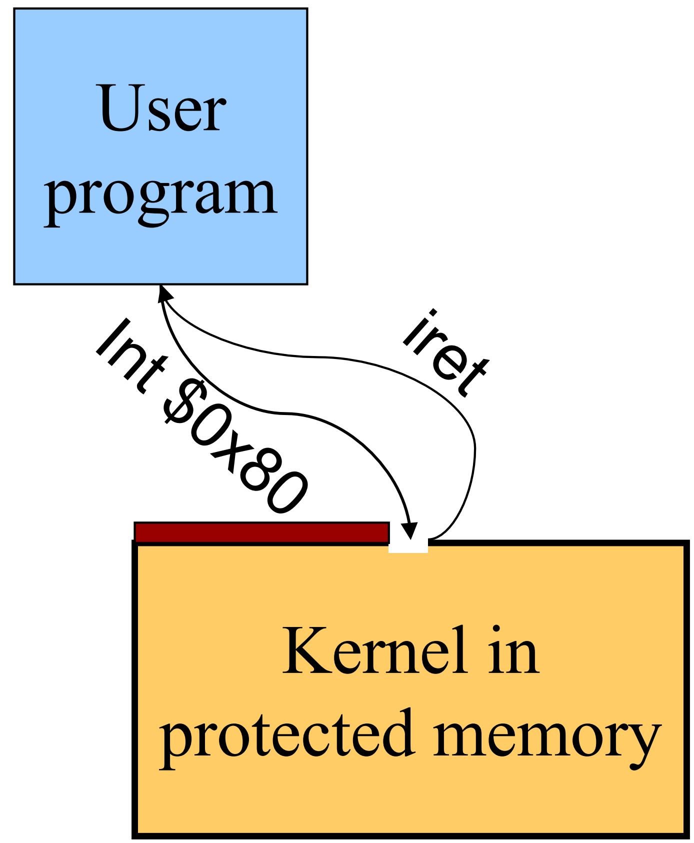
  move READ to R_0

  int $0x80

  move result to R_{result}

  }

Linux: 80
NT: 2E



Q. What system call does int $0x80 correspond to?

System Call Entry Point

EntryPoint:

switch to kernel stack

save context

check R_0

call the real code pointed by R_0

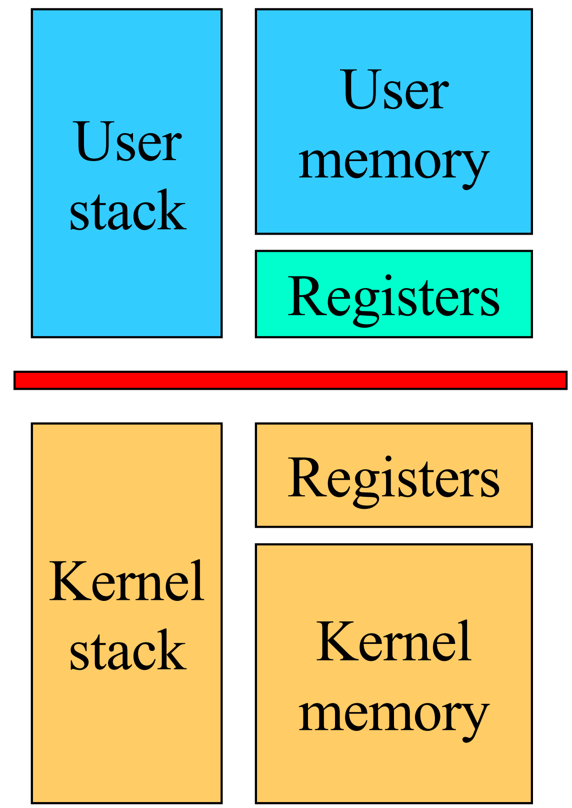
place result in R_{result}

restore context

switch to user stack

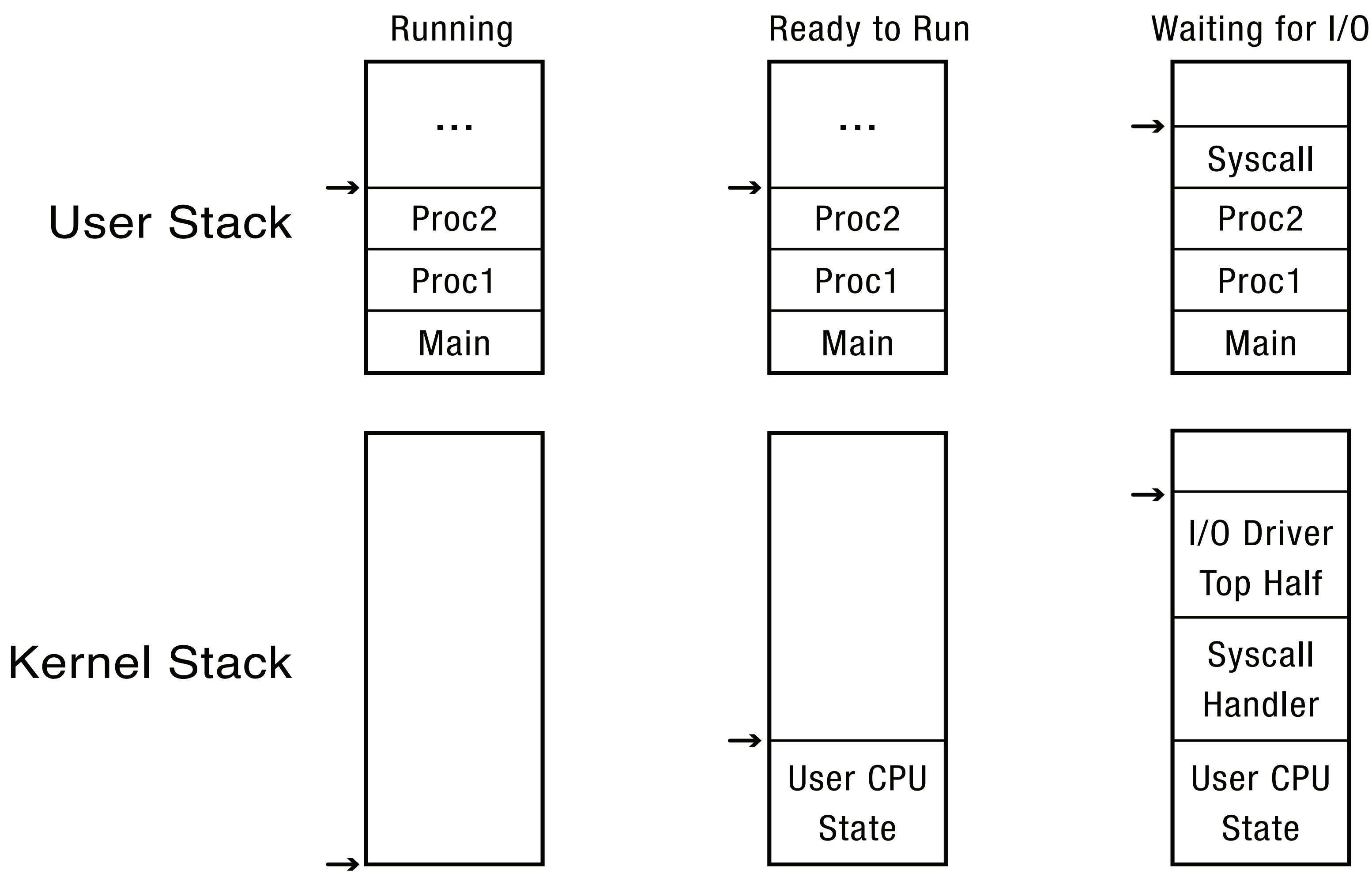
iret (change to user mode and return)

(Assumes passing parameters in registers)



Kernel stacks

Per-processor, located in kernel memory. Why can't the interrupt handler run on the stack of the interrupted user process?



System call stubs

User Program

```
main () {
   file_open(arg1, arg2);
}
```

(1) (6)

User Stub

```
file_open(arg1, arg2) {
   push #SYSCALL_OPEN
   trap
   return
}
```

(2)

Hardware Trap

Trap Return

(5)

Kernel

```
file_open(arg1, arg2) {
      // do operation
}
```

(3) (4)

Kernel Stub

```
file_open_handler() {
   // copy arguments
   //   from user memory
   //   check arguments
   file_open(arg1, arg2);
   // copy return value
   //   into user memory
   return;
}
```

Design Issues

- System calls
  - There is one result register; what about more results?
  - How do we pass errors back to the caller?

- Q. What criteria should you use to decide what should be a system call versus a library call? What are the most important goals for each?

Backward compatibility...

NAME

open - open a file

SYNOPSIS

```
[OH] #include <sys/stat.h>

#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

The use of *open*() to create a regular file is preferable to the use of *creat()*, because the latter is redundant and included only for historical reasons.

Division of Labor (Separation Of Concerns)

Memory management example

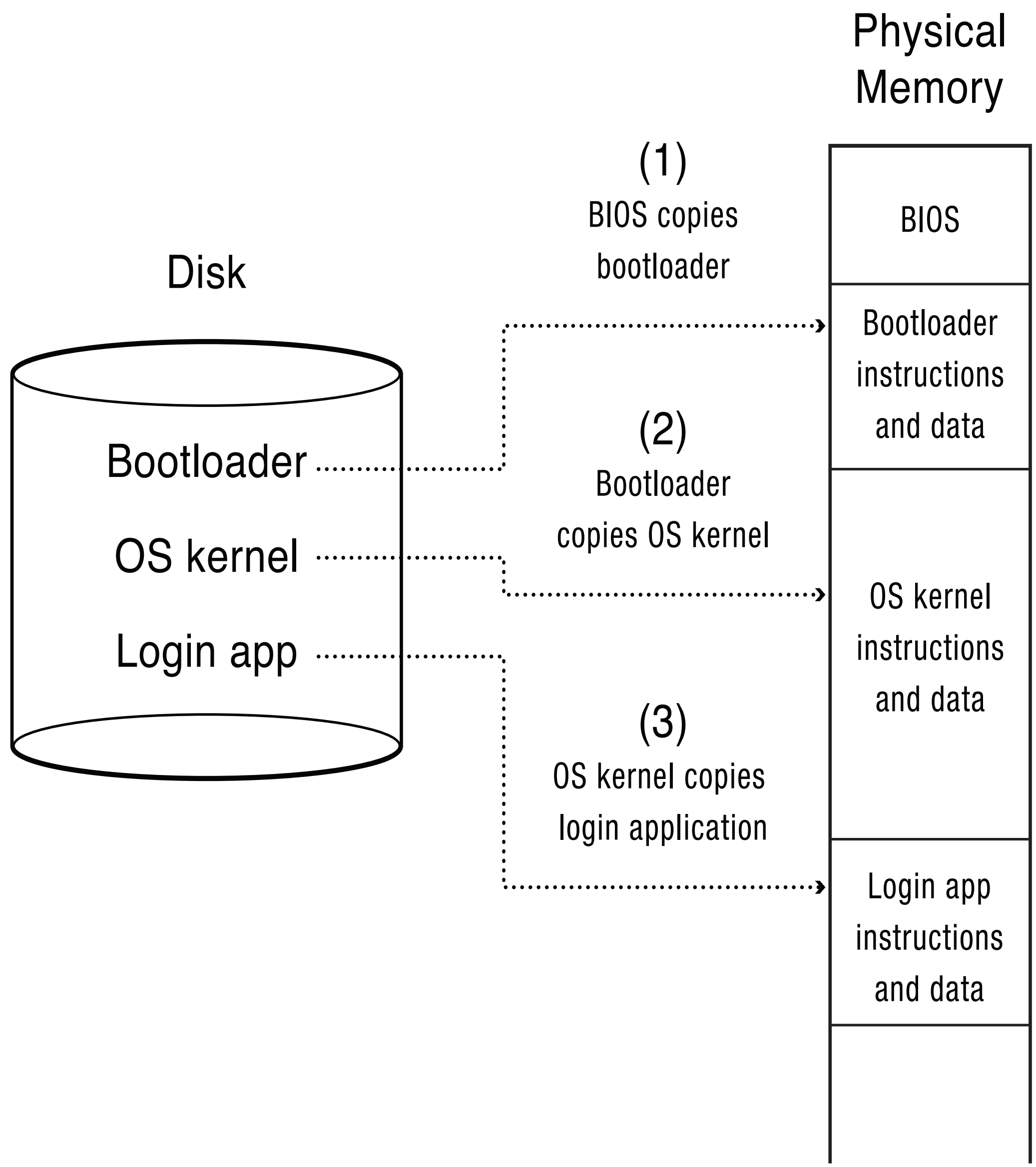
- Kernel
  - Allocates "pages" with protection
  - Allocates a big chunk (many pages) to library
  - Does not care about small allocations
- Library
  - Provides malloc/free for allocation and deallocation
  - Applications use them to manage memory
  - When reaching the end, library asks kernel for more

Today

- Overview of OS functionality
- Overview of OS components
- Interacting with the OS
- Booting a Computer

Booting a Computer

- Power up a computer
- Processor reset
  - Set to known state
  - Jump to ROM code (for x86, this is the BIOS)
- 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 BIOS code (normally at C000h)
- Look for other devices ROM BIOS code
  - IDE/ATA disk ROM BIOS at C8000h 9=818200d)
- 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
  - Hard disk or USB drive or CD/DVD
- 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
  - Could be OS or another feature-rich bootloader (e.g. GRUB), which then loads the actual OS

Summary

- Protection mechanism
  - Architecture support: two modes
  - Software traps (exceptions)
- OS structures
  - Monolithic, layered, microkernel and virtual machine
- System calls
  - Implementation
  - Design issues
  - Tradeoffs with library calls