COS 318: Operating Systems

Overview

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318/

Announcements

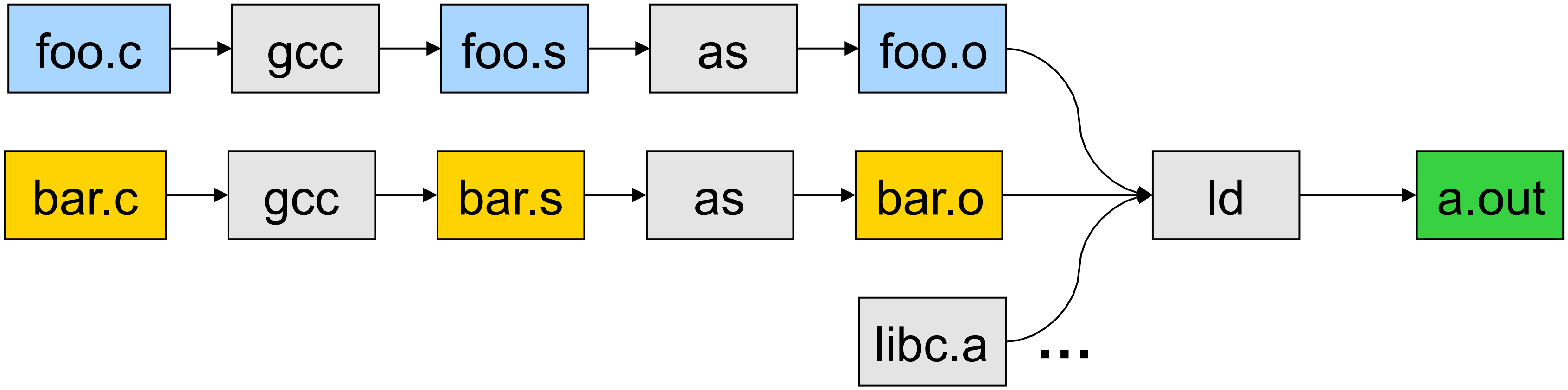
- Precepts:
  - Tue (Tonight)! 7:30pm-8:30pm, 105 CS building
- Design review:
  - Mon 9/26: 6-9pm, 010 Friend center. Sign up online.
- Project 1 due:
  - 10/5 at noon!
- Reminder:
  - Find a project partner and email the pairing to mrm@cs and vivek@cs.
  - (Please cc your partner too, so we… uhhm.. know this is a mutual decision! ☺)

Today

- Overview of OS structure
  - What does the OS need to do?
  - What other support/functionality does it build on from hardware?
- Overview of OS components

A view from user-level software

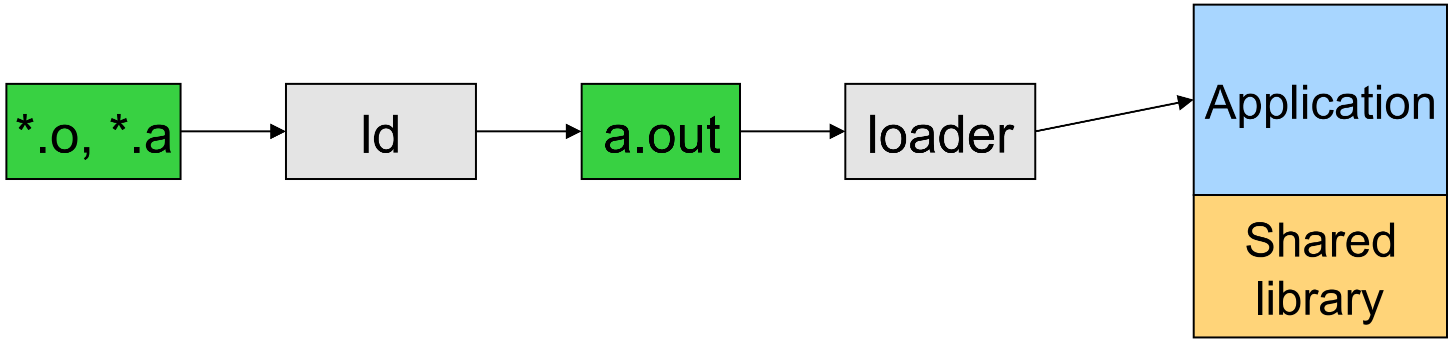
Pipeline of Creating An Executable File



- 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 elf, ld, and nm
  - Read the document of ELF

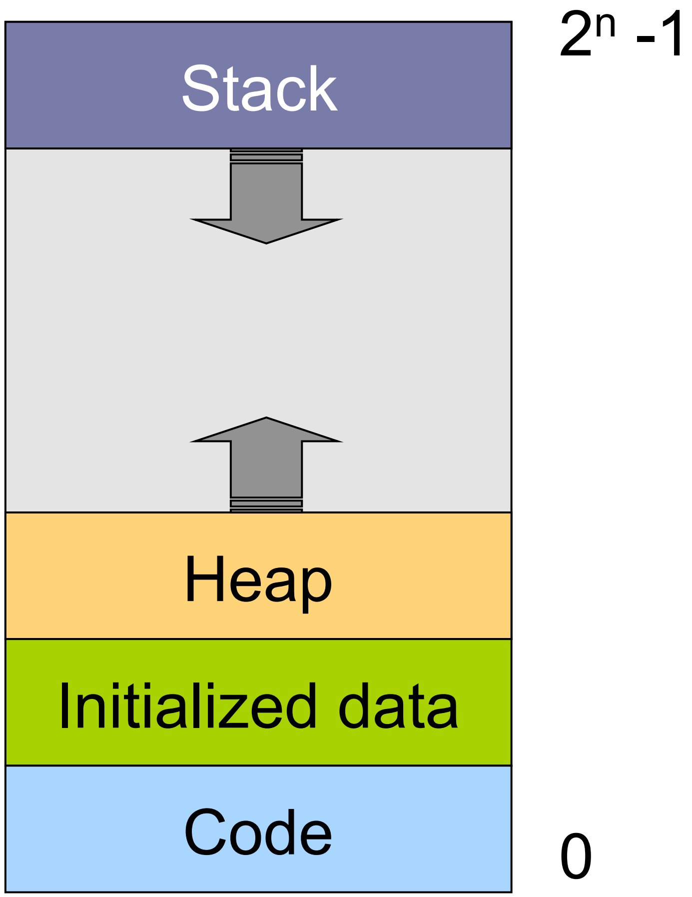
Execution (Run An Application)

- 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
  - E.g., on Linux, “man ld-linux”

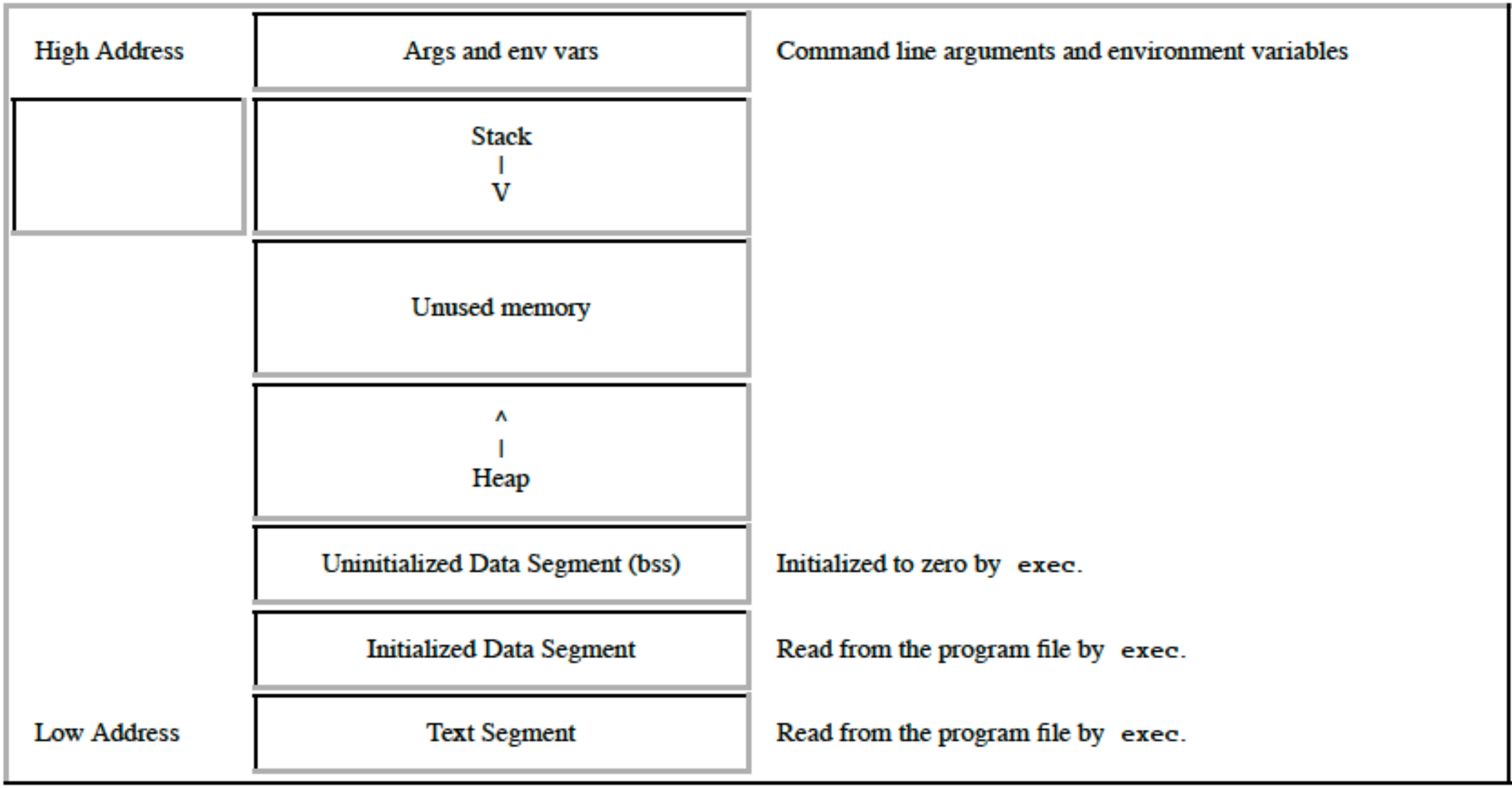


What's An Application?

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



In slightly more detail…



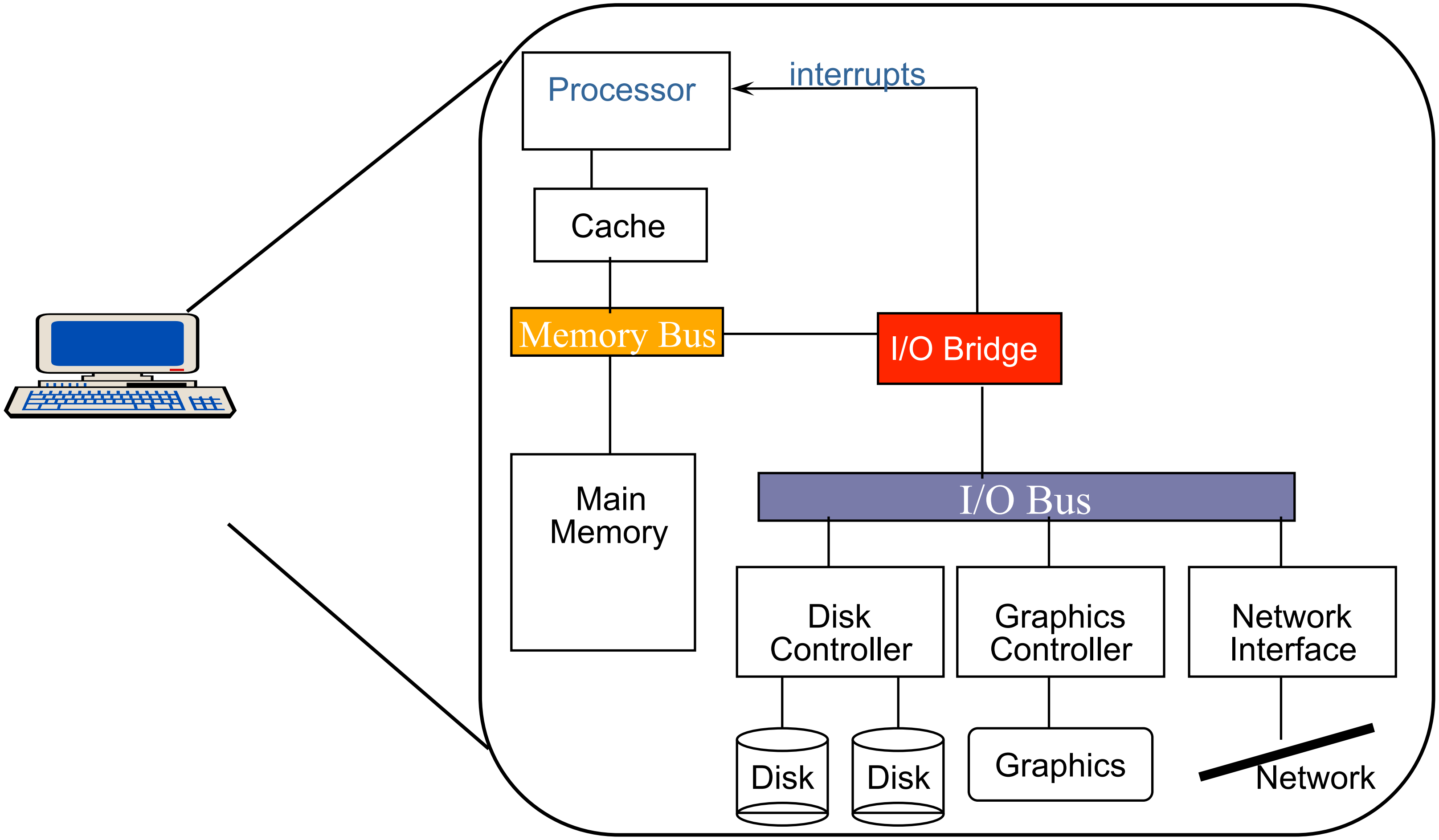
High Address	Args and env vars	Command line arguments and environment variables
	Stack | V	
	Unused memory	
	^ | Heap	
	Uninitialized Data Segment (bss)	Initialized to zero by `exec`.
	Initialized Data Segment	Read from the program file by `exec`.
Low Address	Text Segment	Read from the program file by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

Responsibilities

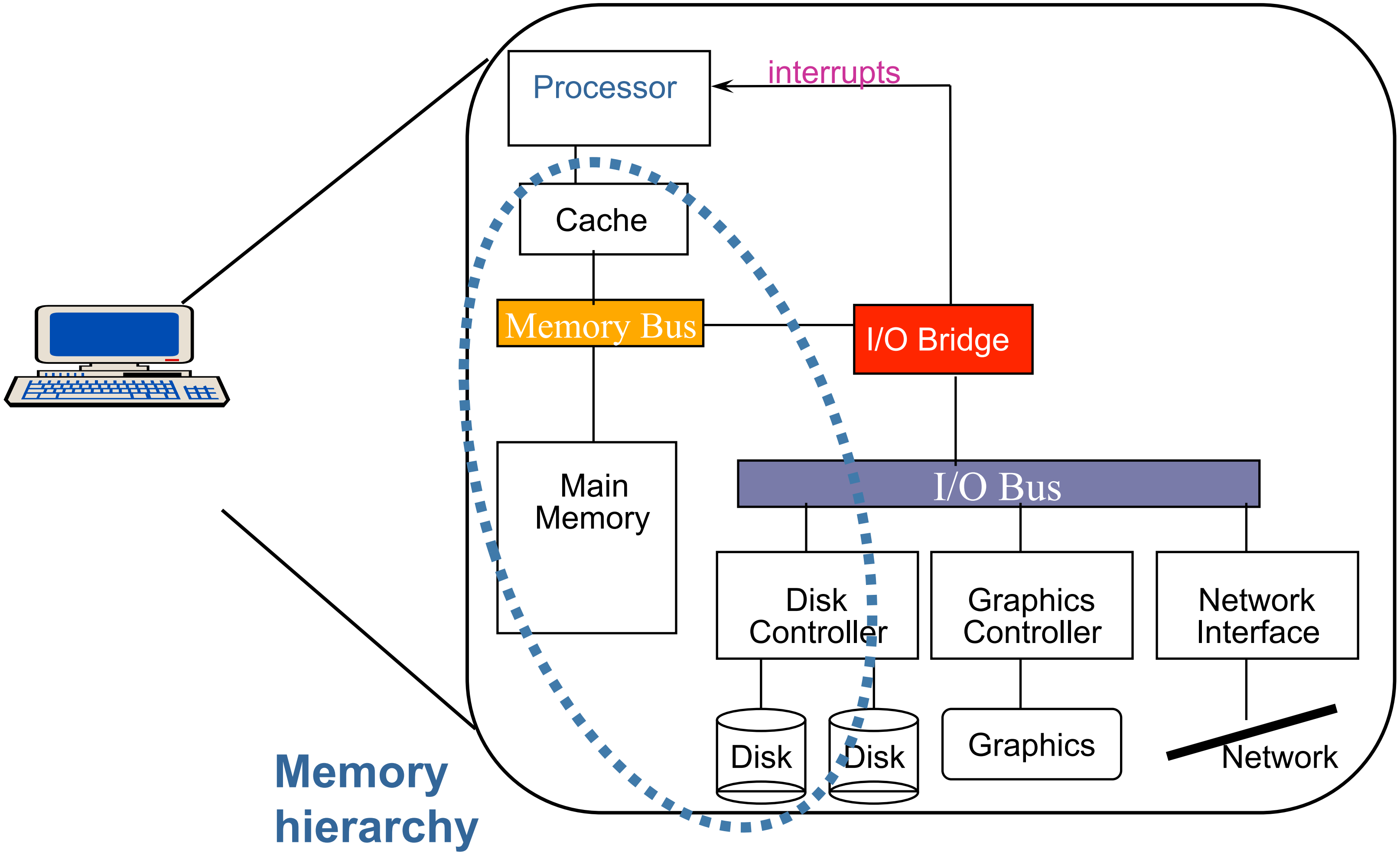
- Stack
  - Layout by compiler
  - Allocate/deallocate by process creation (fork) and termination
  - Names are relative to stack pointer and entirely local
- Heap
  - Linker and loader say the starting address
  - Allocate/deallocate by library calls such as malloc() and free()
  - Application program 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

A view from hardware

System Organization

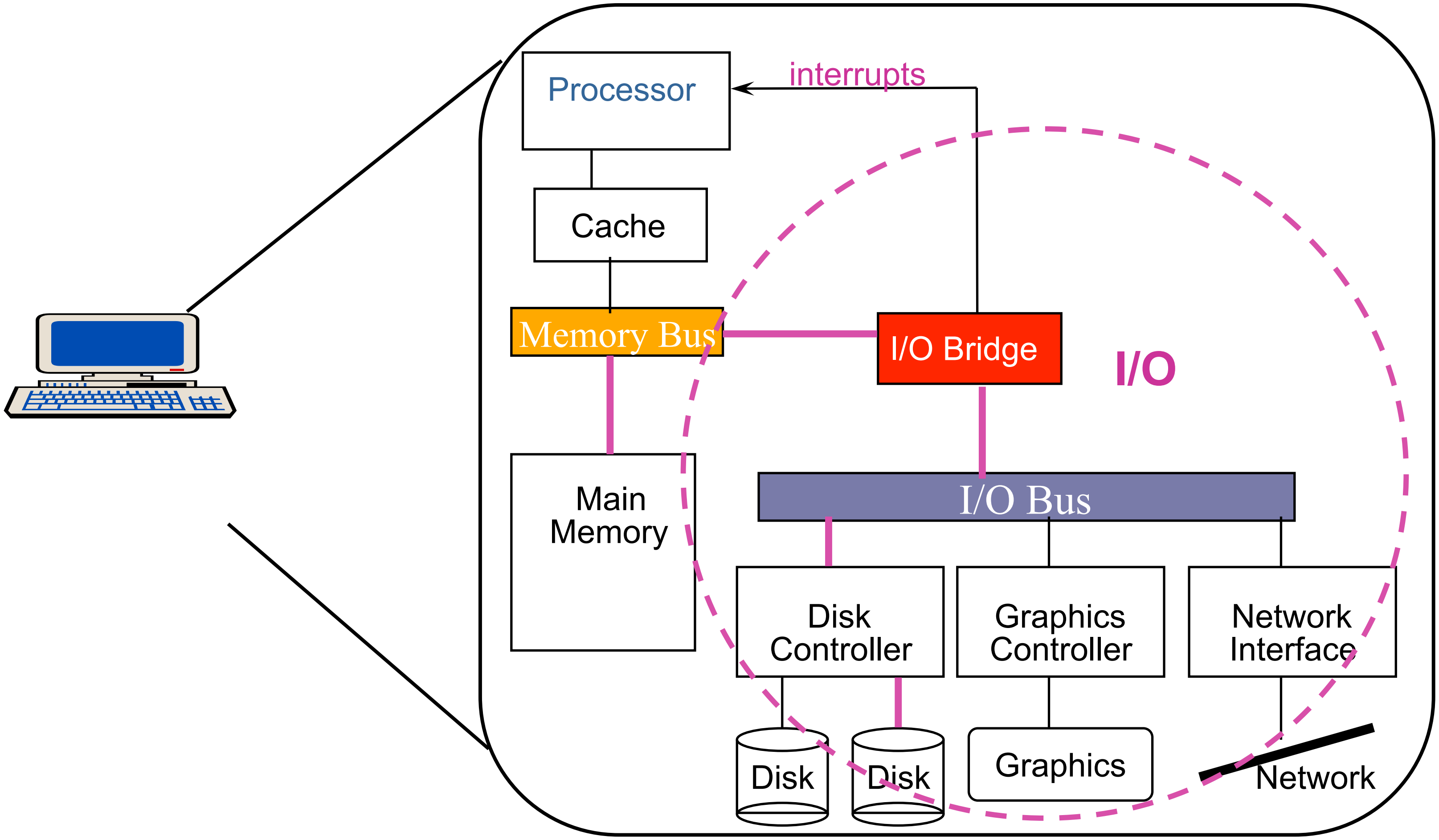


System Organization

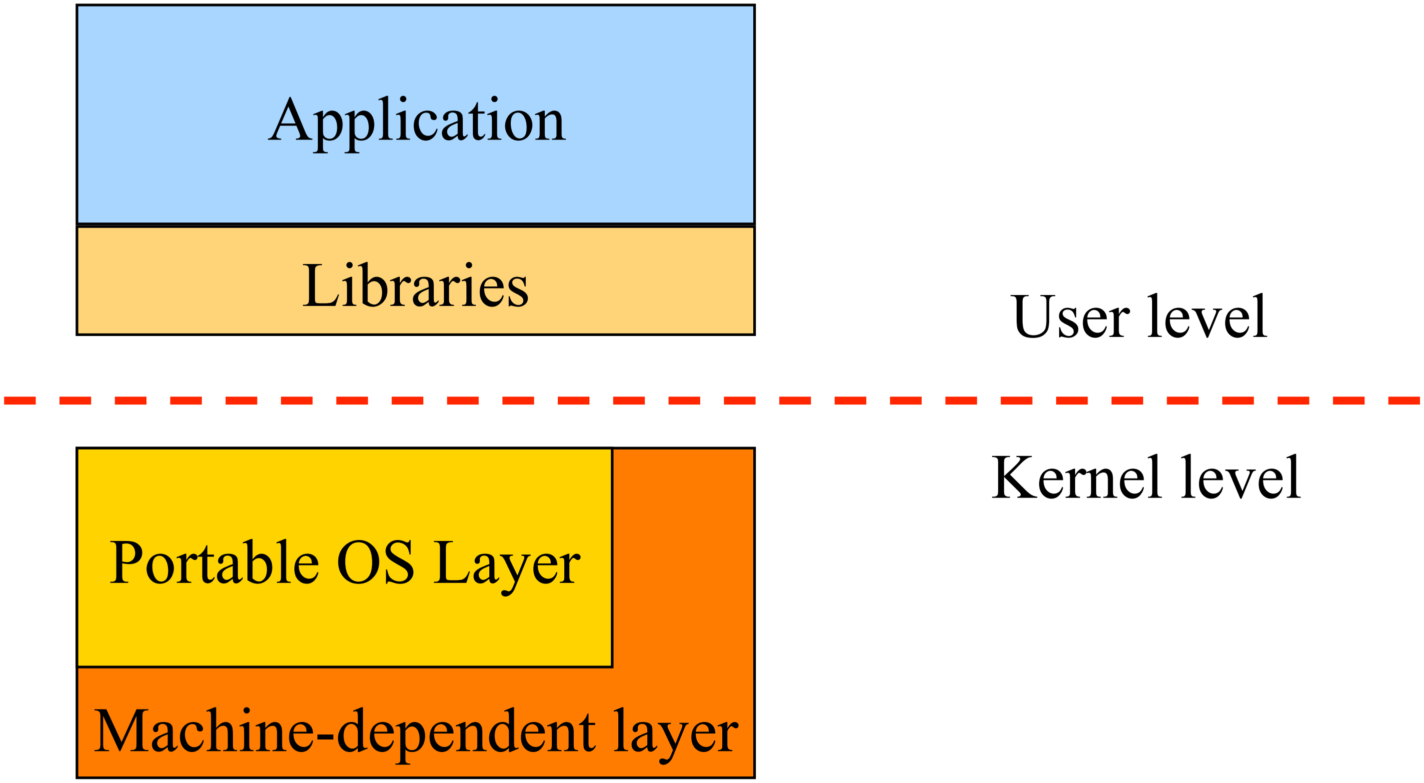


Memory hierarchy

System Organization



Typical Unix OS Structure



Application

Libraries

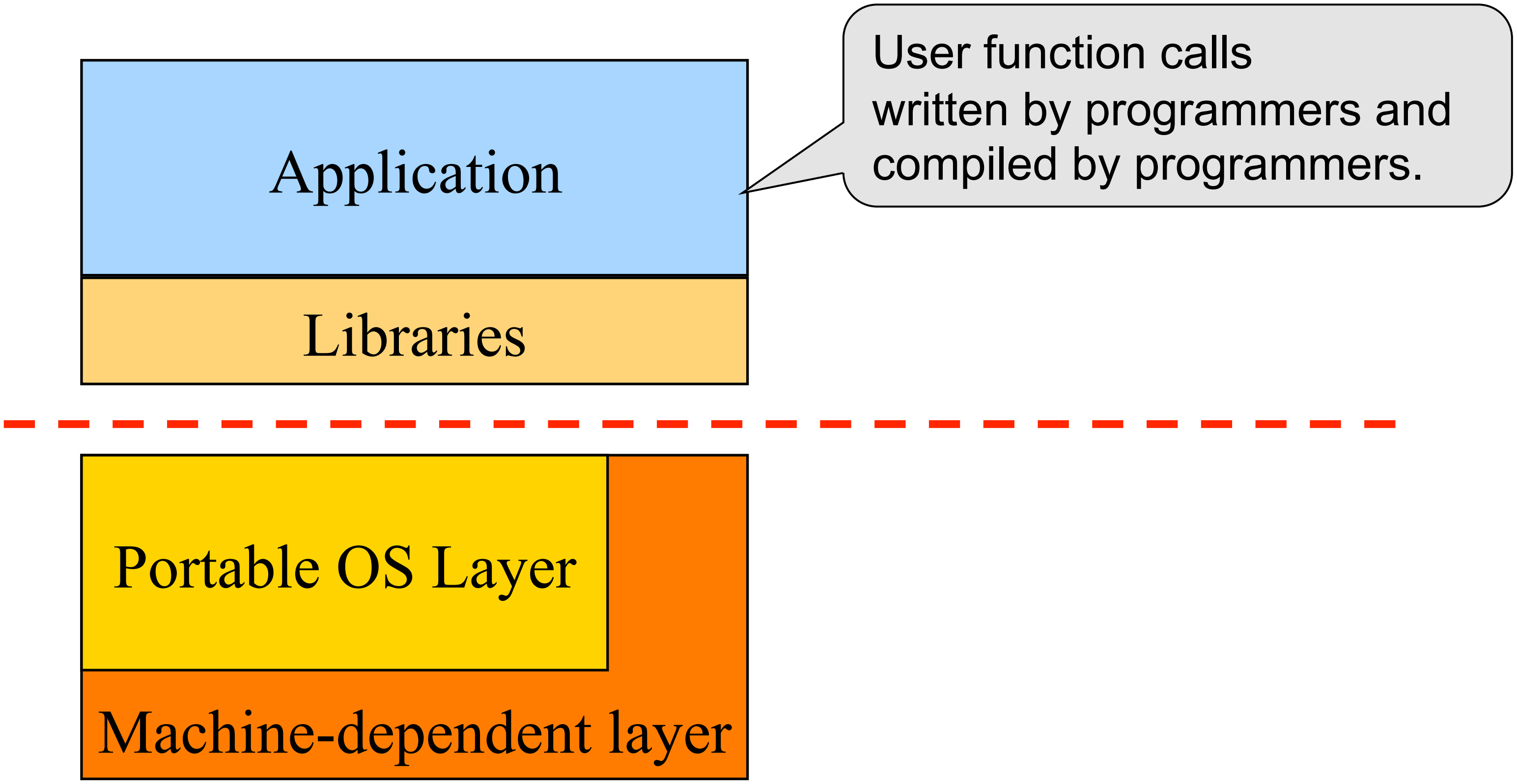
User level

Kernel level

Portable OS Layer

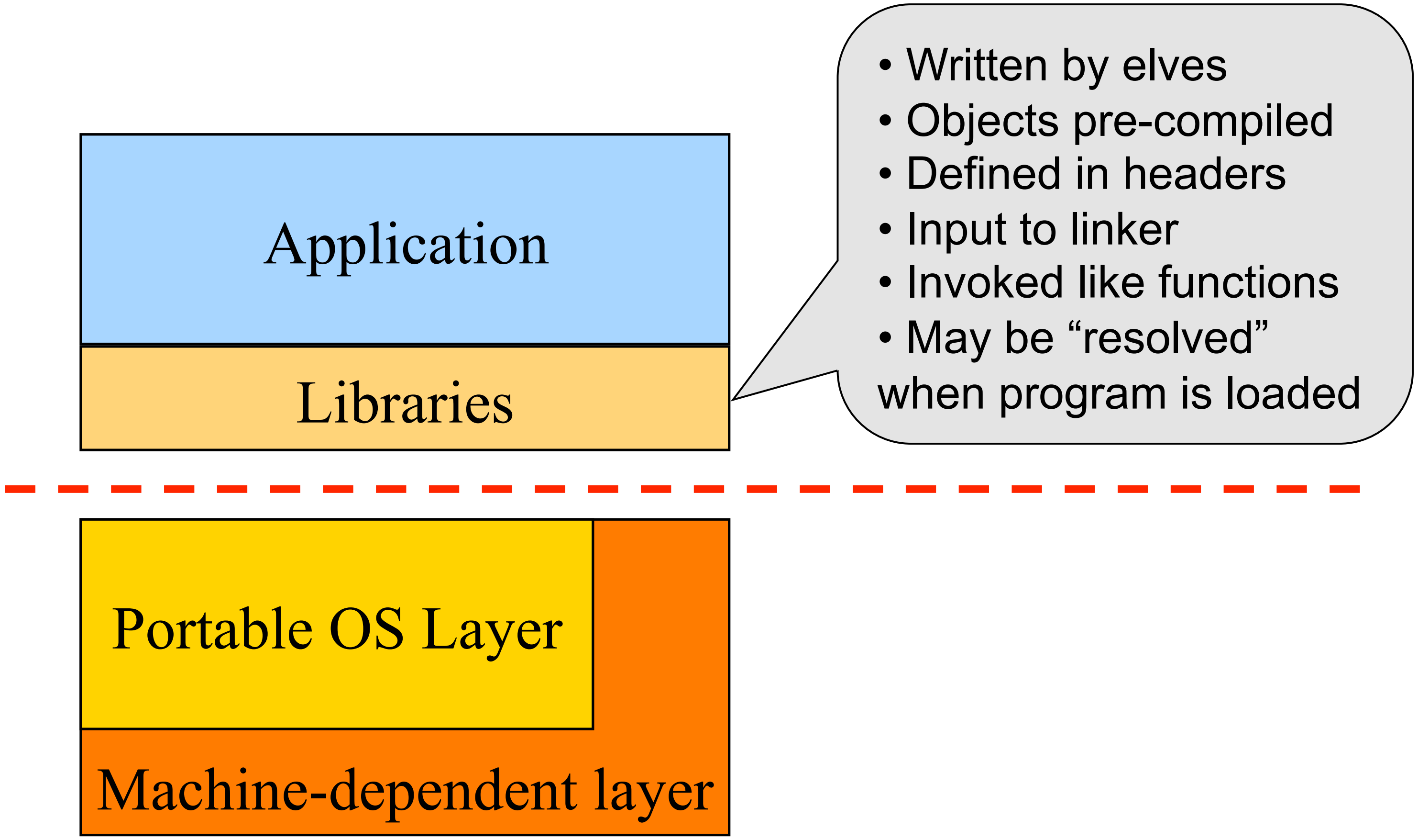
Machine-dependent layer

Typical Unix OS Structure



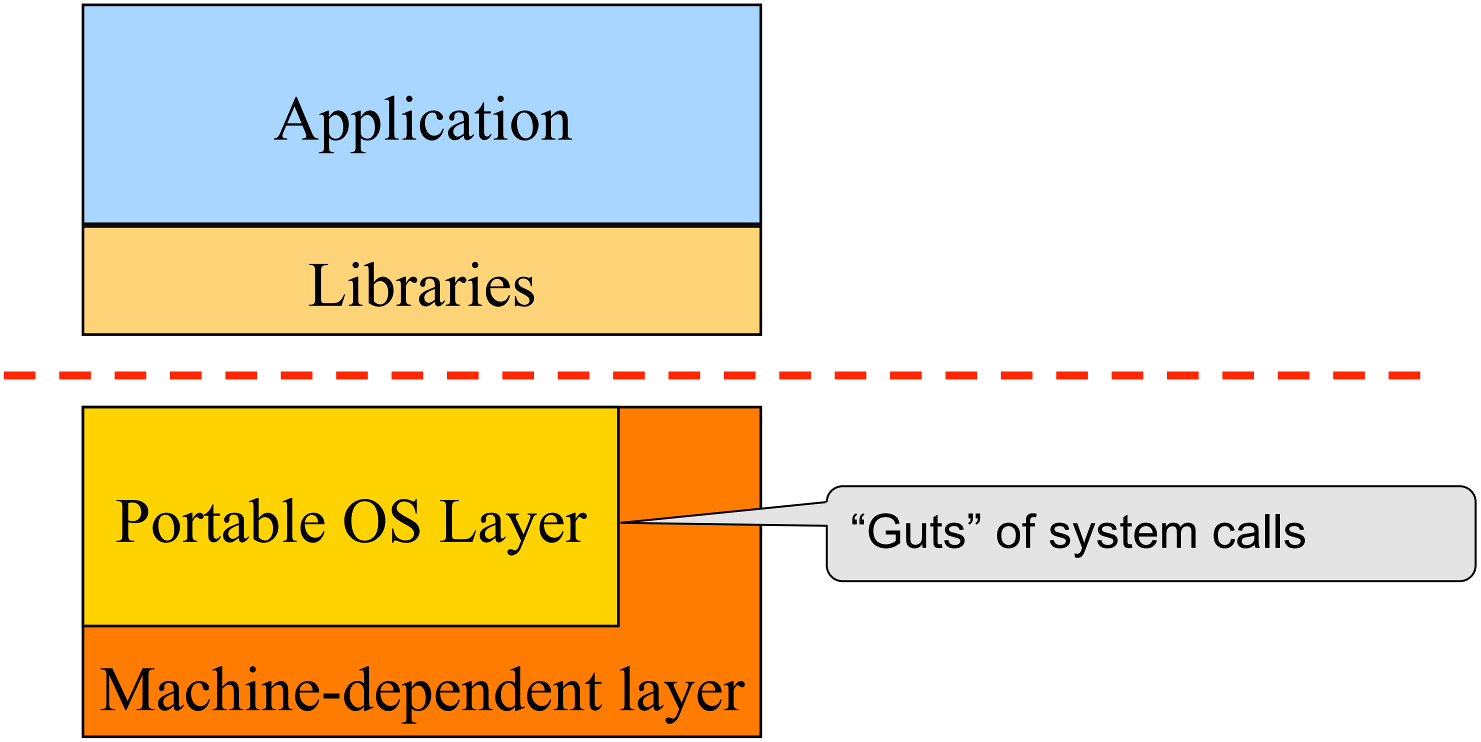
User function calls written by programmers and compiled by programmers.

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

Typical Unix OS Structure


Application

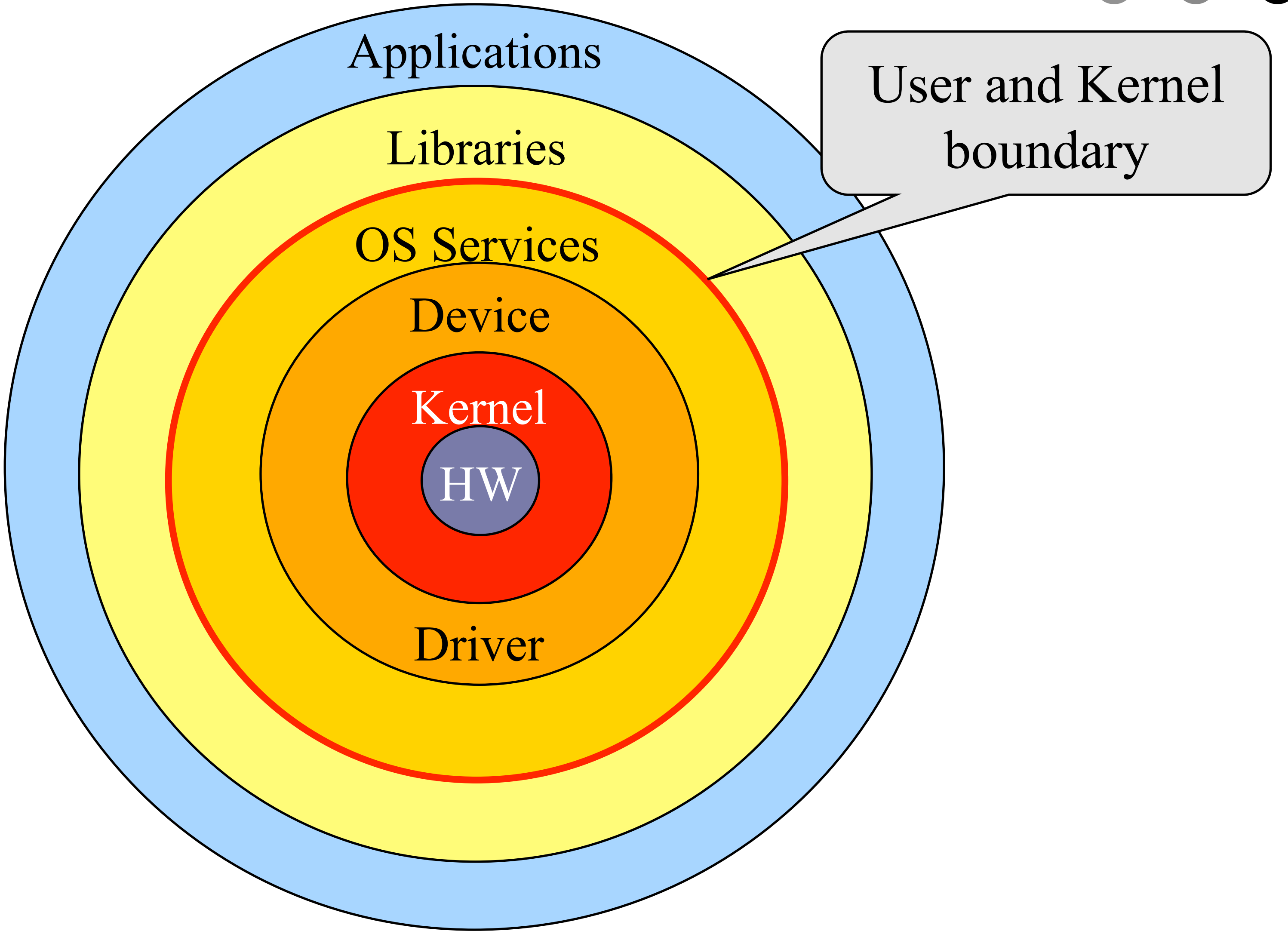
Libraries

Portable OS Layer

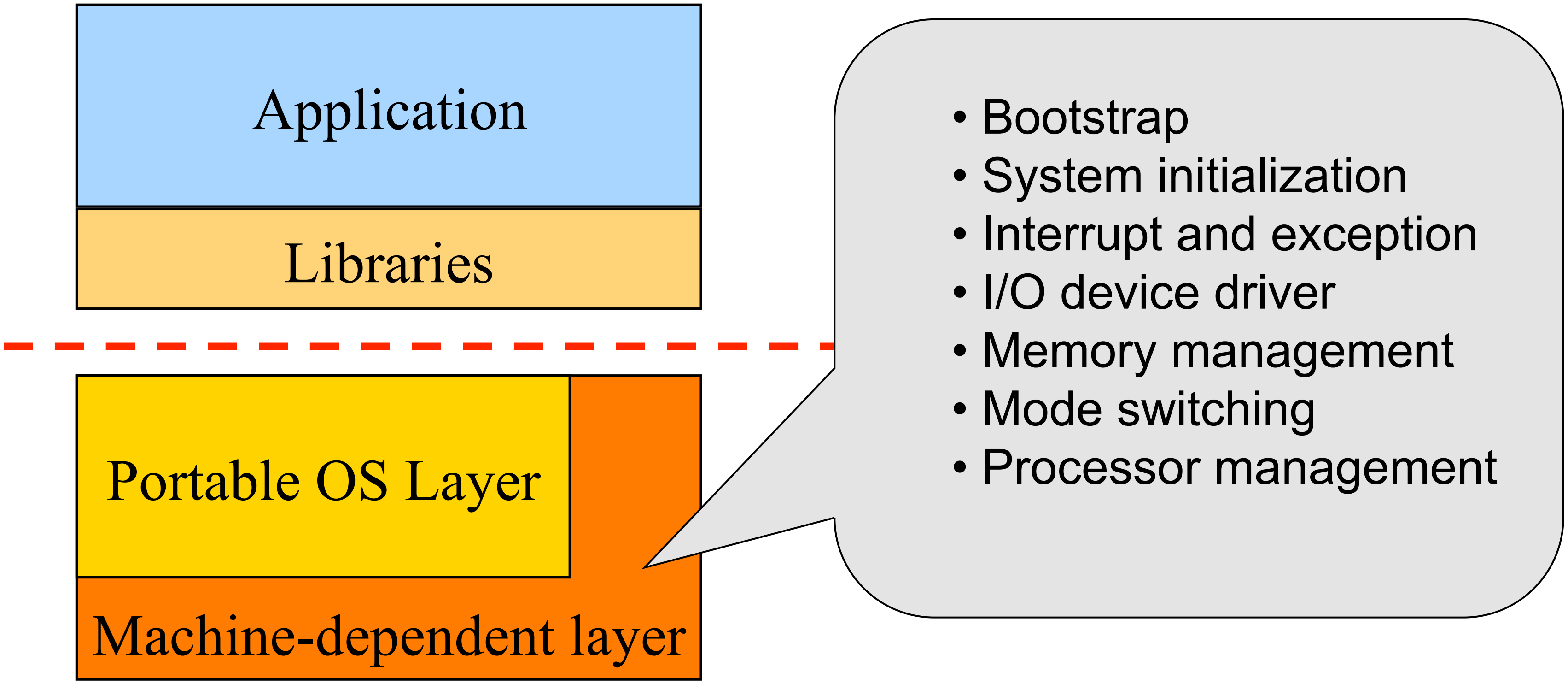
Machine-dependent layer

"Guts" of system calls

Software “Onion” Layers



Typical Unix OS Structure



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

OS's need to:

- Manage and switch between processes
- Manage and protect memory resources
- Interface and provide safe correct access to I/O devices
- …

- How?

OS’s need to: (one selected example)

- Manage and switch between processes
  - What is needed for this?
- Hw/sw interface issues?

Interrupts and Exceptions

- Change in control flow caused by something other than a jump or branch instruction
- Interrupt is external event
  - devices: disk, network, keyboard, etc.
  - clock for timeslicing
  - These are useful events, must do something when they occur.
- Exception is potential problem with program
  - segmentation fault
  - bus error
  - divide by 0
  - Don’t want my bug to crash the entire machine
  - page fault (virtual memory…)

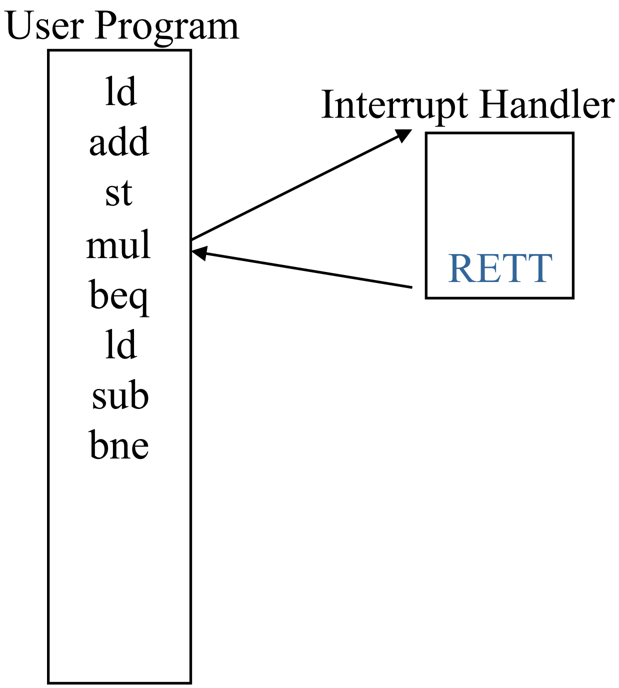
CPU Handling interrupt

- CPU stops current operation*, saves current program counter and other processor state ** needed to continue at interrupted instruction.
- Accessing vector table, in memory, it jumps to address of appropriate interrupt service routine for this event.
- Handler does what needs to be done.
- Restores saved state at interrupted instruction

* At what point in the execution cycle does this make sense?

** Need someplace to save it! Data structures in OS kernel.

OS Handling an Interrupt/Exception

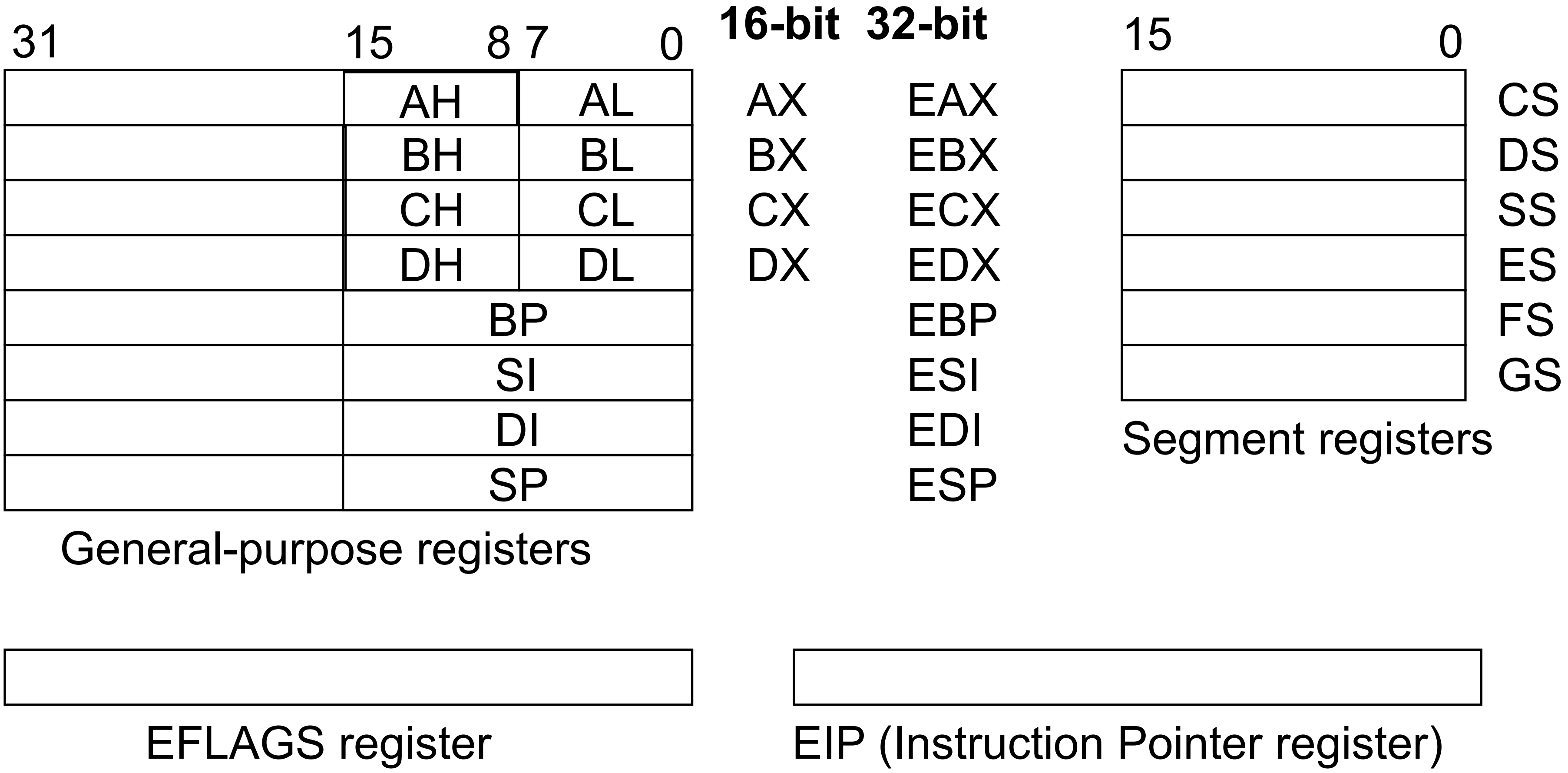


- Invoke specific kernel routine based on type of interrupt
  - interrupt/exception handler
- Must determine what caused interrupt
  - could use software to examine each device
  - PC = interrupt_handler
- Vectored Interrupts
  - PC = interrupt_table[i]
  - kernel initializes table at boot time
- Clear the interrupt
- May return from interrupt (RETT) to different process (e.g, context switch)

A "Typical" RISC Processor

- 32-bit fixed format instruction
- 32 (32,64)-bit GPR (general purpose registers)
- Status registers (condition codes)
- Load/Store Architecture
  - Only accesses to memory are with `load`/`store` instructions
  - All other operations use registers
  - addressing mode: base register + 16-bit offset
- Not Intel x86 architecture!

x86 Architecture Registers



31	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

General-purpose registers

15 0	
	CS
	DS
	SS
	ES
	FS
	GS

Segment registers

EFLAGS register

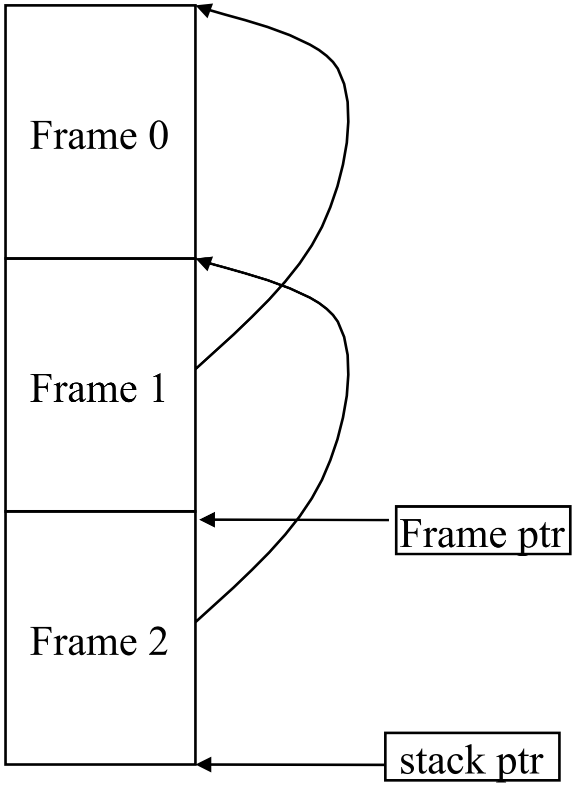
EIP (Instruction Pointer register)

Program Stack

- Well defined register is stack pointer
- Stack is used for
  - passing parameters (function, method, procedure, subroutine)
  - storing local variables

A stack frame (Activation Record)

Return results
Return Address
Old frame ptr
arg1
arg2
Local variables



First few return results and arguments can be mapped to specific registers (calling conventions)

An Execution Context

- The state of the CPU associated with a thread of control (process)
  - general purpose registers (integer and floating point)
  - status registers (e.g., condition codes)
  - program counter, stack pointer
- Need to be able to switch between contexts
  - better utilization of machine (overlap I/O of one process with computation of another)
  - timeslicing: sharing the machine among many processes
  - different modes (Kernel v.s. user)

Context Switches

- Save current execution context
  - Save registers and program counter
  - information about the context (e.g., ready, blocked)
- Restore other context
- Need data structures in kernel to support this
  - process control block
- Why do we context switch?
  - Timeslicing: HW clock tick
  - I/O begin and/or end
- How do we know these events occur?
  - Interrupts...

User / Kernel Modes

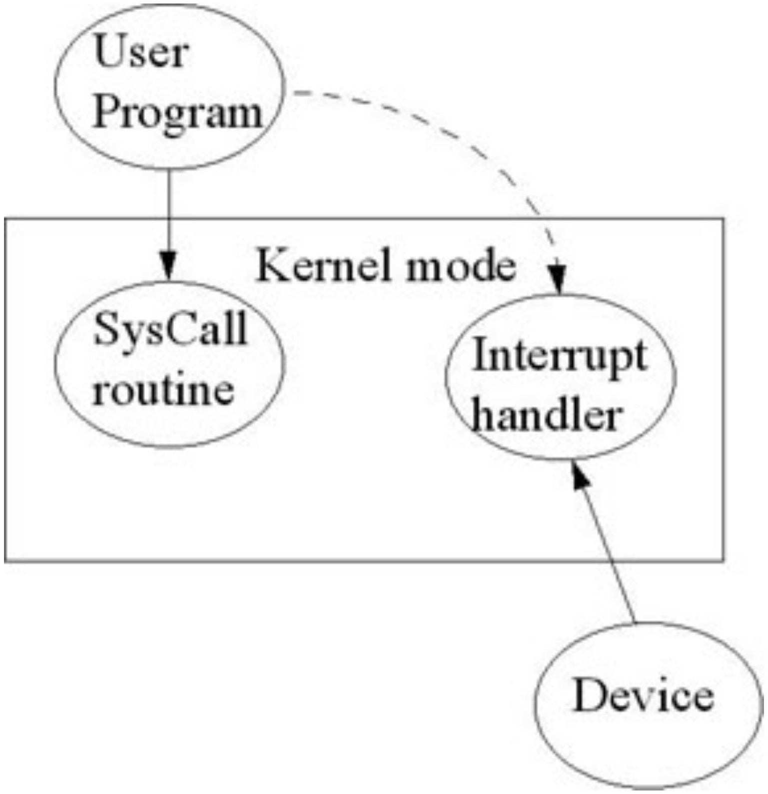
- Hardware support to differentiate between what we'll allow user code to do by itself
  (user mode) and what we'll have the OS do
  (kernel mode).
- Mode indicated by status bit in protected processor register.
- Privileged instructions can only be executed in kernel mode (I/O instructions).

Execution Mode

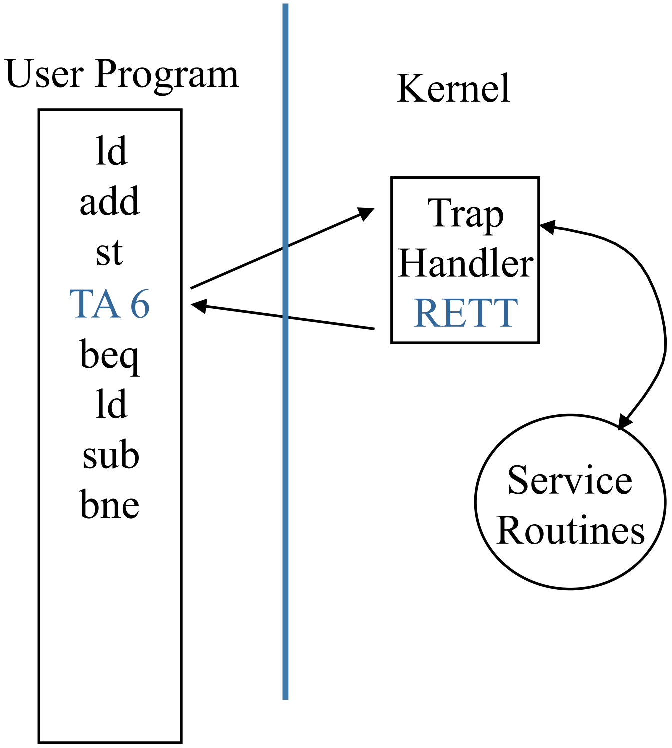
- What if interrupt occurs while in interrupt handler?
  - *Problem*: Could lose information for one interrupt
    clear of interrupt #1, clears both #1 and #2
  - *Solution*: disable interrupts
- Disabling interrupts is a protected operation
  - Only the kernel can execute it
  - user v.s. kernel mode
  - mode bit in CPU status register
- Other protected operations
  - installing interrupt handlers
  - manipulating CPU state (saving/restoring status registers)
- Changing modes
  - interrupts
  - system calls (trap instruction)

Crossing Protection Boundaries

- For a user to do something "privileged", it must invoke an OS procedure providing that service. How?
- System Calls
  - special trap instruction that causes an exception which vectors to a kernel handler
  - parameters indicate which system routine called



A System Call



- Special Instruction to change modes and invoke service
  - read/write I/O device
  - create new process
- Invokes specific kernel routine based on argument
- kernel defined interface
- May return from trap to different process (e.g, context switch)
- RETT, instruction to return to user process

CPU Handles Interrupt (with User Code)

- CPU stops current operation, goes into kernel mode, saves current program counter and other processor state needed to continue at interrupted instruction.
- Accessing vector table, in memory, jump to address of appropriate interrupt service routine for this event.
- Handler does what needs to be done.
- Restores saved state at interrupted instruction. Returns to user mode.

Multiple User Programs

- Sharing system resources requires that we protect programs from other incorrect programs.
  - protect from a bad user program walking all over the memory space of the OS and other user programs (memory protection).
  - protect from runaway user programs never relinquishing the CPU (e.g., infinite loops) (timers).
  - preserving the illusion of non-interruptable instruction sequences (synchronization mechanisms - ability to disable/enable interrupts, special "atomic" instructions).

CPU Handles Interrupt (Multiple Users)

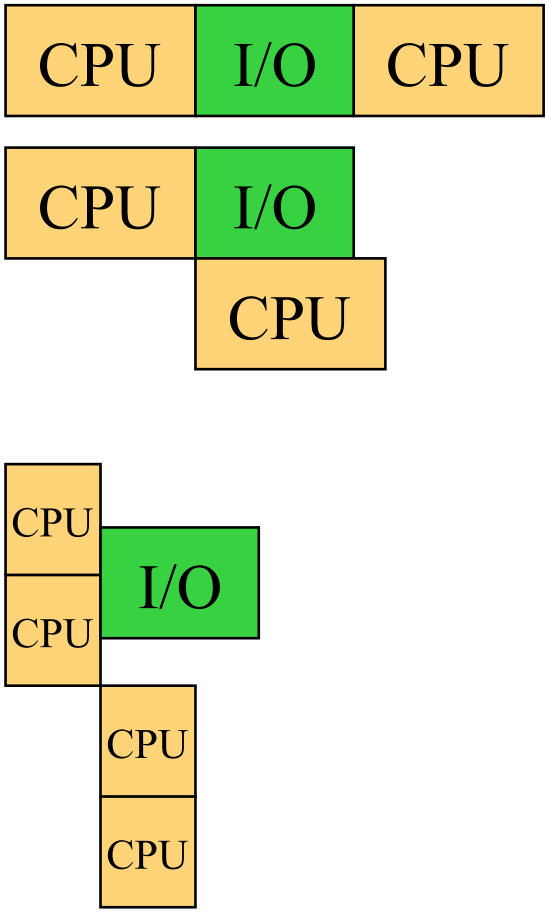
- CPU stops current operation, goes into kernel mode, saves current program counter and other processor state needed to continue at interrupted instruction.
- Accessing vector table, in memory, jump to address of appropriate interrupt service routine for this event.
- Handler does what needs to be done.
- Restores saved state at interrupted instruction (with multiple processes, it is the saved state of the process that the *scheduler* selects to run next). Returns to user mode.

Today

- Overview of OS structure
- Overview of OS components

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

Register: 1x

L1 cache: 2-4x

L2 cache: ~10x

L3 cache: ~50x

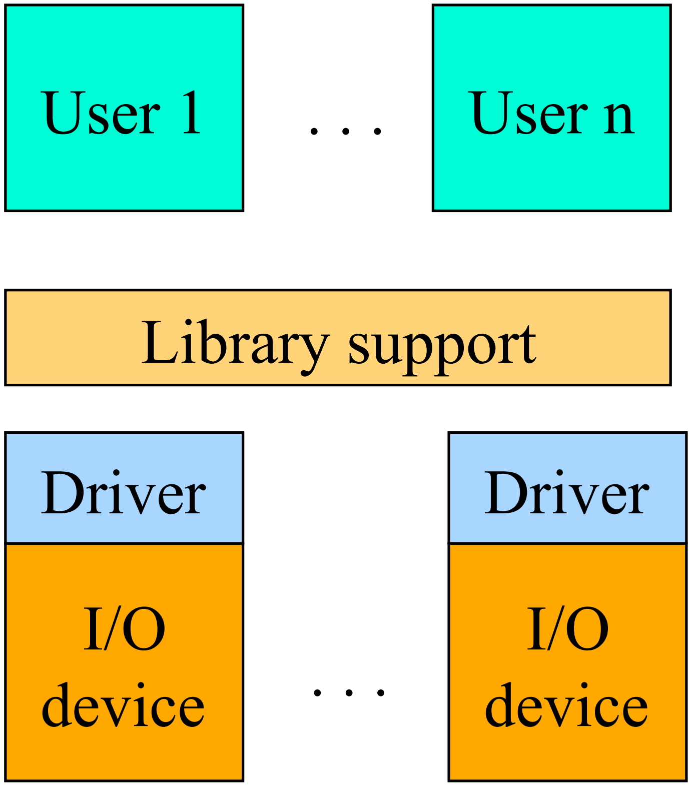
DRAM: ~200-500x

Disks: ~30M x

Archive storage: >1000M x

I/O Device Management

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

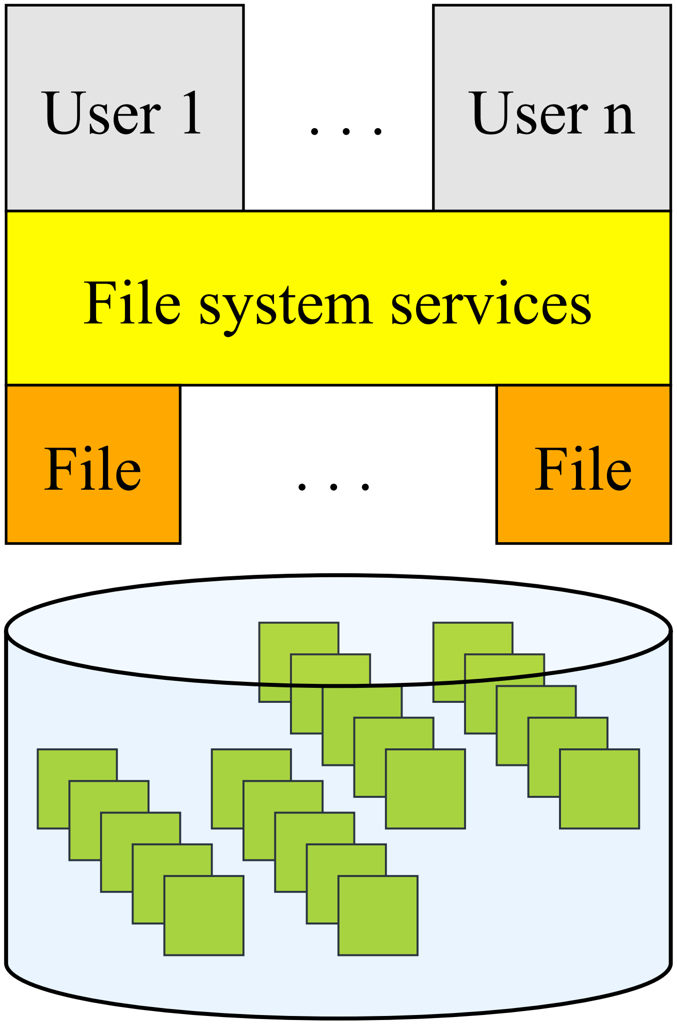

User 1
. . .
User n

Library support

Driver
I/O device
. . .
Driver
I/O device

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

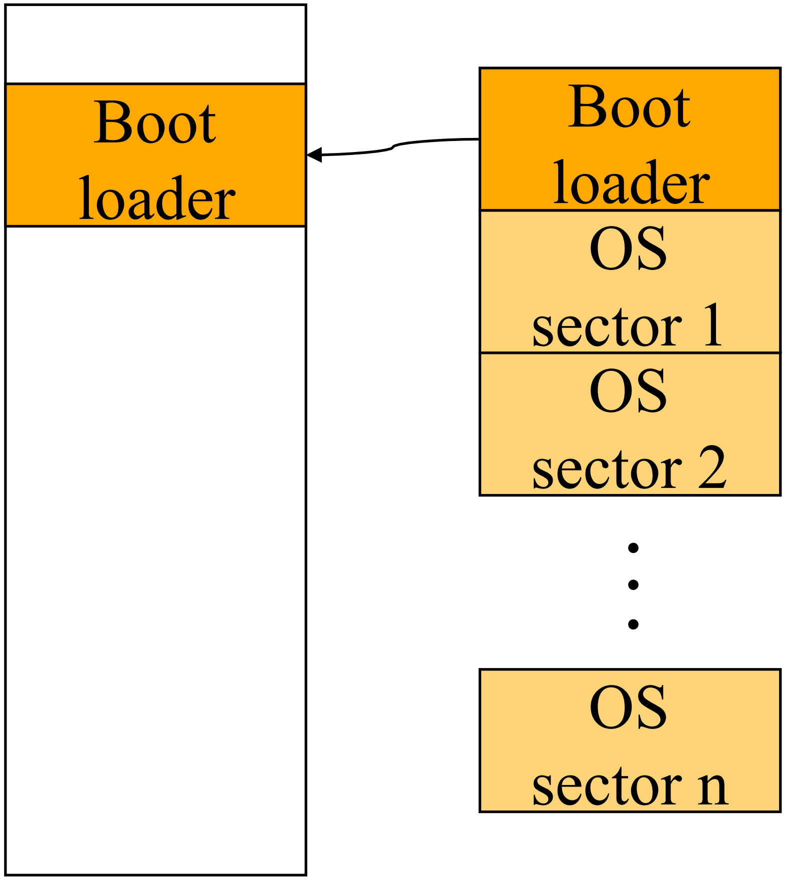


Window Systems

- 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: Why is BIOS in ROM? Can BIOS be on disk?



System Boot

- Power on (processor waits until Power Good Signal)

Maps to FFFFFFF0h= 2^{32}-16

- Processor jumps on a PC (“Intel Inside”) 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

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)

SCSI disks: must often provide their own BIOS

- 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 IRQ
- Detect and configure 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

Summary & Key Ideas