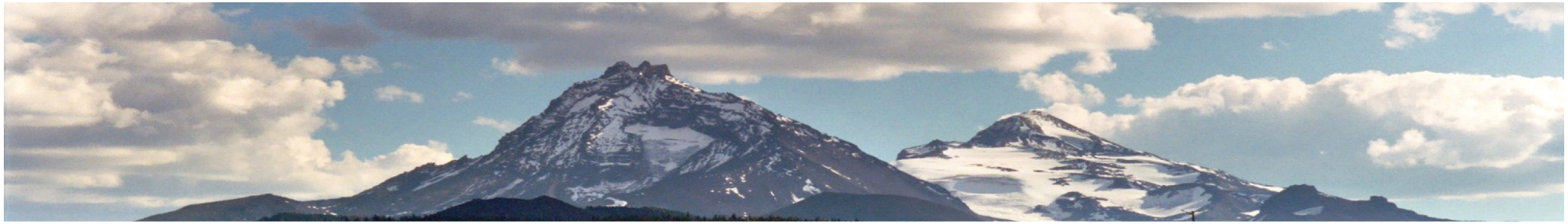


CS 415
Operating Systems
OS Structure and System Calls

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

- Project 1 to be posted tomorrow

Outline

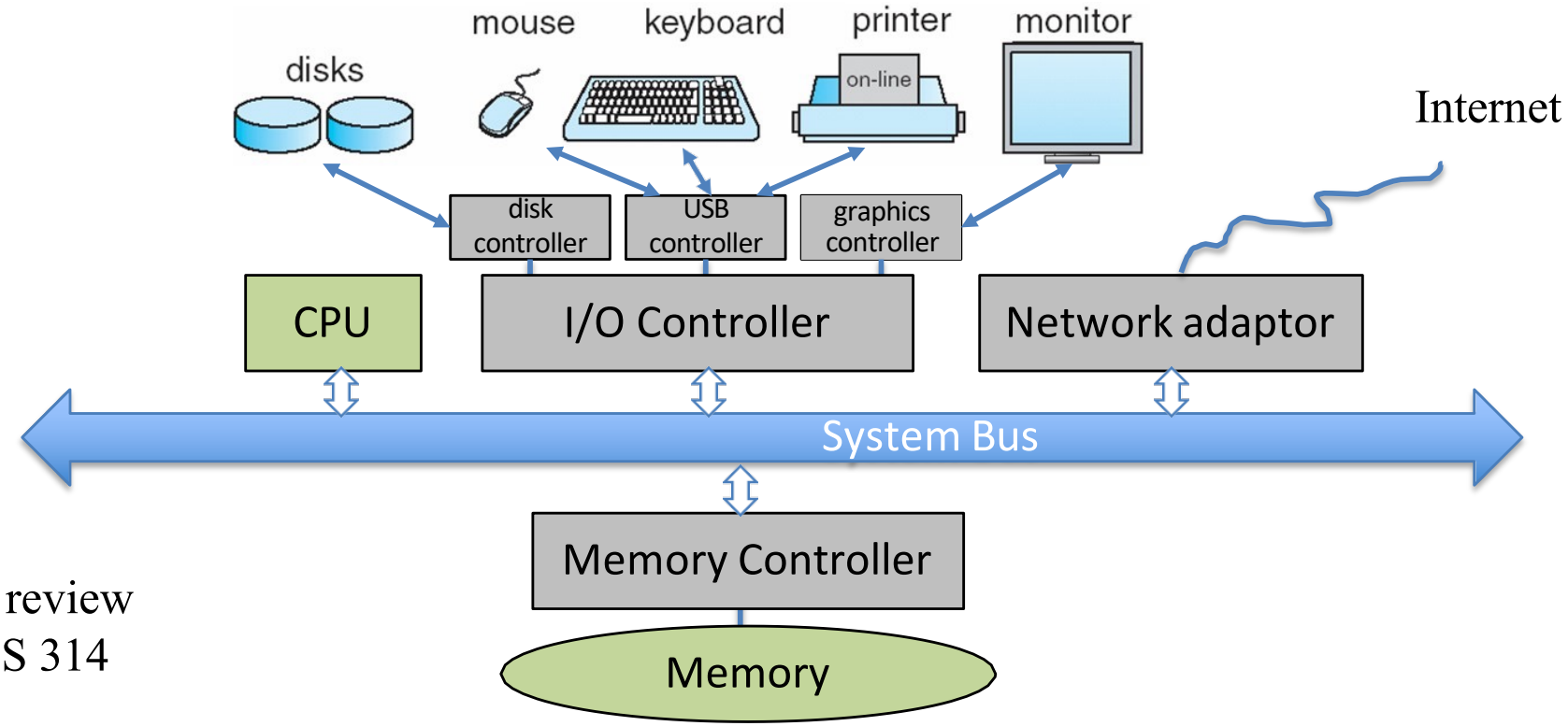
- Hardware and OS relationship
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation

Objectives

- To describe computer system organization
- To describe the services an operating system provides to users, processes, and other systems
- To discuss OS Services and Hardware Support
- To explain how operating systems handle system calls

Canonical System Hardware

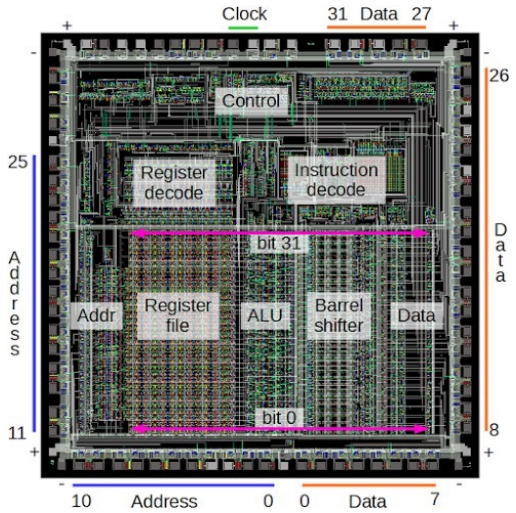
- *CPU*: processor to perform computations
- *Memory*: hold instructions and data
- *I/O devices*: disk, monitor, network, printer, …
- *Bus*: systems interconnection for communication



Quick review of CIS 314

CPU Architecture

- CPUs are semiconductor device with digital logic
  - Program counter (PC) and registers
  - Instruction set architecture (ISA)
    - specifies the instructions a CPU can execute
    - n-bit architecture (e.g., 32-bit, 64-bit) relates to data width
  - Arithmetic logical unit (ALU)
    - Performs computations



ARM1 chip

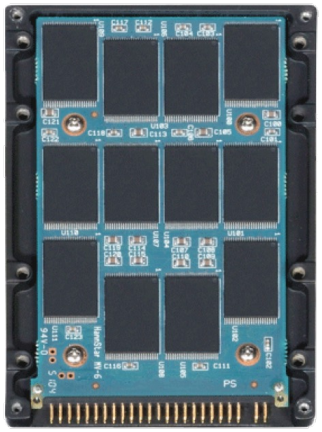
- Registers
  - Defined as part of ISA (how many, what type, length, …)
  - CPU's scratchpads for program execution
  - Fastest memory available in a computer system
  - For storing: instruction, data, address
- Clock to synchronizes constituent circuits
- CPU state: PC + registers values between

Memory

- Random access memory (RAM)
  - Semiconductor DIMMs on PCB
  - Volatile dynamic RAM (DRAM)
- Use “main” memory for instruction and data
- OS manages main memory (allocation to processes, OS, …)
- CPU fetches instruction / data into cache / registers
- Memory controller implements logic for:
  - Reading/Writing to DRAM
  - Refreshing DRAM to maintain contents
  - Address translation for virtual memory
- Cache
  - Fast memory close to CPU
  - Faster than main memory, but more expensive
  - Managed by the hardware (not seen by the OS)

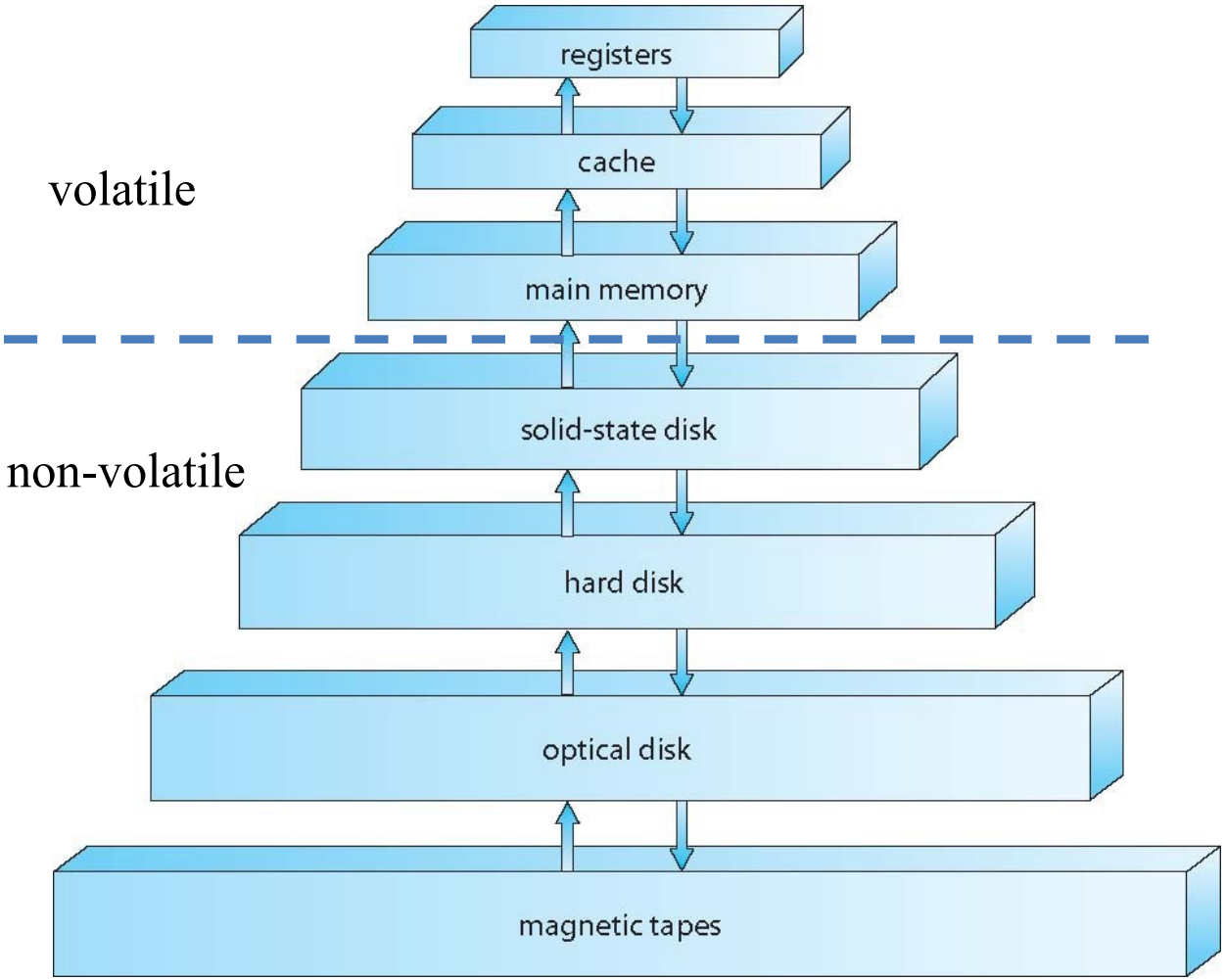
I/O Devices, Hard Disks, and SSD

- Large variety, varying speeds
  - Disk, tape, monitor, mouse, keyboard, NIC, …
  - Serial or parallel interfaces
- Each device has a controller
  - Hides low-level details from OS (hardware interface)
  - Manages data flow between device and CPU/memory
- Hard disks are secondary storage devices
  - Mechanically operated with sequential access
  - Cheap (bytes / $), but slow
  - Orders of magnitude slower than main memory



- Solid state devices (SSD) are increasingly common

Storage+Memory Hierarchy

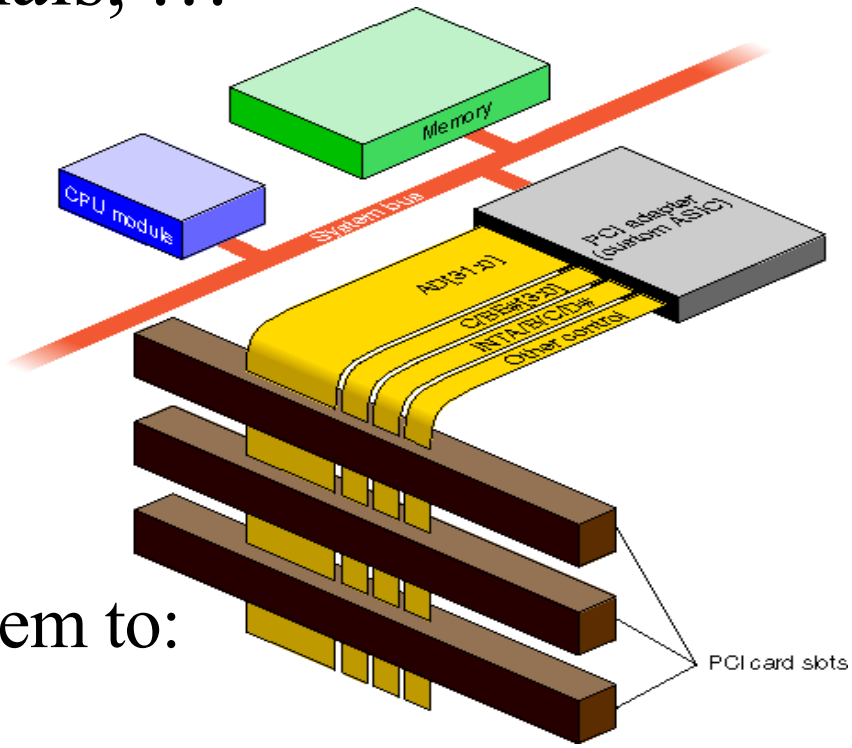


Fast Expensive Small

Slow Cheap Large

Interconnects

- A bus is hardware interconnect for supporting the exchange of data, control, signals, …
  - Physical specification
  - Defined by a protocol
  - Data and control arbitration
- System bus for CPU connection to memory
- PCI bus for devices
  - Connects CPU-memory subsystem to:
    - fast devices
    - expansion bus that connects slow devices
- Other device "bus" types
  - SCSI, IDE, USB, …

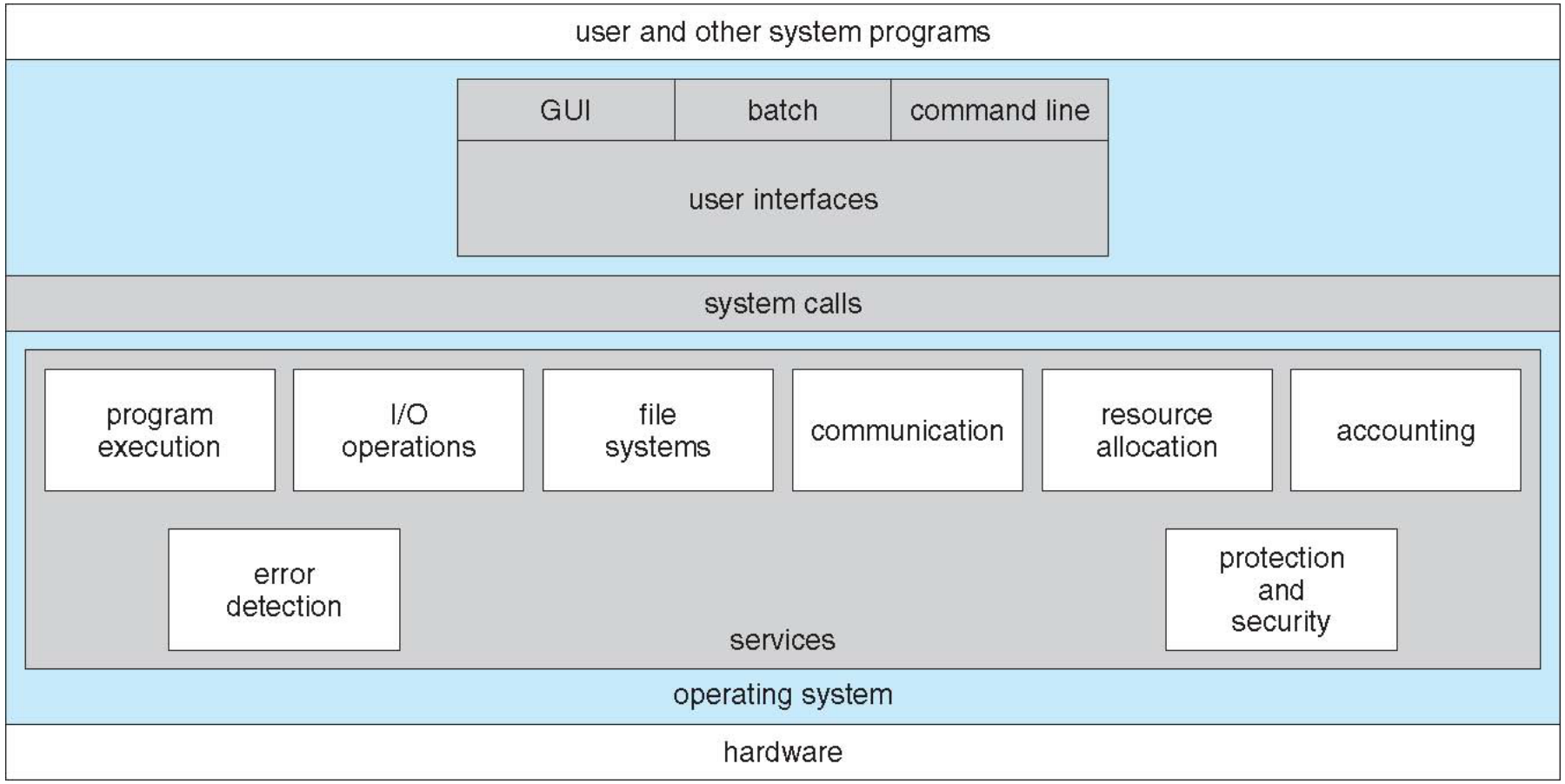


https://en.wikipedia.org/wiki/Peripheral_Component_Interconnect

Operating System Services

- OS provides an environment for program execution
- Some OS services are helpful to the user:
  - User interface (UI)
  - Program execution (load, run, terminate)
  - I/O operations (file or device)
  - File-system manipulation (file, directories, permission)
  - Communications (same computer or several over network)
  - Error detection (hardware and software)
- Some OS services are for efficient system operations
  - Resource allocation (across multiple, concurrent jobs)
  - Accounting (how much and what kind of resources)
  - Protection and security (resources, users)

View of Operating System Services



OS Services and Hardware Support

- Protection
  - Kernel/User mode and protected instructions
  - Base / Limit registers
- Scheduling
  - Timer
- System Calls
  - Trap Instructions
- Efficient I/O
  - Interrupts
  - Memory-mapping
- Synchronization
  - Atomic Instructions
- Virtual memory
  - Translation lookaside buffer (TLB)

Evolution of computer architecture and hardware has been hand-in-hand with OS design and development !!!

Kernel / User Mode

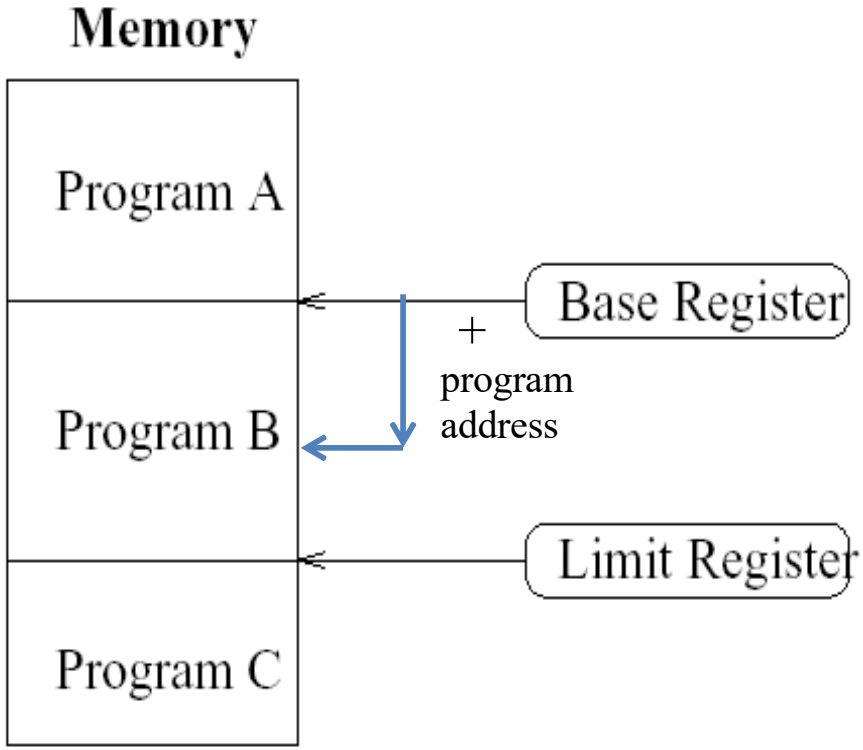
- A modern CPU has at least two ***execution modes***
  - Indicated by status bit in a protected CPU register
  - OS kernel runs in *privileged* mode
    - also called *kernel* or *supervisor* mode
  - Applications run in *normal* or *user* mode
    - not privileged mode
- Certain operations need to run in privileged mode
  - Must switch the processor to privileged mode
    - allows processor to execute certain privileged instructions
  - Example: interfacing with devices
  - Example: error handling
- OS can switch the processor back to user mode
  - CPU can then only access user process address space
  - Can not execute privileged instructions in user mode

Protected Instructions

- OS runs a user program as a ***process***
- A processes executes code of a compiled program
  - A process is a program in execution
- Instructions that require privilege, such as:
  - Direct access to I/O
  - Modify page table pointers or the TLB
  - Enable and disable interrupts
  - Halt the machine
  - Access sensitive registers or perform sensitive operations
- Only allow these instructions in privilege modes
  - Otherwise, random user programs could crash the machine
- User processes do not run in privilege mode
- CPU modes are a form of execution protection

Memory Protection

- User processes must protect their memory from each other
- OS must be protected from user processes
- Memory referencing hardware to protect memory regions
- A simple ***memory address translation*** mechanism
  - Base register contains an address offset in physical memory
  - Limit register is the maximum address that can be referenced
  - Loaded by OS before execution
- CPU checks each memory reference to make sure it is in the proper range
- Used for both for instruction and data addresses
- Ensures references can not access other memory regions and corrupt memory



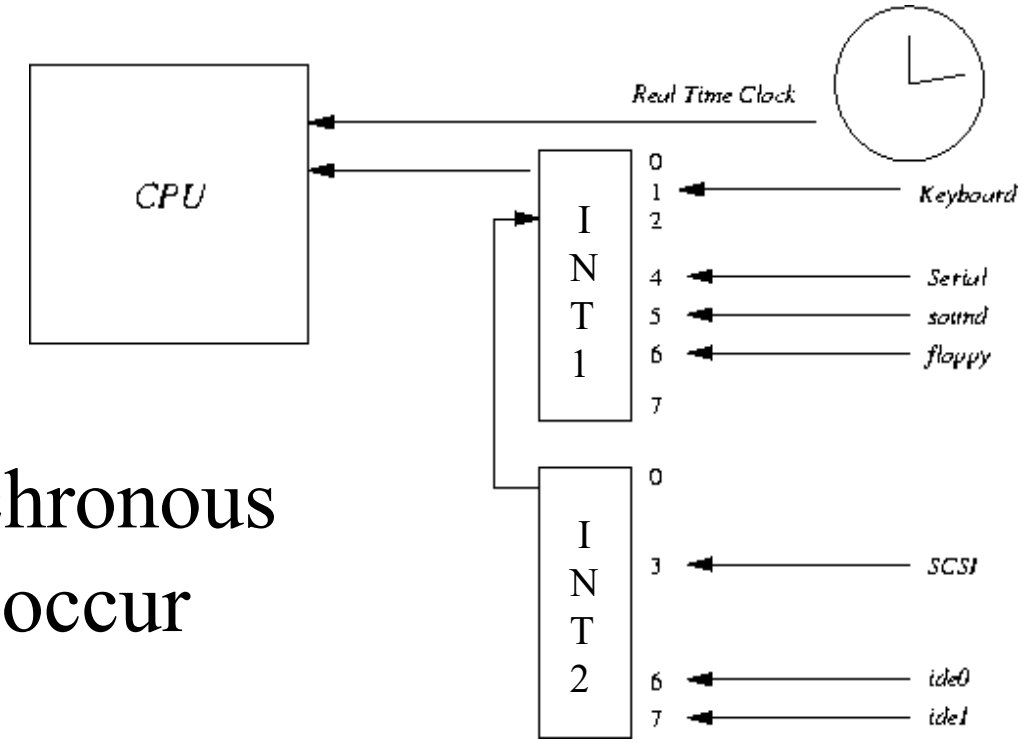
Interrupts

- Interrupts make events that occur in the system visible for the OS to observe
- OS polls for events (*polling*)
  - "Are we there yet?" "no!" (repeat…)
  - Inefficient use of resources
  - Why?

- OS is interrupted when events occur
  - "We're there!" signal
  - I/O device has own logic (processor)
  - When device operation finishes, it pulls on interrupt bus
  - CPU "handles" interrupt

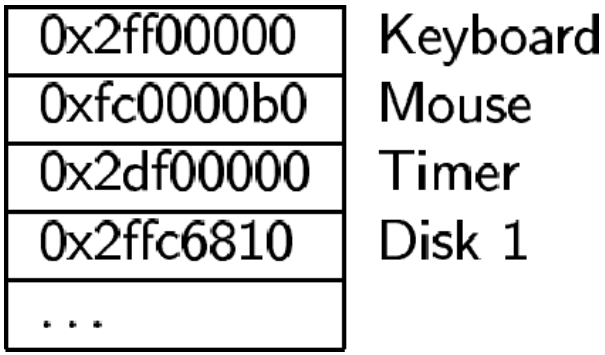
Hardware Interrupts

- Signal from a hardware device
  - Physical interrupt inputs to the processor
  - Multiple hardware components can generate interrupts
- Examples
  - Timer (clock)
  - Keyboard, mouse
  - End of DMA transfer
  - Memory page fault
- Synchronous and asynchronous
- Multiple interrupts can occur simultaneously
- OS will decide which to service



Interrupt Vectoring

- Interrupts are signals that indicate some need for attention by the OS
- Represent:
  - Normal events to be noticed and acted upon
    - device notification
    - software system call
  - Abnormal conditions to be corrected (divide by 0)
  - Abnormal conditions that cannot be corrected (disk failure)
- *Interrupt vectors* are used to decide what to do with different interrupts
  - Address where interrupt routines live in the OS for different interrupt types



Timer

- OS needs timers for:
  - Time of day, CPU scheduling, …
  - There can be multiple timers for different things
- Based on a hardware clock source (e.g., 1 Ghz)
  - Count-down timer (e.g., 1 Ghz to 1 Khz)
  - Generates an interrupt at 0 (e.g., every 1 msec)
- Use timers to ensure that certain future events occur
- Most importantly, it is a way for OS to regain control
- User programs can set up their own “software” timers

Software Interrupts

- Software interrupts (*traps*)
  - Special interrupt instructions (run in privilege mode)
    - `int` is the interrupt instruction
    - `int 0x80` passes control to interrupt vector `0x80`
  - Exceptions
    - some can be fixed (e.g., page fault)
    - some cannot (e.g., divide by zero)
- All invoke OS, just like a hardware interrupt
  - Trap starts running OS code in supervisor access space
  - This space can not be overwritten by the user program

How a process runs (high level)

- OS keeps track of which process is assigned to which sections in memory along with other details
- For a new process to run, memory is assigned by the OS, which puts the code in that location
  - Switch to user mode
  - Start running at first address of the program
- OS keeps record of every process
  - This is the *process context*
  - Assigned memory, current program counter, …
  - Enough info to restart process where it left off

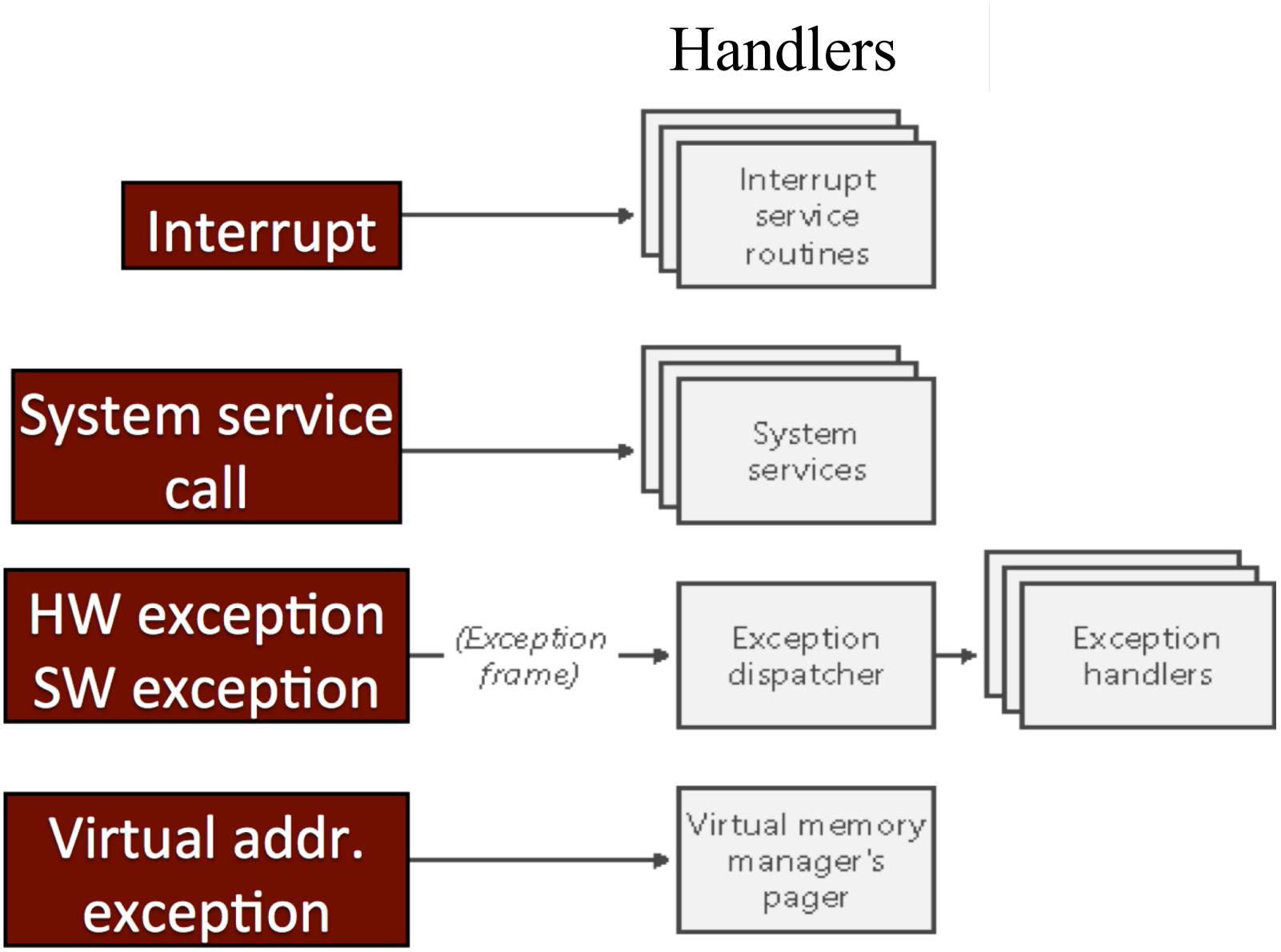
Then along comes an interrupt …

- Eventually a hardware interrupt or a trap (software interrupt) will happen
  - Example: received input from keyboard
- OS records state of running process's context
  - Stored in a *process control block* (PCB)
- Next, OS services the interrupt
  - Example: send something to the printer
- Finally, OS picks process to restart
  - Maybe the one that was running, maybe not
  - Depends on scheduling!
  - Moves back into user mode

Interrupt Handling

- Each interrupt has an *interrupt handler*
- When an *interrupt request* (IRQ) is received
  - If interrupt mask allows interrupt, then …
  - Save state of current process
    - at time of interrupt something else may be running
    - state: Registers (stack pointer), program counter, …
  - Execute handler
  - Return to current process or another process

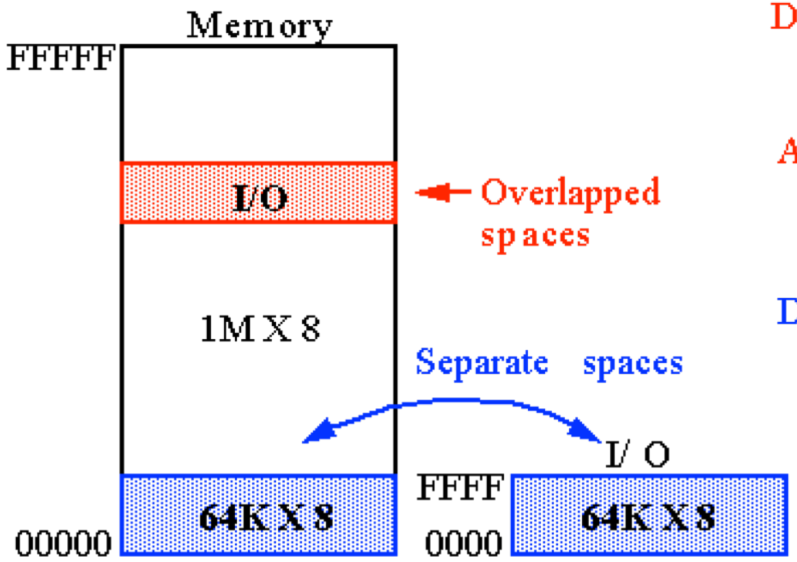
Interrupt (Trap) Handlers



Device Access

- Port I/O (blue)
  - Uses special I/O instructions
  - Port number, device address (not process address)
- Memory-mapped I/O (red)
  - Uses memory instructions (load/store)
    - memory-mapped device registers
  - Does not require special instructions

IORC: I/O Read Control
IOWC: I/O Write Control



Disadvantage:
A portion of the memory space is used for I/O devices.

Advantage:
$\overline{IORC}$ and $\overline{IOWC}$ not required.
Any data transfer instruction.

Disadvantage:
Hardware using M/$\overline{IO}$ and W/$\overline{R}$ needed to develop signals $\overline{IORC}$ and $\overline{IOWC}$.
Requires IN, OUT, INS and OUTS

Direct Memory Access (DMA) - 1/2

Without DMA (using Port IO)

•The CPU is responsible for **every single data movement** between the device and memory.

•Example: Reading from a disk (controller buffer → CPU register→ memory)

- CPU issues a read command to the disk controller.
- When data is ready, CPU executes a loop:
  - Read data from the device's I/O controller buffer to CPU registers.
  - Store it in memory.
  - Repeat until all data is copied.

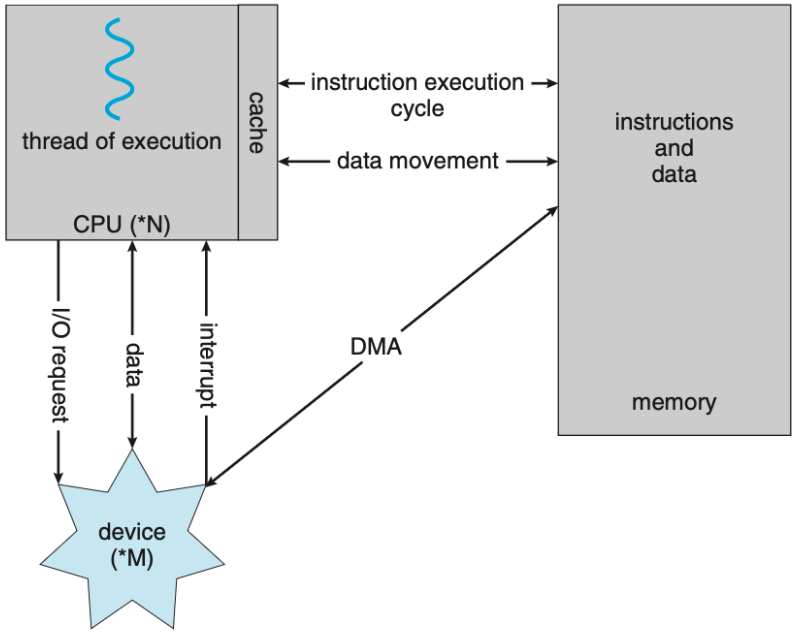
•**Why slow?**

- Each transfer needs a CPU instruction.
- CPU spends cycles waiting for the device (I/O devices are much slower than CPU).
- Memory–I/O–CPU path gets saturated with small, serialized operations.
- CPU can't do useful work during this time — it's busy copying.

Direct Memory Access (DMA) - 2/2

With DMA (using Memory-mapped I/O)

- CPU only tells the DMA controller: "Transfer N bytes from device to this memory buffer."
- DMA controller (or the device itself in modern designs) handles the bulk transfer directly over the system bus.
- Data moves in large blocks without CP intervention.
- CPU is free to run other instructions while the transfer happens in concurrently.



Concurrency

- *Concurrency* is the notion of multiple things happening (being able to happen) "at the same time"
  - Things are happening *concurrently (*they are *concurrent)*
  - Processes (and threads) can execute *concurrently*
- Distinguish between *logical* and *physical* concurrency
- *Logical concurrency* is when there are multiple "threads of execution" that are capable of executing
  - Does not say anything about how they execute on a computer
  - It may be that the "threads of execution" are not actually executing!
- *Physical concurrency* relates to actual execution
  - Multiple "threads of execution" running simultaneously
  - Default physical concurrency (only 1 "thread of execution")
- Concurrency is important for developing systems software

Why do we care about concurrency?

- Concurrency is important for developing complex systems software, like the operating system itself
  - It allows different processes to be doing different things at the same "logical" time
  - Allows for the separation of concerns within the OS software, both functionally and with respect to the processes that implement those functions
- It is exceedingly more difficult to develop a single program that has to manage all of the functional complexity itself
- Instead, we can use the power of processes and threads to realize concurrent execution and gain a valuable software design approach that logically separates operations between them

Synchronization

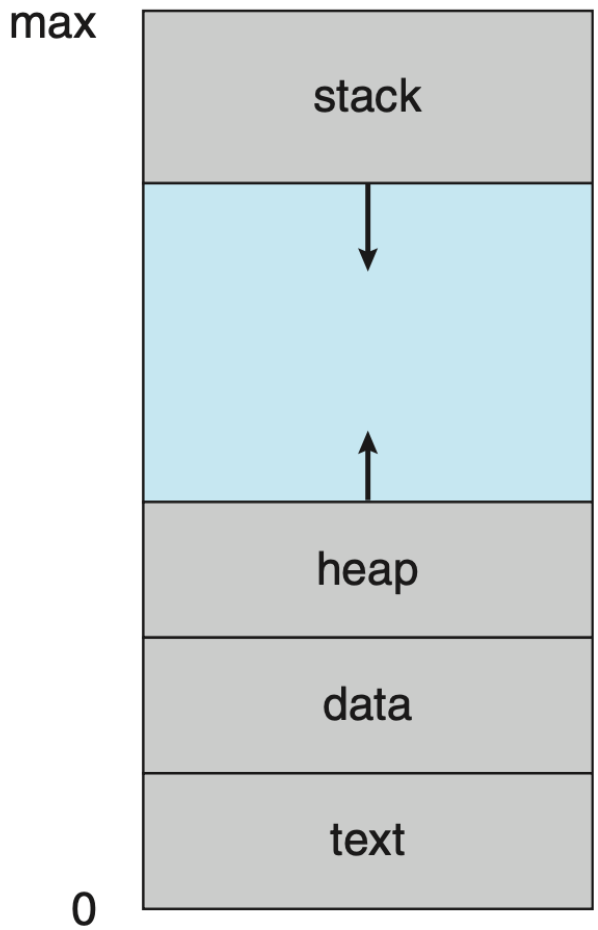
- How can OS synchronize concurrent processes?
  - Multiple threads, processes, interrupts, DMA
- CPU must provide mechanism for atomicity
  - Series of instructions that execute as one or not at all
- One approach:
  - Disable interrupts, perform action, enable interrupts
- Advantages:
  - Requires no hardware support
  - Conceptually simple
- Disadvantages:
  - Could cause starvation

A Modern Synchronization Approach

- Use hardware support for atomic instructions
  - Small set of instructions that cannot be interrupted
- Examples:
  - *Test-and-set* (TST)
    if word contains given value, set to new value
  - *Compare-and-swap* (CAS)
    if word equals value, swap old value with new
  - Intel: LOCK prefix (XCHG, ADD, DEC, …)
- Used to implement locks

Process Address Space

- ***Process address space*** is all locations that are addressable by the process
- Every running program can have its own private address space
- Can restrict use of addresses so as to isolate different area
  - Restrictions enforced by OS
  - *Text segment* is where read only program instructions are stored
  - *Data segment* hold the data for the global variables
  - *Heap segment* allows for dynamic data allocations
  - *Stack segment* holds the stack frames



Virtual Memory

- Provide the illusion of infinite memory
- OS loads *pages* from disk as needed
  - Page: fixed sized block of data
- Many benefits
  - Allows the execution of programs that may not fit entirely in memory
- OS needs to maintain mapping between physical and virtual memory
  - Page tables stored in memory

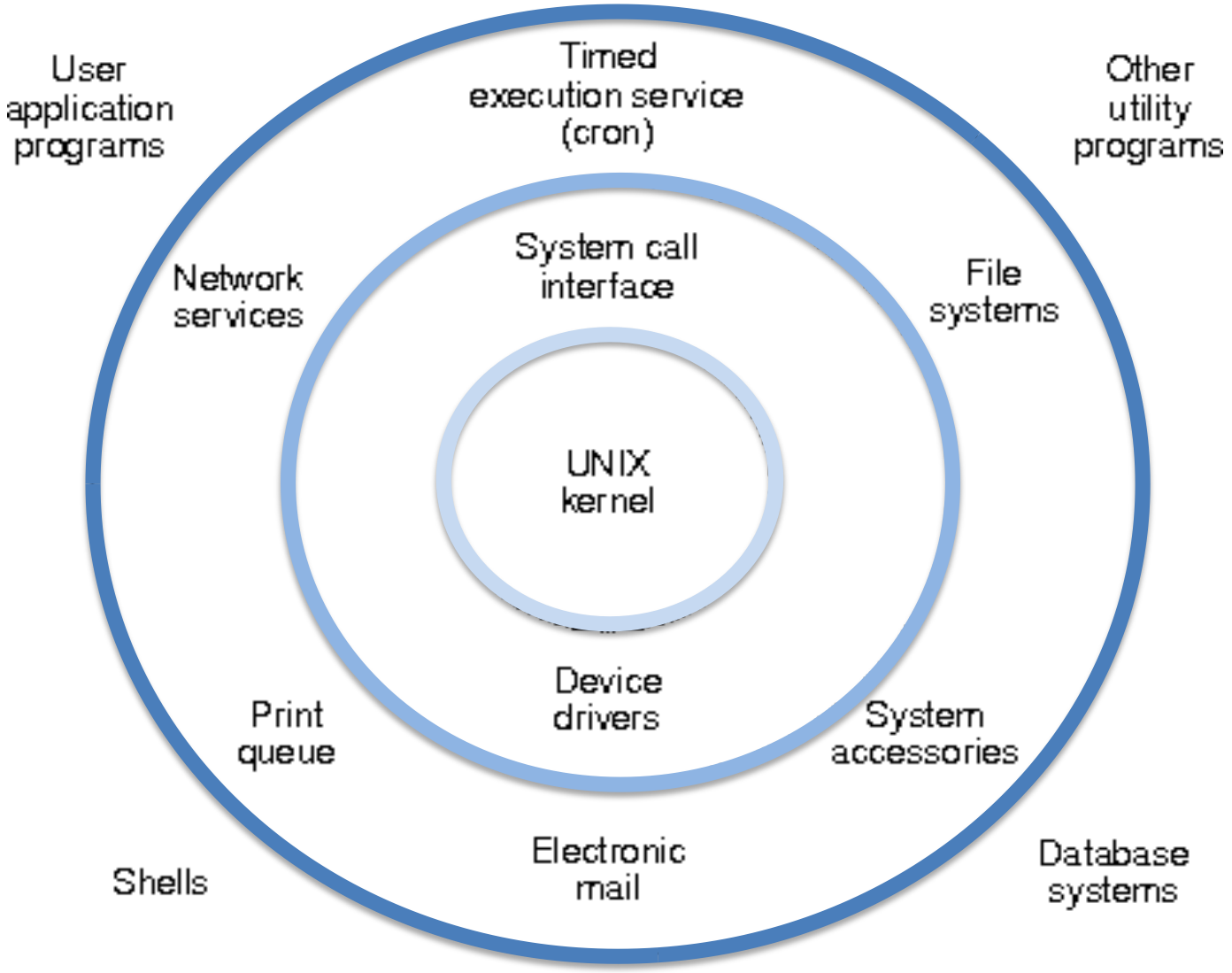
Address Translation Hardware

- Early virtual memory systems used to do translation in software
  - Meaning the OS did it
  - An additional memory access for each memory access!
- Address translation hardware solved this problem
  - *Translation lookaside buffer* (TLB)
- Modern CPUs contain TLB hardware
  - Fast cache
  - Modern workloads are TLB-miss dominated
  - Good things often come in small sizes
    - we have seen other instances of this with hardware

Takeaway Message

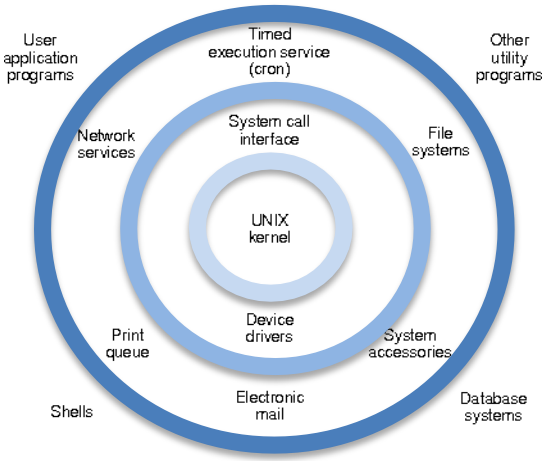
- Modern architectures provide lots of features to help the OS do its job
  - Protection mechanisms (modes)
  - Interrupts
  - Device I/O
  - Synchronization
  - Virtual Memory (TLB)
- Otherwise impossible or impractical in software
- Which of these are essential?
- Which are useful but not essential?

Operating System Layers



System Layers

- Application
- Libraries (in application process)
- System Services
- OS API
- Operating system kernel
- Hardware



Applications to Libraries

- Application programming interface (API)
- Libraries
  - Example: *libc*
- Library routines
  - Example: *printf()* of *stdio.h*
- All within the process's address space
  - Statically linked
    - libraries are included as part of the application code
    - calls are resolved at compile time
  - Dynamically linked
    - libraries are loaded by the OS at execution time as needed
    - jump tables and pointers are resolved dynamically by linker

Libraries to System Routines

- System call interface
  - UNIX man pages, section 2
  - Examples
    - *open()*, *read()*, *write()* – defined in *unistd.h*
  - Call these via libraries? *fopen()* vs. *open()*
  - See links in schedule
- Special files
  - Drivers
  - */proc*
  - *sysfs*

System to Hardware

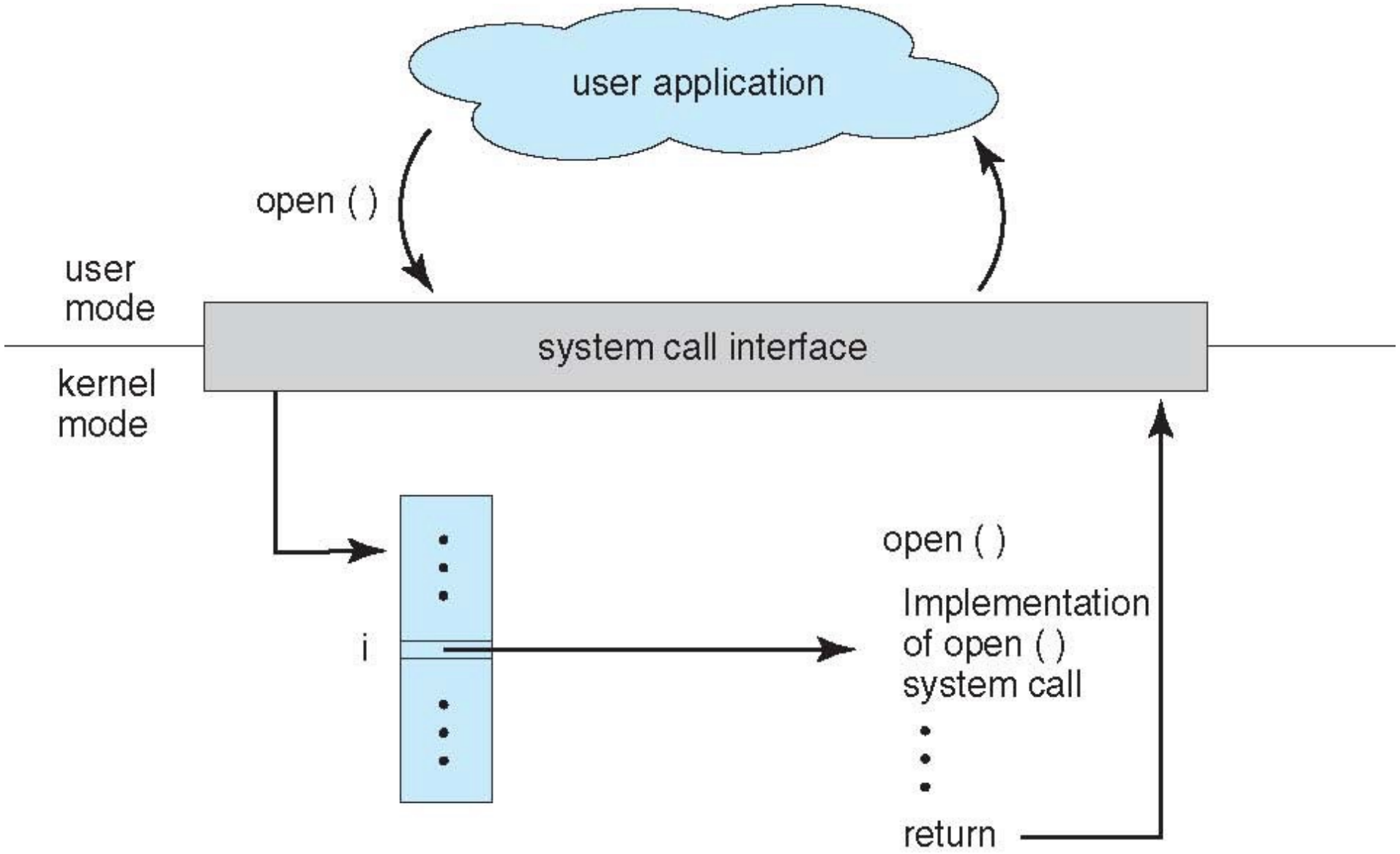
- Software-hardware interface
- OS kernel functions
  - Concepts ⟺ Managers (hardware)
  - Files ⟺ Files systems (drivers and devices)
  - Address space ⟺ Virtual memory (memory)
  - Programs ⟺ Process model (CPU, ISA)
- OS provides abstractions of devices and hardware objects
  - These abstractions are represented in software running in the OS and data structures that it maintains

Systems Calls

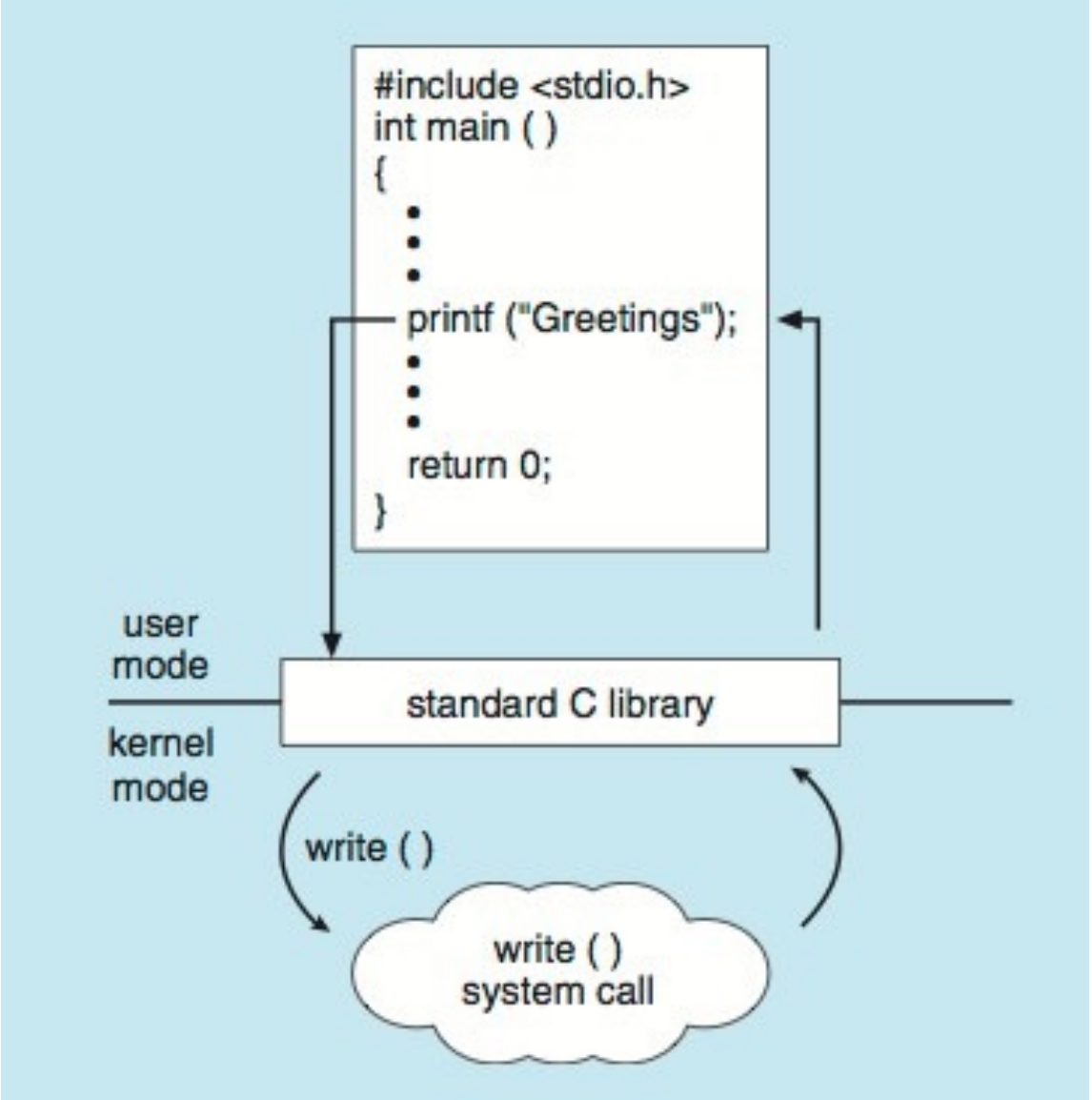
- Programming interface to OS services via system libraries
- Typically written in a high-level language (C, C++)
- Mostly accessed by programs via a high-level *application programming interface* (API) (versus direct system call use)
  - Win32 (Windows), POSIX (Unix, Linux, MacOS), Java (JVM)
- Typically, a number is associated with each system call
  - System-call interface maintains a table indexed by call #
- System call interface invokes the intended system call in OS kernel and returns status of the system call and return values
- Caller just obeys API and understand what OS will do
  - Most details of OS interface hidden by API

Note: names of systems calls in slides are generic

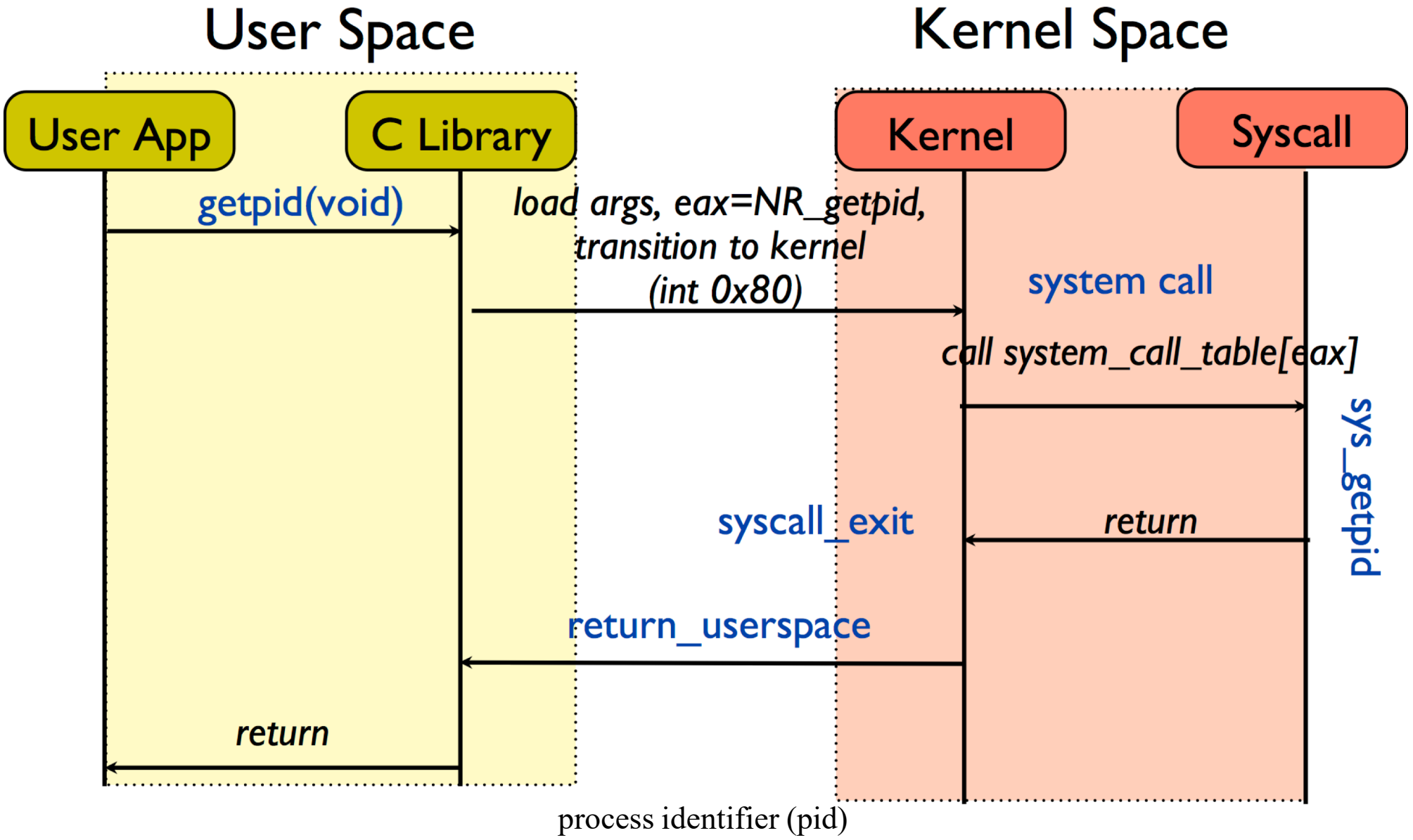
System Call – OS Relationship



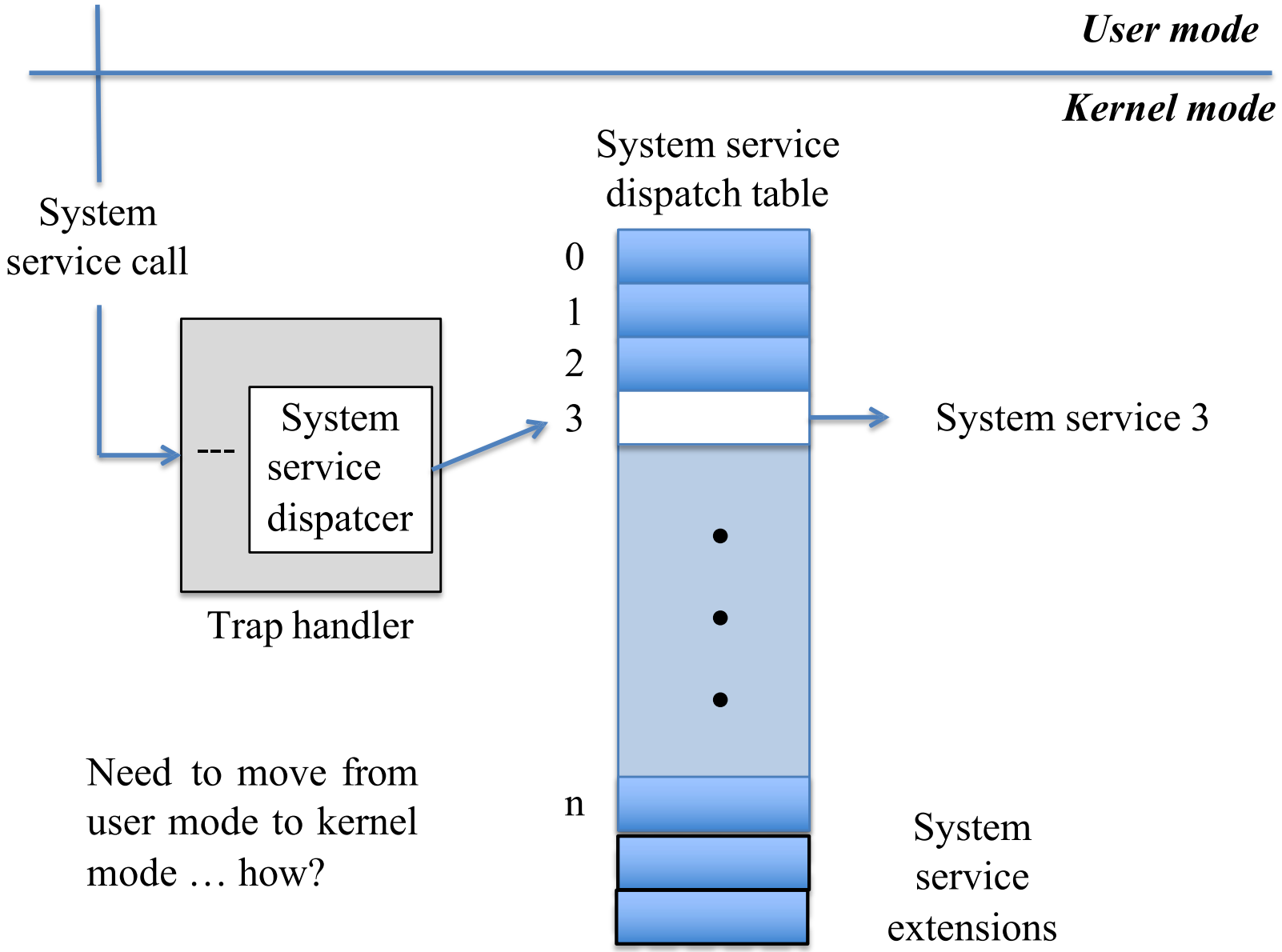
Standard C Library Example (printf())



System Call Example (getpid())



System Call Handling

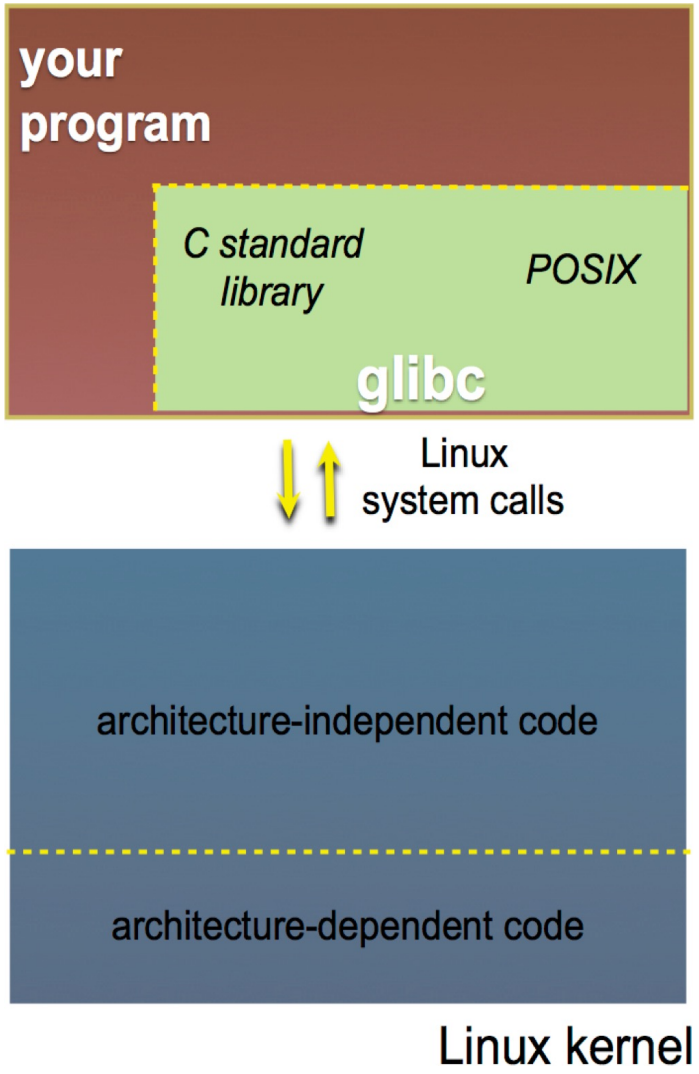


System Call Process

- Procedure call in user process
- Initial work in user mode — *libc*
- Trap instruction to invoke kernel — *int 0x80*
- Invoke system calls — *sys_read, mmap2*
- Do I/O — *read from disk*
- Wait — *disk is slow*
- Completion interrupt handling
- Return-from-interrupt instruction
- Final work in library — *libc*
- Return to user code

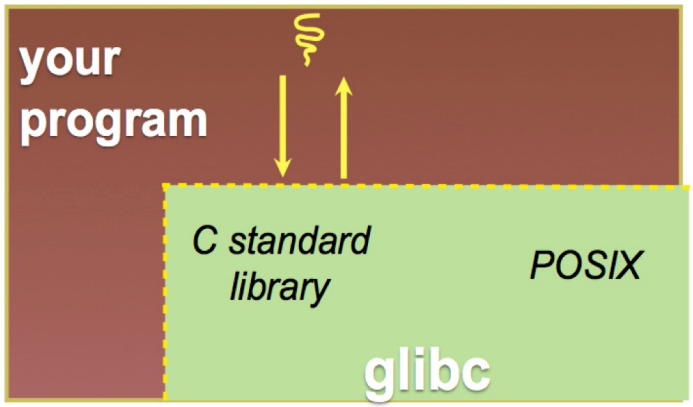
Details on x86 / Linux

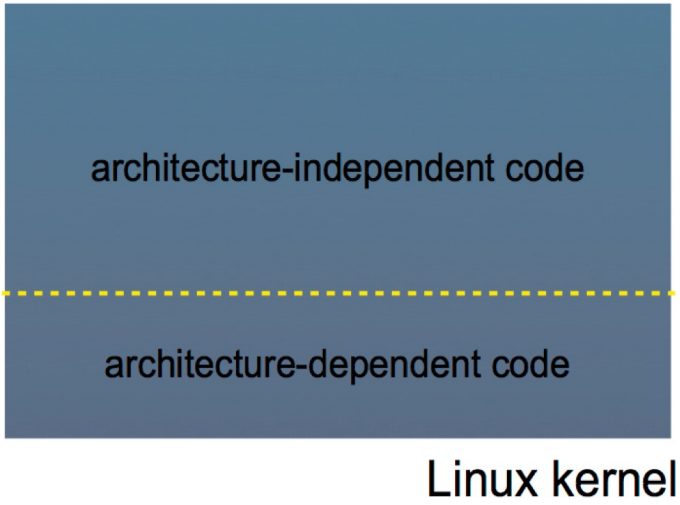
- A more accurate picture:
  - Consider a typical Linux process
  - Its course of execution can be in several places
    - in your program's code
    - in C standard library
    - in the kernel model (system calls)



Details on x86 / Linux

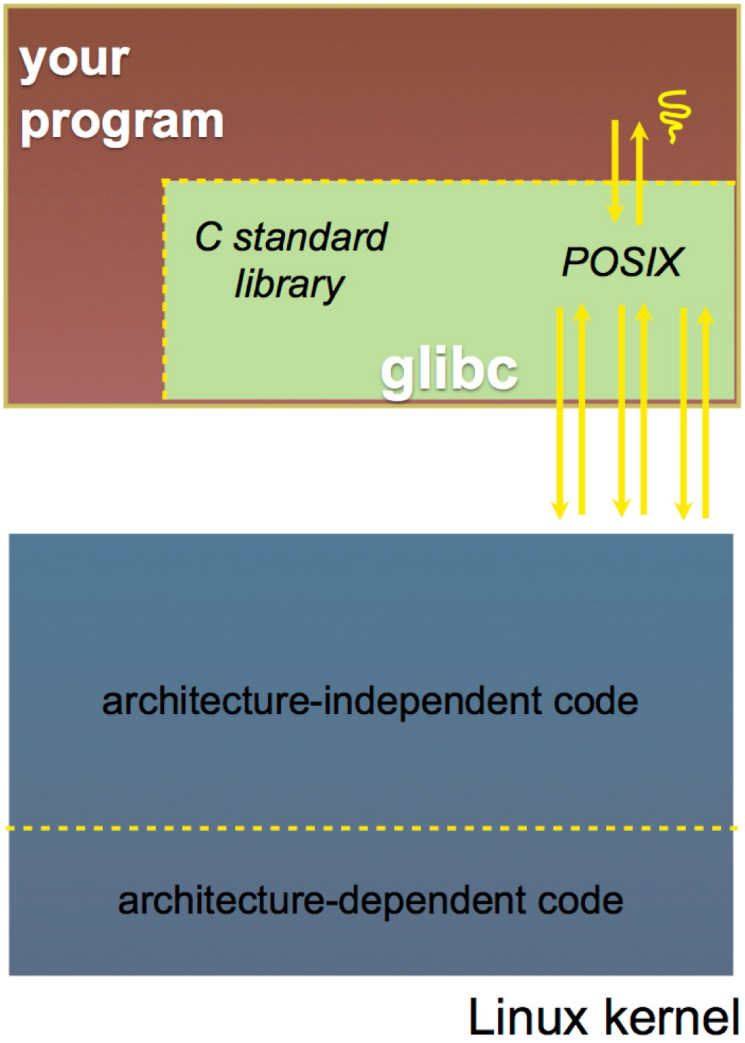
- Some routines your program invokes may be entirely handled by glibc
  - Without involving the kernel
    - for example, *strcmp( )* from stdio.h
  - There is some initial overhead when invoking functions in dynamically linked libraries …
  - … but after symbols are resolved, invoking glibc routines is nearly as fast as a function call within your program itself





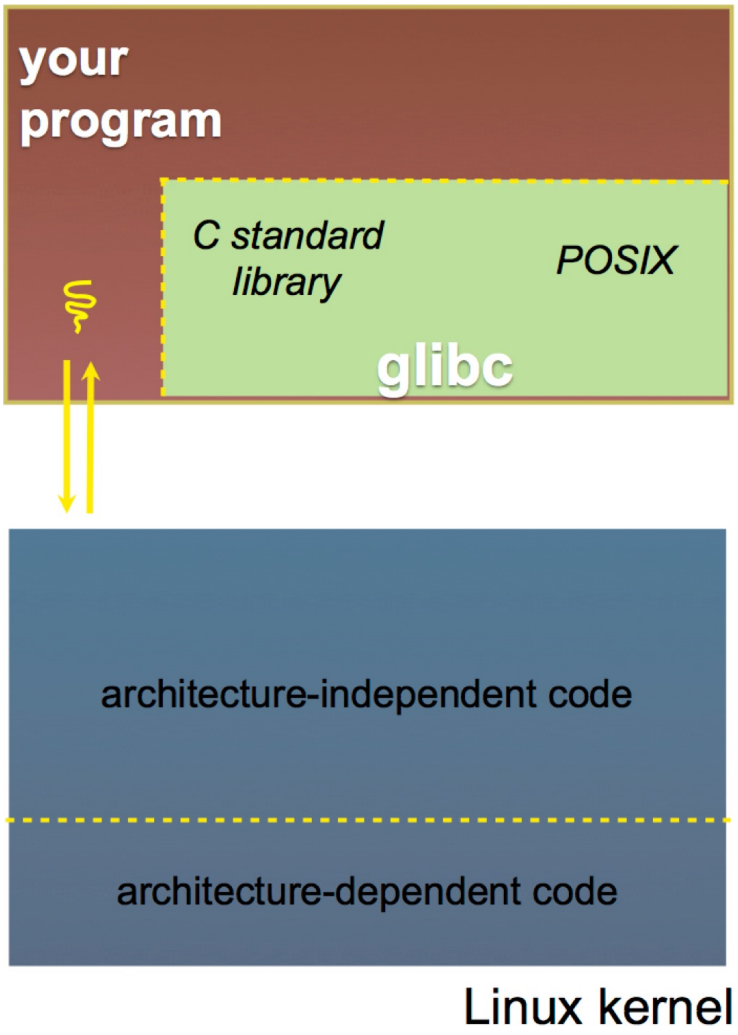
Details on x86 / Linux

- Some routines may be handled by glibc, but they in turn invoke Linux system calls
  - Example: POSIX wrappers around Linux syscalls
    - POSIX *readdir( )* invokes the underlying Linux *readdir( )*
  - Example: C *stdio* functions that read and write from files
    - *fopen( ), fclose( ), fprintf( ), …* invoke underlying Linux *open( ), read( ), write( ), close( ), …*



Details on x86 / Linux

- Your program can choose to directly invoke Linux system calls as well
  - Nothing forces you to link with glibc and use it
  - But relying on directly invoked Linux system calls may make your program less portable across UNIX varieties



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

System Programs

- System programs provide a convenient environment for program development and execution
- They can be divided into categories:
  - File manipulation
  - Status information
  - File modification
  - Programing language support (program development)
  - Program loading and execution
  - Application programs
- Most user's view of the operating system is defined by system programs, not the actual system calls

File Interface

- Goal:
  - Provide a uniform abstraction for accessing the OS and its resources
- Abstraction:
  - File
- Use file system calls to access OS services
  - Devices, sockets, pipes, and so on
  - Also us in OS in general

I/O with System Calls

- Much I/O is based on a streaming model
  - Sequence of bytes
- *write()* sends a stream of bytes somewhere
- *read()* blocks until a stream of input is ready
- Annoying details:
  - Might fail, can block for a while
  - Working with file descriptors
  - Arguments are pointers to character buffers
  - See the *read()* and *write()* man pages

File Descriptors

- A process might have several different I/O streams in use at any given time
- These are specified by a kernel data structure called a *file descriptor*
  - Each process has its own table of file descriptors
- *open()* associates a file descriptor with a file
- *close()* destroys a file descriptor
- Standard input and standard output are usually associated with a terminal
  - More on that later

Regular File

- File has a *pathname*: */tmp/foo*
- Can open the file
  - *int fd = open("/tmp/foo", O_RDWR)*
  - For reading and writing
- Can read from and write to the file
  - *bytes = read(fd, buf, max); /* buf get output */*
  - *bytes = write(fd, buf, len); /* buf has input */*

flags for read/write access

pointer to buffer

Socket File

- File has a pathname: */tmp/bar*
  - Files provide a persistence for a communication channel
  - Usually used for local communication (UNIX domain sockets)
- Open, read, and write via socket operations
  - *sockfd = socket(AF_UNIX, TCP_STREAM, 0);*
  - *local.path* is set to */tmp/bar*
  - *bind (sockfd, &local, len)*
  - Use sock operations to read and write

Device File

- Files for interacting with physical devices
  - */dev/null* (no device)
  - */dev/cdrom* (CD-drive)
- Use file system operations, but are handled in device-specific ways
  - *open()*, *read()*, *write()* correspond to device-specific functions (act as function pointers!)
  - Also, use *ioctl* (I/O control) to interact (later)

Sysfs File and /proc Files

- These files enable reading from and writing to kernel
- */proc* files
  - Enable reading of kernel state for a process
  - Process information pseudo-file system
  - Does not contain "real" files, but runtime system information
    - System memory
    - Devices mounted
    - Hardware configuration
  - A lot of system utilities are simply calls to files in this directory
  - By altering files located in this directory you can even read/change kernel parameters (*sysctl*) while the system is running.
- *sysfs* files
  - Provide functions that update kernel data
    - file's write function updates kernel based on input data

Other System Calls

- It's possible to hook the output of one program into the input of another
  - *pipe()*
- It's possible to block until one of several file descriptor streams is ready
  - *select()*
- Special calls for dealing with network
  - AF_INET sockets, and so on
- Send a message to other (or all) processes
  - *signal()*
- Most of these in section 2 of manual

Syscall Functionality

- System calls are the main interface between processes and the OS
  - Like an extended "instruction set" for user programs that hide many details
  - First Unix system had a couple dozen system calls
  - Current systems have many more
    - >300 in Linux and >500 in FreeBSD
  - Understanding the system call interface of a given OS lets you write useful programs under it
- Natural questions to ask:
  - Is this the right interface? how to evaluate?
  - How can these system calls be implemented?

OS Design and Implementation

- Design and implementation of OS is not "solvable" in full, but some approaches have proven successful
- Internal structure of different operating systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- There are *user* goals and *system* goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

OS Policy versus Mechanism

- Important principle to separate
  Policy: What will (should) be done?
  Mechanism: How to do it?
  - Mechanisms determine how to do something
  - Policies decide what will be done
- Separation of policy from mechanism is a **VERY** important principle
  - Allows maximum flexibility if policy decisions are to be changed later
  - Universal principle
- Specifying and designing an OS is a highly creative task of software engineering

Summary

- Operating systems must balance many needs
  - Impression that each process has individual use of system
  - Comprehensive management of system resources
- Operating system structures try to make use of system resources straightforward
  - Libraries
  - System services
  - System calls and other interfaces

Next Class

- Processes