CIS 415: Operating Systems

OS Structure

Prof. Kevin Butler
Spring 2014

“Flagship” Universities

- Why attend a research “flagship” university as an undergraduate?
- Why UO?

Why Do Research?

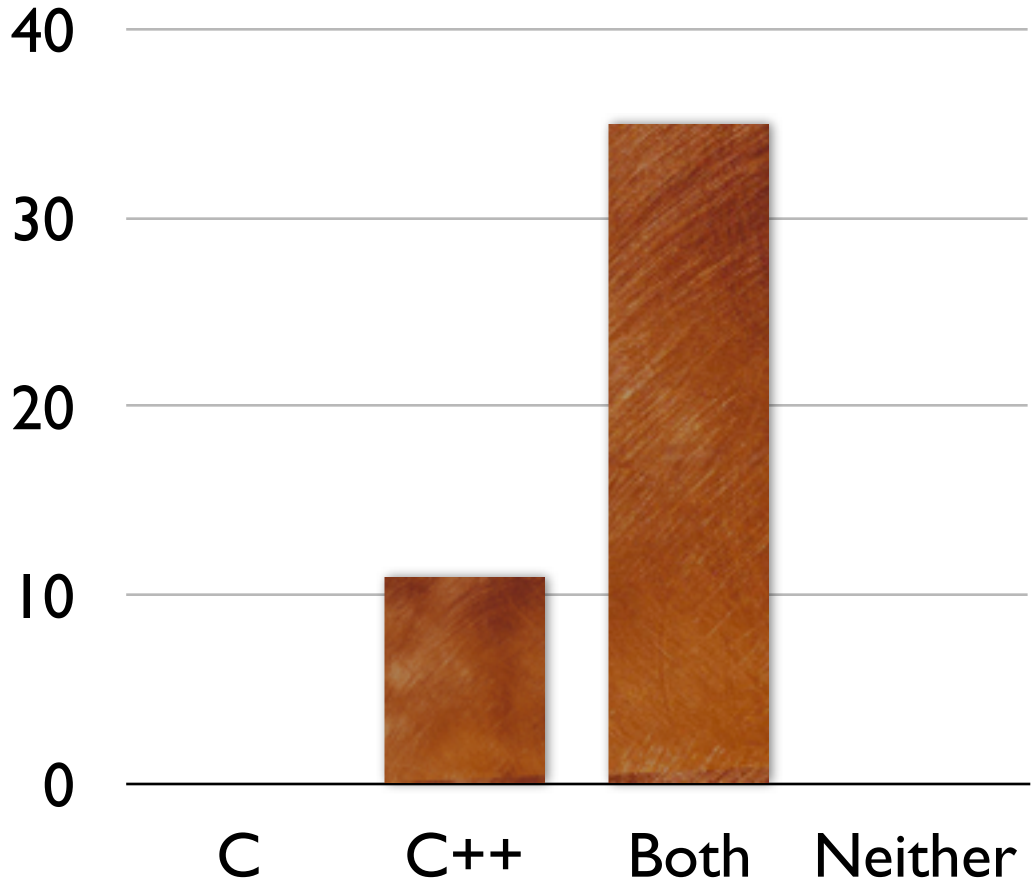
- As an undergraduate:
  - pursue your interests
  - be involved with discovering new ideas/techniques
  - hone problem-solving skills
  - work with faculty in a far different way than with courses
- As a graduate student:
  - It's expected that you create new contributions to the field
  - Increase your depth of knowledge and understanding
  - Be 5-10 years ahead of the mainstream

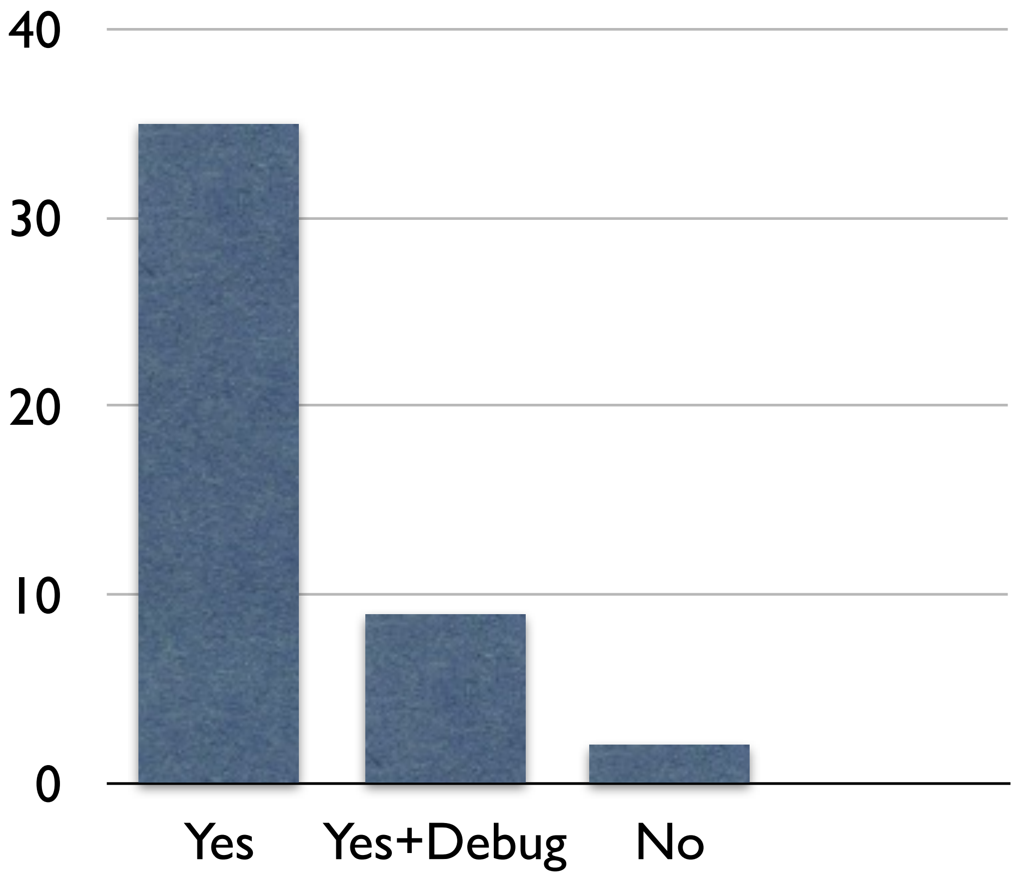
Research and Your Career

- Want to go to grad school?
  - Why? How will you know without doing research?
  - Publications are impressive and starting to be expected at the top schools
  - Similarly the case if you want to compete for major awards and scholarships/fellowships
- What if I don’t want to go into academia?
  - Top research labs (e.g., MSR) look for strong pub record
  - Publishing in peer-reviewed venue gives you authority
  - Even outside labs, shows ability to go deep and explain

Survey Results

UNIVERSITY OF OREGON



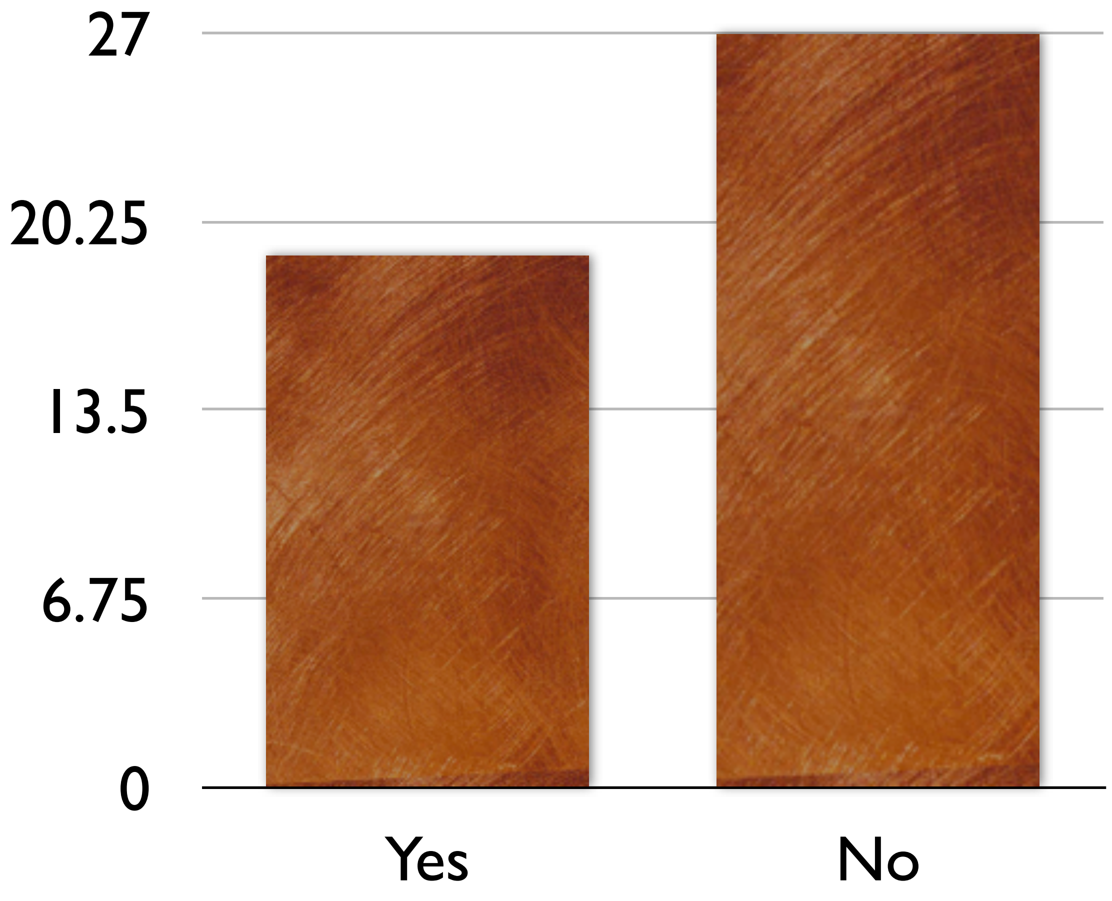


Languages used

C Pointers

Survey Results

`(*(x+5))[5] = &y`



Survey Results

```
unsigned char *mystery_function(unsigned short bufsize) {
unsigned char *tmp_buf;

if (bufsize == 0)
return NULL;

tmp_buf = malloc(bufsize);
if (tmp_buf == NULL)
return NULL;

if (verify_something() == 0) /* something bad happened */
return NULL;

return tmp_buf;
}
```

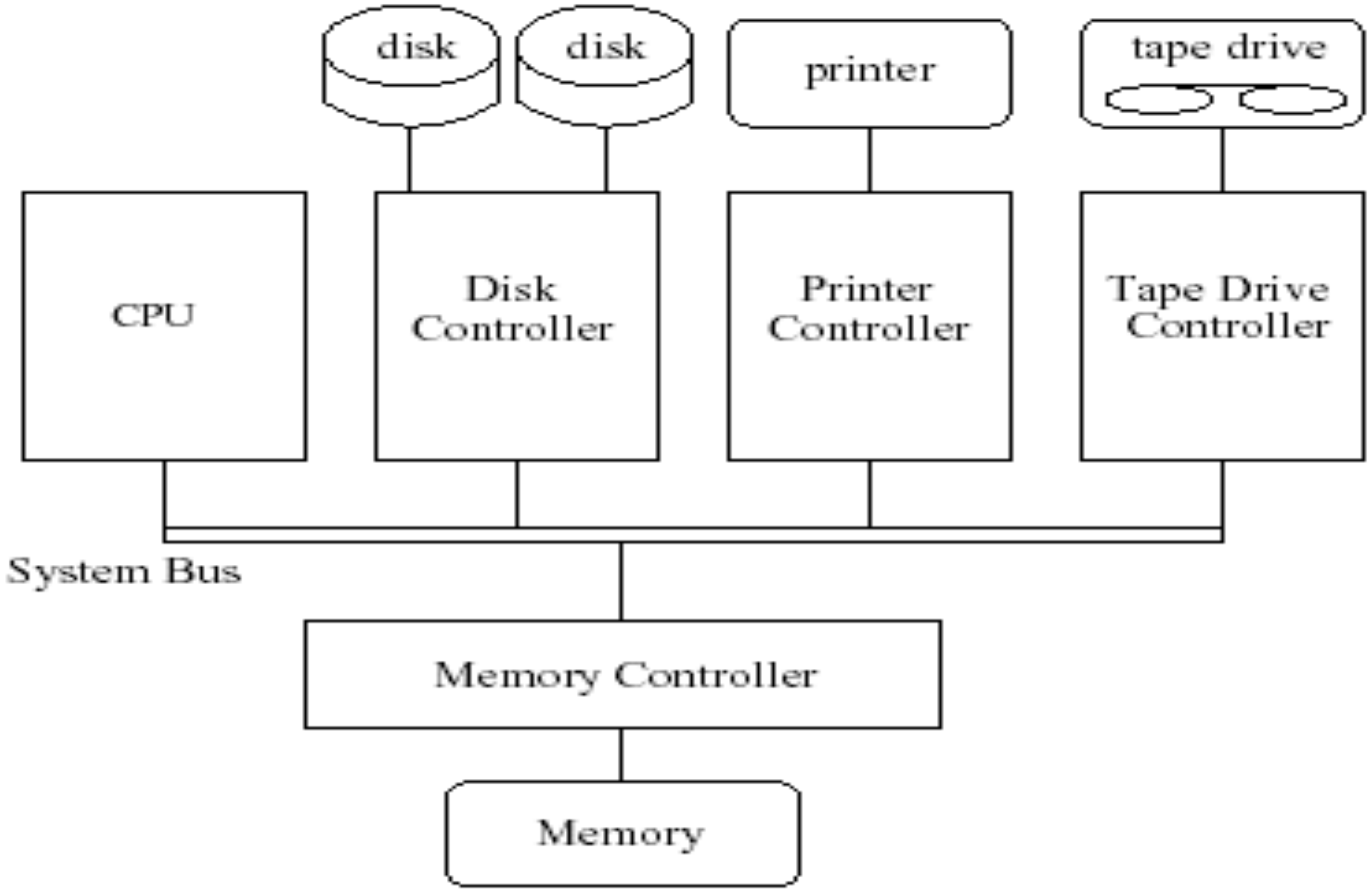
free(tmp_buf);

Other results

- A few of you are Linux experts, most have some experience, a few with none
- 2/3 have done socket programming, 4/5 have written a makefile, 1/3 have written multithreaded code
- Half have used man pages: man sigaction(2)?
- Not much programming with semaphore/mutex
- Not much exposure to process creation

Canonical System Hardware

- CPU: Processor to perform computations
- Memory: Programs and data
- I/O Devices: Disk, monitor, printer, …
- System Bus: Communication channel between the above

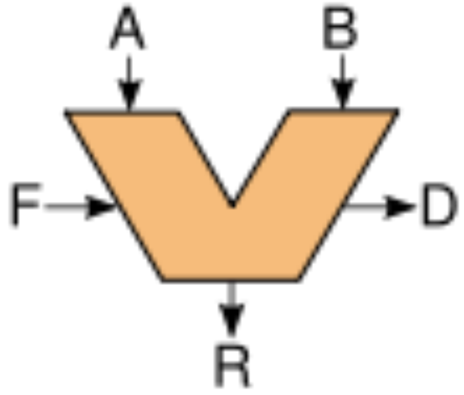


CPU

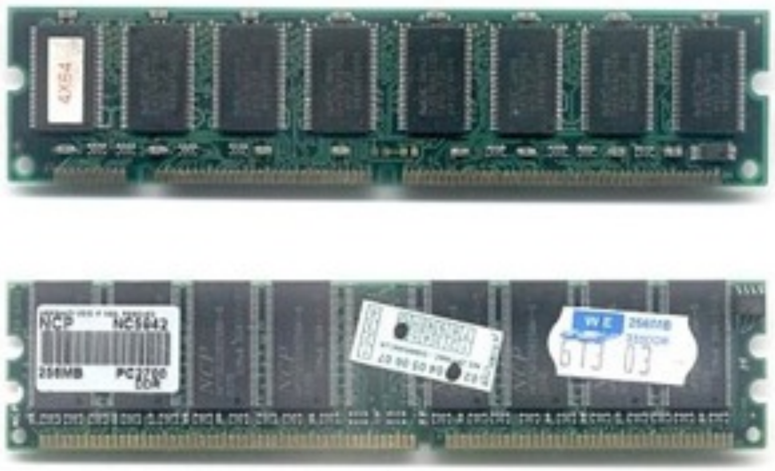
- CPU
  - Semiconductor device, digital logic (combinational and sequential)
  - Can be viewed as a combination of many circuits
- Clock
  - Synchronizes constituent circuits
- Registers
  - CPU's scratchpads; very fast; loads/stores
  - Most CPUs designed so that a register can store a memory address
    - n-bit architecture
- Cache
  - Fast memory close to CPU
  - Faster than main memory, more expensive
  - Not seen by the OS

CPU Instruction Execution

- Arithmetic Logic Unit (ALU)
- Program counter
  - Instruction address
- Instruction from the control unit (F)
- CPU data registers
  - Input A and B and Output R



Memory/RAM



- Semiconductor device
  - DIMMs mounted on PCBs
  - Random access: RAM
  - DRAM: Volatile, need to refresh
    - Capacitors lose contents within few tens of msecs
- CPU accesses RAM to fill registers
- OS sees and manages memory
  - Programs/data need to be brought to RAM
- Memory controller: Chip that implements the logic for
    - Reading/Writing to RAM (Mux/Demux)
    - Refreshing DRAM contents

Memory Access

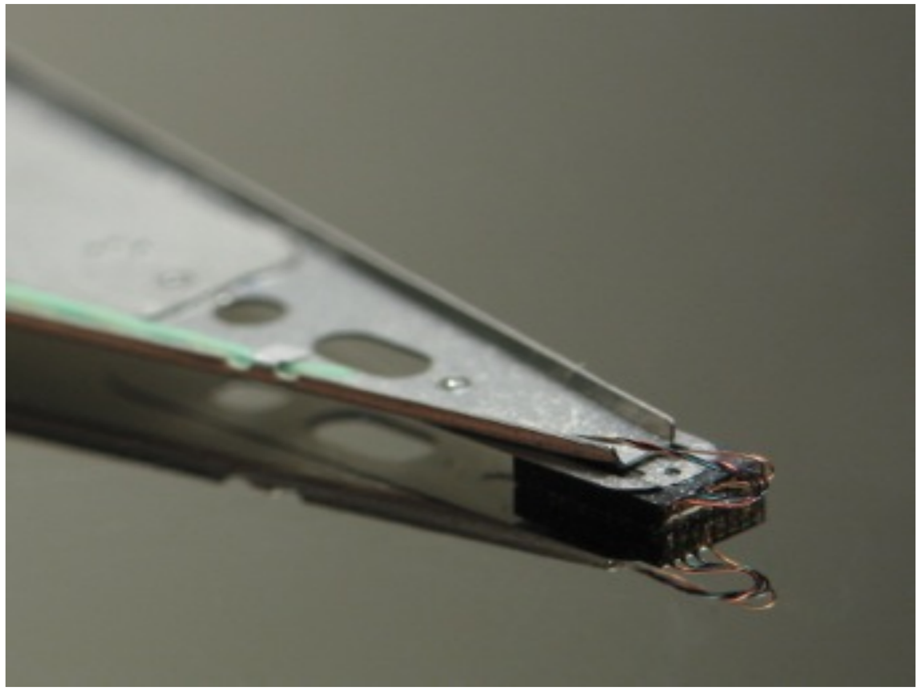
- Instructions
  - Program counter is used to fetch into control unit
  - Fetched into instruction register
- Data
  - Load/store instructions
  - Move data between memory locations

I/O Devices

- Large variety, varying speeds
  - Disk, tape, monitor, mouse, keyboard, NIC
  - Serial vs parallel
- Each has a controller
  - Hides low-level details from OS
  - Manages data flow between device and CPU/memory

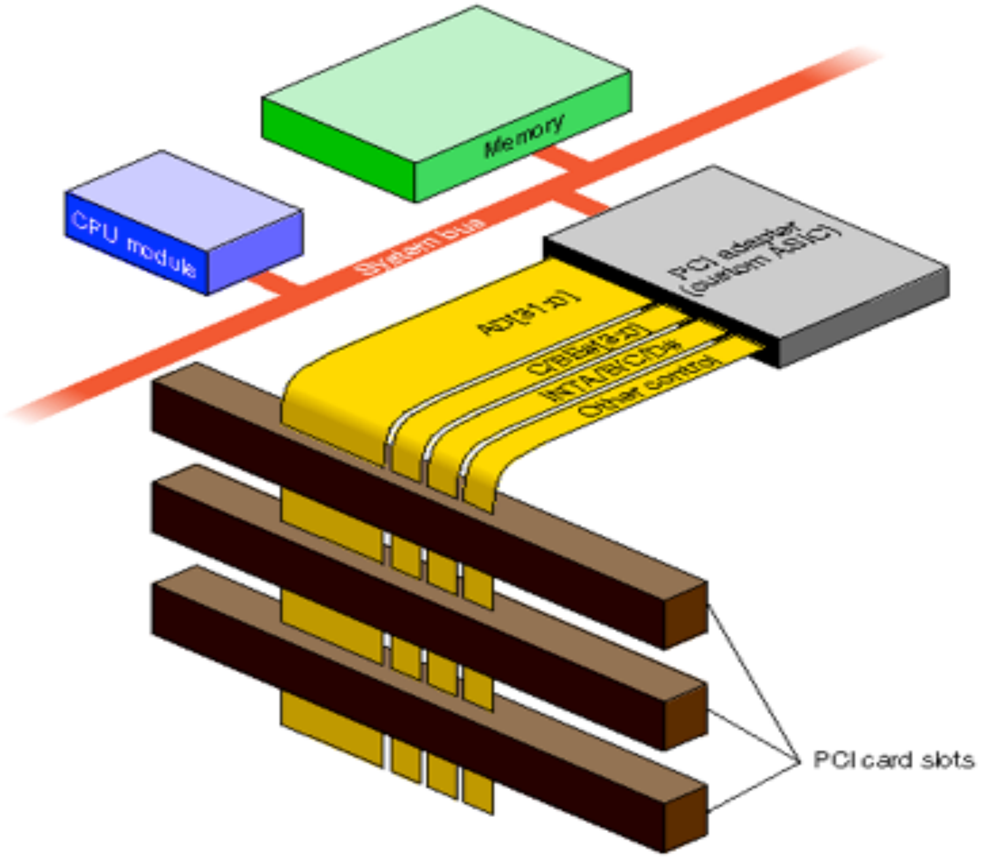
Hard Disk

- Secondary storage
- Mechanically operated
  - Sequential access
- Cheap => Abundant
- Very slow
  - Orders of magnitude
- Increasingly common: SSD
  - where in storage hierarchy?



Interconnects

- A bus is an interconnect for flow of data and information
  - Wires, protocol
  - Data arbitration
- System Bus
- PCI Bus
  - Connects CPU-memory subsystem to
    - Fast devices
    - Expansion bus that connects slow devices
- SCSI, IDE, USB, …
  - Will return to these later



Services & Hardware Support

- *Protection:* Kernel/User mode, 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)

Kernel/User Mode

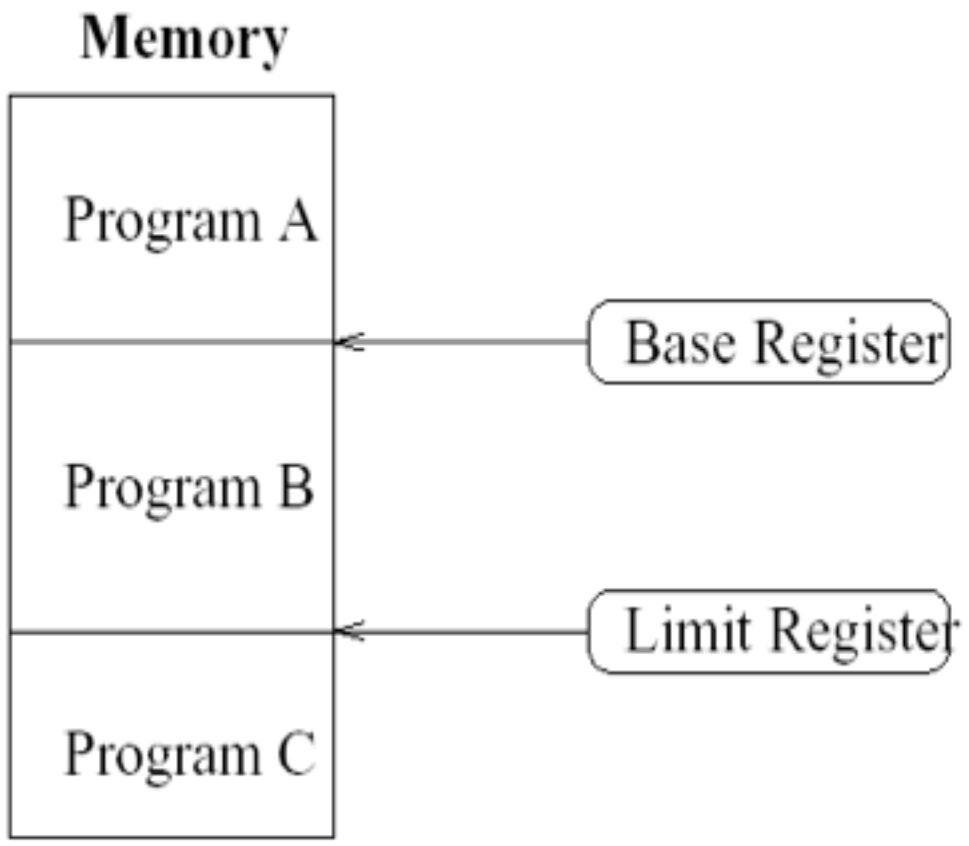
- A modern CPU has at least two modes
  - Indicated by status bit in protected CPU register
  - OS kernel runs in privileged mode
    - Also called kernel or supervisor mode
  - Applications run in normal mode
- OS can switch the processor to user mode
  - CPU can only access own address space, can’t talk to devices
- Events that need the OS to run switch the processor to privileged mode
  - E.g., division by zero
- OS definition: *Software that runs in privileged mode*

Protected Instructions

- Instructions that require privilege
  - Direct access to I/O
  - Modify page table pointers, TLB
  - Enable & disable interrupts
  - Halt the machine, etc.
- Access sensitive registers or perform sensitive operations

Base and Limit Registers

- Hardware support to protect memory regions
- Loaded by OS before starting program
- CPU checks each reference
- Instruction & data addresses
- Ensures reference in range



Interrupts

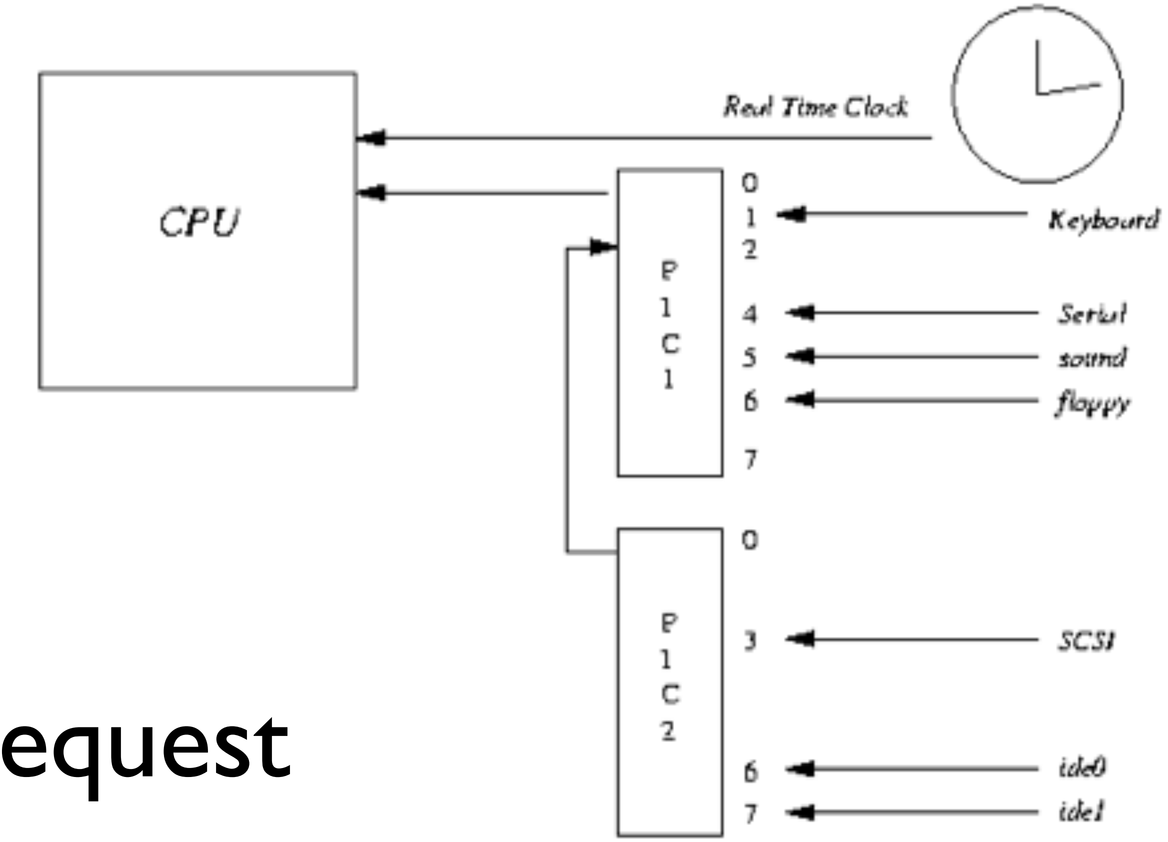
- Polling = "are we there yet?" "no!" (repeat…)
  - Inefficient use of resources
  - Annoys the CPU
- Interrupt = silence, then: "we're there"
  - I/O device has own processor
  - When finished, device sends interrupt on bus
  - CPU "handles" interrupt

Interrupts

- Asynchronous signal indicating need for attention
    - Replaces polling for events
- Represent
    - Normal events to be noticed and acted upon
        - Device notification
        - Software system call
    - Abnormal conditions to be corrected
    - Abnormal conditions that cannot be corrected

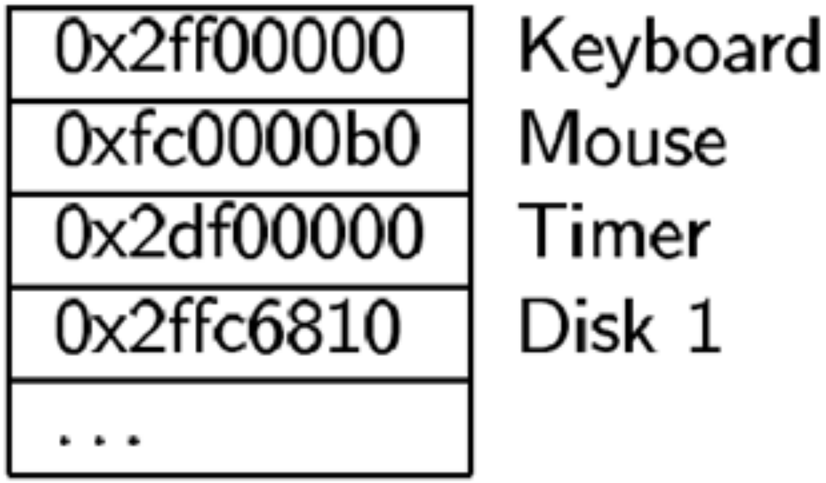
Hardware Interrupts

- Signal from a device
  - Implemented by a controller (e.g., memory)
- Examples
  - Timer
  - Keyboard, mouse
  - End of DMA transfer
- Response to processor request
- Unsolicited response



Timer

- OS needs timers for
  - Time of day
  - CPU scheduling
- Interrupt vector for timer



0x2ff00000	Keyboard
0xfc0000b0	Mouse
0x2df00000	Timer
0x2ffc6810	Disk 1
...	

Software Interrupts

- Software interrupts (Traps)
  - Special interrupt instructions
    - int 0x80 -- System call
  - 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, can't 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 and start running at first address of the program
- OS keeps record of every process
    - assigned memory, current program counter, etc.
    - This is the process *context*
    - Enough info to restart process where it left off

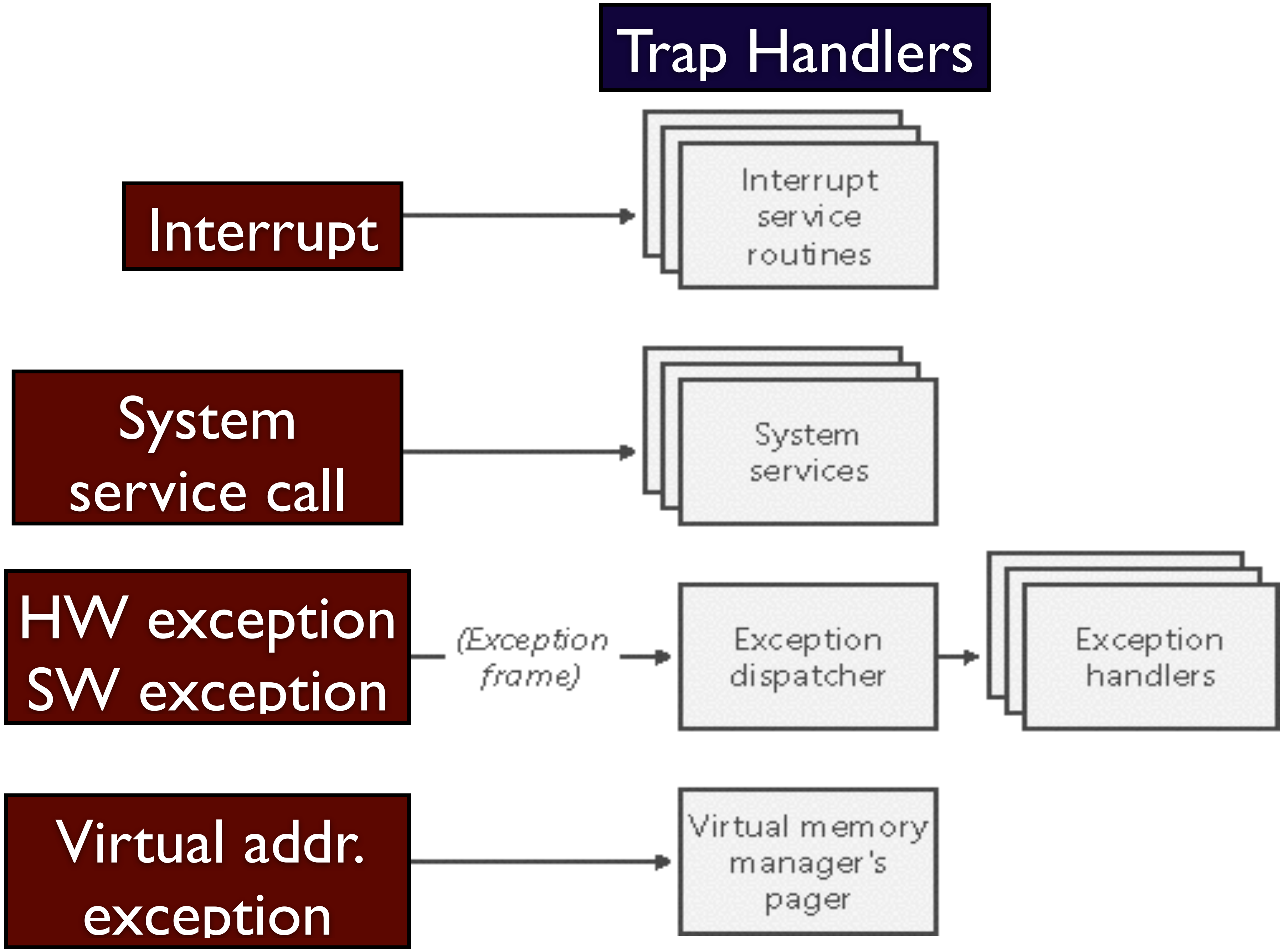
Dealing with interrupts

- Eventually a hardware interrupt or a trap will happen
  - e.g., received input from keyboard, clock ticked, etc
- OS records state of running process's context
  - stored in a *process control block (PCB)*
- Next, OS services the interrupt
  - e.g., send something to the printer
- Finally, pick process to restart
  - maybe the one that was running, maybe not (scheduling!)
  - moves back into user mode

Interrupt Handling (details)

- Each interrupt has a corresponding *Interrupt Handler*
- When an interrupt request (IRQ) is received
  - If interrupt mask allows interrupt
  - Save state of current processing
    - At time of interrupt something else may be running
    - State: Registers (stack ptr), program counter, etc.
  - Execute handler
  - Return to current processing

Interrupt Handling



Trap Handlers

Interrupt

System service call

HW exception
SW exception

Virtual addr. exception

Multiple Interrupts

Executing in user mode

↓

Make system call

↓

Disk Interrupt

↓

Clock Interrupt

Kernel context layer 1

Execute syscall, save user registers

Kernel context layer 2

Execute disk handler

Save register context of syscall

Kernel context layer 3

Execute disk handler

Save register context of disk

Device Access

- Port I/O
  - Uses special I/O instructions
  - Port number, device address
    - Separate from process address space
- Memory-mapped I/O
  - Uses memory instructions (load/store)
    - To access memory-mapped device registers
  - Does not require special instructions
    - But consumes some memory for I/O

Direct Memory Access

- Direct access to I/O controller through memory
- Reserve area of memory for communication with device (“DMA”)
  - Video RAM:
    - CPU writes frame buffer
    - Video card displays it
- Fast and convenient

Synchronization

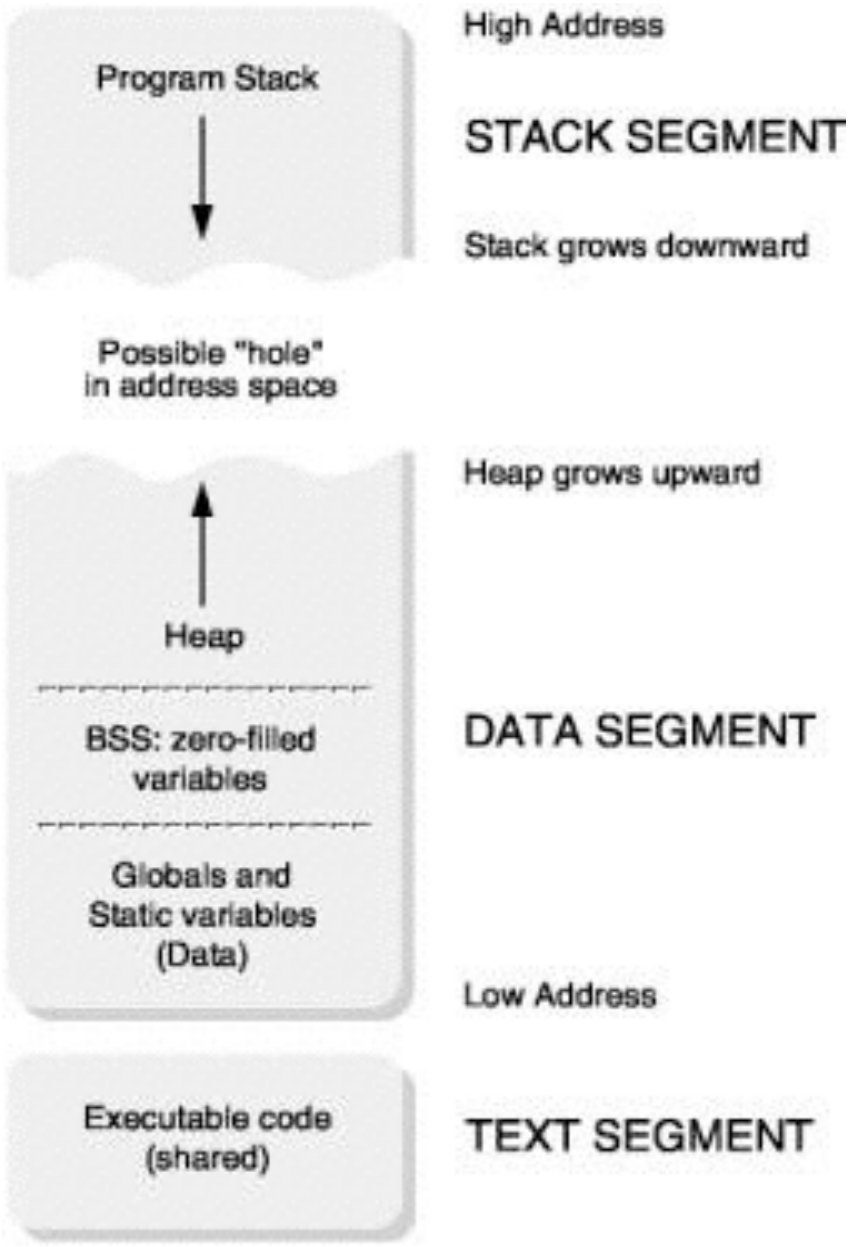
- How can OS synchronize concurrent processes?
  - E.g., multiple threads, processes & interrupts, DMA
- CPU must provide mechanism for atomicity
  - Series of instructions that execute as one or not at all

Synchronization: How-To

- One approach:
  - Disable interrupts
  - Perform action
  - Enable interrupts
- Advantages:
  - Requires no hardware support
  - Conceptually simple
- Disadvantages:
  - Could cause starvation
- Modern approach: atomic instructions (e.g., test & set, compare & swap, Intel LOCK instruction)

Process Address Space

- All locations addressable by the process
- Can restrict use of addresses (RW)
- Restrictions enforced by OS
- Every running program can have its own private address space
  - How?



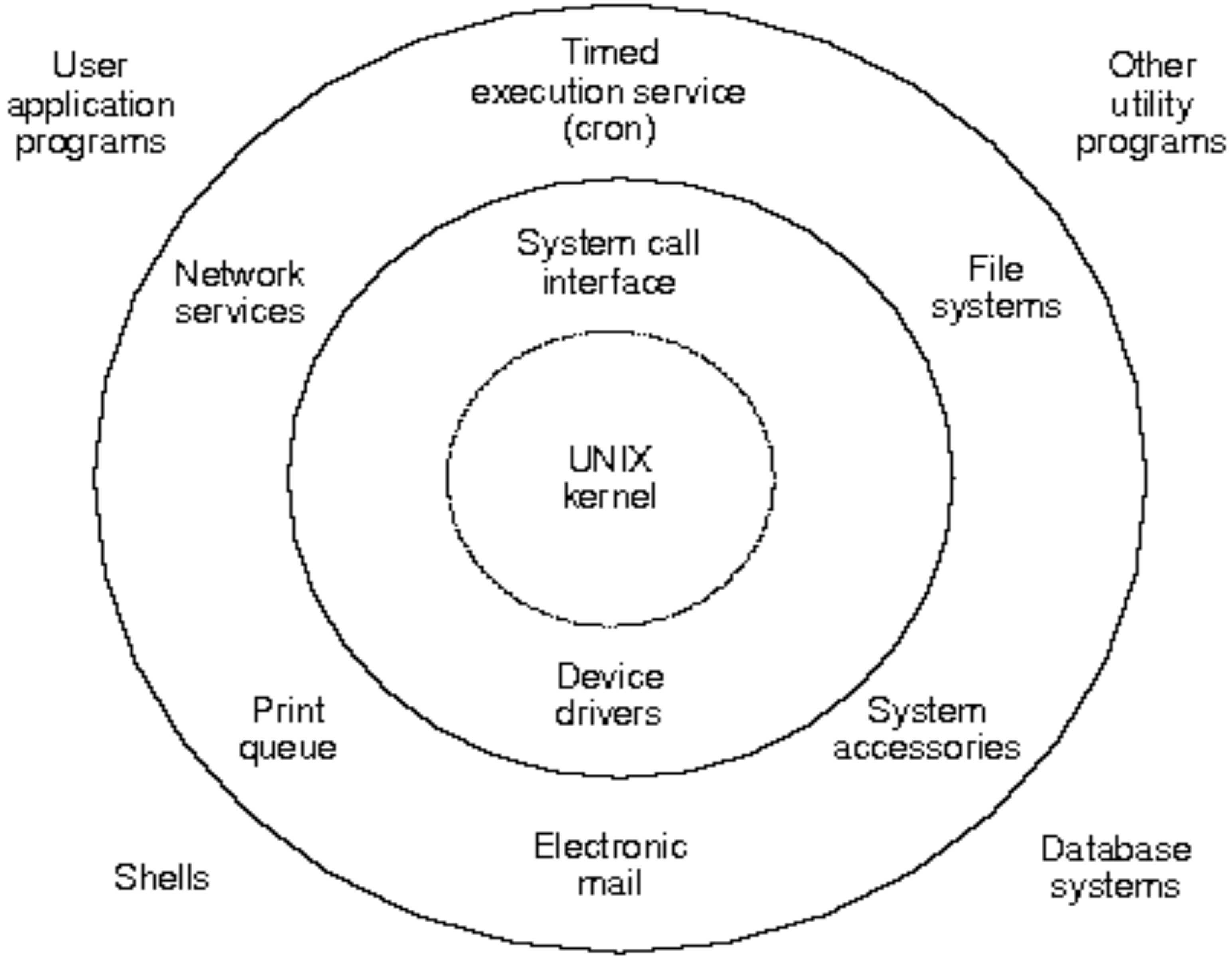
Virtual Memory

- 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 (think MS Office)
- OS needs to maintain mapping between physical and virtual memory
  - Page tables stored in memory

Translation Lookaside Buffer

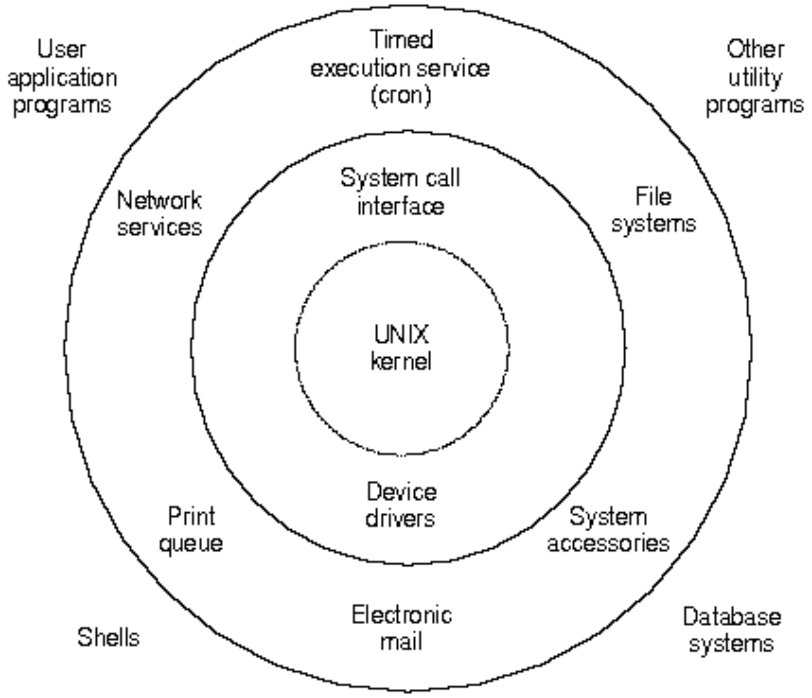
- Initial virtual memory systems used to do translation in software
  - Meaning the OS did it
  - An additional memory access for each memory access!
    - S.l.o.w.!!!
- Modern CPUs contain hardware to do this: the TLB
  - Fast cache
  - Modern workloads are TLB-miss dominated
  - Good things often come in small sizes
    - We have seen other instances of this

Operating System Layers



System Layers

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



Applications to Libraries

- Application Programming Interface
  - Library functions (e.g., libc)
- Examples
  - `printf` of stdio.h
- All within the process's address space
  - Static and Dynamic linking

Applications to Services

- Provide syntactic sugar for using resources
  - Printing, program mgmt, network mgmt, file mgmt, etc.
  - E.g., `chmod`
- Provide special functions beyond OS
  - E.g., `cron`
- UNIX `man` pages, sections 1 and 8

Libraries to System

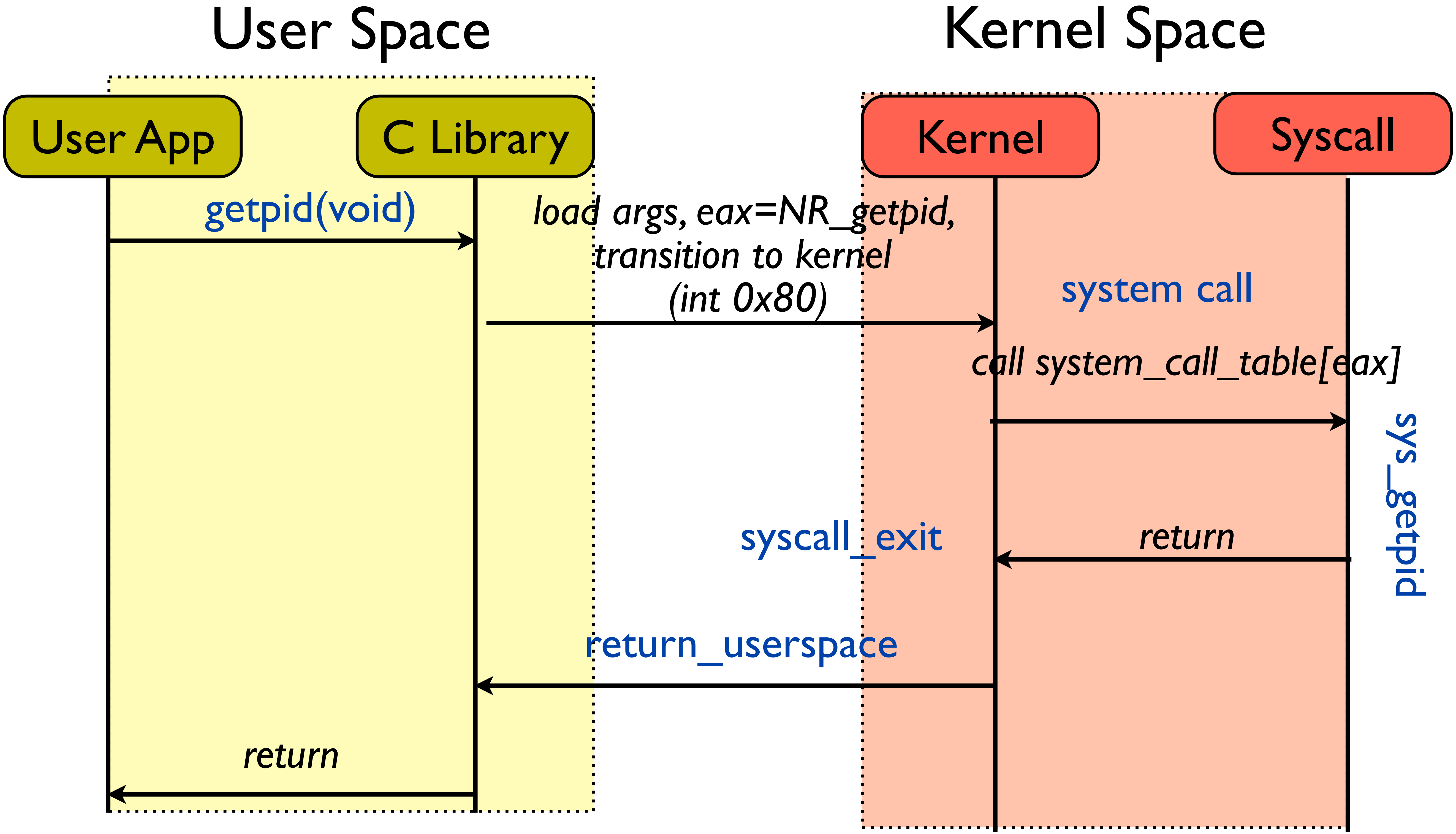
- System call interface
  - UNIX man pages, section 2
  - Examples
    - open, read, write – defined in `unistd.h`
  - Call these via libraries? `fopen` vs. `open`

- Special files
  - Drivers, `/proc`, sysfs

System to Hardware

- Software-hardware interface
- OS kernel functions
  - Concepts == Managers -- Hardware
  - Files == filesystems – drivers/devices
  - Address space == virtual memory -- memory
  - Instruction Set == process model -- CPU
- OS provides abstractions of devices and hardware objects (files)

System Calls: Overview



System Call Handling

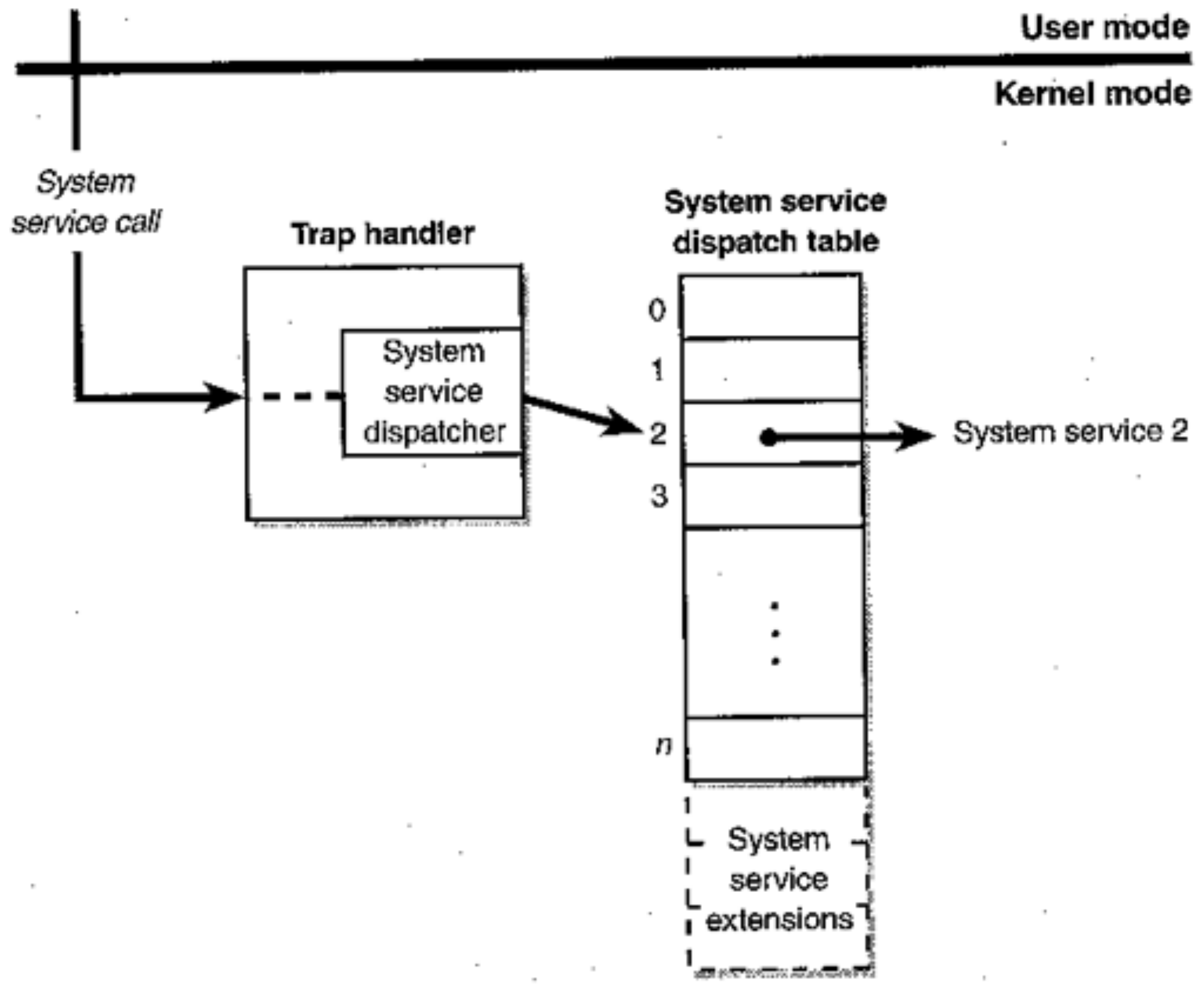


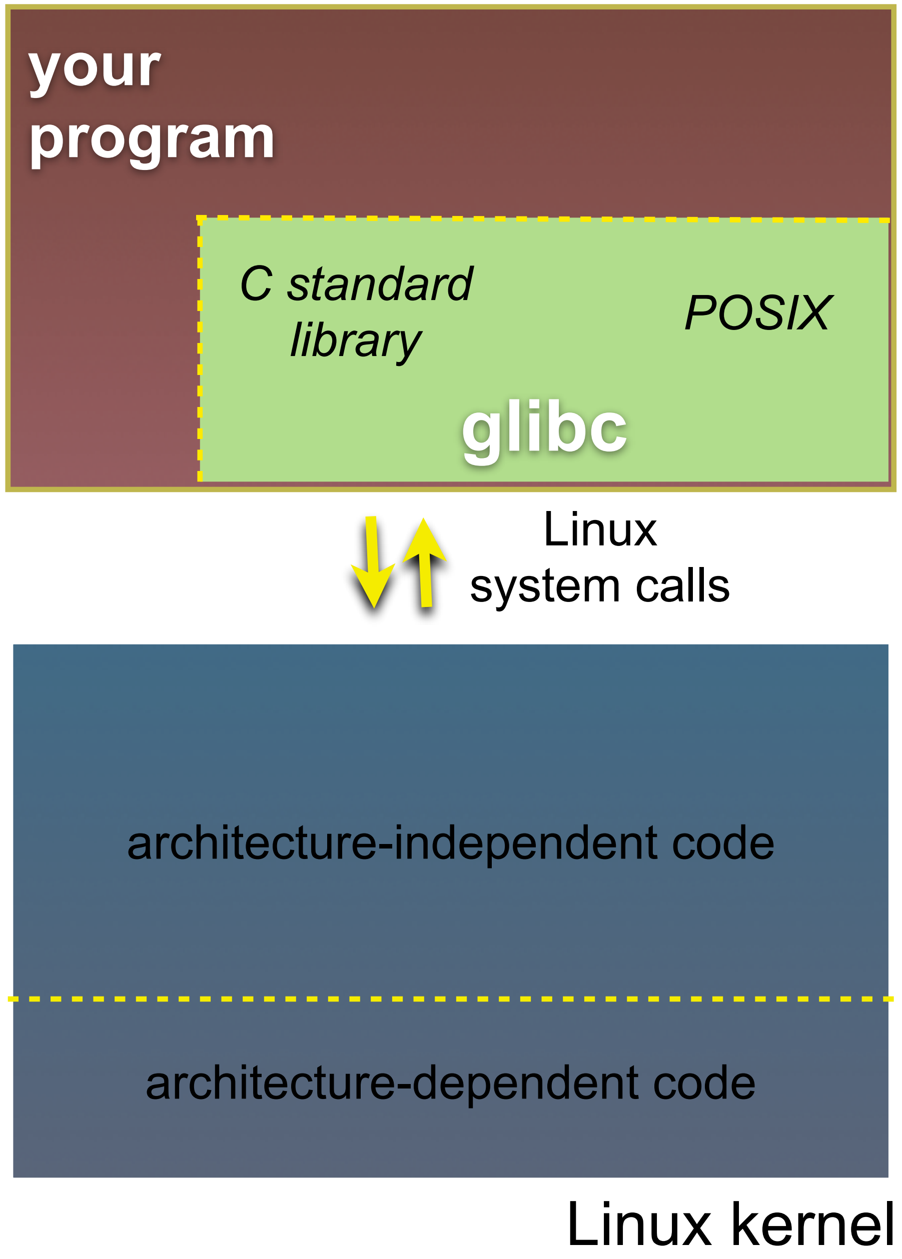
Figure 3-7
System service exceptions

System Call Handling

Procedure call in user process	
Initial work in user mode	*(libc)*
Trap instruction to invoke kernel	*(int 0x80)*
Preparation	*(e.g., sys_read, mmap2)*
I/O command	*(read from disk)*
Wait	*(disk is slow)*
Completion	*(interrupt handling)*
Return-from-interrupt instruction	
Final work in user mode	*(libc)*
Ordinary return instruction	

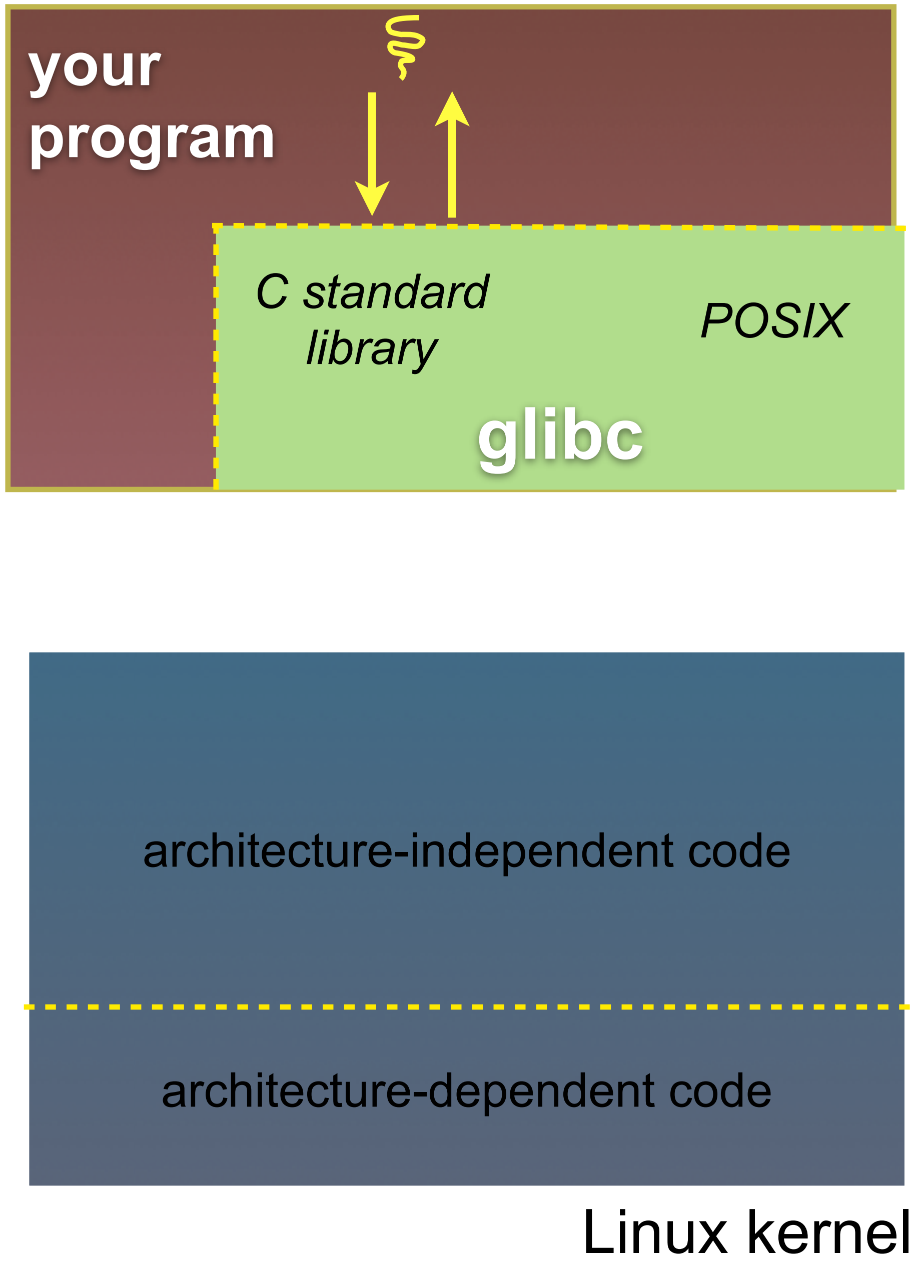
Details on x86 / Linux

- A more accurate picture:
  - consider a typical Linux process
  - its thread of execution can be several places
    - in your program's code
    - in **glibc**, a shared library containing the C standard library, POSIX support, and more
    - in the Linux architecture-independent code
    - in Linux x86-32/x86-64 code



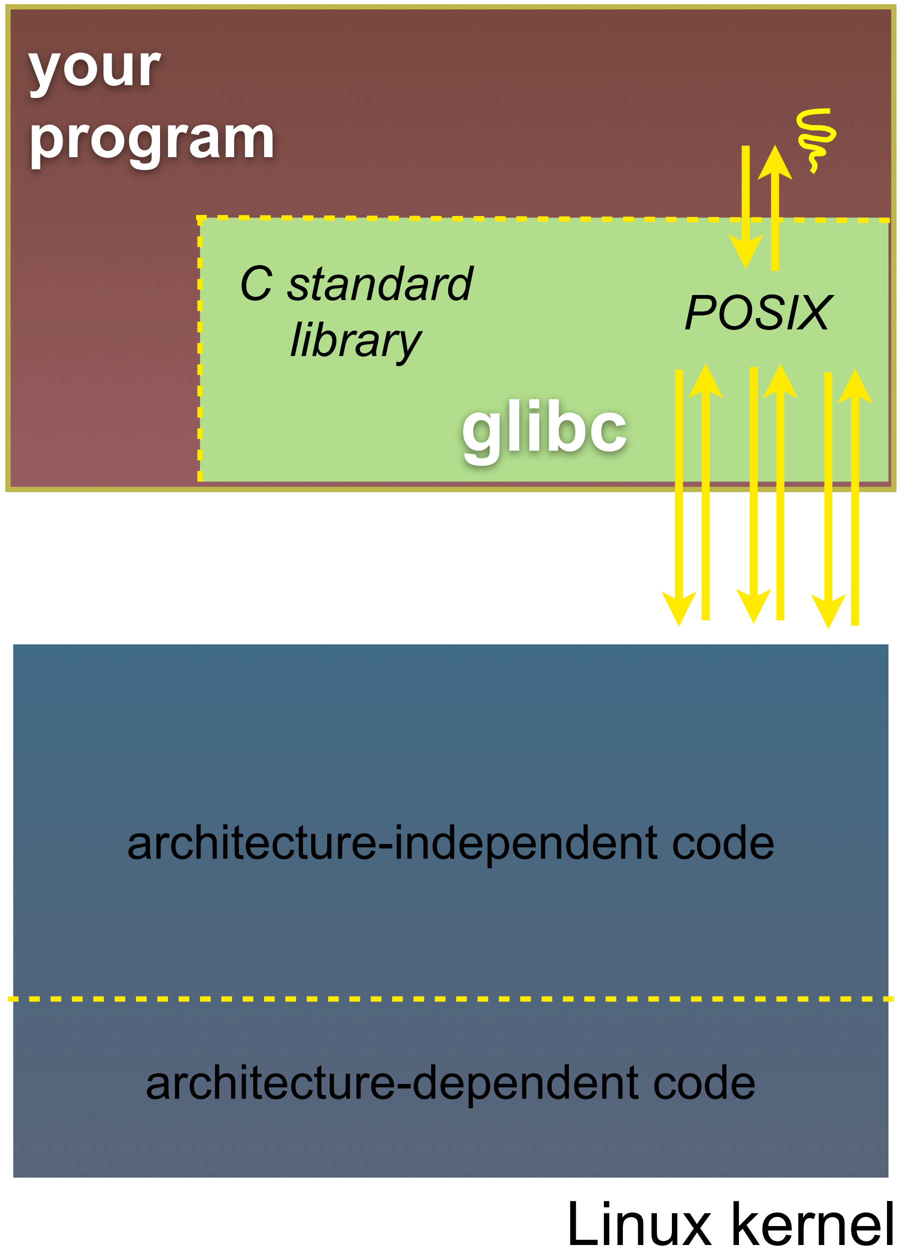
Details on x86 / Linux

- Some routines your program invokes may be entirely handled by glibc
  - without involving the kernel
    - e.g., **strcmp( )** from stdio.h
  - ∃ 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



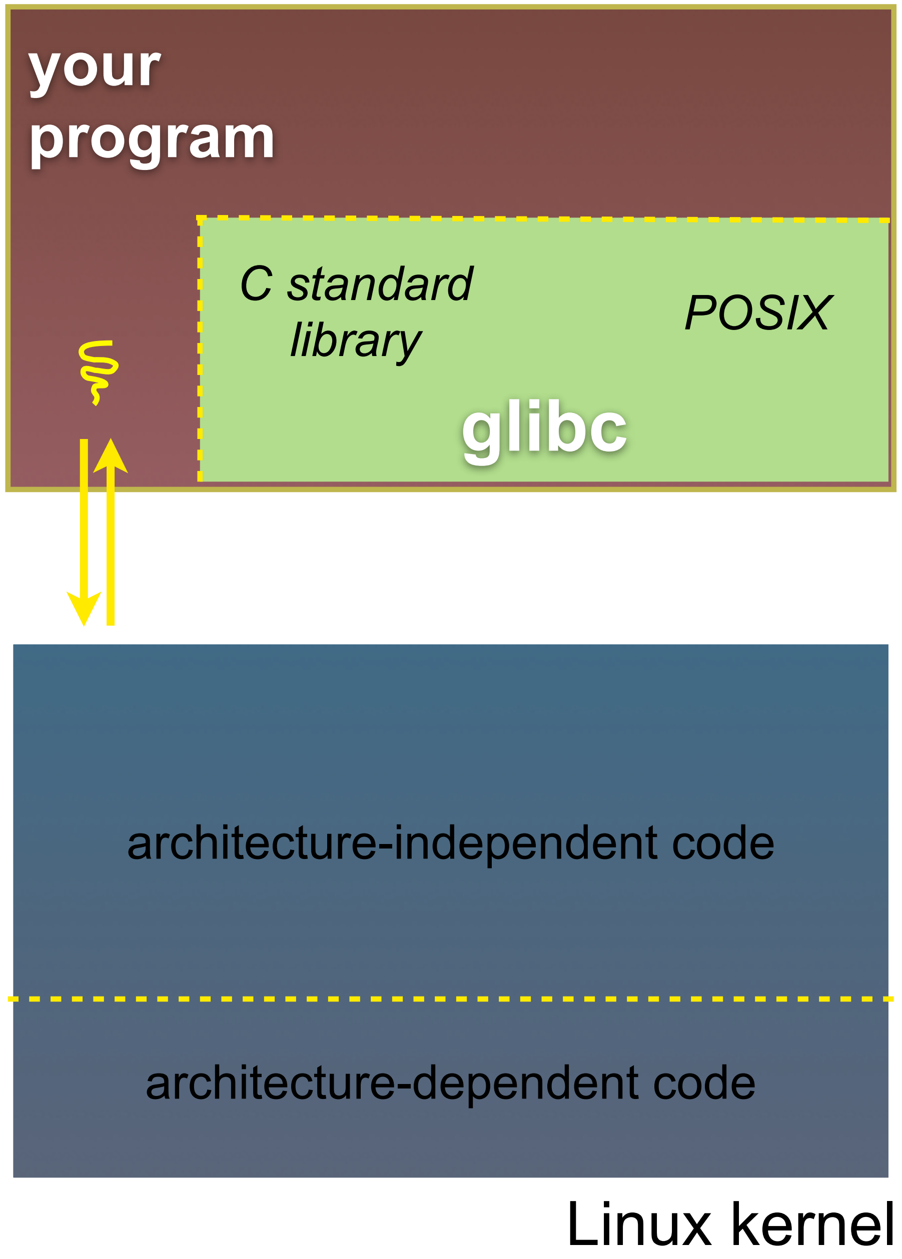
Details on x86 / Linux

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



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



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, etc.
  - And 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
  - 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 (do nothing)
  - /dev/cdrom (CD-drive)
- Use file system operations, but are handled in device-specific ways
  - `open`, `read`, `write` correspond to device-specific functions
    - 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
- 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, etc.
- Send a message to other (or all) processes: `signal()`
- Most of these in section 2 of manual
  - e.g., `man 2 select`

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, >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?

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
- Project 1 out