Operating Systems Introduction

Computing Devices Everywhere

What are Operating Systems?

Why do we have them, in the first place?

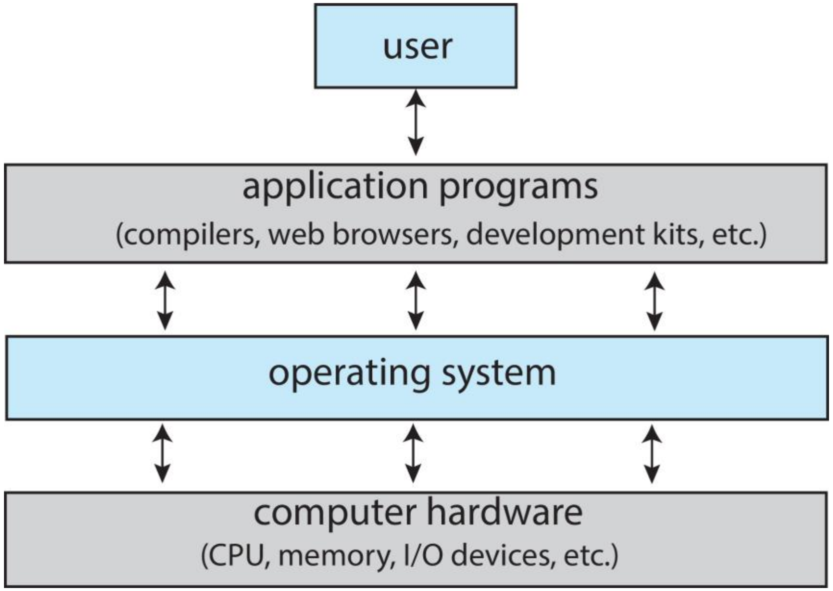
Why are they important?

What do they do for us?

Operating Systems to Help

- Helping you perform complex operations
  - Using various hardware and probably various bits of software
  - Developing programs for the computers
- While hiding the complexity
- And making sure nothing gets in the way of anything else

What is an Operating System Anyway?


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

- System software intended to provide support for higher level applications
  - Some higher-level system software
  - But primarily for user processes
- The interface that sits between the hardware and everything else
- The software that hides nasty details
  - Of hardware, software, and common tasks
- Implements a virtual machine (abstraction) that is (hopefully) **easier and safer to program and use** than the raw hardware

What Does an OS Do?

- It manages hardware for programs
  - Allocates hardware and manages its use
  - Enforces controlled sharing (and privacy)
  - Oversees execution and handles problems
- It abstracts (virtualizes) the hardware
  - Makes it easier to use and improves software portability
  - Optimizes performance
  - It provides new and powerful abstractions beyond the bare hardware

Hardware Abstraction in OS

- Each process thinks it has all memory and CPU time
- Each process thinks it owns all devices
- Different Devices appear to have same interface
- Device Interfaces more powerful than raw hardware
  - Bitmapped display ⇒ windowing system
  - Ethernet card ⇒ reliable, ordered, networking (TCP/IP)
- Provide protection and portability
- Fault Isolation
  - Processes unable to directly impact other processes
  - Bugs cannot crash whole machine

Operating System Definition

- No universally accepted definition
  - It is low level software . . .
  - That provides better, more usable abstractions of the hardware below it
  - To allow easy, safe, fair use and sharing of those resources
- Generally speaking, OS consists of the **kernel, middleware frameworks** and **system programs**
  - "The one program running at all times on the computer" is the **kernel.**
  - **Middleware** provides additional services to APP developers.
  - It may also include **system programs,** programs associated (shipped) with OS
    - Programs not directly associative with OS are called **application programs**
  - Is *web browser* a system or application program?

System Boot

- Operating system must be made available to hardware so hardware can start it
- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution
- Kernel loads and system is then **running**

Why Are You Studying Them?

- High probability none of you will ever write an operating system
  - Or even fix an operating system bug
- So why should you care about them?
- You will work with operating systems
  - Practically all computing devices you will ever use has an operating system
  - You rely on *services* that they offer
  - You use their features when you write programs

Another Good Reason

Many hard problems have been tackled in the context of operating systems

- How to coordinate separate computations
- How to manage shared resources
- How to virtualize hardware and software
- How to organize communications
- How to protect your computing resources

The operating system solutions are often applicable to programs and systems you write.

General OS Trends

- The operating system makes the computer work - it is a key abstraction layer for applications
  - How to get the OS to give users an illusion of infinite memory, CPUs, resources, world-wide computing, etc.
- They have grown larger and more sophisticated
- As systems change the OS must adapt (e.g., new hardware, software).
- Their role has fundamentally changed
  - From shepherding the use of the hardware
  - To shielding the applications from the hardware
  - To providing powerful application computing platform
  - To becoming a sophisticated “traffic cop”

Simple OS: What if only one application?

Examples:

- Very early computers
- Early PCs
- Embedded controllers (elevators, cars, …)

OS becomes just a library of standard services

- Standard device drivers
- Interrupt handlers
- Math libraries

More complex OS: Multiple Apps

- Full Coordination and Protection
  - Manage interactions between different users
  - Multiple programs running simultaneously
  - Multiplex and protect Hardware Resources
    - CPU, Memory, I/O devices like disks, printers, etc
- Facilitator
  - Still provides Standard libraries, facilities
- Next we look a little at the history of OS

Moore’s Law Change Drives OS Change

	1981	2006	Factor
CPU MHz, Cycles/inst	10 3—10	3200x4 0.25—0.5	1,280 6—40
DRAM capacity	128KB	4GB	32,768
Disk capacity	10MB	1TB	100,000
Net bandwidth	9600 b/s	1 Gb/s	110,000
# addr bits	16	32	2
#users/machine	10s	≤ 1	≤ 0.1
Price	$25,000	$4,000	0.2

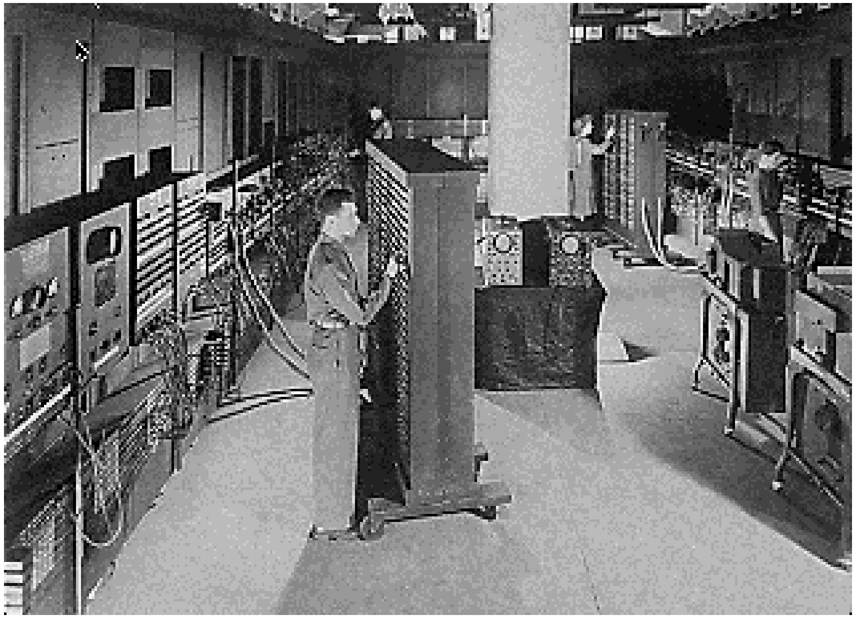
Typical academic computer 1981 vs. 2006

Moore’s law effects

- Nothing like this in any other area of business
- Transportation in over 200 years:
  - 2 orders of magnitude from horseback @10mph to Concorde @1000mph
  - Computers do this every decade!
- What does this mean for us?
  - Techniques have to vary over time to adapt to changing tradeoffs
- We place a lot more emphasis on principles
  - The key concepts underlying computer systems
  - Less emphasis on facts that are likely to change over the next few years…
- Let’s examine the way changes in $/MIPS has radically changed how OS’s work

Dawn of time
ENIAC: (1945—1955)

- “The machine designed by Drs. Eckert and Mauchly was a monstrosity. When it was finished, the ENIAC filled an entire room, weighed thirty tons, and consumed two hundred kilowatts of power.”
- http://ei.cs.vt.edu/~history/ENIAC.Richey.HTML



History Phase 1 (1948—1970)
Hardware Expensive, Humans Cheap

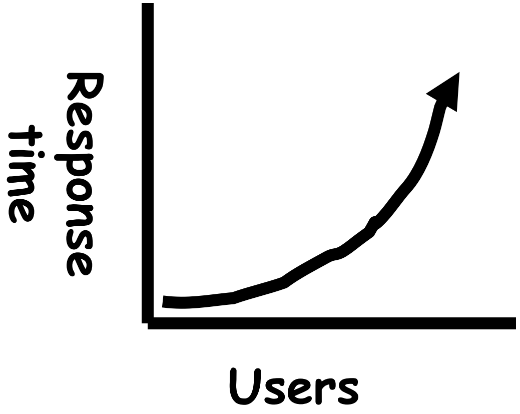
- When computers cost millions of $’s, optimize for more efficient use of the hardware!
  - Lack of interaction between user and computer
- User at console: one user at a time
- Batch monitor: load program, run, print
- Optimize to better use hardware
  - When user thinking at console, computer idle ⇒ BAD!
  - Feed computer batches and make users wait
- *No protection:* what if batch program has bugs?
- *Batch* is stilled used for running jobs in modern supercomputers.

History Phase 1½ (late 60s/early 70s)

- Data channels, Interrupts: overlap I/O and compute
  - DMA – Direct Memory Access for I/O devices
  - I/O can be completed asynchronously
- Multiprogramming: several programs run simultaneously
  - Small jobs not delayed by large jobs
  - More overlap between I/O and CPU
  - Need memory protection between programs and/or OS
- Complexity gets out of hand:
  - Multics: announced in 1963, ran in 1969
    - 1777 people "contributed to Multics" (30-40 core dev)
    - Turing award lecture from Fernando Corbató (key researcher): "On building systems that will fail"
  - OS 360: released with 1000 known bugs (APARs)
    - "Anomalous Program Activity Report"
- OS finally becomes an important science:
  - How to deal with complexity???
  - UNIX based on Multics, but vastly simplified

History Phase 2 (1970 – 1985) Hardware Cheaper, Humans Expensive

- Computers available for tens of thousands of dollars instead of millions
- OS Technology maturing/stabilizing
- *Interactive* timesharing:
  - Use cheap terminals (~$1000) to let multiple users interact with the system at the same time
  - Sacrifice CPU time to get better response time
  - Users do debugging, editing, and email online
- Problem: Thrashing
  - Performance very non-linear response with load
  - Thrashing caused by many factors including
    - Swapping, queuing



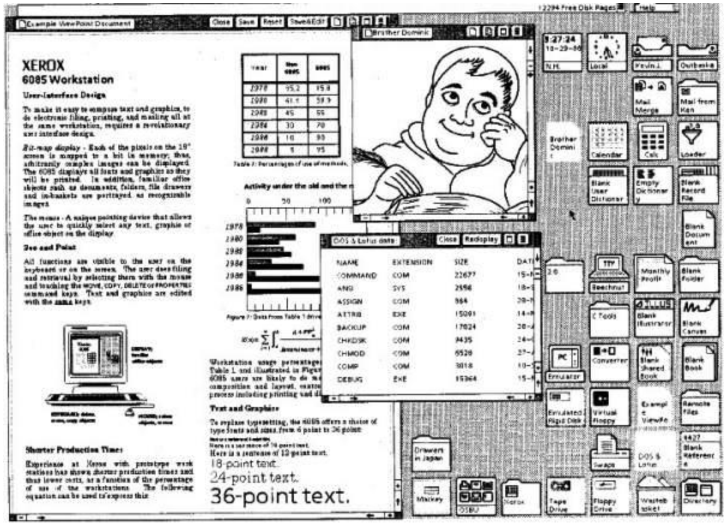
History Phase 3 (1981—)
Hardware Very Cheap, Humans Very Expensive

- Computer costs $1K, Programmer costs $100K/year
  - If you can make someone 1% more efficient by giving them a computer, it's worth it!
  - Use computers to make people more efficient
- Personal computing:
  - Computers cheap, so give everyone a PC
- Limited Hardware Resources Initially:
  - OS becomes a subroutine library
  - One application at a time (MSDOS, CP/M, …)
- Eventually PCs become powerful:
  - OS regains all the complexity of a "big" OS
  - multiprogramming, memory protection, etc (Windows NT,OS/2)
- Question: As hardware gets cheaper does need for OS go away?

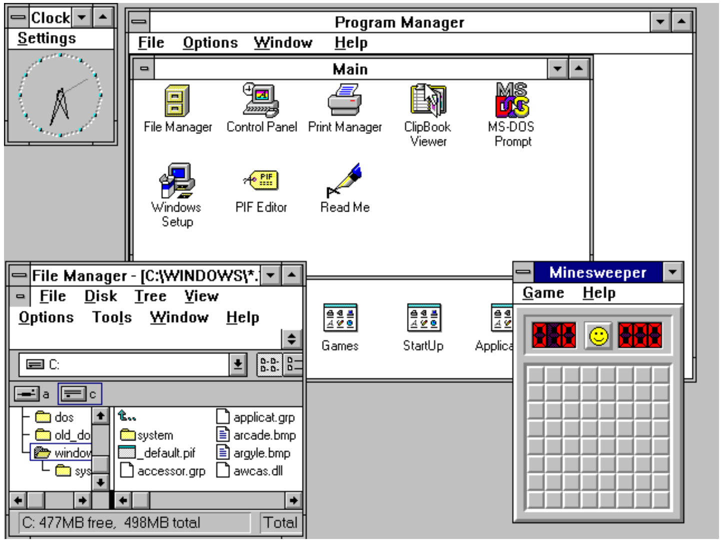
History Phase 3 (con’t)
Graphical User Interfaces

- Xerox Star: 1981
  - Originally a research project (Alto)
  - First “mice”, “windows”
- Apple Lisa/Machintosh: 1984
  - “Look and Feel” suit 1988
- Microsoft Windows:
  - Win 1.0 (1985)
  - Win 3.1 (1990)
  - Win 95 (1995)
  - Win NT (1993)
  - Win 2000 (2000)
  - Win XP (2001)
  - Win Vista (2007)
  - Win 7 (2009)
  - Win 10 (2015)
  - Win 11 (2021)

Xerox Star

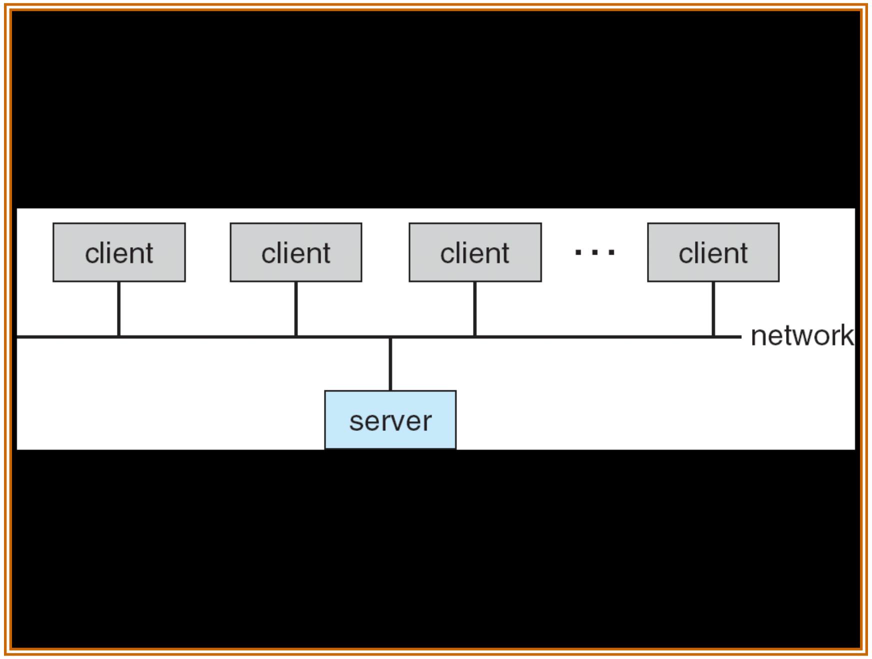


Windows 3.1



History Phase 4 (1989—): Distributed Systems

- Networking (Local Area Networking)
  - Different machines share resources
  - Printers, File Servers, Web Servers
  - Client – Server Model
- Services
  - Computing
  - File Storage



History Phase 5 (1995—): Mobile Systems

- Ubiquitous Mobile Devices
  - Laptops, smart phones, internet of things (IOT)
  - Small, portable, and inexpensive
  - Limited capabilities (memory, CPU, power, etc…)
- Wireless/Wide Area Networking
  - Leveraging the infrastructure
  - Huge distributed pool of resources extend devices
  - Traditional computers split into pieces. Wireless keyboards/mice, CPU distributed, storage remote
- Cloud Computing
  - Delivers computing, storage, even apps as a service across a network

History of OS: Summary

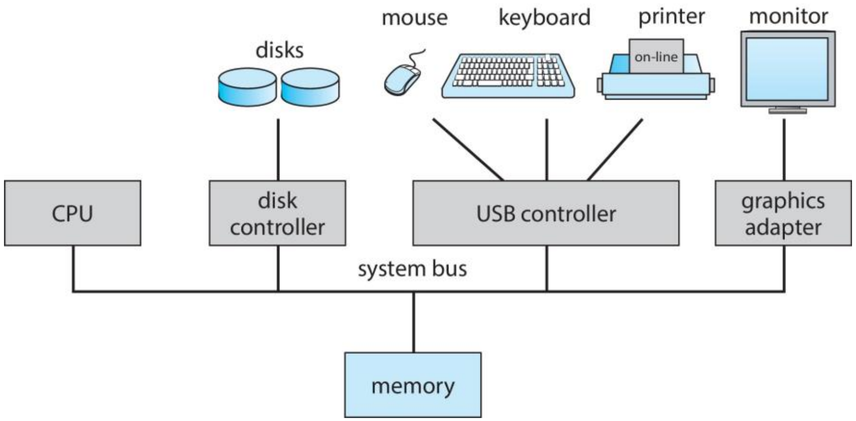
- Change is continuous and OSs should adapt
  - Not: look how stupid batch processing was
  - But: Made sense at the time
  - OS features migrated from mainframes ⇒ PCs ⇒ Mobile

Any thoughts about future OS?

- Situation today is much like the late 60s [poll]
  - Small OS: 100K lines
  - Large OS: >10M lines (5M for the browser!)
    - 100-1000 people-years
- Complexity still reigns
  - NT under development from early 90's to late 90's
    - Never worked very well
  - Windows XP has 45M lines of code,
  - New ones come along very rarely

Computer System Organization

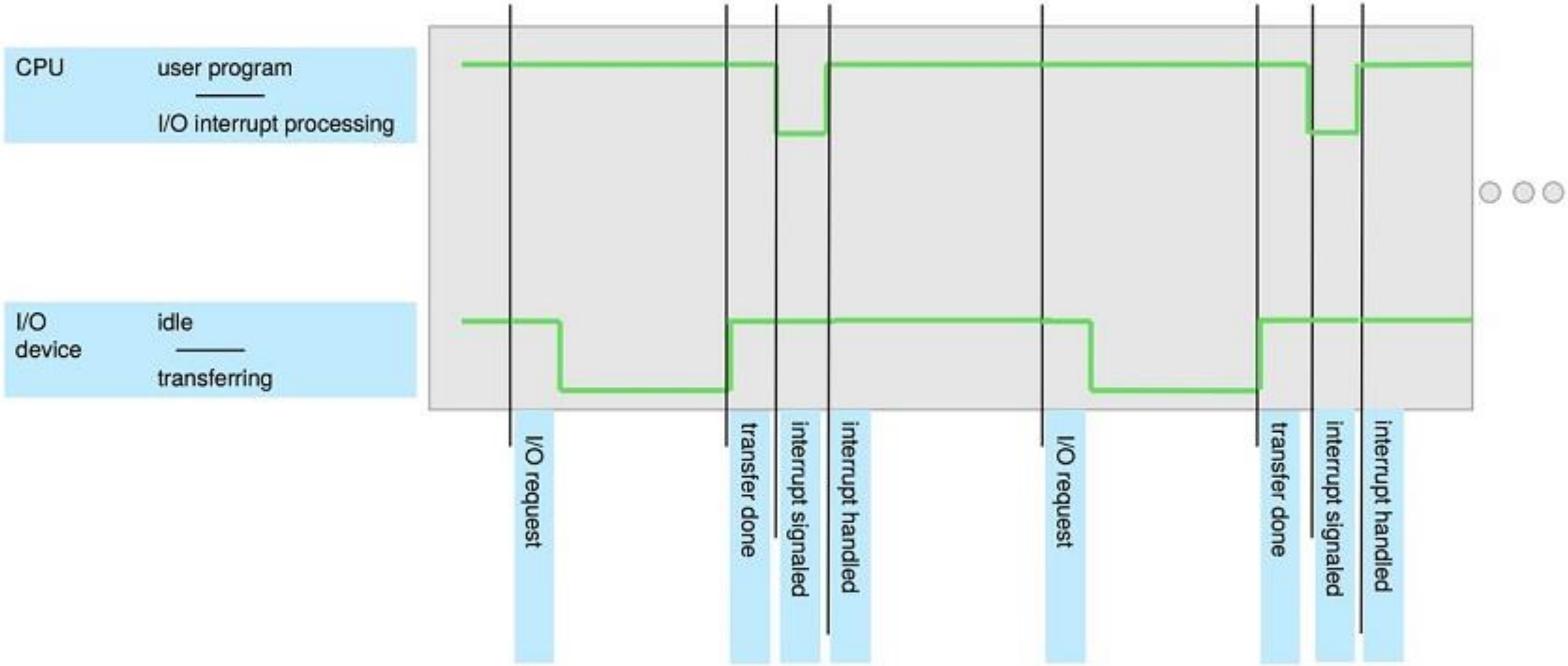
- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

Interrupt Timeline

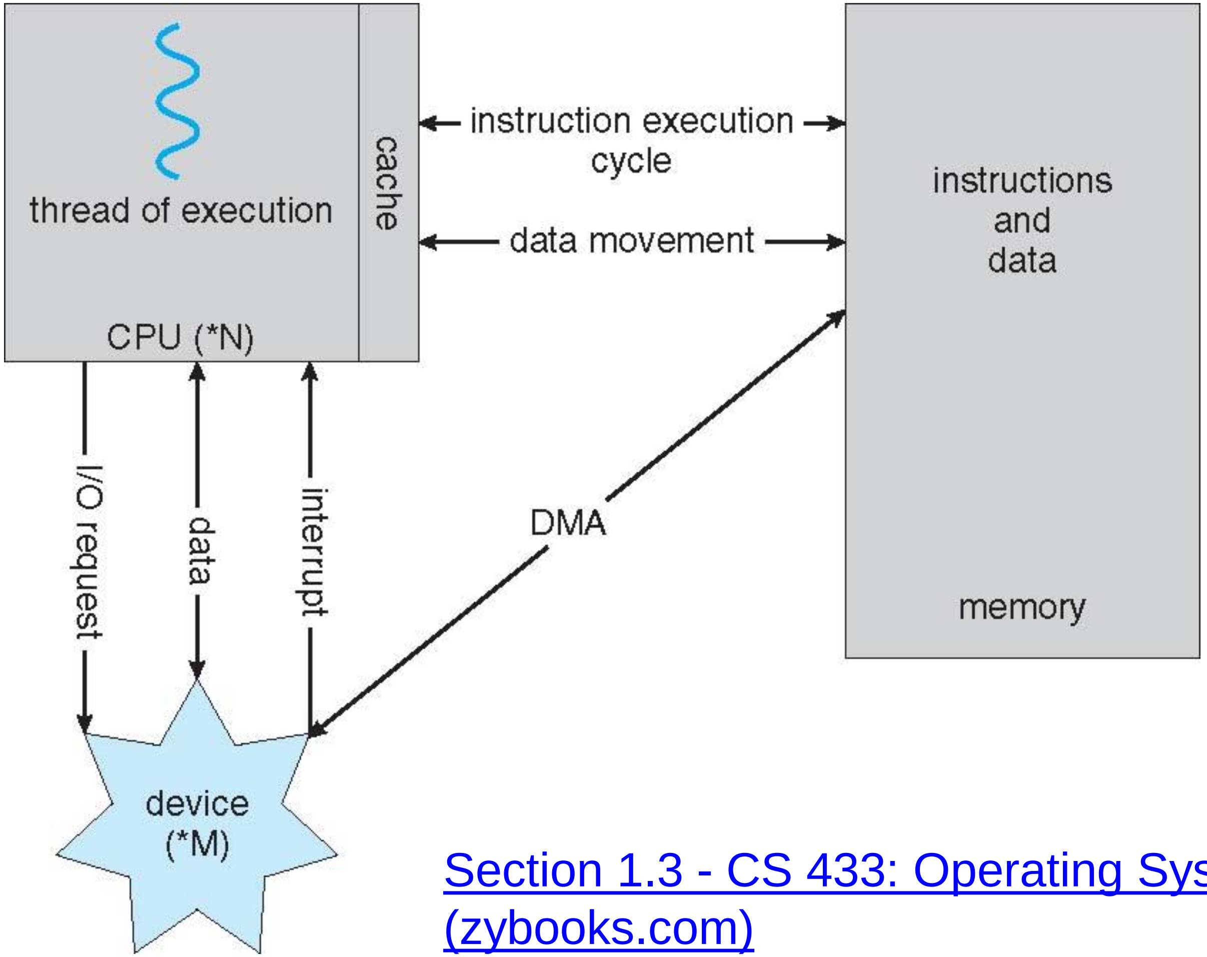


Interactive demo

Interrupt Driven O.S.

- Most modern operating systems are interrupt driven.
  - Start-up: 1) Load O.S. (kernel) and start it running.
  - 2) O.S. waits for an event (an interrupt).
- Definition of interrupt:
  - An **interrupt** is a method to ensure that the CPU takes note of an event.
- Types of interrupt:
  - Hardware interrupts (e.g. from an I/O device)
  - Software interrupt (from an executing process)
    - Also called a **trap** or an **exception**
    - Generated to signal error (e.g. divide by zero)
    - or to request service from OS (system calls for I/O requests).

How a Modern Computer Works



[Section 1.3 - CS 433: Operating Systems (zybooks.com)]

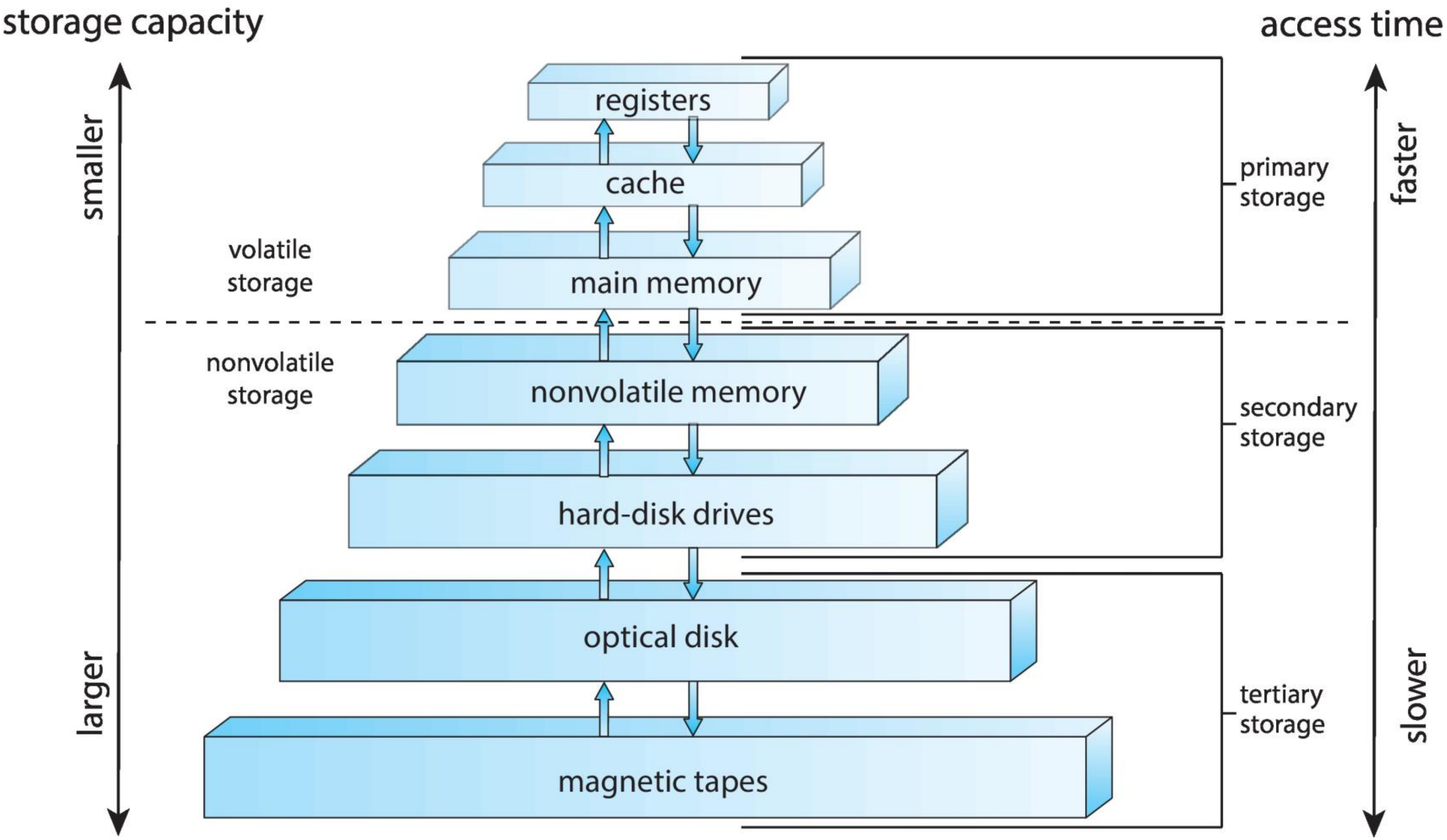
Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.
- The O.S. determines which type of interrupt has occurred.
  - Separate segments of code determine what action should be taken for each type of interrupt.
- The OS executes the sequence of commands associated with the given interrupt.
- The OS recovers the stored information from the original process and continues execution.

Storage Hierarchy

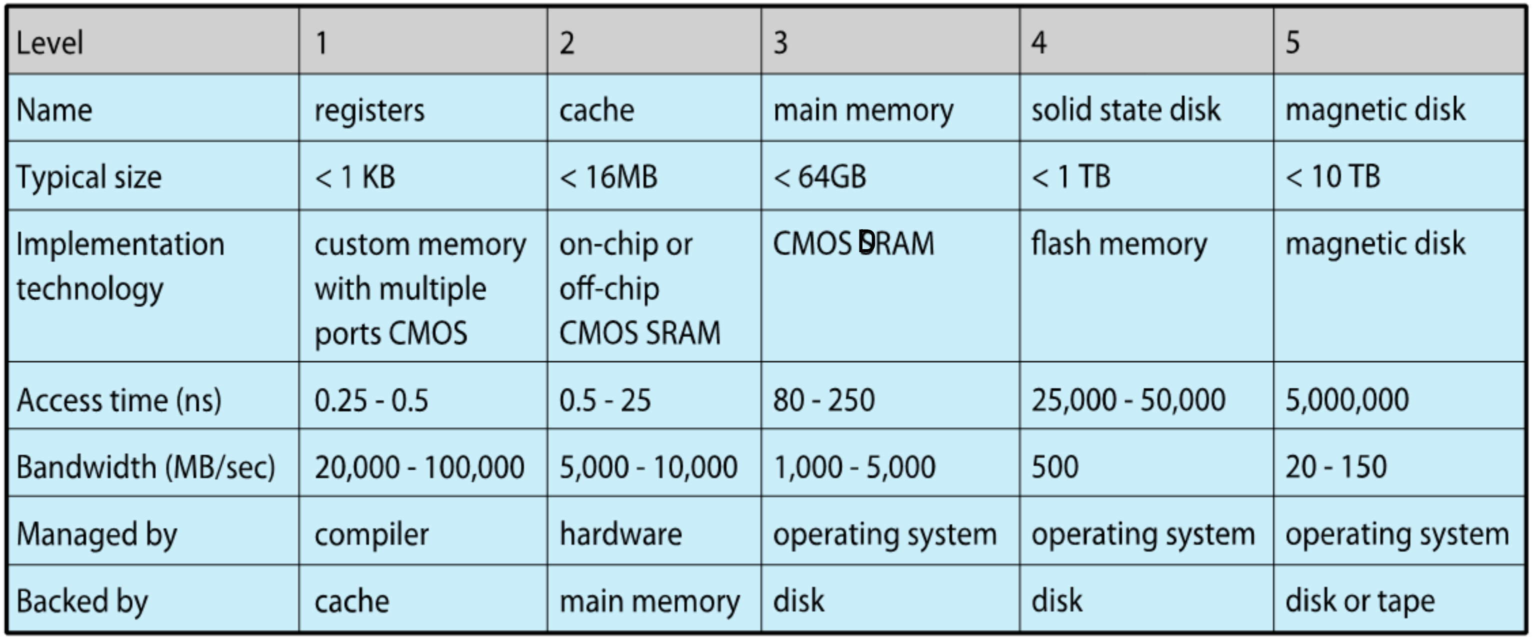
- Storage systems organized in hierarchy
  - Speed, Cost, Volatility
- **Main memory** – only large storage media that the CPU can access directly
- **Hard disks** – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Non-volatile memory (NVM)** devices– faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular as capacity and performance increases, price drops

Storage-Device Hierarchy



Performance of Various Levels of Storage

- **Caching** – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage
- Movement between levels of storage hierarchy can be explicit or implicit



Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

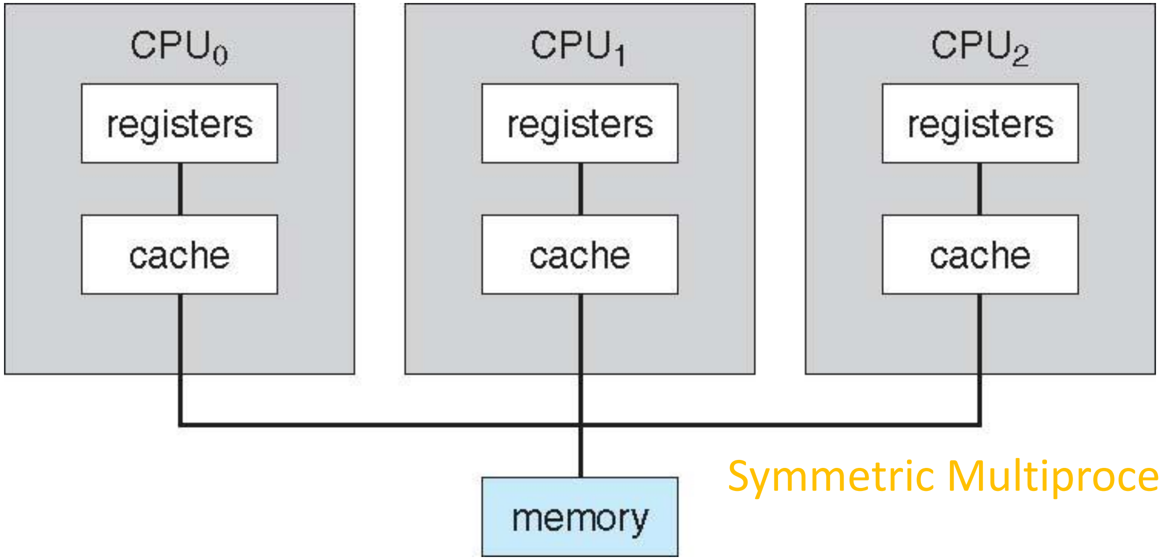
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Computer-System Architecture

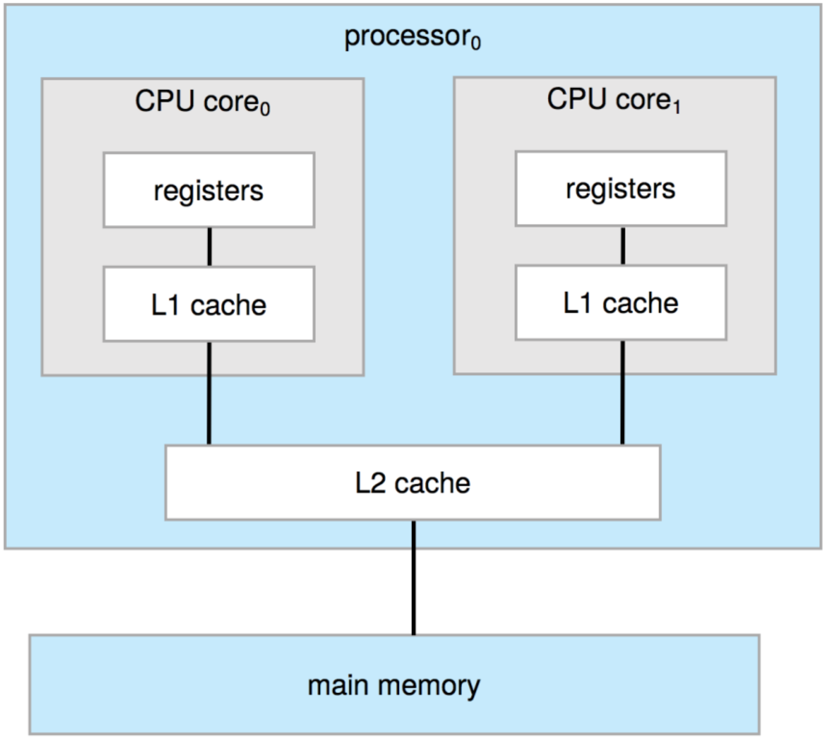
- Multiprocessors systems growing in use and importance
    1. Asymmetric Multiprocessing
    2. Symmetric Multiprocessing (SMP)

What is potential bottleneck in this architecture?



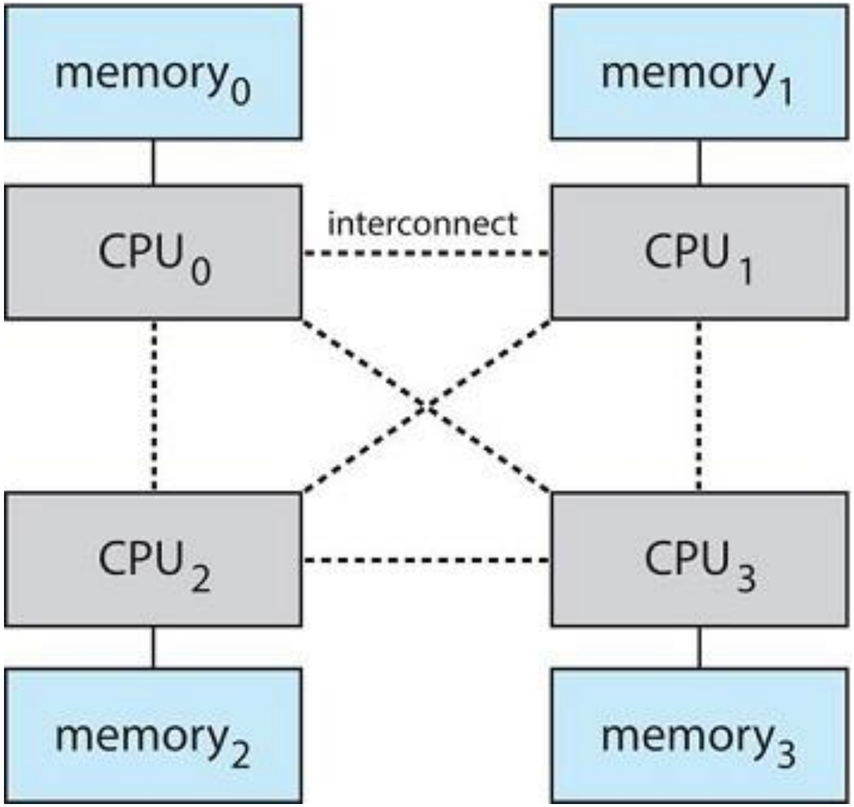
Symmetric Multiprocessing

A Dual-Core Design



- **Multicores** on a same processor chip
- A computer may have multi processor chips.

NUMA Architecture



- non-uniform memory access
  - Fast access to local memory, but slower access to remote memory across CPU modules

What are the pros and cons of the NUMA architecture?

Summary

- I/O devices and the CPU can execute concurrently
- Most modern operating systems are interrupt driven.
- Storages are organized into a hierarchy and caching is used to improve performance.
- Multiprocessor systems become common

Operating Systems Components (main pieces of the OS)

- Process Management
- Main-Memory Management
- File System Management
- I/O System management
- Protection and Security
- Networking
- User Interfaces

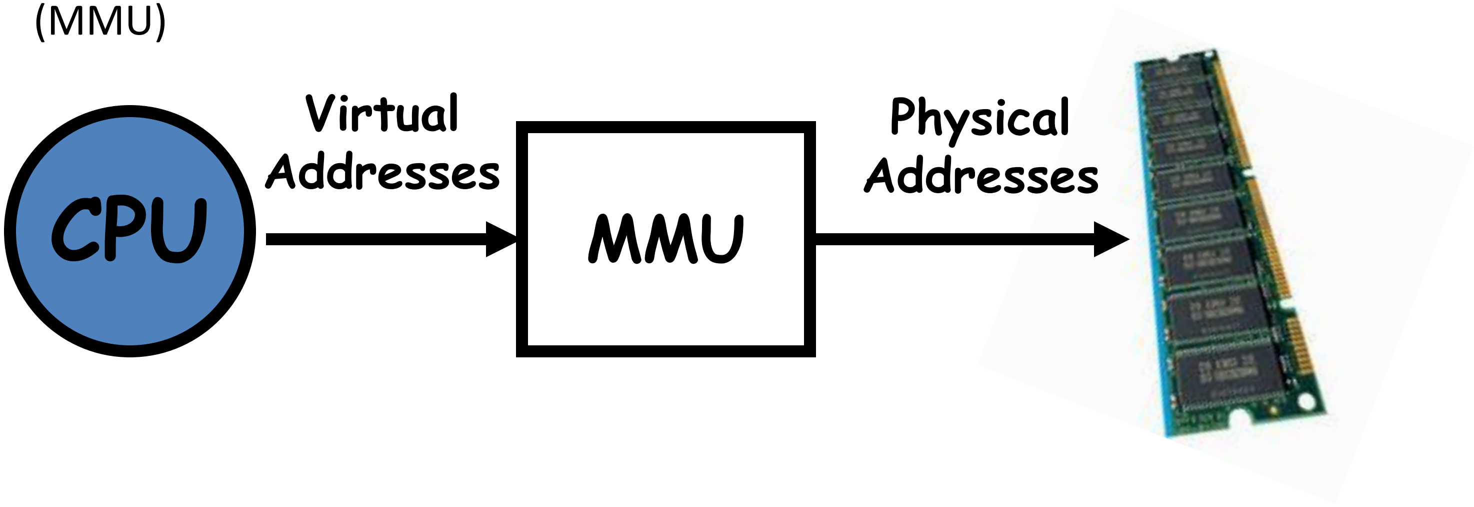
Example: Protecting Processes from Each Other

- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation

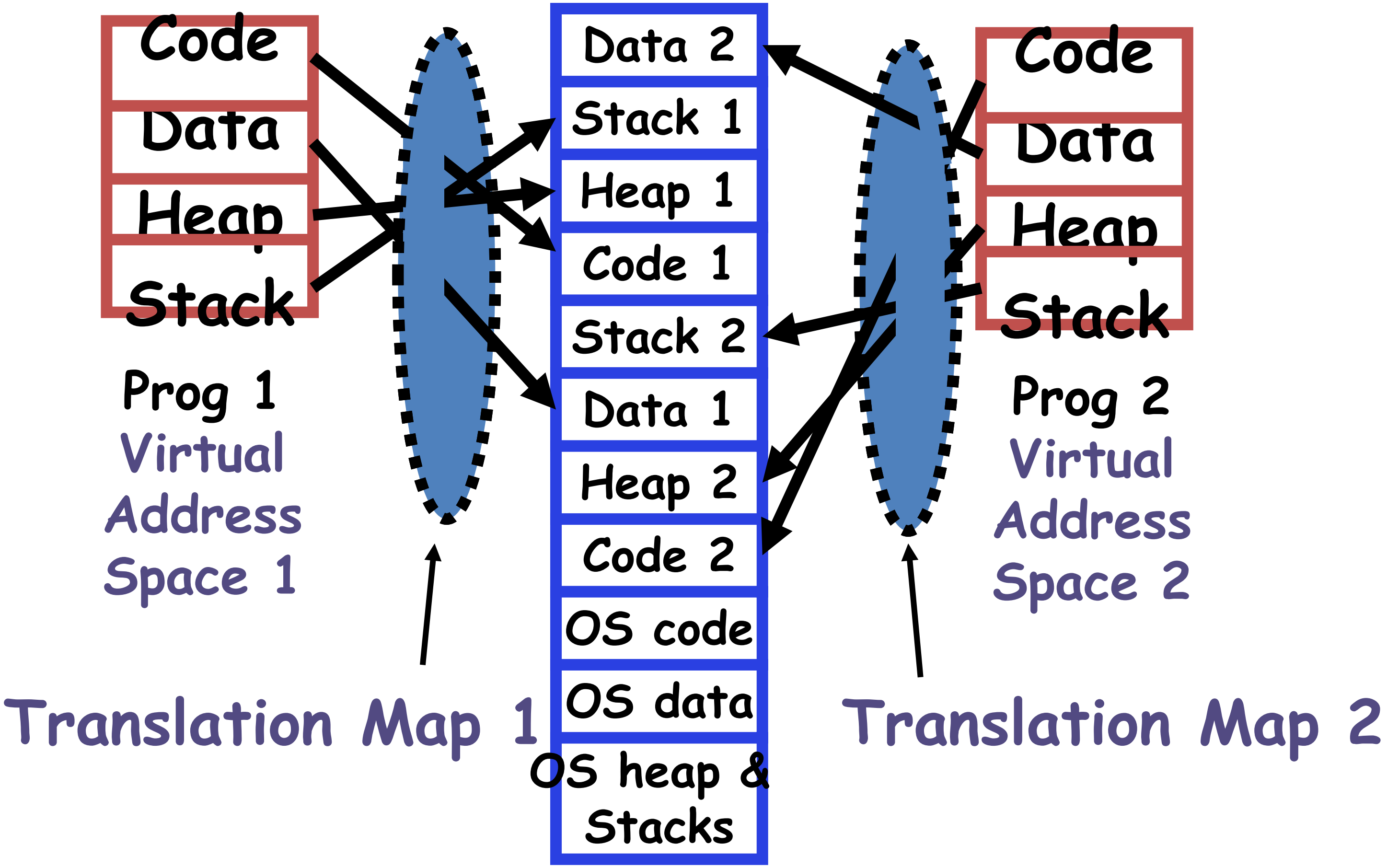
Run OSTEP mem demo

Address Translation

- Address Space
  - A group of memory addresses usable by something
  - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
  - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
  - Mapping *often* performed in Hardware by Memory Management Unit (MMU)



Example of Address Translation



Code
Data
Heap
Stack

Prog 1
Virtual
Address
Space 1

Data 2
Stack 1
Heap 1
Code 1
Stack 2
Data 1
Heap 2
Code 2
OS code
OS data
OS heap &
Stacks

Code
Data
Heap
Stack

Prog 2
Virtual
Address
Space 2

Translation Map 1

Translation Map 2

Physical Address Space

Operating-System Protections

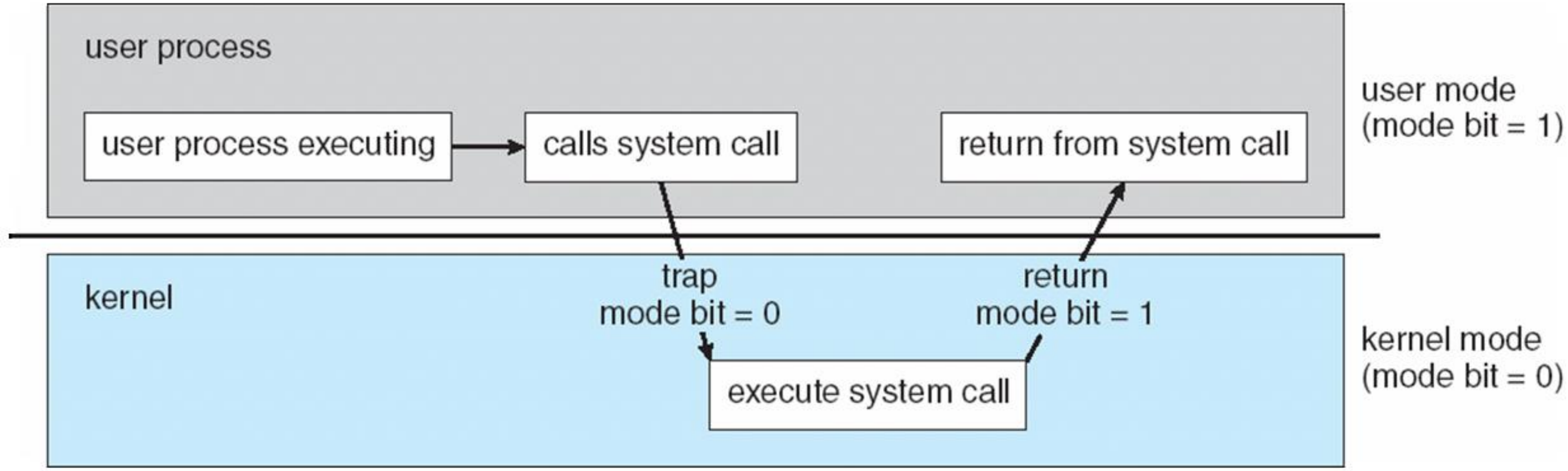
- Can a user process change the translation map?
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify translation map (page tables) in user mode
    - Attempt to modify ⇒ Exception generated
- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call (trap) changes mode to kernel, return from call resets it to user

Kernel Mode Instructions

- Which of following machine instructions must be run in kernel mode?
  A. Set value of timer
  B. Write to memory
  C. Modify a register
  D. Turn off interrupts
  E. Issue a trap instruction
  F. Access I/O devices directly

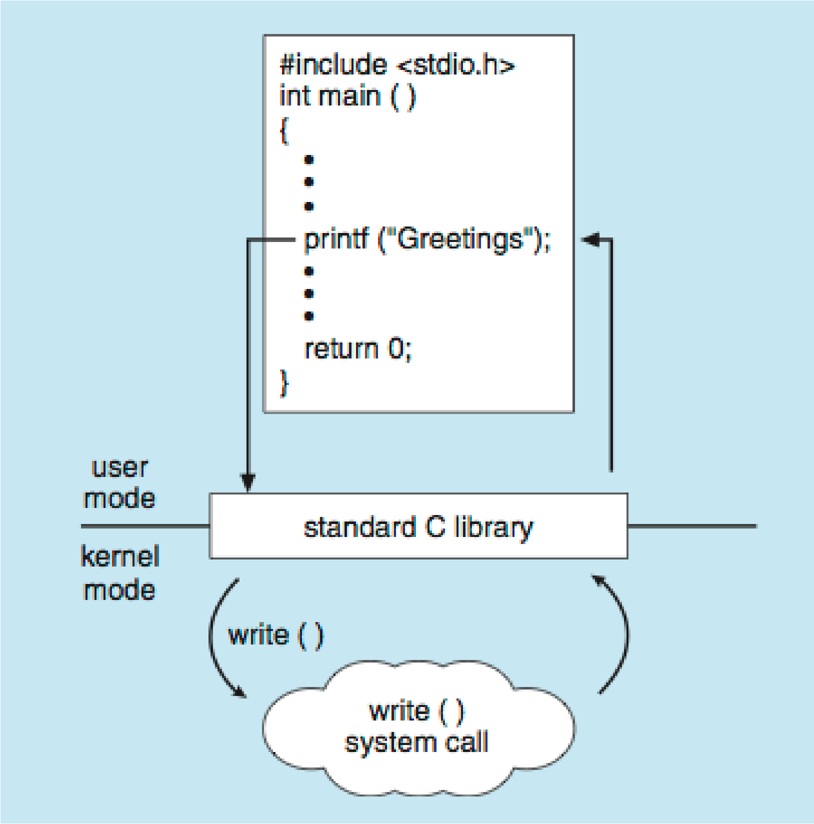
Dual Mode Operation

- Transitions from user mode to kernel mode:
  - Hardware interrupt (generated by hardware)
    - E.g. Timer to prevent infinite loop / process hogging resources
    - I/O device finished transmission
  - Software interrupt generated by program error or request, called **exception** or **trap**
    - Exceptions: Division by zero, …
    - System call: request for operating system service



Standard C Library Example

- C program invoking `printf()` library call, which calls `write()` system call

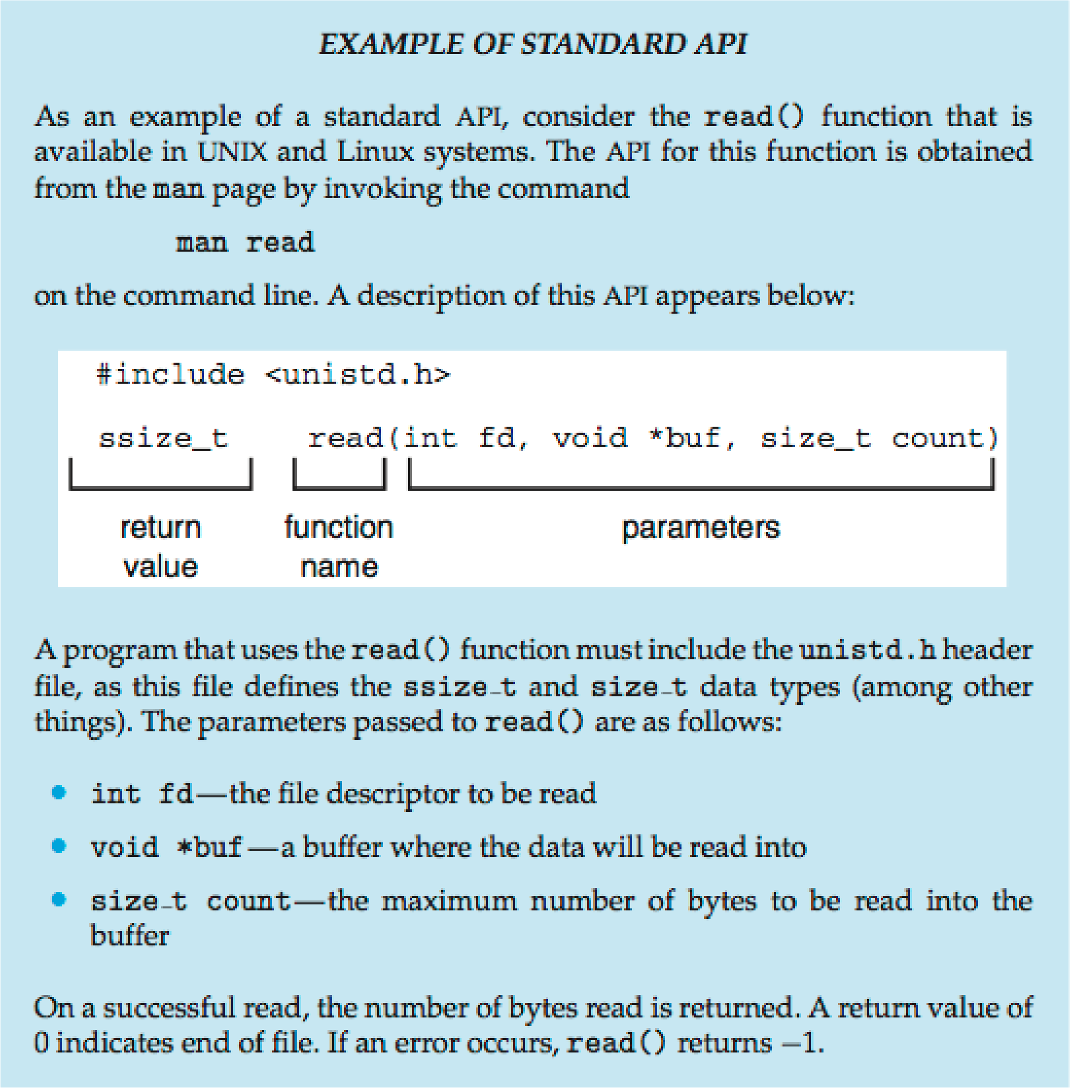


```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
  - Use printf() in standard C library instead of write()
- Why use high-level libraris rather than actual system calls?
  (Note that the system-call names used throughout this text are generic)
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of Standard API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad\underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

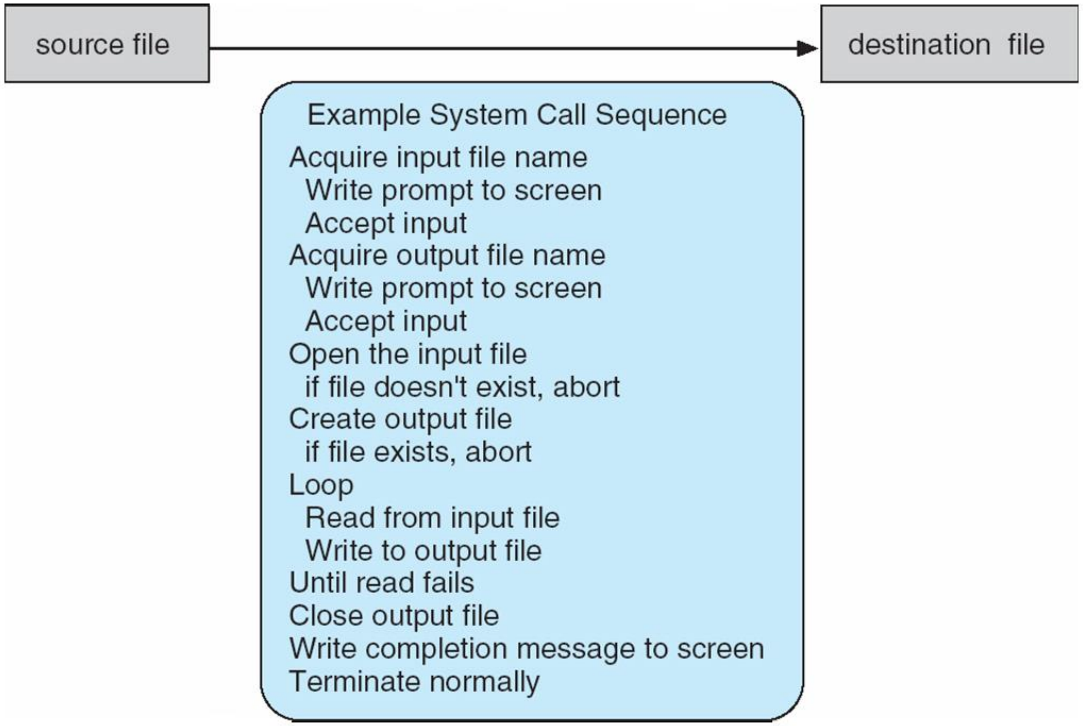
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Example of System Calls

- System call sequence to copy the contents of one file to another file, e.g. `cp in.txt out.txt`



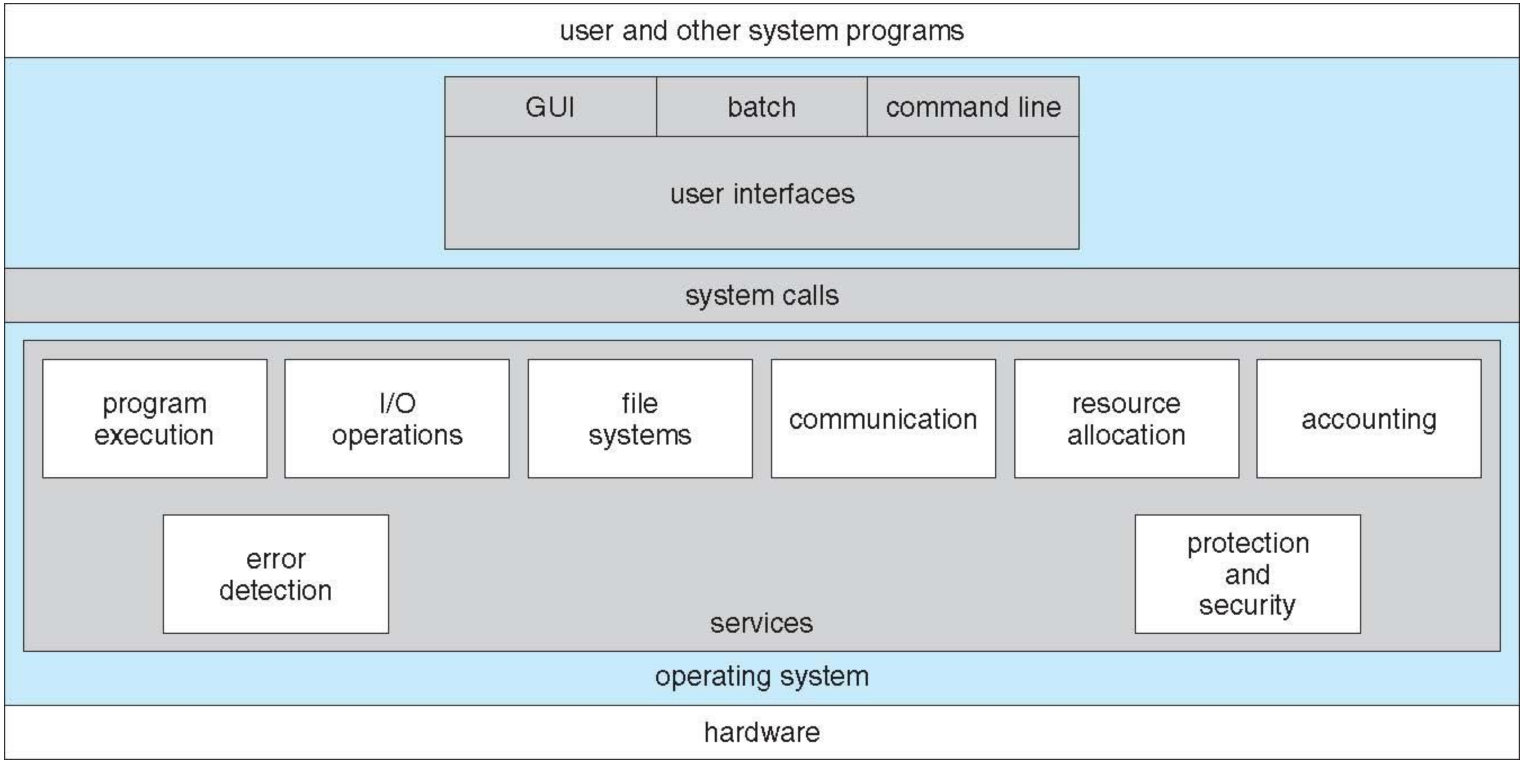
source file

destination file

Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

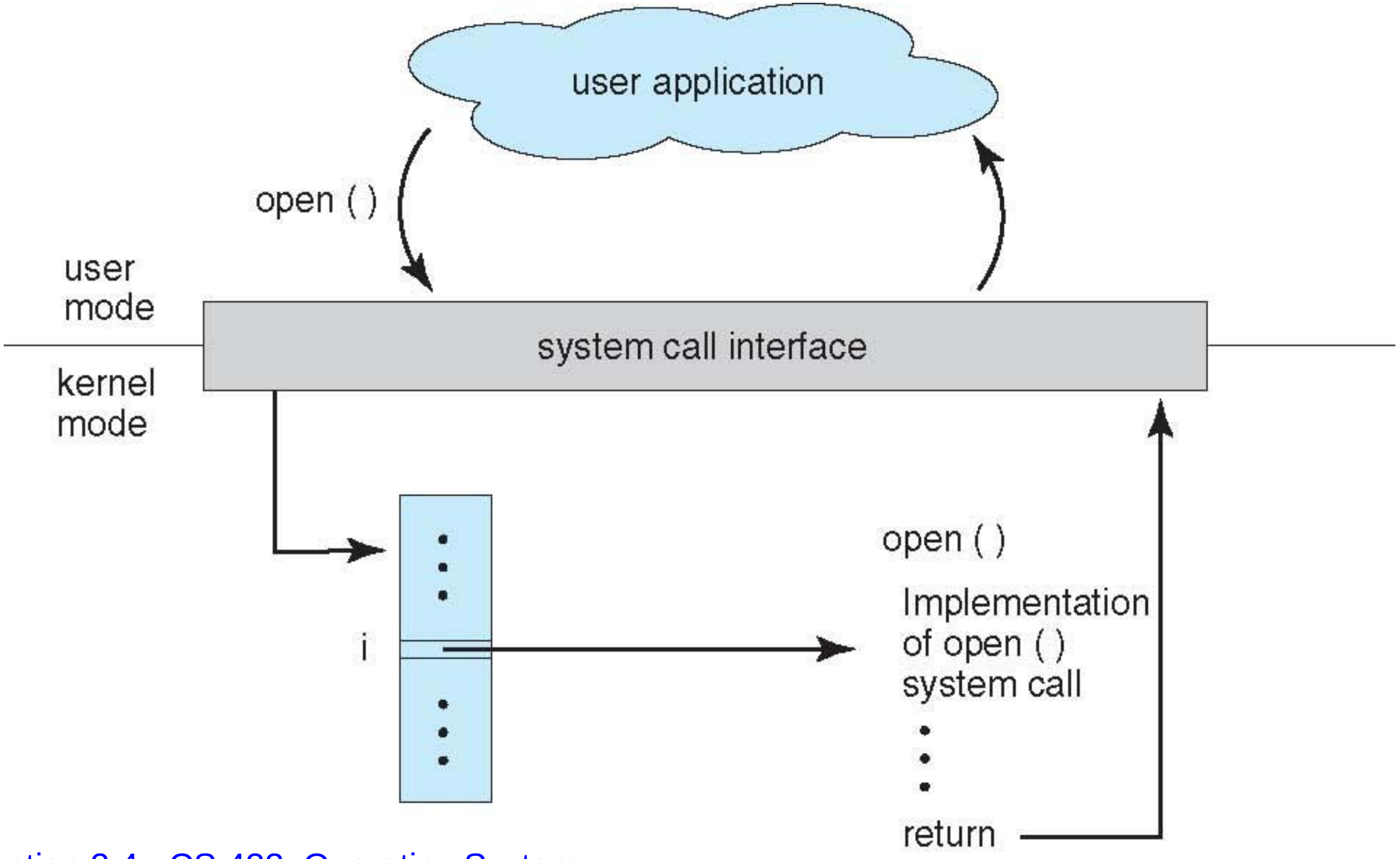
A View of Operating System Services



System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship

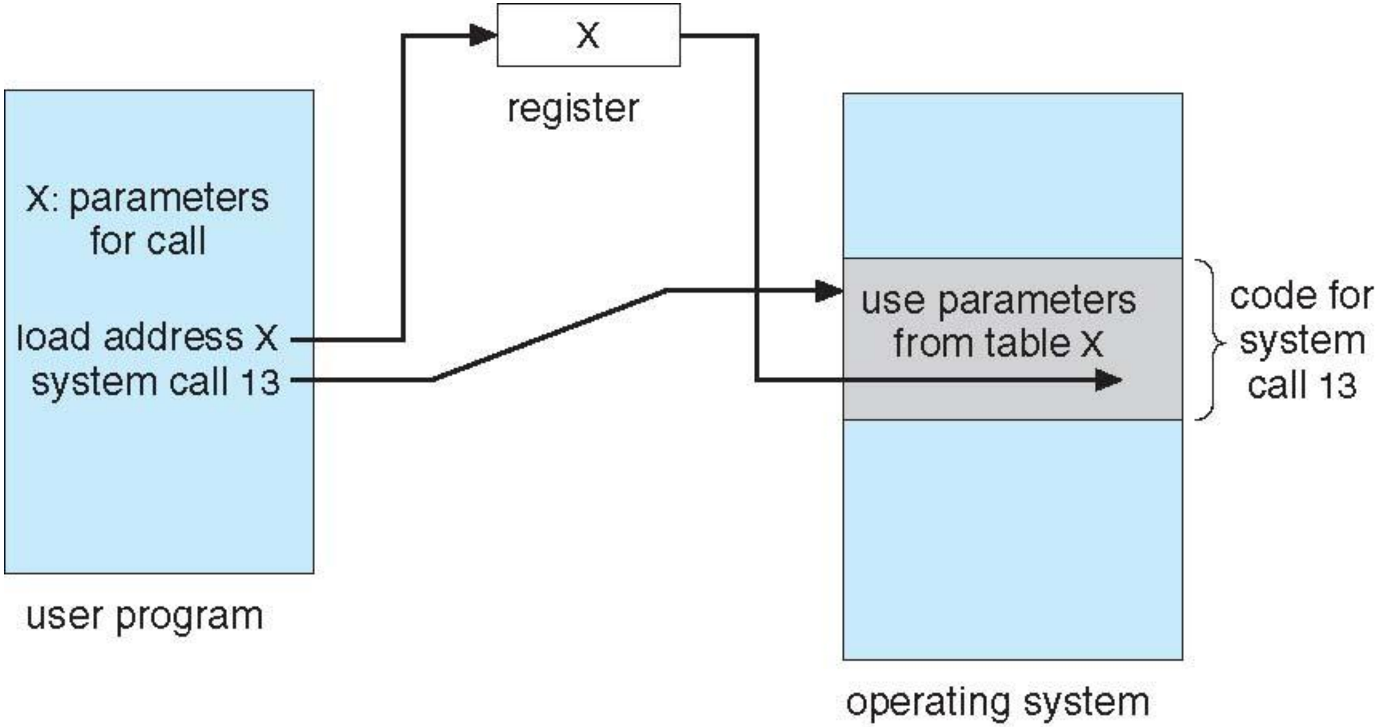


[Section 2.4 - CS 433: Operating Systems (zybooks.com)]

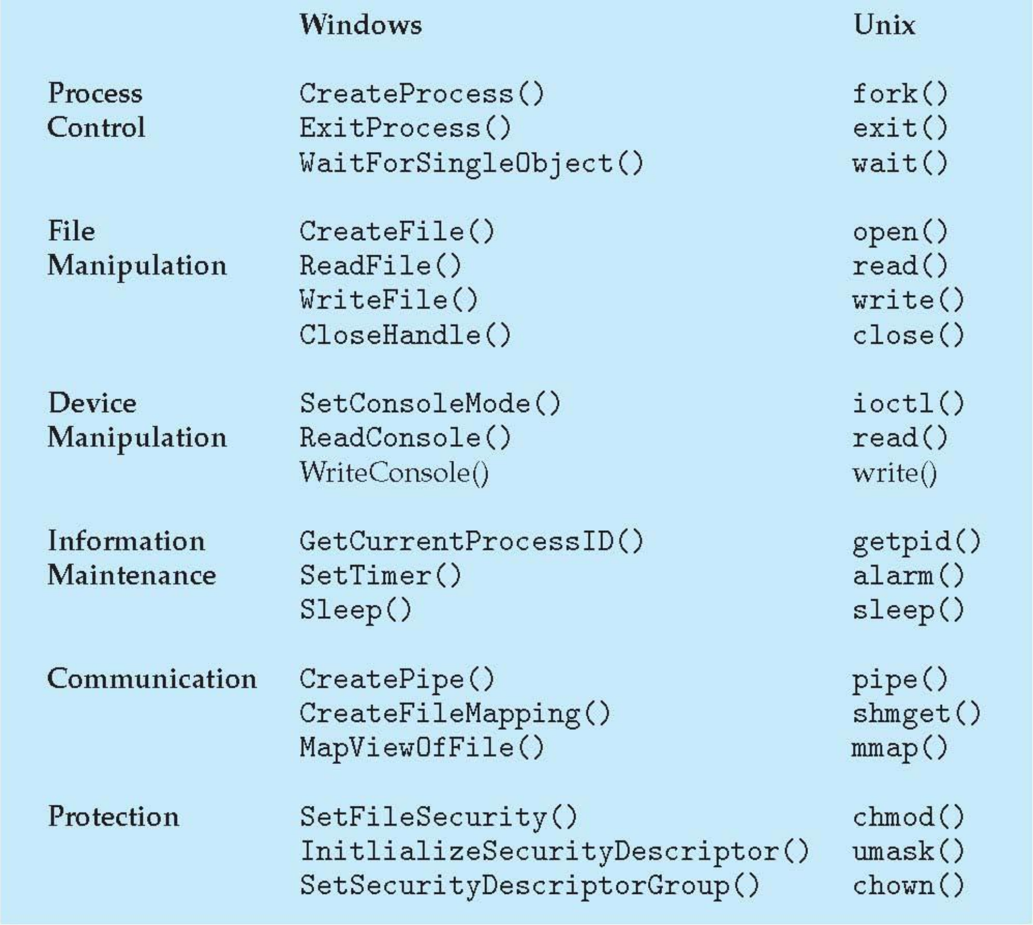
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

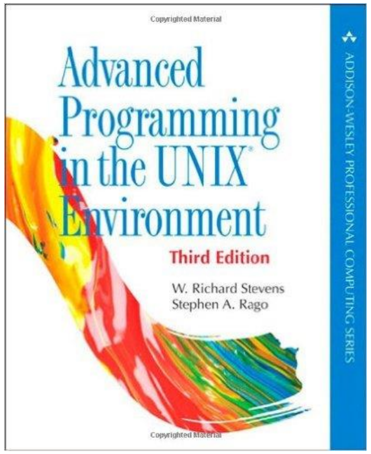
Parameter Passing via Table

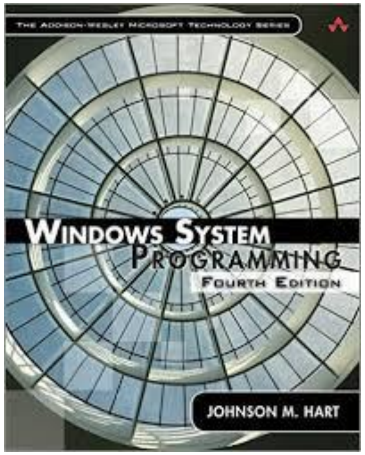


Examples of Windows and Unix System Calls



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()


Copyrighted Material
Advanced Programming in the UNIX® Environment
Third Edition
W. Richard Stevens
Stephen A. Rago
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Copyrighted Material


THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES
WINDOWS SYSTEM PROGRAMMING
FOURTH EDITION
JOHNSON M. HART

System Programs

What are the differences between system calls and system programs?

- System programs provide a convenient environment for program development and execution
  - File management
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
- Most users' view of the operation system is defined by system programs, not the actual system calls

OS Summary

- Operating systems simplify application development by providing standard services, which can be accessed through system call interface
- Operating systems coordinate resources and protect users from each other
- Operating systems can provide an array of fault containment, fault tolerance, and fault recovery
- Complexity is always out of control
  - However, “Resistance is NOT Useless!”

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- **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

Policy vs. Mechanism

- Important principle to separate

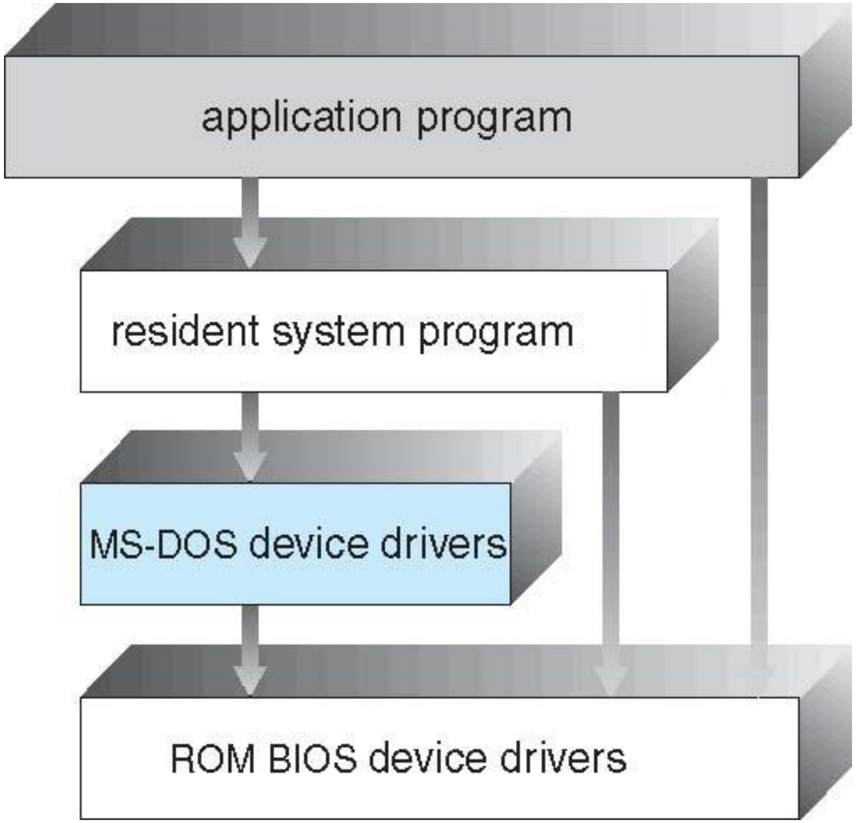
  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is highly creative task of **software engineering**

Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Simple Structure

- e.g MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

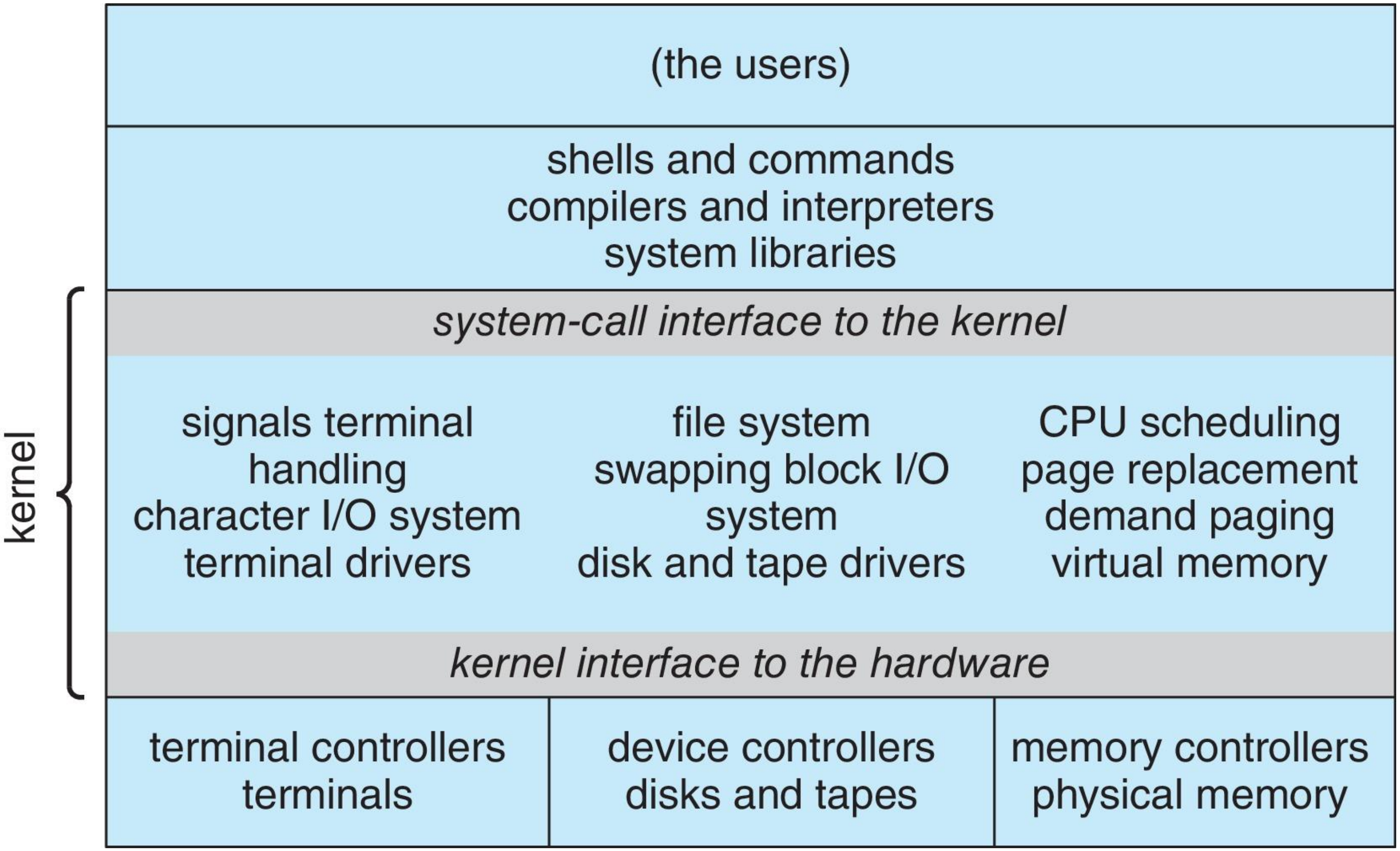


UNIX : Also "Simple" Structure

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
- Unix philosophy emphasizes building simple, short, clear, modular, and extensible code

Traditional UNIX System Structure

Beyond simple but not fully layered



(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

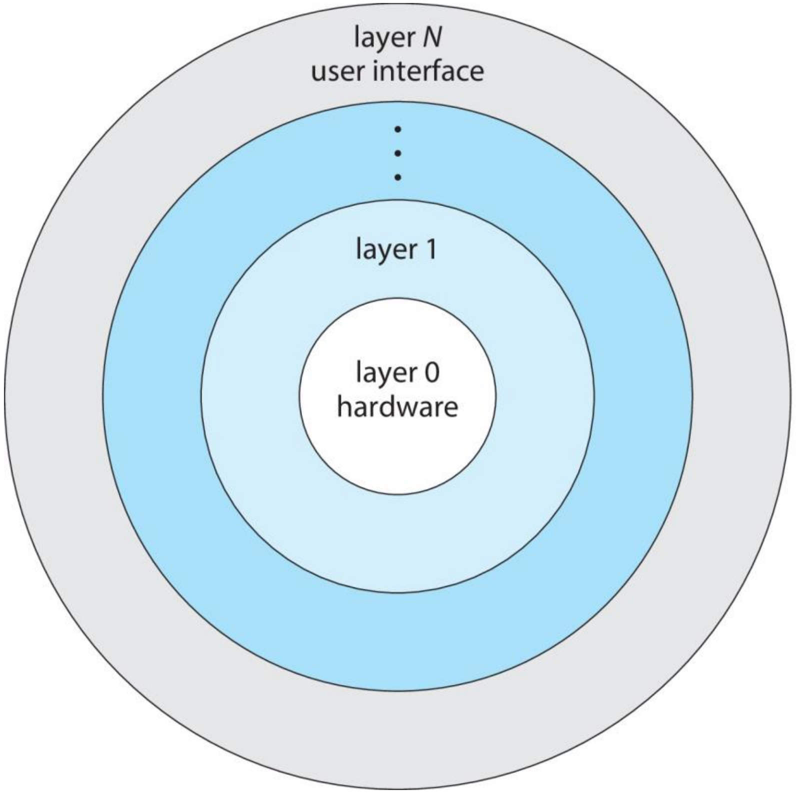
device controllers
disks and tapes

memory controllers
physical memory

kernel

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

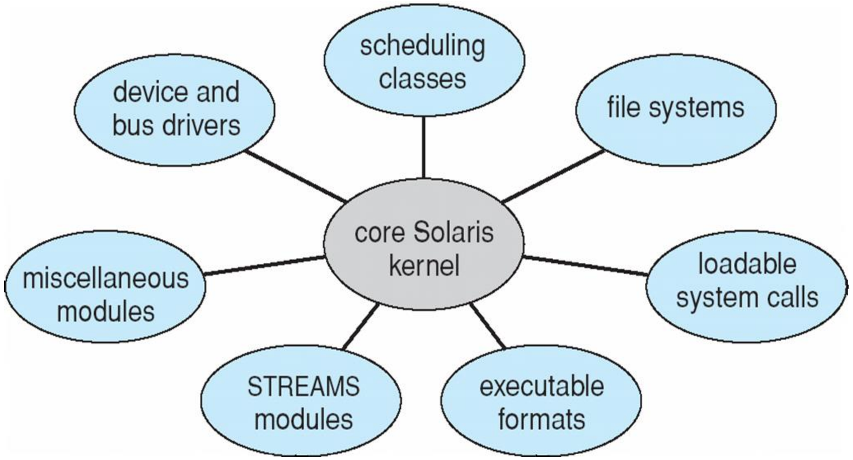


Layered Structure

- Operating system is divided many layers (levels)
  - Each built on top of lower layers
  - Bottom layer (layer 0) is hardware
  - Highest layer (layer N) is the user interface
- Each layer uses functions (operations) and services of only lower-level layers
  - Advantage: modularity ⇒ Easier debugging/Maintenance
    - Example: Machine-dependent vs. independent layers
    - Easier migration between platforms
    - Easier evolution of hardware platform
  - However, few OS uses pure layered structure. Why?
    - Overhead of layers.
    - Not always possible: some OS functionalities are tightly coupled.
      - For example: process scheduler and virtual memory manager.

Modules-based Structure

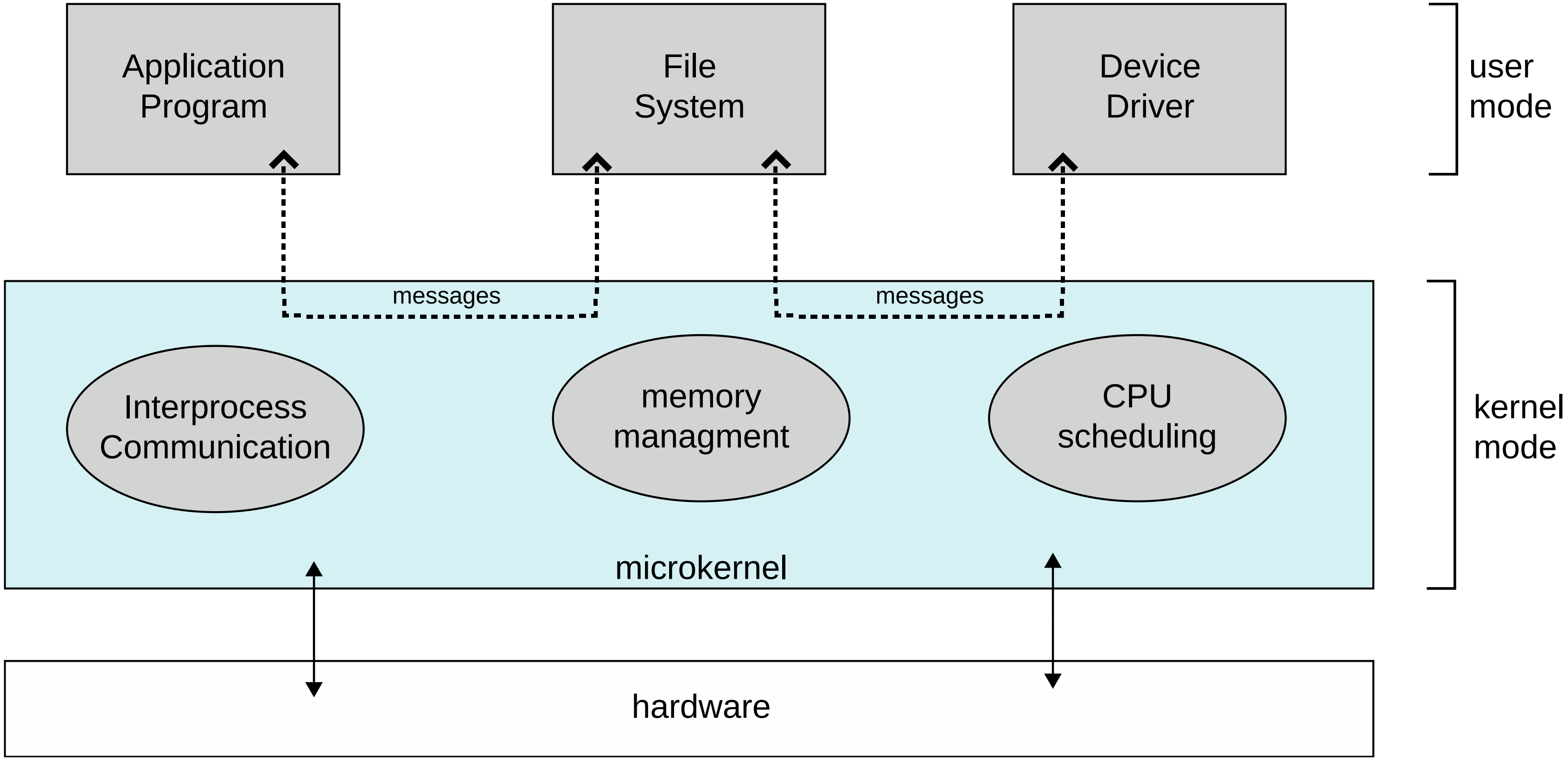
- Most modern operating systems implement modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc



Microkernel System Structure

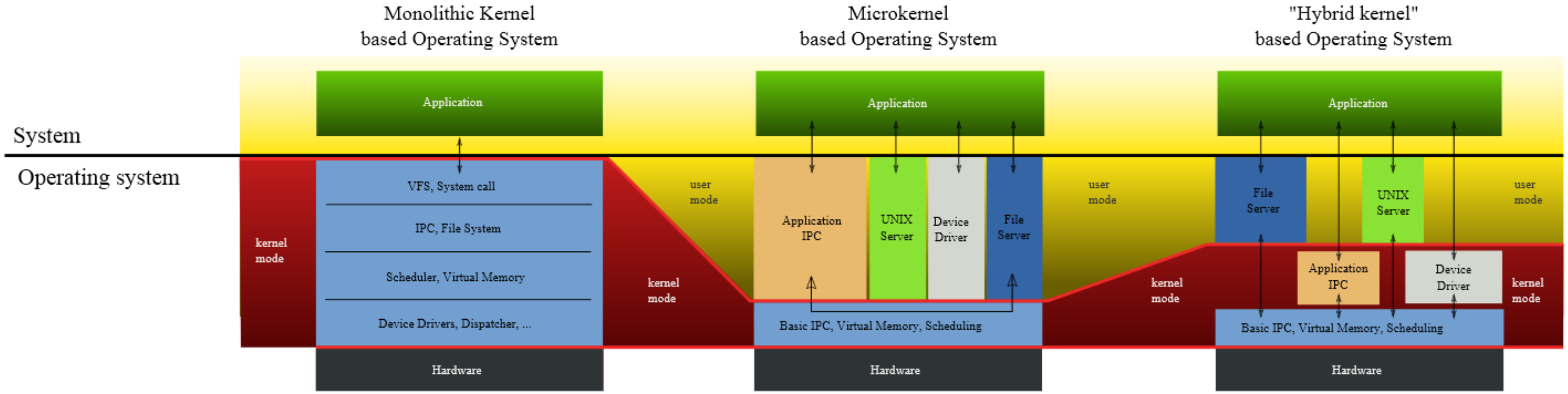
- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure (cont)



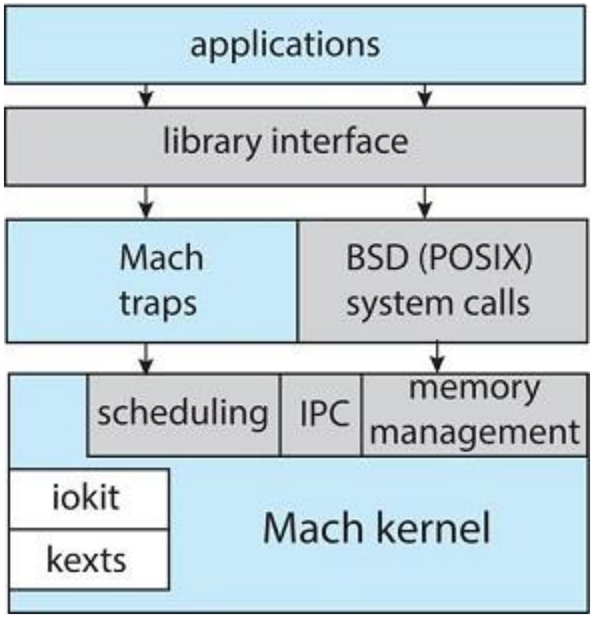
Hybrid Systems

- Most modern operating systems actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels are in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus user-mode services for different subsystem ***personalities,*** e.g. windows subsystems for linux (WSL)



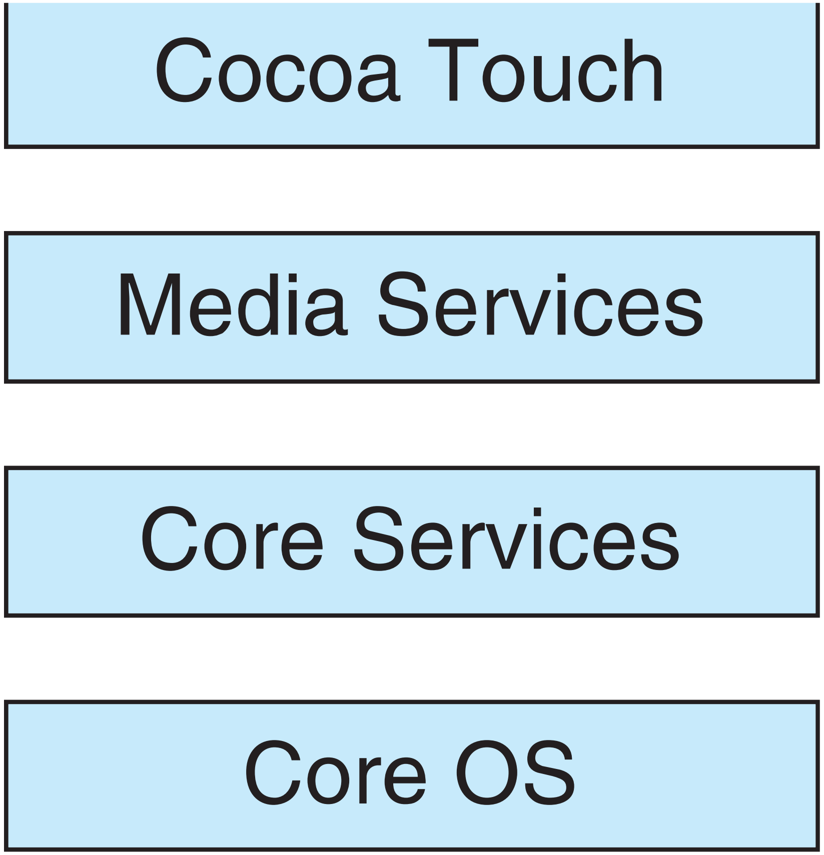
Mac OS X Structure

- Apple Mac OS X Darwin kernel is hybrid, layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel.
  - plus I/O kit and dynamically loadable modules (called **kernel extensions**)

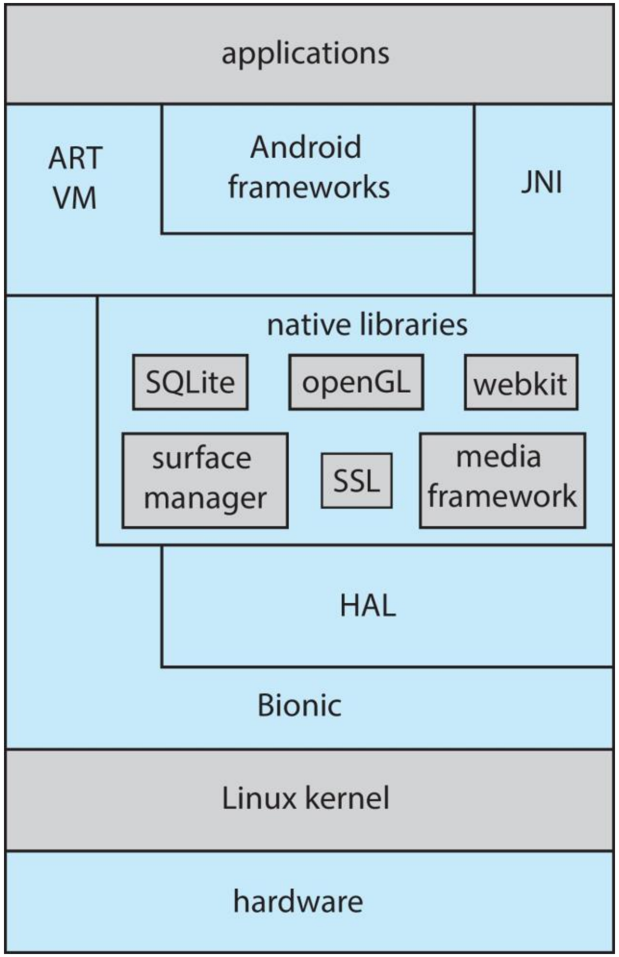


iOS

- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
  - **Cocoa Touch** Objective-C API for developing iOS apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



Android Architecture



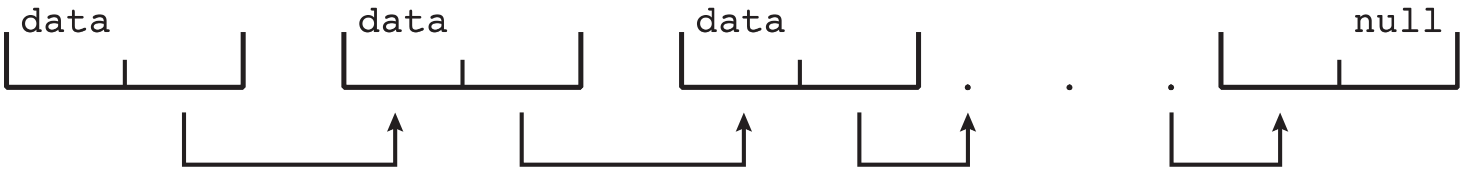
Implementation Issues
(How is the OS implemented?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since both change
- Algorithms used
  - Linear, Tree-based, Log Structured, etc…
- Backward compatibility issues
  - Very important for Windows
- System generation/configuration
  - How to make generic OS fit on specific hardware

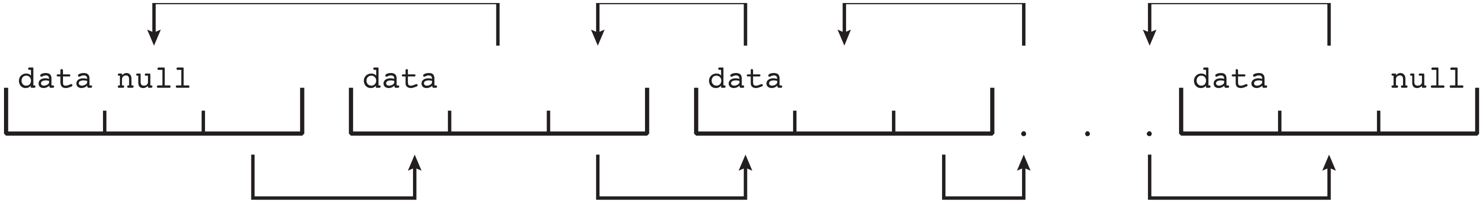
Kernel Data Structures

Many similar to standard programming data structures

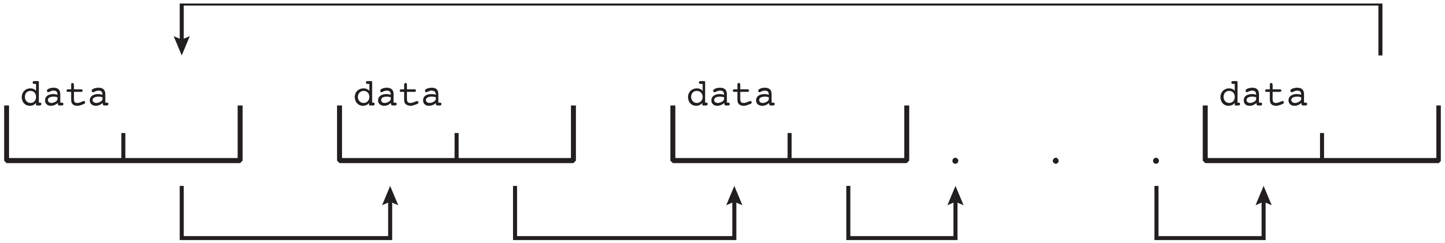
Singly linked list



Doubly linked list

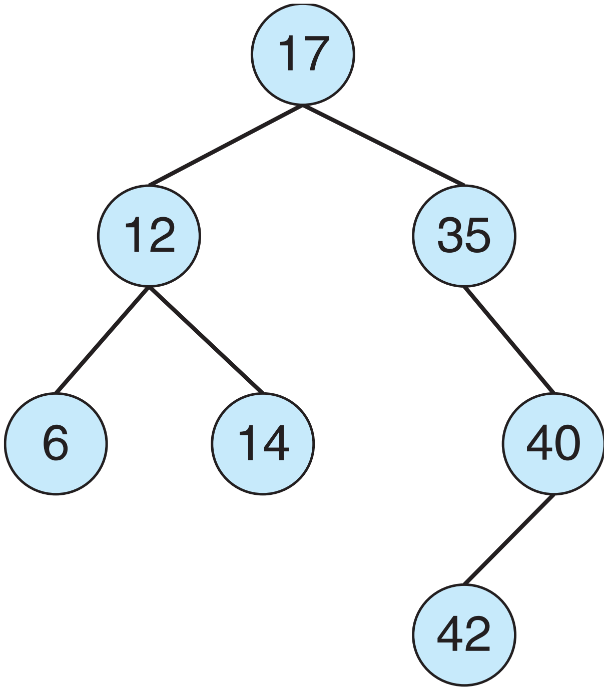


Circular linked list



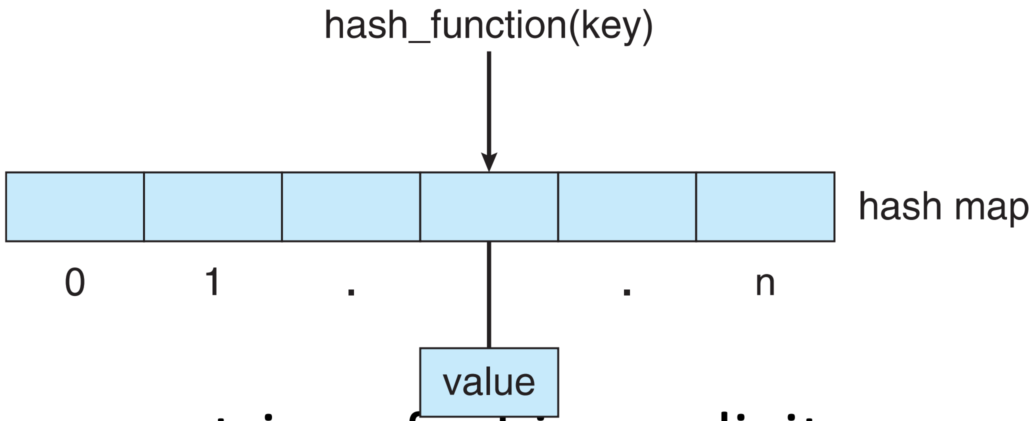
Kernel Data Structures - Tree

- **Binary search tree**
  left <= right
  - Search performance of a list is *O(n)*
  - **Balanced binary search tree** is *O(log n)*



Kernel Data Structures - Hash

- **Hash function** can create a **hash map**



- **Bitmap** – string of *n* binary digits representing the status of *n* items
- Linux data structures defined in ***include*** files `<linux/list.h>, <linux/kfifo.h>, <linux/rbtree.h>`

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generates **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Kernighan’s Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”