Session Plan

- Introduction to Operating Systems
  - What is an OS? What does an OS do?
  - Evolution of OS
  - Why study OS?
  - OS functionalities
  - OS components
  - OS structures

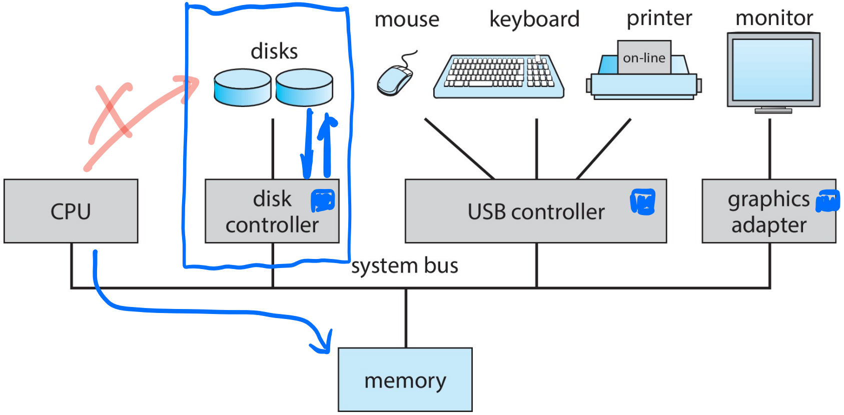
Overview of OS Functionalities

- Computer System Organization
- Interrupts
- System Calls

Computer System Organization

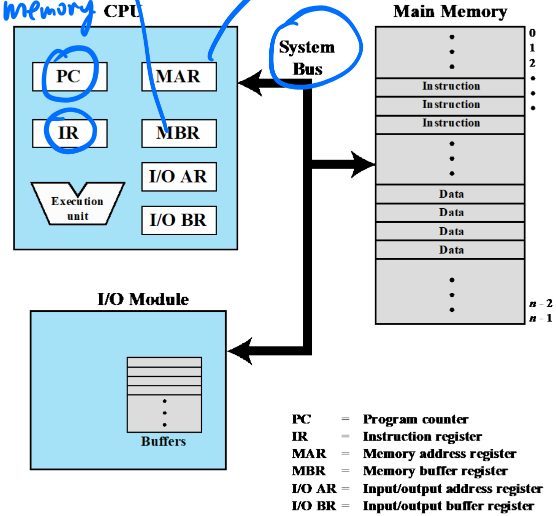
- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles

device driver



Contains the data to be written/read to/from memory

addr in memory for the next r/w



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

Figure 1.1 Computer Components: Top-Level View

Computer System Operation

- I/O devices and the CPU can execute concurrently

contend for memory cycles

- 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

1. device driver loads the registers in the device controller
2. device controller examines the contents
3. device controller starts the transfer of data
4. Once the transfer is done, device controller informs the driver
5. Driver gives control back to the OS

Interrupts

SW program

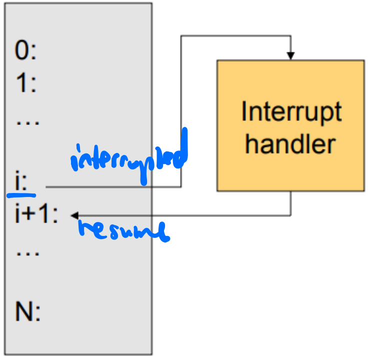
Timer

HW I/O

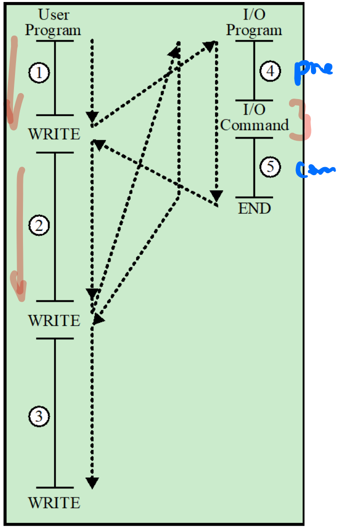
HW failures

e.g. arithmetic overflow, division by zero, reference outside the allowed memory space

- Raised by external events
- Interrupt handler is in the kernel
  - Switch to another process
  - Overlap I/O with CPU
  - …
- Eventually resume the interrupted process
- A way for CPU to wait for long-latency events (like I/O) to happen



Flow of Control w/o Interrupts



(a) No interrupts

prep

complete

①

④

I/O operations; processor waits

⑤

②

④

I/O operations; processor waits

⑤

③

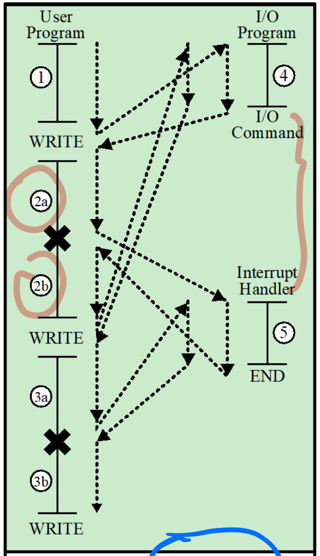
⋮

Flow of Control w/ Interrupts

overlap of CPU & I/O

increase resource utilization

long I/O wait?



(b) Interrupts; short I/O wait

1
4
I/O 2a
I/O operations concurrent w/ processor executing
5
2b
4
I/O 3a
I/O operations concurrent w/ processor executing
5
3b
⋮

Common Functions of Interrupts

O(1)

Int	Addr of service routine
01	- - -
02	- - -
03	- - -

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines

table

- Interrupt architecture must save the address of the interrupted instruction
- A trap or exception is a software-generated interrupt caused either by an error or a user request
- An operating system is *interrupt driven*

method 1: ① invoke a generic routine

→ ② call the interrupt-specific routine

method 2: A table of pointers to interrupt routines

indexed

↑ invoked through the table

keyboard

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter

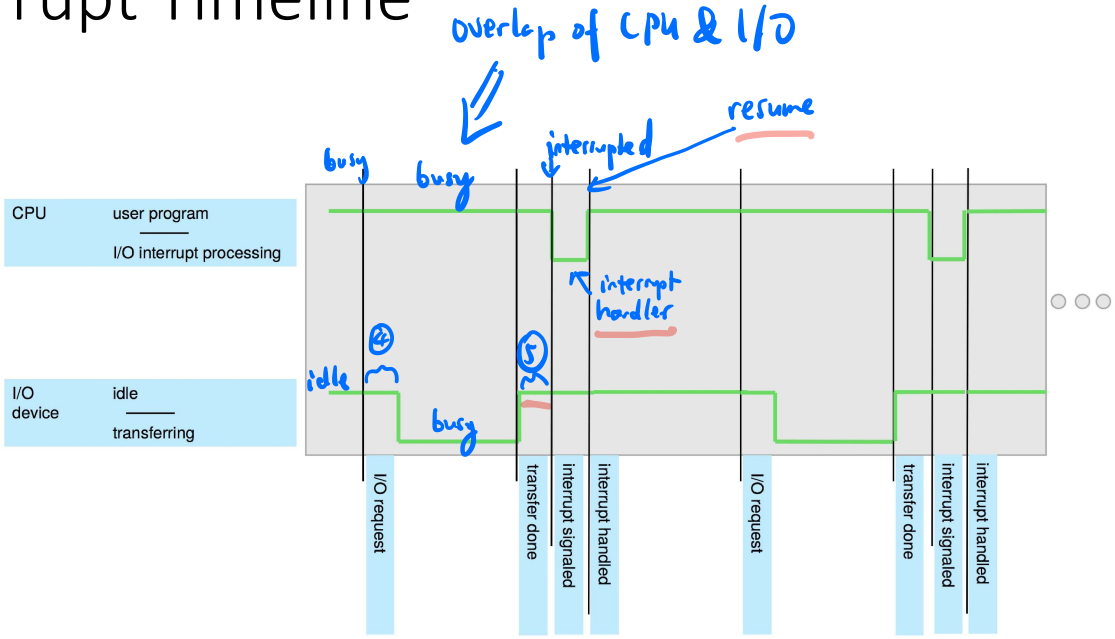
CPU constantly needs to read the keyboard memory to see if a key is pressed

- Determines which type of interrupt has occurred:
  - polling busy waiting
  - vectored interrupt system look up in the table

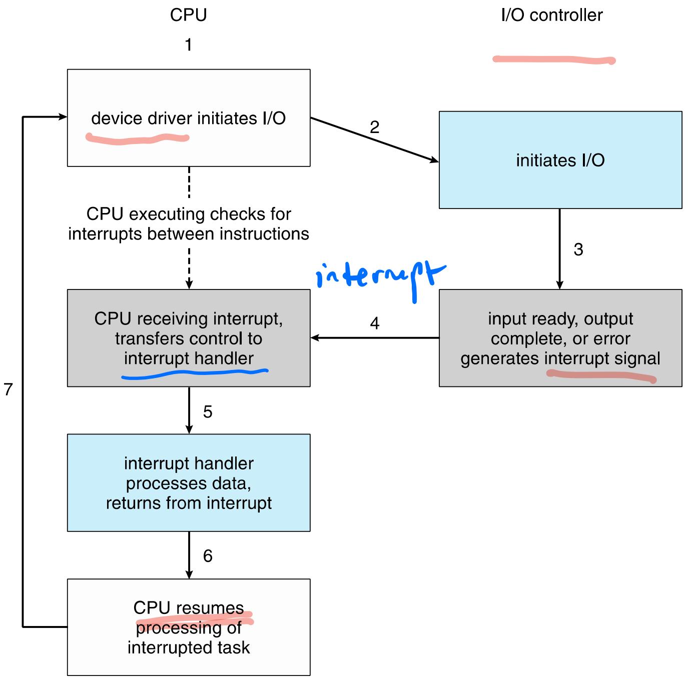
add extra circuitry so the keyboard can alert the CPU when there is a key press

- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Timeline



Interrupt-driven I/O Cycle



CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
interrupt
4
CPU receiving interrupt, transfers control to interrupt handler
input ready, output complete, or error generates interrupt signal
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
7

Operating-System Operations

- Interrupt driven (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (exception or trap):
    - Software error (e.g., division by zero)
    - Request for operating system service
    - Other process problems include infinite loop, processes modifying each other or the operating system

Operating-System Operations (cont.)

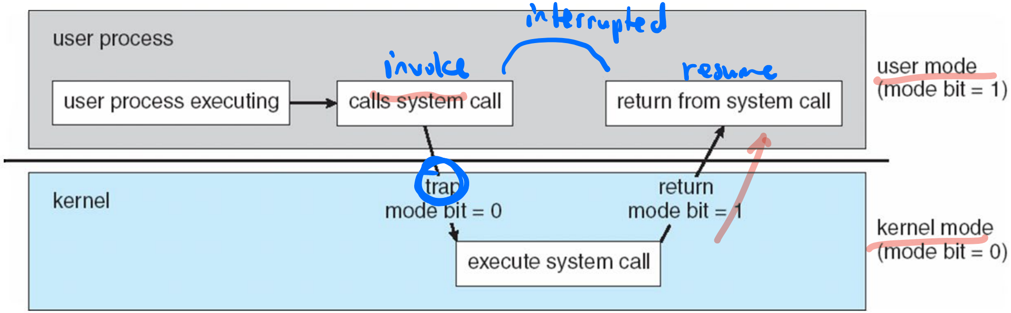
- Dual-mode operation allows OS to protect itself and other system components

  non-privileged

  - User mode and kernel mode both non-privileged and privileged
  - 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 changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
  - i.e. virtual machine manager (VMM) mode for guest VMs

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock.
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt



System Calls

- Goal: Do things applications can’t do in unprivileged mode
- 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 Programming Interface (API) rather than direct system call use
- 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)
- Why use API rather than system calls?

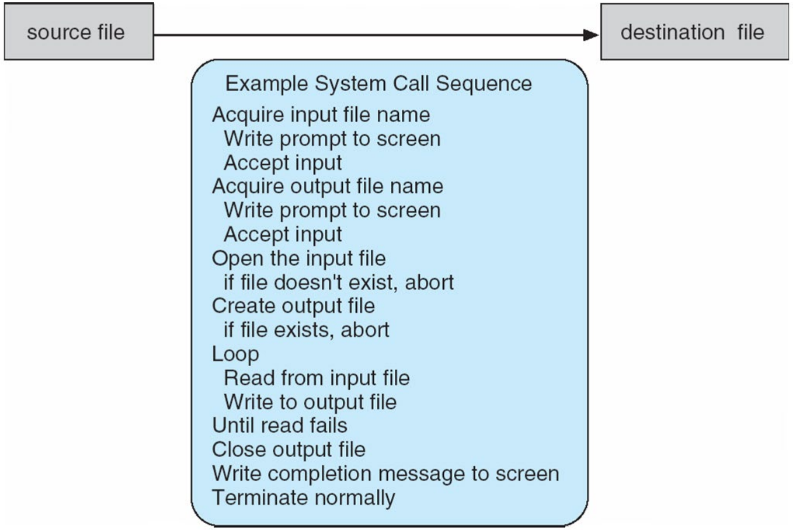
^
direct

Major System Calls in Linux: File Management

- `fd = open(file, how, …)`
  - Open a file for reading, writing, or both
- `s = close(file)`
  - Close an open file
- `n = read(fd, buf, nbytes)`
  - Read data from a file into a buffer
- `n = write(fd, buf, nbytes)`
  - Write data from a buffer into a file
- `pos = lseek(fd, offset, whence)`
  - Move the file pointer
- `s = stat(name, &buf)`
  - Get a file’s status info

Example of System Calls

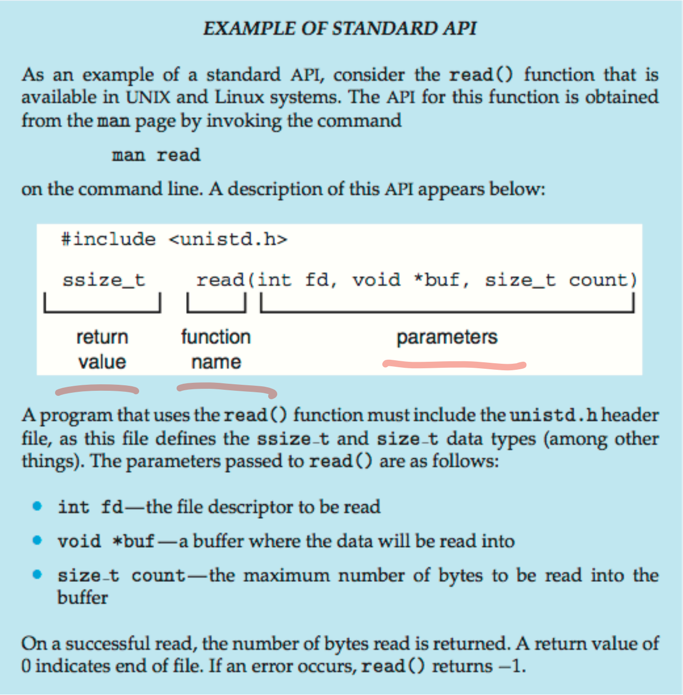
- System call sequence to copy the contents of one file to another file



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
```

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>

ssize_t     read(int fd, void *buf, size_t count)
```

return value

function name

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:

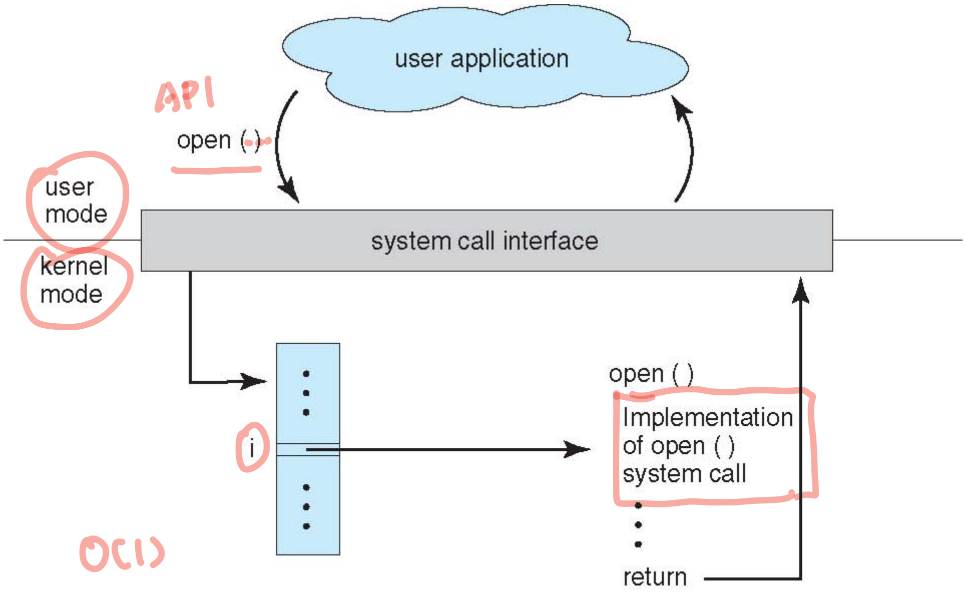
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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.

System Call Implementation

- The run-time support system (run-time libraries) provides a system-call interface, that intercepts functions calls in the API and invoke the necessary system call within the operating system
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes the 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

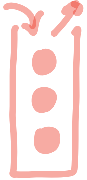


System Call Parameter Passing

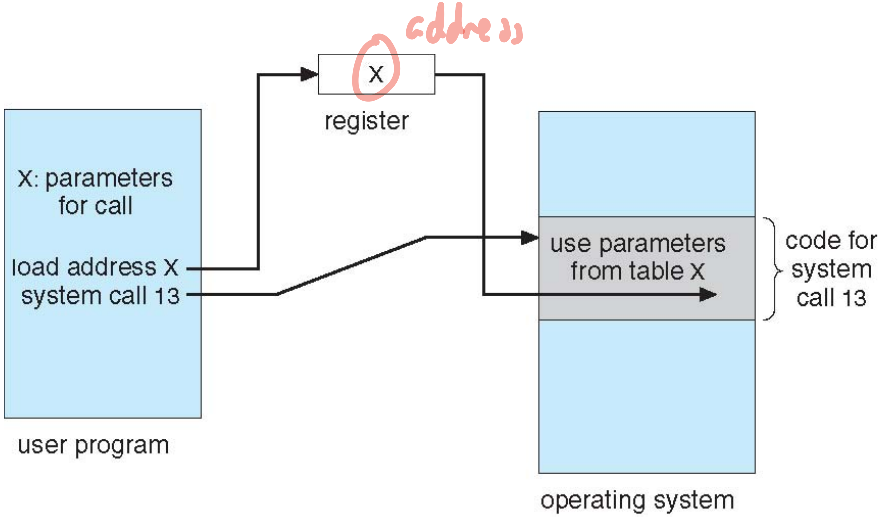
- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register

    one address
    - 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

    LIFO
  - Block and stack methods do not limit the number or length of parameters being passed



Parameter Passing via Table



Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes

Types of System Calls (cont.)

- File management
    - create file, delete file
    - open, close file
    - read, write, reposition
    - get and set file attributes
- Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices

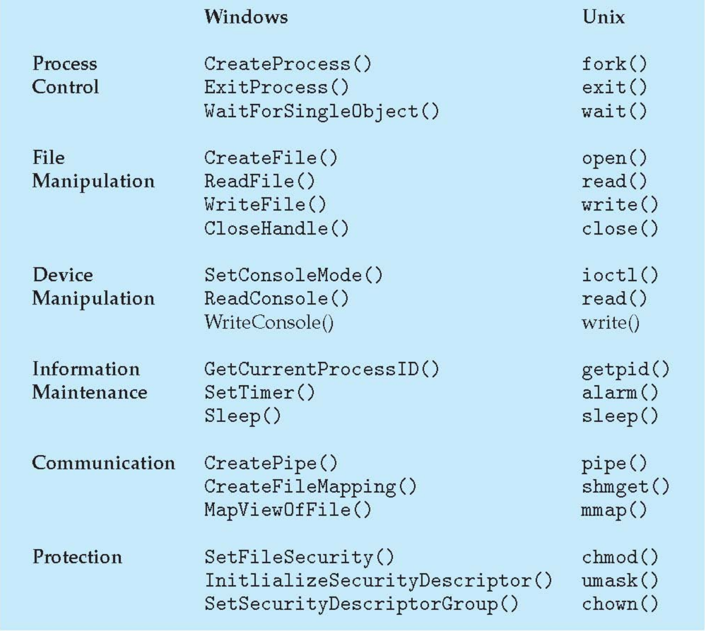
Types of System Calls (cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name
    - From client to server
  - Shared-memory model create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls (cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

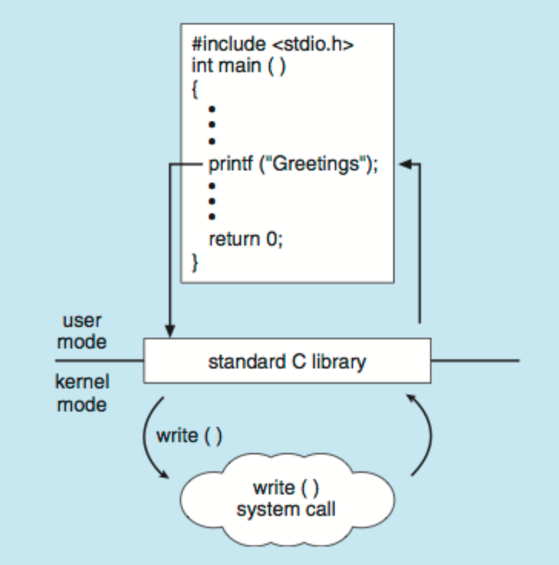
Examples of Windows and Unix System Calls



	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()

Standard C Library Example

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



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

Overview of OS Components

- Process management
- Memory management
- I/O device management
- File system

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one program counter specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

- The operating system is responsible for the following activities in connection with process management:
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
  - Providing mechanisms for deadlock handling

Storage-Device Hierarchy

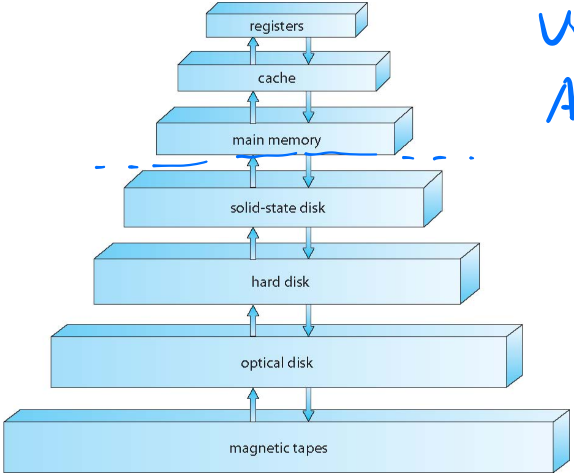
Capacity

Speed

Cost

Volatility

Access frequency



Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - Random access
  - Typically volatile
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
  - 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
  - Solid-state disks – faster than hard disks, nonvolatile
    - Various technologies
    - Becoming more popular

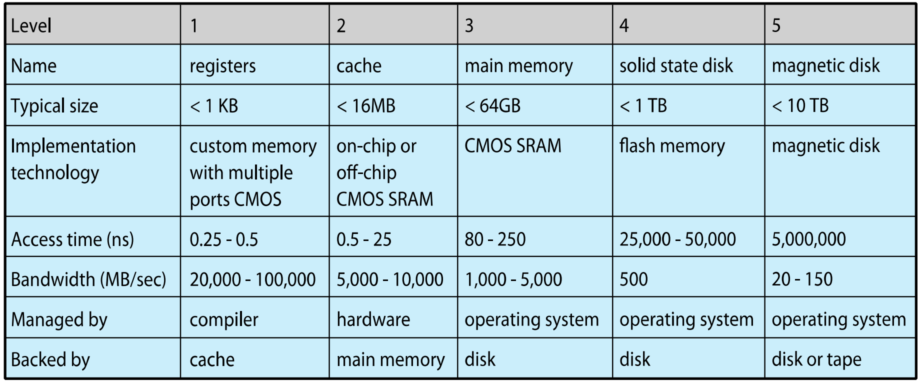
Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- Caching – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- Device driver for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Caching

- Skew rule: 80% requests hit on 20% hottest data
- 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

Performance of Various Levels of Storage

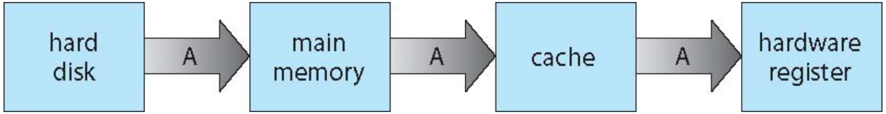


Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 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

Movement between levels of storage hierarchy can be explicit or implicit

Migration of data "A" from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist
  - Various solutions covered in Chapter 17

Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and when
    - Optimizing CPU utilization and computer response to users
- Memory management activities
    - Keeping track of which parts of memory are currently being used and by whom
    - Deciding which processes (or parts thereof) and data to move into and out of memory
    - Allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - file
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications
  - Varies between WORM (write-once, read-many-times) and RW (read-write)

I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices
- Interrupt handlers and device drivers are crucial in the design of efficient I/O subsystems

Protection and Security

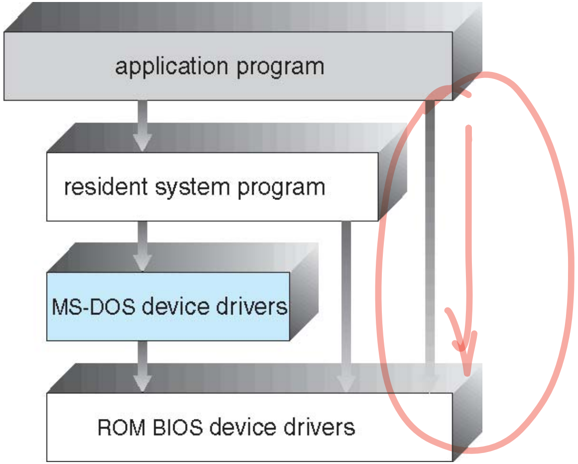
- Protection – any mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of the system against internal and external attacks
    - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
    - User identities (user IDs, security IDs) include name and associated number, one per user
    - User ID then associated with all files, processes of that user to determine access control
    - Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
    - Privilege escalation allows user to change to effective ID with more rights

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel - Mach

Simple Structure – MS-DOS

- 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

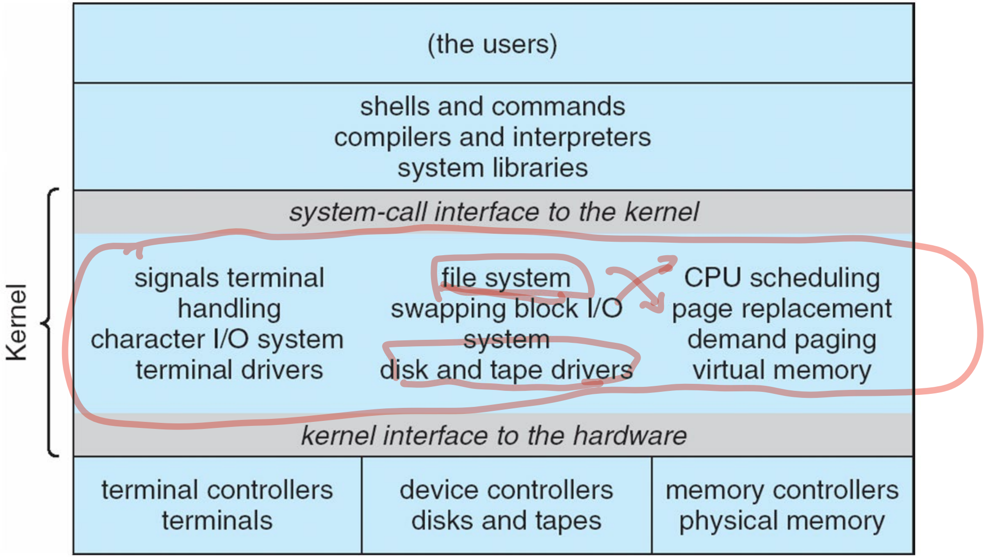


Non Simple Structure -- Unix

- 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

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

monolithic

+ shared kernel space

+ good performance

- Instability

- Inflexibility

- hard to maintain / extend

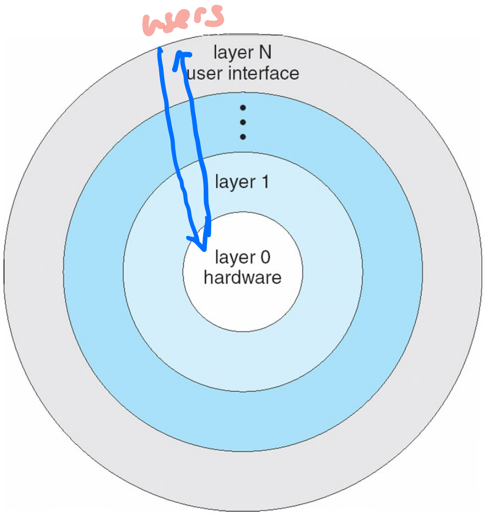
Layered Approach

modular

UNIX

MULTICS

- 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



THE

5 The operator

4 User programs

3 I/O management

2 Comm.

1 memory mana.

0 CPU allocation & multipro-gramming

+ simple to construct and debug

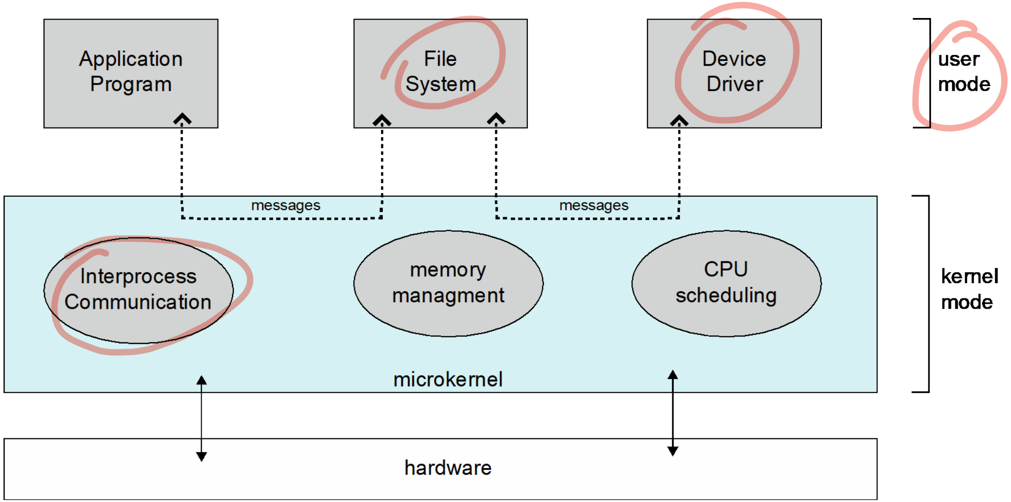
− communication overhead

− impractical to define the functionality of each layer

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



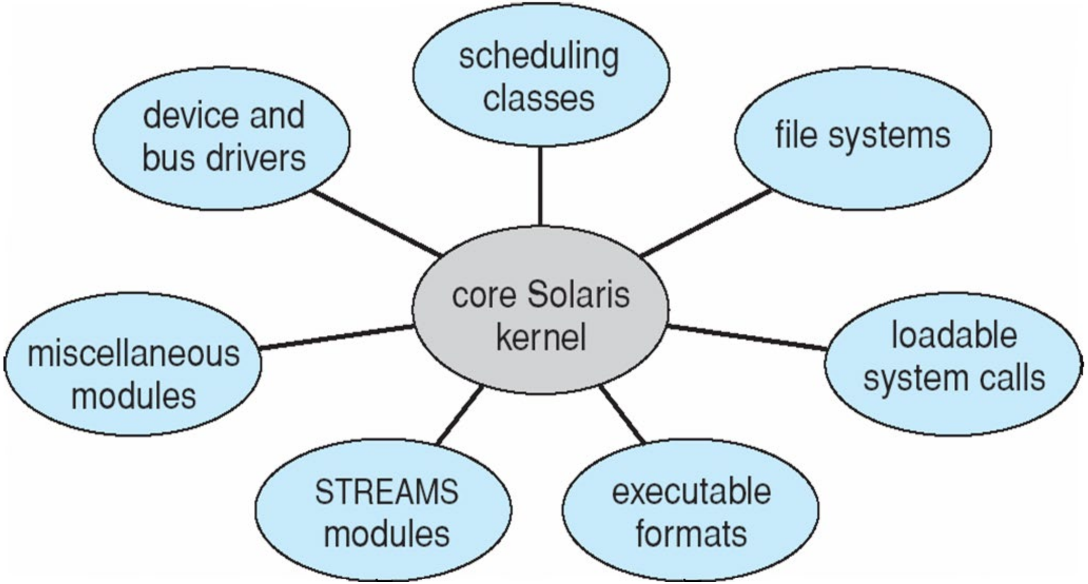
Modules

- Many modern operating systems implement loadable kernel 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.

+ more flexible than the layered approach

+ more efficient than the microkernel approach

Solaris Modular Approach



Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
  - One thing you learned in this lecture
  - One thing you didn’t understand