CSC 539: Operating Systems Structure and Design

Spring 2006

Computer system structures

- computer system operation, interrupts
- I/O
- memory
- hardware protection

Operating system structures

- OS components & services
- system calls
- system structure

Computer system operation

virtually all modern computers utilize the von Neumann architecture

1. *CPU (Central Processing Unit):* perform calculations, fetch & execute instructions
2. *Memory:* store data and program instructions
3. *I/O devices:* allow for communication with users and other computers

- to allow for concurrent execution with the CPU, each I/O device has its own controller
- each device controller has a local buffer
- I/O is from the device to local buffer of controller
- device controller informs CPU that it is done by causing an interrupt
- CPU moves data between memory and the local buffers
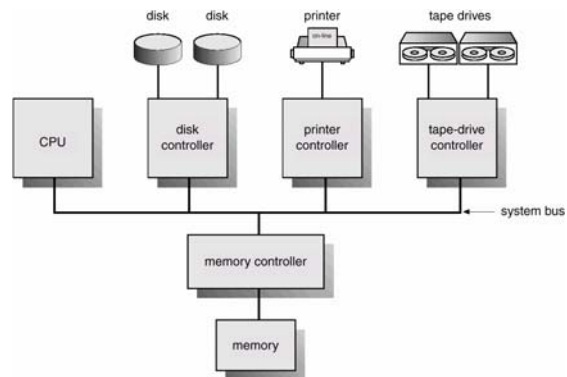
Interrupts

operating systems are interrupt-driven

- an *interrupt* is a request for service from the CPU
    can be generated by hardware via the system bus (e.g., segmentation fault)
    can be generated by software (e.g., *system call* for I/O, *trap* for division by zero)

when an interrupt is received, the CPU must

1. save the address of the interrupted instruction
2. disable (or queue) incoming interrupts while processing this one
3. transfer control to the appropriate interrupt service routine

    this is usually done via an *interrupt vector* (table of addresses for interrupt service routines), stored in first ~100 memory locations
4. after processing the interrupt, enable incoming interrupts
5. restore interrupted instruction & resume processing

3

I/O structures

I/O device controller moves data between the device and its local buffer

to start an I/O operation:

- CPU loads the appropriate registers in the device controller
- the device controller examines these registers to determine what actions to take
    (e.g., read request → start transferring data from device to buffer,
        write request → start transferring data from buffer to device)
- when action is completed, the device controller informs the CPU via an interrupt

interrupt timeline for process doing output

4

2
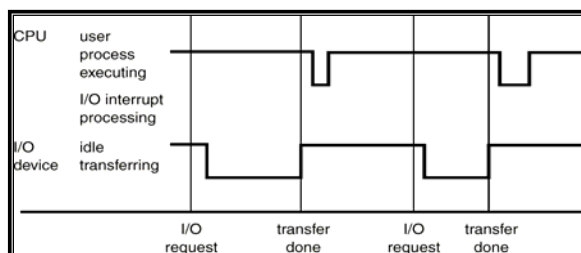
I/O modes

synchronous I/O
- after I/O starts, control returns to user program only upon I/O completion
- can be accomplished via a *wait instruction*, or a *wait loop*
- at most one I/O request is outstanding at a time, no simultaneous I/O processing
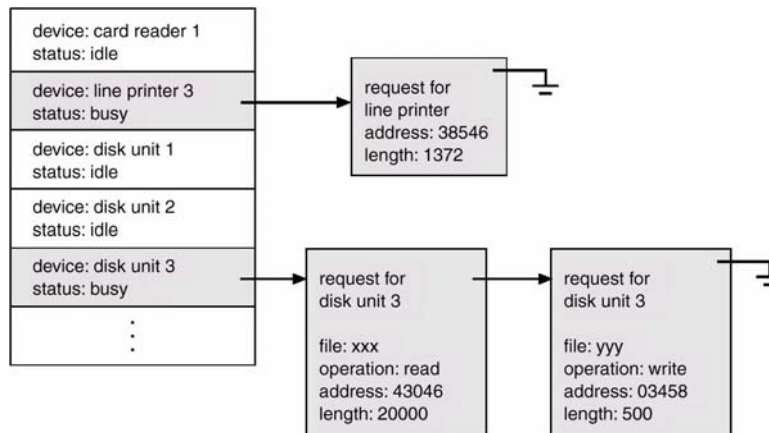
asynchronous I/O
- after I/O starts, control returns to user program without waiting for I/O completion.
- *device-status table* contains entry for each I/O device indicating its type, address, and state.
- OS indexes into I/O device table to determine device status and to modify an entry

synchronous I/O   **asynchronous I/O**
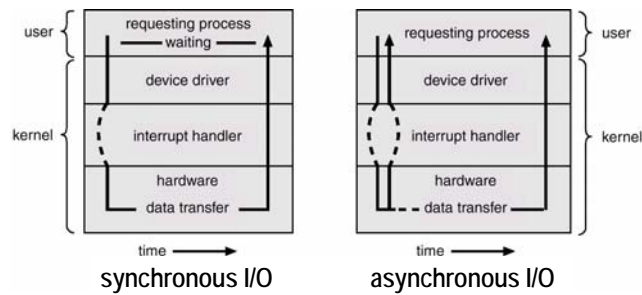
5

Device-status table

tradeoffs between synchronous and asynchronous I/O?

6

Direct Memory Access (DMA)

simple keyboard processing:
- user types character at keyboard
- keyboard controller sends interrupt to CPU
- CPU must complete current instruction, then save state
- control is transferred to interrupt service routine, which:
  - stores char in a buffer & increments buffer pointer
  - sets flag to notify OS that input is available (can be transferred to requesting program)
- must restore CPU state and resume processing

for high-speed I/O devices, the combined overhead of the interrupts may be too costly
- Direct Memory Access: a *device driver* assigns a specific memory segment to the device controller
- device controller can transfer an entire block of data directly to/from main memory without CPU intervention.
- only one interrupt is generated per block, rather than the one interrupt per byte

7

Storage structure

memory is organized in a hierarchy
tradeoffs: speed, cost, volatility

primary memory
- fast, expensive, volatile
- data is stored in electronic circuitry
- only storage CPU can access directly

secondary memory
- slow, cheap, permanent
- data is stored magnetically or optically or "physically"
- can store massive amounts of inactive data, must be copied to primary memory to be accessed

8

4
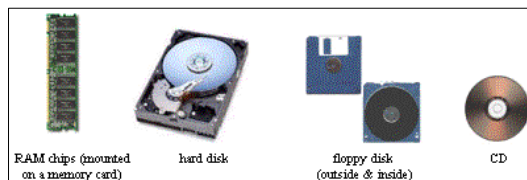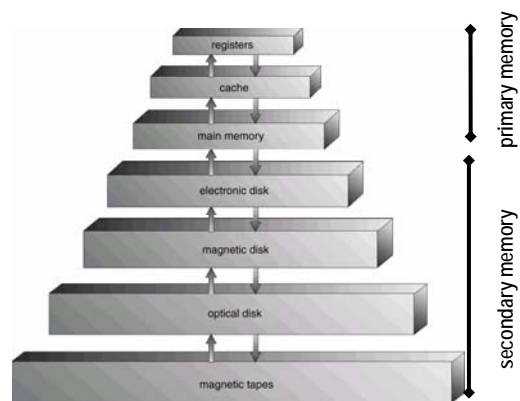
Primary memory: RAM vs. cache

RAM and cache both store "active" data in (volatile) electronic circuitry
- cache uses faster, more expensive technology
- Level-1 cache is stored directly on the CPU chip (runs at speeds comparable to processor speed, ~10x RAM access speed common)
- Level-2 cache is stored on nearby chip, ~2x RAM access speed common

common approach:
- when data from RAM is needed by CPU, first copy into cache
- CPU then accesses cache directly
- cache retains recently used (most active?) data, fast access if needed again

  note: cache is to RAM as RAM is to secondary memory
  (RAM to cache usually handled by hardware; disk to RAM usually handled by OS)

| magnetic disk | A | main memory | A | cache | A | hardware register |
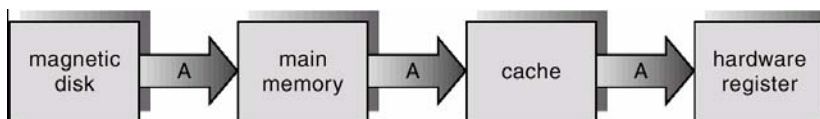
9

Primary memory: caching

caching is an important principle of computer systems
- primary memory cache
- CPU instruction caching
- virtual memory/paging
- Web page caching

since caching implies duplicate copies of data, cache management is tricky
- changes must eventually propagate back down the hierarchy

- if multiple processes running, must ensure that each gets most recent update
   for RAM cache, this usually managed at hardware level

    how is it handled with Web page caching?

10

5

Magnetic media

hard disks
- metal or glass platters covered with a magnetic recording material
- disk surface is logically divided into *tracks*, which are subdivided into *sectors*.
- data access requires moving read-write head to desired track, rotating disk to desired sector
  - *seek time:* 10-20 msec common
  - *data rate:* 5-40 MB/sec common
- for mainframes/servers, may have multiple disks with separate read-write heads
- for PC, 1 disk: 20–120GB common
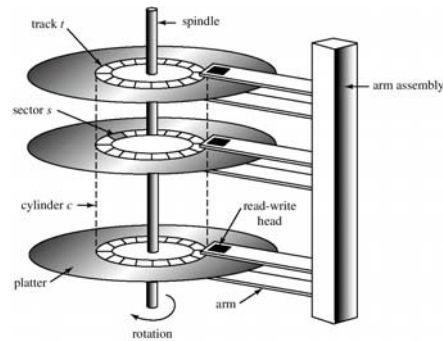
floppy disks
- small, portable version of hard disk
- 3.5" plastic disk, 1.44 GB

magnetic tape
- tape is still used, mainly for backup
- large capacity, but accessed sequentially

11

Compact disks

- CD-ROM stores digital data (~780 MB) as pits burned onto the surface of a plastic disk
- unlike a hard disk or floppy disk, data is stored on a single, continuous track
  - track is ~0.5 microns (μm) wide
  - tracks are separated by ~1.6 μm
  - pit depth is ~0.125 μm
  - linear length of track is ~5 km
- data representation utilizes error-correcting codes, interleaving of data
- disk is rotated and data is read by tracking mechanism
  - fires low-power laser at bottom surface, can read reflections
- CD-R and CD-RW are similar, but utilize chemical layers on the disk surface that change reflectivity after laser exposure

12

6
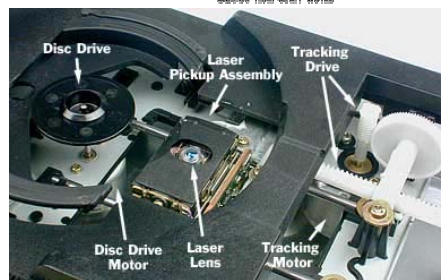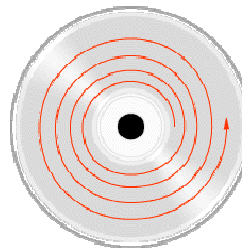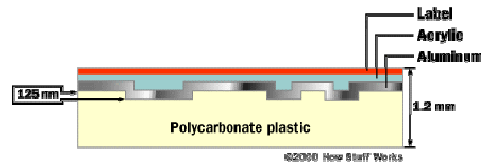
Hardware protection

OS must prevent user jobs from interfering with the operation of the system
tools: dual-mode operation, privileged instructions, memory protection, timer interrupts

dual-mode operation
- can provide hardware support to differentiate between modes of operation
  - *user mode:* execution done on behalf of user
  - *monitor mode* (a.k.a. *kernel* or *system mode*): execution done on behalf of OS

- *mode bit* is added to computer hardware to indicate the current mode: monitor (0) or user (1)

- when an interrupt or fault occurs, hardware switches to monitor mode

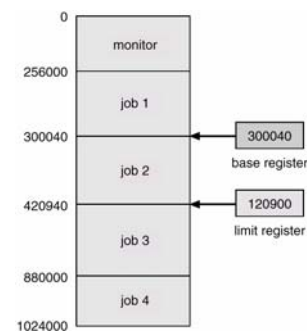- privileged instructions (e.g., I/O) can be issued only in monitor mode

Interrupt/fault

monitor   user

set user mode
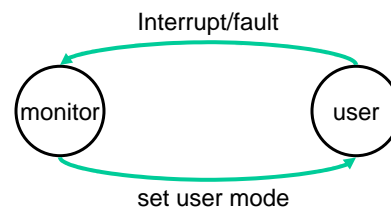
13

Hardware protection (cont.)

privileged instructions
- all I/O instructions are privileged instructions
  1. instead of performing I/O operation directly, user program must make a system call
  2. OS, executing in monitor mode, checks validity of request and does the I/O
  3. input is returned to the program by the OS

memory protection
- must provide memory protection at least for the interrupt vector and the interrupt service routines
- memory protection implemented in hardware using:
  - *base register* – holds the smallest legal physical memory address.
  - *limit register* – contains the size of the range
- when in user mode, CPU hardware compares every address with defined range, disallows out-of-range

- load instructions for the *base* and *limit* registers are privileged instructions.

0	
monitor	
256000	
job 1	
300040	300040 → base register
job 2	
420940	120900 → limit register
job 3	
880000	
job 4	
1024000	

14

Hardware protection (cont.)

timer interrupts

- in addition to protecting I/O (privileged instructions) and memory (address checking), must also ensure that the OS retains control

    can't have jobs that run forever, or fail to return control to the OS when done

- a *timer* can be used to limit the execution time for a job

    timer is really just a counter, initialized at start of job & decremented every clock tick

    when timer reaches the value 0, an interrupt occurs & job is terminated

- as before, loading the timer is a privileged instruction

QUESTION: is there a connection between timer interrupts and time sharing?

15

Components of an OS

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

16

8

Process management

a *process* is a program in execution (more specifics later)
- process is the basic unit of work on a computer
- process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task

the OS is responsible for the following activities w.r.t. process management.
- process creation and deletion
- process suspension and resumption
- provide mechanisms for:
   process synchronization
   process communication
   handling deadlock
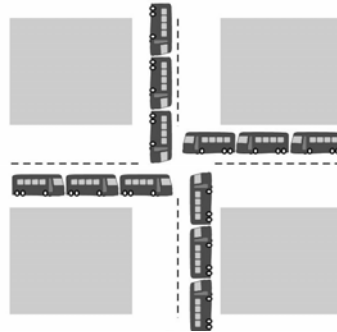
17

Main memory management

programs do not become processes until they are loaded into main memory and begin to execute

main memory is a large array of words or bytes, each with its own address
- it is a repository of quickly accessible data shared by the CPU and I/O devices
- it is volatile – contents are lost in the case of system failure or power loss

the OS is responsible for the following activities w.r.t. memory management:
- keep track of which parts of memory are currently being used and by whom
- decide which processes to load when memory space becomes available
- allocate and deallocate memory space as needed

| Job 3 |
| Job 2 |
| Job 1 |
| Operating system |

Memory partitions
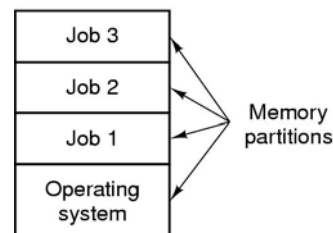
18

File management

a *file* is a collection of related information defined by its creator
- files represent programs (both source and object forms) and data
- the OS maps logical files to physical devices

the OS is responsible for the following activities w.r.t. file management:
- file creation and deletion
- directory creation and deletion
- support of primitives for manipulating files and directories
- mapping files onto secondary storage
- file backup on stable (nonvolatile) storage media
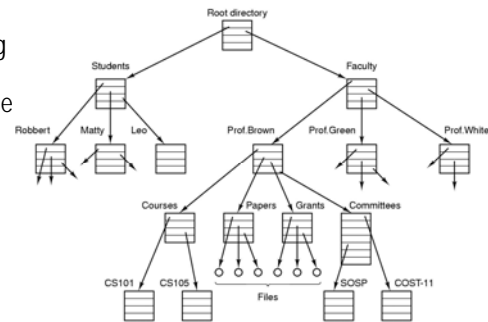
19

Other components of the OS

I/O system management
- a buffer-caching system
- a general device-driver interface
- drivers for specific hardware devices

secondary storage management
- free space management
- storage allocation
- disk scheduling

networking support

protection system

command interpreter

20

OS services

from the user's perspective:

- program execution – capability to load a program into memory and to run it
- I/O operations – since user programs are restricted, OS must provide I/O
- file system manipulation – capability to read, write, create, and delete files
- communication – exchange of information between processes executing either on the same computer or in a distributed network (implemented via *shared memory* or *message passing*)
- error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs

from the system's perspective:

- resource allocation – to multiple users or multiple jobs running at the same time
- accounting – accumulate usage statistics and bill users accordingly
- protection – ensuring that all access to system resources is controlled

21

System calls

system calls provide the interface between a process and the OS
- whenever a process needs to do something that only the OS can do – it performs a system call
- generally available as assembly-language instructions
- wrapper libraries provide a high-level interface (e.g., UNIX fork and exec in C/C++, Windows API)

- there are system calls to do process control, file manipulation, device manipulation, info maintenance, communication, …

parameters may be passed between a process and the OS
1. can pass parameters in *registers*.
2. can store the parameters in a table in memory, and the table address is passed as a parameter in a register
3. can *push* the parameters onto the *stack* by the program, and *pop* off the stack by operating system
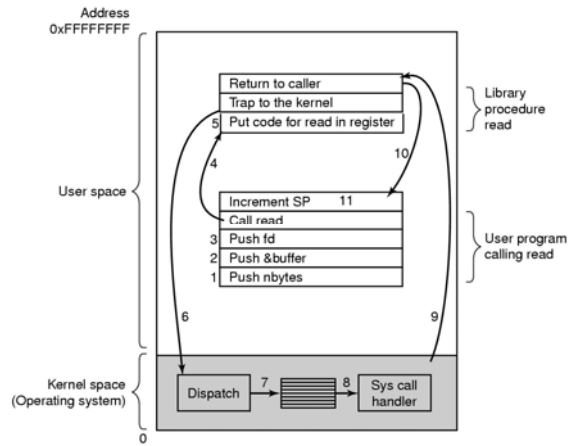
22

Steps in making a system call

consider the UNIX system call:

`read(fd, buffer, nbytes)`

which reads n bytes of data from a file (given a file descriptor) into a buffer

11 steps:
- 1-3: push parameters onto stack
- 4: call library routine
- 5: code for read placed in register
- 6: trap to OS
- 7-8: OS saves state, calls the appropriate handler
- 9-10: return control back to user program
- 11: pop parameters off stack

Address
0xFFFFFFFF

Return to caller
Trap to the kernel
5 | Put code for read in register

4

10

User space

Increment SP   11
Call read
3 | Push fd
2 | Push &buffer
1 | Push nbytes

6

9

Library procedure read

User program calling read

Kernel space
(Operating system)

Dispatch   7   8   Sys call handler

0

language libraries (e.g., `<iostream>`, `<fstream>`) hide these system calls, make interface easier for user

23

What is the kernel?

the boundaries of an OS are fuzzy
- is Internet Explorer a part of the Windows OS?

the *kernel* is the essential core of the OS
- consists of those parts that must be in memory, ready to execute, at all times
  - e.g., scheduling a process to execute
    handling an interrupt from a device
    reacting to a page fault in a virtual memory system

- all other programs may be classified as either
  - *system programs:* provide an environment for program development and execution
    e.g, user interfaces to system calls, programs for directory management, …

  - *applications programs:* provide user with tools for solving problems in various domains
    e.g., word processor, spreadsheet, graphics editor, …

24

Example: MS-DOS

MS-DOS is a single-tasking OS (single user, single process)
- command interpreter is invoked when the computer is started
- to run a program, that program is loaded into memory – overwriting some of the command interpreter
- when program terminates, control is returned to the command interpreter which reloads its overwritten parts

at startup   **running a program**
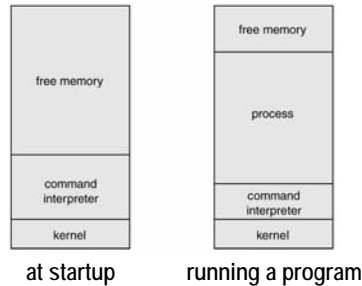
- can get some of benefits of multiprogramming via "terminate & stay resident" system call (reserves space so that process code remains in memory)

25

Example: UNIX

UNIX is a multi-tasking OS (multiple users, multiple processes)
- each user runs their own shell (command interpreter), e.g., sh, csh, bash, …
- to start a process, the shell executes a `fork` system call, the selected program is loaded into memory via an `exec` system call, and the new process executes
- depending on the command, the shell may wait for the process to finish or else continue as the process runs in the "background"
- when a process is done, it executes an exit system call to terminate, returning a status code that can be accessed by the shell

most UNIX commands are implemented by systems programs
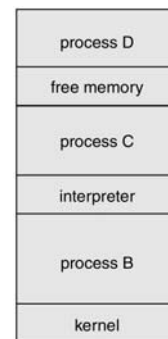- command interpreter uses command to identify external file
- then loads and executes that program

  e.g., `rm` command finds & executes `/usr/bin/rm`

  advantages: command interpreter is small
          can extend OS functionality by adding system programs
  disadvantages: slower than direct execution, must handle parameters
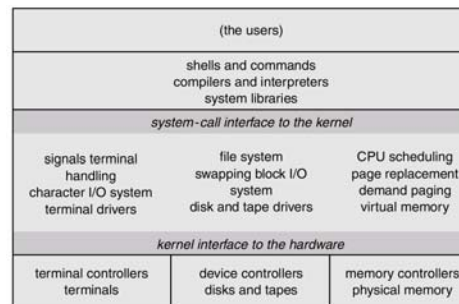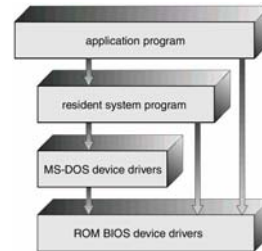   also, inconsistencies between commands

26

13

OS structure: simple

many OS's lack a well-defined structure

- MS-DOS began as small, simple, limited
    - designed for early 80's PC, limited memory & supporting hardware
    - as such, no clear division into modules

| application program |
| resident system program |
| MS-DOS device drivers |
| ROM BIOS device drivers |

- UNIX was also initially limited by hardware functionality
    - OS is divided into kernel & system programs
    - kernel provides the file system, CPU scheduling, memory management, and other OS functions

(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

---

OS structure: layered → microkernel

layered approach: OS is divided into distinct layers
- bottom layer = hardware; highest layer = user interface
- layers are selected such that each uses functions and services of adjacent layers

advantage: modularity simplifies development & debugging
disadvantages: requires careful definition of layers; tend to be less efficient (since commands must pass through layers)

microkernel approach: minimal kernel, only essential components
- must at least provide process & memory management, communications
- communications between client programs and services via message passing
    - e.g., client program & file server send messages through microkernel

advantages: easier to extend a microkernel; easier to port the OS to new architectures; more reliable (less code is running in kernel mode); more secure
disadvantages: performance decreases due to increased overhead

- Mach (CMU, mid 80s) is basis for Digital UNIX, NextStep, mklinux, MacOS X, …
- Windows NT was microkernel, but bad performance led to a more monolithic XP

OS structure: modules

modular approach: perhaps best current methodology

- uses object-oriented design techniques to build modular kernel
- kernel has set of core components & dynamically links in other services at boot or run time

advantages:

> well-defined modules as in layered system, but more flexible (any module can call any other module)

> kernel is small as in micokernel, but more efficient (message passing not required for communication between layers)

- used in modern versions of UNIX: Solaris, Linux, …
- hybrid approach used in Mac OS X

> kernel combines Mach and BSD Unix in layers, other services provided by modules

29