## Previous lecture review

Operating system is a program that acts as an intermediary between a user of computer and computer hardware making the use of the machine easy and more efficient

- The development of the OSes followed the evolution of the hardware; primary factor - dramatic increase in speed
- OS changed in size and sophistication from simple batch systems to modern multi-tasking, multi-user systems.
- Additional features are being developed to accommodate parallel, distributed and real-time systems
- Machine sharing and other improvements increased the number of problems OS has to solve:
  - complex task scheduling
  - protection
  - access to secondary storage (disks), networking, etc.



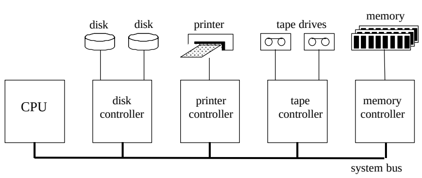
## Lecture 2: Computer System Structures

We go over the aspects of computer architecture relevant to OS design

- overview
- input and output (I/O) organization
- storage structure
- protection
  - I/O protection
  - memory protection
  - CPU protection



## Computer system operation




All devices are attached to *controllers* which are joined by *system bus*

- Controllers and CPU can execute concurrently
- memory is a critical resource - memory controller synchronizes access to memory
- Device controller contains registers for communication with that device
  - Input register, output register — for data
  - Control register — to tell it what to do
  - Status register — to see what it's done

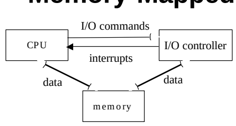


## Input/Output

- *Synchronous* I/O — CPU waits while I/O proceeds
- *Asynchronous* I/O — I/O proceeds concurrently with CPU execution
  - Software-polling (programmed) I/O:
    - CPU starts an I/O operation, and continuously *polls* (checks) that device until the I/O operation finishes
  - Interrupt-driven I/O:
    - Device controller has its own processor, and executes asynchronously with CPU
      - Device controller puts an interrupt signal on the bus when it needs CPU's attention
    - When CPU receives an interrupt:
      1. It saves the CPU state and invokes the appropriate *interrupt handler* using the *interrupt vector* (addresses of OS routines to handle various events)
      2. Handler must save and restore software state (e.g., registers it will modify)
      3. CPU restores CPU state



## Direct Memory Access, Memory-Mapped I/O

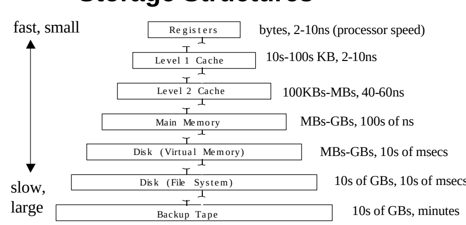



Used for fast devices i.e. disk, network

- DMA
  - I/O controller can transfer block of data to/from memory without going through CPU
  - OS allocates buffer in memory, tells I/O device to use that buffer
  - I/O device operates asynchronously with CPU, interrupts CPU when finished
  - note, that DMA controller competes with CPU for memory access
- Memory-mapped I/O - convenient way of addressing I/O devices, no specific I/O instructions are needed (used for video cards, ports)
  - device registers are mapped to memory addresses, when accessed the data moves directly to registers



## Storage Structures




- the faster the storage the more expensive (less reliable) it is
- this leads to the pyramidal structure of computer storage and *caching*
- *locality of reference:* programs tend to access the same info multiple times; caching is based on this principle:
  - when info is needed, look on this level
  - if it's not on this level, get it from the next slower level, and save a copy here in case it's needed again sometime soon

## Magnetic disks (secondary storage)

- Provide secondary storage for system (after main memory)
- Technology
  - Covered with magnetic material
  - Read / write head “floats” just above surface of disk
  - Hierarchically organized as platters, tracks, sectors (blocks)
- Devices
  - Hard (moving-head) disk — one or more platters, head moves across tracks
  - Floppy disk — disk covered with hard surface, read / write head sits on disk, slower, smaller, removable, rugged
  - CDROM — uses laser, read-only or read/write, high-density



## Protection

- multiprogramming and timesharing require that the memory and I/O of the OS and user processes be protected against each other
  - old OSes like DOS/Windows and MacOS do not support this kind of protection
- two modes of CPU execution introduced: *user mode* and *kernel mode*
  - in kernel / privileged / monitor mode, *privileged instructions* can:
    - access I/O devices, control interrupts
    - manipulate the state of the memory (page table, TLBs, etc.)
    - halt the machine
    - change the mode
  - requires architectural support:
    - mode bit in a protected register
    - privileged instructions, which can only be executed in kernel mode



## I/O protection

- To prevent illegal I/O, or simultaneous I/O requests from multiple processes, perform all I/O via privileged instructions
  - User programs must make a *system call* to the OS to perform I/O
- when user process makes a system call:
  - a *trap* (software-generated interrupt) occurs, which causes:
    - the appropriate interrupt handler to be invoked using the interrupt vector
    - Kernel mode to be set
  - interrupt handler:
    - Saves state
    - Performs requested I/O (if appropriate)
    - Restores state, sets user mode, and returns to calling program



## Memory protection

- must protect:
  - OS’s memory from user programs (can’t overwrite, can’t access)
  - memory of one process from another process
  - protect memory of user process from OS
- simplest and most common technique:
  - *Base register* —smallest legal address
  - *Limit register* — size of address range
  - Base and limit registers are loaded by OS before running a particular process
  - CPU checks each address (instruction & data) generated in user mode
    - Any attempt to access memory outside the legal range results in a trap to the OS
- Additional hardware support is provided for virtual memory



## CPU protection

- Use a timer to prevent CPU from being hogged by one process (either maliciously, or due to an error)
  - Set timer to cause an interrupt after a specified period (small fraction of a second) called *time slice*
  - When interrupt occurs, control transfers to OS, which decides which process to execute for next time interval (maybe the same process, maybe another one)
- Also use timer to implement time sharing
  - At end of each time interval, OS switches to another process
  - *Context switch* = save state of that process, update Process Control Block for each of the two processes, restore state of next process



## OS services and Architecture support

| OS Service | Hardware Support |
|---|---|
| I/O | interrupts<br>memory-mapped I/O<br>caching |
| Data access | memory hierarchies |
| Protection | system calls<br>kernel & user mode<br>privileged instructions<br>interrupts & traps<br>base & limit registers<br>timers |
| Scheduling & Error recovery | timers |