1. Computer System Structure and Components

Computer System Layers

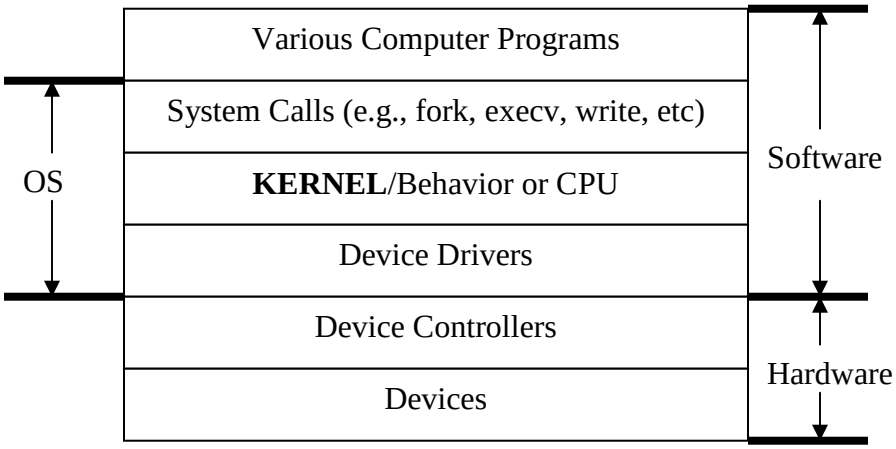

Figure 1 Conceptual Layers of Computer System in more detail

Hardware Components of Computer System

CPU, Device Controllers, Devices, and Bus

An Example of Computer System Structure

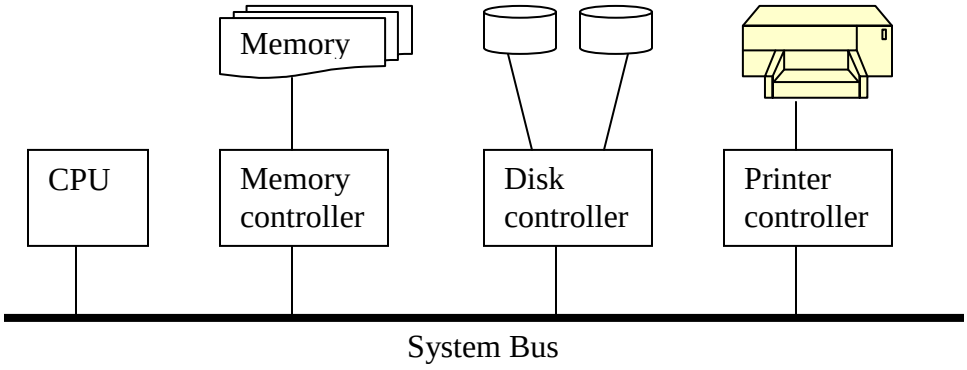

Figure 2 Example of a Computer System Structure

2. Interrupt and Trap

→ Definition: Software or hardware mechanism used to signal the occurrence of an event. (Sending an event signal to the CPU)

→ Examples: Completion of an I/O operation, Invalid operation (e.g., division by zero), Invalid memory or file access, etc.

→ Type of Interrupts

- Hardware interrupt: generated by hardware (e.g., device controller, CPU timer, hardware failure such as parity error and power failure, etc.)
- Software Interrupt: generated by software (e.g., Kernel code for a system call)
- Traps: Software interrupt caused by an exceptional software condition or error (e.g., divide by zero, incorrect use of an instruction, etc.)

→ Motivations of Interrupts

- Efficiency: Does not waist CPU time to poll to devices to check the status of the device.
- Fast response of CPU: CPU can be aware of an event as soon as it occurs (no polling).

→ Interrupt Processing

1) Event occurs and CPU is interrupted
2) CPU stops current process (job).
3) Disable interrupt
4) CPU stores the information related to the current process (e.g., program counter, contents of registers, etc)
5) OS looks up the interrupt service table (interrupt vector table) and find the corresponding interrupt service routine.
6) Executes the interrupt service routine.
7) Resume the interrupted process.
8) Enable interrupt

→ Another polling problem: CPU needs to know who triggered an interrupt

- Solution 1: Polling (polling overhead)
- Solution 2: Each interrupt has only one generator (interrupt table and routines are augmented/duplicated)

→ Device Status Table: Each entry indicates the status (e.g., busy and idle) of the corresponding device. When the status is "busy", the requests are queued and processed later when the device becomes "idle". A device entry can have a chain of the requests for the device.

3. I/O Structure

→ Programmed I/O

1) Application program (process) calls an I/O system call and waits for the I/O to complete.
2) I/O system call routine calls the corresponding I/O function of the device driver.
3) Device driver fills the registers of the device controller and signal "start"
4) The device controller starts the I/O and CPU checks the flag register of the device controller periodically (polling overhead).
5) After each unit I/O, the device controller sets the flag in its register.
6) CPU finds the flag and moves the data to/from the main memory from/to the controller's buffer.
7) Repeat from 2, 3, or 4)[1] to 6) until all data is transferred to/from the device.

- CPU time is wasted (polling I/O devices)

→ Interrupt-driven I/O

1) Application program (process) calls an I/O system call and waits for the I/O completion.
2) I/O system call routine calls the corresponding I/O function of the device driver.
3) Device driver fills the registers of the device controller and signal "start"
4) The device controller starts the I/O and CPU is free from the polling.
5) After each unit I/O, the device controller generates an interrupt signal.
6) CPU stops current process and moves the data to/from main memory from/to the controller's buffer.
7) Repeat from 2, 3, or 4) to 6) until all data are transferred to/from the device.

- CPU is free from the polling. Thus, CPU can process another job while the device is working on 4) and 5). However, in a high-speed device, such as disk, step 4) is fast and frequently interrupts the CPU. As a result, the CPU must switch to the interrupt routines very frequently to move the data in the controller buffer to main memory (switching overhead). Good for byte (or word)-based slower devices (e.g., keyboard, serial port)

→ Direct Memory Access (DMA)

1) Application program (process) calls an I/O system call and waits for the I/O completion.
2) I/O system call routine calls the corresponding I/O function of the device driver with a destination (main memory location).
3) Device driver fills the registers in the device controller and signal "start"
4) The device controller starts the I/O and the CPU is free from the polling.

[1] Depends on the OS, device, device driver, and the device controller.

5) After each unit I/O, the device controller (or DMA controller) moves the data to/from the main memory from/to the controller's buffer.
6) Repeat from 2, 3, or 4) and 5) until all data are transferred to/from the device.
7) Device controller generates an interrupt signal.

- Since the device controller accesses main memory frequently, CPU processes other jobs at a slower speed. Good for block-based fast I/O devices (e.g., disk and tape, usually block size is 512, 1K, 2K, or 4K bytes).

→ Memory-Mapped I/O

Ranges of main memory addresses are directly mapped to separate device registers. So, I/O is main memory access. To perform I/O, CPU reads or writes data in these specific memory locations. After the read/write, the data keep flow through the system bus and duplicated in the device controller's registers. Then, device performs I/O and the data transferred to the controller's registers are also duplicated in the main memory.

- Additional memory overhead. Good for very fast byte (word)-based devices (e.g., video display)

4. Storage Structure

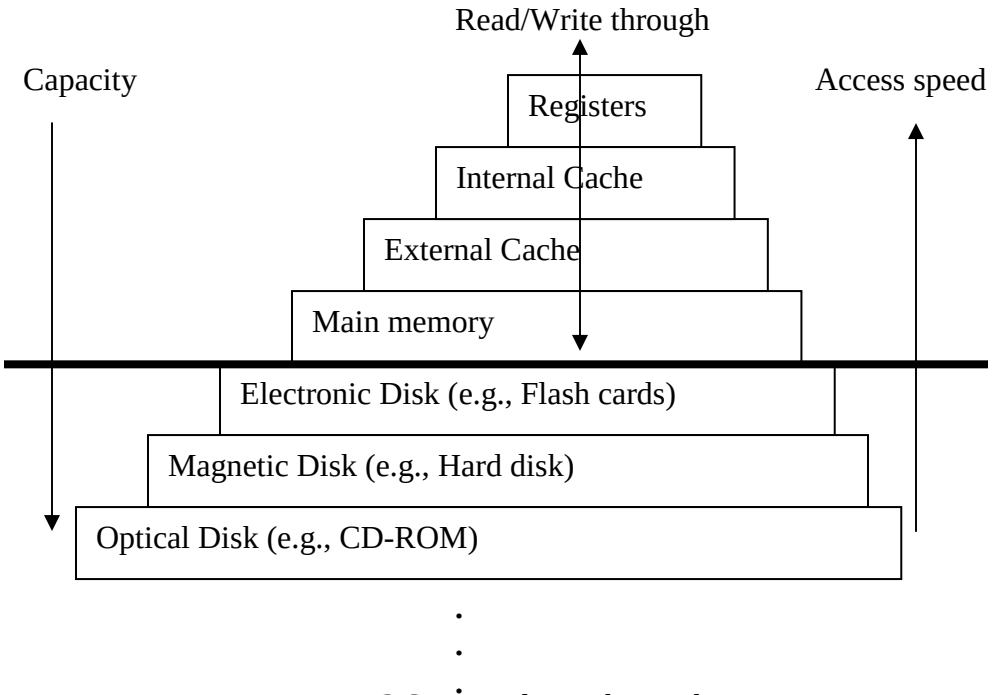

Figure 3 Storage-device hierarchy

→ Coherency and Consistency problem: multiple copies of data (see "Read/Write through on Figure 3)

- The problem is more complex in tightly-coupled systems

- The problem is much more complex in loosely-coupled systems

5. Hardware Protection

→ Example 1. An application program is trying to write on OS kernel code in main memory.

→ Example 2. An application program is trying to write on another program in main memory.

→ Example 3. An application runs infinite loop and hold CPU time infinitely.

→ Example 4. An application program prints indefinitely long data repeatedly.

→ Solution: Modern computer systems attack this problem by using "dual-mode operation". In dual-mode systems, only OS can access I/O devices, memory, and CPU in its "monitor" mode. User's process will never be allowed to directly access system resources. Application programs access these resources through "system calls" indirectly (c.f., OS needs hardware "mode bit" for this solution: 8088 does not have it, after 80486 supports this bit and, as a result, Windows/NT and OS/2 support dual mode operation).

→ OS may needs CPU timer and two additional memory registers (base and limit)