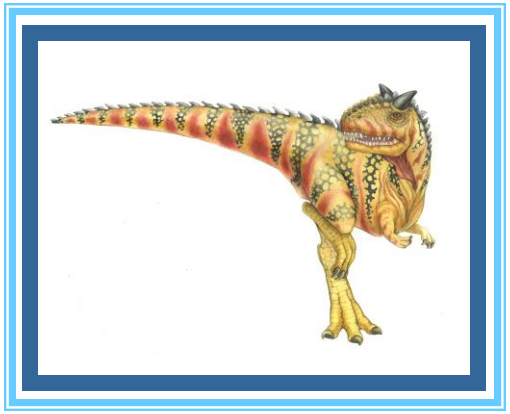
Computer-System Organization



Computer System Organization

- A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access between components and shared memory (Figure 1.2).

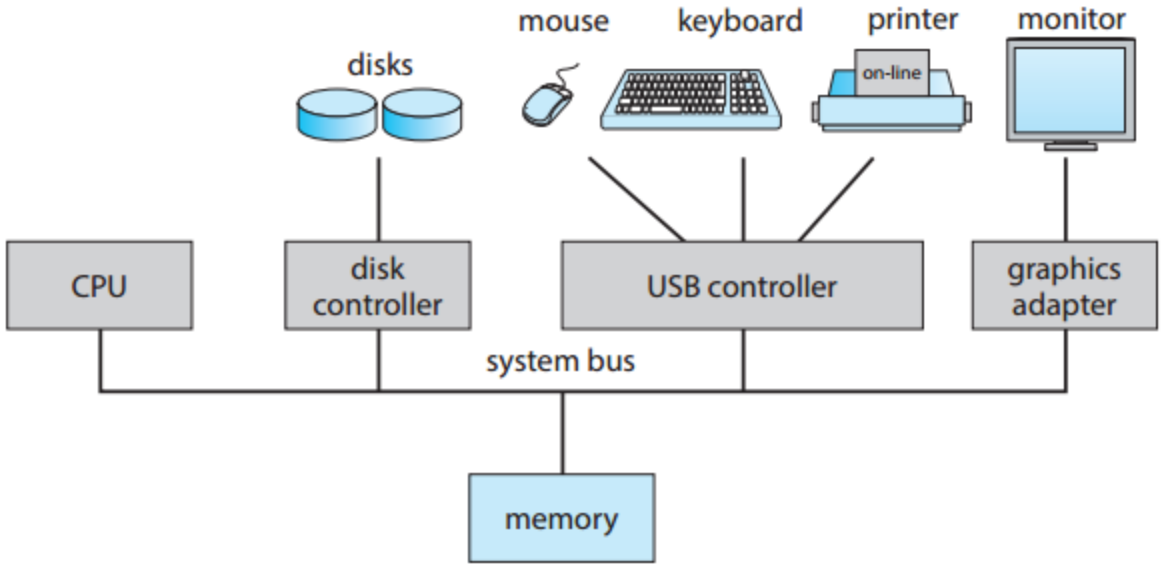


Figure 1.2 A typical PC computer system.

Computer System Organization

- Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display).
- Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect.
- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage

Computer System Organization

- Typically, operating systems have a device driver for each device controller.
- This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.
- The CPU and the device controllers can execute in parallel, competing for memory cycles.
- To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

Interrupts

- Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller.
- The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”).
- The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation.
- The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read.

Interrupts

- For other operations, the device driver returns status information such as “write completed successfully” or “device busy”.
- But how does the controller inform the device driver that it has finished its operation? This is accomplished via an interrupt.
- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
- Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

Interrupts

- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.
- The fixed location usually contains the starting address where the service routine for the interrupt is located.
- The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.
- Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common.
- The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information.

Interrupts

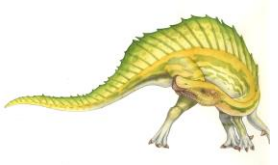
- The routine, in turn, would call the interrupt-specific handler.
- However, interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed.
- The interrupt routine is called indirectly through the table, with no intermediate routine needed.
- Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices.
- This array, or interrupt vector, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.
- Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

Interrupts

- The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt.
- If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning.
- After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

Interrupt Implementation

- The basic interrupt mechanism works as follows. The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction.
- When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the interrupt-handler routine by using that interrupt number as an index into the interrupt vector.
- It then starts execution at the address associated with that index.

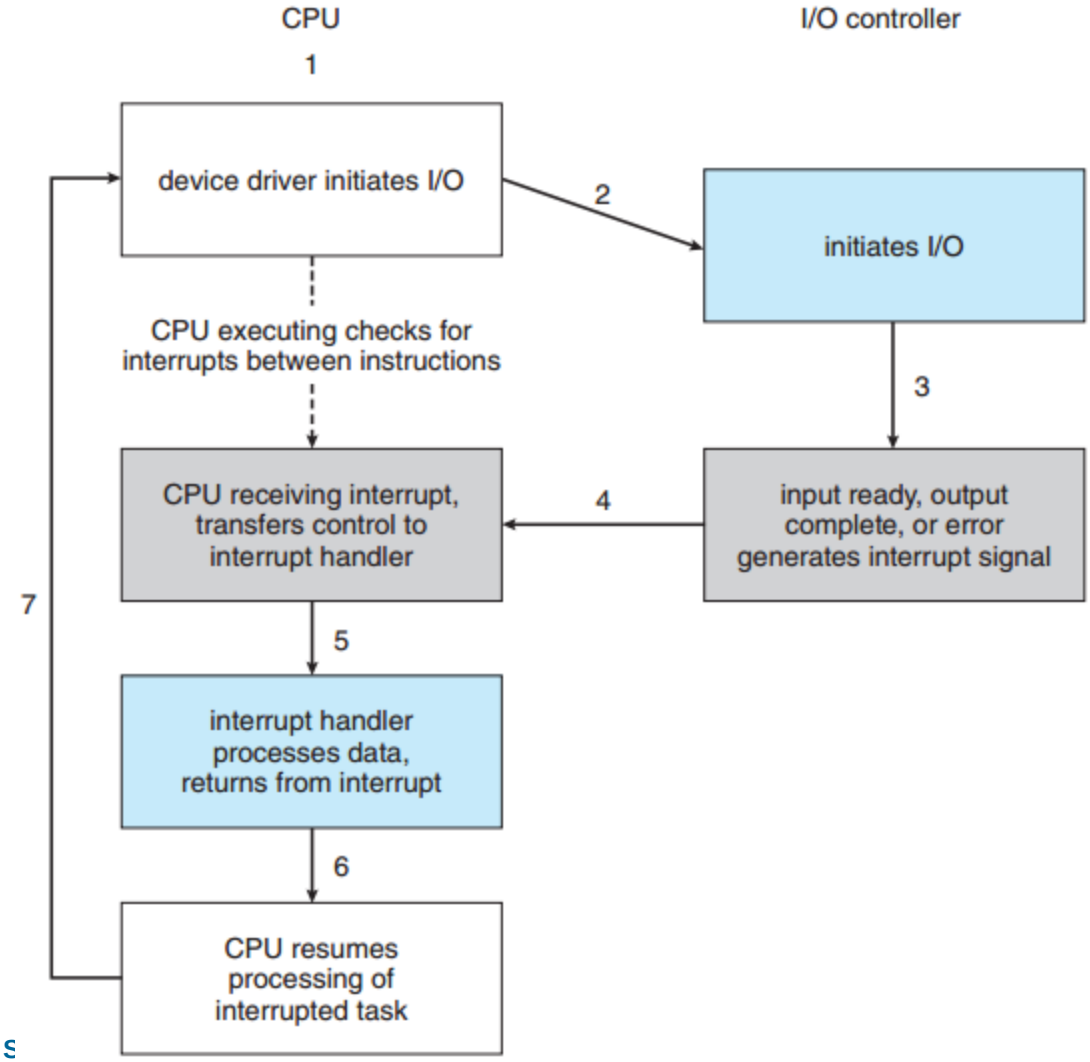


Interrupt Implementation

- The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return_from_interrupt instruction to return the CPU to the execution state prior to the interrupt.
- We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt handler, and the handler ***clears*** the interrupt by servicing the device.

Interrupt Implementation

- Figure 1.4 summarizes the interrupt-driven I/O cycle.



Interrupt Implementation

- The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service.
- In a modern operating system, however, we need more sophisticated interrupt handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

- In modern computer hardware, these three features are provided by the CPU and the interrupt-controller hardware.

Interrupt Implementation

- Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.
- The second interrupt line is **maskable** : it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
- The maskable interrupt is used by device controllers to request service.
- The interrupt mechanism also implements a system of **interrupt priority levels**.
- These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

Interrupt Implementation

- In summary, interrupts are used throughout modern operating systems to handle asynchronous events.
- Device controllers and hardware faults raise interrupts. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities.
- Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.