2 Computer System Structure

The operating system must ensure the correct operation of the computer system. To ensure that user programs will not interfere with the proper operation of the system, the hardware must provide appropriate mechanisms to ensure correct behavior. We need to know the basic computer architecture that makes it possible to write a functional operating system.

2.1 Computer-System Operation

A modern, general-purpose computer system consists of a **CPU** and a number of device controllers that are connected through a common bus that provides access to shared memory (Figure 2.1) . Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

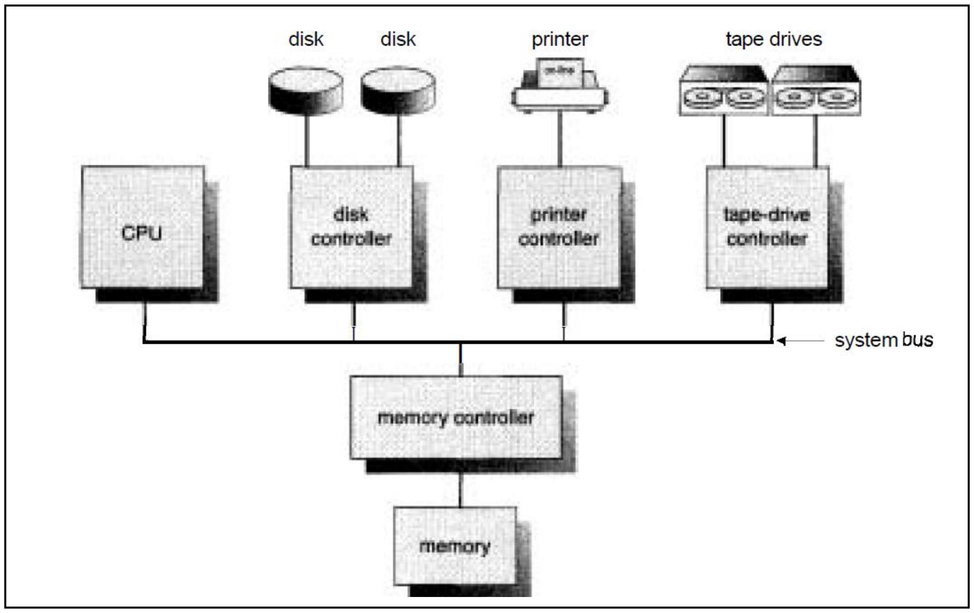

Figure **2.1** A modern computer system.

For a computer to start running needs to have an initial program. This initial program tends to be simple. It is stored in read-only memory (ROM). The bootstrap program must locate and load into memory the operating-system kernel. The operating system then starts executing the first process, such as "init," and waits for some event to occur. The occurrence of an event is usually signaled by an interrupt from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU. Software may trigger an interrupt by executing a special operation called a system call.

Modern operating systems are interrupt driven. Events are almost always signaled by the occurrence of an interrupt or a trap. A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt. 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 handling this transfer would be to invoke a generic routine to examine the interrupt information; the routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly, and, given that only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead. The interrupt routine is then called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory. 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 device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.

The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. 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.

2.2 I/O Structure

A general-purpose computer system consists of a CPU and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, there may be more than one attached device. For instance, the **small computer-systems interface (SCSI)** controller can have seven or more devices attached to it. 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. The size of the local buffer within a device controller varies from one controller to another, depending on the particular device being controlled.

2.2.1 I/O Interrupts

To start an I/O operation, the CPU loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take. For example, if it finds a read request, the controller will start the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the CPU that it has finished its operation. It accomplishes this communication by triggering an interrupt. This situation will occur, in general, as the result of a user process requesting I/O. Once the I/O is started, two courses of action are possible. In the simplest case, the I/O is started; then, at I/O completion, control is returned to the user process. This case is known as **synchronous** I/O. The other possibility, called **asynchronous** I/O, returns control to the user program without waiting for the I/O to complete. The I/O then can continue while other system operations occur (Figure 2.2).

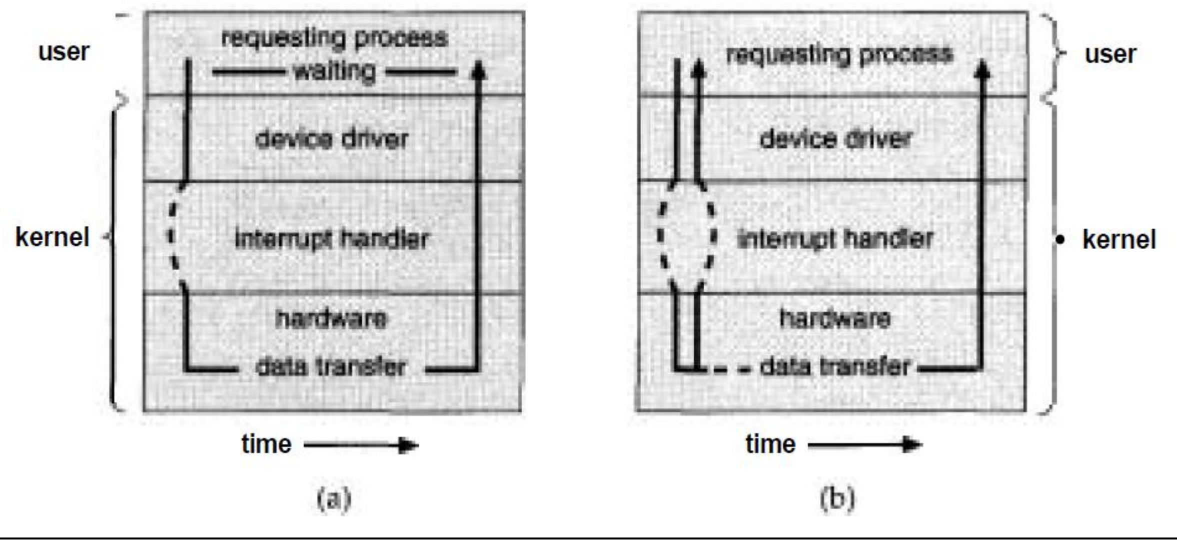

Fig. 2.2 Two I/O methods: (a) synchronous, and (b) asynchronous

Waiting for I/O completion may be accomplished in one of two ways. Some computers have a special **wait** instruction that idles the CPU until the next interrupt. Machines that do not have such an instruction may have a wait loop:

```
Loop: jmp Loop
```

This tight loop simply continues until an interrupt occurs, transferring control to another part of the operating system. Such a loop might also need to poll any I/O devices that do not support the interrupt structure; instead, these devices simply set a flag in one of their registers and expect the operating system to notice that flag.

If the CPU always waits for I/O completion, at most one I/O request is outstanding at a time. Thus, whenever an I/O interrupt occurs, the operating system knows exactly which device is interrupting. On the other hand, this approach excludes concurrent I/O operations to several devices, and also excludes the possibility of overlapping useful computation with I/O.

A better alternative is to start the I/O and then to continue processing other operating-system or user program code. A system call is then needed to allow the user program to wait for I/O completion, if desired. If no user programs are ready to run, and the operating system has no other work to do, we still require the **wait** instruction or idle loop, as before. We also need to be able to keep track of many I/O requests at the same time. For this purpose, the operating system uses a table containing an entry for each I/O device: the **device-status table** (Figure 2.3). Each table entry indicates the device's type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device. Since it is possible for other processes to issue requests to the same device, the operating system will also maintain a wait queue-a list of waiting requests-for each I/O device.

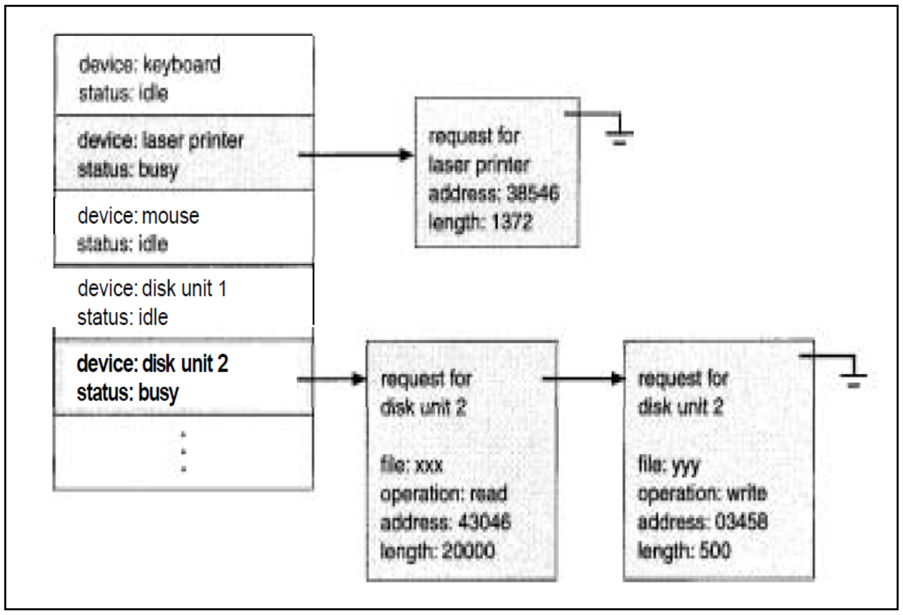

Fig. 2.3 Device-status table

An I/O device interrupts when it needs service. When an interrupt occurs, the operating system first determines which I/O device caused the interrupt. It then indexes into the I/O device table to determine the status of that device, and modifies the table entry to reflect the occurrence of the interrupt. For most devices, an interrupt signals completion of an I/O request. If there are additional requests waiting in the queue for this device, the operating system starts processing the next request. Finally, control is returned from the I/O interrupt. If a process was waiting for this request to complete (as recorded in the device-status table), we can now return control to it. Otherwise, we return to whatever we were doing before the I/O interrupt: to the execution of the user program (the program started an I/O operation and that operation has now finished, but the program has not yet waited for the operation to complete) or to the wait loop (the program started two or more I/O operations and is waiting for a particular one to finish, but this interrupt was from one of the other operations). In a time-sharing system, the main advantage of asynchronous I/O is increased system efficiency. While I/O is taking place, the system CPU can be used for processing or starting I/O to other devices. Because I/O can be slow compared to processor speed, the system makes efficient use of its facilities.