Chapter Two

2.1. Computer-System Operation

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 to shared memory (Figure 2.1). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). 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.

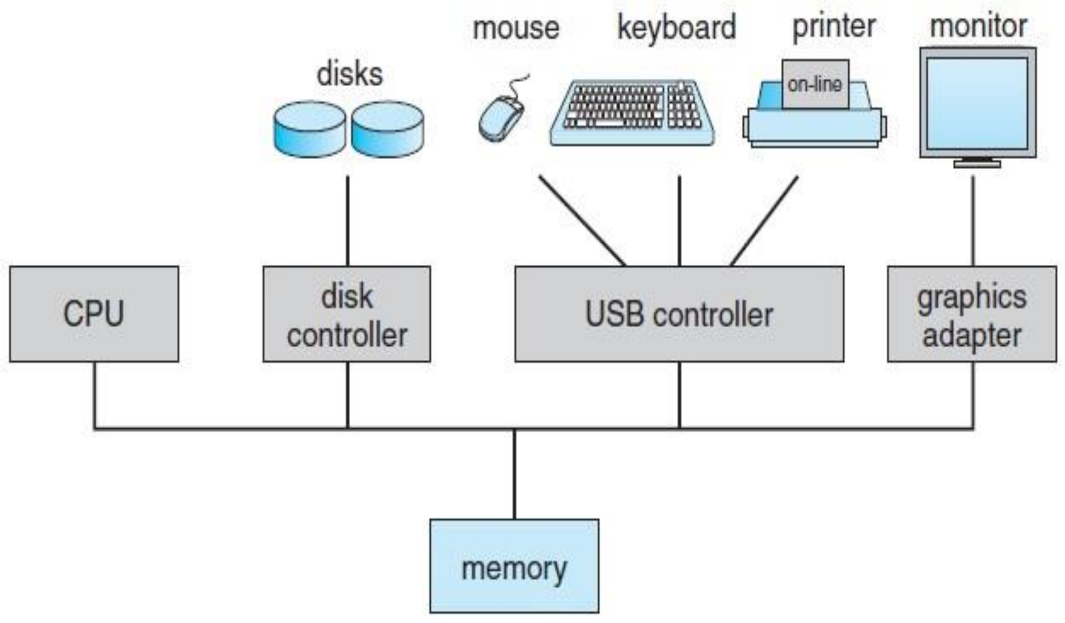

Figure 2-1 A modern computer system

For a computer to start running for instance, when it is powered up or rebooted it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running. On UNIX, the first system process is “init,” and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

2.2. I/O Interrupts

The occurrence of an event is usually signalled 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, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**). 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 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. Since only a predefined number of interrupts is possible, 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 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.

2.3. Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**. Computers use other forms of memory as well. We have already mentioned read-only memory, ROM) and electrically erasable programmable read-only memory, EEPROM). Because ROM cannot be changed, only static programs, such as the bootstrap program described earlier, are stored there. The immutability of ROM is of use in game cartridges. EEPROM can be changed but cannot-be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution. A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program. Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory. Many programs then use the disk as both the source and the destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.In a larger sense, however, the storage structure that we have described consisting of registers, main memory, and magnetic disks is only one of many possible storage systems. Others include cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility. The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another other properties being the same then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. In addition to differing in speed and cost, the various storage systems are either volatile or non-volatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **non-volatile storage** for safe keeping. In the hierarchy shown in Figure 2.2, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are non-volatile.

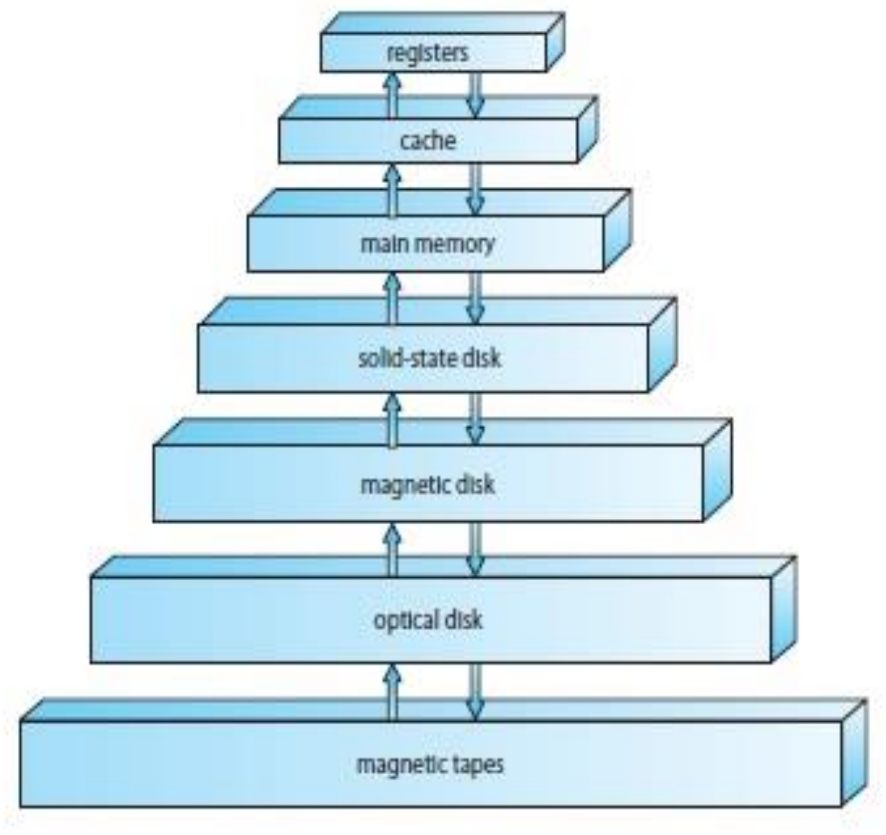

Figure 2-2 Storage-device hierarchy.

2.4.Hardware Protection

To improve system utilization, the O.S began to share system resources among several programs simultaneously. Multi programming put several programs in memory at the same time. This sharing created both improved utilization and increased problems. When the system was run without sharing an error in a program could cause problems for only the one program that was running. With sharing many process could be affected by a bug in one program.

2.4.1. Dual-Mode Operation

To ensure proper operation we must protect the O.S and all programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken is to H/W support to allow as differentiating among various modes of executions. Therefore we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit called mode bit is added to H/W to indicate the current mode; monitor (0) or user (1). With the mode bit we are able to distinguish between an execution that is done on behalf of the O.S, and one that is done on behalf of the user. The dual mode of operation provides us with the means for protecting the O.S from errant users and errant users from one another. The H/W allows privileged instructions to be executed in only monitor mode.

2.4.2. I/O Protection

To prevent a user from performing illegal I/O we define all I/O instructions to be privileged instructions. Thus user cannot issue I/O instructions directly they must do it through the O.S. For I/O protection to be complete we must be sure that a user program can never gain control of the Computer in monitor mode.

2.4.3. Memory Protection

To ensure correct operation we must protect the interrupt vector from modification by a user program. Also we must protect the interrupt service routines in the O.S from modification. What we need to separate each program's memory space is an ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers usually a base and a limit as illustrated in figure 2.3.

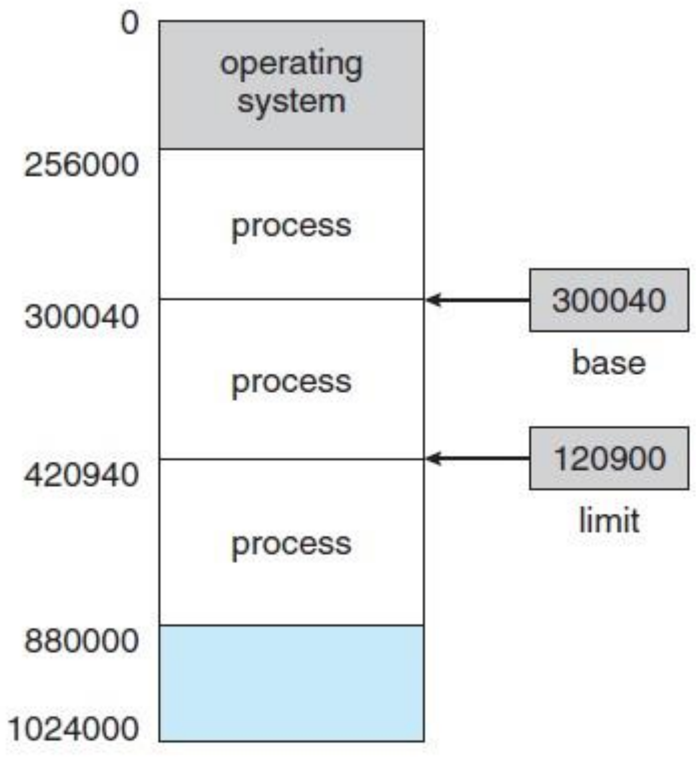

Figure 2-3 A base and a limit register define a logical address space

The base register holds the smallest legal physical memory address; the limit register contains the size of the range. For example if the base register holds 300040 and limit register is 120900 then the program can legally access all addresses from 300040 through 420940 inclusive.

The CPU H/W comparing every address generated in user mode with registers accomplishes this protection. Any attempt by a program executing in user mode to access monitor memory or other user's memory or other users memory results in a trap to the monitor which treats the attempt as a fatal error (figure 2.4).This scheme prevents the user program from modifying the code or data structures of either the O.S or other users.

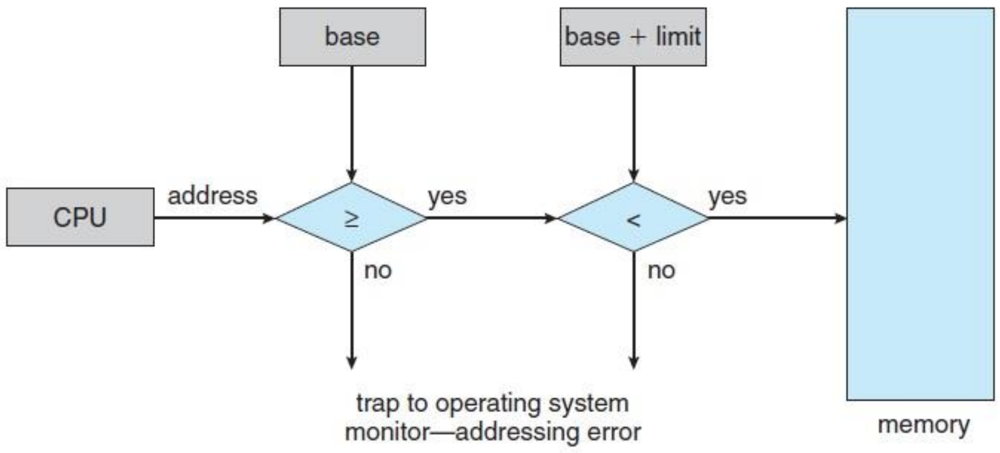

Figure 2-4 Hardware address protection with base and limit registers

The base and limit registers can be loaded by only the O.S which uses a special privileged instruction. Since privileged instructions can be executed in only monitor mode therefore only O.S can load the base and limit registers. This scheme allows the monitor to change the value of the registers but prevents user programs from changing the registers contents.

2.4.4. CPU Protection

The third piece of the protection is ensuring that the O.S maintains control, we must prevent a user program from an infinite loop, and never returning control to the O.S. To achieve this goal we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (1/60 second) or variable (from 1 millisecond to 1 second). To control the timer the O.S sets the counter, according to fixed-rate clock. Every time that the clock ticks the counter is decremented. When the counter reaches (0) on interrupt occurs, and control transfers automatically to the O.S, which may treat the interrupt as a fatal error or may give the program more time.

2.5. System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions. Before we discuss how an operating system makes system calls available, let’s first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call. Possible error conditions for each operation can require additional system calls. When the program tries to open the input file, for example, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors; depending on the output device (for example, no more disk space).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call).