1.5 OPERATING SYSTEM STRUCTURE

Since operating system is a very large and complex software, supports large number of functions. It should be developed as a collection of several smaller modules with carefully defined inputs, outputs and functions rather than a single piece of software. In this section, we will examine different operating system structure.

1.5.1 Layered Structure Approach

The operating system architecture based on layered approach consists of number of layers (leve'ls), each built on top of lower layers. The bottom layer is the hardware; the highest layer is the user interface. The first system constructed in this way was the THE system built by E.W. Dijkestra (1968) and his students. The THE system was a simple batch operating system which had 32k of 27 bit words.

The system supported 6 layers in (Figure 7).

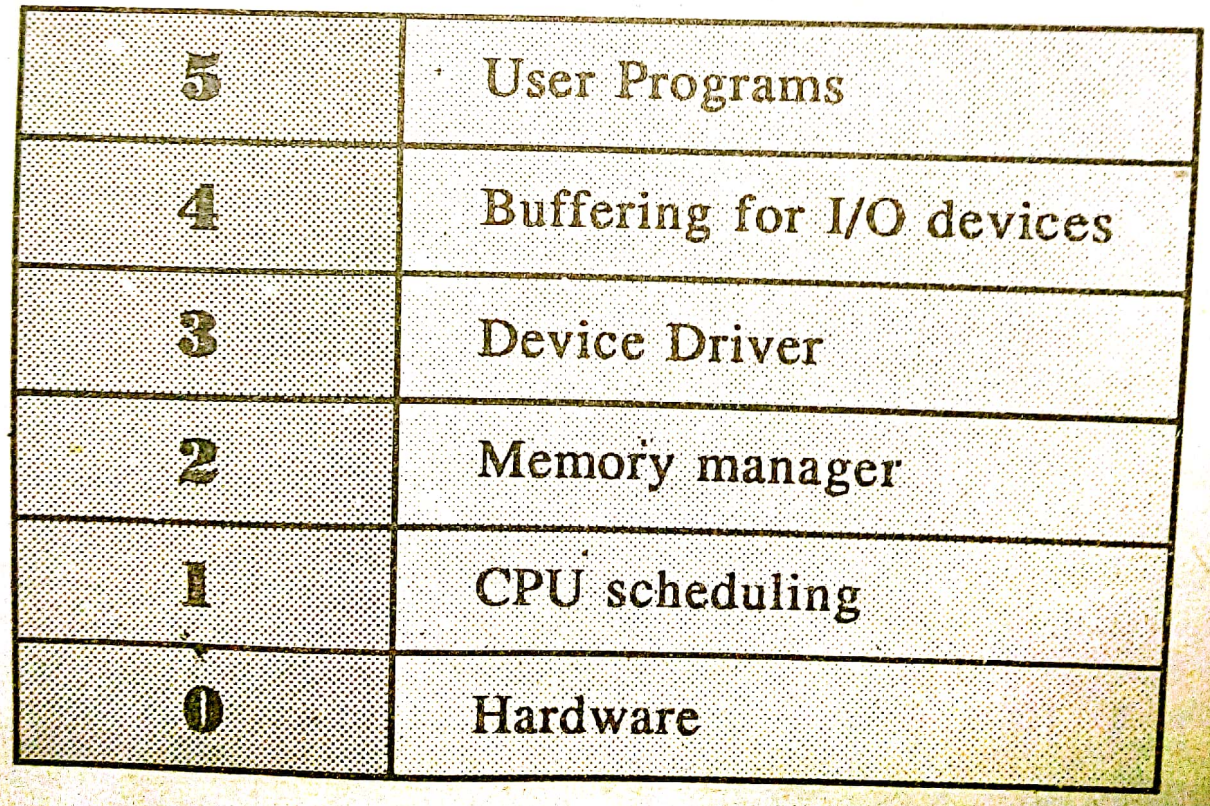

Figure 7: The layered structure of THE operating system

As shown in figure 7, layer 0 dealt with hardware; the higher layer layer1 handled allocation of jobs to processor. The next layer implemented memory management. The memory management scheme was virtual memory (to be discussed in Unit 3). Level 3 contained the

1.5.2 Kernel Approach

Kernel is that part of operating system which directly makes interface with hardware system. Its main functions are:

- To provide a mechanism for creation and deletion of processes
- To provide processor scheduling, memory management and I/O management
- To provide mechanism for synchronisation of processes so that processes synchronize their actions.
- To provide mechanism for interprocess communication.

The UNIX operating system is based on kernel approach (figure 8). It consists of two separatable parts: (i) Kernel (ii) System Programs

As shown in the figure 8, kernel is between system programs and hardware. The kernel supports the file system, processor scheduling, memory management and other operating system functions through system calls. UNIX operating system supports a large number of system calls for process management and other operating system functions. Through these system calls program utilises the services of operating system (kernel).

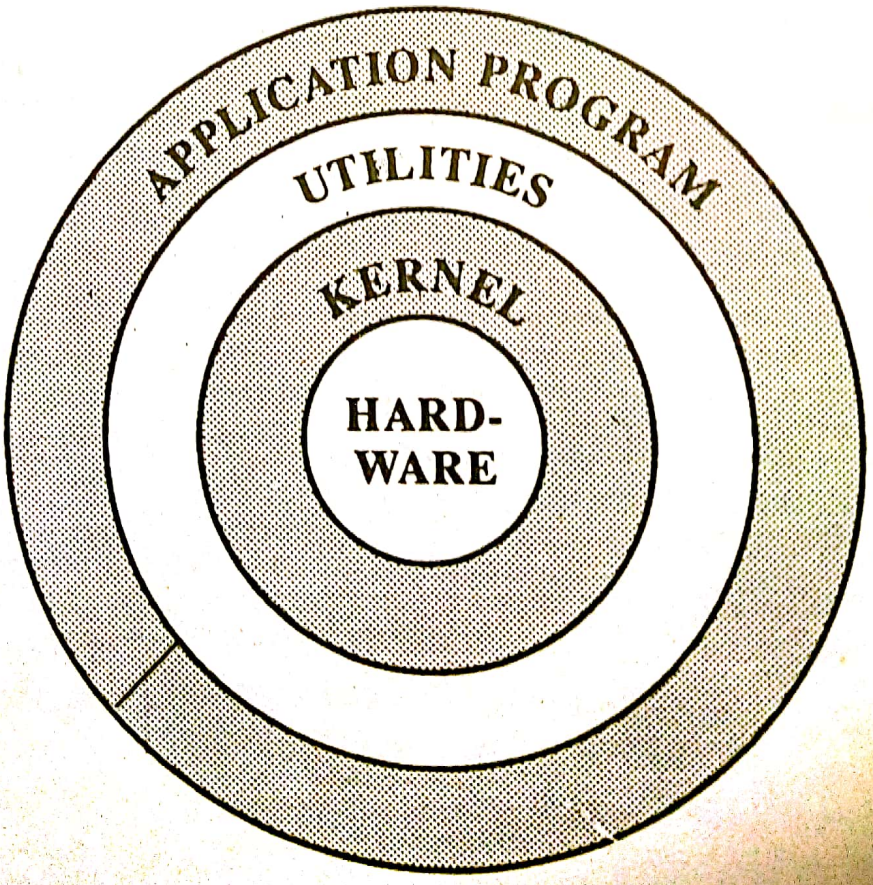

Figure 8: UNIX operating system structure

type of situation is analogous to communication line of telephone company which enables separate and isolated conversations over the same wire(s).

The following figure illustrates this concept.

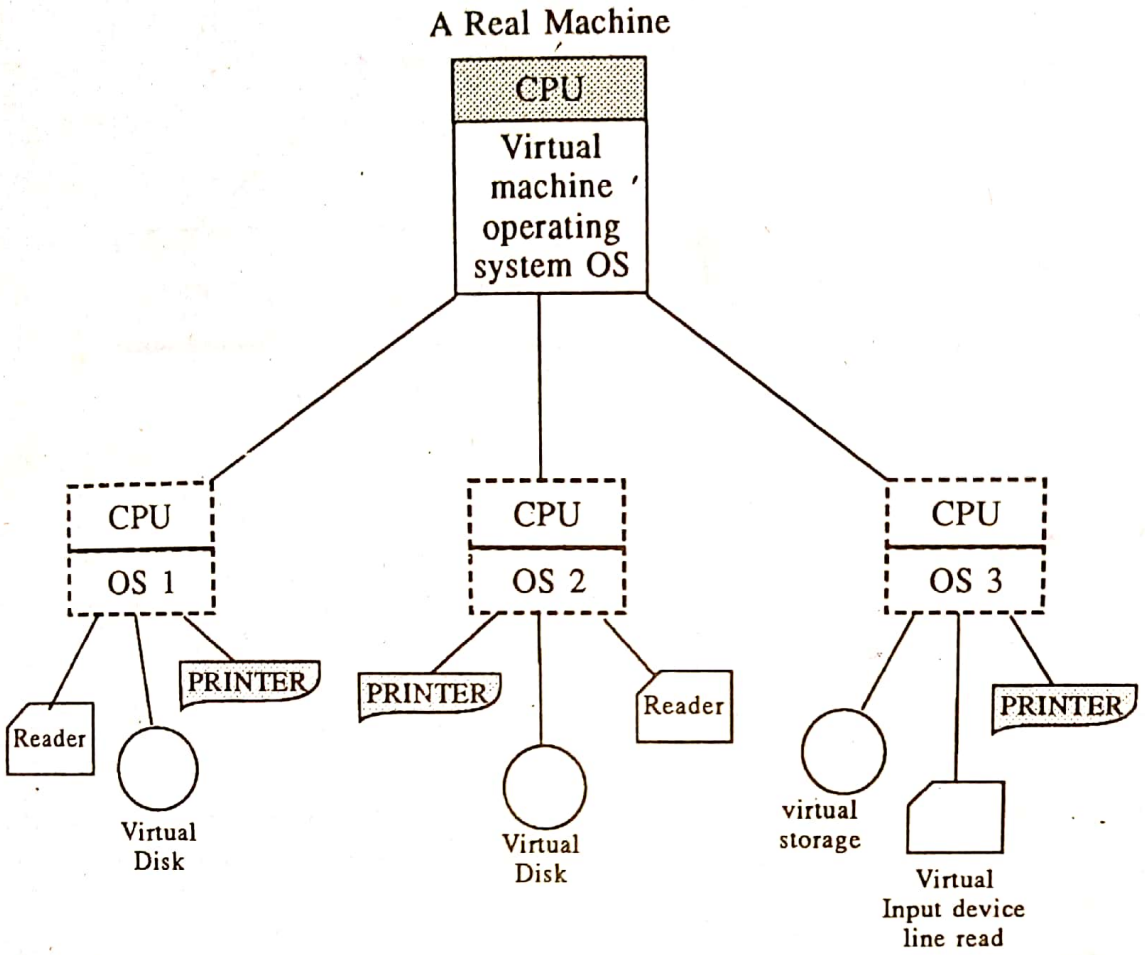

Figure 9: Creation of several virtual machines by a single physical machine

From the user's view point, virtual machine can be made to appear to very similar to existing real machine or they can be entirely different. An important aspect of this technique is that each user can run operating system of his own choice. This fact is depicted by OS_1 (Operating System 1), OS_2, OS_3 etc. in figure 9.

To understand this concept, let us try to understand the difference between conventional multiprogramming system (figure 10) and virtual machine multiprogramming (figure 11). In conventional multiprogramming processes are allocated a portion of the real machines resources. The same machine resources are distributed among several processes.

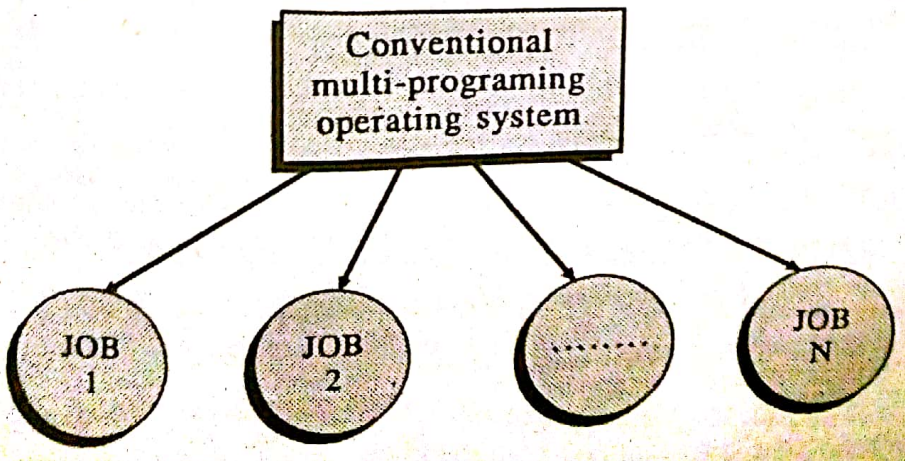

Figure 10 : Conventional multiprogramming

1.5.4 Client-Server Model

VM/370 gains much in simplicity by moving a large part of the traditional operating system code (implementing the extended machine) into a higher layer itself. VM/370 is still a complex program because simulating a number of virtual 370s is not that simple (especially if you want to do it efficiently).

A trend in modern operating systems is to take this idea of moving code up into higher layers even further, and remove as much as possible from the operating system, leaving a minimal kernel. The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (now known as the client process) sends the request to a server process, which then does the work and sends back the answer.

In this model, shown in figure 12, all the kernel does is to handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service or memory service. This way, each part becomes small and manageable. Furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

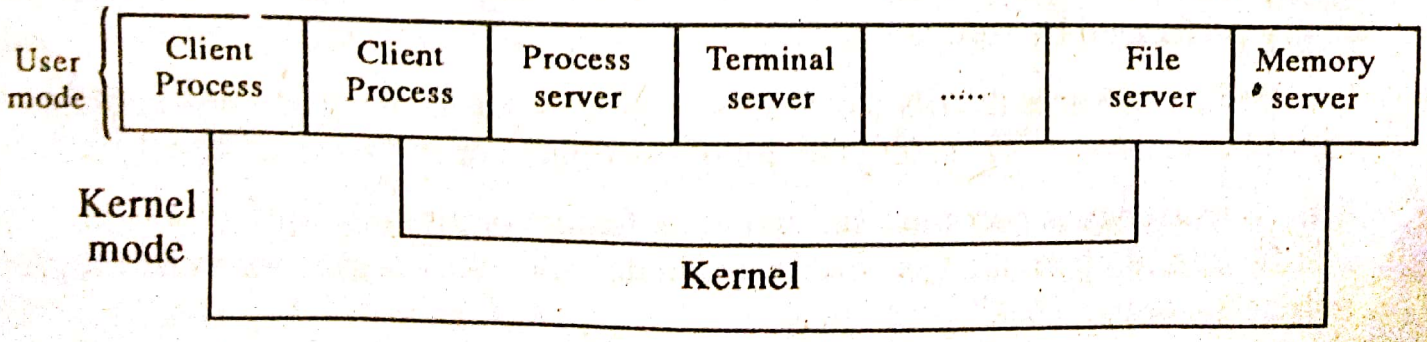

Figure 12: The Client-Server Model

Another advantage of the client-server model is its adaptability to use in distributed systems (figure 13). If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

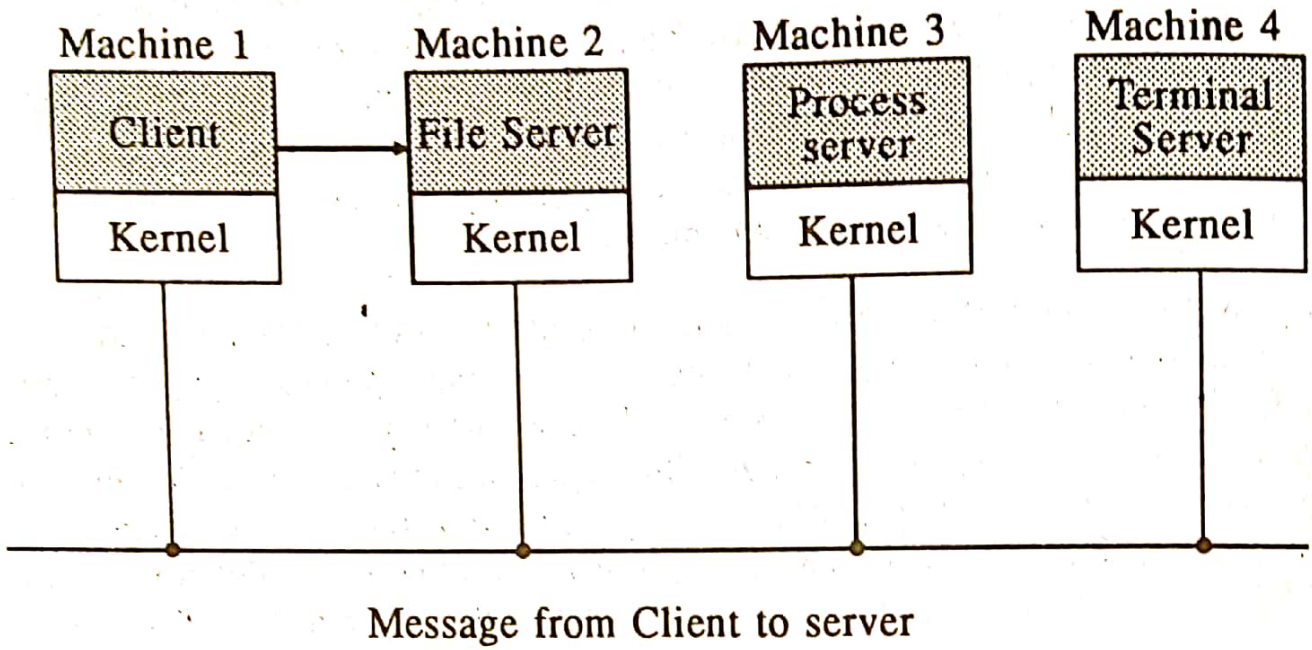

Figure 13: The Client-Server Model in a distributed System

The picture painted above of a kernel that handles only the transport of messages from clients to servers and back is not completely realistic. Some operating system functions (such as loading commands into the physical I/O device registers) are difficult, if not impossible, to do from user-space programs. There are two ways of dealing with this problem. One way is to have some critical server processes (e.g. I/O device drivers) actually run in kernel mode, with complete access to all the hardware, but still communicate with other processes using the normal message mechanism.

The other way is to build a minimal amount of **mechanism** into the kernel, but leave the **policy** decisions up to servers in user space. For example, the kernel might recognize that a message sent to a certain special address means to take the contents of that message and load it into the I/O device registers for some disk, to start a disk read. In this example, the kernel would not even inspect the bytes in the message to see if they were valid or meaningful; it would just blindly copy them into the disk's device registers. (Obviously some scheme for limiting such messages to authorised processes only must be used.) The split between mechanism and policy is an important concept; it occurs again and again in operating systems in various contexts.