2) No model answer is provided.

3) The X window system does not define any particular style of interface but provides a mechanism for supporting many styles. X is also network-based in contrast with other GUIs, or the X architecture is based on the premise that an application can run on one computer while the graphical presentation of the application's output and the responses from the user can occur on another computer.

Check Your Progress 4

1) No model answer is given.

2) No model answer is given.

1.11 FURTHER READINGS

1) *Communication of ACM*, April 1993.

2) *Object Orientation: Concepts, Languages, Databases. User Interfaces*, Khosafian, Setrag & Razmik Abnours, New York; Wiley & Sons, 1990.

UNIT 2 INTRODUCTION TO OPERATING SYSTEM

Structure **Page Nos.**

2.0 Introduction 34

2.1 Objectives 35

2.2 What is an Operating System? 35

2.3 Evolution of Operating System 36
- 2.3.1 Serial Processing
- 2.3.2 Batch Processing
- 2.3.3 Multiprogramming

2.4 Operating System Structure 41
- 2.4.1 Layered Structure Approach
- 2.4.2 Virtual Machine
- 2.4.3 Client-Server Model
- 2.4.4 Kernel Approach

2.5 Classification of Advanced Operating System 47
- 2.5.1 Architecture Driven Operating System
- 2.5.2 Application Driven Operating System

2.6 Characteristics of Modern Operating System 51
- 2.6.1 Microkernel Architecture
- 2.6.2 Multithreading
- 2.6.3 Symmetric Multiprocessing

2.7 Summary 53

2.8 Solutions/ Answers 53

2.9 Further Readings 54

2.0 INTRODUCTION

An operating system is a system software which may be viewed as an organized collection of software consisting of procedures for operating a computer and providing an environment for execution of programs. It acts as an interface between users and the hardware of a computer system.

There are many important reasons for studying operating systems. Some of them are:

1) User interacts with the computer through the operating system in order to accomplish his task since it is his primary interface with a computer.

2) It helps users to understand the inner functions of a computer very closely.

3) Many concepts and techniques found in the operating system have general applicability in other applications.

The introductory concepts and principles of an operating system will be the main issues for discussion in this unit. The unit starts with the basic definition and then goes on to explain the stages of evolution of operating systems. It further gives details of several approaches to operating system design.

In the last two subsections of the unit we classify an advanced operating system and explain some characteristics of modern operating systems.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- List stages of evolution of operating systems:
- Classify different types of operating systems, and
- Compare different approaches to operating system design.

2.2 WHAT IS AN OPERATING SYSTEM?

An operating system is an essential component of a computer system. The primary objectives of an operating system are to make the computer system convenient to use and to utilise computer hardware in an efficient manner.

An operating system is a large collection of software, which manages the resources of the computer system, such as **memory, processor, file system** and **input/output devices.** It keeps track of the status of each resource and decides who will have control over computer resources, for how long and when. The positioning of an operating system in the overall computer system is shown in *Figure 1.*

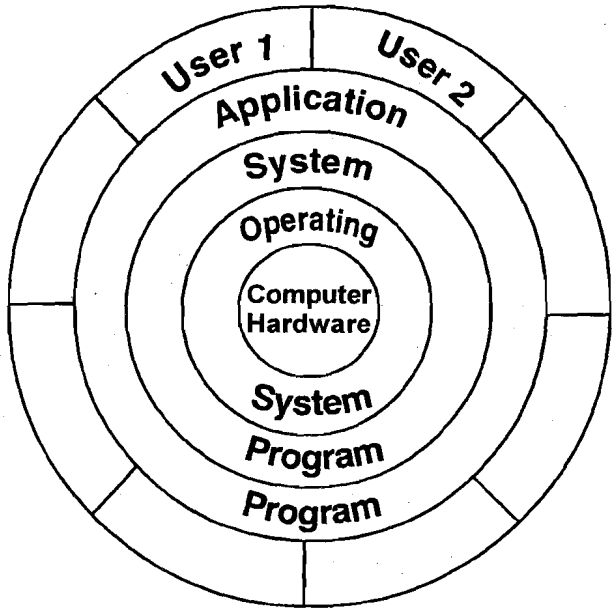


Figure 1: Component of Computer System

From the Figure, it is clear that the operating system directly controls computer hardware resources. Other programs rely on facilities provided by the operating system to gain access to computer system resources. There are two ways one can interact with the operating system:

1) By means of operating **System Call** in a program

2) Directly by means of **Operating System Commands.**

System Call: System calls provide the interface to a running program and the operating system. The user program receives operating system services through the set of system calls. Earlier these calls were available in assembly language instructions but now a day these features are supported through high level languages like C, Pascal

etc., which replace assembly language for system programming. The use of system calls in C or Pascal programs very much resemble pre-defined functions or subroutine calls.

As an example of how system calls are used, let us consider a simple program to copy data from one file to another. In an interactive system, the following system calls will be generated by the operating system:

- Prompt message for inputting two file names and reading it from the terminal.
- Open source and destination file.
- Prompt error message in case the source file cannot be open because it is protected against access or because the destination file cannot be created since there is already a file with this name.
- Read the source file
- Write into the destination file
- Display status information regarding various Read/Write error conditions. For example, the program may find that the end of the file has been reached or that there was a hardware failure. The write operation may encounter various errors, depending upon the output device (no more disk space, physical end of tape, printer out of paper and so on).
- Close both files after the entire file is copied.

As we can observe, a user program makes heavy use of the operating system. All interaction between the program and its environment must occur as the result of requests from the program to the operating system.

Operating System Commands: Apart from system calls, users may interact with the operating system directly by means of operating system commands.

For example, if you want to list files or sub-directories in **MS-DOS**, you invoke **dir** command. In either case, the operating system acts as an interface between users and the hardware of the computer system. The fundamental goal of computer systems is to solve user problems. Towards this goal computer hardware is designed. Since the bare hardware alone is not very easy to use, programs (software) are developed. These programs require certain common operations, such as controlling peripheral devices. The command function of controlling and allocating resources are then brought together into one piece of software; the operating system.

To see what operating systems are and what operating systems do, let us consider how they have evolved over the years. By tracing their evolution, we can identify the common elements of operating systems and examine how and why they have developed as they have.

2.3 EVOLUTION OF OPERATING SYSTEMS

An operating system may process its task serially (sequentially) or concurrently (several tasks simultaneously). It means that the resources of the computer system may be dedicated to a single program until its completion or they may be allocated among several programs in different stages of execution. The feature of the operating system to execute multiple programs interleaved fashion or different time cycles is called **multiprogramming systems.** In this section, we will try to trace the evolution of the operating system. In particular, we will describe **serial processing, batch processing** and **multiprogramming.**

2.3.1 Serial Processing

Programming in 1's and 0's (machine language) was quite common for early computer systems. Instructions and data used to be fed into the computer by means of console switches or perhaps through a hexadecimal keyboard. Programs used to be started by loading the program computer register with the address of the first instruction of a program and its result (program) used to be examined by the contents of various registers and memory locations of the machine. Therefore, programming in this style caused a low utilisation of both users and machine.

The advent of Input/output devices, such as punched cards, paper tape and language translators (Compiler /Assemblers) brought a significant step in computer system utilisation. Programs started being coded into programming language first changed into object code (binary code) by translator and then automatically loaded into memory by a program called **loader.** After transferring control to the loaded program, the execution of a program begins and its result gets displayed or printed. Once in memory, the program may be re-run with a different set of input data.

The process of development and preparation of a program in such an environment is slow and cumbersome due to serial processing and numerous manual processing. In a typical sequence first the editor is called to create a source code of user program written in programming language, then the translator is called to convert a source code into binary code and then finally the loader is called to load the executable program into the main memory for execution. If syntax errors are detected, the whole process must be redone from the beginning.

The next development was the replacement of card-decks with standard input/output and some useful library programs, which were further linked with user programs through system software called **linker.** While there was a definite improvement over machine language approach, the serial mode of operation is obviously not very efficient. This results in low utilization of resources.

2.3.2 Batch Processing

Utilisation of computer resources and improvement in programmer's productivity was still a major problem. During the time that tapes were being mounted or the programmer was operating the console, the CPU was sitting idle.

The next logical step in the evolution of the operating system was to automate the sequencing of operations involved in program execution and in the mechanical aspects of program development. Jobs with similar requirements were batched together and run through the computer as a group. For example, suppose the operator received one FORTRAN program, one COBOL program and another FORTRAN program. If he runs them in that order, he would have to set up for FORTRAN program environment (loading the FORTRAN compiler tapes), then set up COBOL program and finally FORTRAN program again. If he runs the two FORTRAN programs as a batch, however, he could set up only once for FORTRAN thus saving operator's time.

Batching similar jobs brought utilisation of system resources quite a bit. But there were still problems. For example, when a job is stopped, the operator would have to notice that fact by observing the console, determine why the program stopped and then load the card reader or paper tape reader with the next job and restart the computer. During this transition from one job to the next, the CPU sat idle.

To overcome this idle time, a small program called a **resident monitor** was created which is always resident in the memory. It automatically sequenced one job to another job. The resident monitor acts according to the directives given by a programmer through **control cards, which** contain information like marking of job's beginnings and endings, commands for loading and executing programs, etc. These commands belong to **job control language**. These job control language commands are included with user program and data. Here is an example of job control language commands.

$COB	- Execute the COBOL compiler
$JOB	- First card of a job
$END	- Last card of a job
$LOAD	- Load program into memory
$RUN	- Execute the user program

Figure 2 shows a sample card deck set up for a simple batch system.

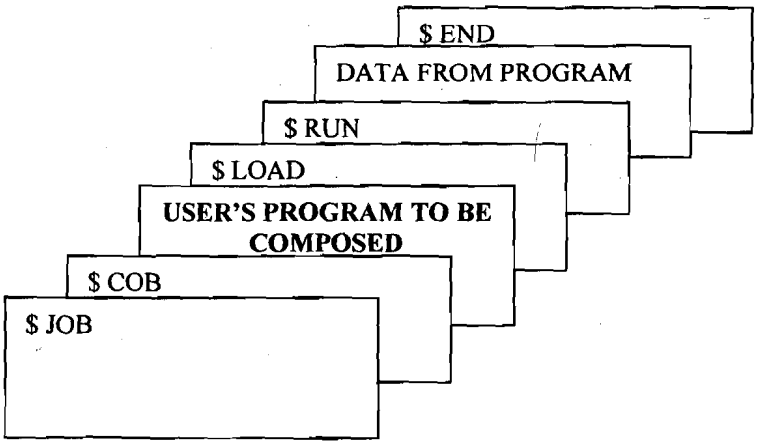


Figure 2: Card Deck for Cobol Program for a Simple Batch System

With sequencing of program execution mostly automated by batch operating system, the speed discrepancy between the fast CPU and the comparatively slow input/output devices such as card readers, printers emerged as a major performance bottleneck. Even a slow CPU works in the microsecond range, with millions of instructions per second. A fast card reader, on the other hand, might read 1200 cards per minute. Thus, the difference in speed between the CPU and its input/output devices may be three orders of magnitude or more.

The relative slowness of input/output devices can mean that the CPU is often waiting for input/output. As an example, an assembler or compiler may be able to process 300 or more cards per second. A fast card reader, on the other hand, may be able to read only 1200 cards per minute. This means that assembling or compiling a 1200 card program would require only 4 seconds of CPU time but 60 seconds to read. Thus, the CPU would be idle for 56 out of 60 seconds or 93.3 per cent of the time. The resulting CPU utilization is only 6.7 per cent. The process is similar for output operations. The problem is that while an input/output is occurring, the CPU is idle, waiting for the input/output to complete; while the CPU is executing, input/output devices are idle.

Over the years, of course, improvements in technology resulted in faster input/output devices. But CPU speed increased even faster. According to Moore's Law, CPU speed is getting doubled every 18 months. Therefore, the need was to increase the throughput and resources utilization by overlapping input/output and processing operations. Channels, peripheral controllers and later dedicated input/output processors brought a major improvement in this direction. DMA (Direct Memory Access) chip which directly transfers the entire block of data from its own buffer to main memory without intervention by the CPU was a major development. While the CPU is executing, DMA can transfer data between high-speed input/output devices and the main memory. The CPU requires to be interpreted per block only by DMA. Apart from DMA, there are two other approaches to improving system performance by overlapping input, output and processing. These are called **buffering** and **spooling**.

Buffering is a method of overlapping input, output and processing of a single job. The idea is quite simple. After data has been read and the CPU is about to start operating on it, the input device is instructed to begin the next input immediately. The CPU and input device are then both busy. With luck, by the time the CPU is ready for the next data item, the input device will have finished reading it. The CPU can then begin processing the newly read data, while the input device starts to read the following data. Similarly, this can be done for output. In this case, the CPU creates data that is put into a buffer until an output device can accept it.

If the CPU is, on the average, much faster than an input device, buffering will be of little use. If the CPU is always faster, then it always finds an empty buffer and has to wait for the input device. For output, the CPU can proceed at full speed until, eventually, all system buffers are full. Then the CPU must wait for the output device. This situation occurs with **input/output bound** jobs where the amount of input/output device, the speed of execution is controlled by the input/output device, not by the speed of the CPU.

A more sophisticated form of input/output buffering is called **SPOOLING** (Simultaneous Pheripheral Operation On Line) which essentially uses the hard disk (secondary memory) as a very large buffer (*Figure 3*) for reading and for storing output files.

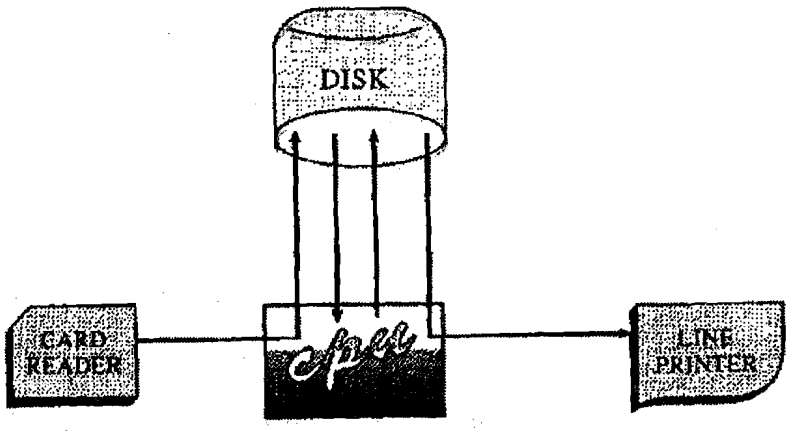


Figure 3: Spooling

Buffering overlaps input, output and processing of a single job **whereas Spooling allows CPU to overlap the input of one job with computation and output of other jobs.** Therefore this approach is better than buffering. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job.

2.3.3 Multiprogramming

Buffering and Spooling improve system performance by overlapping the input, output and computation of a single job, but both of them have their limitations. A single user cannot always keep CPU or I/O devices busy at all times. Multiprogramming offers a more efficient approach to increase system performance. In order to increase the resource utilisation, systems supporting multiprogramming approach allow more than one job (program) to reside in the memory to utilise CPU time at any moment. More number of programs competing for system resources better will mean better resource utilisation.

The idea is implemented as follows. The main memory of a system contains more than one program *(Figure 4)*.

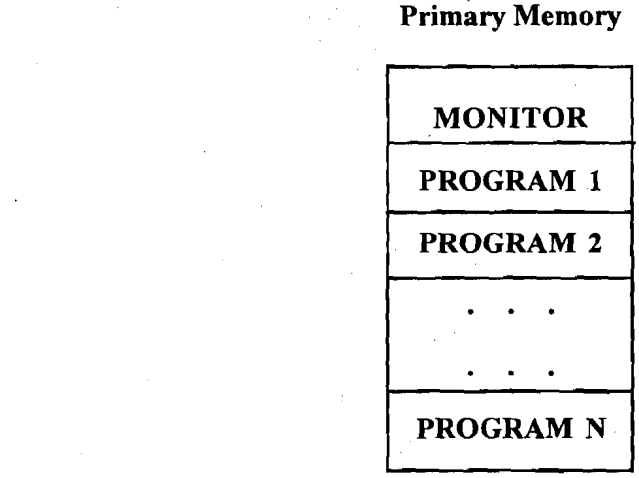
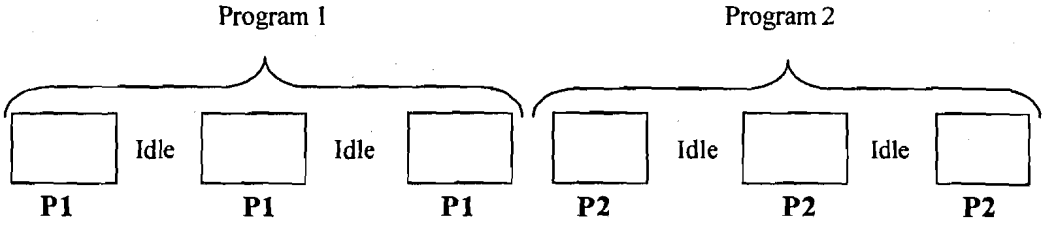
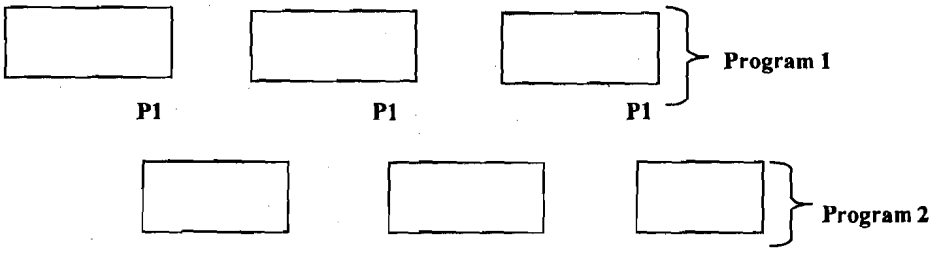


Figure 4: Memory Layout in Multiprogramming Environment

The operating system picks one of the programs and starts executing. During execution of program 1 it needs some I/O operation to complete in a sequential execution environment (*Figure 5a*). The CPU would then sit idle whereas in a multiprogramming system, (*Figure 5b*) the operating system will simply switch over to the next program (program 2).



(a): Sequential Execution



(b): Execution in Multiprogramming Environment

Figure 5: Multiprogramming

When that program needs to wait for some I/O operation, it switches over to program 3 and so on. If there is no other new program left in the main memory, the CPU will pass its control back to the previous programs.

Multiprogramming has traditionally been employed to increase the resources utilisation of a computer system and to support multiple simultaneously interactive users (terminals).

Compared to the operating system, which supports only sequential execution, the multiprogramming system requires some form of CPU and memory management strategies, which will be discussed in the next section.

2.4 OPERATING SYSTEM STRUCTURE

Since operating system is a very large and complex software, supports a 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.

2.4.1 Layered Structure Approach

The operating system architecture based on layered approach consists of a number of layers (levels), 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 (*Figure 6*).

5	User Programs
4	Buffering for I/O Devices
3	Device Driver
2	Memory Manager
1	CPU Scheduling
0	Hardware

Figure 6: The layered structure of THE operating system

As shown in *Figure 6*, layer 0 dealt with hardware; the higher layer 1 handled allocation of jobs to processor. The next layer implemented memory management. Level 3 contained the device driver for the operator's console. By placing it, as well as I/O buffering, at level 4, above memory management, the device buffers could be placed in virtual memory. The I/O buffering was also above the operator's console, so that I/O error conditions could be output to the operator's console.

The main advantages of the layered approach is modularity which helps in debugging and verification of the system easily. The layers are designed in such a way that it uses operation and services only of a layer below it. A higher layer need not know how these operations are implemented, only what these operations do. Hence each layer hides implementation details from higher-level layers. Any layer can be debugged without any concern about the rest of the layer.

The major difficulty with the layered approach is definition of a new level i.e. how to differentiate one level from another. Since a layer can use the services of a layer below it, it should be designed carefully. For example, the device driver for secondary memory must be at a lower level than the memory management routines since memory management requires the ability to use the backing store.

2.4.2 Virtual Machine

It is a concept which creates the illusion of a real machine. It is created by a virtual machine operating system that makes a single real machine appear to be several real machines. This type of situation is analogous to the communication line of Telephone Company, which enables separate and isolated conversations over the same wire(s).

The following *Figure* illustrates this concept.

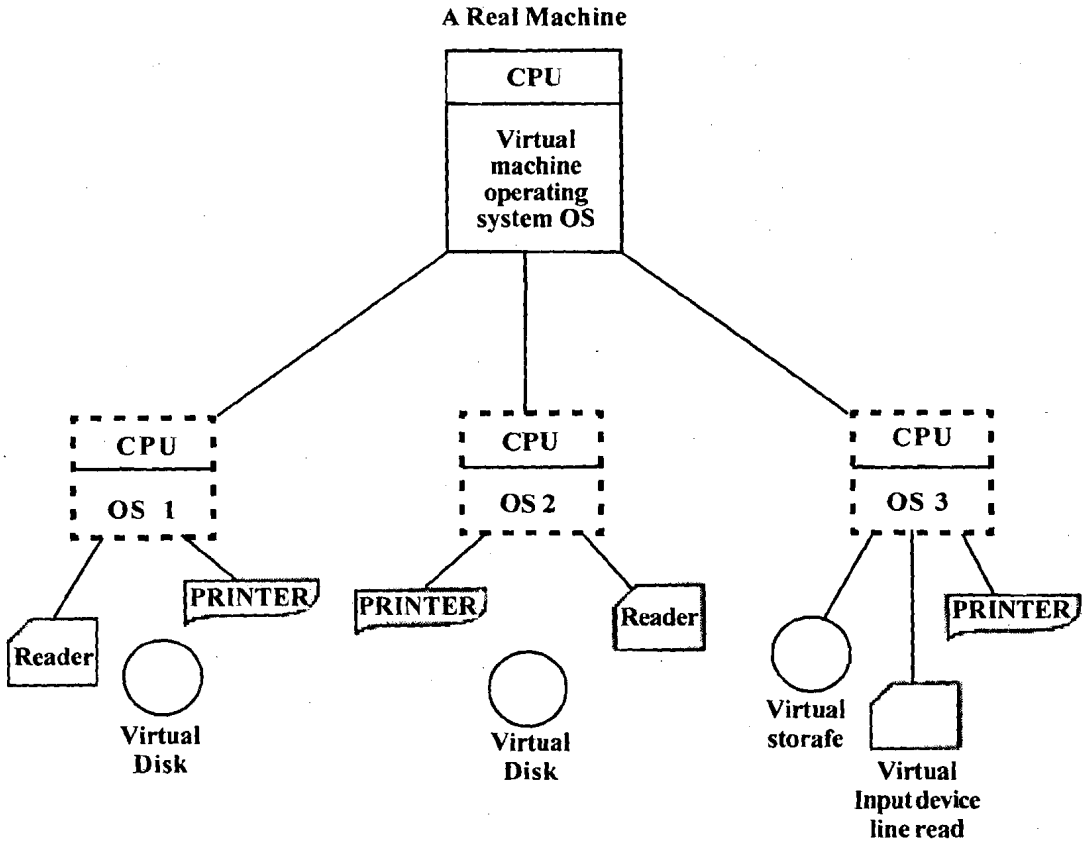


Figure 7: Creation of several virtual machines by a single physical machine

From the user's viewpoint, a virtual machine can be made to appear very similar to an existing real machine or they can be entirely different. An important aspect of this technique is that each user can run the operating system of his own choice. This fact is depicted by OS_1 (Operating System 1), OS_2, OS_3 etc. as in *Figure 7*.

To understand this concept, let us try to understand the difference between conventional multiprogramming system (Figure 8) and virtual machine multiprogramming (*Figure 9*). In conventional multiprogramming, processes are allocated a portion of the real machine resources. The same machine resources are distributed among several processes.

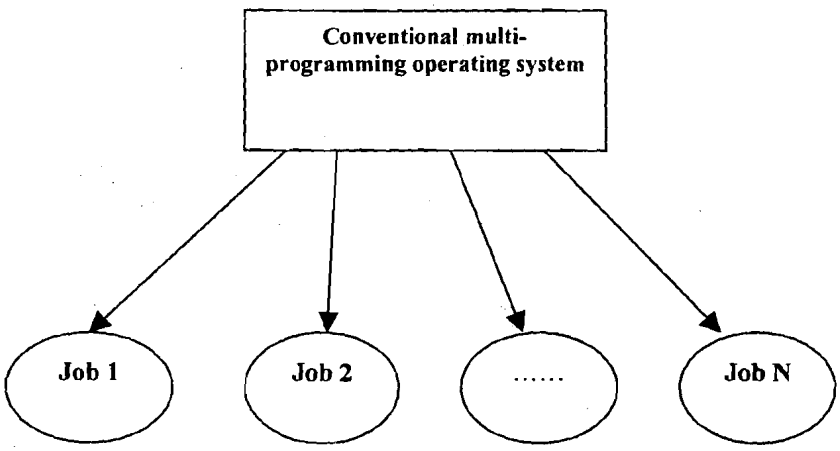


Figure 8: Conventional multiprogramming

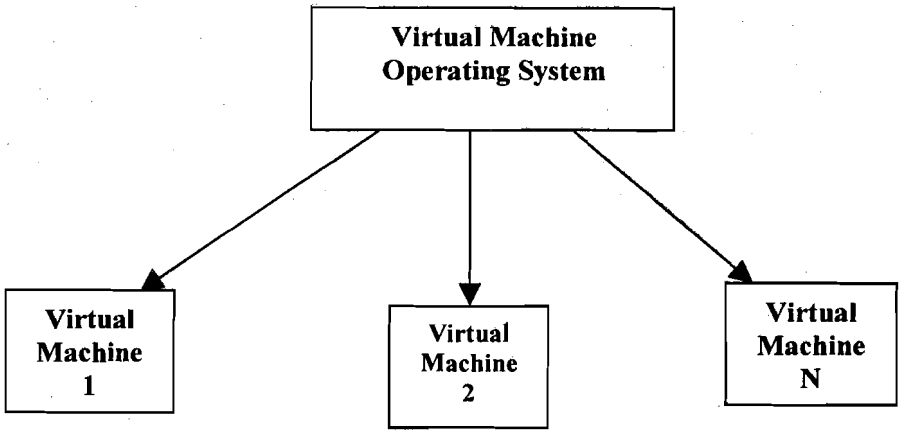


Figure 9: Virtual Machine Multiprogramming

In the virtual multiprogramming system, a single real machine gives the illusion of several virtual machines, each having its own virtual processor, storage and I/O devices possibly with much larger capacities.

The virtual machine has many uses and advantages:

1) Concurrent running of dissimilar operating systems by **different users.**

2) **Elimination of certain** conversion problems.

3) **Software development:** Programs can be developed and debugged for machine configurations that is different from those of host (for example virtual operating system VM/370 can produce virtual 370 that are different from the real 370 such as larger main memory).

4) **Security and Privacy:** The high degree of separation between independent virtual machines aids in ensuring privacy and security. The most widely used operating system in this category is VM/370. It manages IBM/370 computer and creates the illusion that each of several users has a complete system 370 (including wide range of I/O devices) which can run different operating systems at once, each of them on its own virtual machine.

It is also possible through software to share files existing on physical disk and much information through virtual communication software.

The virtual machines are created by sharing the resources of the physical computer. CPU scheduling can be used to share the CPU and make it appear that users have their own processor. Users are thus given their own virtual machine. They can run on their virtual machines any software desired. The virtual machine software is concerned with multiprogramming multiple virtual machines onto a physical machine but need not consider any other software support from user.

The heart of the system, known as the **virtual machine monitor,** runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in *Figure 7*. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are exact copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the hardware. In fact, different virtual machines can, and usually do, run different operating systems. Some run one of the descendants of OS/360 for batch processing, while other ones run a simple, single-user, interactive system called CMS (Conversational Monitor System) for time-sharing users.

When a CMS program executes a system call, the call is trapped to the operating system in its own virtual machine, not to VM/370; just as it would if it were running on a real machine instead of a virtual one. CMS then issues the normal hardware I/O instructions for reading its virtual disk or whatever is needed to carry out the call. These I/O instructions are trapped by VM/370, which then performs them as part of its simulation of the real hardware. By making a complete separation of the functions of multiprogramming and providing an extended machine, each of the pieces can be mush simpler and more flexible.

The virtual machine concept has several advantages. Notice that there is complete protection. Each machine is completely isolated from all other virtual machines, so there is no problem with protection. On the other hand, there is no sharing. To provide sharing, two approaches have been implemented. First, it is possible to share a minidisk. This scheme is modeled after a physical shared disk, but implemented by software. With this technique, files can be shared. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. Again, the network is modeled after physical communication networks, but implemented in software.

Such a virtual machine system is a perfect vehicle for operating systems research and development. Normally changing an operating system is a difficult process. Since operating systems are large and complex programs, it is difficult to be sure that a change in one point does not cause obscure bugs in some other part. This situation can be particularly dangerous because of the power of the operating system. Since the operating system executes in monitor mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

But the operating system runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use, while changes are made and tested. This is commonly called system development time. Since it makes the system unavailable to users, system development time is often scheduled late at night or on weekends.

A virtual machine system can eliminate much of this problem. System programmers are given their own virtual machine and system development is done on the virtual machine, instead of on a physical machine. The normal system operation seldom need be disrupted for system development.

2.4.3 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 stimulating 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 up into a 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 10,* all that the kernel does is to handle the communication between clients and servers. By splitting the operating system up into parts, each part is made to handle one facet of the system, such as file service, process service, and 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.

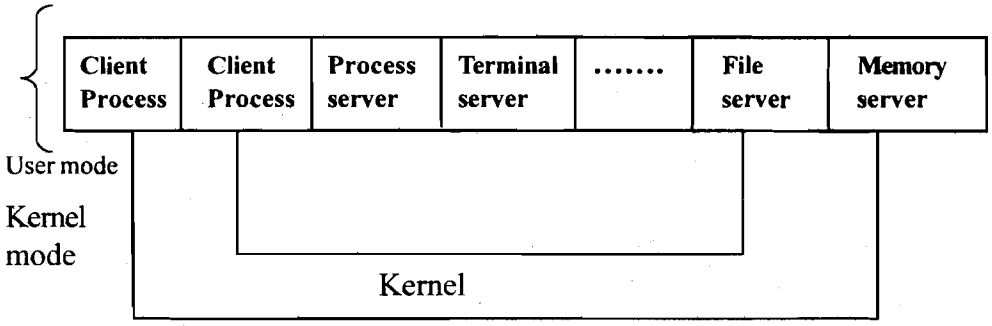


Figure 10: The Client-Server Model

Another advantage of the client-server model is its adaptability to use in distributed systems (*Figure 11*). 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.

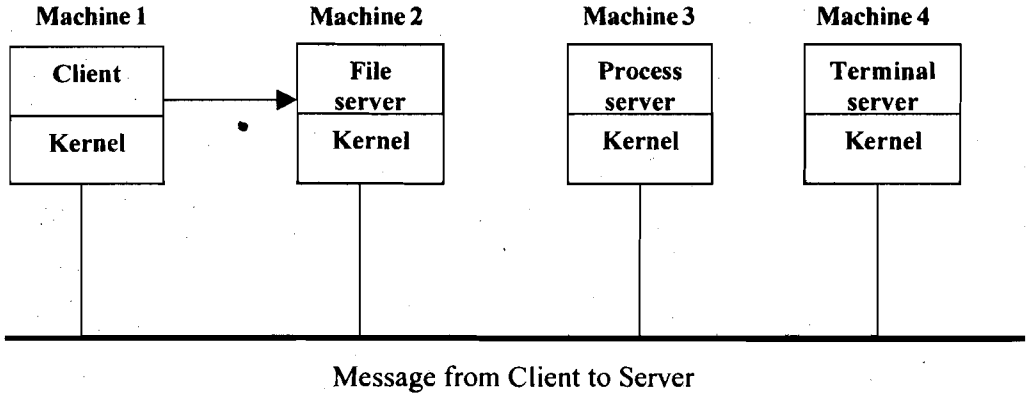


Figure 11: The Client-Server Model in a distributed System

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 from 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 authorized 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.

2.4.4 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 a mechanism for synchronisation of processes so that processes synchronise their actions.
- To provide a mechanism for interprocess communication.

The UNIX operating system is based on kernel approach (*Figure 12*). It consists of two separable parts: (i) Kernel (ii) System Programs.

As shown in the *Figure 12,* the 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. The UNIX operating system supports a large number of system calls for process management and other operating system functions. Through these system calls, the program utilises the services of the operating system (kernel).

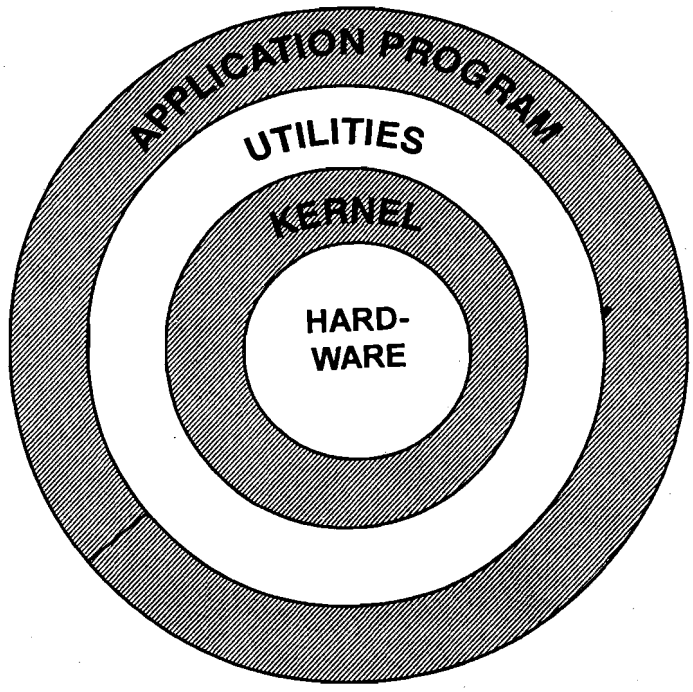


Figure 12: UNIX operating system structure

Check Your Progress 1

1) What is a system call?

.....

.....

.....

.....

2) Define the essential difference between:

   a) Spooling

   b) Buffering

2.5 CLASSIFICATION OF ADVANCED OPERATING SYSTEMS

The drive for advanced operating systems has come from two directions. First, it has come from advances in the architecture of multicomputer systems and is now driven by a wide variety of high-speed architectures. Hardware design of extremely fast parallel and distributed systems is fairly well understood. These architectures offer great potential for speed up but they also present a substantial challenge to operating system designers. The following *Figure 13* gives a broad classification of the advanced operating system.

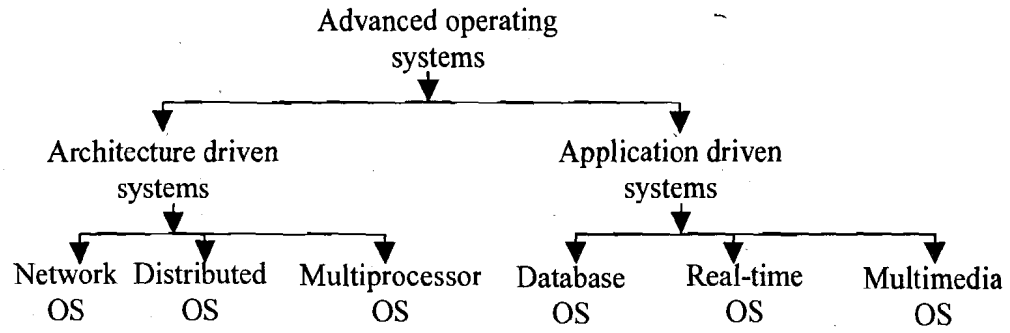


Figure 13: Classification of Advanced OS

A second class advanced operating system is driven by applications. There are several important applications that require special operating system support, as a requirement as well as for efficiency. General-purpose operating systems are too broad in nature and inefficient and fail to provide adequate support for such applications. Three specific applications, namely, multimedia operating systems, database systems and real-time systems, have received considerable attention in the past and the operating system issues for these systems have been extensively examined.

2.5.1 Architecture Driven Operating System

A brief discussion of three operating systems in the category of the Architecture Driven OS follows:

Network Operating System

A network operating system is a collection of software and associated protocols that allows a set of autonomous computers, which are interconnected by a computer network, to be used together in a convenient and cost-effective manner. In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to an other machine.

Some of typical characteristics of network operating systems which make it different from a distributed operating system (discussed in the next section) are the followings:

- Each computer has its own private operating system instead of running part of a global system wide operating system.
- Each user normally works on his/her own system: using a different system requires some kind of remote login, instead of having the operating system dynamically allocate processes to CPUs.
- Users are typically aware of where each of their files are kept and must move file from one system to another with explicit file transfer commands instead of having file placement managed by the operating system.

The system has little or no fault tolerance; if 5% of the personal computers crash, only 5% of the users are out of business.

The network operating system offers many capabilities including:

- Allowing users to access the various resources of the network hosts.
- Controlling access so that only users in the proper authorization are allowed to access particular resources.
- Making the use of remote resources appear to be identical to the use of local resources.
- Providing up-to-the minute network documentation on-line.

As we said earlier, the key issue that distinguishes a network operating system from a distributed one is how aware the users are of the fact that multiple machines are being used. The visibility occurs in three primary areas; file system, protection and program execution.

Distributed Operating System

Distributed operating systems are operating systems for a network of autonomous computers connected by a communication network. A distributed operating system controls and manages the hardware and software resources of a distributed system such that its users view the entire system as a powerful monolithic computer system. When a program is executed in a distributed system, the user is not aware of where the program is executed or of the location of the resources accessed.

The basic issues in the design of a distributed operating system are the same as in a traditional operating system, viz., process synchronisation, deadlocks, scheduling, file systems, interprocess communication, memory and buffer management, failure recovery, etc. However, several idiosyncrasies of a distributed system, namely, the lack of both shared memory and a physical global clock, and unpredictable communication delays make the design of distributed operating systems much more difficult.

Network operating systems focus on the use of remote services and resources existing on a network of computer systems. Distributed operating systems focus on effective utilization of resources in distributed computing environments.

Distributed systems provide many advantages concerning cost-effectiveness of both computations and resources. The primary advantages are:

- Resource sharing
- Reliability
- Communication
- Incremental growth

Resource sharing has been the main motivation for distributed systems. The earliest form of a distributed system was a computer network which enabled the use of specialized hardware and software resources by geographically distant users. Resource sharing continues to be an important aspect of distributed systems today. However, the nature of distribution and sharing of resources has changed due to advances in networking technology. Sharing of resources is now equally meaningful in a local area network (LAN), for example sharing laser printers in the lab.

One aspect of reliability is availability of a resource despite failures in a system. A distributed environment can offer enhanced availability of resources through redundancy of resources and communication paths. For example, availability of a disk resource can be increased by having two or more disks located at different sites in the system. If one disk is unavailable due to a disk or site failure, a program can use some other disk. The availability of a data resource, e.g., a file, can be similarly enhanced by keeping copies of the file at various sites in the system.

Communication between users at different locations is greatly facilitated using a distributed system. There are two important aspects to communication. First, users have unique id's in a distributed system. Their use in communication automatically invokes the security mechanisms of the OS, thereby ensuring privacy and authenticity of communication. Second, use of a distributed system also implies continued availability of communication when users migrate to different sites of a system.

Distributed systems are capable of incremental growth, i.e., the capabilities of a system (e.g. its processing power) can be enhanced at a price proportional to the nature and size of the enhancement. A major advantage of this feature is that enhancements need not be planned in advance. This is in contrast to the classical mainframe architectures where enhancements often took the form of upgradation, that is, replacement of subsystems by more powerful ones—hence enhancement costs tended to be disproportionate to the nature and size of an enhancement.

Distributed systems today cover a wide spectrum of computer hardware, software and topological configurations; resource sharing services range from off-line access to real-time access and topologies vary from locally distributed to geographically distributed.

Multiprocessor Operating Systems

A typical multiprocessor system consists of a set of processors that share a set of physical memory blocks over an interconnection network. Thus, a multiprocessor system is a tightly coupled system where processors share an address space. A multiprocessor operating system controls and manages the hardware and software resources such that users view the entire systems as a powerful uniprocessor system; a user is not aware of the presence of multiple processors and the interconnection network.

The basic issues in the design of a multiprocessor operating system are the same as in a traditional operating system. However, the issues of process synchronisation, task scheduling, memory management, and protection and security become more complex because the main memory is shared by many physical processors.

2.5.2 Application Driven Operating System

A brief discussion of three operating systems in the category of the Architecture Driven OS follows:

Multimedia Operating System

The operating system is the shield of the computer hardware against all software components. It provides a comfortable environment for the execution of programs, and it ensures effective utilisation of the computer hardware. The operating system offers various services related to the essential resources of a computer: CPU, main memory, storage and all input and output devices.

For the processing of audio and video, multimedia application demands that humans perceive these media in a natural, error-free way. These continuous media data

originate at sources like microphones, cameras and files. From these sources, the data are transferred to destinations like loudspeakers, video windows and files located at the same computer or at a remote station. On the way from source to sink, the digital data are processed by at least some type of move, copy or transmit operation. In this data manipulation process there are always many resources which are under the control of the operating system. The integration of discrete and continuous multimedia data demands additional services from many operating system components.

The major aspect in this context is real-time processing of continuous media data. Process management must take into account the timing requirements imposed by the handling of multimedia data. Appropriate scheduling methods should be applied. In contrast to the traditional real-time operating systems, multimedia operating systems also have to consider tasks without hard timing restrictions under the aspect of fairness.

To obey timing requirements, single components are conceived as resources that are reserved prior to execution. This concept of resource reservation has to cover all resources on a data path, i.e. all resources that deal with continuous media. It also may affect parts of the application that process continuous media data. In distributed systems, for example, resource management also comprises network capacity.

The communication and synchronization between single processes must meet the restrictions of real-time requirements and timing relations among different media. The main memory is available as a shared resource to single processes.

In multimedia systems, memory management has to provide access to data with a guaranteed timing delay and efficient data manipulation functions. For instance, physical data copy operations must be avoided due to their negative impact on performance; buffer management operations (such as are known from communication systems) should be used.

Database Operating System

Database systems place special requirements on operating systems. These requirements have their roots in the specific environment that database systems support. A database system must support: the concept of a transaction; operations to store, retrieve, and manipulate a large volume of data efficiently; primitives for concurrency control, and system failure recovery. To store temporary data and data retrieved from secondary storage, it must have a buffer management scheme.

Real-time Operating System

The main characteristics of the real-time systems is the correctness of the computation. This correctness does not apply to error-free computation, but also on the time in which the result is processed and produced. Therefore, the real-time systems must execute its critical workload in time to prevent failures. Examples of applications requiring support of real-time systems are process control, air traffic control, guidance of missile systems, etc.

Real-time systems also place special requirements on operating system, which have their roots in the specific application that the real-time system is supporting. A distinct feature of real-time systems is that jobs have completion deadlines. A job should be completed before its deadline to be of use (in soft real-time system) or to avert a disaster (in hard real-time systems). The major issue in the design of real-time operating systems is the scheduling of jobs in such a way that a maximum number of jobs satisfy their deadlines. Other issues include designing languages and primitives to effectively prepare and execute a job schedule.

Check Your Progress 2

1) What are the basic design issues in the distributed operating systems.

.....

.....

.....

2) What is the basic difference between Network operating systems and Distributed operating systems.

.....

.....

.....

2.6 CHARACTERISTICS OF MODERN OPERATING SYSTEM

Over the years, there has been a gradual evolution of operating system structure and capabilities. However, in recent years a number of new design elements have been introduced into both new operating systems and new releases of existing operating systems that create a major change in the nature of operating systems. These modern operating systems respond to new developments in hardware and new applications. Among the key hardware drivers are multiprocessor machines, greatly increased machine speed, high speed network attachments, and increasing the size and variety of memory storage devices. In the application arena, multimedia applications, Internet and Web access, and client/server computing have influenced operating system design.

The rate of change in the demands on operating systems requires not just modifications and enhancements to existing architectures but new ways of organising the operating system. A wide range of different approaches and design elements has been tried in both expiremental and commercial operating systems, but much of the work fits into the following categories:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing

Most operating systems, until recently, featured a large **monolithic kernel.** Most of what is thought of as operating system functionality is provided in these large kernels, including scheduling, file system, networking, device drivers, memory management and more. Typically, a monolithic kernel is implemented as a single process with all elements sharing the same address space. A **microkernel architecture** assigns only a few essential functions to the kernel, including address spaces, interprocessor communication (IPC), and basic scheduling. Other OS services are provided by processes, sometimes called servers, that run in user mode and are treated like any other application by the microkernel. This approach decouples kernel and server development. Servers may be customised to specific application or environment requirements. The microkernel approach simplifies implementation, provides flexibility, and is well suited to a distributed environment. In essence, a microkernel interacts with local and remote server processes in the same way, facilitating construction of distributed systems.

2.6.1 Microkernel Architecture

A **microkernel architecture** assigns only a few essential functions to the kernel, including address spaces, interprocess communication (IPC), and basic scheduling. Other OS services are provided by processes, sometimes called servers, that run in user mode and are treated like any other application by the microkernel. This approach decouples kernel and server development. Servers may be customised to specific application or environment requirements. The microkernels approach simplifies implementation, provides flexibility, and is well suited to a distributed environment. In essence, a microkernel interacts with local and remote server processes in the same way, facilitating construction of distributed systems.

2.6.2 Multithreading

Multithreading is a technique in which a process executing an application is divided into threads that can run concurrently. We can make the following distinction:

- **Threads:** A dispatchable unit of work. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptable so that the processor can turn to another thread.
- **Process:** A collection of one or more threads and associated system resources (such as memory containing code and data, open files, and devices). This corresponds closely to the concept of a program in execution. By breaking a single application into multiple threads, the programmer has great control over the modularity of the application and the timing of application related events.

Multithreading is useful for applications that perform a number of essentially independent tasks that do not need to be serialised. An example is a database server that listens for and processes numerous client requests. With multiple threads running within the same process, switching back and forth among threads involves less processor overhead than a major process switch between different processes. Threads are also useful for structuring processes that are part of the OS kernel as described in subsequent chapters.

2.6.3 Symmetric Multiprocessing

Until recently, virtually all single-user personal computers and workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced computers with multiple microprocessors. To achieve greater efficiency and reliability, one technique is to employ **symmetric multiprocessing (SMP),** a term that refers to a computer hardware architecture and also to the operating system behaviour that reflects that architecture. A symmetric multiprocessor can be defined as a standalone computer system with the following characteristics:

1) There are multiple processors.
2) These processors share the same main memory and I/O facilities, interconnected by a communications bus or other internal connection scheme.
3) All processors can perform the same functions (hence the term Symmetric).

The operating system of an SMP schedules processes or threads across all of the processors. SMP has a number of potential advantages over uniprocessor architecture, including the following:

- **Performance:** If the work to be done by a computer can be organised so that some portions of the work can be done in parallel, then a system with multiple

processors will yield greater performance than one with a single processor of the same type. With multiprogramming, only one process can execute at a time; meanwhile all other processes are waiting for the processor. With multiprocessing, more than one process can be running simultaneously, each on a different processor.

- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential rather than guaranteed benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

Multithreading and SMP are often discussed together, but the two are independent facilities. Even on a uniprocessor machine, multithreading is useful for structuring applications and kernel processes. An SMP machine is useful for non-threaded processes, because several processes can run in parallel. However, the two facilities complement each other and can be used effectively together.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronisation among processors. A different problem is to provide the appearance of a single system for a cluster of separate computers – a multicomputer system. In this case, we are dealing with a collection of entities (computers), each with its own main memory, secondary memory, and other I/O modules. A distributed operating system provides the illusion of a single main memory space and a single secondary memory space, plus other unified access facilities, such as a distributed file system. Although clusters are becoming increasingly popular, and there are many cluster products on the market, the state of the art for distributed operating systems lags that of uniprocessor and SMP operating systems.

The most recent innovation in operating system design is the use of object oriented technologies. Object oriented design lends discipline to the process of adding modular extensions to a small kernel. At the operating system level, an object-based structure enables programmers to customise an operating system without disrupting system integrity. Object orientation also eases the development of distributed tools and a full-blown distributed operating system.

2.7 SUMMARY

The operating system is an essential component of system software, which consists of procedures for managing computer resources. Initially computers were operated from the front console. System software such as Assemblers, Loaders and Compilers greatly helped in software development but also required substantial setup time. To reduce the setup time an operator was hired and similar jobs were batched together.

Batch systems allowed automatic job sequenting by a resident monitor and improved the overall utilisation of systems greatly. The computer no longer had to wait for human operations – but CPU utilisation was still low because of slow speed of I/O

devices compared to the CPU. A new concept buffering was developed to improve system performance by overlapping the input, output and computation of a single job. Spooling was another new concept in improving the CPU utilisation by overlapping input of one job with the computation and output of other jobs.

Operating systems are now almost always written in a higher-level language (C, PASCAL, etc.). UNIX was the first operating system developed in C language. This feature improves their implementation, maintenance and portability.

The operating system provides a number of services. At the lowest level, there are system calls, which allow a running program to make a request from the operating system directly. At a higher level, there is a command interpreter, which supports a mechanism for a user to issue a request without writing a program.

In this unit, we began with tracing the evolution of the operating system through serial processing, batch processing and multiprogramming. We also presented different operating system models: Layered structured, Kernel based, virtual machine system and client server model.

At the end we presented classification and characteristics of emerging operating systems.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) System calls provide the interface between a running program and the operating system. User programs receives OS services through the set of system calls.

2) Both buffering and spooling are used to improve system performance. Buffering overlaps input, output and processing of a single job whereas spooling allows CPU to overlap the input of the job with computation and output of another jobs. Spooling is better than buffering.

Check Your Progress 2

1) The basic design issues of a distributed operating system are the same as in a conventional operating system, viz., process synchronisation, deadlocks, scheduling, file systems, interprocess communication, memory and buffer management, failure recovery, etc., but what makes its design complexity are the lack of shared memory and common physical global clock and unpredictable communication delays.

2) The basic difference between the two is that the network operating system focuses on the use of remote services and resources existing on a network operating system whereas the distributed operating system focuses on the effective utilisation of resources on distributed environment.

2.9 FURTHER READINGS

1) *Operating System Concepts* by Abraham Silberschatz and James L. Peterson, Addision Wesely.

2) *Operating Systems Design and Implementation* by Andrew S. Tanenbaum, Prentice Hall of India.