UNIT 1 OPERATING SYSTEM : AN OVERVIEW

Structure **Page Nos.**

1.0 INTRODUCTION

Computer software can be divided into two main categories: application software and system software. Application software consists of the programs for performing tasks particular to the machine's utilisation. This software is designed to solve a particular problem for users. Examples of application software include spreadsheets, database systems, desktop publishing systems, program development software, and games.

On the other hand, system software is more transparent and less noticed by the typical computer user. This software provides a general programming environment in which programmers can create specific applications to suit their needs. This environment provides new functions that are not available at the hardware level and performs tasks related to executing the application program. System software acts as an interface between the hardware of the computer and the application software that users need to run on the computer. The most important type of system software is the operating system.

An **Operating System** (OS) is a collection of programs that acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute the programs. Operating Systems are viewed as resource managers. The main resource is the computer hardware in the form of processors, storage, input/output devices, communication devices, and data. Some of the operating system functions are: implementing the user interface, sharing hardware among users, allowing users to share data among themselves, preventing users from interfering with one another, scheduling resources among users, facilitating input/output, recovering from errors, accounting for resource usage, facilitating parallel operations, organising data for secure and rapid access, and handling network communications.

This unit presents the definition of the operating system, goals of the operating system, generations of OS, different types of OS and functions of OS.

1.1 OBJECTIVES

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

- understand the purpose of an operating system;
- describe the general goals of an operating system;
- discuss the evolution of operating systems;
- describe various functions performed by the OS;
- list, discuss and compare various types of OS, and
- describe various structures of operating system.

1.2 WHAT IS AN OPERATING SYSTEM?

In a computer system, we find four main components: the hardware, the operating system, the application software and the users. In a computer system the hardware provides the basic computing resources. The applications programs define the way in which these resources are used to solve the computing problems of the users. The operating system controls and coordinates the use of the hardware among the various systems programs and application programs for the various users.

We can view an operating system as a **resource allocator**. A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, files storage space, input/output devices etc. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system fairly and efficiently.

An operating system is a **control program**. This program controls the execution of user programs to prevent errors and improper use of the computer. Operating systems exist because: they are a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of a computer system is to execute user programs and solve user problems.

While there is no universally agreed upon definition of the concept of an operating system, the following is a reasonable starting point:

A computer's operating system is a group of programs designed to serve two basic purposes:

- To control the allocation and use of the computing system's resources among the various users and tasks, and
- To provide an interface between the computer hardware and the programmer that simplifies and makes feasible the creation, coding, debugging, and maintenance of application programs.

An effective operating system should accomplish the following functions:

- Should act as a command interpreter by providing a user friendly environment.
- Should facilitate communication with other users.
- Facilitate the directory/file creation along with the security option.
- Provide routines that handle the intricate details of I/O programming.
- Provide access to compilers to translate programs from high-level languages to machine language
- Provide a loader program to move the compiled program code to the computer's memory for execution.
- Assure that when there are several active processes in the computer, each will get fair and non-interfering access to the central processing unit for execution.
- Take care of storage and device allocation.
- Provide for long term storage of user information in the form of files.
- Permit system resources to be shared among users when appropriate, and be protected from unauthorised or mischievous intervention as necessary.

Though systems programs such as editors and translators and the various utility programs (such as sort and file transfer program) are not usually considered part of the operating system, the operating system is responsible for providing access to these system resources.

The abstract view of the components of a computer system and the positioning of OS is shown in the *Figure 1.*

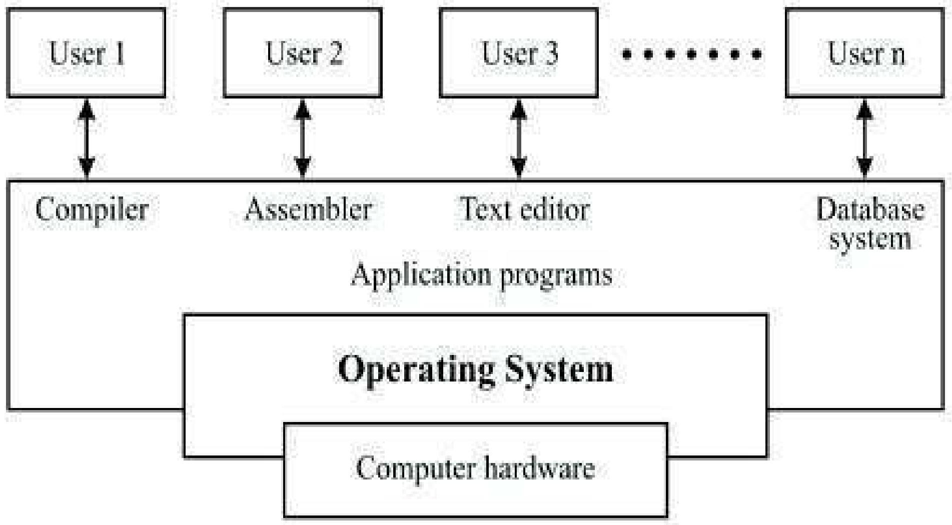


Figure 1: Abstract View of the Components of a Computer System

1.3 GOALS OF AN OPERATING SYSTEM

The primary objective of a computer is to execute an instruction in an efficient manner and to increase the productivity of processing resources attached with the computer system such as hardware resources, software resources and the users. In other words, we can say that maximum CPU utilisation is the main objective, because it is the main device which is to be used for the execution of the programs or instructions. We can brief the goals as:

- The primary goal of an operating system is to make the computer **convenient** to use.
- The secondary goal is to use the hardware in an **efficient** manner.

1.4 GENERATIONS OF OPERATING SYSTEMS

We have studied the definition and the goals of the operating systems. More on the evolution of operating systems can be referred from Unit 2 in Block 1 of MCS-022 (Operating System Concepts and Networking Management).

Operating systems have been evolving over the years. We will briefly look at this development of the operating systems with respect to the evolution of the hardware / architecture of the computer systems in this section. Since operating systems have historically been closely tied with the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. We may not exactly map the operating systems generations to the generations of the computer, but roughly it provides the idea behind them.

We can roughly divide them into five distinct generations that are characterised by hardware component technology, software development, and mode of delivery of computer services.

1.4.1 0th Generation

The term 0th generation is used to refer to the period of development of computing, which predated the commercial production and sale of computer equipment. We consider that the period might be way back when Charles Babbage invented the Analytical Engine. Afterwards the computers by John Atanasoff in 1940; the Mark I, built by Howard Aiken and a group of IBM engineers at Harvard in 1944; the ENIAC, designed and constructed at the University of Pencylvania by Wallace Eckert and John Mauchly and the EDVAC, developed in 1944-46 by John Von Neumann, Arthur Burks, and Herman Goldstine (which was the first to fully implement the idea of the stored program and serial execution of instructions) were designed. The development of EDVAC set the stage for the evolution of commercial computing and operating system software. The hardware component technology of this period was *electronic vacuum tubes*.

The actual operation of these early computers took place **without the benefit of an operating system**. Early programs were written in machine language and each contained code for initiating operation of the computer itself.

The mode of operation was called “open-shop” and this meant that users signed up for computer time and when a user’s time arrived, the entire (in those days quite large) computer system was turned over to the user. The individual user (programmer) was responsible for all machine set up and operation, and subsequent clean-up and preparation for the next user. This system was clearly inefficient and dependent on the varying competencies of the individual programmer as operators.

1.4.2 First Generation (1951-1956)

The first generation marked the beginning of commercial computing, including the introduction of Eckert and Mauchly's UNIVAC I in early 1951, and a bit later, the IBM 701 which was also known as the Defence Calculator. The first generation was characterised again by the vacuum tube as the active component technology.

Operation continued without the benefit of an operating system for a time. The mode was called "closed shop" and was characterised by the appearance of hired operators who would select the job to be run, initial program load the system, run the user's program, and then select another job, and so forth. Programs began to be written in higher level, procedure-oriented languages, and thus the operator's routine expanded. The operator now selected a job, ran the translation program to assemble or compile the source program, and combined the translated object program along with any existing library programs that the program might need for input to the linking program, loaded and ran the composite linked program, and then handled the next job in a similar fashion.

Application programs were run one at a time, and were translated with absolute computer addresses that bound them to be loaded and run from these reassigned storage addresses set by the translator, obtaining their data from specific physical I/O device. There was no provision for moving a program to different location in storage for any reason. Similarly, a program bound to specific devices could not be run at all if any of these devices were busy or broken.

The inefficiencies inherent in the above methods of operation led to the development of the ***mono-programmed operating system***, which eliminated some of the human intervention in running job and provided programmers with a number of desirable functions. The OS consisted of a permanently resident kernel in main storage, and a job scheduler and a number of utility programs kept in secondary storage. User application programs were preceded by control or specification cards (in those day, computer program were submitted on data cards) which informed the OS of what system resources (software resources such as compilers and loaders; and hardware resources such as tape drives and printer) were needed to run a particular application. The systems were designed to be operated as batch processing system.

These systems continued to operate under the control of a human operator who initiated operation by mounting a magnetic tape that contained the operating system executable code onto a "boot device", and then pushing the IPL (Initial Program Load) or "boot" button to initiate the bootstrap loading of the operating system. Once the system was loaded, the operator entered the date and time, and then initiated the operation of the job scheduler program which read and interpreted the control statements, secured the needed resources, executed the first user program, recorded timing and accounting information, and then went back to begin processing of another user program, and so on, as long as there were programs waiting in the input queue to be executed.

The first generation saw the evolution from hands-on operation to closed shop operation to the development of mono-programmed operating systems. At the same time, the development of programming languages was moving away from the basic machine languages; first to assembly language, and later to procedure oriented languages, the most significant being the development of FORTRAN by John W. Backus in 1956. Several problems remained, however, the most obvious was the inefficient use of system resources, which was most evident when the CPU waited while the relatively slower, mechanical I/O devices were reading or writing program data. In addition, system protection was a problem because the operating system kernel was not protected from being overwritten by an erroneous application program.

Moreover, other user programs in the queue were not protected from destruction by executing programs.

1.4.3 Second Generation (1956-1964)

The second generation of computer hardware was most notably characterised by ***transistors*** replacing vacuum tubes as the hardware component technology. In addition, some very important changes in hardware and software architectures occurred during this period. For the most part, computer systems remained card and tape-oriented systems. Significant use of random access devices, that is, disks, did not appear until towards the end of the second generation. Program processing was, for the most part, provided by large centralised computers operated under ***mono-programmed batch processing operating systems***.

The most significant innovations addressed the problem of excessive central processor delay due to waiting for input/output operations. Recall that programs were executed by processing the machine instructions in a strictly sequential order. As a result, the CPU, with its high speed electronic component, was often forced to wait for completion of I/O operations which involved mechanical devices (card readers and tape drives) that were order of magnitude slower. This problem led to the introduction of the data channel, an integral and special-purpose computer with its own instruction set, registers, and control unit designed to process input/output operations separately and asynchronously from the operation of the computer's main CPU near the end of the first generation, and its widespread adoption in the second generation.

The data channel allowed some I/O to be buffered. That is, a program's input data could be read "ahead" from data cards or tape into a special block of memory called a buffer. Then, when the user's program came to an input statement, the data could be transferred from the buffer locations at the faster main memory access speed rather than the slower I/O device speed. Similarly, a program's output could be written another buffer and later moved from the buffer to the printer, tape, or card punch. What made this all work was the data channel's ability to work asynchronously and concurrently with the main processor. Thus, the slower mechanical I/O could be happening concurrently with main program processing. This process was called I/O overlap.

The data channel was controlled by a channel program set up by the operating system I/O control routines and initiated by a special instruction executed by the CPU. Then, the channel independently processed data to or from the buffer. This provided communication from the CPU to the data channel to initiate an I/O operation. It remained for the channel to communicate to the CPU such events as data errors and the completion of a transmission. At first, this communication was handled by polling – the CPU stopped its work periodically and polled the channel to determine if there is any message.

Polling was obviously inefficient (imagine stopping your work periodically to go to the post office to see if an expected letter has arrived) and led to another significant innovation of the second generation – the interrupt. The data channel was able to interrupt the CPU with a message – usually "I/O complete." Infact, the interrupt idea was later extended from I/O to allow signalling of number of exceptional conditions such as arithmetic overflow, division by zero and time-run-out. Of course, interval clocks were added in conjunction with the latter, and thus operating system came to have a way of regaining control from an exceptionally long or indefinitely looping program.

These hardware developments led to enhancements of the operating system. I/O and data channel communication and control became functions of the operating system, both to relieve the application programmer from the difficult details of I/O programming and to protect the integrity of the system to provide improved service to users by segmenting jobs and running shorter jobs first (during "prime time") and relegating longer jobs to lower priority or night time runs. System libraries became more widely available and more comprehensive as new utilities and application software components were available to programmers.

In order to further mitigate the I/O wait problem, system were set up to spool the input batch from slower I/O devices such as the card reader to the much higher speed tape drive and similarly, the output from the higher speed tape to the slower printer. In this scenario, the user submitted a job at a window, a batch of jobs was accumulated and spooled from cards to tape "off line," the tape was moved to the main computer, the jobs were run, and their output was collected on another tape that later was taken to a satellite computer for off line tape-to-printer output. User then picked up their output at the submission windows.

Toward the end of this period, as random access devices became available, tape-oriented operating system began to be replaced by disk-oriented systems. With the more sophisticated disk hardware and the operating system supporting a greater portion of the programmer's work, the computer system that users saw was more and more removed from the actual hardware-users saw a virtual machine.

The second generation was a period of ***intense operating system development***. Also it was the period for sequential batch processing. But the sequential processing of one job at a time remained a significant limitation. Thus, there continued to be low CPU utilisation for I/O bound jobs and low I/O device utilisation for CPU bound jobs. This was a major concern, since computers were still very large (room-size) and expensive machines. Researchers began to experiment with multiprogramming and multiprocessing in their computing services called the time-sharing system. A noteworthy example is the Compatible Time Sharing System (CTSS), developed at MIT during the early 1960s.

1.4.4 Third Generation (1964-1979)

The third generation officially began in April 1964 with IBM's announcement of its System/360 family of computers. Hardware technology began to use integrated circuits (ICs) which yielded significant advantages in both speed and economy.

Operating system development continued with the introduction and widespread adoption of multiprogramming. This marked first by the appearance of more sophisticated I/O buffering in the form of spooling operating systems, such as the HASP (Houston Automatic Spooling) system that accompanied the IBM OS/360 system. These systems worked by introducing two new systems programs, a system reader to move input jobs from cards to disk, and a system writer to move job output from disk to printer, tape, or cards. Operation of spooling system was, as before, transparent to the computer user who perceived input as coming directly from the cards and output going directly to the printer.

The idea of taking fuller advantage of the computer's data channel I/O capabilities continued to develop. That is, designers recognised that I/O needed only to be initiated by a CPU instruction – the actual I/O data transmission could take place under control of separate and asynchronously operating channel program. Thus, by switching control of the CPU between the currently executing user program, the system reader program, and the system writer program, it was possible to keep the slower mechanical I/O device running and minimizes the amount of time the CPU spent waiting for I/O completion. The net result was an increase in system throughput and resource utilisation, to the benefit of both user and providers of computer services.

This concurrent operation of three programs (more properly, apparent concurrent operation, since systems had only one CPU, and could, therefore execute just one instruction at a time) required that additional features and complexity be added to the operating system. First, the fact that the input queue was now on disk, a direct access device, freed the system scheduler from the first-come-first-served policy so that it could select the "best" next job to enter the system (looking for either the shortest job or the highest priority job in the queue). Second, since the CPU was to be shared by the user program, the system reader, and the system writer, some processor allocation rule or policy was needed. Since the goal of spooling was to increase resource utilisation by enabling the slower I/O devices to run asynchronously with user program processing, and since I/O processing required the CPU only for short periods to initiate data channel instructions, the CPU was dispatched to the reader, the writer, and the program in that order. Moreover, if the writer or the user program was executing when something became available to read, the reader program would preempt the currently executing program to regain control of the CPU for its initiation instruction, and the writer program would preempt the user program for the same purpose. This rule, called the static priority rule with preemption, was implemented in the operating system as a system dispatcher program.

The spooling operating system in fact had multiprogramming since more than one program was resident in main storage at the same time. Later this basic idea of multiprogramming was extended to include more than one active user program in memory at time. To accommodate this extension, both the scheduler and the dispatcher were enhanced. The scheduler became able to manage the diverse resource needs of the several concurrently active used programs, and the dispatcher included policies for allocating processor resources among the competing user programs. In addition, memory management became more sophisticated in order to assure that the program code for each job or at least that part of the code being executed, was resident in main storage.

The advent of large-scale multiprogramming was made possible by several important hardware innovations such as:

- The widespread availability of large capacity, high-speed disk units to accommodate the spooled input streams and the memory overflow together with the maintenance of several concurrently active program in execution.

- Relocation hardware which facilitated the moving of blocks of code within memory without any undue overhead penalty.

- The availability of storage protection hardware to ensure that user jobs are protected from one another and that the operating system itself is protected from user programs.

- Some of these hardware innovations involved extensions to the interrupt system in order to handle a variety of external conditions such as program malfunctions, storage protection violations, and machine checks in addition to I/O interrupts. In addition, the interrupt system became the technique for the user program to request services from the operating system kernel.

- The advent of privileged instructions allowed the operating system to maintain coordination and control over the multiple activities now going on with in the system.

Successful implementation of multiprogramming opened the way for the development of a new way of delivering computing services-time-sharing. In this environment, several terminals, sometimes up to 200 of them, were attached (hard wired or via telephone lines) to a central computer. Users at their terminals, “logged in” to the central system, and worked interactively with the system. The system’s apparent concurrency was enabled by the multiprogramming operating system. Users shared not only the system hardware but also its software resources and file system disk space.

The third generation was an exciting time, indeed, for the development of both computer hardware and the accompanying operating system. During this period, the topic of operating systems became, in reality, a major element of the discipline of computing.

1.4.5 Fourth Generation (1979 – Present)

The fourth generation is characterised by the appearance of the personal computer and the workstation. Miniaturisation of electronic circuits and components continued and large scale integration (LSI), the component technology of the third generation, was replaced by very large scale integration (VLSI), which characterizes the fourth generation. VLSI with its capacity for containing thousands of transistors on a small chip, made possible the development of desktop computers with capabilities exceeding those that filled entire rooms and floors of building just twenty years earlier.

The operating systems that control these desktop machines have brought us back in a full circle, to the open shop type of environment where each user occupies an entire computer for the duration of a job’s execution. This works better now, not only because the progress made over the years has made the virtual computer resulting from the operating system/hardware combination so much easier to use, or, in the words of the popular press “user-friendly.”

However, improvements in hardware miniaturisation and technology have evolved so fast that we now have inexpensive workstation – class computers capable of supporting multiprogramming and time-sharing. Hence the operating systems that supports today’s personal computers and workstations look much like those which were available for the minicomputers of the third generation. Examples are Microsoft’s DOS for IBM-compatible personal computers and UNIX for workstation.

However, many of these desktop computers are now connected as networked or distributed systems. Computers in a networked system each have their operating systems augmented with communication capabilities that enable users to remotely log into any system on the network and transfer information among machines that are connected to the network. The machines that make up distributed system operate as a virtual single processor system from the user’s point of view; a central operating system controls and makes transparent the location in the system of the particular processor or processors and file systems that are handling any given program.

☞ **Check Your Progress 1**

1) What are the main responsibilities of an Operating System?

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

1.5 TYPES OF OPERATING SYSTEMS

In this section we will discuss about the different types of operating systems. Also study the types of operating systems from Unit 2 in Block 1 of MCS-022 (Operating System Concepts and Networking Management).

Modern computer operating systems may be classified into three groups, which are distinguished by the nature of interaction that takes place between the computer user and his or her program during its processing. The three groups are called batch, time-sharing and real-time operating systems.

1.5.1 Batch Processing Operating System

In a batch processing operating system environment users submit jobs to a central place where these jobs are collected into a batch, and subsequently placed on an input queue at the computer where they will be run. In this case, the user has no interaction with the job during its processing, and the computer's response time is the turnaround time−the time from submission of the job until execution is complete, and the results are ready for return to the person who submitted the job.

1.5.2 Time Sharing

Another mode for delivering computing services is provided by time sharing operating systems. In this environment a computer provides computing services to several or many users concurrently on-line. Here, the various users are sharing the central processor, the memory, and other resources of the computer system in a manner facilitated, controlled, and monitored by the operating system. The user, in this environment, has nearly full interaction with the program during its execution, and the computer's response time may be expected to be no more than a few second.

1.5.3 Real Time Operating System (RTOS)

The third class is the real time operating systems, which are designed to service those applications where response time is of the essence in order to prevent error, misrepresentation or even disaster. Examples of real time operating systems are those which handle airlines reservations, machine tool control, and monitoring of a nuclear power station. The systems, in this case, are designed to be interrupted by external signals that require the immediate attention of the computer system.

These real time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy.

A number of ***other definitions*** are important to gain an understanding of operating systems:

1.5.4 Multiprogramming Operating System

A multiprogramming operating system is a system that allows more than one active user program (or part of user program) to be stored in main memory simultaneously. Thus, it is evident that a time-sharing system is a multiprogramming system, but note that a multiprogramming system is not necessarily a time-sharing system. A batch or real time operating system could, and indeed usually does, have more than one active user program simultaneously in main storage. Another important, and all too similar, term is "multiprocessing".

1.5.5 Multiprocessing System

A multiprocessing system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications. More on multiprocessor system can be studied in Unit-1 of Block-3 of this course.

1.5.6 Networking Operating System

A networked computing system is a collection of physical interconnected computers. The operating system of each of the interconnected computers must contain, in addition to its own stand-alone functionality, provisions for handing communication and transfer of program and data among the other computers with which it is connected.

Network operating systems are not fundamentally different from single processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote files access, but these additions do not change the essential structure of the operating systems.

1.5.7 Distributed Operating System

A distributed computing system consists of a number of computers that are connected and managed so that they automatically share the job processing load among the constituent computers, or separate the job load as appropriate particularly configured processors. Such a system requires an operating system which, in addition to the typical stand-alone functionality, provides coordination of the operations and information flow among the component computers. The networked and distributed computing environments and their respective operating systems are designed with more complex functional capabilities. 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 another. Each machine runs its own local operating system and has its own user (or users).

A distributed operating system, in contrast, is one that appears to its users as a traditional uni-processor system, even though it is actually composed of multiple processors. In a true distributed system, users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uni-processor operating system, because distributed and centralised systems differ in critical ways. Distributed systems, for example, often allow program to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimise the amount of parallelism achieved. More on distributed systems can be studied in Unit-2 of Block-3 of this course.

1.5.8 Operating Systems for Embedded Devices

As embedded systems (PDAs, cellphones, point-of-sale devices, VCR's, industrial robot control, or even your toaster) become more complex hardware-wise with every generation, and more features are put into them day-by-day, applications they run require more and more to run on actual operating system code in order to keep the development time reasonable. Some of the popular OS are:

- Nexus's Conix - an embedded operating system for ARM processors.
- Sun's Java OS - a standalone virtual machine not running on top of any other OS; mainly targeted at embedded systems.
- Palm Computing's Palm OS - Currently the leader OS for PDAs, has many applications and supporting companies.
- Microsoft's Windows CE and Windows NT Embedded OS.

Let us study various examples of the popular operating systems in the next section.

- ***Please go through the various Views and Structure of operating system from Unit 2 in Block 1 of MCS-022 (Operating System Concepts and Networking Management).***

1.6 DESIRABLE QUALITIES OF OS

The desirable qualities of an operating system are in terms of: Usability, Facilities, Cost, and Adaptability.

- Usability:
  - Robustness
  - *Accept all valid inputs and can handle them.*
  - Consistency
  - Proportionality
  - Convenience
  - Powerful with high level facilities.
- Facilities:
  - Sufficient for intended use
  - Complete
  - Appropriate.
- Costs:
  - Want low cost and efficient services.
  - Good algorithms.
    Make use of space/time tradeoffs, special hardware.
  - Low overhead.
    Cost of doing nothing should be low. E.g., idle time at a terminal.
  - Low maintenance cost.
    System should not require constant attention.
- Adaptability:
  - Tailored to the environment.
    Support necessary activities. Do not impose unnecessary restrictions. What are the things people do most -- make them easy.
  - Changeable over time.
    Adapt as needs and resources change. e.g., expanding memory and new devices, or new user population.
  - Extendible-Extensible
    Adding new facilities and features - which look like the old ones.

1.7 OPERATING SYSTEMS : SOME EXAMPLES

In the earlier section we had seen the types of operating systems. In this section we will study some popular operating systems.

1.7.1 DOS

DOS (Disk Operating System) was the first widely-installed operating system for personal computers. It is a master control program that is automatically run when you start your personal computer (PC). DOS stays in the computer all the time letting you run a program and manage files. It is a single-user operating system from Microsoft for the PC. It was the first OS for the PC and is the underlying control program for Windows 3.1, 95, 98 and ME. Windows NT, 2000 and XP emulate DOS in order to support existing DOS applications.

1.7.2 UNIX

UNIX operating systems are used in widely-sold workstation products from Sun Microsystems, Silicon Graphics, IBM, and a number of other companies. The UNIX environment and the client/server program model were important elements in the development of the Internet and the reshaping of computing as centered in networks rather than in individual computers. Linux, a UNIX derivative available in both "free software" and commercial versions, is increasing in popularity as an alternative to proprietary operating systems.

UNIX is written in **C**. Both UNIX and C were developed by AT&T and freely distributed to government and academic institutions, causing it to be ported to a wider variety of machine families than any other operating system. As a result, UNIX became synonymous with "open systems".

UNIX is made up of the kernel, file system and shell (command line interface). The major shells are the Bourne shell (original), C shell and Korn shell. The UNIX vocabulary is exhaustive with more than 600 commands that manipulate data and text in every way conceivable. Many commands are cryptic, but just as Windows hid the DOS prompt, the Motif GUI presents a friendlier image to UNIX users. Even with its many versions, UNIX is widely used in mission critical applications for client/server and transaction processing systems. The UNIX versions that are widely used are Sun's Solaris, Digital's UNIX, HP's HP-UX, IBM's AIX and SCO's UnixWare. A large number of IBM mainframes also run UNIX applications, because the UNIX interfaces were added to MVS and OS/390, which have obtained UNIX branding. Linux, another variant of UNIX, is also gaining enormous popularity. More details can be studied in Unit-3 of Block-3 of this course.

1.7.3 WINDOWS

Windows is a personal computer operating system from Microsoft that, together with some commonly used business applications such as Microsoft Word and Excel, has become a *de facto* "standard" for individual users in most corporations as well as in most homes. Windows contains built-in networking, which allows users to share files and applications with each other if their PC's are connected to a network. In large enterprises, Windows clients are often connected to a network of UNIX and NetWare servers. The server versions of Windows NT and 2000 are gaining market share, providing a Windows-only solution for both the client and server. Windows is supported by Microsoft, the largest software company in the world, as well as the Windows industry at large, which includes tens of thousands of software developers.

This networking support is the reason why Windows became successful in the first place. However, Windows 95, 98, ME, NT, 2000 and XP are complicated operating environments. Certain combinations of hardware and software running together can cause problems, and troubleshooting can be daunting. Each new version of Windows has interface changes that constantly confuse users and keep support people busy, and Installing Windows applications is problematic too. Microsoft has worked hard to make Windows 2000 and Windows XP more resilient to installation of problems and crashes in general. More details on Windows 2000 can be studied in Unit-4 of Block -3 of this course.

1.7.4 MACINTOSH

The Macintosh (often called "the Mac"), introduced in 1984 by Apple Computer, was the first widely-sold personal computer with a graphical user interface (GUI). The Mac was designed to provide users with a natural, intuitively understandable, and, in general, "user-friendly" computer interface. This includes the mouse, the use of icons or small visual images to represent objects or actions, the point-and-click and click-and-drag actions, and a number of window operation ideas. Microsoft was successful in adapting user interface concepts first made popular by the Mac in its first Windows operating system. The primary disadvantage of the Mac is that there are fewer Mac applications on the market than for Windows. However, all the fundamental applications are available, and the Macintosh is a perfectly useful machine for almost everybody. Data compatibility between Windows and Mac is an issue, although it is often overblown and readily solved.

The Macintosh has its own operating system, Mac OS which, in its latest version is called Mac OS X. Originally built on Motorola's 68000 series microprocessors, Mac versions today are powered by the PowerPC microprocessor, which was developed jointly by Apple, Motorola, and IBM. While Mac users represent only about 5% of the total numbers of personal computer users, Macs are highly popular and almost a cultural necessity among graphic designers and online visual artists and the companies they work for.

In this section we will discuss some services of the operating system used by its users. Users of operating system can be divided into two broad classes: command language users and system call users. Command language users are those who can interact with operating systems using the commands. On the other hand system call users invoke services of the operating system by means of run time system calls during the execution of programs.

☞ **Check Your Progress 2**

1) Discuss different views of the Operating Systems.

………………………………………………………………………..…

………………………………………………………………………..

2) Summarise the characteristics of the following operating systems:

- Batch Processing
- Time Sharing
- Real time

………………………………………………………………………..…

………………………………………………………………………..…

1.8 FUNCTIONS OF OS

The main functions of an operating system are as follows:

- Process Management
- Memory Management
- Secondary Storage Management
- I/O Management
- File Management
- Protection
- Networking Management
- Command Interpretation.

1.8.1 Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

The operating system is responsible for the following activities in connection with processes management:

- The creation and deletion of both user and system processes
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling.

More details can be studied in the Unit-2 of this course.

1.8.2 Memory Management

Memory is the most expensive part in the computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.

There are various algorithms that depend on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management.

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

Memory management techniques will be discussed in great detail in Unit-1 of Block-2 of this course.

1.8.3 Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modem computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence the proper management of disk storage is of central importance to a computer system.

There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern.

The operating system is responsible for the following activities in connection with disk management:

- Free space management
- Storage allocation
- Disk scheduling.

More details can be studied in Unit-3 of Block-2 of this course.

1.8.4 I/O Management

One of the purposes of an operating system is to hide the peculiarities or specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The operating system is responsible for the following activities in connection to I/O management:

- A buffer caching system
- To activate a general device driver code
- To run the driver software for specific hardware devices as and when required.

More details can be studied in the Unit-3 of Block-2 of this course.

1.8.5 File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms: magnetic tape, disk, and drum are the most common forms. Each of these devices has it own characteristics and physical organisation.

For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general a files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.

The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organised into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

The operating system is responsible for the following activities in connection to the file management:

- The creation and deletion of files.
- The creation and deletion of directory.
- The support of primitives for manipulating files and directories.
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.
- Protection and security of the files.

1.8.6 Protection

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorisation from the operating system.

For example, memory addressing hardware ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorised or incompetent user. More on protection and security can be studied in Unit-4 of Block-3.

1.8.7 Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies and the problems of connection and security.

A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

1.8.8 Command Interpretation

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system.

Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed. This program is variously called (1) the control card interpreter, (2) the command line interpreter, (3) the shell (in Unix), and so on. Its function is quite simple: get the next command statement, and execute it.

The command statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

The *Figure 2* depicts the role of the operating system in coordinating all the functions.

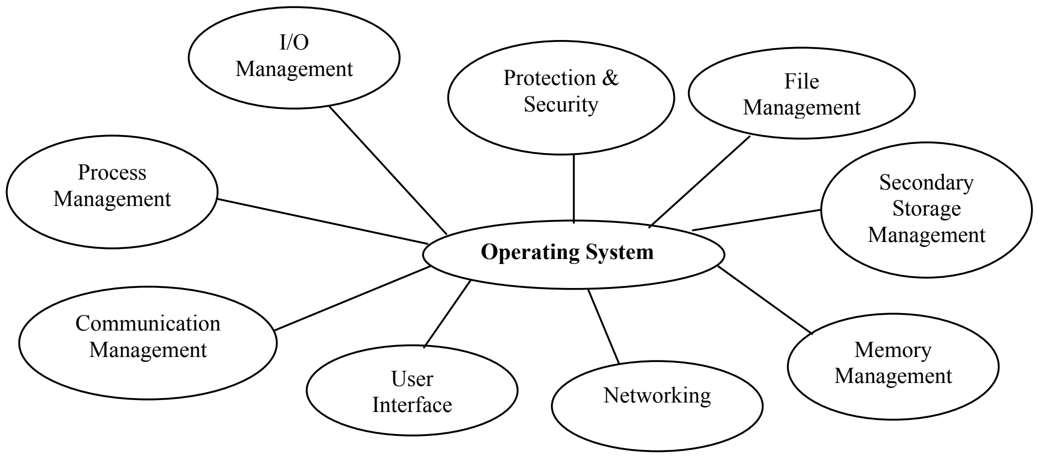


Figure 2: Functions Coordinated by the Operating System

☞ **Check Your Progress 3**

1) Mention the advantages and limitations of Multiuser Operating Systems

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

2) What is a Multitasking system and mention its advantages.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

3) Illustrate a simple operating system for a security control system.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

1.9 SUMMARY

This Unit presented the principle operation of an operating system. In this unit we had briefly described about the history, the generations and the types of operating systems.

An operating system is a program that acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute programs. The primary goal of an operating system is to make the computer **convenient** to use. And the secondary goal is to use the hardware in an **efficient** manner.

Operating systems may be classified by both how many tasks they can perform "simultaneously" and by how many users can be using the system "simultaneously". That is: *single-user* or *multi-user* and *single-task* or *multi-tasking*. A multi-user system must clearly be multi-tasking. In the next unit we will discuss the concept of processes and their management by the OS.

1.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The most important type of system software is the operating system. An operating system has three main responsibilities:

   - Perform basic tasks, such as recognising input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.
   - Ensure that different programs and users running at the same time do not interfere with each other.
   - Provide a software platform on top of which other programs (i.e., application software) can run.

The first two responsibilities address the need for managing the computer hardware and the application programs that use the hardware. The third responsibility focuses on providing an interface between application software and hardware so that application software can be efficiently developed. Since the operating system is already responsible for managing the hardware, it should provide a programming interface for application developers.

Check Your Progress 2

1) The following are the different views of the operating system.

As a scheduler/allocator:

- The operating system has resources for which it is in charge.
- Responsible for handing them out (and later recovering them).
- Resources include CPU, memory, I/O devices, and disk space.

As a virtual machine:

- Operating system provides a "new" machine.
  - This machine could be the same as the underlying machine. Allows many users to believe they have an entire piece of hardware to themselves.
  - This could implement a different, perhaps more powerful, machine. Or just a different machine entirely. It may be useful to be able to completely simulate another machine with your current hardware.

As a multiplexer:

- Allows sharing of resources, and provides protection from interference.
- Provides for a level of cooperation between users.
- Economic reasons: we have to take turns.

2) **Batch Processing**: This strategy involves reading a series of jobs (called a batch) into the machine and then executing the programs for each job in the batch. This approach does not allow users to interact with programs while they operate.

Time Sharing: This strategy supports multiple interactive users. Rather than preparing a job for execution ahead of time, users establish an interactive session with the computer and then provide commands, programs and data as they are needed during the session.

Real Time: This strategy supports real-time and process control systems. These are the types of systems which control satellites, robots, and air-traffic control. The dedicated strategy must guarantee certain response times for particular computing tasks or the application is useless.

Check Your Progress 3

1) The advantage of having a multi-user operating system is that normally the hardware is very expensive, and it lets a number of users share this expensive resource. This means the cost is divided amongst the users. It also makes better use of the resources. Since the resources are shared, they are more likely to be in use than sitting idle being unproductive.

One limitation with multi-user computer systems is that as more users access it, the performance becomes slower and slower. Another limitation is the cost of hardware, as a multi-user operating system requires a lot of disk space and memory. In addition, the actual software for multi-user operating systems tend to cost more than single-user operating systems.

2) A multi-tasking operating system provides the ability to run more than one program at once. For example, a user could be running a word processing package, printing a document, copying files to the floppy disk and backing up selected files to a tape unit. Each of these tasks the user is doing appears to be running at the same time.

   A multi-tasking operating system has the advantage of letting the user run more than one task at once, so this leads to increased productivity. The disadvantage is that more programs that are run by the user, the more memory is required.

3) An operating system for a security control system (such as a home alarm system) would consist of a number of programs. One of these programs would gain control of the computer system when it is powered on, and initialise the system. The first task of this initialise program would be to reset (and probably test) the hardware sensors and alarms. Once the hardware initialisation was complete, the operating system would enter a continual monitoring routine of all the input sensors. If the state of any input sensor is changed, it would branch to an alarm generation routine.

1.11 FURTHER READINGS

1) Madnick and Donovan, *Operating systems – Concepts and design,* McGrawHill Intl. Education.

2) Milan Milenkovic, *Operating Systems, Concepts and Design*, TMGH, 2000.

3) D.M. Dhamdhere, *Operating Systems, A concept-based approach*, TMGH, 2002.

4) Abraham Silberschatz and James L, *Operating System Concepts*. Peterson, Addition Wesley Publishing Company.

5) Harvay M. Deital, *Introduction to Operating Systems,* Addition Wesley Publishing Company.

6) Andrew S. Tanenbaum, *Operating System Design and Implementation,* PHI.

UNIT 2 PROCESSES

Structure **Page Nos.**

2.0 INTRODUCTION

In the earlier unit we have studied the overview and the functions of an operating system. In this unit we will have detailed discussion on the processes and their management by the operating system. The other resource management features of operating systems will be discussed in the subsequent units.

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when it is created, some initialisation data (input) may be passed along. For example, a process whose function is to display the status of a file, say F1, on the screen, will get the name of the file F1 as an input and execute the appropriate program to obtain the desired information.

We emphasize that a program by itself is not a process; a program is a passive entity, while a process is an active entity. It is known that two processes may be associated with the same program; they are nevertheless considered two separate execution sequences.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code. All of those processes can potentially execute concurrently.

The operating system is responsible for the following activities in connection with processes managed.

- The creation and deletion of both user and system processes;
- The suspension is resumption of processes;
- The provision of mechanisms for process synchronization, and
- The provision of mechanisms for deadlock handling.

We will learn the operating system view of the processes, types of schedulers, different types of scheduling algorithms, in the subsequent sections of this unit.

2.1 OBJECTIVES

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

- understand the concepts of process, various states in the process and their scheduling;
- define a process control block;
- classify different types of schedulers;
- understand various types of scheduling algorithms, and
- compare the performance evaluation of the scheduling algorithms.

2.2 THE CONCEPT OF PROCESS

The term “process” was first used by the operating system designers of the MULTICS system way back in 1960s. There are different definitions to explain the concept of process. Some of these are, a process is:

- An instance of a program in execution
- An asynchronous activity
- The “animated spirit” of a procedure
- The “locus of control” of a procedure in execution
- The “dispatchable” unit
- Unit of work individually schedulable by an operating system.

Formally, we can define a **process** is an executing program, including the current values of the program counter, registers, and variables. The subtle difference between a process and a program is that the program is a group of instructions whereas the process is the activity.

In multiprogramming systems, processes are performed in a pseudo-parallelism as if each process has its own processor. In fact, there is only one processor but it switches back and forth from process to process. Henceforth, by saying *execution* of a process, we mean the processor’s operations on the process like changing its variables, etc. and *I/O work* means the interaction of the process with the I/O operations like reading something or writing to somewhere. They may also be named as “*processor (CPU) burst”* and *“I/O burst”* respectively. According to these definitions, we classify programs as:

- **Processor- bound program**: A program having long processor bursts (execution instants)

- **I/O- bound program**: A program having short processor bursts.

The operating system works as the computer system software that assists hardware in performing process management functions. Operating system keeps track of all the active processes and allocates system resources to them according to policies devised to meet design performance objectives. To meet process requirements OS must maintain many data structures efficiently. The process abstraction is fundamental means for the OS to manage concurrent program execution. OS must interleave the execution of a number of processes to maximize processor use while providing reasonable response time. It must allocate resources to processes in conformance with a specific policy. In general, a process will need certain resources such as CPU time, memory, files, I/O devices etc. to accomplish its tasks. These resources are allocated to the process when it is created. A single processor may be shared among several processes with some scheduling algorithm being used to determine when to stop work on one process and provide service to a different one which we will discuss later in this unit.

Operating systems must provide some way to create all the processes needed. In simple systems, it may be possible to have all the processes that will ever be needed be present when the system comes up. In almost all systems however, some way is needed to create and destroy processes as needed during operations. In UNIX, for instant, processes are created by the *fork* system call, which makes an identical copy of the calling process. In other systems, system calls exist to create a process, load its memory, and start it running. In general, processes need a way to create other processes. Each process has one parent process, but zero, one, two, or more children processes.

For an OS, the process management functions include:

- Process creation
- Termination of the process
- Controlling the progress of the process
- Process Scheduling
- Dispatching
- Interrupt handling / Exceptional handling
- Switching between the processes
- Process synchronization
- Interprocess communication support
- Management of Process Control Blocks.

2.2.1 Implicit and Explicit Tasking

A separate process at run time can be either–

- Implicit tasking and
- Explicit tasking.

Implicit tasking means that processes are defined by the system. It is commonly encountered in general purpose multiprogramming systems. In this approach, each program submitted for execution is treated by the operating system as an independent process. Processes created in this manner are usually transient in the sense that they are destroyed and disposed of by the system after each run.

Explicit tasking means that programs explicitly define each process and some of its attributes. To improve the performance, a single logical application is divided into various related processes. Explicit tasking is used in situations where high performance in desired system programs such as parts of the operating system and real time applications are common examples of programs defined processes. After dividing process into several independent processes, a system programs defines the confines of each individual process. A parent process is then commonly added to create the environment for and to control execution of individual processes.

Common reasons for and uses of explicit tasking include:

- **Speedup:** Explicit tasking can result in faster execution of applications.
- **Driving I/O devices that have latency:** While one task is waiting for I/O to complete, another portion of the application can make progress towards completion if it contains other tasks that can do useful work in the interim.
- **User convenience:** By creating tasks to handle individual actions, a graphical interface can allow users to launch several operations concurrently by clicking on action icons before completion of previous commands.
- **Multiprocessing and multicomputing:** A program coded as a collection of tasks can be relatively easily posted to a multiprocessor system, where individual tasks may be executed on different processors in parallel.

Distributed computing network server can handle multiple concurrent client sessions by dedicating an individual task to each active client session.

2.2.2 Processes Relationship

In the concurrent environment basically processes have two relationships, ***competition*** *and* ***cooperation***. In the concurrent environment, processes compete with each other for allocation of system resources to execute their instructions. In addition, a collection of related processes that collectively represent a single logical application cooperate with each other. There should be a proper operating system to support these relations. In the competition, there should be proper resource allocation and protection in address generation.

We distinguish between *independent process* and *cooperating process*. A process is independent if it cannot affect or be affected by other processes executing in the system.

Independent process: These type of processes have following features:

- Their state is not shared in any way by any other process.
- Their execution is deterministic, i.e., the results of execution depend only on the input values.
- Their execution is reproducible, i.e., the results of execution will always be the same for the same input.
- Their execution can be stopped and restarted without any negative effect.

Cooperating process: In contrast to independent processes, cooperating processes can affect or be affected by other processes executing the system. They are characterised by:

- Their states are shared by other processes.
- Their execution is not deterministic, i.e., the results of execution depend on relative execution sequence and cannot be predicted in advance.

Their execution is irreproducible, i.e., the results of execution are not always the same for the same input.

2.2.3 Process States

As defined, a process is an independent entity with its own input values, output values, and internal state. A process often needs to interact with other processes. One process may generate some outputs that other process uses as input. For example, in the shell command

cat file1 file2 file3 | grep tree

The first process, running *cat*, concatenates and outputs three files. Depending on the relative speed of the two processes, it may happen that *grep* is ready to run, but there is no input waiting for it. It must then block until some input is available. It is also possible for a process that is ready and able to run to be blocked because the operating system is decided to allocate the CPU to other process for a while.

A process state may be in one of the following:

- **New :** The process is being created.
- **Ready :** The process is waiting to be assigned to a processor.
- **Running :** Instructions are being executed.
- **Waiting/Suspended/Blocked :** The process is waiting for some event to occur.
- **Halted/Terminated :** The process has finished execution.

The transition of the process states are shown in *Figure 1.*and their corresponding transition are described below:

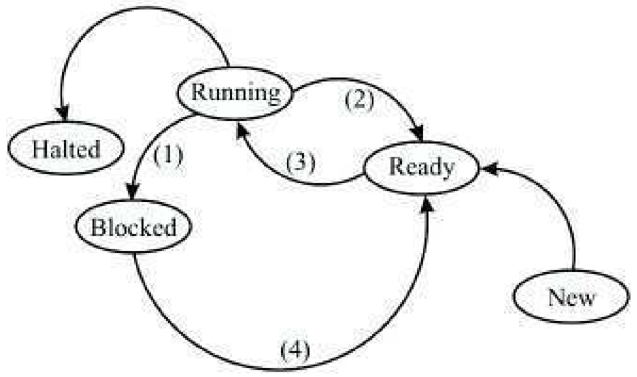


Figure 1: Typical process states

As shown in *Figure 1*, four transitions are possible among the states.
Transition 1 appears when a process discovers that it cannot continue. In order to get into blocked state, some systems must execute a system call *block*. In other systems, when a process reads from a pipe or special file and there is no input available, the process is automatically blocked.

Transition 2 and 3 are caused by the process scheduler, a part of the operating system. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all other processes have had their share and it is time for the first process to run again.

Transition 4 appears when the external event for which a process was waiting was happened. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available.

Using the process model, it becomes easier to think about what is going on inside the system. There are many processes like user processes, disk processes, terminal processes, and so on, which may be blocked when they are waiting for some thing to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is ready to run again.

The process model, an integral part of an operating system, can be summarized as follows. The lowest level of the operating system is the scheduler with a number of processes on top of it. All the process handling, such as starting and stopping processes are done by the scheduler. More on the schedulers can be studied is the subsequent sections.

2.2.4 Implementation of Processes

To implement the process model, the operating system maintains a table, an array of structures, called the *process table or process control block (PCB) or Switch frame.* Each entry identifies a process with information such as process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information. In other words, it must contain everything about the process that must be saved when the process is switched from the running state to the ready state so that it can be restarted later as if it had never been stopped. The following is the information stored in a PCB.

- Process state, which may be new, ready, running, waiting or halted;
- Process number, each process is identified by its process number, called process ID;
- Program counter, which indicates the address of the next instruction to be executed for this process;
- CPU registers, which vary in number and type, depending on the concrete microprocessor architecture;
- Memory management information, which include base and bounds registers or page table;
- I/O status information, composed I/O requests, I/O devices allocated to this process, a list of open files and so on;
- Processor scheduling information, which includes process priority, pointers to scheduling queues and any other scheduling parameters;
- List of open files.

A process structure block is shown in *Figure 2*.

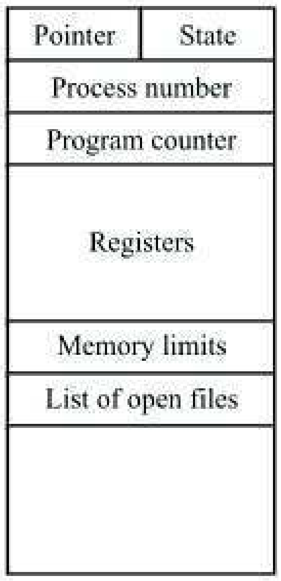


Figure 2: Process Control Block Structure

Context Switch

A *context switch* (also sometimes referred to as a *process switch* or a *task switch*) is the switching of the CPU (central processing unit) from one process or to another. A *context* is the contents of a CPU's registers and *program counter* at any point in time.

A context switch is sometimes described as the kernel suspending *execution of one process* on the CPU and resuming *execution of some other process* that had previously been suspended.

Context Switch: Steps

In a context switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue normally.

The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called a process control block (PCB). Now, in order to switch processes, the PCB for the first process must be created and saved. The PCBs are sometimes stored upon a per-process stack in the kernel memory, or there may be some specific operating system defined data structure for this information.

Let us understand with the help of an example. Suppose if two processes A and B are in ready queue. If CPU is executing Process A and Process B is in wait state. If an interrupt occurs for Process A, the operating system suspends the execution of the first process, and stores the current information of Process A in its PCB and context to the second process namely Process B. In doing so, the program counter from the PCB of Process B is loaded, and thus execution can continue with the new process. The switching between two processes, Process A and Process B is illustrated in the *Figure 3* given below:

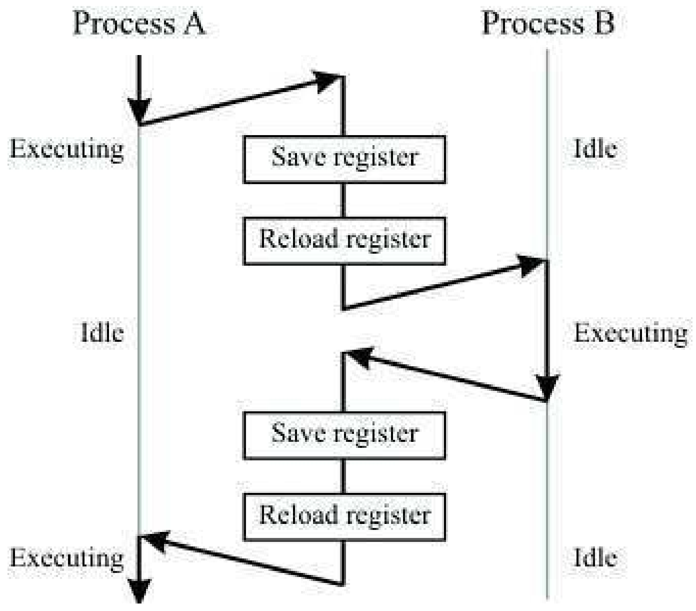


Figure 3: Process Switching between two processes

2.2.5 Process Hierarchy

Modern general purpose operating systems permit a user to create and destroy processes. A process may create several new processes during its time of execution. The creating process is called parent process, while the new processes are called child processes. There are different possibilities concerning creating new processes:

- **Execution**: The parent process continues to execute concurrently with its children processes or it waits until all of its children processes have terminated (sequential).
- **Sharing**: Either the parent and children processes share all resources (likes memory or files) or the children processes share only a subset of their parent's resources or the parent and children processes share no resources in common.

A parent process can terminate the execution of one of its children for one of these reasons:

- The child process has exceeded its usage of the resources it has been allocated. In order to do this, a mechanism must be available to allow the parent process to inspect the state of its children processes.

- The task assigned to the child process is no longer required.

Let us discuss this concept with an example. In UNIX this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process. After a fork both parent and child keep running (indeed they have the *same* program text) and each can fork off other processes. This results in a process tree. The root of the tree is a special process created by the OS during startup. A process can *choose* to wait for children to terminate. For example, if C issued a wait( ) system call it would block until G finished. This is shown in the *Figure 4*.

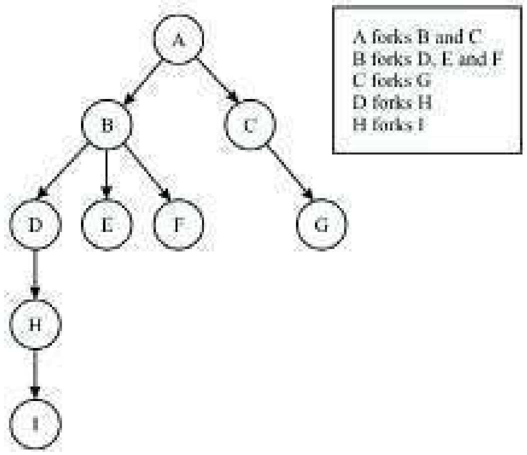


Figure 4: Process hierarchy

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is *automatically* blocked and waits until the second is finished.

2.2.6 Threads

Threads, sometimes called lightweight processes (LWPs) are independently scheduled parts of a single program. We say that a task is *multithreaded* if it is composed of several independent sub-processes which do work on common data, and if each of those pieces could (at least in principle) run in parallel.

If we write a program which uses threads – there is only one program, one executable file, one task in the normal sense. Threads simply enable us to split up that program into logically separate pieces, and have the pieces run independently of one another, until they need to communicate. In a sense, threads are a further level of *object orientation* for multitasking systems. They allow certain *functions* to be executed in parallel with others.

On a truly parallel computer (several CPUs) we might imagine parts of a program (different subroutines) running on quite different processors, until they need to communicate. When one part of the program needs to send data to the other part, the two independent pieces must be synchronized, or be made to wait for one another. But what is the point of this? We can always run independent procedures in a program as separate *programs*, using the process mechanisms we have already introduced. They could communicate using normal interprocesses communication. Why introduce another new concept? Why do we need threads?

The point is that threads are cheaper than normal processes, and that they can be scheduled for execution in a user-dependent way, with less overhead. Threads are cheaper than a whole process because they do not have a full set of resources each. Whereas the process control block for a heavyweight process is large and costly to context switch, the PCBs for threads are much smaller, since each thread has only a stack and some registers to manage. It has no open file lists or resource lists, no accounting structures to update. All of these resources are shared by all threads within the process. Threads can be assigned priorities – a higher priority thread will get put to the front of the queue. Let's define heavy and lightweight processes with the help of a table.

Object	**Resources**
Thread (Light Weight Processes)	Stack, set of CPU registers, CPU time
Task (Heavy Weight Processes)	1 thread, PCB, Program code, memory segment etc.
Multithreaded task	*n*- threads, PCB, Program code, memory segment etc.

Why use threads?

The sharing of resources can be made more effective if the scheduler knows known exactly what each program was going to do in advance. Of course, the scheduling algorithm can never know this – but the programmer who wrote the program does know. Using threads it is possible to organise the execution of a program in such a way that something is always being done, whenever the scheduler gives the heavyweight process CPU time.

- Threads allow a programmer to switch between lightweight processes when it is best for the program. (The programmer has control).
- A process which uses threads does not get more CPU time than an ordinary process – but the CPU time it gets is used to do work on the threads. It is possible to write a more efficient program by making use of threads.
- Inside a heavyweight process, threads are scheduled on a FCFS basis, unless the program decides to force certain threads to wait for other threads. If there is only one CPU, then only one thread can be running at a time.
- Threads context switch without any need to involve the kernel−the switching is performed by a user level library, so time is saved because the kernel doesn't need to know about the threads.

2.2.7 Levels of Threads

In modern operating systems, there are two levels at which threads operate: system or kernel threads and user level threads. If the kernel itself is multithreaded, the scheduler assigns CPU time on a thread basis rather than on a process basis. A kernel level thread behaves like a virtual CPU, or a power-point to which user-processes can connect in order to get computing power. The kernel has as many system level threads as it has CPUs and each of these must be shared between all of the user-threads on the system. In other words, the maximum number of user level threads which can be active at any one time is equal to the number of system level threads, which in turn is equal to the number of CPUs on the system.

Since threads work "inside" a single task, the normal process scheduler cannot normally tell which thread to run and which not to run – that is up to the program. When the kernel schedules a process for execution, it must then find out from that process which is the next thread it must execute. If the program is lucky enough to have more than one processor available, then several threads can be scheduled at the same time.

Some important implementations of threads are:

- The Mach System / OSF1 (user and system level)
- Solaris 1 (user level)

- Solaris 2 (user and system level)
- OS/2 (system level only)
- NT threads (user and system level)
- IRIX threads
- POSIX standardized user threads interface.

☞ **Check Your Progress 1**

1) Explain the difference between a process and a thread with some examples.

……………………………………………………………………………………..

…………..……………………………………………………………………

……………………………………………………………………………………..

2) Identify the different states a live process may occupy and show how a process moves between these states.

……………………………………………………………………………………

…..………………………………………………………………………………

……………………………………………………………………………………..

3) Define what is meant by a context switch. Explain the reason many systems use two levels of scheduling.

……………………………………………………………………………………

…..……………………………………………………………………………..

……………………………………………………………………………………

2.3 SYSTEM CALLS FOR PROCESS MANAGEMENT

In this section we will discuss system calls typically provided by the kernels of multiprogramming operating systems for process management. System calls provide the interface between a process and the operating system. These system calls are the routine services of the operating system. As an example of how system calls are used, consider writing a simple program to read data from one file and to copy them to another file. There are two names of two different files in which one input file and the other is the output file. One approach is for the program to ask the user for the names of the two files. 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 character that the two files have. Once the two file names are obtained the program must open the input file and create the output file. Each of these operations requires another system call and may encounter possible error conditions. When the program tries to open the input file, it may find that no file of that name exists or that the file is protected against access. In these cases the program should print the message on the console and then terminate abnormally which require another system call. If the input file exists then we must create the new output file. We may find an output file with the same name. This situation may cause the program to abort or we may delete the existing file and create a new one. After opening both files, we may enter a loop that reads from input file and writes to output file. Each read and write must return status information regarding various possible error conditions. Finally, after the entire file is copied the program may close both files. Examples of some operating system calls are:

Create: In response to the *create* call the operating system creates a new process with the specified or default attributes and identifier. Some of the parameters definable at the process creation time include:

- level of privilege, such as system or user
- priority
- size and memory requirements
- maximum data area and/or stack size
- memory protection information and access rights
- other system dependent data.

Delete: The delete service is also called destroy, terminate or exit. Its execution causes the operating system to destroy the designated process and remove it from the system.

Abort: It is used to terminate the process forcibly. Although a process could conceivably abort itself, the most frequent use of this call is for involuntary terminations, such as removal of malfunctioning process from the system.

Fork/Join: Another method of process creation and termination is by means of *FORK/JOIN* pair, originally introduced as primitives for multiprocessor system. The *FORK* operations are used to split a sequence of instruction into two concurrently executable sequences. *JOIN* is used to merge the two sequences of code divided by the *FORK* and it is available to a parent process for synchronization with a child.

Suspend: The *suspend* system call is also called *BLOCK* in some systems. The designated process is suspended indefinitely and placed in the *suspend* state. A process may be suspended itself or another process when authorised to do so.

Resume: The *resume* system call is also called *WAKEUP* in some systems. This call resumes the target process, which is presumably suspended. Obviously a suspended process can not resume itself because a process must be running to have its operating system call processed. So a suspended process depends on a partner process to issue the resume.

Delay: The system call *delay* is also known as *SLEEP*. The target process is suspended for the duration of the specified time period. The time may be expressed in terms of system clock ticks that are system dependent and not portable or in standard time units such as seconds and minutes. A process may delay itself or optionally, delay some other process.

Get_Attributes: It is an enquiry to which the operating system responds by providing the current values of the process attributes, or their specified subset, from the PCB.

Change Priority: It is an instance of a more general SET-PROCESS-ATTRIBUTES system call. Obviously, this call is not implemented in systems where process priority is static.

2.4 PROCESS SCHEDULING

Scheduling is a fundamental operating system function. All computer resources are scheduled before use. Since CPU is one of the primary computer resources, its scheduling is central to operating system design. **Scheduling** refers to a set of policies and mechanisms supported by operating system that controls the order in which the work to be done is completed. A **scheduler** is an operating system program (module) that selects the next job to be admitted for execution. The main objective of scheduling is to increase CPU utilisation and higher throughput. **Throughput** is the amount of work accomplished in a given **time interval.** CPU scheduling is the basis of operating system which supports multiprogramming concepts. By having a number of programs in computer memory at the same time, the CPU may be shared among them. This mechanism improves the overall efficiency of the computer system by getting more work done in less time. In this section we will describe the scheduling objectives, the three types of schedulers and performance criteria that schedulers may use in maximizing system performance. Finally at the end of the unit, we will study various scheduling algorithms.

2.4.1 Scheduling Objectives

The primary objective of scheduling is to improve system performance. Various objectives of the scheduling are as follows:

- **Maximize throughput:** Scheduling should attempt to service the largest possible number of processes per unit time.
- Maximize the number of interactive user receiving acceptable response times.
- **Be predictable:** A given job should utilise the same amount of time and should cost the same regardless of the load on the system.
- **Minimize overhead:** Scheduling should minimize the wasted resources overhead.
- **Balance resource use:** The scheduling mechanisms should keep the resources of the system busy. Processes that will use under utilized resources should be favoured.
- **Achieve a balance between response and utilisation:** The best way to guarantee good response times is to have sufficient resources available whenever they are needed. In real time system fast responses are essential, and resource utilisation is less important.
- **Avoid indefinite postponement :** It would be fair if all processes are treated the same, and no process can suffer indefinite postponement.
- **Enforce Priorities:** In environments in which processes are given priorities, the scheduling mechanism should favour the higher-priority processes.
- **Give preference to processes holding key resources:** Even though a low priority process may be holding a key resource, the process may be in demand by high priority processes. If the resource is not perceptible, then the scheduling mechanism should give the process better treatment that it would ordinarily receive so that the process will release the key resource sooner.
- **Degrade gracefully under heavy loads:** A scheduling mechanism should not collapse under heavy system load. Either it should prevent excessive loading by not allowing new processes to be created when the load in heavy or it should provide service to the heavier load by providing a moderately reduced level of service to all processes.

2.4.2 Types of Schedulers

If we consider batch systems, there will often be more processes submitted than the number of processes that can be executed immediately. So, the incoming processes are spooled onto a disk. There are three types of schedulers. They are:

- Short term scheduler
- Long term scheduler
- Medium term scheduler

Short term scheduler: The short-term scheduler selects the process for the processor among the processes which are already in queue (in memory). The scheduler will execute quite frequently (mostly at least once every 10 milliseconds). It has to be very fast in order to achieve a better processor utilisation. The short term scheduler, like all other OS programs, has to execute on the processor. If it takes 1 millisecond to choose a process that means (1 / (10 + 1)) = 9% of the processor time is being used for short time scheduling and only 91% may be used by processes for execution.

Long term scheduler: The long-term scheduler selects processes from the process pool and loads selected processes into memory for execution. The long-term scheduler executes much less frequently when compared with the short term scheduler. It controls the degree of multiprogramming (no. of processes in memory at a time). If the degree of multiprogramming is to be kept stable (say 10 processes at a time), the long-term scheduler may only need to be invoked till the process finishes execution. The long-term scheduler must select a good process mix of I/O-bound and processor bound processes. If most of the processes selected are I/O-bound, then the ready queue will almost be empty, while the device queue(s) will be very crowded. If most of the processes are processor-bound, then the device queue(s) will almost be empty while the ready queue is very crowded and that will cause the short-term scheduler to be invoked very frequently. Time-sharing systems (mostly) have no long-term scheduler. The stability of these systems either depends upon a physical limitation (no. of available terminals) or the self-adjusting nature of users (if you can't get response, you quit). It can sometimes be good to reduce the degree of multiprogramming by removing processes from memory and storing them on disk. These processes can then be reintroduced into memory by the **medium-term scheduler.** This operation is also known as swapping. Swapping may be necessary to improve the process mix or to free memory.

2.4.3 Scheduling Criteria

The goal of scheduling algorithm is to identify the process whose selection will result in the "best" possible system performance. There are different scheduling algorithms, which has different properties and may favour one class of processes over another, which algorithm is best, to determine this there are different characteristics used for comparison. The scheduling algorithms which we will discuss in the next section, determines the importance of each of the criteria.

In order to achieve an efficient processor management, OS tries to select the most appropriate process from the ready queue. For selecting, the relative importance of the following may be considered as performance criteria:

CPU Utilization: The key idea is that if the CPU is busy all the time, the utilization factor of all the components of the system will be also high. CPU utilization is the ratio of busy time of the processor to the total time passes for processes to finish.

Processor Utilization = (Processor busy time) / (Processor busy time + Processor idle time)

Throughput: It refers to the amount of work completed in a unit of time. One way to measure throughput is by means of the number of processes that are completed in a unit of time. The higher the number of processes, the more work apparently is being done by the system. But this approach is not very useful for comparison because this is dependent on the characteristics and resource requirement of the process being executed. Therefore to compare throughput of several scheduling algorithms it should be fed into the process with similar requirements. The throughput can be calculated by using the formula:

Throughput = (No. of processes completed) / (Time unit)

Turnaround Time : It may be defined as interval from the time of submission of a process to the time of its completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, CPU time and I/O operations.

Turnaround Time = t(Process completed) – t(Process submitted)

Waiting Time: This is the time spent in the ready queue. In multiprogramming operating system several jobs reside at a time in memory. CPU executes only one job at a time. The rest of jobs wait for the CPU. The waiting time may be expressed as turnaround time, less than the actual processing time.

Waiting time = Turnaround Time - Processing Time

But the scheduling algorithm affects or considers the amount of time that a process spends waiting in a ready queue. Thus rather than looking at turnaround time waiting time is usually the waiting time for each process.

Response time: It is most frequently considered in time sharing and real time operating system. However, its characteristics differ in the two systems. In time sharing system it may be defined as interval from the time the last character of a command line of a program or transaction is entered to the time the last result appears on the terminal. In real time system it may be defined as interval from the time an internal or external event is signalled to the time the first instruction of the respective service routine is executed.

Response time = t(first response) – t(submission of request)

One of the problems in designing schedulers and selecting a set of its performance criteria is that they often conflict with each other. For example, the fastest response time in time sharing and real time system may result in low CPU utilisation.

Throughput and CPU utilization may be increased by executing the large number of processes, but then response time may suffer. Therefore, the design of a scheduler usually requires balance of all the different requirements and constraints. In the next section we will discuss various scheduling algorithms.

☞ **Check Your Progress 2**

1) Distinguish between a foreground and a background process in UNIX.

………………………………………………………………………………………...

…………………………………………………………………………………………

…………………………………………………………………………………………

2) Identify the information which must be maintained by the operating system for each live process.

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

2.5 SCHEDULING ALGORITHMS

Now let’s discuss some processor scheduling algorithms again stating that the goal is to select the most appropriate process in the ready queue.

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated to the CPU. There are several scheduling algorithms which will be examined in this section. A major division among scheduling algorithms is that whether they support **pre-emptive** or **non-preemptive scheduling** discipline.

Preemptive scheduling: Preemption means the operating system moves a process from running to ready without the process requesting it. An OS implementing this algorithms switches to the processing of a new request before completing the processing of the current request. The preempted request is put back into the list of the pending requests. Its servicing would be resumed sometime in the future when it is scheduled again. Preemptive scheduling is more useful in high priority process which requires immediate response. For example, in Real time system the consequence of missing one interrupt could be dangerous.

Round Robin scheduling, priority based scheduling or event driven scheduling and SRTN are considered to be the preemptive scheduling algorithms.

Non–Preemptive scheduling: A scheduling discipline is non-preemptive if once a process has been allotted to the CPU, the CPU cannot be taken away from the process. A non-preemptible discipline always processes a scheduled request to its completion. In non-preemptive systems, jobs are made to wait by longer jobs, but the treatment of all processes is fairer.

First come First Served (FCFS) and Shortest Job First (SJF), are considered to be the non-preemptive scheduling algorithms.

The decision whether to schedule preemptive or not depends on the environment and the type of application most likely to be supported by a given operating system.

2.5.1 First-Come First-Serve (FCFS)

The simplest scheduling algorithm is First Come First Serve (FCFS). Jobs are scheduled in the order they are received. FCFS is non-preemptive. Implementation is easily accomplished by implementing a queue of the processes to be scheduled or by storing the time the process was received and selecting the process with the earliest time.

FCFS tends to favour CPU-Bound processes. Consider a system with a CPU-bound process and a number of I/O-bound processes. The I/O bound processes will tend to execute briefly, then block for I/O. A CPU bound process in the ready should not have to wait long before being made runable. The system will frequently find itself with all the I/O-Bound processes blocked and CPU-bound process running. As the I/O operations complete, the ready Queue fill up with the I/O bound processes.

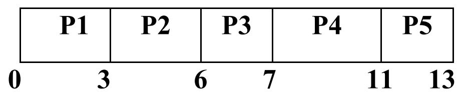
Under some circumstances, CPU utilisation can also suffer. In the situation described above, once a CPU-bound process does issue an I/O request, the CPU can return to process all the I/O-bound processes. If their processing completes before the CPU-bound process's I/O completes, the CPU sits idle. So with no preemption, component utilisation and the system throughput rate may be quite low.

Example:

Calculate the turn around time, waiting time, average turnaround time, average waiting time, throughput and processor utilization for the given set of processes that arrive ***at a given arrive time*** shown in the table, with the length of processing time given in milliseconds:

Process	Arrival Time	Processing Time
P1	0	3
P2	2	3
P3	3	1
P4	5	4
P5	8	2

If the processes arrive as per the arrival time, the Gantt chart will be



Time	Process Completed	Turn around Time = t(Process Completed) –t(Process Submitted)	Waiting Time = Turn around time – Processing time
0	-	-	-
3	P1	3 – 0 = 3	3 – 3 = 0
6	P2	6 – 2 = 4	4 – 3 = 1
7	P3	7 – 1 = 6	6 – 1 = 5
11	P4	11– 4 = 7	7 – 4 = 3
13	P5	13 – 2 = 11	11– 2 = 9

Average turn around time = (3+4+6+7+11) / 5 = 6.2

Average waiting time = (0+1+5+3+9) / 5 = 3.6

Processor utilization = (13/13)*100 = 100%

Throughput = 5/13 = 0.38

Note: If all the processes arrive at time 0, then the order of scheduling will be P3, P5, P1, P2 and P4.

2.5.2 Shortest-Job First (SJF)

This algorithm is assigned to the process that has smallest next CPU processing time, when the CPU is available. In case of a tie, FCFS scheduling algorithm can be used. It is originally implemented in a batch-processing environment. SJF relied on a time estimate supplied with the batch job.

As an example, consider the following set of processes with the following processing time which arrived at the same time.

Process	**Processing Time**
P1	06
P2	08
P3	07
P4	03

Using SJF scheduling because the shortest length of process will first get execution, the Gantt chart will be:

P4	**P1**	**P3**	**P2**

0 3 9 16 24

Because the shortest processing time is of the process P4, then process P1 and then P3 and Process P2. The waiting time for process P1 is 3 ms, for process P2 is 16 ms, for process P3 is 9 ms and for the process P4 is 0 ms as –

Time	**Process Completed**	**Turn around Time = t (Process Completed)– t (Process Submitted)**	**Waiting Time = Turn around time – Processing time**
0	-	-	-
3	P4	3 – 0 = 3	3 – 3 = 0
9	P1	9 – 0 = 9	9 – 6 = 3
16	P3	16 – 0 = 16	16 – 7 = 9
24	P2	24 – 0= 24	24 – 8 = 16

Average turn around time = (3+9+16+24) / 4 = 13

Average waiting time = (0+3+9+16) / 4 = 7

Processor utilization = (24/24)*100 = 100%

Throughput = 4/24 = 0.16

2.5.3 Round Robin (RR)

Round Robin (RR) scheduling is a preemptive algorithm that relates the process that has been waiting the longest. This is one of the oldest, simplest and widely used algorithms. The round robin scheduling algorithm is primarily used in time-sharing and a multi-user system environment where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. Basically the CPU time is divided into time slices.

Each process is allocated a small time-slice called quantum. No process can run for more than one quantum while others are waiting in the ready queue. If a process needs more CPU time to complete after exhausting one quantum, it goes to the end of ready queue to await the next allocation. To implement the RR scheduling, Queue data structure is used to maintain the Queue of Ready processes. A new process is added at the tail of that Queue. The CPU schedular picks the first process from the ready Queue, Allocate processor for a specified time Quantum. After that time the CPU schedular will select the next process is the ready Queue.

Consider the following set of process with the processing time given in milliseconds.

Process	Processing Time
P1	24
P2	03
P3	03

If we use a time **Quantum of 4 milliseconds** then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time Quantum, and the CPU is given to the next process in the Queue, Process P2. Since process P2 does not need and milliseconds, it quits before its time Quantum expires. The CPU is then given to the next process, Process P3 one each process has received 1 time Quantum, the CPU is returned to process P1 for an additional time quantum. The Gantt chart will be:

P1	**P2**	**P3**	**P1**	**P1**	**P1**	**P1**	**P1**

0 4 7 10 14 18 22 26 30

Process	**Processing Time**	**Turn around time = t(Process Completed) – t(Process Submitted)**	**Waiting Time = Turn around time – Processing time**
P1	24	30 – 0 = 30	30 – 24 = 6
P2	03	7 – 0 = 7	7 – 3 = 4
P3	03	10 – 0 =10	10 – 3 = 7

Average turn around time = (30+7+10)/3 = 47/3 = 15.66

Average waiting time = (6+4+7)/3 = 17/3 = 5.66

Throughput = 3/30 = 0.1

Processor utilization = (30/30) * 100 = 100%

2.5.4 Shortest Remaining Time Next (SRTN)

This is the preemptive version of shortest job first. This permits a process that enters the ready list to preempt the running process if the time for the new process (or for its next burst) is less than the *remaining* time for the running process (or for its current burst). Let us understand with the help of an example.

Consider the set of four processes arrived as per timings described in the table:

Process	Arrival time	Processing time
P1	0	5
P2	1	2
P3	2	5
P4	3	3

At time 0, only process P1 has entered the system, so it is the process that executes. At time 1, process P2 arrives. At that time, process P1 has 4 time units left to execute At this juncture process 2's processing time is less compared to the P1 left out time (4 units). So P2 starts executing at time 1. At time 2, process P3 enters the system with the processing time 5 units. Process P2 continues executing as it has the minimum number of time units when compared with P1 and P3. At time 3, process P2 terminates and process P4 enters the system. Of the processes P1, P3 and P4, P4 has the smallest remaining execution time so it starts executing. When process P1 terminates at time 10, process P3 executes. The Gantt chart is shown below:

P1	**P2**	**P4**	**P1**	**P3**

0 1 3 6 10 15

Turnaround time for each process can be computed by subtracting the time it terminated from the arrival time.

Turn around Time = t(Process Completed)– t(Process Submitted)

The turnaround time for each of the processes is:

P1: 10 – 0 = 10

P2: 3 – 1 = 2

P3: 15 – 2 = 13

P4: 6 – 3 = 3

The average turnaround time is (10+2+13+3) / 4 = 7

The waiting time can be computed by subtracting processing time from turnaround time, yielding the following 4 results for the processes as

P1: 10 – 5 = 5

P2: 2 – 2 = 0

P3: 13 – 5 = 8

P4: 3 – 3 = 0

The average waiting time = (5+0+8+0) / 4 = 3.25milliseconds

Four jobs executed in 15 time units, so throughput is 15 / 4 = 3.75 time units/job.

2.5.5 Priority Based Scheduling or Event-Driven (ED) Scheduling

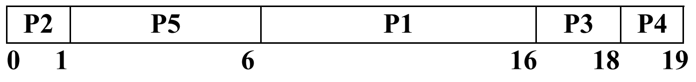
A priority is associated with each process and the scheduler always picks up the highest priority process for execution from the ready queue. Equal priority processes are scheduled FCFS. The level of priority may be determined on the basis of resource requirements, processes characteristics and its run time behaviour.

A major problem with a priority based scheduling is indefinite blocking of a lost priority process by a high priority process. In general, completion of a process within finite time cannot be guaranteed with this scheduling algorithm. A solution to the problem of indefinite blockage of low priority process is provided by aging priority. Aging priority is a technique of gradually increasing the priority of processes (of low priority) that wait in the system for a long time. Eventually, the older processes attain high priority and are ensured of completion in a finite time.

As an example, consider the following set of five processes, assumed to have arrived at the same time with the length of processor timing in milliseconds: –

Process	Processing Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Using priority scheduling we would schedule these processes according to the following Gantt chart:



Time	Process Completed	Turn around Time = t(Process Completed) – t(Process Submitted)	Waiting Time = Turn around time – Processing time
0	-	-	-
1	P2	1 – 0 = 1	1 – 1 = 0
6	P5	6 – 0 = 6	6 – 2 = 4
16	P1	16 – 0 = 16	16 – 10 = 6
18	P3	18 – 0 = 18	18 – 2 = 16
19	P4	19 – 0 = 19	19 – 1 = 18

Average turn around time = (1+6+16+18+19) / 5 = 60/5 = 12
Average waiting time = (6+0+16+18+1) / 5 = 8.2
Throughput = 5/19 = 0.26
Processor utilization = (30/30) * 100 = 100%

Priorities can be defined either internally or externally. Internally defined priorities use one measurable quantity or quantities to complete the priority of a process.

2.6 PERFORMANCE EVALUATION OF THE SCHEDULING ALGORITHMS

Performance of an algorithm for a given set of processes can be analysed if the appropriate information about the process is provided. But how do we select a CPU-scheduling algorithm for a particular system? There are many scheduling algorithms so the selection of an algorithm for a particular system can be difficult. To select an algorithm there are some specific criteria such as:

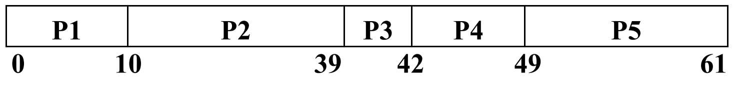
- Maximize CPU utilization with the maximum response time.
- Maximize throughput.

For example, assume that we have the following five processes arrived at time 0, in the order given with the length of CPU time given in milliseconds.

Process	Processing time
P1	10
P2	29
P3	03
P4	07
P5	12

First consider the FCFS scheduling algorithm for the set of processes.

For FCFS scheduling the Gantt chart will be:



Process	Processing time	Waiting time
P1	10	0
P2	29	10
P3	03	39
P4	07	42
P5	12	49

Average Waiting Time: (0+10+39+42+49) / 5 = 28 milliseconds.

Now consider the SJF scheduling, the Gantt chart will be:

P3	P4	P1	P5	P2

```
0    3         10             20            32           61
```

Process	Processing time	Waiting time
P1	10	10
P2	29	32
P3	03	00
P4	07	03
P5	12	20

Average Waiting Time: (10+32+0+3+20)/5 = 13 milliseconds.

Now consider the Round Robin scheduling algorithm with a quantum of 10 milliseconds. The Gantt chart will be:

P1	P2	P3	P4	P5	P2	P5	P2

```
0        10        20    23       30         40        49    52          61
```

Process	Processing time	Waiting time
P1	10	0
P2	29	32
P3	03	20
P4	07	23
P5	12	40

Average waiting time = (0+32+20+23+40) / 5 = 23 milliseconds

Now if we compare average waiting time above algorithms, we see that SJF policy results in less than one half of the average waiting time to that of FCFS scheduling; the RR algorithm gives us an intermediate value.

So performance of algorithm can be measured when all the necessary information is provided.

☞ Check Your Progress 3

1) Explain the difference between voluntary or co-operative scheduling and preemptive scheduling. Give two examples of preemptive and of non-preemptive scheduling algorithms.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) Outline how different process priorities can be implemented in a scheduling algorithm. Explain why these priorities need to be dynamically adjusted.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) Draw the Gantt chart for the FCFS policy, considering the following set of processes that arrive at time 0, with the length of CPU time given in milliseconds. Also calculate the Average Waiting Time.

Process	Processing Time
P1	13
P2	08
P3	83

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4) For the given five processes arriving at time 0, in the order with the length of CPU time in milliseconds:

Process	Processing Time
P1	10
P2	29
P3	03
P4	07

Consider the FCFS, SJF and RR (time slice= 10 milliseconds) scheduling algorithms for the above set of process which algorithm would give the minimum average waiting time?

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

2.7 SUMMARY

A process is an instance of a program in execution. A process can be defined by the system or process is an important concept in modem operating system. Processes provide a suitable means for informing the operating system about independent activities that may be scheduled for concurrent execution. Each process is represented by a process control block (PCB). Several PCB's can be linked together to form a queue of waiting processes. The selection and allocation of processes is done by a scheduler. There are several scheduling algorithms. We have discussed FCFS, SJF, RR, SJRT and priority algorithms along with the performance evaluation.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) A process is an instance of an executing program and its data. For example, if you were editing 3 files simultaneously, you would have 3 processes, even though they might be sharing the same code.

   A thread is often called a lightweight process. It is a "child" and has all the state information associated with a child process but does not run in a separate address space i.e., doesn't have its own memory. Since they share memory, each thread has access to all variables and any change made by one thread impacts all the others. Threads are useful in multi-client servers where each thread can service a separate client connection. In some stand-alone processes where you can split the processing such that one thread can continue processing while another is blocked e.g., waiting for an I/O request or a timer thread can implement a regular repaint of a graphics canvas.

This is how simple animations are accomplished in systems like NT and Windows95. Unix is a single-threaded operating system and can't do this at the operating system level although some applications e.g., Java can support it via software.

2) A process can be in one of the following states:

Ready i.e., in the scheduling queue waiting it's turn for the CPU
Running i.e., currently occupying the CPU
Blocked i.e., sleeping waiting for a resource request to be satisfied.
Halted i.e., process execution is over.

- A process moves from ready to running, when it is dispatched by the scheduler.
- A process moves from running to ready, when it is pre-empted by the scheduler.
- A process moves from running to blocked when it issues a request for a resource.
- A process moves from blocked to ready when completion of the request is signaled.

3) A context switch occurs whenever a different process is moved into the CPU. The current state of the existing process (register and stack contents, resources like open files etc.) must be stored and the state of the new process restored. There is a significant overhead in context switching. However, in a system with Virtual Memory, some processes and some of the state information will be swapped to disks. The overhead involved here is too great to undertake each quantum. Such systems use a two-level scheduler. The high-level scheduler is responsible for bring ready processes from disk into memory and adding them to ready queue. This is done regularly (e.g., every 10 secs) but not each quantum (e.g., < every 100 msecs). The low level scheduler is responsible for the context switching between CPU and ready queue.

Check Your Progress 2

1) Every process has a parent which invoked it. In the UNIX environment, the parent can wait for the child to complete after invoking it (foreground child process) or continue in a ready state (background child process).

2) For each process, the operating system needs to maintain

- id information - process id, parent process id
- summary status - blocked, ready, running, swapped to disk
- owner information - user id, group id
- scheduling info - priority, nice value, CPU usage
- location info - resident or not, memory areas allocated
- state info - register values (instruction pointer etc.), stack, resources like open files etc.

Check Your Progress 3

1) Preemptive multi-tasking means that the operating system decides when a process has had its quantum of CPU, and removes it from the CPU to the ready queue. This is present in UNIX, NT and Windows 95.
Voluntary and co-operative means that only the process itself decide to vacate the CPU e.g., when it finishes or blocks on a request or yields the CPU. This is what Windows3.x uses.

First Come First Served is a typical non-premptive algorithm.
Round Robin and Priority Queues are typical preemptive algorithms.
Shortest Job First can be either.

2) Some processes are more critical to overall operation than others e.g., kernel activity. I/O bound processes, which seldom use a full quantum, need a boost over compute bound tasks. This priority can be implemented by any combination of higher positioning in the ready queue for higher priority more frequent occurrences in the ready list for different priorities different quantum lengths for different priorities. Priority scheduling is liable to **starvation**. A succession of high priority processes may result in a low priority process never reaching the head of the ready list. Priority is normal a combination of static priority (set by the user or the class of the process) and a dynamic adjustment based on it's history (e.g., how long it has been waiting). NT and Linux also boost priority when processes return from waiting for I/O.

3) If the process arrives in the order P1, P2 and P3, then the Gantt chart will be as:

```
|    P1    |     P2     |               P3                 |
0         13           21                                 104
```

Process	**Processing time**	**Waiting Time**
P1	13	00
P2	08	13
P3	83	21

Average Waiting Time: (0+13+21)/3 = 11.33 ms.

4) For FCFS algorithm the Gantt chart is as follows:

```
| P1          | P2       | P3      | P4       | P5             |
0            10         39        42         49               51
```

Process	**Processing Time**	**Waiting time**
P1	10	0
P2	29	10
P3	3	39
P4	7	42
P5	12	49

Average Waiting Time = (0+10+39+42+49) / 5 = 5

2. With SJF scheduling algorithm, we have

```
| P3      | P4        | P1           | P5           | P2                    |
0         3          10             20             32                      61
```

Process	**Processing Time**	**Waiting time**
P1	10	10
P2	29	32
P3	3	00
P4	7	3
P5	12	20

Average Waiting Time = (10+32+00+03+20)/5 = 13 milliseconds.
With round robin scheduling algorithm (time quantum = 10 milliseconds)

P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	
0	**10**	**20**	**23**	**30**	**40**	**50**	**52**	**61**

Process	Processing Time	Waiting time
P1	10	0
P2	29	32
P3	03	20
P4	07	23
P5	12	40

Average Waiting Time = (0+32+20+23+40)/5 = 23 milliseconds.

From the above calculations of average waiting time we found that SJF policy results in less than one half of the average waiting time obtained from FCFS, while Round Robin gives intermediate result.

2.9 FURTHER READINGS

1) Abraham Silberschatz and James L, *Operating system Concepts*. Peterson, Addition Wesely Publishing Company.

2) Andrew S. Tanenbaum, *Operating System Design and Implementation,* PHI

3) D.M. Dhamdhere, *Operating Systems, A Concept-based Approach*, TMGH, 2002.

4) Harvay M. Deital, *Introduction to Operating Systems,* Addition Wesely Publishing Company

5) Madnick and Donovan, *Operating systems – Concepts and Design* Mc GrawHill Intl. Education.

6) Milan Milenkovic, *Operating Systems, Concepts and Design*, TMGH, 2000.

UNIT 3 INTERPROCESS COMMUNICATION AND SYNCHRONIZATION

Structure **Page Nos.**

3.0 INTRODUCTION

In the earlier unit we have studied the concept of processes. In addition to process scheduling, another important responsibility of the operating system is process synchronization. Synchronization involves the orderly sharing of system resources by processes.

Concurrency specifies two or more sequential programs (a sequential program specifies sequential execution of a list of statements) that may be executed concurrently as a parallel process. For example, an airline reservation system that involves processing transactions from many terminals has a natural specification as a concurrent program in which each terminal is controlled by its own sequential process. Even when processes are not executed simultaneously, it is often easier to structure as a collection of cooperating sequential processes rather than as a single sequential program.

A simple batch operating system can be viewed as 3 processes−a reader process, an executor process and a printer process. The reader reads cards from card reader and places card images in an input buffer. The executor process reads card images from input buffer and performs the specified computation and store the result in an output buffer. The printer process retrieves the data from the output buffer and writes them to a printer. Concurrent processing is the basis of operating system which supports multiprogramming.

The operating system supports concurrent execution of a program without necessarily supporting elaborate form of memory and file management. This form of operation is also known as multitasking. One of the benefits of multitasking is that several processes can be made to cooperate in order to achieve their goals. To do this, they must do one of the following:

Communicate: Interprocess communication (IPC) involves sending information from one process to another. This can be achieved using a "mailbox" system, a socket which behaves like a virtual communication network (loopback), or through the use of “pipes”. Pipes are a system construction which enable one process to open another process as if it were a file for writing or reading.

Share Data: A segment of memory must be available to both the processes. (Most memory is locked to a single process).

Waiting: Some processes wait for other processes to give a signal before continuing. This is an issue of synchronization.

In order to cooperate concurrently executing processes must communicate and synchronize. Interprocess communication is based on the use of *shared variables* (variables that can be referenced by more than one process) or *message passing*.

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects. This only works if the events perform an action or detect an action are constrained to happen in that order. Thus one can view synchronization as a set of constraints on the ordering of events. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.

In this unit, let us study the concept of interprocess communication and synchronization, need of semaphores, classical problems in concurrent processing, critical regions, monitors and message passing.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- identify the significance of interprocess communication and synchronization;
- describe the two ways of interprocess communication namely shared memory and message passing;
- discuss the usage of semaphores, locks and monitors in interprocess and synchronization, and
- solve classical problems in concurrent programming.

3.2 INTERPROCESS COMMUNICATION

Interprocess communication (IPC) is a capability supported by operating system that allows one *process* to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprogramming systems, but it is not generally supported by single-process operating systems such as DOS. OS/2 and MS-Windows support an IPC mechanism called Dynamic Data Exchange.

IPC allows the process to communicate and to synchronize their actions without sharing the same address space. This concept can be illustrated with the example of a shared printer as given below:

Consider a machine with a single printer running a time-sharing operation system. If a process needs to print its results, it must request that the operating system gives it access to the printer’s device driver. At this point, the operating system must decide whether to grant this request, depending upon whether the printer is already being used by another process. If it is not, the operating system should grant the request and allow the process to continue; otherwise, the operating system should deny the request and perhaps classify the process as a waiting process until the printer becomes available. Indeed, if two processes were given simultaneous access to the machine's printer, the results would be worthless to both.

Consider the following related definitions to understand the example in a better way:

Critical Resource: It is a resource shared with constraints on its use (e.g., memory, files, printers, etc).

Critical Section: It is code that accesses a critical resource.

Mutual Exclusion: At most one process may be executing a critical section with respect to a particular critical resource simultaneously.

In the example given above, the printer is the *critical resource*. Let's suppose that the processes which are sharing this resource are called process A and process B. The *critical sections* of process A and process B are the sections of the code which issue the print command. In order to ensure that both processes do not attempt to use the printer at the same, they must be granted *mutually exclusive* access to the printer driver.

First we consider the interprocess communication part. There exist two complementary inter-process communication types: a) shared-memory system and b) message-passing system. It is clear that these two schemes are not mutually exclusive, and could be used simultaneously within a single operating system.

3.2.1 Shared-Memory System

Shared-memory systems require communication processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory scheme, the responsibility for providing communication rests with the application programmers. The operating system only needs to provide shared memory.

A critical problem occurring in shared-memory system is that two or more processes are reading or writing some shared variables or shared data, and the final results depend on who runs precisely and when. Such situations are called *race conditions*. In order to avoid race conditions we must find some way to prevent more than one process from reading and writing shared variables or shared data at the same time, i.e., we need the concept of *mutual exclusion* (which we will discuss in the later section*)*. It must be sure that if one process is using a shared variable, the other process will be excluded from doing the same thing.

3.2.2 Message-Passing System

Message passing systems allow communication processes to exchange messages. In this scheme, the responsibility rests with the operating system itself.

The function of a message-passing system is to allow processes to communicate with each other without the need to resort to shared variable. An interprocess communication facility basically provides two operations: *send* (message) and *receive* (message). In order to send and to receive messages, a communication link must exist between two involved processes. This link can be implemented in different ways. The possible basic implementation questions are:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of process?
- What is the capacity of a link? That is, does the link have some buffer space? If so, how much?
- What is the size of the message? Can the link accommodate variable size or fixed-size message?
- Is the link unidirectional or bi-directional?

In the following we consider several methods for logically implementing a communication link and the send/receive operations. These methods can be classified into two categories: a) Naming, consisting of direct and indirect communication and b) Buffering, consisting of capacity and messages proprieties.

Direct Communication

In direct communication, each process that wants to send or receive a message must explicitly name the recipient or sender of the communication. In this case, the send and receive primitives are defined as follows:

• send (P, message)	To send a message to process P
• receive (Q, message)	To receive a message from process Q

This scheme shows the *symmetry* in addressing, i.e., both the sender and the receiver have to name one another in order to communicate. In contrast to this, *asymmetry* in addressing can be used, i.e., only the sender has to name the recipient; the recipient is not required to name the sender. So the send and receive primitives can be defined as follows:

• send (P, message)	To send a message to the process P
• receive (id, message)	To receive a message from any process; *id* is set to the name of the process with whom the communication has taken place.

Indirect Communication

With indirect communication, the messages are sent to, and received from a *mailbox*. A mailbox can be abstractly viewed as an object into which messages may be placed and from which messages may be removed by processes. In order to distinguish one from the other, each mailbox owns a unique identification. A process may communicate with some other process by a number of different mailboxes. The send and receive primitives are defined as follows:

• send (A, message)	To send a message to the mailbox A
• receive (A, message)	To receive a message from the mailbox A

Mailboxes may be owned either by a process or by the system. If the mailbox is owned by a process, then we distinguish between the *owner* who can only receive from this mailbox and *user* who can only send message to the mailbox. When a process that owns a mailbox terminates, its mailbox disappears. Any process that sends a message to this mailbox must be notified in the form of an exception that the mailbox no longer exists.

If the mailbox is owned by the operating system, then it has an existence of its own, i.e., it is independent and not attached to any particular process. The operating system provides a mechanism that allows a process to: a) create a new mailbox, b) send and receive message through the mailbox and c) destroy a mailbox. Since all processes with access rights to a mailbox may terminate, a mailbox may no longer be accessible by any process after some time. In this case, the operating system should reclaim whatever space was used for the mailbox.

Capacity Link

A link has some capacity that determines the number of messages that can temporarily reside in it. This propriety can be viewed as a queue of messages attached to the link. Basically there are three ways through which such a queue can be implemented:

***Zero capacity*:** This link has a message queue length of zero, i.e., no message can wait in it. The sender must wait until the recipient receives the message. The two processes must be synchronized for a message transfer to take place. The zero-capacity link is referred to as a message-passing system without buffering.

***Bounded capacity*:** This link has a limited message queue length of *n*, i.e., at most *n* messages can reside in it. If a new message is sent, and the queue is not full, it is placed in the queue either by copying the message or by keeping a pointer to the message and the sender should continue execution without waiting. Otherwise, the sender must be delayed until space is available in the queue.

***Unbounded capacity*:** This queue has potentially infinite length, i.e., any number of messages can wait in it. That is why the sender is never delayed.

Bounded and unbounded capacity link provide message-passing system with automatic buffering.

Messages

Messages sent by a process may be one of three varieties: a) fixed-sized, b) variable-sized and c) typed messages. If only fixed-sized messages can be sent, the physical implementation is straightforward. However, this makes the task of programming more difficult. On the other hand, variable-size messages require more complex physical implementation, but the programming becomes simpler. Typed messages, i.e., associating a type with each mailbox, are applicable only to indirect communication. The messages that can be sent to, and received from a mailbox are restricted to the designated type.

3.3 INTERPROCESS SYNCHRONIZATION

When two or more processes work on the same data simultaneously strange things can happen. Suppose, when two parallel threads attempt to update the same variable simultaneously, the result is unpredictable. The value of the variable afterwards depends on which of the two threads was the last one to change the value. This is called a *race condition*. The value depends on which of the threads wins the race to update the variable. What we need in a mulitasking system is a way of making such situations predictable. This is called *serialization*. Let us study the serialization concept in detail in the next section.

3.3.1 Serialization

The key idea in process synchronization is *serialization*. This means that we have to go to some pains to *undo* the work we have put into making an operating system perform several tasks in parallel. As we mentioned, in the case of print queues, parallelism is not always appropriate.

Synchronization is a large and difficult topic, so we shall only undertake to describe the problem and some of the principles involved here.

There are essentially two strategies to serializing processes in a multitasking environment.

- The scheduler can be disabled for a short period of time, to prevent control being given to another process during a critical action like modifying shared data. This method is very inefficient on multiprocessor machines, since all other processors have to be halted every time one wishes to execute a critical section.

- A protocol can be introduced which all programs sharing data must obey. The protocol ensures that processes have to queue up to gain access to shared data. Processes which ignore the protocol ignore it at their own peril (and the peril of the remainder of the system!). This method works on multiprocessor machines also, though it is more difficult to visualize. The responsibility of serializing important operations falls on programmers. The OS cannot impose any restrictions on silly behaviour−it can only provide tools and mechanisms to assist the solution of the problem.

3.3.2 Mutexes: Mutual Exclusion

When two or more processes must share some object, an arbitration mechanism is needed so that they do not try to use it at the same time. The particular object being shared does not have a great impact on the choice of such mechanisms. Consider the following examples: two processes sharing a printer must take turns using it; if they attempt to use it simultaneously, the output from the two processes may be mixed into an arbitrary jumble which is unlikely to be of any use. Two processes attempting to update the same bank account must take turns; if each process reads the current balance from some database, updates it, and then writes it back, one of the updates will be lost.

Both of the above examples can be solved if there is some way for each process to exclude the other from using the shared object during critical sections of code. Thus the general problem is described as the mutual exclusion problem. The mutual exclusion problem was recognised (and successfully solved) as early as 1963 in the Burroughs AOSP operating system, but the problem is sufficiently difficult widely understood for some time after that. A significant number of attempts to solve the mutual exclusion problem have suffered from two specific problems, the lockout problem, in which a subset of the processes can conspire to indefinitely lock some other process out of a critical section, and the deadlock problem, where two or more processes simultaneously trying to enter a critical section lock each other out.

On a uni-processor system with non-preemptive scheduling, mutual exclusion is easily obtained: the process which needs exclusive use of a resource simply refuses to relinquish the processor until it is done with the resource. A similar solution works on a preemptively scheduled uni-processor: the process which needs exclusive use of a resource disables interrupts to prevent preemption until the resource is no longer needed. These solutions are appropriate and have been widely used for short critical sections, such as those involving updating a shared variable in main memory. On the other hand, these solutions are not appropriate for long critical sections, for example, those which involve input/output. As a result, users are normally forbidden to use these solutions; when they are used, their use is restricted to system code.

Mutual exclusion can be achieved by a system of *locks*. A mutual exclusion lock is colloquially called a *mutex*. You can see an example of mutex locking in the multithreaded file reader in the previous section. The idea is for each thread or process to try to obtain locked-access to shared data:

```
Get_Mutex(m);

// Update shared data

Release_Mutex(m);
```

This protocol is meant to ensure that only one process at a time can get past the function Get_Mutex. All other processes or threads are made to wait at the function Get_Mutex until that one process calls Release_Mutex to release the lock. A method for implementing this is discussed below. Mutexes are a central part of multithreaded programming.

3.3.3 Critical Sections: The Mutex Solution

A critical section is a part of a program in which is it necessary to have exclusive access to shared data. Only one process or a thread may be in a critical section at any one time. The characteristic properties of the code that form a Critical Section are:

- Codes that refer one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in "read-update-write" fashion by another thread.
- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it possibly in use by another thread.

In the past it was possible to implement this by generalising the idea of interrupt masks. By switching off interrupts (or more appropriately, by switching off the scheduler) a process can guarantee itself uninterrupted access to shared data. This method has drawbacks:

i) Masking interrupts can be dangerous− there is always the possibility that important interrupts will be missed;

ii) It is not general enough in a multiprocessor environment, since interrupts will continue to be serviced by other processors−so all processors would have to be switched off;

iii) It is too harsh. We only need to prevent two programs from being in their critical sections simultaneously if they share the same data. Programs A and B might share different data to programs C and D, so why should they wait for C and D?

In 1981 G.L. Peterson discovered a simple algorithm for achieving mutual exclusion between two processes with PID equal to 0 or 1. The code is as follows:

```
int turn;
int interested[2];

void Get_Mutex (int pid)

{
int other;
other = 1 - pid;
interested[process] = true;
turn = pid;

while (turn == pid && interested[other])  // Loop until no one
   {                                      // else is interested
   }
}

Release_Mutex (int pid)

{
interested[pid] = false;
}
```

Where more processes are involved, some modifications are necessary to this algorithm. The key to serialization here is that, if a second process tries to obtain the mutex, when another already has it, it will get caught in a loop, which does not terminate until the other process has released the mutex. This solution is said to involve *busy waiting*−i.e., the program actively executes an empty loop, wasting CPU cycles, rather than moving the process out of the scheduling queue. This is also called a *spin lock*, since the system 'spins' on the loop while waiting. Let us see another algorithm which handles critical section problem for ***n*** processes.

3.3.4 Dekker's solution for Mutual Exclusion

Mutual exclusion can be assured even when there is no underlying mechanism such as the test-and-set instruction. This was first realised by *T. J. Dekker* and published (by *Dijkstra*) in 1965. Dekker's algorithm uses busy waiting and works for only two processes. The basic idea is that processes record their interest in entering a critical section (in Boolean variables called "need") and they take turns (using a variable called "turn") when both need entry at the same time. Dekker's solution is shown below:

```
type processid = 0..1;
```

```
var need: array [ processid ] of boolean { initially false };
    turn: processid { initially either 0 or 1 };

procedure dekkerwait( me: processid );
var other: processid;
begin
    other := 1 - me;
    need[ me ] := true;
    while need[ other ] do begin { there is contention }
        if turn = other then begin
            need[ me ] := false { let other take a turn };
            while turn = other do { nothing };
            need[ me ] := true { re-assert my interest };
        end;
    end;
end { dekkerwait };

procedure dekkersignal( me: processid );
begin
    need[ me ] := false;
    turn := 1 - me { now, it is the other's turn };
end { dekkersignal };
```

Dekkers solution to the mutual exclusion problem requires that each of the contending processes have a unique process identifier which is called "me" which is passed to the wait and signal operations. Although none of the previously mentioned solutions require this, most systems provide some form of process identifier which can be used for this purpose.

It should be noted that Dekker's solution does rely on one very simple assumption about the underlying hardware; it assumes that if two processes attempt to write two different values in the same memory location at the same time, one or the other value will be stored and not some mixture of the two. This is called the atomic update assumption. The atomically updatable unit of memory varies considerably from one system to another; on some machines, any update of a word in memory is atomic, but an attempt to update a byte is not atomic, while on others, updating a byte is atomic while words are updated by a sequence of byte updates.

3.3.5 Bakery's Algorithm

Bakery algorithm handles critical section problem for *n* processes as follows:

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (ticket #, process id #)
    - o (a,b) < (c,d) if a < c or if a = c and b < d
    - o $\max(a_0, \ldots, a_{n-1})$ is a number, k, such that $k >= a_i$ for i = 0, . . . , n - 1
- Shared data

    boolean choosing[n]; //initialise all to false
    int number[n]; //initialise all to 0

- Data structures are initialized to false and 0, respectively.

The algorithm is as follows:

```
do
 {
   choosing[i] = true;
   number[i] = max(number[0], number[1], ...,number[n-1]) + 1;
   choosing[i] = false;

     for(int j = 0; j < n; j++)
     {
             while (choosing[j]== true)
             {
             /*do nothing*/
             }

             while ((number[j]!=0) &&
              (number[j],j)< (number[i],i))
                    // see Reference point
                 {
                        /*do nothing*/
                 }
       }

           do critical section
                number[i] = 0;
           do remainder section
  }while (true)
```

In the next section we will study how the semaphores provides a much more organise approach of synchronization of processes.

3.4 SEMAPHORES

Semaphores provide a much more organised approach to controlling the interaction of multiple processes than would be available if each user had to solve all interprocess communications using simple variables, but more organization is possible. In a sense, semaphores are something like the ***goto*** statement in early programming languages; they can be used to solve a variety of problems, but they impose little structure on the solution and the results can be hard to understand without the aid of numerous comments. Just as there have been numerous control structures devised in sequential programs to reduce or even eliminate the need for ***goto*** statements, numerous specialized concurrent control structures have been developed which reduce or eliminate the need for semaphores.

Definition: The effective synchronization tools often used to realise mutual exclusion in more complex systems are semaphores. A semaphore ***S*** is an integer variable which can be accessed only through two standard atomic operations: ***wait and signal***. The definition of the wait and signal operation are:

```
wait(S): while S ≤ 0 do skip;
            S := S – 1;

signal(S): S := S + 1;
```

or in C language notation we can write it as:

```
        wait(s)
         {
           while (S<=0)
```

```
  {
    /*do nothing*/
  }
  S= S-1;
}

signal(S)
{
  S = S + 1;
}
```

It should be noted that the test *($S \le 0$)* and modification of the integer value of S which is S := S – 1 must be executed without interruption. In general, if one process modifies the integer value of S in the wait and signal operations, no other process can simultaneously modify that same S value. We briefly explain the usage of semaphores in the following example:

Consider two currently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has completed. This scheme can be implemented by letting P_1 and P_2 share a common semaphore *synch*, initialised to 0, and by inserting the statements:

S_1;
signal(synch);
in the process P_1 and the statements:
wait(synch);
S_2;
in the process P_2.

Since *synch* is initialised to 0, P_2 will execute S_2 only after P_1 has involved signal (synch), which is after S_1.

The disadvantage of the semaphore definition given above is that it requires *busy-waiting*, i.e., while a process is in its critical region, any either process it trying to enter its critical region must continuously loop in the entry code. It's clear that through busy-waiting, CPU cycles are wasted by which some other processes might use those productively.

To overcome busy-waiting, we modify the definition of the wait and signal operations. When a process executes the wait operation and finds that the semaphore value is not positive, the process *blocks* itself. The block operation places the process into a waiting state. Using a scheduler the CPU then can be allocated to other processes which are ready to run.

A process that is blocked, i.e., waiting on a semaphore S, should be restarted by the execution of a signal operation by some other processes, which changes its state from blocked to ready. To implement a semaphore under this condition, we define a semaphore as:

```
struct semaphore
 {
   int value;
   List *L;   //a list of processes
 }
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to this list. A signal operation removes one process from the list of waiting processes, and awakens it. The semaphore operation can be now defined as follows:

```
wait(S)
 {
   S.value = S.value -1;
   if (S.value <0)
```

```
    {
      add this process to S.L;
      block;
    }
  }
  signal(S)
  {
    S.value = S.value + 1;
    if (S.value <= 0)
    {
      remove a process P from S.L;
      wakeup(P);
    }
  }
```

The block operation suspends the process. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

One of the almost critical problem concerning implementing semaphore is the situation where two or more processes are waiting *indefinitely* for an event that can be only caused by one of the waiting processes: these processes are said to be *deadlocked*. To illustrate this, consider a system consisting of two processes P_1 and P_2, each accessing two semaphores S and Q, set to the value one:

```
P1              P2
wait(S);        wait(Q);
wait(Q);        wait(S);
...             ...
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose P_1 executes wait(S) and then P_2 executes wait(Q). When P_1 executes wait(Q), it must wait until P_2 executes signal(Q). It is no problem, P_2 executes wait(Q), then signal(Q). Similarly, when P_2 executes wait(S), it must wait until P_1 executes signal(S). Thus these signal operations cannot be carried out, P_1 and P_2 are deadlocked. It is clear, that a set of processes are in a deadlocked state, when every process in the set is waiting for an event that can only be caused by another process in the set.

3.5 CLASSICAL PROBLEMS IN CONCURRENT PROGRAMMING

In this section, we present a large class of concurrency control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

3.5.1 Producers/Consumers Problem

Producer – Consumer processes are common in operating systems. The problem definition is that, a producer (process) produces the information that is consumed by a consumer (process). For example, a compiler may produce assembly code, which is consumed by an assembler. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized. These problems can be solved either through unbounded buffer or bounded buffer.

- **With an unbounded buffer**

  The unbounded-buffer producer- consumer problem places no practical limit on the size of the buffer .The consumer may have to wait for new items, but the producer can always produce new items; there are always empty positions in the buffer.

- **With a bounded buffer**

  The bounded buffer producer problem assumes that there is a fixed buffer size. In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Shared Data

```
char item;              //could be any data type
char                  buffer[n];
semaphore full = 0;      //counting semaphore
semaphore empty = n;     //counting semaphore
semaphore mutex = 1;     //binary semaphore
char           nextp,nextc;
```

Producer Process

```
do
{
 produce an item in nextp
 wait (empty);
 wait (mutex);
 add nextp to buffer
 signal (mutex);
 signal (full);
}
 while (true)
```

Consumer Process

```
do
{
 wait( full );
 wait( mutex );
 remove an item from buffer to nextc
 signal( mutex );
 signal( empty );
 consume the item in nextc;
}
```

3.5.2 Readers and Writers Problem

The readers/writers problem is one of the classic synchronization problems. It is often used to compare and contrast synchronization mechanisms. It is also an eminently used practical problem. A common paradigm in concurrent applications is isolation of shared data such as a variable, buffer, or document and the control of access to that data. This problem has two types of clients accessing the shared data. The first type, referred to as readers, only wants to read the shared data. The second type, referred to as writers, may want to modify the shared data. There is also a designated central data server or controller. It enforces exclusive write semantics; if a writer is active then no other writer or reader can be active. The server can support clients that wish to both read and write. The readers and writers problem is useful for modeling processes which are competing for a limited shared resource. Let us understand it with the help of a practical example:

An airline reservation system consists of a huge database with many processes that read and write the data. Reading information from the database will not cause a problem since no data is changed. The problem lies in writing information to the database. If no constraints are put on access to the database, data may change at any moment. By the time a reading process displays the result of a request for information to the user, the actual data in the database may have changed. What if, for instance, a process reads the number of available seats on a flight, finds a value of one, and reports it to the customer? Before the customer has a chance to make their reservation, another process makes a reservation for another customer, changing the number of available seats to zero.

The following is the solution using semaphores:

Semaphores can be used to restrict access to the database under certain conditions. In this example, semaphores are used to prevent any writing processes from changing information in the database while other processes are reading from the database.

```
semaphore mutex = 1;    // Controls access to the reader count
semaphore db = 1;       // Controls access to the database
int reader_count; // The number of reading processes accessing the data

Reader()
{
  while (TRUE) {                  // loop forever
    down(&mutex);                 // gain access to reader_count
    reader_count = reader_count + 1;  // increment the reader_count
    if (reader_count == 1)
        down(&db);   //If this is the first process to read the database,
                              // a down on db is executed to prevent access to the
                              // database by a writing process
    up(&mutex);                       // allow other processes to access reader_count
    read_db();                      // read the database
    down(&mutex);                     // gain access to reader_count
    reader_count = reader_count - 1;      // decrement reader_count
    if (reader_count == 0)
        up(&db);                    // if there are no more processes reading from the
                              // database, allow writing process to access the data
    up(&mutex);                       // allow other processes to access
reader_countuse_data();
                              // use the data read from the database (non-critical)
}

Writer()
{
  while (TRUE) {                  // loop forever
    create_data();                    // create data to enter into database (non-critical)
    down(&db);                        // gain access to the database
    write_db();                     // write information to the database
    up(&db);                        // release exclusive access to the database
}
```

3.5.3 Dining Philosophers Problem

Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the centre of the table is a large bowl of rice. A philosopher needs two chopsticks to eat. Only 5 chop sticks are available and a chopstick is placed between each pair of philosophers. They agree that each will only use the chopstick to his immediate right and left. From time to time, a philosopher gets hungry and tries to grab the two chopsticks that are immediate left and right to him. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he finishes eating, he puts down both his chopsticks and starts thinking again.

Here's a solution for the problem which does not require a process to write another process's state, and gets equivalent parallelism.

```
#define N 5                                /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
```

```
#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N];

/* one per philosopher, all 0 */
void get_forks(int i) {
        state[i] = HUNGRY;
        while ( state[i] == HUNGRY ) {
                P(mutex);
                if ( state[i] == HUNGRY &&
                state[LEFT] != EATING &&
                state[RIGHT(i)] != EATING ) {
                state[i] = EATING;
                V(s[i]);
            }
        V(mutex);
        P(s[i]);
        }
 }

void put_forks(int i) {
        P(mutex);
        state[i]= THINKING;
        if ( state[LEFT(i)] == HUNGRY ) V(s[LEFT(i)]);
        if ( state[RIGHT(i)] == HUNGRY) V(s[RIGHT(i)]);
        V(mutex);
}
void philosopher(int process) {
        while(1) {
        think();
        get_forks(process);
        eat();
        put_forks();
        }
}
```

3.5.4 Sleeping Barber Problem

A barber shop consists of a waiting room with chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barber shop and all chairs are occupied, then the customer leaves the shop. if the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

The following is a sample solution for the sleeping barber problem.

```
# define CHAIRS 5               // chairs for waiting customers

typedef int semaphore;          // use this for imagination

semaphore customers = 0;        // number of customers waiting for service
semaphore barbers – 0;          // number of barbers waiting for customers
semaphore mutex = 1;            // for mutual exclusion
int waiting = 0;                //customers who are waiting for a haircut

void barber(void)
{
 while (TRUE) {
           down(&customers);       //go to sleep if no of customers are zero
           down(&mutex);           //acquire access to waiting
           waiting = waiting -1 ;  //decrement count of waiting customers
           up(&barbers);           //one barber is now ready for cut hair
           up(&mutex);             //release waiting
           cut_hair();             //this is out of critical region for hair cut
           }
}

void customer(void)
{
           down(&mutex);           //enter critical region
           if (waiting < CHAIRS)  //if there are no free chairs, leave
           {
           waiting = waiting +1;   //increment count of waiting customers
           up(&customers);         //wake up barber if necessary
           up(&mutex);             //release access to waiting
           down(&barbers);         //go to sleep if no of free barbers is zero
           get_haircut();          //be seated and be serviced
           } else
                     {
                     up (&mutex);    // shop is full: do no wait
                     }
}
```

Explanation:

- This problem is similar to various queuing situations
- The problem is to program the barber and the customers without getting into race conditions
  - Solution uses three semaphores:
    - *customers*; counts the waiting customers
    - *barbers*; the number of barbers (0 or 1)
    - *mutex*; used for mutual exclusion
    - also need a variable *waiting*; also counts the waiting customers; (reason; no way to read the current value of semaphore)
  - The barber executes the procedure *barber*, causing him to block on the semaphore *customers* (initially 0);
  - The barber then goes to sleep;
  - When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region;
  - If another customer enters, shortly thereafter, the second one will not be able to do anything until the first one has released *mutex;*
  - The customer then checks to see if the number of waiting customers is less than the number of chairs;
  - If not, he releases *mutex* and leaves without a haircut;
  - If there is an available chair, the customer increments the integer variable, *waiting;*
  - Then he does an **up** on the semaphore *customers;*
  - When the customer releases *mutex*, the barber begins the haircut.

3.6 LOCKS

Locks are another synchronization mechanism. A lock has got two atomic operations (similar to semaphore) to provide mutual exclusion. These two operations are Acquire and Release. A process will acquire a lock before accessing a shared variable, and later it will be released. A process locking a variable will run the following code:

```
Lock-Acquire();
critical section
Lock-Release();
```

The difference between a lock and a semaphore is that a lock is released only by the process that have acquired it earlier. As we discussed above any process can increment the value of the semaphore. To implement locks, here are some things you should keep in mind:

- To make Acquire () and Release () atomic
- Build a wait mechanism.
- Making sure that only the process that acquires the lock will release the lock.

3.7 MONITORS AND CONDITION VARIABLES

When you are using semaphores and locks you must be very careful, because a simple misspelling may lead that the system ends up in a deadlock. Monitors are written to make synchronization easier and correctly. Monitors are some procedures, variables, and data structures grouped together in a package.

An early proposal for organising the operations required to establish mutual exclusion is the explicit critical section statement. In such a statement, usually proposed in the form “critical x do y”, where “x” is the name of a semaphore and “y” is a statement, the actual wait and signal operations used to ensure mutual exclusion were implicit and automatically balanced. This allowed the compiler to trivially check for the most obvious errors in concurrent programming, those where a wait or signal operation was accidentally forgotten. The problem with this statement is that it is not adequate for many critical sections.

A common observation about critical sections is that many of the procedures for manipulating shared abstract data types such as files have critical sections making up their entire bodies. Such abstract data types have come to be known as monitors, a term coined by C. A. R. Hoare. Hoare proposed a programming notation where the critical sections and semaphores implicit in the use of a monitor were all implicit. All that this notation requires is that the programmer encloses the declarations of the procedures and the representation of the data type in a monitor block; the compiler supplies the semaphores and the wait and signal operations that this implies. Using Hoare's suggested notation, shared counters might be implemented as shown below:

```
var value: integer;
procedure increment;
begin
        value := value + 1;
end { increment };
end { counter };

var i, j: counter;
```

Calls to procedures within the body of a monitor are done using record notation; thus, to increment one of the counters declared in above example, one would call “i.increment”. This call would implicitly do a wait operation on the semaphore implicitly associated with “i”, then execute the body of the “increment” procedure before doing a signal operation on the semaphore. Note that the call to "i.increment" implicitly passes a specific instance of the monitor as a parameter to the "increment" procedure, and that fields of this instance become global variables to the body of the procedure, as if there was an implicit "with" statement.

There are a number of problems with monitors which have been ignored in the above example. For example, consider the problem of assigning a meaning to a call from within one monitor procedure to a procedure within another monitor. This can easily lead to a deadlock. For example, when procedures within two different monitors each calling the other. It has sometimes been proposed that such calls should never be allowed, but they are sometimes useful! We will study more on deadlocks in the next units of this course.

The most important problem with monitors is that of waiting for resources when they are not available. For example, consider implementing a queue monitor with internal procedures for the enqueue and dequeue operations. When the queue empties, a call to dequeue must wait, but this wait must not block further entries to the monitor through the enqueue procedure. In effect, there must be a way for a process to temporarily step outside of the monitor, releasing mutual exclusion while it waits for some other process to enter the monitor and do some needed action.

Hoare's suggested solution to this problem involves the introduction of condition variables which may be local to a monitor, along with the operations wait and signal. Essentially, if s is the monitor semaphore, and c is a semaphore representing a condition variable, "wait c" is equivalent to "signal(s); wait(c); wait(s)" and "signal c" is equivalent to "signal(c)". The details of Hoare's wait and signal operations were somewhat more complex than is shown here because the waiting process was given priority over other processes trying to enter the monitor, and condition variables had no memory; repeated signalling of a condition had no effect and signaling a condition on which no process was waiting had no effect. Following is an example monitor:

```
monitor synch
    integer i;
    condition c;
    procedure producer(x);
    .
    .
    end;
    procedure consumer(x);
    .
    .
    end;
  end monitor;
```

There is only one process that can enter a monitor, therefore every monitor has its own waiting list with process waiting to enter the monitor.

Let us see the dining philosopher's which was explained in the above section with semaphores, can be re-written using the monitors as:

Example: Solution to the Dining Philosophers Problem using Monitors

```
monitor dining-philosophers
{
  enum state {thinking, hungry, eating};
  state state[5];
  condition self[5];

  void pickup (int i)
  {
    state[i] = hungry;
    test(i);
    if (state[i] != eating)
    self[i].wait;
  }

    void putdown (int i)
  {
    state[i] = thinking;
    test(i+4 % 5);
    test(i+1 % 5);
  }

  void test (int k)
  {
    if ((state[k+4 % 5] != eating) && (state[k]==hungry)
                          && state[k+1 % 5] != eating))
    {
      state[k] = eating;
      self[k].signal;
    }
  }

  init
  {
    for (int i = 0; i< 5; i++)
      state[i] = thinking;
  }
}
```

Condition Variables

If a process cannot enter the monitor it must block itself. This operation is provided by the condition variables. Like locks and semaphores, the condition has got a wait and a signal function. But it also has the broadcast signal. Implementation of condition variables is part of a synch.h; it is your job to implement it. Wait (), Signal () and Broadcast () have the following semantics:

- Wait() releases the lock, gives up the CPU until signaled and then re-acquire the lock.
- Signal() wakes up a thread if there are any waiting on the condition variable.
- Broadcast() wakes up all threads waiting on the condition.

When you implement the condition variable, you must have a queue for the processes waiting on the condition variable.

☞ Check Your Progress 1

1) What are race conditions? How race conditions occur in Operating Systems?

…………………………………………………………………………………………

…………………………………………………………………………………………

2) What is a critical section? Explain.

…………………………………………………………………………………………

…………………………………………………………………………………………

3) Provide the solution to a classical synchronization problem namely "cigarette smoker's problem". The problem is defined as follows:

There are four processes in this problem: three smoker processes and an agent process. Each of the smoker processes will make a cigarette and smoke it. To make a cigarette requires tobacco, paper, and matches. Each smoker process *has one of the three items. i.e., one process has tobacco, another has paper, and a third has matches. The agent has an infinite supply of all three. The agent places two of the three items on the table, and the smoker that has the third item makes the cigarette.*

……………………………………………………………………………………

……………………………………………………………………………………

3.8 SUMMARY

Interprocess communication provides a mechanism to allow process to communicate with other processes. Interprocess communication system is best provided by a message passing system. Message systems can be defined in many different ways. If there are a collection of cooperating sequential processes that share some data, mutual exclusion must be provided. There are different methods for achieving the mutual exclusion. Different algorithms are available for solving the critical section problem which we have discussion in this unit. The bakery algorithm is used for solving the *n* process critical section problem.

Interprocess synchronization provides the processes to synchronize their activities. Semaphores can be used to solve synchronization problems. Semaphore can only be accessed through two atomic operations and can be implemented efficiently. The two operations are *wait* and *signal*.

There are a number of classical synchronization problems which we have discussed in this unit (such as producer- consumer problem, readers – writers problem and d dining – philosophers problem). These problems are important mainly because they are examples for a large class of concurrency-control problems. In the next unit we will study an important aspect called as "Deadlocks"

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one "customer" thread at a time should be allowed to examine and update the shared variable.

   Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

2) The most synchronization problem confronting cooperating processes, is to control the access between the shared resources. Suppose two processes share access to a file and at least one of these processes can modify the data in this shared area of memory. That part of the code of each program, where one process is reading from or writing to a shared memory area, is a *critical section* of code; because we must ensure that only one process execute a critical section of code at a time. The critical section problem is to design a protocol that the processes can use to coordinate their activities when one wants to enter its critical section of code.

3) This seems like a fairly easy solution. The three smoker processes will make a cigarette and smoke it. If they can't make a cigarette, then they will go to sleep.

The agent process will place two items on the table, and wake up the appropriate smoker, and then go to sleep. All semaphores except lock are initialised to 0. Lock is initialised to 1, and is a mutex variable.

Here's the code for the ***agent process***.

```
do forever {
    P( lock );
    randNum = rand(1,3); // Pick a random number from 1-3
   if (randNum == 1) {
           // Put tobacco on table
          // Put paper on table
          V(smoker_match);  // Wake up smoker with match
           }
   else if (randNum == 2) {
          // Put tobacco on table
           // Put match on table
   V(smoker_paper);  // Wake up smoker with paper
          }
           else {
                  // Put match on table
                  // Put paper on table
                 V(smoker_tobacco);
          } // Wake up smoker with tobacco

    V(lock);
   P(agent);  //  Agent sleeps

   }  // end forever loop
```

The following is the code for one of the smokers. The others are analogous.

```
do forever {
      P(smoker_tobacco);  // Sleep right away
     P(lock);
     // Pick up match
      // Pick up paper
     V(agent);
      V(lock);
      // Smoke
      }
```

3.10 FURTHER READINGS

1) Milenkovic, Milan, "*Operating Systems Concepts and Design*", McGraw-Hill, 2nd Edition.

2) Tanenbaum, Andrew, *Modern Operating Systems*, Prentice-Hall.

3) Silberschatz, Abraham and Galvin Peter, "*Operating System Concepts*", Addison-Wesley.

4) D.M. Dhamdhere, "*Operating Systems A concept-based Approach*", Tata McGraw-Hill.

5) William Stalling, "*Operating System*", Prentice Hall.

6) Deitel, Harvey M. , "*An introduction to Operating System*", 4th ed., Addison-Wesley.

UNIT 4 DEADLOCKS

Structure **Page Nos**

4.0 INTRODUCTION

In a computer system, we have a finite number of *resources* to be distributed among a number of competing processes. These system resources are classified in several types which may be either physical or logical. Examples of physical resources are Printers, Tape drivers, Memory space, and CPU cycles. Examples of logical resources are Files, Semaphores and Monitors. Each resource type can have some identical instances.

A process must request a resource before using it and release the resource after using it. It is clear that the number of resources requested cannot exceed the total number of resources available in the system.

In a normal operation, a process may utilise a resource only in the following sequence:

- *Request*: if the request cannot be immediately granted, then the requesting process must wait until it can get the resource.
- *Use:* the requesting process can operate on the resource.
- *Release:* the process releases the resource after using it.

Examples for request and release of system resources are:

- Request and release the device,
- Opening and closing file,
- Allocating and freeing the memory.

The operating system is responsible for making sure that the requesting process has been allocated the resource. A system table indicates if each resource is free or allocated, and if allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

In some cases, several processes may compete for a fixed number of resources. A process requests resources and if the resources are not available at that time, it enters a wait state. It may happen that it will never gain access to the resources, since those resources are being held by other waiting processes.

For example, assume a system with one tape drive and one plotter. Process P1 requests the tape drive and process P2 requests the plotter. Both requests are granted. Now PI requests the plotter (without giving up the tape drive) and P2 requests the tape drive (without giving up the plotter). Neither request can be granted so both processes enter a situation called the **deadlock** situation.

A **deadlock** is a situation where a group of processes is permanently blocked as a result of each process having acquired a set of resources needed for its completion and having to wait for the release of the remaining resources held by others thus making it impossible for any of the deadlocked processes to proceed.

In the earlier units, we have gone through the concept of process and the need for the interprocess communication and synchronization. In this unit we will study about the deadlocks, its characterisation, deadlock avoidance and its recovery.

4.1 OBJECTIVES

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

- define a deadlock;
- understand the conditions for a deadlock;
- know the ways of avoiding deadlocks, and
- describe the ways to recover from the deadlock situation.

4.2 DEADLOCKS

Before studying about deadlocks, let us look at the various types of resources. There are two types of resources namely: Pre-emptable and Non-pre-emptable Resources.

- **Pre-emptable resources**: This resource can be taken away from the process with no ill effects. Memory is an example of a pre-emptable resource.
- **Non-Preemptable resource**: This resource cannot be taken away from the process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment.

Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with. Let us see how a deadlock occurs.

Definition: A set of processes is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources and none of them can be awakened. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant.

Let us understand the deadlock situation with the help of examples.

Example 1: The simplest example of deadlock is where process 1 has been allocated a non-shareable resource *A*, say, a tap drive, and process 2 has been allocated a non-sharable resource *B*, say, a printer. Now, if it turns out that process 1 needs resource *B* (printer) to proceed and process 2 needs resource *A* (the tape drive) to proceed and these are the only two processes in the system, each has blocked the other and all useful work in the system stops. This situation is termed as deadlock.

The system is in deadlock state because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.

Example 2: Consider a system with three disk drives. Suppose there are three processes, each is holding one of these three disk drives. If each process now requests another disk drive, three processes will be in a deadlock state, because each process is waiting for the event "disk drive is released", which can only be caused by one of the other waiting processes. Deadlock state involves processes competing not only for the same resource type, but also for different resource types.

Deadlocks occur most commonly in multitasking and client/server environments and are also known as a "Deadly Embrace". Ideally, the programs that are deadlocked or the operating system should resolve the deadlock, but this doesn't always happen.

From the above examples, we have understood the concept of deadlocks. In the examples we were given some instances, but we will study the necessary conditions for a deadlock to occur, in the next section.

4.3 CHARACTERISATION OF A DEADLOCK

Coffman (1971) identified **four necessary conditions** that must hold simultaneously for a deadlock to occur.

4.3.1 Mutual Exclusion Condition

The resources involved are non-shareable. At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

4.3.2 Hold and Wait Condition

In this condition, a requesting process already holds resources and waiting for the requested resources. A process, holding a resource allocated to it waits for an additional resource(s) that is/are currently being held by other processes.

4.3.3 No-Preemptive Condition

Resources already allocated to a process cannot be preempted. Resources cannot be removed forcibly from the processes. After completion, they will be released voluntarily by the process holding it.

4.3.4 Circular Wait Condition

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

Let us understand this by a common example. Consider the traffic deadlock shown in the *Figure 1.*

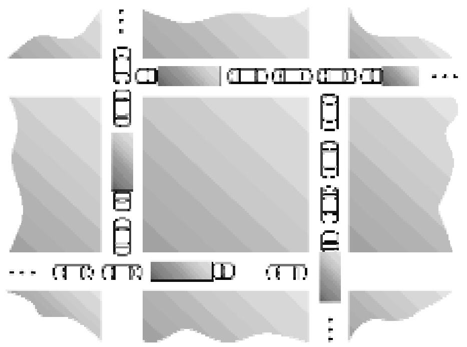


Figure 1: Traffic Deadlock

Consider each section of the street as a resource. In this situation:

- Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
- Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
- Non-preemptive condition applies, since a section of the street that is occupied by a vehicle cannot be taken away from it.
- Circular wait condition applies, since each vehicle is waiting for the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of the street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.
It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread, that is, each thread has access to the resources held by the process.

4.4 A RESOURCE ALLOCATION GRAPH

The idea behind the resource allocation graph is to have a graph which has two different types of nodes, the process nodes and resource nodes (process represented by circles, resource node represented by rectangles). For different instances of a resource, there is a dot in the resource node rectangle. For example, if there are two identical printers, the printer resource might have two dots to indicate that we don't really care which is used, as long as we acquire the resource.

The edges among these nodes represent resource allocation and release. Edges are directed, and if the edge goes from resource to process node that means the process has acquired the resource. If the edge goes from process node to resource node that means the process has requested the resource.

We can use these graphs to determine if a deadline has occurred or may occur. If for example, all resources have only one instance (all resource node rectangles have one dot) and the graph is circular, then a deadlock *has* occurred. If on the other hand some resources have several instances, then a deadlock *may* occur. If the graph is not circular, a deadlock cannot occur (the *circular wait* condition wouldn't be satisfied). The following are the tips which will help you to check the graph easily to predict the presence of cycles.

- If no cycle exists in the resource allocation graph, there is no deadlock.
- If there is a cycle in the graph and each resource has only one instance, then there is a deadlock. In this case, a cycle is a necessary and sufficient condition for deadlock.
- If there is a cycle in the graph, and each resource has more than one instance, there may or may not be a deadlock. (A cycle may be broken if some process outside the cycle has a resource instance that can break the cycle). Therefore, a cycle in the resource allocation graph is a necessary but not sufficient condition for deadlock, when multiple resource instances are considered.

Example:

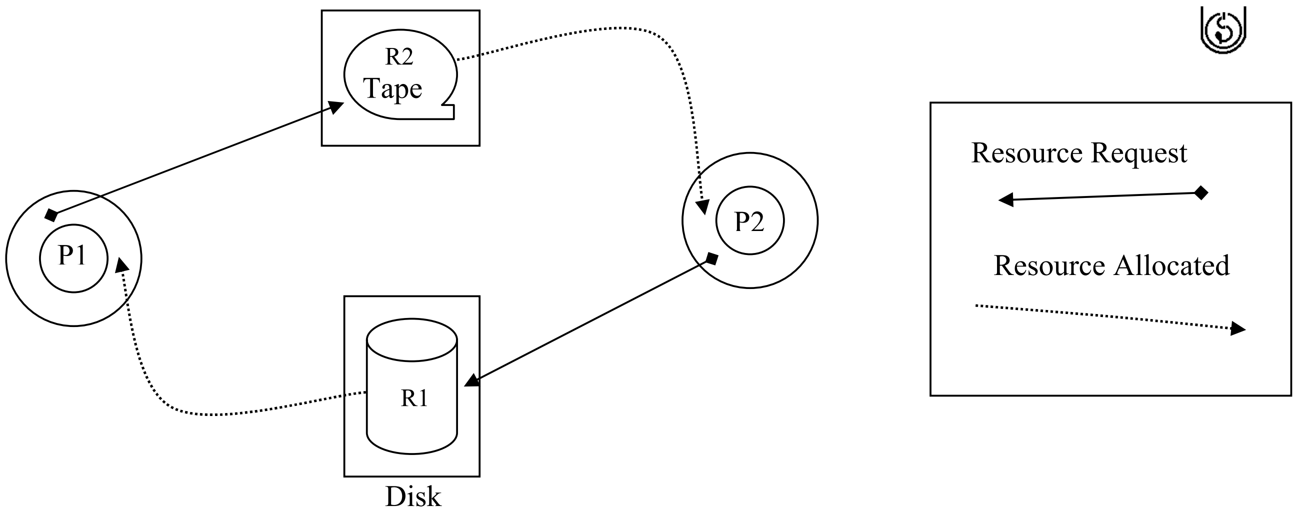


Figure 2: Resource Allocation Graph Showing Deadlock

The above graph shown in *Figure 2* has a cycle and is in Deadlock.

$R1 \longrightarrow P1 \quad P1 \longrightarrow R2$
$R2 \longrightarrow P2 \quad P2 \longrightarrow R1$

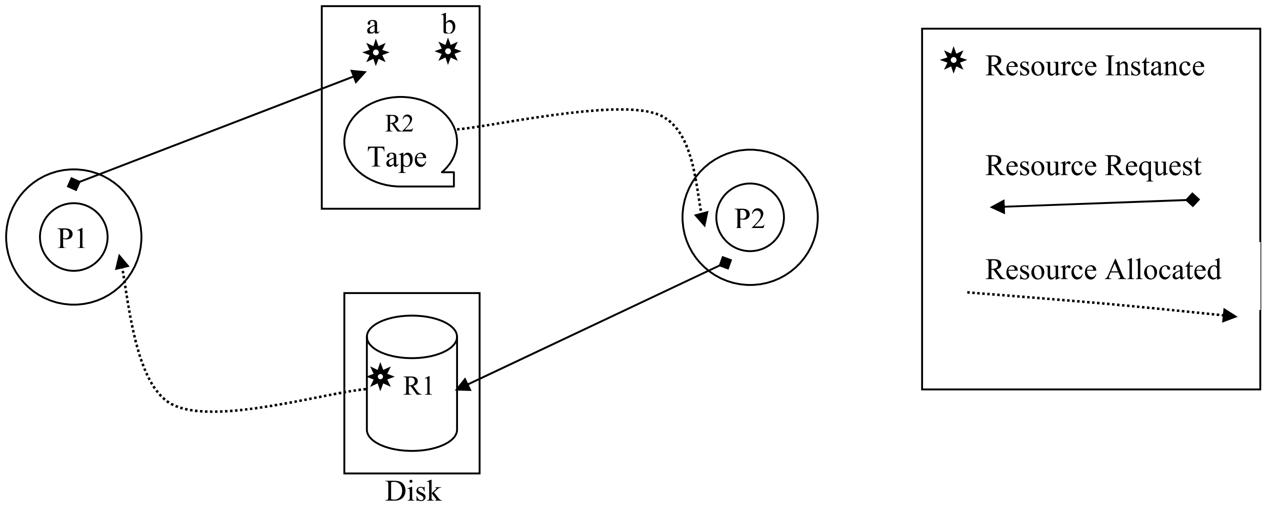


Figure 3: Resource Allocation Graph having a cycle and not in a Deadlock

The above graph shown in *Figure 3* has a cycle and is not in Deadlock.

(Resource 1 has one instance shown by a star)

(Resource 2 has two instances a and b, shown as two stars)

R1 $\longrightarrow$ P1 P1 $\longrightarrow$ R2 **(a)**

R2 **(b)** $\longrightarrow$ P2 P2 $\longrightarrow$ R1

If P1 finishes, P2 can get R1 and finish, so there is no Deadlock.

4.5 DEALING WITH DEADLOCK SITUATIONS

There are possible strategies to deal with deadlocks. They are:

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

Let's examine each strategy one by one to evaluate their respective strengths and weaknesses.

4.5.1 Deadlock Prevention

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions. Let us study Havender's algorithm.

Havender's Algorithm

Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-shareable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of "Hold and Wait" Condition

There are two possibilities for the elimination of the second condition. The first alternative is that a process request be granted all the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources.

For example, a program requiring ten tap drives must request and receive all ten drives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation), since not all the required resources may become available at once.

Elimination of "No-preemption" Condition

The nonpreemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated, to relinquish all of its currently held resources, so that other processes may use them to finish their needs. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed, while the second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources, the process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.

High Cost

When a process releases resources, the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of "Circular Wait" Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing all processes to request the resources in order (increasing or decreasing). This strategy imposes a total ordering of all resource types, and requires that each process requests resources in a numerical order of enumeration. With this rule, the resource allocation graph can never have a cycle.
For example, provide a global numbering of all the resources, as shown in the given *Table 1*:

Table 1: Numbering the resources

Number	**Resource**
1	Floppy drive
2	Printer
3	Plotter
4	Tape Drive
5	CD Drive

Now we will see the rule for this:

Rule: Processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

This strategy, if adopted, may result in low resource utilisation and in some cases starvation is possible too.

4.5.2 Deadlock Avoidance

This approach to the deadlock problem anticipates a deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and act accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. The most famous deadlock avoidance algorithm, from Dijkstra [1965], is the Banker's algorithm. It is named as Banker's algorithm because the process is analogous to that used by a banker in deciding if a loan can be safely made a not.

The Banker's Algorithm is based on the banking system, which never allocates its available cash in such a manner that it can no longer satisfy the needs of all its customers. Here we must have the advance knowledge of the maximum possible claims for each process, which is limited by the resource availability. During the run of the system we should keep monitoring the resource allocation status to ensure that no circular wait condition can exist.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. The following are the features that are to be considered for avoidance of the deadlock s per the Banker's Algorithms.

- Each process declares maximum number of resources of each type that it may need.
- Keep the system in a safe state in which we can allocate resources to each process in some order and avoid deadlock.

- Check for the safe state by finding a safe sequence: <P1, P2, ..., Pn> where resources that Pi needs can be satisfied by available resources plus resources held by Pj where j < i.
- Resource allocation graph algorithm uses claim edges to check for a safe state.

The resource allocation state is now defined by the number of available and allocated resources, and the maximum demands of the processes. Subsequently the system can be in either of the following states:

- **Safe state:** Such a state occur when the system can allocate resources to each process (up to its maximum) in some order and avoid a deadlock. This state will be characterised by a safe sequence. It must be mentioned here that we should not falsely conclude that all unsafe states are deadlocked although it may eventually lead to a deadlock.
- **Unsafe State:** If the system did not follow the safe sequence of resource allocation from the beginning and it is now in a situation, which may lead to a deadlock, then it is in an unsafe state.
- **Deadlock State:** If the system has some circular wait condition existing for some processes, then it is in deadlock state.

Let us study this concept with the help of an example as shown below:

Consider an analogy in which 4 processes (P1, P2, P3 and P4) can be compared with the customers in a bank, resources such as printers etc. as cash available in the bank and the Operating system as the Banker.

Table 2

Processes	Resources used	Maximum resources
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Let us assume that total available resources = 10

In the above table, we see four processes, each of which has been granted a number of maximum resources that can be used. The Operating system reserved only 10 resources rather than 22 units to service them. At a certain moment, the situation becomes:

Table 3

Processes	Resources used	Maximum resources
P1	1	6
P2	1	5
P3	2	4
P4	4	7

Available resources = 2

Safe State: The key to a state being safe is that there is at least one way for all users to finish. In other words the state of *Table 2* is safe because with 2 units left, the operating system can delay any request except P3, thus letting P3 finish and release all four resources. With four units in hand, the Operating system can let either P4 or P2 have the necessary units and so on.

Unsafe State: Consider what would happen if a request from P2 for one more unit was granted in *Table 3*. We would have following situation as shown in *Table 4*.

Table 4

Processes	Resources used	Maximum resources
P1	1	6
P2	2	5
P3	2	4
P4	4	7

Available resource = 1

This is an unsafe state.

If all the processes request for their maximum resources respectively, then the operating system could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence of a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus used to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it is postponed until later. Haberman [1969] has shown that executing of the algorithm has a complexity proportional to N^2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

Limitations of the Banker's Algorithm

There are some problems with the Banker's algorithm as given below:

- It is time consuming to execute on the operation of every resource.
- If the claim information is not accurate, system resources may be underutilised.
- Another difficulty can occur when a system is heavily loaded. Lauesen states that in this situation "so many resources are granted away that very few safe sequences remain, and as a consequence, the jobs will be executed sequentially". Therefore, the Banker's algorithm is referred to as the "Most Liberal" granting policy; that is, it gives away everything that it can without regard to the consequences.
- New processes arriving may cause a problem.
    - The process's claim must be less than the total number of units of the resource in the system. If not, the process is not accepted by the manager.
    - Since the state without the new process is safe, so is the state with the new process. Just use the order you had originally and put the new process at the end.
    - Ensuring fairness (starvation freedom) needs a little more work, but isn't too hard either (once every hour stop taking new processes until all current processes finish).
- A resource becoming unavailable (e.g., a tape drive breaking), can result in an unsafe state.

4.5.3 Deadlock Detection and Recovery

Detection of deadlocks is the most practical policy, which being both liberal and cost efficient, most operating systems deploy. To detect a deadlock, we must go about in a recursive manner and simulate the most favoured execution of each unblocked process.

- An unblocked process may acquire all the needed resources and will execute.

- It will then release all the acquired resources and remain dormant thereafter.
- The now released resources may wake up some previously blocked process.
- Continue the above steps as long as possible.
- If any blocked processes remain, they are **deadlocked.**

Recovery from Deadlock

Recovery by process termination

In this approach we terminate deadlocked processes in a systematic way taking into account their priorities. The moment, enough processes are terminated to recover from deadlock, we stop the process terminations. Though the policy is simple, there are some problems associated with it.

Consider the scenario where a process is in the state of updating a data file and it is terminated. The file may be left in an incorrect state by the unexpected termination of the updating process. Further, processes should be terminated based on some criterion/policy. Some of the criteria may be as follows:

- Priority of a process
- CPU time used and expected usage before completion
- Number and type of resources being used (can they be preempted easily?)
- Number of resources needed for completion
- Number of processes needed to be terminated
- Are the processes interactive or batch?

Recovery by Checkpointing and Rollback (Resource preemption)

Some systems facilitate deadlock recovery by implementing *checkpointing and rollback.* Checkpointing is saving *enough state of a process* so that the process can be restarted at the point in the computation where the checkpoint was taken. Autosaving file edits are a form of checkpointing. Checkpointing costs depend on the underlying algorithm. Very simple algorithms (like linear primality testing) can be checkpointed with a few words of data. More complicated processes may have to save all the process state and memory.

If a deadlock is detected, one or more processes are restarted from their last checkpoint. Restarting a process from a checkpoint is called *rollback.* It is done with the expectation that the resource requests will not interleave again to produce deadlock.

Deadlock recovery is generally used when deadlocks are rare, and the cost of recovery (process termination or rollback) is low.

Process checkpointing can also be used to improve reliability (long running computations), assist in process migration, or reduce startup costs.

☞ Check Your Progress 1

1) What is a deadlock and what are the four conditions that will create the deadlock situation?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2) How can deadlock be avoided? Explain with the help of an example.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

4.6 SUMMARY

A deadlock occurs a process has some resource and is waiting to acquire another resource, while that resource is being held by some process that is waiting to acquire the resource that is being held by the first process.

A deadlock needs four conditions to occur: Mutual Exclusion, Hold and Wait, Non-Preemption and Circular Waiting.

We can handle deadlocks in three major ways: We can prevent them, handle them when we detect them, or simply ignore the whole deadlock issue altogether.

4.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set. A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- *Mutual Exclusion*: At least one resource must be held in a non-shareable mode; that is, only one process at a time can use the resource. If another process requests the resource, the requesting process must be delayed until the resource has been released.
- *Hold and Wait*: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- *No Preemption*: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- *Circular Wait*: A set {P0, P1, P2, …, Pn} of waiting processes must exist such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

2) Deadlock avoidance deals with processes that declare before execution how many resources they may need during their execution. Given several processes and resources, if we can allocate the resources in some order as to prevent a deadlock, the system is said to be in a safe state. If a deadlock is possible, the system is said to be in an unsafe state. The idea of avoiding a deadlock is to simply not allow the system to enter an unsafe state which may cause a deadlock. We can define what makes an unsafe state.

For example, consider a system with 12 tape drives and three processes: P0, P1 and P2. P0 may require 10 tape drives during execution, P1 may require 4, and P2 may require up to 9. Suppose that at some moment in time, P0 is holding on to 5 tape drives, P1 holds 2 and P2 holds 2 tape drives. The system is said to be in a safe state, since there is a safe sequence that avoids the deadlock. This sequence implies that P1 can instantly get all of its needed resources (there are 12 total tape drives, and P1 already has 2, so the maximum it needs is 2, which it can get since there are 3 free tape drives). Once it finishes executing, it releases all 4 resources, which makes for 5 free tape drives, at which point, P0 can execute (since the maximum it needs is 10), after it finishes, P2 can proceed since the maximum it needs is 9.

Now, here’s an unsafe sequence: Let’s say at some point of time P2 requests one more resource to make its holding resources 3. Now the system is in an unsafe state, since only P1 can be allocated resources, and after it returns, it will only leave 4 free resources, which is not enough for either P0 or P2, so the system enters a deadlock.

4.8 FURTHER READINGS

1) Madnick and Donovan, *Operating systems – Concepts and Design,* McGrawHill Intl. Education.

2) Milan Milenkovic, *Operating Systems, Concepts and Design*, TMGH, 2000.

3) D.M. Dhamdhere, *Operating Systems, A concept-based approach*, TMGH, 2002.

4) Abraham Silberschatz and James L, *Operating System Concepts*.

Peterson, Addition Wesely Publishing Company, New Delhi.

5) Harvay M. Deital, *Introduction to Operating systems,* Addition Wesely Publishing Company, New Delhi.

6) Andrew S. Tanenbaum, *Operating System Design and Implementation,* PHI, New Delhi.

UNIT 1 MEMORY MANAGEMENT

Structure **Page Nos.**

1.0 INTRODUCTION

In Block 1 we have studied about introductory concepts of the OS, process management and deadlocks. In this unit, we will go through another important function of the Operating System – the memory management.

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each location with its own address. Interaction is achieved through a sequence of reads/writes of specific memory address. The CPU fetches from the program from the hard disk and stores in memory. If a program is to be executed, it must be mapped to absolute addresses and loaded into memory.

In a multiprogramming environment, in order to improve both the CPU utilisation and the speed of the computer's response, several processes must be kept in memory. There are many different algorithms depending on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The Operating System is responsible for the following activities in connection with memory management:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

In the multiprogramming environment operating system dynamically allocates memory to multiple processes. Thus memory plays a significant role in the important aspects of computer system like performance, S/W support, reliability and stability.

Memory can be broadly classified into two categories–the primary memory (like cache and RAM) and the secondary memory (like magnetic tape, disk etc.). The memory is a resource that needs effective and efficient management. The part of OS that perform this vital task of memory management is known **as memory manager**. In multiprogramming system, as available memory is shared among number of processes, so the allocation speed and the efficient memory utilisation (in terms of minimal overheads and reuse/relocation of released memory block) are of prime concern. Protection is difficult to achieve with relocation requirement, as location of process and absolute address in memory is unpredictable. But at run-time, it can be done. Fortunately, we have mechanisms supporting protection like processor (hardware) support that is able to abort the instructions violating protection and trying to interrupt other processes.

This unit collectively depicts such memory management related responsibilities in detail by the OS. Further we will discuss, the basic approaches of allocation are of two types:

Contiguous Memory Allocation: Each programs data and instructions are allocated a single contiguous space in memory.

Non-Contiguous Memory Allocation: Each programs data and instructions are allocated memory space that is not continuous. This unit focuses on contiguous memory allocation scheme.

1.1 OBJECTIVES

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

- describe the various activities handled by the operating system while performing the memory management function;
- to allocate memory to the processes when they need it;
- reallocation when processes are terminated;
- logical and physical memory organisation;
- memory protection against unauthorised access and sharing;
- to manage swapping between main memory and disk in case main storage is small to hold all processes;
- to summarise the principles of memory management as applied to paging and segmentation;
- compare and contrast paging and segmentation techniques, and
- analyse the various memory portioning/partitioning techniques including overlays, swapping, and placement and replacement policies.

1.2 OVERLAYS AND SWAPPING

Usually, programs reside on a disk in the form of executable files and for their execution they must be brought into memory and must be placed within a process. Such programs form the ready queue. In general scenario, processes are fetched from ready queue, loaded into memory and then executed. During these stages, addresses may be represented in different ways like in source code addresses or in symbolic form (ex. LABEL). Compiler will bind this symbolic address to relocatable addresses (for example, 16 bytes from base address or start of module). The linkage editor will bind these relocatable addresses to absolute addresses. Before we learn a program in memory we must bind the memory addresses that the program is going to use. Binding is basically assigning which address the code and data are going to occupy. You can bind at compile-time, load-time or execution time.

Compile-time: If memory location is known a priori, absolute code can be generated.

Load-time: If it is not known, it must generate relocatable at complete time.

Execution-time: Binding is delayed until run-time; process can be moved during its execution. We need H/W support for address maps (base and limit registers).

For better memory utilisation all modules can be kept on disk in a relocatable format and only main program is loaded into memory and executed. Only on need the other routines are called, loaded and address is updated. Such scheme is called *dynamic loading*, which is user's responsibility rather than OS.But Operating System provides library routines to implement dynamic loading.

In the above discussion we have seen that entire program and its related data is loaded in physical memory for execution. But what if process is larger than the amount of memory allocated to it? We can overcome this problem by adopting a technique called as *Overlays*. Like dynamic loading, overlays can also be implemented by users without OS support. The entire program or application is divided into instructions and data sets such that when one instruction set is needed it is loaded in memory and after its execution is over, the space is released. As and when requirement for other instruction arises it is loaded into space that was released previously by the instructions that are no longer needed. Such instructions can be called as overlays, which are loaded and unloaded by the program.

Definition: An overlay is a part of an application, which has been loaded at same origin where previously some other part(s) of the program was residing.

A program based on overlay scheme mainly consists of following:

- A "root" piece which is always memory resident
- Set of overlays.

Overlay gives the program a way to extend limited main storage. An important aspect related to overlays identification in program is concept of mutual exclusion i.e., routines which do not invoke each other and are not loaded in memory simultaneously.

For example, suppose total available memory is 140K. Consider a program with four subroutines named as: ***Read ( ), Function1( ), Function2( )*** and ***Display( ).*** First, *Read* is invoked that reads a set of data. Based on this data set values, conditionally either one of routine *Function1* or *Function2* is called. And then *Display* is called to output results. Here, *Function1* and *Function2* are mutually exclusive and are not required simultaneously in memory. The memory requirement can be shown as in *Figure 1*:

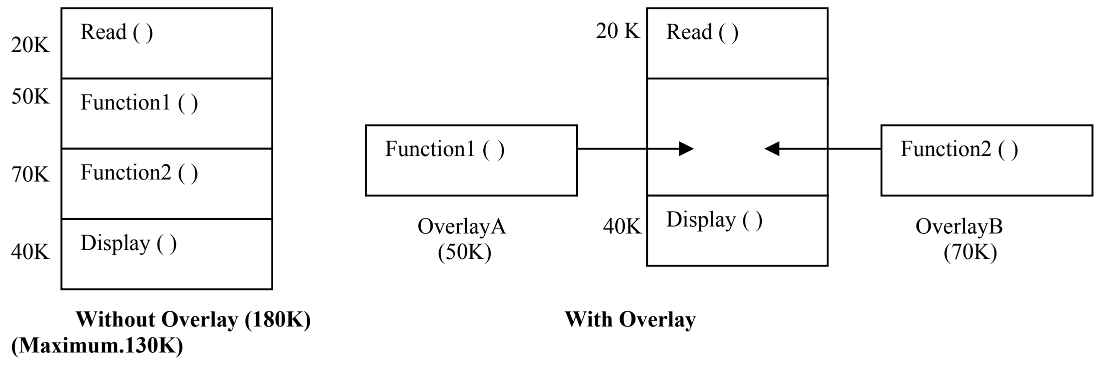


Figure 1: Example of overlay

Without the overlay it requires 180 K of memory and with the overlay support memory requirement is 130K. Overlay manager/driver is responsible for loading and unloading on overlay segment as per requirement. But this scheme suffers from following limitations:

- Require careful and time-consuming planning.
- Programmer is responsible for organising overlay structure of program with the help of file structures etc. and also to ensure that piece of code is already loaded when it is called.
- Operating System provides the facility to load files into overlay region.

Swapping

Swapping is an approach for memory management by bringing each process in entirety, running it and then putting it back on the disk, so that another program may be loaded into that space. Swapping is a technique that lets you use a disk file as an extension of memory. Lower priority user processes are swapped to backing store (disk) when they are waiting for I/O or some other event like arrival of higher priority processes. This is *Rollout Swapping*. Swapping the process back into store when some event occurs or when needed (may be in a different partition) is known as *Roll-in swapping*. *Figure 2* depicts technique of swapping:

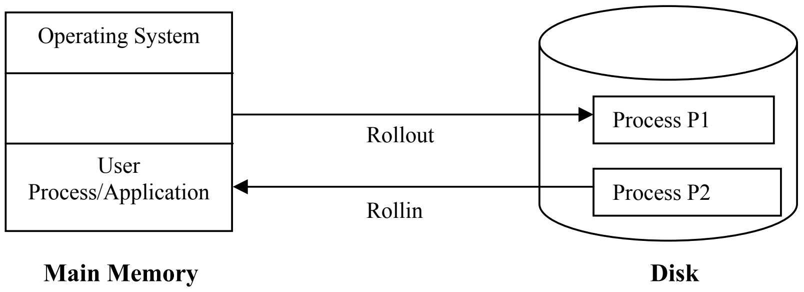


Figure 2: Swapping

Major benefits of using swapping are:

- Allows higher degree of multiprogramming.
- Allows dynamic relocation, i.e., if address binding at execution time is being used we can swap in different location else in case of compile and load time bindings processes have to be moved to same location only.
- Better memory utilisation.
- Less wastage of CPU time on compaction, and
- Can easily be applied on priority-based scheduling algorithms to improve performance.

Though swapping has these benefits but it has few limitations also like entire program must be resident in store when it is executing. Also processes with changing memory requirements will need to issue system calls for requesting and releasing memory. It is necessary to know exactly how much memory a user process is using and also that it is blocked or waiting for I/O.

If we assume a data transfer rate of 1 megabyte/sec and access time of 10 milliseconds, then to actually transfer a 100Kbyte process we require:

Transfer Time = 100K / 1,000 = 1/10 seconds
= 100 milliseconds

Access time = 10 milliseconds

Total time = 110 milliseconds

As both the swap out and swap in should take place, the total swap time is then about 220 milliseconds (above time is doubled). A round robin CPU scheduling should have a time slot size much larger relative to swap time of 220 milliseconds. Also if process is not utilising memory space and just waiting for I/O operation or blocked, it should be swapped.

1.3 LOGICAL AND PHYSICAL ADDRESS SPACE

The computer interacts via logical and physical addressing to map memory. Logical address is the one that is generated by CPU, also referred to as virtual address. The program perceives this address space. Physical address is the actual address understood by computer hardware i.e., memory unit. Logical to physical address translation is taken care by the Operating System. The term *virtual memory* refers to the abstraction of separating LOGICAL memory (i.e., memory as seen by the process) from PHYSICAL memory (i.e., memory as seen by the processor). Because of this separation, the programmer needs to be aware of only the logical memory space while the operating system maintains two or more levels of physical memory space.

In compile-time and load-time address binding schemes these two tend to be the same. These differ in execution-time address binding scheme and the MMU (Memory Management Unit) handles translation of these addresses.

Definition: MMU (as shown in the *Figure 3*) is a hardware device that maps logical address to the physical address. It maps the virtual address to the real store location. The simple MMU scheme adds the relocation register contents to the base address of the program that is generated at the time it is sent to memory.

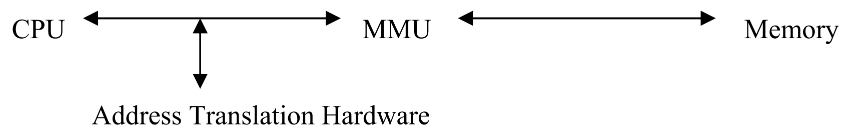


Figure 3: Role of MMU

The entire set of logical addresses forms logical address space and set of all corresponding physical addresses makes physical address space.

1.4 SINGLE PROCESS MONITOR (MONOPROGRAMMING)

In the simplest case of single-user system everything was easy as at a time there was just one process in memory and no address translation was done by the operating system dynamically during execution. Protection of OS (or part of it) can be achieved by keeping it in ROM.We can also have a separate OS address space only accessible in supervisor mode as shown in *Figure 4*.

The user can employ overlays if memory requirement by a program exceeds the size of physical memory. In this approach only one process at a time can be in running state in the system. Example of such system is MS-DOS which is a single tasking system having a command interpreter. Such an arrangement is limited in capability and performance.

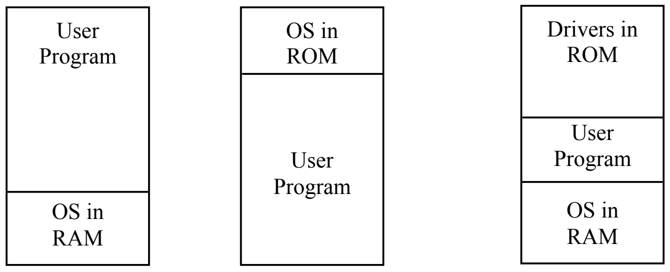


Figure 4: Single Partition System

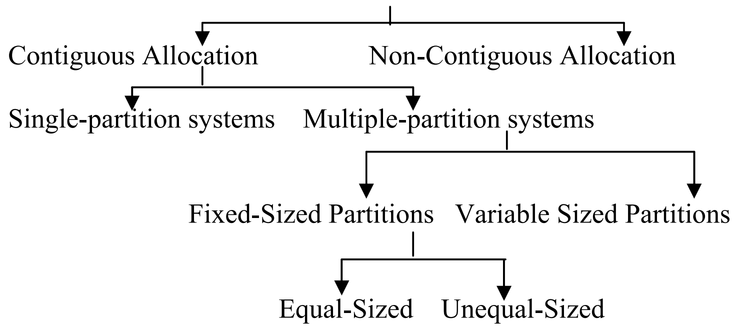
1.5 CONTIGUOUS ALLOCATION METHODS

In a practical scenario Operating System could be divided into several categories as shown in the hierarchical chart given below:

1) Single process system

2) Multiple process system with two types: Fixed partition memory and variable partition memory.

Memory Allocation



Further we will learn these schemes in next section.

Partitioned Memory allocation:

The concept of multiprogramming emphasizes on maximizing CPU utilisation by overlapping CPU and I/O.Memory may be allocated as:

- Single large partition for processes to use or
- Multiple partitions with a single process using a single partition.

1.5.1 Single-Partition System

This approach keeps the Operating System in the lower part of the memory and other user processes in the upper part. With this scheme, Operating System can be protected from updating in user processes. Relocation-register scheme known as *dynamic relocation* is useful for this purpose. It not only protects user processes from each other but also from changing OS code and data. Two registers are used: relocation register, contains value of the smallest physical address and limit register, contains logical addresses range. Both these are set by Operating System when the job starts. At load time of program (i.e., when it has to be relocated) we must establish "addressability" by adjusting the relocation register contents to the new starting address for the program. The scheme is shown in *Figure 5.*

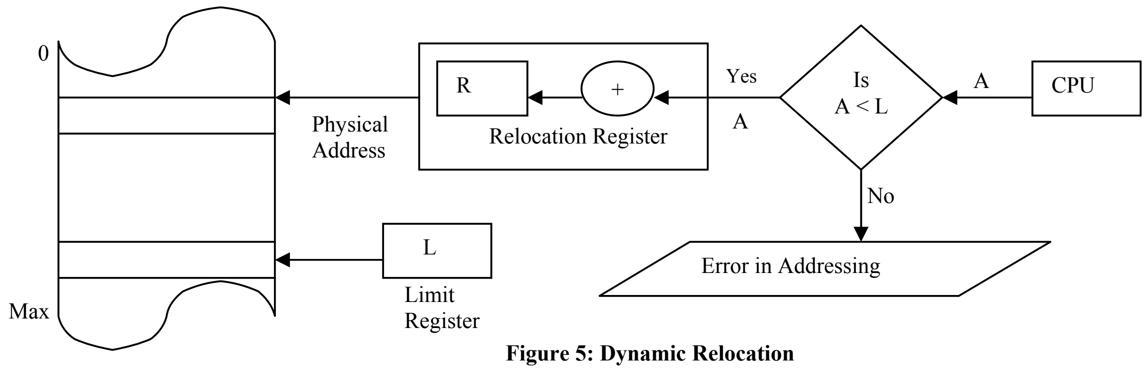


Figure 5: Dynamic Relocation

The contents of a relocation register are implicitly added to any address references generated by the program. Some systems use base registers as relocation register for easy addressability as these are within programmer's control. Also, in some systems, relocation is managed and accessed by Operating System only.

To summarize this, we can say, in dynamic relocation scheme if the logical address space range is 0 to *Max* then physical address space range is R+0 to R+Max (where R is relocation register contents). Similarly, a limit register is checked by H/W to be sure that logical address generated by CPU is not bigger than size of the program.

1.5.2 Multiple Partition System: Fixed-sized partition

This is also known as *static partitioning* scheme as shown in *Figure 6*. Simple memory management scheme is to divide memory into *n* (possibly unequal) fixed-sized partitions, each of which can hold exactly one process. The degree of multiprogramming is dependent on the number of partitions. IBM used this scheme for systems 360 OS/MFT (Multiprogramming with a fixed number of tasks). The partition boundaries are not movable (must reboot to move a job). We can have one queue per partition or just a single queue for all the partitions.

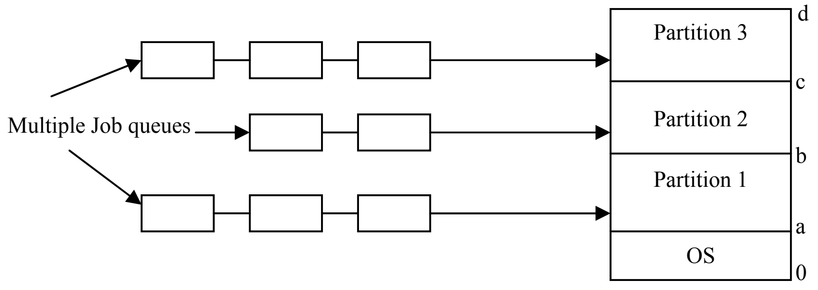


Figure 6: Multiple Partition System

Initially, whole memory is available for user processes and is like a large block of available memory. Operating System keeps details of available memory blocks and occupied blocks in tabular form. OS also keeps track on memory requirements of each process. As processes enter into the input queue and when sufficient space for it is available, process is allocated space and loaded. After its execution is over it releases its occupied space and OS fills this space with other processes in input queue. The block of available memory is known as a *Hole*. Holes of various sizes are scattered throughout the memory. When any process arrives, it is allocated memory from a hole that is large enough to accommodate it. This example is shown in *Figure 7:*

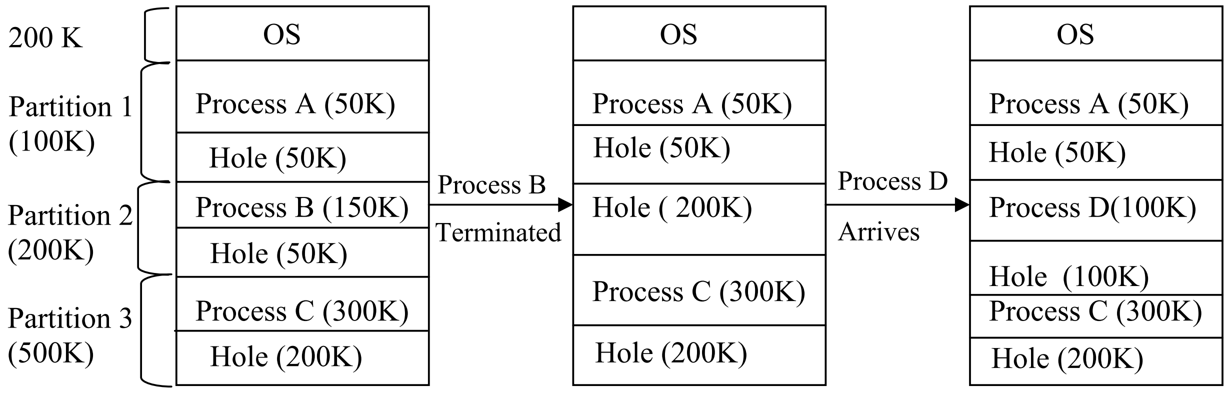


Figure 7: Fixed-sized Partition Scheme

If a hole is too large, it is divided into two parts:

1) One that is allocated to next process of input queue

2) Added with set of holes.

Within a partition if two holes are adjacent then they can be merged to make a single large hole. But this scheme suffers from fragmentation problem. Storage fragmentation occurs either because the user processes do not completely accommodate the allotted partition or partition remains unused, if it is too small to hold any process from input queue. Main memory utilisation is extremely inefficient. Any program, no matter how small, occupies entire partition. In our example, process B takes 150K of partition2 (200K size). We are left with 50K sized hole. This phenomenon, in which there is wasted space internal to a partition, is known as *internal fragmentation*. It occurs because initially process is loaded in partition that is large enough to hold it (i.e., allocated memory may be slightly larger than requested memory). “Internal” here means memory that is internal to a partition, but is not in use.

Variable-sized Partition:

This scheme is also known as *dynamic partitioning*. In this scheme, boundaries are not fixed. Processes accommodate memory according to their requirement. There is no wastage as partition size is exactly same as the size of the user process. Initially when processes start this wastage can be avoided but later on when they terminate they leave holes in the main storage. Other processes can accommodate these, but eventually they become too small to accommodate new jobs as shown in *Figure 8*.

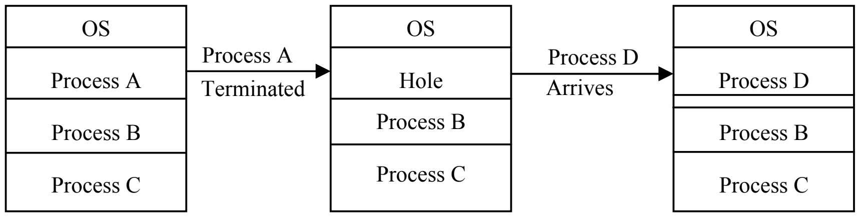


Figure 8: Variable sized partitions

IBM used this technique for OS/MVT (Multiprogramming with a Variable number of Tasks) as the partitions are of variable length and number. But still fragmentation anomaly exists in this scheme. As time goes on and processes are loaded and removed from memory, fragmentation increases and memory utilisation declines. This wastage of memory, which is external to partition, is known as *external fragmentation*. In this, though there is enough total memory to satisfy a request but as it is not contiguous and it is fragmented into small holes, that can't be utilised.

External fragmentation problem can be resolved by **coalescing holes** and **storage compaction**. **Coalescing** holes is process of merging existing hole adjacent to a process that will terminate and free its allocated space. Thus, new adjacent holes and existing holes can be viewed as a single large hole and can be efficiently utilised. There is another possibility that holes are distributed throughout the memory. For utilising such scattered holes, shuffle all occupied areas of memory to one end and leave all free memory space as a single large block, which can further be utilised. This mechanism is known as *Storage Compaction*, as shown in *Figure 9*.

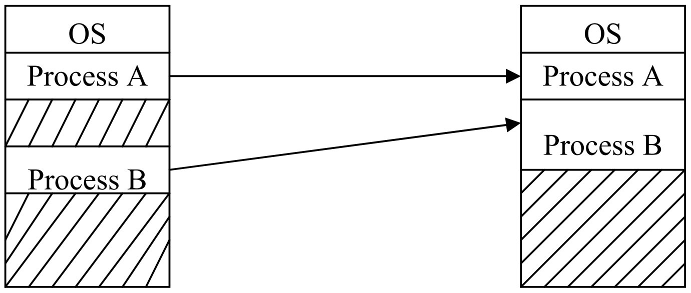


Figure 9: Storage Compaction

But storage compaction also has its limitations as shown below:

1) It requires extra overheads in terms of resource utilisation and large response time.

2) Compaction is required frequently because jobs terminate rapidly. This enhances system resource consumption and makes compaction expensive.

3) Compaction is possible only if dynamic relocation is being used (at run-time). This is because the memory contents that are shuffled (i.e., relocated) and executed in new location require all internal addresses to be relocated.

In a multiprogramming system memory is divided into a number of fixed size or variable sized partitions or regions, which are allocated to running processes. For example: a process needs *m* words of memory may run in a partition of *n* words where *n* is greater than or equal to *m.* The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmentation and external fragmentation. The difference (*n-m*) is called internal fragmentation, memory which is internal to a partition but is not being use. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

In order to solve this problem, we can either compact the memory making large free memory blocks, or implement paging scheme which allows a program's memory to be noncontiguous, thus permitting a program to be allocated physical memory wherever it is available.

☞ **Check Your Progress 1**

1) What are the four important tasks of a memory manager?

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

……………

2) What are the three tricks used to resolve absolute addresses?

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

……………

3) What are the problems that arise with absolute addresses in terms of swapping?

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

…………...

1.6 PAGING

We will see the principles of operation of the paging in the next section.

Paging scheme solves the problem faced in variable sized partitions like external fragmentation.

1.6.1 Principles of Operation

In a paged system, logical memory is divided into a number of fixed sizes 'chunks' called *pages*. The physical memory is also predivided into same fixed sized blocks (as is the size of pages) called *page frames*. The page sizes (also the frame sizes) are always powers of 2, and vary between 512 bytes to 8192 bytes per page. The reason behind this is implementation of paging mechanism using page number and page offset. This is discussed in detail in the following sections:

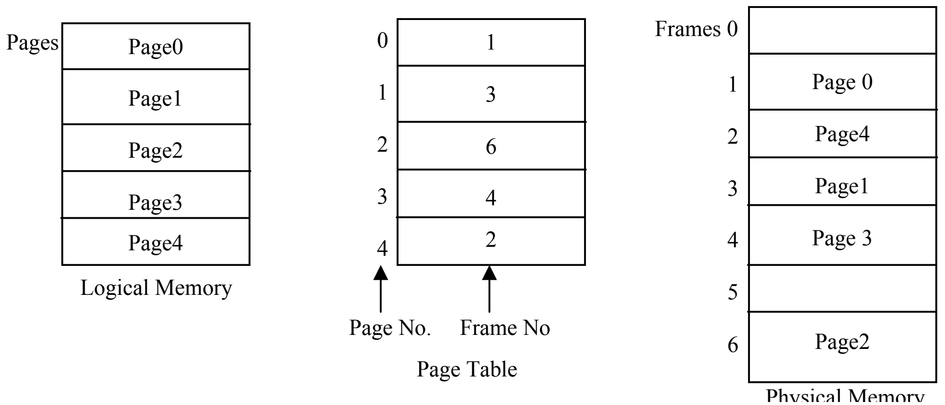


Figure 10: Principle of operation of paging

Each process page is loaded to some memory frame. These pages can be loaded into contiguous frames in memory or into noncontiguous frames also as shown in *Figure 10*. The external fragmentation is alleviated since processes are loaded into separate holes.

1.6.2 Page Allocation

In variable sized partitioning of memory every time when a process of size *n* is to be loaded, it is important to know the best location from the list of available/free holes. This dynamic storage allocation is necessary to increase efficiency and throughput of system. Most commonly used strategies to make such selection are:

1) **Best-fit Policy:** Allocating the hole in which the process fits most "tightly" i.e., the difference between the hole size and the process size is the minimum one.
2) **First-fit Policy:** Allocating the first available hole (according to memory order), which is big enough to accommodate the new process.
3) **Worst-fit Policy:** Allocating the largest hole that will leave maximum amount of unused space i.e., leftover space is maximum after allocation.

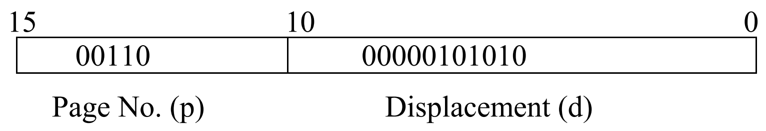
Now, question arises which strategy is likely to be used? In practice, best-fit and first-fit are better than worst-fit. Both these are efficient in terms of time and storage requirement. Best-fit minimize the leftover space, create smallest hole that could be rarely used. First-fit on the other hand requires least overheads in its implementation because of its simplicity. Possibly worst-fit also sometimes leaves large holes that could further be used to accommodate other processes. Thus all these policies have their own merits and demerits.

1.6.3 Hardware Support for Paging

Every logical page in paging scheme is divided into two parts:

1) A page number (p) in logical address space
2) The displacement (or offset) in page p at which item resides (i.e., from start of page).

This is known as Address Translation scheme. For example, a 16-bit address can be divided as given in *Figure* below:



Here, as page number takes 5bits, so range of values is 0 to 31(i.e. 2^5-1). Similarly, offset value uses 11-bits, so range is 0 to 2023(i.e., 2^{11}–1). Summarizing this we can say paging scheme uses 32 pages, each with 2024 locations.

The table, which holds virtual address to physical address translations, is called the **page table**. As displacement is constant, so only translation of virtual page number to physical page is required. This can be seen diagrammatically in *Figure 11.*

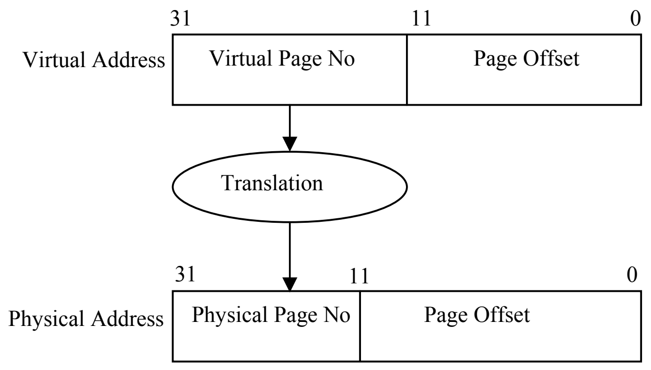


Figure 11: Address Translation scheme

Page number is used as an index into a page table and the latter contains base address of each corresponding physical memory page number (Frame). This reduces dynamic relocation efforts. The Paging hardware support is shown diagrammatically in *Figure 12*:

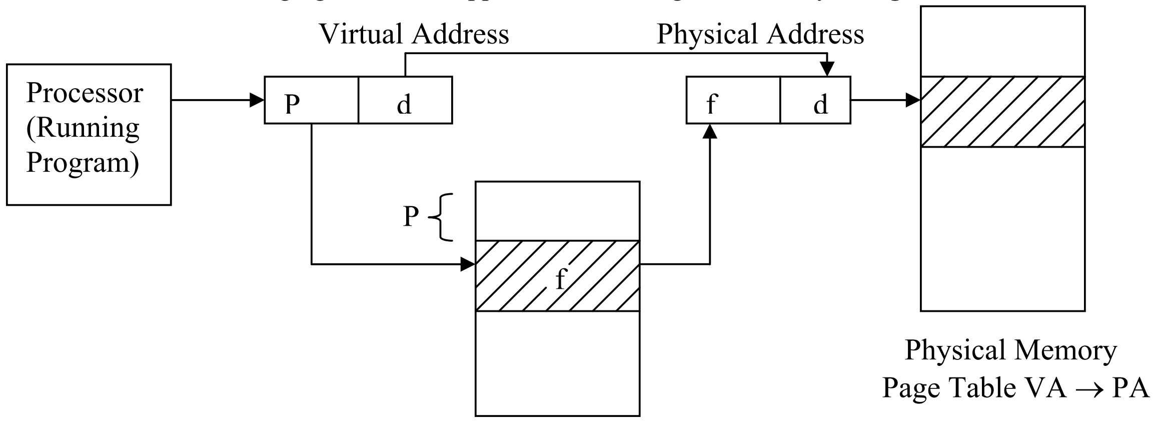


Figure 12: Direct Mapping

Paging address Translation by direct mapping

This is the case of direct mapping as page table sends directly to physical memory page. This is shown in *Figure 12*. But disadvantage of this scheme is its speed of translation. This is because page table is kept in primary storage and its size can be considerably large which increases instruction execution time (also access time) and hence decreases system speed. To overcome this additional hardware support of registers and buffers can be used. This is explained in next section.

Paging Address Translation with Associative Mapping

This scheme is based on the use of dedicated registers with high speed and efficiency. These small, fast-lookup cache help to place the entire page table into a content-addresses associative storage, hence speed-up the lookup problem with a cache. These are known as associative registers or Translation Look-aside Buffers (TLB's). Each register consists of two entries:

1) Key, which is matched with logical page p.
2) Value which returns page frame number corresponding to p.

It is similar to direct mapping scheme but here as TLB's contain only few page table entries, so search is fast. But it is quite expensive due to register support. So, both direct and associative mapping schemes can also be combined to get more benefits. Here, page number is matched with all associative registers simultaneously. The percentage of the number of times the page is found in TLB's is called hit ratio. If it is not found, it is searched in page table and added into TLB. But if TLB is already full then page replacement policies can be used. Entries in TLB can be limited only. This combined scheme is shown in *Figure 13.*

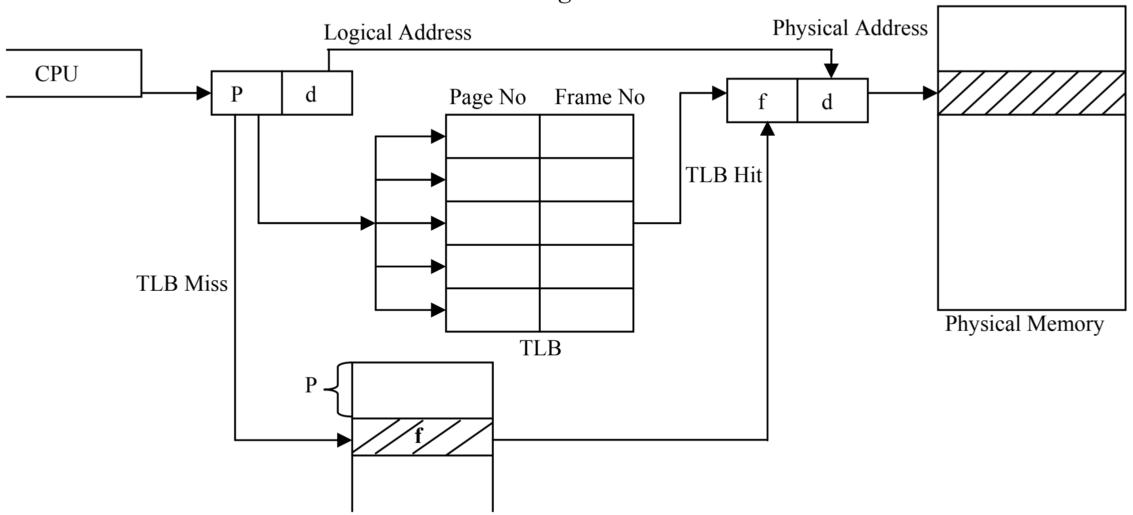


Figure 13: Combined Associative/ Direct Mapping

1.6.4 Protection and Sharing

Paging hardware typically also contains some protection mechanism. In page table corresponding to each frame a protection bit is associated. This bit can tell if page is read-only or read-write. Sharing code and data takes place if two page table entries in different processes point to same physical page, the processes share the memory. If one process writes the data, other process will see the changes. It is a very efficient way to communicate. Sharing must also be controlled to protect modification and accessing data in one process by another process. For this programs are kept separately as procedures and data, where procedures and data that are non-modifiable (pure/reentrant code) can be shared. Reentrant code cannot modify itself and must make sure that it has a separate copy of per-process global variables. Modifiable data and procedures cannot be shared without concurrency controls. Non-modifiable procedures are also known as pure procedures or reentrant codes (can't change during execution). For example, only one copy of editor or compiler code can be kept in memory, and all editor or compiler processes can execute that single copy of the code. This helps memory utilisation. Major advantages of paging scheme are:

1) Virtual address space must be greater than main memory size.i.e., can execute program with large logical address space as compared with physical address space.

2) Avoid external fragmentation and hence storage compaction.

3) Full utilisation of available main storage.

Disadvantages of paging include internal fragmentation problem i.e., wastage within allocated page when process is smaller than page boundary. Also, extra resource consumption and overheads for paging hardware and virtual address to physical address translation takes place.

1.7 SEGMENTATION

In the earlier section we have seen the memory management scheme called as paging. In general, a user or a programmer prefers to view system memory as a collection of variable-sized segments rather than as a linear array of words. Segmentation is a memory management scheme that supports this view of memory.

1.7.1 Principles of Operation

Segmentation presents an alternative scheme for memory management. **This scheme** divides the logical address space into variable length chunks, called segments, with no proper ordering among them. Each segment has a name and a length. For simplicity, segments are referred by a segment number, rather than by a name. Thus, the logical addresses are expressed as a pair of segment number and offset within segment. It allows a program to be broken down into logical parts according to the user view of the memory, which is then mapped into physical memory. Though logical addresses are two-dimensional but actual physical addresses are still one-dimensional array of bytes only.

1.7.2 Address Translation

This mapping between two is done by segment table, which contains segment base and its limit. The segment base has starting physical address of segment, and segment limit provides the length of segment. This scheme is depicted in *Figure 14*.

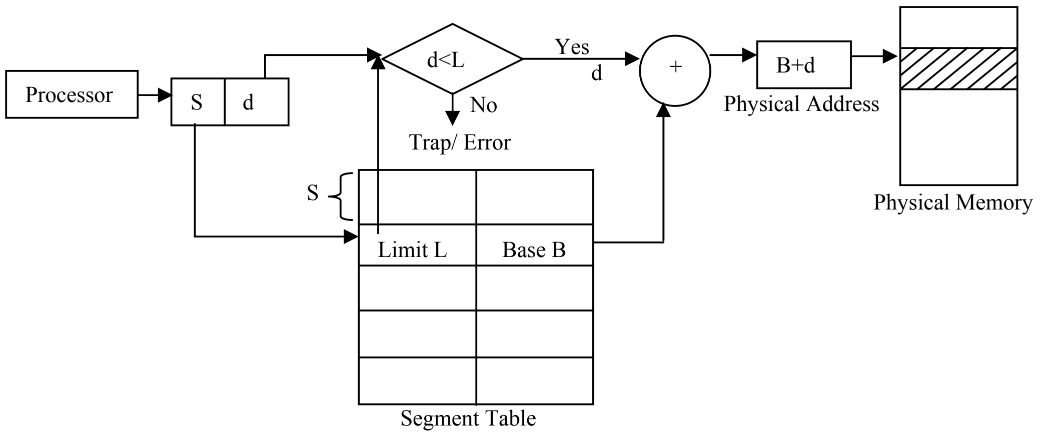
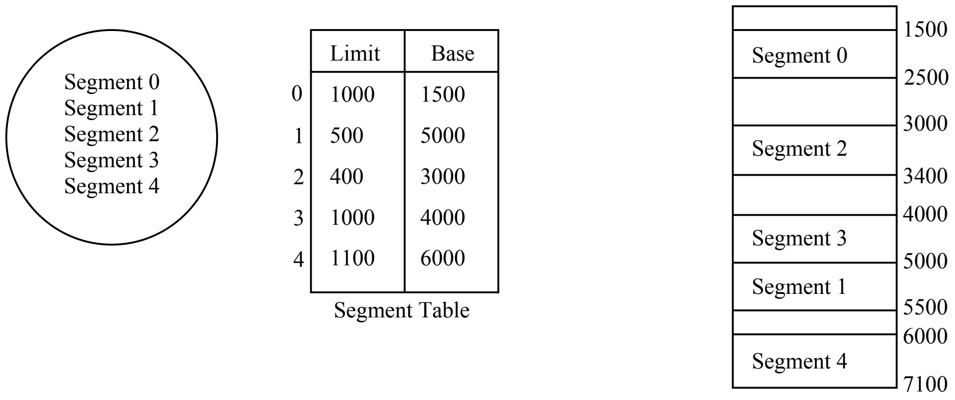


Figure 14: Address Translation

The offset d must range between 0 and segment limit/length, otherwise it will generate address error. For example, consider situation shown in *Figure 15*.



	Limit	Base
0	1000	1500
1	500	5000
2	400	3000
3	1000	4000
4	1100	6000

Figure 15: Principle pf operation of representation

This scheme is similar to variable partition allocation method with improvement that the process is divided into parts. For fast retrieval we can use registers as in paged scheme. This is known as a segment-table length register (STLR). The segments in a segmentation scheme correspond to logical divisions of the process and are defined by program names. Extract the segment number and offset from logical address first. Then use segment number as index into segment table to obtain segment base address and its limit /length. Also, check that the offset is not greater than given limit in segment table. Now, general physical address is obtained by adding the offset to the base address.

1.7.3 Protection and Sharing

This method also allows segments that are read-only to be shared, so that two processes can use shared code for better memory efficiency. The implementation is such that no program can read from or write to segments belonging to another program, except the segments that have been set up to be shared. With each segment-table entry protection bit specifying segment as read-only or execute only can be used. Hence illegal attempts to write into a read-only segment can be prevented.

Sharing of segments can be done by making common /same entries in segment tables of two different processes which point to same physical location. Segmentation may suffer from external fragmentation i.e., when blocks of free memory are not enough to accommodate a segment. Storage compaction and coalescing can minimize this drawback.

☞ **Check Your Progress 2**

1) What is the advantage of using Base and Limit registers?

……………………………………………………………………………….……..
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
…………

2) How does lookup work with TLB's?

……………………………………………………………………………….……..
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
……………

3) Why is page size always powers of 2?

……………………………………………………………………………….……..
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
……………

4) A system with 18-bit address uses 6 bits for page number and next 12 bits for offset. Compute the total number of pages and express the following address according to paging scheme 001011(page number) and 000000111000(offset)?

……………………………………………………………………………….……..
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
……………

1.8 SUMMARY

In this unit, we have learnt how memory resource is managed and how processes are protected from each other. The previous two sections covered memory allocation techniques like swapping and overlays, which tackle the utilisation of memory. Paging and segmentation was presented as memory management schemes. Both have their own merits and demerits. We have also seen how paging is based on physical form of process and is independent of the programming structures, while segmentation is dependent on logical structure of process as viewed by user. We have also considered fragmentation (internal and external) problems and ways to tackle them to increase level of multiprogramming and system efficiency. Concept of relocation and compaction helps to overcome external fragmentation.

1.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) i) Keep track of which parts of memory are in use.

   ii) Allocate memory to processes when they require it.

   iii) Protect memory against unauthorised accesses.

   iv) Simulate the appearance of a bigger main memory by moving data automatically between main memory and disk.

2) i) Position Independent Code [PIC]

   ii) Dynamic Relocation

   iii) Base and Limit Registers.

3) When a program is loaded into memory at different locations, the absolute addresses will not work without software or hardware tricks.

Check Your Progress 2

1) These help in dynamic relocation. They make a job easy to move in memory.

2) With TLB support steps determine page number and offset first. Then look up page number in TLB. If it is there, add offset to physical page number and access memory location. Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB.Then restart the instruction.

3) Paging is implemented by breaking up an address into a page and offset number. It is efficient to break address into X page bits and Y offset bits, rather than calculating address based on page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

4) Page Number (6 Bits)=001011 = 11(decimal)

   Page Number range= $(2^6–1)$ = 63

Displacement (12 Bits)=000000111000 = 56(decimal)

1.10 FURTHER READINGS

1) H.M.Deitel, *Operating Systems*, published by Pearson Education Asia, New Delhi.

2) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts, Wiley and Sons* (Asia) Publications, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, published by Prentice Hall of India Pvt. Ltd., New Delhi.

4) Colin Ritchie, *Operating Systems incorporating UNIX and Windows*, BPB Publications, New Delhi.

UNIT 2 VIRTUAL MEMORY

Structure **Page Nos.**

2.0 INTRODUCTION

In the earlier unit, we have studied Memory Management covering topics like the overlays, contiguous memory allocation, static and dynamic partitioned memory allocation, paging and segmentation techniques. In this unit, we will study an important aspect of memory management known as Virtual memory.

Storage allocation has always been an important consideration in computer programming due to the high cost of the main memory and the relative abundance and lower cost of secondary storage. Program code and data required for execution of a process must reside in the main memory but the main memory may not be large enough to accommodate the needs of an entire process. Early computer programmers divided programs into the sections that were transferred into the main memory for the period of processing time. As the program proceeded, new sections moved into the main memory and replaced sections that were not needed at that time. In this early era of computing, the programmer was responsible for devising this overlay system.

As higher-level languages became popular for writing more complex programs and the programmer became less familiar with the machine, the efficiency of complex programs suffered from poor overlay systems. The problem of storage allocation became more complex.

Two theories for solving the problem of inefficient memory management emerged -- static and dynamic allocation. ***Static*** allocation assumes that the availability of memory resources and the memory reference string of a program can be predicted. ***Dynamic*** allocation relies on memory usage increasing and decreasing with actual program needs, not on predicting memory needs.

Program objectives and machine advancements in the 1960s made the predictions required for static allocation difficult, if not impossible. Therefore, the dynamic allocation solution was generally accepted, but opinions about implementation were still divided. One group believed the programmer should continue to be responsible for storage allocation, which would be accomplished by system calls to allocate or deal locate memory. The second group supported **automatic storage allocation** performed by the operating system, because of increasing complexity of storage allocation and emerging importance of multiprogramming. In 1961, two groups proposed a one-level memory store. One proposal called for a very large main memory to alleviate any need for storage allocation. This solution was not possible due to its very high cost. The second proposal is known as virtual memory.

In this unit, we will go through virtual memory and related topics.

2.1 OBJECTIVES

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

- discuss why virtual memory is needed;
- define virtual memory and its underlying concepts;
- describe the page replacement policies like Optimal, FIFO and LRU;
- discuss the concept of thrashing, and
- explain the need of the combined systems like Segmented paging and Paged segmentation.

2.2 VIRTUAL MEMORY

It is common for modern processors to be running multiple processes at one time. Each process has an address space associated with it. To create a whole complete address space for each process would be much too expensive, considering that processes may be created and killed often, and also considering that many processes use only a tiny bit of their possible address space. Last but not the least, even with modern improvements in hardware technology, machine resources are still finite. Thus, it is necessary to share a smaller amount of physical memory among many processes, with each process being given the appearance of having its own exclusive address space.

The most common way of doing this is a technique called virtual memory, which has been known since the 1960s but has become common on computer systems since the late 1980s. The virtual memory scheme divides physical memory into blocks and allocates blocks to different processes. Of course, in order to do this sensibly it is highly desirable to have a protection scheme that restricts a process to be able to access only those blocks that are assigned to it. Such a protection scheme is thus a necessary, and somewhat involved, aspect of any virtual memory implementation. One other advantage of using virtual memory that may not be immediately apparent is that it often reduces the time taken to launch a program, since not all the program code and data need to be in physical memory before the program execution can be started. Although sharing the physical address space is a desirable end, it was not the sole reason that virtual memory became common on contemporary systems. Until the late 1980s, if a program became too large to fit in one piece in physical memory, it was the programmer's job to see that it fit. Programmers typically did this by breaking programs into pieces, each of which was mutually exclusive in its logic. When a program was launched, a main piece that initiated the execution would first be loaded into physical memory, and then the other parts, called overlays, would be loaded as needed.

It was the programmer's task to ensure that the program never tried to access more physical memory than was available on the machine, and also to ensure that the proper overlay was loaded into physical memory whenever required. These responsibilities made for complex challenges for programmers, who had to be able to divide their programs into logically separate fragments, and specify a proper scheme to load the right fragment at the right time. Virtual memory came about as a means to relieve programmers creating large pieces of software of the wearisome burden of designing overlays.

Virtual memory automatically manages two levels of the memory hierarchy, representing the main memory and the secondary storage, in a manner that is invisible to the program that is running. The program itself never has to bother with the physical location of any fragment of the virtual address space. A mechanism called relocation allows for the same program to run in any location in physical memory, as well. Prior to the use of virtual memory, it was common for machines to include a relocation register just for that purpose. An expensive and messy solution to the hardware solution of a virtual memory would be software that changed all addresses in a program each time it was run. Such a solution would increase the running times of programs significantly, among other things.

Virtual memory enables a program to ignore the physical location of any desired block of its address space; a process can simply seek to access any block of its address space without concern for where that block might be located. If the block happens to be located in the main memory, access is carried out smoothly and quickly; else, the virtual memory has to bring the block in from secondary storage and allow it to be accessed by the program.

The technique of virtual memory is similar to a degree with the use of processor caches. However, the differences lie in the block size of virtual memory being typically much larger (64 kilobytes and up) as compared with the typical processor cache (128 bytes and up). The hit time, the miss penalty (the time taken to retrieve an item that is not in the cache or primary storage), and the transfer time are all larger in case of virtual memory. However, the miss rate is typically much smaller. (This is no accident−since a secondary storage device, typically a magnetic storage device with much lower access speeds, has to be read in case of a miss, designers of virtual memory make every effort to reduce the miss rate to a level even much lower than that allowed in processor caches).

Virtual memory systems are of two basic kinds—those using fixed-size blocks called pages, and those that use variable-sized blocks called segments.

Suppose, for example, that a main memory of 64 megabytes is required but only 32 megabytes is actually available. To create the illusion of the larger memory space, the memory manager would divide the required space into units called pages and store the contents of these pages in mass storage. A typical page size is no more than four kilobytes. As different pages are actually required in main memory, the memory manager would exchange them for pages that are no longer required, and thus the other software units could execute as though there were actually 64 megabytes of main memory in the machine.

In brief we can say that virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that the program can be larger than physical memory. Virtual memory can be implemented via demand paging and demand segmentation.

2.2.1 Principles of Operation

The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilise. For example, virtual memory might contain twice as many addresses as main memory. A program using all of virtual memory, therefore, would not be able to fit in main memory all at once. Nevertheless, the computer could execute such a program by copying into the main memory those portions of the program needed at any given point during execution.

To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.

Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory management unit

(MMU) translates virtual addresses to physical addresses at run-time. The process of translating virtual addresses into real addresses is called ***mapping***. The copying of virtual pages from disk to main memory is known as ***paging*** or ***swapping***.

Some physical memory is used to keep a list of references to the most recently accessed information on an I/O (input/output) device, such as the hard disk. The optimisation it provides is that it is faster to read the information from physical memory than use the relevant I/O channel to get that information. This is called ***caching***. It is implemented inside the OS.

2.2.2 Virtual Memory Management

This section provides the description of how the virtual memory manager provides virtual memory. It explains how the logical and physical address spaces are mapped to one another and when it is required to use the services provided by the Virtual Memory Manager.

Before going into the details of the management of the virtual memory, let us see the functions of the virtual memory manager. It is responsible to:

- Make portions of the logical address space resident in physical RAM
- Make portions of the logical address space immovable in physical RAM
- Map logical to physical addresses
- Defer execution of application-defined interrupt code until a safe time.

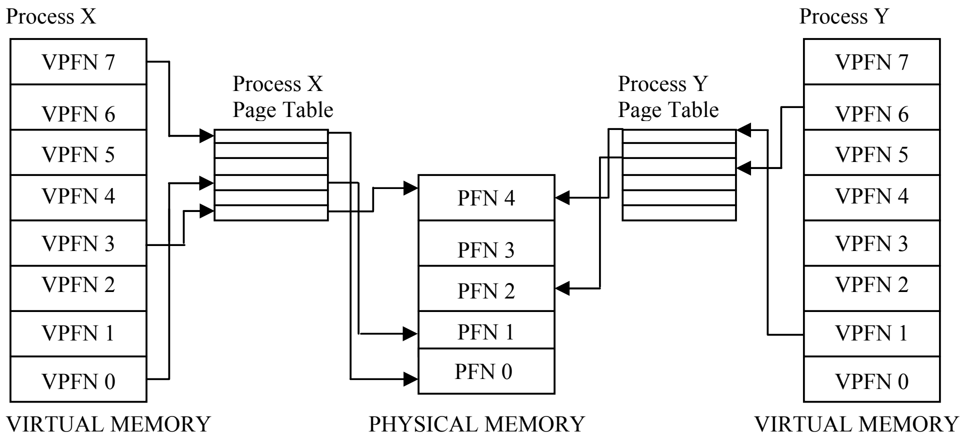


Figure 1: Abstract model of Virtual to Physical address mapping

Before considering the methods that various operating systems use to support virtual memory, it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location of operands in the memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into small blocks called ***pages***. These pages are all of the same size. (It is not necessary that all the pages should be of same size but if they were not, the system would be very hard to administer). Linux on Alpha AXP systems uses 8 Kbytes pages and on Intel x86 systems it uses 4 Kbytes pages. Each of these pages is given a unique number; the page frame number (PFN) as shown in the *Figure 1*.

In this paged model, a virtual address is composed of two parts, an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses ***page tables***.

The *Figure1* shows the virtual address spaces of two processes, process *X* and process *Y*, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process *X's* virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process *Y's* virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- *Valid flag* : This indicates if this page table entry is valid.
- *PFN* : The physical page frame number that this entry is describing.
- *Access control information* : This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at the *Figure1* and assuming a page size of *0x2000* bytes (which is decimal 8192) and an address of *0x2194* in process *Y's* virtual address space then the processor would translate that address into offset *0x194* into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However, the processor delivers it, this is known as a ***page fault*** and the operating system is notified of the faulting virtual address and the reason for the page fault. A page fault is serviced in a number of steps:

i) Trap to the OS.

ii) Save registers and process state for the current process.

iii) Check if the trap was caused because of a page fault and whether the page reference is legal.

iv) If yes, determine the location of the required page on the backing store.

v) Find a free frame.

vi) Read the required page from the backing store into the free frame. (During this I/O, the processor may be scheduled to some other process).

vii) When I/O is completed, restore registers and process state for the process which caused the page fault and save state of the currently executing process.

viii) Modify the corresponding PT entry to show that the recently copied page is now in memory.

ix) Resume execution with the instruction that caused the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process *Y's* virtual page frame number 1 is mapped to physical page frame number 4 which starts at *0x8000* (4 x *0x2000*). Adding in the *0x194* byte offset gives us a final physical address of *0x8194*. By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order.

2.2.3 Protection and Sharing

Memory protection in a paged environment is realised by protection bits associated with each page, which are normally stored in the page table. Each bit determines a page to be read-only or read/write. At the same time the physical address is calculated with the help of the page table, the protection bits are checked to verify whether the memory access type is correct or not. For example, an attempt to write to a read-only memory page generates a trap to the operating system indicating a memory protection violation.

We can define one more protection bit added to each entry in the page table to determine an invalid program-generated address. This bit is called valid/invalid bit, and is used to generate a trap to the operating system. Valid/invalid bits are also used to allow and disallow access to the corresponding page. This is shown in *Figure 2*.

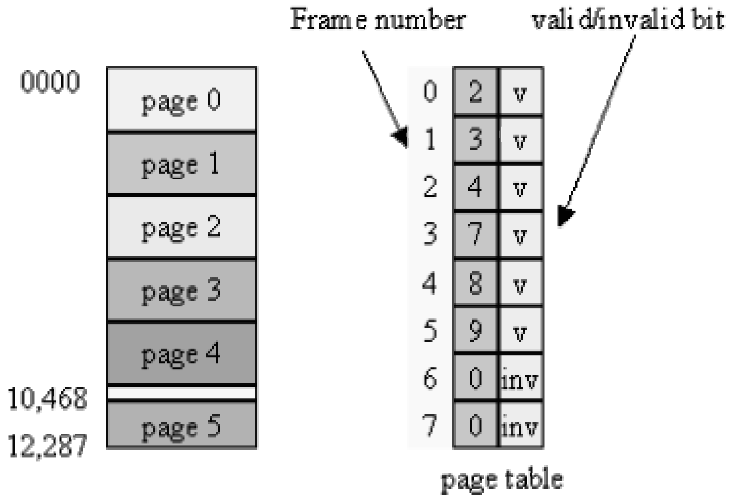


Figure 2: Protection bit in the page table Shared pages

The paging scheme supports the possibility of sharing common program code. For example, a system that supports 40 users, each of them executes a text editor. If the text editor consists of 30 KB of code and 5 KB of data, we need 1400 KB. If the code is reentrant, i.e., it never changes by any write operation during execution (non-self-modifying code) it could be shared as presented in *Figure 3*.

Only one copy of the editor needs to be stored in the physical memory. In each page table, the included editor page is mapped onto the same physical copy of the editor, but the data pages are mapped onto different frames. So, to support 40 users, we only need one copy of the editor, i.e., 30 KB, plus 40 copies of the 5 KB of data pages per user; the total required space is now 230 KB instead of 1400 KB.

Other heavily used programs such as assembler, compiler, database systems etc. can also be shared among different users. The only condition for it is that the code must be reentrant. It is crucial to correct the functionality of shared paging scheme so that the pages are unchanged. If one user wants to change a location, it would be changed for all other users.

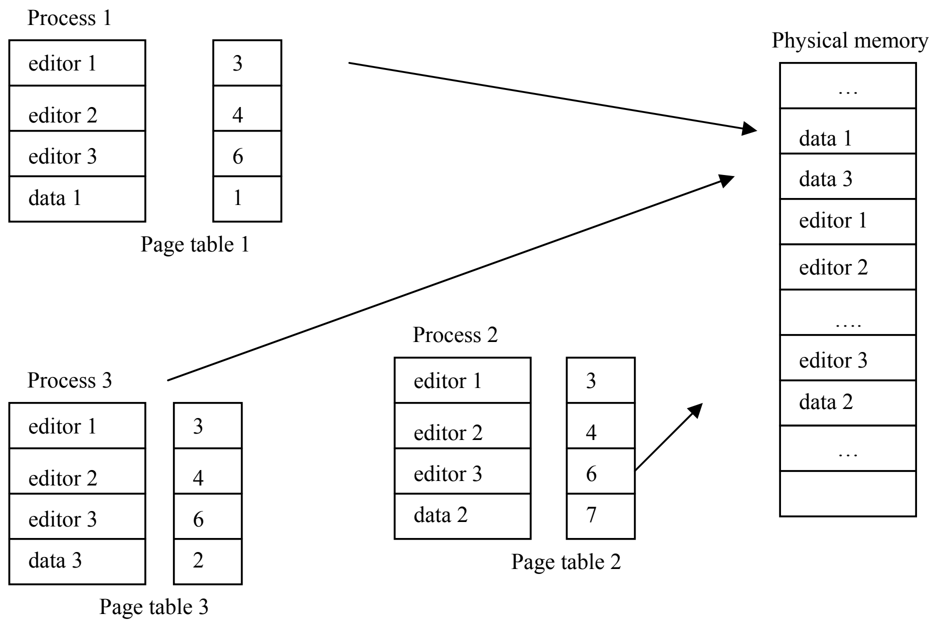


Figure 3: Paging scheme supporting the sharing of program code

2.3 DEMAND PAGING

In a multiprogramming system memory is divided into a number of fixed-size or variable-sized partitions or regions that are allocated to running processes. For example: a process needs *m* words of memory may run in a partition of *n* words where $n \geq m$. The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmented into many scattered blocks. We distinguish between *internal fragmentation* and *external fragmentation*. The difference *(n – m)* is called internal fragmentation, memory that is internal to a partition but is not being used. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

In order to solve this problem, we can either ***compact*** the memory making large free memory blocks, or implement ***paging*** scheme which allows a program's memory to be non-contiguous, thus permitting a program to be allocated to physical memory.

Physical memory is divided into fixed size blocks called ***frames***. Logical memory is also divided into blocks of the same, fixed size called ***pages***. When a program is to be executed, its pages are loaded into any available memory frames from the disk. The disk is also divided into fixed sized blocks that are the same size as the memory frames.

A very important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. Normally, a user believes that memory is one contiguous space containing only his/her program. In fact, the logical memory is scattered through the physical memory that also contains other programs. Thus, the user can work correctly with his/her own view of memory because of the address translation or address mapping. The address mapping, which is controlled by the operating system and transparent to users, translates logical memory addresses into physical addresses.

Because the operating system is managing the memory, it must be sure about the nature of physical memory, for example: which frames are available, which are allocated; how many total frames there are, and so on. All these parameters are kept in a data structure called ***frame table*** that has one entry for each physical frame of memory indicating whether it is free or allocated, and if allocated, to which page of which process.

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to load only virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. Also, if the database query is a search query then it is not necessary to load the code from the database that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as ***demand paging***.

When a process attempts to access a virtual address that is not currently in memory the CPU cannot find a page table entry for the virtual page referenced. For example, in *Figure 1*, there is no entry in *Process X's* page table for virtual PFN 2 and so if *Process X* attempts to read from an address within virtual PFN 2 the CPU cannot translate the address into a physical one. At this point the CPU cannot cope and needs the operating system to fix things up. It notifies the operating system that a ***page fault*** has occurred and the operating system makes the process wait whilst it fixes things up. The CPU must bring the appropriate page into memory from the image on disk. Disk access takes a long time, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual PFN is added to the processes page table. The process is then restarted at the point where the memory fault occurred. This time the virtual memory access is made, the CPU can make the address translation and so the process continues to run. This is known as demand paging and occurs when the system is busy but also when an image is first loaded into memory. This mechanism means that a process can execute an image that only partially resides in physical memory at any one time.

The valid/invalid bit of the page table entry for a page, which is swapped in, is set as valid. Otherwise it is set as invalid, which will have no effect as long as the program never attempts to access this page. If all and only those pages actually needed are swapped in, the process will execute exactly as if all pages were brought in.

If the process tries to access a page, which was not swapped in, i.e., the valid/invalid bit of this page table, entry is set to invalid, then a page fault trap will occur. Instead of showing the "invalid address error" as usually, it indicates the operating system's failure to bring a valid part of the program into memory at the right time in order to minimize swapping overhead.

In order to continue the execution of process, the operating system schedules a disk read operation to bring the desired page into a newly allocated frame. After that, the corresponding page table entry will be modified to indicate that the page is now in memory. Because the state (program counter, registers etc.) of the interrupted process was saved when the page fault trap occurred, the interrupted process can be restarted at the same place and state. As shown, it is possible to execute programs even though parts of it are not (yet) in memory.

In the extreme case, a process without pages in memory could be executed. Page fault trap would occur with the first instruction. After this page was brought into memory, the process would continue to execute. In this way, page fault trap would occur further until every page that is needed was in memory. This kind of paging is called ***pure demand paging***. Pure demand paging says that "never bring a page into memory until it is required".

2.4 PAGE REPLACEMENT POLICIES

Basic to the implementation of virtual memory is the concept of ***demand paging***. This means that the operating system, and not the programmer, controls the swapping of pages in and out of main memory, as the active processes require them. When a process needs a non-resident page, the operating system must decide which resident page is to be replaced by the requested page. The part of the virtual memory which makes this decision is called the ***replacement policy***.

There are many approaches to the problem of deciding which page is to replace but the object is the same for all-the policy that selects the page that will not be referenced again for the longest time. A few page replacement policies are described below.

2.4.1 First In First Out (FIFO)

The First In First Out (FIFO) replacement policy chooses the page that has been in the memory the longest to be the one replaced.

Belady's Anomaly

Normally, as the number of page frames increases, the number of page faults should decrease. However, for FIFO there are cases where this generalisation will fail! This is called Belady's Anomaly. Notice that OPT's never suffers from Belady's anomaly.

2.4.2 Second Chance (SC)

The Second Chance (SC) policy is a slight modification of FIFO in order to avoid the problem of replacing a heavily used page. In this policy, a reference bit *R* is used to keep track of pages that have been recently referenced. This bit is set to 1 each time the page is referenced. Periodically, all the reference bits are set to 0 by the operating system to distinguish pages that have not been referenced recently from those that have been. Using this bit, the operating system can determine whether old pages are still being used (i.e., $R = 1$). If so, the page is moved to the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues. Thus heavily accessed pages are given a "second chance."

2.4.3 Least Recently Used (LRU)

The Least Recently Used (LRU) replacement policy chooses to replace the page which has not been referenced for the longest time. This policy assumes the recent past will approximate the immediate future. The operating system keeps track of when each page was referenced by recording the time of reference or by maintaining a stack of references.

2.4.4 Optimal Algorithm (OPT)

Optimal algorithm is defined as replace the page that will not be used for the longest period of time. It is optimal in the performance but not feasible to implement because we cannot predict future time.

Example:

Let us consider a 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	1	4
	2	2	2	2	2
		3	3	3	3
			4	5	5

In the above *Figure*, we have 6 page faults.

2.4.5 Least Frequently Used (LFU)

The Least Frequently Used (LFU) replacement policy selects a page for replacement if the page had not been used often in the past. This policy keeps count of the number of times that a page is accessed. Pages with the lowest counts are replaced while pages with higher counts remain in primary memory.

2.5 THRASHING

Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults it is necessary to be in order to appreciate the benefits of virtual memory, thrashing has a negative effect on the system.

As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault. While the transactions in the system are waiting for the paging device, CPU utilisation, system throughput and system response time decrease, resulting in below optimal performance of a system.

Thrashing becomes a greater threat as the degree of multiprogramming of the system increases.

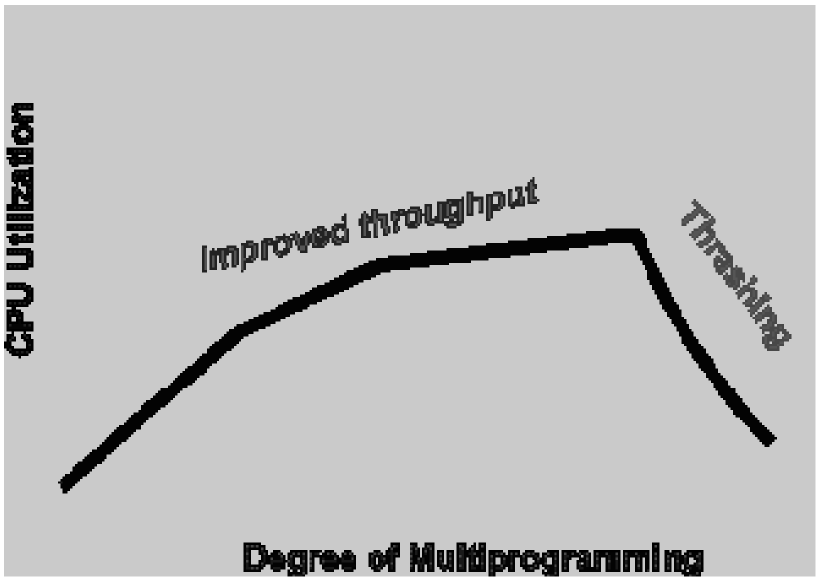


Figure 4: Degree of Multiprogramming

The graph in *Figure 4* shows that there is a degree of multiprogramming that is optimal for system performance. CPU utilisation reaches a maximum before a swift decline as the degree of multiprogramming increases and thrashing occurs in the over-extended system. This indicates that controlling the load on the system is important to avoid thrashing. In the system represented by the graph, it is important to maintain the multiprogramming degree that corresponds to the peak of the graph.

The selection of a replacement policy to implement virtual memory plays an important part in the elimination of the potential for thrashing. A policy based on the local mode will tend to limit the effect of thrashing. In local mode, a transaction will replace pages from its assigned partition. Its need to access memory will not affect transactions using other partitions. If other transactions have enough page frames in the partitions they occupy, they will continue to be processed efficiently.

A replacement policy based on the global mode is more likely to cause thrashing. Since all pages of memory are available to all transactions, a memory-intensive transaction may occupy a large portion of memory, making other transactions susceptible to page faults and resulting in a system that thrashes. To prevent thrashing, we must provide a processes as many frames as it needs. There are two techniques for this–*Working-Set Model and Page-Fault Rate.*

2.5.1 Working-Set Model

Principle of Locality

Pages are not accessed randomly. At each instant of execution a program tends to use only a small set of pages. As the pages in the set change, the program is said to move from one phase to another. The principle of locality states that most references will be to the current small set of pages in use. The examples are shown below:

Examples:

1) Instructions are fetched sequentially (except for branches) from the same page.
2) Array processing usually proceeds sequentially through the array functions repeatedly, access variables in the top stack frame.

Ramification

If we have locality, we are unlikely to continually suffer page-faults. If a page consists of 1000 instructions in self-contained loop, we will only fault once (at most) to fetch all 1000 instructions.

Working Set Definition

The working set model is based on the assumption of locality. The idea is to examine the most recent page references in the working set. If a page is in active use, it will be in the Working-set. If it is no longer being used, it will drop from the working set.

The set of pages currently needed by a process is its working set.

WS(k) for a process P is the number of pages needed to satisfy the last k page references for process P.

WS(t) is the number of pages needed to satisfy a process's page references for the last *t* units of time.

Either can be used to capture the notion of locality.

Working Set Policy

Restrict the number of processes on the ready queue so that physical memory can accommodate the working sets of all ready processes. Monitor the working sets of ready processes and, when necessary, reduce multiprogramming (i.e. swap) to avoid thrashing.

Note: Exact computation of the working set of each process is difficult, but it can be estimated, by using the reference bits maintained by the hardware to implement an aging algorithm for pages.

When loading a process for execution, pre-load certain pages. This prevents a process from having to "fault into" its working set. May be only a rough guess at start-up, but can be quite accurate on swap-in.

2.5.2 Page-Fault Rate

The ***working-set model*** is successful, and knowledge of the working set can be useful for pre-paging, but it is a scattered way to control thrashing. A page-fault frequency (page-fault rate) takes a more direct approach. In this we establish upper and lower bound on the desired page- fault rate. If the actual page fault rate exceeds the upper limit, we allocate the process another frame. If the page fault rate falls below the lower limit, we remove a Frame from the process. Thus, we can directly measure and control the page fault rate to prevent thrashing.

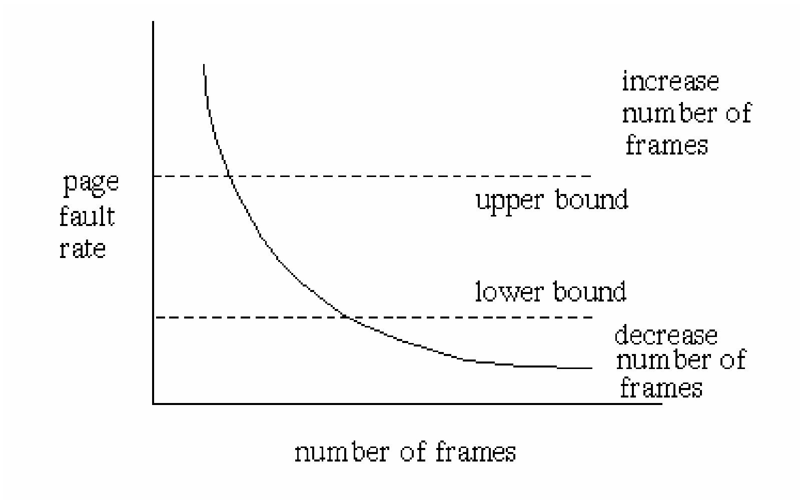


Figure 5: Page-fault frequency

Establish "acceptable" page-fault rate.

- If actual rate too low, process loses frame.
- If actual rate too high, process gains frame.

2.6 DEMAND SEGMENTATION

Segmentation

Programs generally divide up their memory usage by function. Some memory holds instructions, some static data, some dynamically allocated data, some execution frames. All of these memory types have different protection, growth, and sharing requirements. In the monolithic memory allocation of classic Virtual Memory systems, this model isn't well supported.

Segmentation addresses this by providing multiple sharable, protectable, growable address spaces that processes can access.

In pure segmentation architecture, segments are allocated like variable partitions, although the memory management hardware is involved in decoding addresses. Pure segmentation addresses replace the page identifier in the virtual address with a segment identifier, and find the proper segment (not page) to which to apply the offset.

The segment table is managed like the page table, except that segments explicitly allow sharing. Protections reside in the segment descriptors, and can use keying or explicit access control lists to apply them.

Of course, the segment name space must be carefully managed, and thus OS must provide a method of doing this. The file system can come to the rescue here−a process can ask for a file to be mapped into a segment and have the OS return the segment register to use. This is known as memory mapping files. It is slightly different from memory mapping devices, because one file system abstraction (a segment) is providing an interface to another (a file). Memory mapped files may be reflected into the file system or not and may be shared or not at the process's discretion.

The biggest problem with segmentation is the same as with variable sized real memory allocation: managing variable sized partitions can be very inefficient, especially when the segments are large compared to physical memory. External fragmentation can easily result in expensive compaction when a large segment is loaded, and swapping large segments (even when compaction is not required) can be costly.

Demand Segmentation

Same idea as demand paging applied to segments.

If a segment is loaded, base and limit are stored in the Segment Table Entry (STE) and the valid bit is set in the Page Table Entry (PTE). The PTE is accessed for each memory reference. If the segment is not loaded, the valid bit is unset. The base and limit as well as the disk address of the segment is stored in the OS table. A reference to a non-loaded segment generates a segment fault (analogous to page fault). To load a segment, we must solve both the placement question and the replacement question (for demand paging, there is no placement question).

2.7 COMBINED SYSTEMS

The combined systems are of two types. They are:

- Segmented Paging
- Paged Segmentation

2.7.1 Segmented Paging

In a pure paging scheme, the user thinks in terms of a contiguous linear address space, and internal fragmentation is a problem when portions of that address space include conservatively large estimates of the size of dynamic structures. Segmented paging is a scheme in which logically distinct (e.g., dynamically-sized) portions of the address space are deliberately given virtual addresses a LONG way apart, so we never have to worry about things bumping into each other, and the page table is implemented in such a way that the big unused sections don't cost us much. Basically the only page table organisation that doesn't work well with segmented paging is a single linear array. Trees, inverted (hash) tables, and linked lists work fine. If we always start segments at multiples of some large power of two, we can think of the high-order bits of the virtual address as specifying a segment, and the low-order bits as specifying an offset within the segment. If we have tree-structured page tables in which we use k bits to select a child of the root, and we always start segments at some multiple of 2^(word size-k), then the top level of the tree looks very much like a segment table. The only difference is that its entries are selected by the high-order address bits, rather than by some explicit architecturally visible mechanism like segment registers. Basically all modern operating systems on page-based machines use segmented paging.

2.7.2 Paged Segmentation

In a pure segmentation scheme, we still have to do dynamic space management to allocate physical memory to segments. This leads to external fragmentation and forces us to think about things like compaction. It also doesn't lend itself to virtual memory. To address these problems, we can page the segments of a segmented machine. This is paged segmentation. Instead of containing base/bound pairs, the segment table entries of a machine with paged segmentation indicate how to find the page table for the segment. In MULTICS, there was a separate page table for each segment. The segment offset was interpreted as consisting of a page number and a page offset. The base address in the segment table entry is added to the segment offset to produce a "linear address" that is then partitioned into a page number and page offset, and looked up in the page table in the normal way. Note that in a machine with pure segmentation, given a fast way to find base/bound pairs (e.g., segment registers), there is no need for a TLB. Once we go to paged segmentation, we need a TLB. The difference between segmented paging and paged segmentation lies in the user's programming model, and in the addressing modes of the CPU. On a segmented architecture, the user generally specifies addresses using an effective address that includes a segment register specification. On a paged architecture, there are no segment registers. In practical terms, managing segment registers (loading them with appropriate values at appropriate times) is a bit of a nuisance to the assembly language programmer or compiler writer. On the other hand, since it only takes a few bits to indicate a segment register, while the base address in the segment table entry can have many bits, segments provide a means of expanding the virtual address space beyond 2^(word size). We can't do segmented paging on a machine with 16-bit addresses. It's beginning to get problematical on machines with 32-bit addresses. We certainly can't build a MULTICS-style single level store, in which every file is a segment that can be accessed with ordinary loads and stores, on a 32-bit machine. Segmented architectures provide a way to get the effect we want (lots of logically separate segments that can grow without practical bound) without requiring that we buy into very large addresses. As 64-bit architectures become more common, it is possible that segmentation will become less popular. One might think that paged segmentation has an additional advantage over segmented paging: protection information is logically associated with a segment, and could perhaps be specified in the segment table and then left out of the page table. Unfortunately, protection bits are used for lots of purposes other than simply making sure we cannot write your code or execute your data.

☞ **Check Your Progress 1**

1) What are the steps that are followed by the Operating system in order to handle the page fault?

   ………………………………………………………………………………

   ………………………………………………………………………………

2) What is demand paging?

   ………………………………………………………………………………

   ………………………………………………………………………………

3) How can you implement the virtual memory?

   ………………………………………………………………………………

   ………………………………………………………………………………

4) When do the following occurs?

   i) A Page Fault
   ii) Thrashing

   ………………………………………………………………………………

   ………………………………………………………………………………

5) What should be the features of the page swap algorithm?

   ………………………………………………………………………………

   ………………………………………………………………………………

6) What is a working set and what happens when the working set is very different from the set of pages that are physically resident in memory?

   ………………………………………………………………………………

   ………………………………………………………………………………

2.8 SUMMARY

With previous schemes, all the code and data of a program have to be in main memory when the program is running. With virtual memory, only some of the code and data have to be in main memory: the parts needed by the program now. The other parts are loaded into memory when the program needs them without the program having to be aware of this. The size of a program (including its data) can thus exceed the amount of available main memory.

There are two main approaches to virtual memory: paging and segmentation. Both approaches rely on the separation of the concepts virtual address and physical address. Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory management unit (MMU) translates virtual addresses to physical addresses at run-time.

In this unit we have discussed the concept of Virtual memory, its advantages, demand paging, demand segmentation, Page replacement algorithms and combined systems.

2.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) The list of steps that are followed by the operating system in handling a page fault:

   a) If a process refers to a page which is not in the physical memory then an internal table kept with a process control block is checked to verify whether a memory reference to a page was valid or invalid.

   b) If the memory reference to a page was valid, but the page is missing, the process of bringing a page into the physical memory starts.

   c) Free memory location is identified to bring a missing page.

   d) By reading a disk, the desired page is brought back into the free memory location.

   e) Once the page is in the physical memory, the internal table kept with the process and page map table is updated to indicate that the page is now in memory.

   f) Restart the instruction that was interrupted due to the missing page.

2) In demand paging pages are loaded only on demand, not in advance. It is similar to paging system with swapping feature. Rather than swapping the entire program in memory, only those pages are swapped which are required currently by the system.

3) Virtual memory can be implemented as an extension of paged or segmented memory management scheme, also called **demand** paging or **demand segmentation** respectively.

4) A Page Fault occurs when a process accesses an address in a page which is not currently resident in memory. Thrashing occurs when the incidence of page faults becomes so high that the system spends all its time swapping pages rather than doing useful work. This happens if the number of page frames allocated is insufficient to contain the working set for the process either because there is inadequate physical memory or because the program is very badly organised.

5) A page swap algorithm should

   a) Impose the minimum overhead in operation

   b) Not assume any particular hardware assistance

   c) Try to prevent swapping out pages which are currently referenced (since they may still be in the Working Set)

   d) Try to avoid swapping modified pages (to avoid the overhead of rewriting pages from memory to swap area).

6) The working Set is the set of pages that a process accesses over any given (short) space of time. If the Working Set is very different from the set of pages that are physically resident in memory, there will be an excessive number of page faults and thrashing will occur.

2.10 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.

2) H.M.Deitel, *Operating Systems*, Pearson Education Asia, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.

4) Achyut S. Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.

5) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.

6) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.

7) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.

UNIT 3 I/O AND FILE MANAGEMENT

Structure **Page Nos.**

3.0 INTRODUCTION

Input and output devices are components that form part of the computer system. These devices are controlled by the operating system. Input devices such as keyboard, mouse, and sensors provide input signals such as commands to the operating system. These commands received from input devices instruct the operating system to perform some task or control its behaviour. Output devices such as monitors, printers and speakers are the devices that receive commands or information from the operating system.

In the earlier unit, we had studied the memory management of primary memory. The physical memory, as we have already seen, is not large enough to accommodate all of the needs of a computer system. Also, it is not permanent. Secondary storage consists of disk units and tape drives onto which data can be moved for permanent storage. Though there are actual physical differences between tapes and disks, the principles involved in controlling them are the same, so we shall only consider disk management here in this unit.

The operating system implements the abstract concept of the file by managing mass storage devices, such as types and disks. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage Unit, the **file**. Files are mapped by the operating system, onto physical devices.

Definition: A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user.

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms among which magnetic tape, disk, and drum are the most common forms. Each of these devices has their own characteristics and physical organisation.

Normally files are organised into directories to ease their use. When multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. The operating system is responsible for the following activities in connection with file management:

- The creation and deletion of files.
- The creation and deletion of directory.
- The support of primitives for manipulating files and directories.
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.

The most significant problem in I/O system is the speed mismatch between I/O devices and the memory and also with the processor. This is because I/O system involves both H/W and S/W support and there is large variation in the nature of I/O devices, so they cannot compete with the speed of the processor and memory.

A well-designed file management structure makes the file access quick and easily movable to a new machine. Also it facilitates sharing of files and protection of non-public files. For security and privacy, file system may also provide encryption and decryption capabilities. This makes information accessible to the intended user only.

In this unit we will study the I/O and the file management techniques used by the operating system in order to manage them efficiently.

3.1 OBJECTIVES

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

- describe the management of the I/O activities independently and simultaneously with processor activities;
- summarise the full range of views that support file systems, especially the operating system view;
- compare and contrast different approaches to file organisations;
- discuss the disk scheduling techniques, and
- know how to implement the file system and its protection against unauthorised usage.

3.2 ORGANISATION OF THE I/O FUNCTION

The range of I/O devices and the large variation in their nature, speed, design, functioning, usage etc. makes it difficult for the operating system to handle them with any generality. The key concept in I/O software designing is device independence achieved by using uniform naming.

The name of the file or device should simply be a string or an integer and not dependent on the device in any way. Another key issue is sharable versus dedicated devices. Many users can share some I/O devices such as disks, at any instance of time. The devices like printers have to be dedicated to a single user until that user has finished an operation.

The basic idea is to organise the I/O software as a series of layers with the lower ones hiding the physical H/W and other complexities from the upper ones that present simple, regular interface interaction with users. Based on this I/O software can be structured in the following four layers given below with brief descriptions:

(i) **Interrupt handlers**: The CPU starts the transfer and goes off to do something else until the interrupt arrives. The I/O device performs its activity independently and simultaneously with CPU activity. This enables the I/O devices to run asynchronously with the processor. The device sends an interrupt to the processor when it has completed the task, enabling CPU to initiate a further task. These interrupts can be hidden with the help of device drivers discussed below as the next I/O software layer.

(ii) **Device Drivers**: Each device driver handles one device type or group of closely related devices and contains a device dependent code. It accepts individual requests from device independent software and checks that the request is carried out. A device driver manages communication with a specific I/O device by converting a logical request from a user into specific commands directed to the device.

(iii) **Device-independent Operating System Software**: Its basic responsibility is to perform the I/O functions common to all devices and to provide interface with user-level software. It also takes care of mapping symbolic device names onto the proper driver. This layer supports device naming, device protection, error reporting, allocating and releasing dedicated devices etc.

(iv) **User level software**: It consists of library procedures linked together with user programs. These libraries make system calls. It makes I/O call, format I/O and also support spooling. Spooling is a way of dealing with dedicated I/O devices like printers in a multiprogramming environment.

3.3 I/O BUFFERING

A buffer is an intermediate memory area under operating system control that stores data in transit between two devices or between user's work area and device. It allows computation to proceed in parallel with I/O.

In a typical unbuffered transfer situation the processor is idle for most of the time, waiting for data transfer to complete and total read-processing time is the sum of all the transfer/read time and processor time as shown in *Figure 1*.

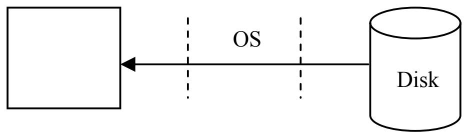


Figure 1: Unbuffered Transfers

In case of single-buffered transfer, blocks are first read into a buffer and then moved to the user's work area. When the move is complete, the next block is read into the buffer and processed in parallel with the first block. This helps in minimizing speed mismatch between devices and the processor. Also, this allows process computation in parallel with input/output as shown in *Figure 2*.

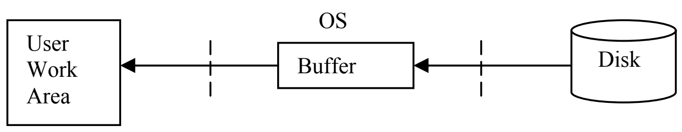


Figure 2: Single Buffering

Double buffering is an improvement over this. A pair of buffers is used; blocks/records generated by a running process are initially stored in the first buffer until it is full. Then from this buffer it is transferred to the secondary storage. During this transfer the other blocks generated are deposited in the second buffer and when this second buffer is also full and first buffer transfer is complete, then transfer from the second buffer is initiated. This process of alternation between buffers continues which allows I/O to occur in parallel with a process's computation. This scheme increases the complexity but yields improved performance as shown in *Figure 3*.

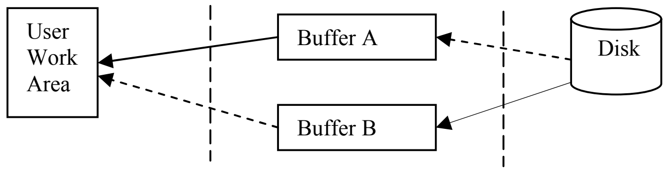


Figure 3: Double Buffering

3.4 DISK ORGANISATION

Disks come in different shapes and sizes. The most obvious distinction between floppy disks, diskettes and hard disks is: floppy disks and diskettes consist, of a single disk of magnetic material, while hard-disks normally consist of several stacked on top of one another. Hard disks are totally enclosed devices which are much more finely engineered and therefore require protection from dust. A hard disk spins at a constant speed, while the rotation of floppy drives is switched on and off. On the Macintosh machine, floppy drives have a variable speed operation, whereas most floppy drives have only a single speed of rotation. As hard drives and tape units become more efficient and cheaper to produce, the role of the floppy disk is diminishing. We look therefore mainly at hard drives.

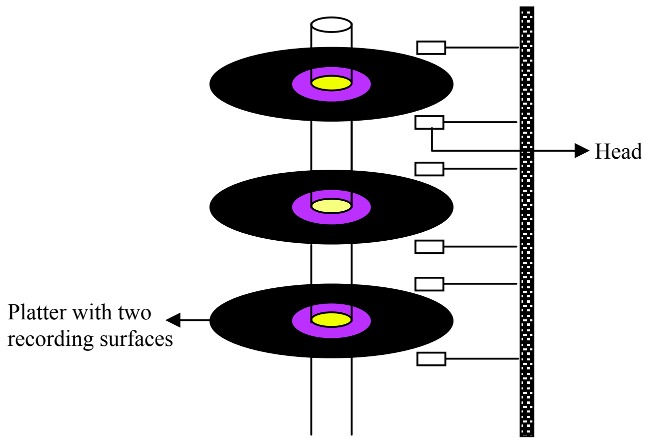


Figure 4: Hard Disk with 3 platters

Looking at the *Figure 4*, we see that a hard disk is composed of several physical disks stacked on top of each other. The disk shown in the *Figure 4* has 3 platters and 6 recording surfaces (two on each platter). A separate read *head* is provided for each *surface*. Although the disks are made of continuous magnetic material, there is a limit to the *density* of information which can be stored on the disk. The heads are controlled by a *stepper motor* which moves them in fixed-distance intervals across each surface. i.e., there is a fixed number of *tracks* on each surface. The tracks on all the surfaces are aligned, and the sum of all the tracks at a fixed distance from the edge of the disk is called a *cylinder*. To make the disk access quicker, tracks are usually divided up into *sectors* – or fixed size regions which lie along tracks. When writing to a disk, data are written in units of a whole number of sectors. (In this respect, they are similar to pages or frames in physical memory). On some disks, the sizes of sectors are decided by the manufacturers in hardware. On other systems (often microcomputers) it might be chosen in software when the disk is prepared for use (*formatting*). Because the heads of the disk move together on all surfaces, we can increase read-write efficiency by allocating blocks in parallel across all surfaces. Thus, if a file is stored in consecutive blocks, on a disk with *n* surfaces and *n* heads, it could read *n* sectors *per-track* without any head movement. When a disk is supplied by a manufacturer, the physical properties of the disk (number of tracks, number of heads, sectors per track, speed of revolution) are provided with the disk. An operating system must be able to adjust to different types of disk. Clearly *sectors per track* is not a constant, nor is the number of tracks. The numbers given are just a convention used to work out a consistent set of addresses on a disk and may not have anything to do with the hard and fast physical limits of the disk. To address any portion of a disk, we need a three component address consisting of (*surface, track and sector*).

The *seek time* is the time required for the disk arm to move the head to the cylinder with the desired sector. The *rotational latency* is the time required for the disk to rotate the desired sector until it is under the read-write head.

3.4.1 Device drivers and IDs

A hard-disk is a *device*, and as such, an operating system must use a *device controller* to talk to it. Some device controllers are simple microprocessors which translate numerical addresses into head motor movements, while others contain small decision making computers of their own. The most popular type of drive for larger personal computers and workstations is the SCSI drive. SCSI (pronounced scuzzy) (Small Computer System Interface) is a protocol and now exists in four variants SCSI 1, SCSI 2, fast SCSI 2, and SCSI 3. SCSI disks live on a *data bus* which is a fast parallel data link to the CPU and memory, rather like a very short network. Each drive coupled to the bus identifies itself by a SCSI address and each SCSI controller can address up to seven units. If more disks are required, a second controller must be added. SCSI is more efficient at multiple accesses sharing than other disk types for microcomputers. In order to talk to a SCSI disk, an operating system must have a SCSI device driver. This is a layer of software which translates disk requests from the operating system's abstract command-layer into the language of signals which the SCSI controller understands.

3.4.2 Checking Data Consistency and Formatting

Hard drives are not perfect: they develop defects due to magnetic dropout and imperfect manufacturing. On more primitive disks, this is checked when the disk is *formatted* and these damaged sectors are avoided. If the sector becomes damaged under operation, the structure of the disk must be patched up by some repair program. Usually the data are lost.

On more intelligent drives, like the SCSI drives, the disk itself keeps a *defect list* which contains a list of all bad sectors. A new disk from the manufacturer contains a starting list and this is updated as time goes by, if more defects occur. Formatting is a process by which the sectors of the disk are:

- (If necessary) created by setting out 'signposts' along the tracks,
- Labelled with an address, so that the disk controller knows when it has found the correct sector.

On simple disks used by microcomputers, formatting is done manually. On other types, like SCSI drives, there is a low-level formatting already on the disk when it comes from the manufacturer. This is part of the SCSI protocol, in a sense. High level formatting on top of this is not necessary, since an advanced enough *filesystem* will be able to manage the hardware sectors. *Data consistency* is checked by writing to disk and reading back the result. If there is disagreement, an error occurs. This procedure can best be implemented inside the hardware of the disk−modern disk drives are small computers in their own right. Another cheaper way of checking data consistency is to calculate a number for each sector, based on what data are in the sector and store it in the sector. When the data are read back, the number is recalculated and if there is disagreement then an error is signalled. This is called a *cyclic redundancy check* (CRC) or *error correcting code*. Some device controllers are intelligent enough to be able to detect bad sectors and move data to a spare 'good' sector if there is an error. Disk design is still a subject of considerable research and disks are improving both in speed and reliability by leaps and bounds.

3.5 DISK SCHEDULING

The disk is a resource which has to be shared. It is therefore has to be scheduled for use, according to some kind of scheduling system. The secondary storage media structure is one of the vital parts of the file system. Disks are the one, providing lot of the secondary storage. As compared to magnetic tapes, disks have very fast access time and disk bandwidth. The access time has two major constituents: seek time and the rotational latency.

The *seek time* is the time required for the disk arm to move the head to the cylinder with the desired sector. The *rotational latency* is the time required for the disk to rotate the desired sector until it is under the read-write head. The *disk bandwidth* is the total number of bytes transferred per unit time.

Both the access time and the bandwidth can be improved by efficient disk I/O requests scheduling. Disk drivers are large single dimensional arrays of logical blocks to be transferred. Because of large usage of disks, proper scheduling algorithms are required.

A scheduling policy should attempt to maximize throughput (defined as the number of requests serviced per unit time) and also to minimize mean response time (i.e., average waiting time plus service time). These scheduling algorithms are discussed below:

3.5.1 FCFS Scheduling

First-Come, First-Served (FCFS) is the basis of this simplest disk scheduling technique. There is no reordering of the queue. Suppose the requests for inputting/ outputting to blocks on the cylinders have arrived, forming the following disk queue:

50, 91, 150, 42, 130, 18, 140, 70, 60

Also assume that the disk head is initially at cylinder 50 then it moves to 91, then to 150 and so on. The total head movement in this scheme is 610 cylinders, which makes the system slow because of wild swings. Proper scheduling while moving towards a particular direction could decrease this. This will further improve performance. FCFS scheme is clearly depicted in *Figure 5*.

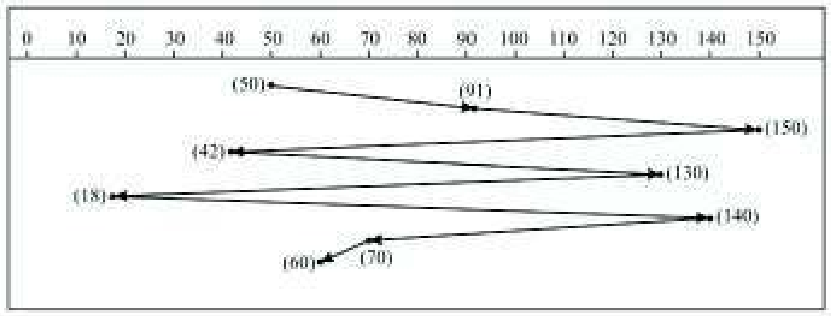


Figure 5: FCFS Scheduling

3.5.2 SSTF Scheduling

The basis for this algorithm is Shortest-Seek-Time-First (SSTF) i.e., service all the requests close to the current head position and with minimum seeks time from current head position.

In the previous disk queue sample the cylinder close to critical head position i.e., 50, is 42 cylinder, next closest request is at 60. From there, the closest one is 70, then 91,130,140,150 and then finally 18-cylinder. This scheme has reduced the total head movements to 248 cylinders and hence improved the performance. Like SJF (Shortest Job First) for CPU scheduling SSTF also suffers from starvation problem. This is because requests may arrive at any time. Suppose we have the requests in disk queue for cylinders 18 and 150, and while servicing the 18-cylinder request, some other request closest to it arrives and it will be serviced next. This can continue further also making the request at 150-cylinder wait for long. Thus a continual stream of requests near one another could arrive and keep the far away request waiting indefinitely. The SSTF is not the optimal scheduling due to the starvation problem. This whole scheduling is shown in *Figure 6*.

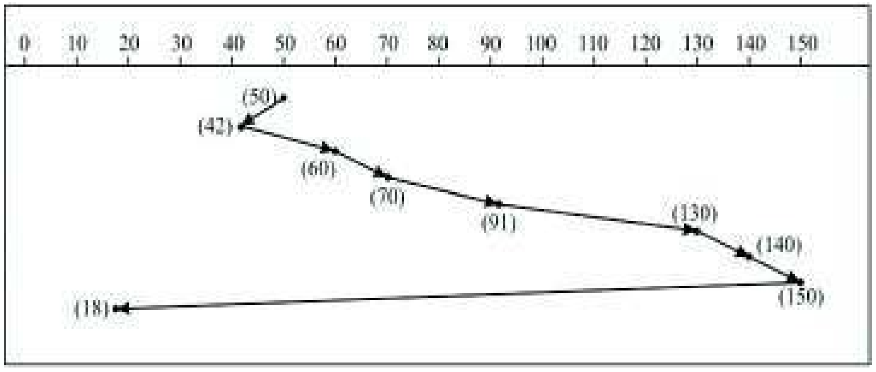


Figure 6: SSTF Scheduling

3.5.3 SCAN Scheduling

The disk arm starts at one end of the disk and service all the requests in its way towards the other end, i.e., until it reaches the other end of the disk where the head movement is reversed and continue servicing in this reverse direction. This scanning is done back and forth by the head continuously.

In the example problem two things must be known before starting the scanning process. Firstly, the initial head position i.e., 50 and then the head movement direction (let it towards 0, starting cylinder). Consider the disk queue again:

91, 150, 42, 130, 18, 140, 70, 60

Starting from 50 it will move towards 0, servicing requests 42 and 18 in between. At cylinder 0 the direction is reversed and the arm will move towards the other end of the disk servicing the requests at 60, 70, 91,130,140 and then finally 150.

As the arm acts like an elevator in a building, the SCAN algorithm is also known as elevator algorithm sometimes. The limitation of this scheme is that few requests need to wait for a long time because of reversal of head direction. This scheduling algorithm results in a total head movement of only 200 cylinders. *Figure 7* shows this scheme:

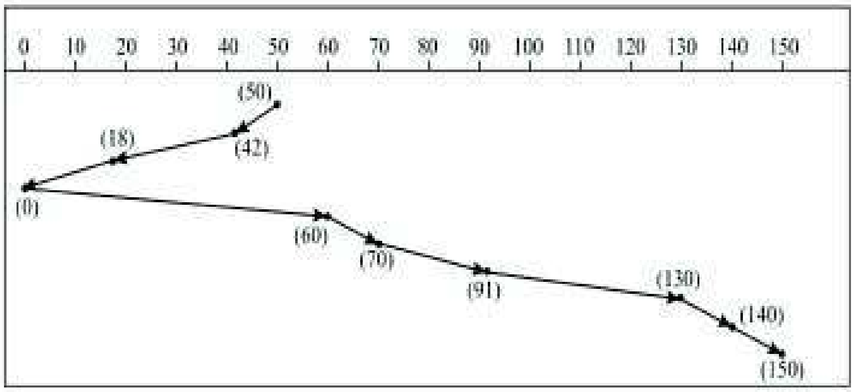


Figure 7: SCAN Scheduling

3.5.4 C-SCAN Scheduling

Similar to SCAN algorithm, C-SCAN also moves head from one end to the other servicing all the requests in its way. The difference here is that after the head reaches the end it immediately returns to beginning, skipping all the requests on the return trip. The servicing of the requests is done only along one path. Thus comparatively this scheme gives uniform wait time because cylinders are like circular lists that wrap around from the last cylinder to the first one.

3.5.5 LOOK and C-LOOK Scheduling

These are just improvements of SCAN and C-SCAN but difficult to implement. Here the head moves only till final request in each direction (first and last ones), and immediately reverses direction without going to end of the disk. Before moving towards any direction the requests are looked, avoiding the full width disk movement by the arm.

The performance and choice of all these scheduling algorithms depend heavily on the number and type of requests and on the nature of disk usage. The file allocation methods like contiguous, linked or indexed, also affect the requests. For example, a contiguously allocated file will generate nearby requests and hence reduce head movements whereas linked or indexed files may generate requests from blocks that are scattered throughout the disk and hence increase the head movements. While searching for files the directories will be frequently accessed, hence location of directories and also blocks of data in them are also important criteria. All these peculiarities force the disk scheduling algorithms to be written as a separate module of the operating system, so that these can easily be replaced. For heavily used disks the SCAN / LOOK algorithms are well suited because they take care of the hardware and access requests in a reasonable order. There is no real danger of starvation, especially in the C-SCAN case. The arrangement of data on a disk plays an important role in deciding the efficiency of data-retrieval.

3.6 RAID

Disks have high failure rates and hence there is the risk of loss of data and lots of downtime for restoring and disk replacement. To improve disk usage many techniques have been implemented. One such technology is RAID (Redundant Array of Inexpensive Disks). Its organisation is based on disk striping (or interleaving), which uses a group of disks as one storage unit. Disk striping is a way of increasing the disk transfer rate up to a factor of *N*, by splitting files across *N* different disks. Instead of saving all the data from a given file on one disk, it is split across many. Since the *N* heads can now search independently, the speed of transfer is, in principle, increased manifold. Logical disk data/blocks can be written on two or more separate physical disks which can further transfer their sub-blocks in parallel. The total transfer rate system is directly proportional to the number of disks. The larger the number of physical disks striped together, the larger the total transfer rate of the system. Hence, the overall performance and disk accessing speed is also enhanced. The enhanced version of this scheme is mirroring or shadowing. In this RAID organisation a duplicate copy of each disk is kept. It is costly but a much faster and more reliable approach. The disadvantage with disk striping is that, if one of the *N* disks becomes damaged, then the data on all *N* disks is lost. Thus striping needs to be combined with a reliable form of backup in order to be successful.

Another RAID scheme uses some disk space for holding parity blocks. Suppose, three or more disks are used, then one of the disks will act as a parity block, which contains corresponding bit positions in all blocks. In case some error occurs or the disk develops a problems all its data bits can be reconstructed. This technique is known as disk striping with parity or block interleaved parity, which increases speed. But writing or updating any data on a disk requires corresponding recalculations and changes in parity block. To overcome this the parity blocks can be distributed over all disks.

3.7 DISK CACHE

Disk caching is an extension of buffering. Cache is derived from the French word cacher, meaning to hide. In this context, a ***cache*** is a collection of blocks that logically belong on the disk, but are kept in memory for performance reasons. It is used in multiprogramming environment or in disk file servers, which maintain a separate section of main memory called disk cache. These are sets of buffers (cache) that contain the blocks that are recently used. The cached buffers in memory are copies of the disk blocks and if any data here is modified only its local copy is updated. So, to maintain integrity, updated blocks must be transferred back to the disk. Caching is based on the assumption that most shortly accessed blocks are likely to be accessed again soon. In case some new block is required in the cache buffer, one block already there can be selected for "flushing" to the disk. Also to avoid loss of updated information in case of failures or loss of power, the system can periodically flush cache blocks to the disk. The key to disk caching is keeping frequently accessed records in the disk cache buffer in primary storage.

☞ **Check Your Progress 1**

1) Indicate the major characteristics which differentiate I/O devices.

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

2) Explain the term device independence. What is the role of device drivers in this context?

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

3) Describe the ways in which a device driver can be implemented?

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

4) What is the advantage of the double buffering scheme over single buffering?

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

5) What are the key objectives of the I/O system?

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

3.8 COMMAND LANGUAGE USER'S VIEW OF THE FILE SYSTEM

The most important aspect of a file system is its appearance from the user's point of view. The user prefers to view the naming scheme of files, the constituents of a file, what the directory tree looks like, protection specifications, file operations allowed on them and many other interface issues. The internal details like, the data structure used for free storage management, number of sectors in a logical block etc. are of less interest to the users. From a user's perspective, a file is the smallest allotment of logical secondary storage. Users should be able to refer to their files by symbolic names rather than having to use physical device names.

The operating system allows users to define named objects called files which can hold interrelated data, programs or any other thing that the user wants to store/save.

3.9 THE SYSTEM PROGRAMMER'S VIEW OF THE FILE SYSTEM

As discussed earlier, the system programmer's and designer's view of the file system is mainly concerned with the details/issues like whether linked lists or simple arrays are used for keeping track of free storage and also the number of sectors useful in any logical block. But it is rare that physical record size will exactly match the length of desired logical record. The designers are mainly interested in seeing how disk space is managed, how files are stored and how to make everything work efficiently and reliably.

3.10 THE OPERATING SYSTEM'S VIEW OF FILE MANAGEMENT

As discussed earlier, the operating system abstracts (maps) from the physical properties of its storage devices to define a logical storage unit i.e., the file. The operating system provides various system calls for file management like creating, deleting files, read and write, truncate operations etc. All operating systems focus on achieving device-independence by making the access easy regardless of the place of storage (file or device). The files are mapped by the operating system onto physical devices. Many factors are considered for file management by the operating system like directory structure, disk scheduling and management, file related system services, input/output etc. Most operating systems take a different approach to storing information. Three common file organisations are byte sequence, record sequence and tree of disk blocks. UNIX files are structured in simple byte sequence form. In record sequence, arbitrary records can be read or written, but a record cannot be inserted or deleted in the middle of a file. CP/M works according to this scheme. In tree organisation each block hold ***n*** keyed records and a new record can be inserted anywhere in the tree. The mainframes use this approach. The OS is responsible for the following activities in regard to the file system:

- The creation and deletion of files
- The creation and deletion of directory
- The support of system calls for files and directories manipulation
- The mapping of files onto disk
- Backup of files on stable storage media (non-volatile).

The coming sub-sections cover these details as viewed by the operating system.

3.10.1 Directories

A file directory is a group of files organised together. An entry within a directory refers to the file or another directory. Hence, a tree structure/hierarchy can be formed. The directories are used to group files belonging to different applications/users. Large-scale time sharing systems and distributed systems store thousands of files and bulk of data. For this type of environment a file system must be organised properly. A File system can be broken into partitions or volumes. They provide separate areas within one disk, each treated as separate storage devices in which files and directories reside. Thus directories enable files to be separated on the basis of user and user applications, thus simplifying system management issues like backups, recovery, security, integrity, name-collision problem (file name clashes), housekeeping of files etc.

The device directory records information like name, location, size, and type for all the files on partition. A root refers to the part of the disk from where the root directory begins, which points to the user directories. The root directory is distinct from sub-directories in that it is in a fixed position and of fixed size. So, the directory is like a symbol table that converts file names into their corresponding directory entries. The operations performed on a directory or file system are:

1) Create, delete and modify files.
2) Search for a file.
3) Mechanisms for sharing files should provide controlled access like read, write, execute or various combinations.
4) List the files in a directory and also contents of the directory entry.
5) Renaming a file when its contents or uses change or file position needs to be changed.
6) Backup and recovery capabilities must be provided to prevent accidental loss or malicious destruction of information.
7) Traverse the file system.

The most common schemes for describing logical directory structure are:

(i) **Single-level directory**

All the files are inside the same directory, which is simple and easy to understand; but the limitation is that all files must have unique names. Also, even with a single user as the number of files increases, it is difficult to remember and to track the names of all the files. This hierarchy is depicted in *Figure 8.*

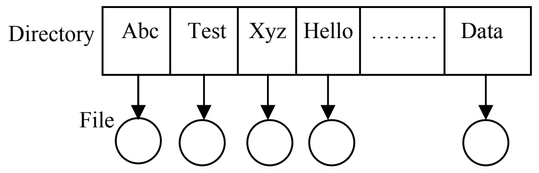


Figure 8: Single-level directory

(ii) **Two-level directory**

We can overcome the limitations of the previous scheme by creating a separate directory for each user, called User File Directory (UFD). Initially when the user logs in, the system's Master File Directory (MFD) is searched which is indexed with respect to username/account and UFD reference for that user. Thus different users may have same file names but within each UFD they should be unique. This resolves name-collision problem up to some extent but this directory structure isolates one user from another, which is not desired sometimes when users need to share or cooperate on some task. *Figure 9* shows this scheme clearly.

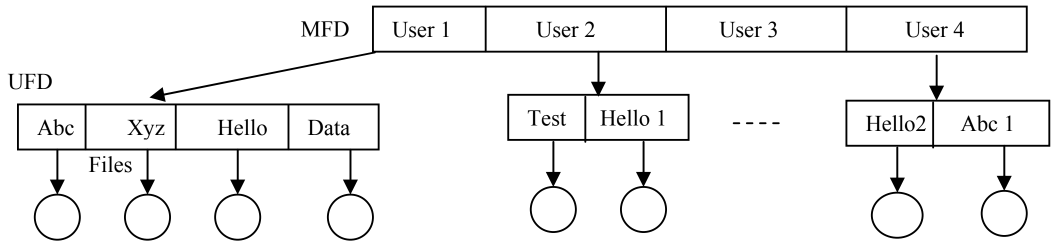


Figure 9: Two-level directory

(iii) **Tree-structured directory**

The two-level directory structure is like a 2-level tree. Thus to generalise, we can extend the directory structure to a tree of arbitrary height. Thus the user can create his/her own directory and subdirectories and can also organise files. One bit in each directory entry defines entry as a file (0) or as a subdirectory (1).

The tree has a root directory and every file in it has a unique path name (path from root, through all subdirectories, to a specified file). The pathname prefixes the filename, helps to reach the required file traversed from a base directory. The pathnames can be of 2 types: absolute path names or relative path names, depending on the base directory. An absolute path name begins at the root and follows a path to a particular file. It is a full pathname and uses the root directory. Relative defines the path from the current directory. For example, if we assume that the current directory is */Hello2* then the file *F4.doc* has the absolute pathname */Hello/Hello2/Test2/F4.doc* and the relative pathname is */Test2/F4.doc*. The pathname is used to simplify the searching of a file in a tree-structured directory hierarchy. *Figure 10* shows the layout:

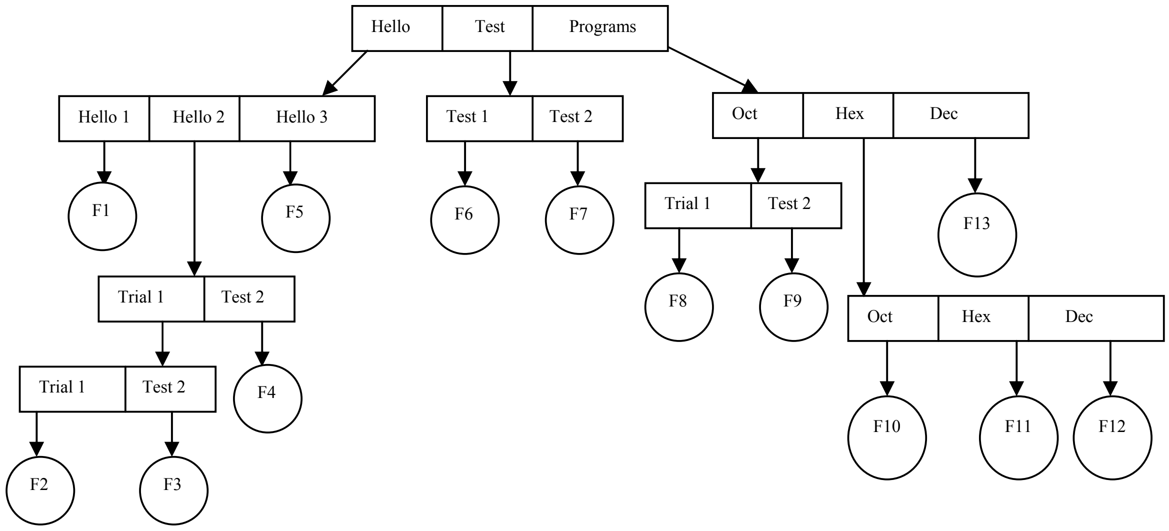


Figure 10: Tree-structured directory

(iv) **Acyclic-graph directory**:

As the name suggests, this scheme has a graph with no cycles allowed in it. This scheme added the concept of shared common subdirectory / file which exists in file system in two (or more) places at once. Having two copies of a file does not reflect changes in one copy corresponding to changes made in the other copy.

But in a shared file, only one actual file is used and hence changes are visible. Thus an acyclic graph is a generalisation of a tree-structured directory scheme. This is useful in a situation where several people are working as a team, and need access to shared files kept together in one directory. This scheme can be implemented by creating a new directory entry known as a link which points to another file or subdirectory. *Figure 11* depicts this structure for directories.

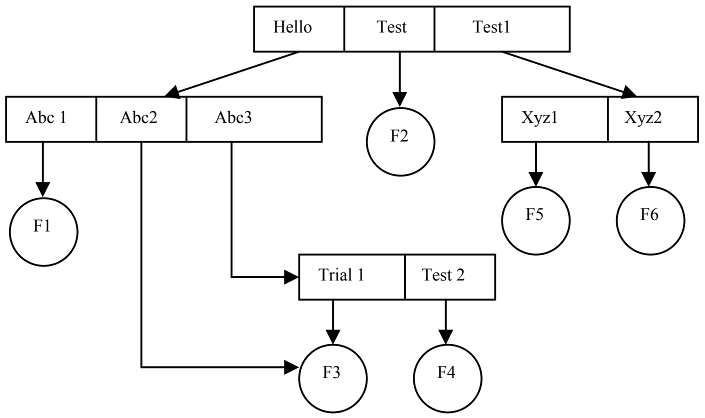


Figure 11: Acyclic-graph directory

The limitations of this approach are the difficulties in traversing an entire file system because of multiple absolute path names. Another issue is the presence of dangling pointers to the files that are already deleted, though we can overcome this by preserving the file until all references to it are deleted. For this, every time a link or a copy of directory is established, a new entry is added to the file-reference list. But in reality as the list is too lengthy, only a count of the number of references is kept. This count is then incremented or decremented when the reference to the file is added or it is deleted respectively.

(v) **General graph Directory**:

Acyclic-graph does not allow cycles in it. However, when cycles exist, the reference count may be non-zero, even when the directory or file is not referenced anymore. In such situation garbage collection is useful. This scheme requires the traversal of the whole file system and marking accessible entries only. The second pass then collects everything that is unmarked on a free-space list. This is depicted in *Figure 12*.

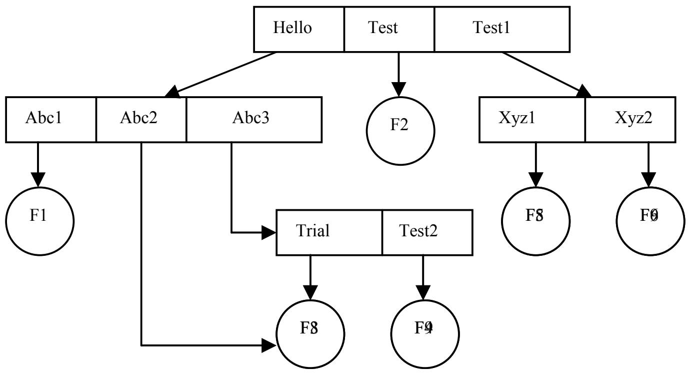


Figure 12: General-graph directory

3.10.2 Disk Space Management

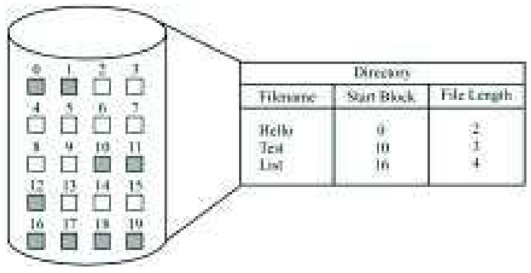
The direct-access of disks and keeping files in adjacent areas of the disk is highly desirable**.** But the problem is how to allocate space to files for effective disk space utilisation and quick access. Also, as files are allocated and freed, the space on a disk become fragmented. The major methods of allocating disk space are:

i) Continuous
ii) Non-continuous (Indexing and Chaining)

i) **Continuous**

This is also known as contiguous allocation as each file in this scheme occupies a set of contiguous blocks on the disk. A linear ordering of disk addresses is seen on the disk. It is used in VM/CMS– an old interactive system. The advantage of this approach is that successive logical records are physically adjacent and require no head movement. So disk seek time is minimal and speeds up access of records. Also, this scheme is relatively simple to implement. The technique in which the operating system provides units of file space on demand by user running processes, is known as dynamic allocation of disk space. Generally space is allocated in units of a fixed size, called an allocation unit or a ‘cluster’ in MS-DOS. Cluster is a simple multiple of the disk physical sector size, usually 512 bytes. Disk space can be considered as a one-dimensional array of data stores, each store being a cluster. A larger cluster size reduces the potential for fragmentation, but increases the likelihood that clusters will have unused space. Using clusters larger than one sector reduces fragmentation, and reduces the amount of disk space needed to store the information about the used and unused areas on the disk.

Contiguous allocation merely retains the disk address (start of file) and length (in block units) of the first block. If a file is n blocks long and it begins with location b (blocks), then it occupies b, b+1, b+2,…, b+n-1 blocks. First-fit and best-fit strategies can be used to select a free hole from available ones. But the major problem here is searching for sufficient space for a new file. *Figure 13* depicts this scheme:



Directory		
Filename	Start Block	File Length
Hello	0	2
Test	10	3
List	16	4

Figure 13: Contiguous Allocation on the Disk

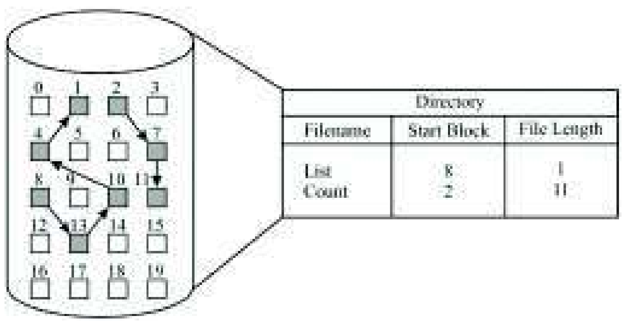
This scheme exhibits similar fragmentation problems as in variable memory partitioning. This is because allocation and deal location could result in regions of free disk space broken into chunks (pieces) within active space, which is called external fragmentation. A repacking routine called compaction can solve this problem. In this routine, an entire file system is copied on to tape or another disk and the original disk is then freed completely. Then from the copied disk, files are again stored back using contiguous space on the original disk. But this approach can be very expensive in terms of time. Also, size-declaration in advance is a problem because each time, the size of file is not predictable. But it supports both sequential and direct accessing. For sequential access, almost no seeks are required. Even direct access with seek and read is fast. Also, calculation of blocks holding data is quick and easy as we need just offset from the start of the file.

ii) Non-Continuous (Chaining and Indexing)

This scheme has replaced the previous ones. The popular non-contiguous storages allocation schemes are:

- Linked/Chained allocation
- Indexed Allocation.

Linked/Chained allocation: All files are stored in fixed size blocks and adjacent blocks are linked together like a linked list. The disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last block of the file. Also each block contains pointers to the next block, which are not made available to the user. There is no external fragmentation in this as any free block can be utilised for storage. So, compaction and relocation is not required. But the disadvantage here is that it is potentially inefficient for direct-accessible files since blocks are scattered over the disk and have to follow pointers from one disk block to the next. It can be used effectively for sequential access only but there also it may generate long seeks between blocks. Another issue is the extra storage space required for pointers. Yet the reliability problem is also there due to loss/damage of any pointer. The use of doubly linked lists could be a solution to this problem but it would add more overheads for each file. A doubly linked list also facilitates searching as blocks are threaded both forward and backward. The *Figure 14* depicts linked /chained allocation where each block contains the information about the next block (i.e., pointer to next block).



Directory		
Filename	Start Block	File Length
List	8	1
Count	2	11

Figure 14: Linked Allocation on the Disk

MS-DOS and OS/2 use another variation on linked list called FAT (File Allocation Table). The beginning of each partition contains a table having one entry for each disk block and is indexed by the block number. The directory entry contains the block number of the first block of file. The table entry indexed by block number contains the block number of the next block in the file. The Table pointer of the last block in the file has EOF pointer value. This chain continues until EOF (end of file) table entry is encountered. We still have to linearly traverse next pointers, but at least we don't have to go to the disk for each of them. 0(Zero) table value indicates an unused block. So, allocation of free blocks with FAT scheme is straightforward, just search for the first block with 0 table pointer. MS-DOS and OS/2 use this scheme. This scheme is depicted in *Figure 15.*

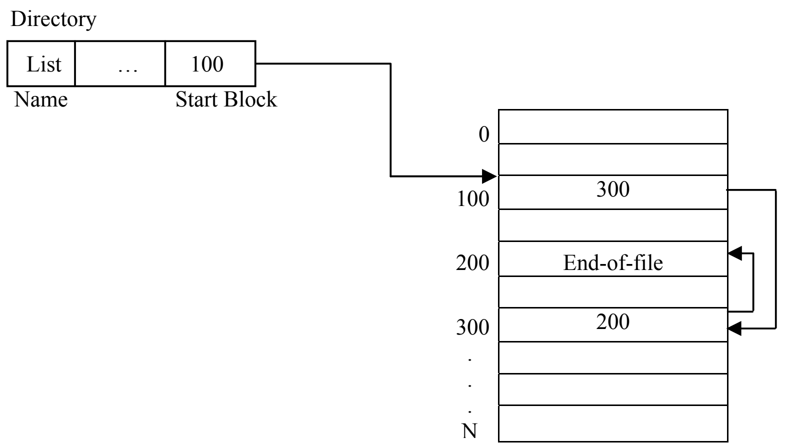


Figure 15: File-Allocation Table (FAT)

Indexed Allocation: In this each file has its own index block. Each entry of the index points to the disk blocks containing actual file data i.e., the index keeps an array of block pointers for each file. So, index block is an array of disk block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. Also, the main directory contains the address where the index block is on the disk. Initially, all the pointers in index block are set to ***NIL***. The advantage of this scheme is that it supports both sequential and random access. The searching may take place in index blocks themselves. The index blocks may be kept close together in secondary storage to minimize seek time. Also space is wasted only on the index which is not very large and there's no external fragmentation. But a few limitations of the previous scheme still exists in this, like, we still need to set maximum file length and we can have overflow scheme of the file larger than the predicted value. Insertions can require complete reconstruction of index blocks also. The indexed allocation scheme is diagrammatically shown in *Figure 16.*

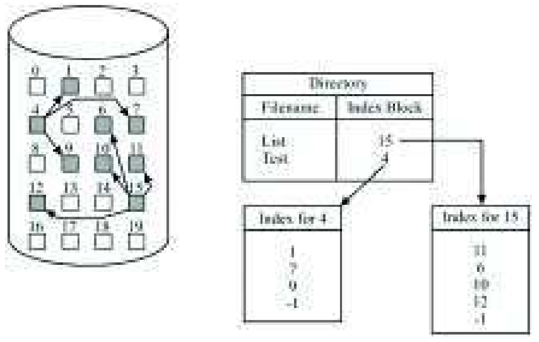


Figure 16: Indexed Allocation on the Disk

3.10.3 Disk Address Translation

We have seen in Unit-1 memory management that the virtual addresses generated by a program is different from the physical. The translation of virtual addresses to physical addresses is performed by MMU. Disk address translation considers the aspects of data storage on the disk. Hard disks are totally enclosed devices, which are more finely engineered and therefore require protection from dust. A hard disk spins at a constant speed. Briefly, hard disks consist of one or more rotating platters. A read-write head is positioned above the rotating surface and is able to read or write the data underneath the current head position. The hard drives are able to present the "geometry" or "addressing scheme" of the drive. Consider the disk internals first. Each track of the disk is divided into sections called sectors. A sector is the smallest physical storage unit on the disk. The size of a sector is always a power of two, and is almost always 512 bytes. A sector is the part of a slice that lies within a track. The position of the head determines which track is being read. A cylinder is almost the same as a track, except that it means all tracks lining up under each other on all the surfaces. The head is equivalent to side(s). It simply means one of the rotating platters or one of the sides on one of the platters. If a hard disk has three rotating platters, it usually has 5 or 6 readable sides, each with its own read-write head.

The MS-DOS file systems allocate storage in clusters, where a cluster is one or more contiguous sectors. MS-DOS bases the cluster size on the size of the partition. As a file is written on the disk, the file system allocates the appropriate number of clusters to store the file's data. For the purposes of isolating special areas of the disk, most operating systems allow the disk surface to be divided into partitions. A partition (also called a cylinder group) is just that: a group of cylinders, which lie next to each other. By defining partitions we divide up the storage of data to special areas, for convenience. Each partition is assigned a separate logical device and each device can only write to the cylinders, which are defined as being its own. To access the disk the computer needs to convert physical disk geometry (the number of cylinders on the disk, number of heads per cylinder, and sectors per track) to a logical configuration that is compatible with the operating system. This conversion is called translation. Since sector translation works between the disk itself and the system BIOS or firmware, the operating system is unaware of the actual characteristics of the disk, if the number of cylinders, heads, and sectors per track the computer needs is within the range supported by the disk. MS-DOS presents disk devices as logical volumes that are associated with a drive code (A, B, C, and so on) and have a volume name (optional), a root directory, and from zero to many additional directories and files.

3.10.4 File Related System Services

A file system enables applications to store and retrieve files on storage devices. Files are placed in a hierarchical structure. The file system specifies naming conventions for files and the format for specifying the path to a file in the tree structure. OS provides the ability to perform input and output (I/O) operations on storage components located on local and remote computers. In this section we briefly describe the system services, which relate to file management. We can broadly classify these under categories:

i) Online services
ii) Programming services.

i) **Online-services**: Most operating systems provide interactive facilities to enable the on-line users to work with files. Few of these facilities are built-in commands of the system while others are provided by separate utility programs. But basic operating systems like MS-DOS with limited security provisions can be potentially risky because of these user owned powers. So, these must be used by technical support staff or experienced users only. For example: DEL *. * Command can erase all the files in the current directory. Also, FORMAT c: can erase the entire contents of the mentioned drive/disk. Many such services provided by the operating system related to directory operations are listed below:

- Create a file
- Delete a file

- Copy a file
- Rename a file
- Display a file
- Create a directory
- Remove an empty directory
- List the contents of a directory
- Search for a file
- Traverse the file system.

ii) **Programming services**: The complexity of the file services offered by the operating system vary from one operating system to another but the basic set of operations like: open (make the file ready for processing), close (make a file unavailable for processing), read (input data from the file), write (output data to the file), seek (select a position in file for data transfer).

All these operations are used in the form of language syntax procedures or built-in library routines, of high-level language used like C, Pascal, and Basic etc. More complicated file operations supported by the operating system provide wider range of facilities/services. These include facilities like reading and writing records, locating a record with respect to a primary key value etc. The software interrupts can also be used for activating operating system functions. For example, Interrupt 21(hex) function call request in MS-DOS helps in opening, reading and writing operations on a file.

In addition to file functions described above the operating system must provide directory operation support also like:

- Create or remove directory
- Change directory
- Read a directory entry
- Change a directory entry etc.

These are not always implemented in a high level language but language can be supplied with these procedure libraries. For example, UNIX uses C language as system programming language, so that all system calls requests are implemented as C procedures.

3.10.5 Asynchronous Input/Output

Synchronous I/O is based on blocking concept while asynchronous is interrupt-driven transfer. If an user program is doing several things simultaneously and request for I/O operation, two possibilities arise. The simplest one is that the I/O is started, then after its completion, control is transferred back to the user process. This is known as synchronous I/O where you make an I/O request and you have to wait for it to finish. This could be a problem where you would like to do some background processing and wait for a key press. Asynchronous I/O solves this problem, which is the second possibility. In this, control is returned back to the user program without waiting for the I/O completion. The I/O then continues while other system operations occur. The CPU starts the transfer and goes off to do something else until the interrupt arrives.

Asynchronous I/O operations run in the background and do not block user applications. This improves performance, because I/O operations and applications processing can run simultaneously. Many applications, such as databases and file servers, take advantage of the ability to overlap processing and I/O. To manage asynchronous I/O, each asynchronous I/O request has a corresponding control block in the application’s address space. This control block contains the control and status information for the request. It can be used again when the I/O operation is completed. Most physical I/O is asynchronous.

After issuing an asynchronous I/O request, the user application can determine when and how the I/O operation is completed. This information is provided in any of three ways:

- The application can poll the status of the I/O operation.
- The system can asynchronously notify the application when the I/O operation is done.
- The application can block until the I/O operation is complete.

Each I/O is handled by using a device-status table. This table holds entry for each I/O device indicating device's type, address, and its current status like bust or idle. When any I/O device needs service, it interrupts the operating system. After determining the device, the Operating System checks its status and modifies table entry reflecting the interrupt occurrence. Control is then returned back to the user process.

☞ **Check Your Progress 2**

1) What is the meaning of the term 'virtual device'? Give an Example.

   ………………………………………………………………………………………

   ………………………………………………………………………………………

2) What are the advantages of using directories?

   ………………………………………………………………………………………

   ………………………………………………………………………………………

3) Explain the advantages of organising file directory structure into a tree structure?

   ………………………………………………………………………………………

   ………………………………………………………………………………………

4) List few file attributes?

   ………………………………………………………………………………………

   ………………………………………………………………………………………

5) Why is SCAN scheduling also called Elevator Algorithm?

   ………………………………………………………………………………………

   ………………………………………………………………………………………

6) In an MS-DOS disk system, calculate the number of entries (i.e., No. of clusters) required in the FAT table. Assume the following parameters:

   Disk Capacity - 40 Mbytes
   Block Size - 512 Bytes
   Blocks/Cluster- 4

   ………………………………………………………………………………………

   ………………………………………………………………………………………

7) Assuming a cluster size of 512 bytes calculate the percentage wastage in file space due to incomplete filling of last cluster, for the file sizes below:

   (i) 1000 bytes (ii) 20,000 bytes

   ………………………………………………………………………………………

   ………………………………………………………………………………………

8) What is meant by an 'alias filename' and explain its UNIX implementation.

   ………………………………………………………………………………………

   ………………………………………………………………………………………

3.11 SUMMARY

This unit briefly describes the aspects of I/O and File Management. We started by looking at I/O controllers, organisation and I/O buffering. We briefly described various buffering approaches and how buffering is effective in smoothing out the speed mismatch between I/O rates and processor speed. We also looked at the four levels of I/O software: the interrupt handlers, device drivers, the device independent I/O software, and the I/O libraries and user-level software.

A well-designed file system should provide a user-friendly interface. The file system generally includes access methods, file management issues like file integrity, storage, sharing, security and protection etc. We have discussed the services provided by the operating system to the user to enable fast access and processing of files.

The important concepts related to the file system are also introduced in this unit like file concepts, attributes, directories, tree structure, root directory, pathnames, file services etc. Also a number of techniques applied to improve disk system performance have been discussed and in summary these are: disk caching, disk scheduling algorithms (FIFO, SSTF, SCAN, CSCAN, LOOK etc.), types of disk space management (contiguous and non-contiguous-linked and indexed), disk address translation, RAID based on interleaving concept etc. Auxiliary storage management is also considered as it is mainly concerned with allocation of space for files:

3.12 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The major characteristics are:

Data Rate	Disk-2Mbytes/sec
	Keyboard-10-15 bytes/sec
Units of transfer Operation	Disk-read, write, seek
	Printer-write, move, select font
Error conditions	Disk-read errors
	Printer-paper out etc.

2) Device independence refers to making the operating system software and user application independent of the devices attached. This enables the devices to be changed at different executions. Output to the printer can be sent to a disk file. Device drivers act as software interface between these I/O devices and user-level software.

3) In some operating systems like UNIX, the driver code has to be compiled and linked with the kernel object code while in some others, like MS-DOS, device drivers are installed and loaded dynamically. The advantage of the former way is its simplicity and run-time efficiency but its limitation is that addition of a new device requires regeneration of kernel, which is eliminated in the latter technique.

4) In double - buffering the transfers occur at maximum rate hence it sustains device activity at maximum speed, while single buffering is slowed down by buffer to transfer times.

5) The key objectives are to maximize the utilisation of the processor, to operate the devices at their maximum speed and to achieve device independence.

Check Your Progress 2

1) A virtual device is a simulation of an actual device by the operating system. It responds to the system calls and helps in achieving device independence. Example: Print Spooler.

2) Directories enable files to be separated on the basis of users and their applications. Also, they simplify the security and system management problems.

3) Major advantages are:

- This helps to avoid possible file name clashes
- Simplifies system management and installations
- Facilitates file management and security of files
- Simplifies running of different versions of same application.

4) File attributes are: Name, Type (in UNIX), Location at which file is stored on disk, size, protection bits, date and time, user identification etc.

5) Since an elevator continues in one direction until there are no more requests pending and then it reverses direction just like SCAN scheduling.

6) Cluster size= 4*512=2048 bytes

Number of clusters = (40*1,000,000)/2048=19531 approximately

7) (i) File size = 1000 bytes
No. of clusters = 1000/512 = 2(approximately)
Total cluster capacity = 2*512 = 1024 bytes
Wasted space = 1024-1000 = 24 bytes
Percentage Wasted space = (24 / 1024)*100
= 2.3%

(ii) File size = 20,000 bytes
No. of clusters = 20,000/512 = 40(approximately)
Total cluster capacity = 40 * 512 = 20480 bytes
Wasted space = 20480-20,000 = 480 bytes
Percentage Wasted space = (480 / 20480) * 100
= 2.3%

8) An alias is an alternative name for a file, possibly in a different directory. In UNIX, a single inode is associated with each physical file. Alias names are stored as separate items in directories but point to the same inode. It allows one physical file to be referenced by two or more different names or same name in different directory.

3.13 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.

2) H.M.Deitel, *Operating Systems*, Pearson Education Asia Place, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.

4) Colin Ritchie, *Operating Systems incorporating UNIX and Windows*, BPB Publications, New Delhi.

5) J. Archer Harris, *Operating Systems*, Schaum's Outlines, TMGH, 2002, New Delhi.

6) Achyut S Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.

7) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.

8) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.

9) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.

10) Gary Nutt, *Operating Systems – A Modern Perspective*, Pearson Education, 2003, New Delhi.

UNIT 4 SECURITY AND PROTECTION

4.0 INTRODUCTION

Modern organisations depend heavily on their information systems and large investments are made on them annually. These systems are now computerised, and networking has been the common trend during the last decade. The availability of information and computer resources within an organisation as well as between cooperating organisations is often critical for the production of goods and services. In addition, data stored in or derived from the system must be correct, i.e., data integrity must be ensured. In some situations, it is also of great importance to keep information confidential.

Computer security is traditionally defined by the three attributes of confidentiality, integrity, and availability. *Confidentiality* is the prevention of unauthorised disclosure of information. *Integrity* is the prevention of unauthorised modification of information, and *availability* is the prevention of unauthorised withholding of information or resources. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorised or incompetent user.

Even if a perfectly secure operating system was created, human error (or purposeful human malice) can make it insecure. More importantly, there is no such thing as a completely secure system. No matter how secure the experts might think a particular system is, there exists someone who is clever enough to bypass the security.

It is important to understand that a trade-off always exists between security and the ease of use. It is possible to be *too* secure but security always extracts a price, typically making it more difficult to use your systems for their intended purposes. Users of a secure system need to continually type in, continually change and memorise complex passwords for every computer operation, or use biometric systems with retina and fingerprint and voice scans. All these measures along with confirmations are likely to bring any useful work to a snail's pace. If the consequences are great enough, then you might have to accept extreme security measures. Security is a relative concept, and gains in security often come only with penalties in performance. To date, most systems designed to include security in the operating system structure have exhibited either slow response times or awkward user interfaces-or both.

This unit discusses about the various types of security threats and the commonly deployed detective and preventive methods.

4.1 OBJECTIVES

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

- identify the security threats and goals;
- mention the role of security policies and mechanisms;
- discuss the significance of authentication and also describe various types of authentication procedures, and
- describe the various models of protection.

4.2 SECURITY THREATS

System security can mean several things. To have system security we need to protect the system from corruption and we need to protect the data on the system. There are many reasons why these need not be secure.

- Malicious users may try to hack into the system to destroy it.
- Power failure might bring the system down.
- A badly designed system may allow a user to accidentally destroy important data.
- A system may not be able to function any longer because one user fills up the entire disk with garbage.

Although discussions of security usually concentrate on the first of these possibilities, the latter two can be equally damaging the system in practice. One can protect against power failure by using un-interruptible power supplies (UPS). These are units which detect quickly when the power falls below a certain threshold and switch to a battery. Although the battery does not last forever−the UPS gives a system administrator a chance to halt the system by the proper route.

The problem of malicious users has been heightened in recent years by the growth of international networks. Anyone connected to a network can try to log on to almost any machine. If a machine is very insecure, they may succeed. In other words, we are not only looking at our local environment anymore, we must consider potential threats to system security to come from any source. The final point can be controlled by enforcing quotas on how much disk each user is allowed to use.

We can classify the security attacks into two types as mentioned below:

1) **Direct:** This is any direct attack on your specific systems, whether from outside hackers or from disgruntled insiders.

2) **Indirect:** This is general random attack, most commonly computer viruses, computer worms, or computer Trojan horses.

These security attacks make the study of security measures very essential for the following reasons:

- **To prevent loss of data:** You don't want someone hacking into your system and destroying the work done by you or your team members. Even if you have good back-ups, you still have to identify that the data has been damaged (which can occur at a critical moment when a developer has an immediate need for the damaged data), and then restore the data as best you can from your backup systems.

- **To prevent corruption of data:** Sometimes, the data may not completely be lost, but just be partially corrupted. This can be harder to discover, because unlike complete destruction, there is still data. If the data seems reasonable, you could go a long time before catching the problem, and cascade failure could result in serious problems spreading far and wide through your systems before discovery.
- **To prevent compromise of data:** Sometimes it can be just as bad (or even worse) to have data revealed than to have data destroyed. Imagine the consequences of important trade secrets, corporate plans, financial data, etc. falling in the hands of your competitors.
- **To prevent theft of data:** Some kinds of data are subject to theft. An obvious example is the list of credit card numbers belonging to your customers. Just about anything associated with money can be stolen.
- **To prevent sabotage:** A disgruntled employee, a dishonest competitor, or even a stranger could use any combination of the above activities to maliciously harm your business. Because of the thought and intent, this is the most dangerous kind of attack, the kind that has the potential for the greatest harm to your business.

Let's now discuss some security-related issues that the OS must deal to maintain confidentiality, integrity, and availability of system resources.

4.3 SECURITY POLICIES AND MECHANISMS

Computer systems and especially their protection mechanisms must be penetration resistant. However, most, or perhaps all, current systems have security holes that make them vulnerable. As long as vulnerabilities remain in a system, the security may be circumvented. It is thus not difficult to understand why system owners would like to know how secure their particular system actually is.

Security evaluations have been performed for quite some time by mainly two different methods. The first is to classify systems into a predefined set of categories, each with different security requirements. The other method is referred to as penetration testing, which is a form of stress testing exposing weaknesses in a system. Here, the idea is to let a group of people, normally very skilled in the computer security area, attack a target system.

A **security policy** establishes accountability for information protection by defining a set of rules, conditions, and practices that regulate how an organisation manages, protects, and distributes sensitive information. While substantial effort may be put in by the vendor in implementing the mechanisms to enforce the policy and developing assurance that the mechanisms perform properly, all efforts fail if the policy itself is flawed or poorly understood. For this reason, the standards require that "there must be an explicit and well-defined security policy enforced by the system". A security policy may address confidentiality, integrity, and/or availability.

4.4 AUTHENTICATION

It is the process of verifying a user's identity (who you are) through the use of a shared secret (such as a password), a physical token or an artifact (such as a key or a smart card), or a biometric measure (such as a fingerprint).

These three types of authentication are commonly referred to as something you have (physical token), something you know (shared secret), and something you are (biometric measure).

The types and rigor of authentication methods and technologies vary according to the security requirements or policies associated with specific situations and implementations.

The goal of authentication is to provide “reasonable assurance” that anyone who attempts to access a system or network is a legitimate user. In other words, authentication is designed to limit the possibility that an unauthorised user can gain access by impersonating as an authorised user. Here, again, the organisation’s security policy should guide how difficult it is for one user to impersonate another. Highly sensitive or valuable information demands stronger authentication technologies than less sensitive or valuable information.

4.4.1 Passwords

The most common and least stringent form of authentication technology demands that users provide only a valid account name and a password to obtain access to a system or network. The password-based authentication is one-way and normally stores the user-id and password combination in a file that may be stored on the server in an encrypted or plaintext file. Most people using the public e-mail systems use this form of authentication.

Protection of Passwords

Some systems store the passwords of all users in a protected file. However, this scheme is vulnerable to accidental disclosure and to abuse by system administrators. UNIX stores only an encrypted form of every password; a string is the password of user X if its encrypted form matches the encrypted password stored for X.

The encrypted password is accessible to everyone, but one cannot find the clear text password without either guessing it or breaking the encryption algorithm.

Data Encryption Standard (DES)

Encryption is based on one-way functions: functions that are cheap to compute but whose inverse is very expensive to compute. A still widely used, though older encryption algorithm is the Data Encryption Standard (DES), which uses a 56bit key.

UNIX does not encrypt passwords with a secret key, instead, it uses the password as the key to encrypt a standard string. The latter method is not as vulnerable to attacks based on special properties of the “secret” key, which must nevertheless be known to many people.

Limitations of Passwords

Users tend to choose passwords that are easy to guess; persistent guessing attacks can find 80-90% of all passwords. Good guesses include the login name, first names, treat names, words in the on-line dictionary, and any of these reversed or doubled.

The way to defeat these attacks is to choose a password that does not fall into any of those categories. Preferably passwords should contain some uppercase letters, numbers and/or special characters; their presence increases the search space by a large factor.

4.4.2 Alternative Forms of Authentication

Alternative forms of authentication include the following technologies:

- **Biometrics:** These systems read some physical characteristic of the user, such as their fingerprint, facial features, retinal pattern, voiceprints, signature analysis, signature motion analysis, typing rhythm analysis, finger length analysis, DNA analysis. These readings are compared to a database of authorised users to determine identity. The main problems of these schemes are high equipment cost and low acceptance rates.
- **Security Devices or Artifacts:** These systems require use of a special-purpose hardware device that functions like a customised key to gain system access. The device may be inserted into the system like a key or used to generate a code that is then entered into the system. The best example is the use of an ATM card, which is inserted in the machine and also requires

password to be entered simultaneously. It holds the user information on either a magnetic strip or a smart chip capable of processing information.

- **Concentric-ring Authentication:** These systems require users to clear additional authentication hurdles as they access increasingly sensitive information. This approach minimizes the authentication burden as users access less sensitive data while requiring stronger proof of identity for more sensitive resources.

Any authentication method may be broken into, given sufficient time, expense, and knowledge. The goal of authentication technologies is to make entry into system expensive and difficult enough that malicious individuals can be deterred from trying to break the security.

4.5 PROTECTION IN COMPUTER SYSTEMS

The use of computers to store and modify information can simplify the composition, editing, distribution, and reading of messages and documents. These benefits are not free, however, part of the cost is the aggravation of some of the security problems discussed above and the introduction of some new problems as well. Most of the difficulties arise because a computer and its programs are shared amongst several users.

For example, consider a computer program that displays portions of a document on a terminal. The user of such a program is very likely not its developer. It is, in general, possible for the developer to have written the program so that it makes a copy of the displayed information accessible to himself (or a third party) without the permission or knowledge of the user who requested the execution of the program. If the developer is not authorised to view this information, security has been violated.

In compensation for the added complexities automation brings to security, an automated system, if properly constructed, can bestow a number of benefits as well. For example, a computer system can place stricter limits on user discretion. In the paper system, the possessor of a document has complete discretion over its further distribution. An automated system that enforces distribution constraints strictly can prevent the recipient of a message or document from passing it to others. Of course, the recipient can always copy the information by hand or repeat it verbally, but the inability to pass it on directly is a significant barrier.

An automated system can also offer new kinds of access control. Permission to execute certain programs can be granted or denied so that specific operations can be restricted to designated users. Controls can be designed so that some users can execute a program but cannot read or modify it directly. Programs protected in this way might be allowed to access information not directly available to the user, filter it, and pass the results back to the user.

Information contained in an automated system must be protected from three kinds of threats: (1) the *unauthorised disclosure* of information, (2) the *unauthorised modification* of information, and (3) the *unauthorised withholding* of information (usually called *denial of service).*

To protect the computer systems, we often need to apply some security models. Let us see in the next section about the various security models available.

4.6 SECURITY MODELS

Security models are more precise and detailed expressions of security policies discussed as above and are used as guidelines to build and evaluate systems. Models can be discretionary or mandatory. In a *discretionary* model, holders of rights can be allowed to transfer them at their discretion. In a *mandatory* model only designated roles are allowed to grant rights and users cannot transfer them. These models can be classified into those based on the access matrix, and multilevel models. The first model controls access while the second one attempts to control information flow.

Security Policy vs. Security Model

The Security Policy outlines several high level points: how the data is accessed, the amount of security required, and what the steps are when these requirements are not met. The security model is more in depth and supports the security policy. Security models are an important concept in the design of any security system. They all have different security policies applying to the systems.

4.6.1 Access Matrix Model

The access matrix model for computer protection is based on abstraction of operating system structures. Because of its simplicity and generality, it allows a variety of implementation techniques, as has been widely used.

There are three principal components in the access matrix model:

- A set of passive *objects,*
- A set of active *subjects,* which may manipulate the objects,
- A set of *rules* governing the manipulation of objects by subjects.

Objects are typically files, terminals, devices, and other entities implemented by an operating system. A subject is a process and a *domain* (a set of constraints within which the process may access certain objects). It is important to note that every subject is also an object; thus it may be read or otherwise manipulated by another subject. The ***access matrix*** is a rectangular array with one row per subject and one column per object. The entry for a particular row and column reflects the mode of access between the corresponding subject and object. The mode of access allowed depends on the type of the object and on the functionality of the system; typical modes are read, write, append, and execute.

Objects / Subject	File 1	File 2	File 3
User 1	*r, w*	*R*	*r, w, x*
User 2	*r*	*R*	*r, w, x*
User 3	*r, w, x*	*r, w*	*r, w, x*

Figure 1: An access matrix

All accesses to objects by subjects are subject to some conditions laid down by an enforcement mechanism that refers to the data in the access matrix. This mechanism, called a *reference monitor,* rejects any accesses (including improper attempts to alter the access matrix data) that are not allowed by the current protection state and rules. References to objects of a given type must be validated by the *monitor* for that type. While implementing the access matrix, it has been observed that the access matrix tends to be very sparse if it is implemented as a two-dimensional array. Consequently, implementations that maintain protection of data tend to store them either row wise, keeping with each subject a list of the objects and access modes allowed on it; or column wise, storing with each object a list of those subjects that may access it and the access modes allowed on each. The former approach is called the ***capability list*** approach and the latter is called the ***access control list*** approach.

In general, access control governs each user's ability to read, execute, change, or delete information associated with a particular computer resource. In effect, access control works at two levels: first, to grant or deny the ability to interact with a resource, and second, to control what kinds of operations or activities may be performed on that resource. Such controls are managed by an access control system. Today, there are numerous methods of access controls implemented or practiced in real-world settings. These include the methods described in the next four sections.

4.6.2 Mandatory Access Control

In a ***Mandatory Access Control (MAC)*** environment, all requests for access to resources are automatically subject to access controls. In such environments, all users and resources are classified and receive one or more security labels (such as “Unclassified,” “Secret,” and “Top Secret”). When a user requests a resource, the associated security labels are examined and access is permitted only if the user’s label is greater than or equal to that of the resource.

4.6.3 Discretionary Access Control

In a ***Discretionary Access Control (DAC)*** environment, resource owners and administrators jointly control access to resources. This model allows for much greater flexibility and drastically reduces the administrative burdens of security implementation.

4.6.4 Rule-Based Access Control

In general, ***rule-based access control*** systems associate explicit access controls with specific system resources, such as files or printers. In such environments, administrators typically establish access rules on a per-resource basis, and the underlying operating system or directory services employ those rules to grant or deny access to users who request access to such resources. Rule-based access controls may use a MAC or DAC scheme, depending on the management role of resource owners.

4.6.5 Role-Based Access Control

Role-based access control (RBAC) enforces access controls depending upon a user’s role(s). Roles represent specific organisational duties and are commonly mapped to job titles such as “Administrator,” “Developer,” or “System Analyst.” Obviously, these roles require vastly different network access privileges.

Role definitions and associated access rights must be based upon a thorough understanding of an organisation’s ***security policy***. In fact, roles and the access rights that go with them should be directly related to elements of the security policy.

4.6.6 Take-Grant Model

The access matrix model, properly interpreted, corresponds very well to a wide variety of actual computer system implementations. The simplicity of the model, its definition of subjects, objects, and access control mechanisms, is very appealing. Consequently, it has served as the basis for a number of other models and development efforts. We now discuss a model based on access matrix.
Take-grant models use graphs to model access control. They have been well researched. Although explained in the terms of graph theory, these models are fundamentally access matrix models. The graph structure can be represented as an adjacency matrix, and labels on the arcs can be coded as different values in the matrix.

In a take-grant model, the protection state of a system is described by a directed graph that represents the same information found in an access matrix. Nodes in the graph are of two types, one corresponding to subjects and the other to objects. An arc directed from a node A to another node B indicates that the subject (or object) A has some access right(s) to subject (or object) B. The arc is labeled with the set of A’s rights to B. The possible access rights are read (r), write (w), take (t), and grant (g). Read and write have the obvious meanings. “*Take*” access implies that node A can take node B’s access rights to any other node.

For example, if there is an arc labeled (r, g) from node B to node C, and if the arc from A to B includes a "t" in its label, then an arc from A to C labeled (r, g) could be added to the graph (see *Figure 2*). Conversely, if the arc from A to B is marked with a "g", B can be granted any access right A possesses. Thus, if A has (w) access to a node D and (g) access to B, an arc from B to D marked (w) can be added to the graph (see *Figure 3*).

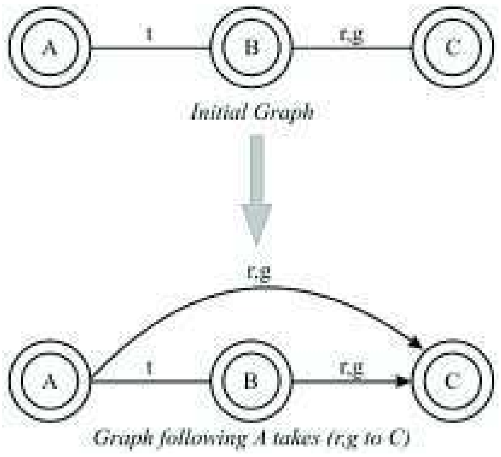


Figure 2: Example of *Take*

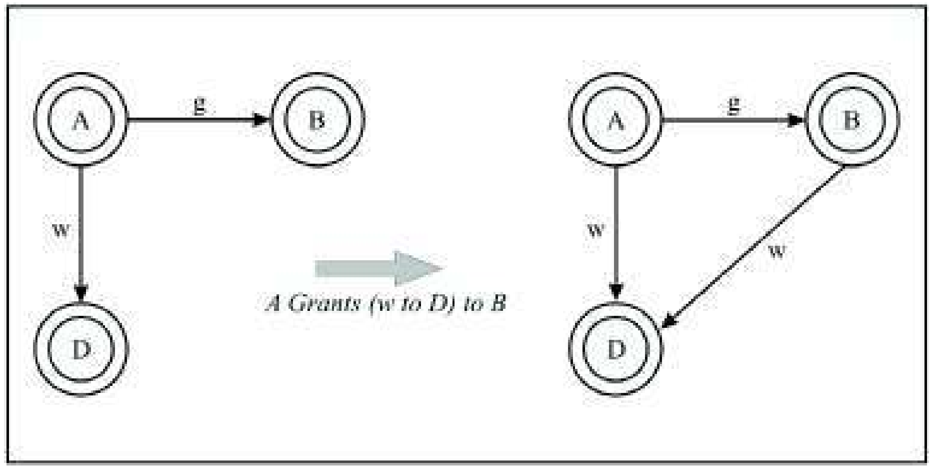


Figure 3: Example of Grant

Because the graph needs only the inclusion of arcs corresponding to non-null entries in the access matrix, it provides a compact way to present the same information given in a relatively sparse access matrix. Capability systems are thus prime candidates for this modeling technique; each arc would then represent a particular capability. Together with the protection graph, the model includes a set of rules for adding and deleting both nodes and arcs to the graph.

Two of these, corresponding to the exercise of "take" and "grant" access rights, have already been described. A "create" rule allows a new node to be added to the graph. If subject A creates a new node Y, both the node Y and an arc AY are added to the graph. The label on AY includes any subset of the possible access rights. A "remove" rule allows an access right to be removed from an arc; if all rights are removed from an arc, the arc is removed as well.

4.6.7 Multilevel Models

This type of models corresponds to the multilevel policies where data is classified into sensitivity levels and users have access according to their clearances. Because of the way they control security they have also been called *data flow* models, they control the allowed flow of data between levels.

- The Bell-La Padula model, intended to control leakage of information between levels.
- The Biba model, which controls the integrity.
- The Lattice model, which generalises the partially ordered levels of the previous models using the concept of mathematical lattices.

Bell and La Padula Model

Bell and La Padula use finite-state machines to formalise their model. They define the various components of the finite state machine, define what it means (formally) for a given system state to be secure, and then consider the transitions that can be allowed so that a secure state can never lead to an insecure state.

In addition to the subjects and objects of the access matrix model, it includes the security levels of the military security system: each subject has a authority and each object has a classification. Each subject also has a *current security level,* which may not exceed the subject's authority. The access matrix is defined as above, and four modes of access are named and specified as follows:

- **Read-only**: subject can read the object but not modify it;
- **Append**: subject can write the object but cannot read it;
- **Execute**: subject can execute the object but cannot read or write it directly; and
- **Read-write**: subject can both read and write the object.

A control attribute, which is like an ownership flag, is also defined. It allows a subject to pass to other subjects some or all of the access modes it possesses for the controlled object. The control attribute itself cannot be passed to other subjects; it is granted to the subject that created the object.

For a system state to be secure, it should hold the following two properties:

(1) The *simple security property:* No subject has read access to any object that has a classification greater than the clearance of the subject; and

(2) The **-property* (pronounced "star-property"): No subject has append-access to an object whose security level is not at least the current security level of the subject; no subject has read-write access to an object whose security level is not equal to the current security level of the subject; and no subject has read access to an object whose security level is not at most the current security level of the subject.

An example restatement of the model is discussed below:

- In this case, we use security levels as (*unclassified* < *confidential* < *secret* < *top-secret*).

The security levels are used to determine appropriate access rights.

The essence of the model can be defined as follows:

- A higher-level subject (e.g., secret level) can always "*read-down*" to the objects with level which is either equal (e.g., secret level) or lower (e.g., confidential / unclassified level). So the system high (top security level in the system) has the read-only access right to all the objects in the entire system.
- A lower-level subject can never "*read-up*" to the higher-level objects, as the model suggests that these objects do not have enough authority to read the high security level information.
- The subject will have the Read-Write access right when the objects are in the same level as the subject.

- A lower-level subject (e.g., confidential level) can always "*write-up*" (**Append** access right) to the objects with level, which is either equal (e.g., confidential level) or higher (e.g., secret / top-secret level). This happens as a result of all the subjects in the higher security levels (e.g., secret / top-secret level) having the authority to read the object from lower levels.
- A higher-level subject can never "*write-down*" to the lower level objects, as the model suggests that these objects are not secure enough to handle the high security level information.

There are also a few problems associated with this model:

- For a subject of a higher-level, all of its information is considered as the same level. So there is no passing of information from this subject to any object in the lower level.
- For an environment where security levels are not hierarchical related, the model does not account for the transfer of information between these domains.
- The model does not allow changes in access permission.
- The model deals only with confidentiality but not on integrity.

Biba Integrity model

This model is a modification of the Bell-La Padula model, with emphasis on the integrity. There are two properties in the Biba model:

- **SI-property** (*simple integrity property*)**:** a subject may have write access to an object only if the security level of the subject is either higher or equals to the level of the object.
- **Integrity property:** a subject has the read-only access to an object *o*, then it can also have the write access to another object *p* only if the security level of *p* is either lower or equals to the level of *o*.

The essence of the model can be defined as follows:

- The read access policy is the same as the Bell-La Padula model.
- No information from a subject can be passed onto an object in a higher security level. This prevents the contamination of the data of higher integrity from the data of lower integrity.

The major problem associated with this model is that there isn't a practical model that can fulfil the confidentiality of the Bell-La Padula model and the integrity of the Biba model.

Information-Flow Models

The significance of the concept of information flow is that it focuses on the actual operations that transfer information between objects. Access control models (such as the original Bell and La Padula model) represent instead the transfer or exercise by subjects of access rights to objects. Information-flow models can be applied to the variables in a program directly, while the access matrix models generally apply to larger objects such as files and processes.

The flow model, compared with the Bell and La Padula model, is relatively uncomplicated. Instead of a series of conditions and properties to be maintained, there is the single requirement that information flow obey the lattice structure as described below. An information-flow model has five components:

- A set of objects, representing information receptacles (e.g., files, program variables, bits),
- A set of processes, representing the active agents responsible for information flow,
- A set of security classes, corresponding to disjoint classes of information,
- An associative and commutative class-combining operator that specifies the class of the information generated by any binary operation on information from two classes, and
- A flow relation that, for any pair of security classes, determines whether information is allowed to flow from one to the other.

Maintaining secure information flow in the modeled system corresponds to ensuring that the actual information that flows between objects do not violate the specified flow relation. This problem is addressed primarily in the context of programming languages. Information flows from an object *x* to an object *y* whenever information stored in *x* is transferred directly to *y* or used to derive information transferred to *y*. Two kinds of information flow, *explicit* and *implicit,* are identified. A flow from *x* to *y* is explicit if the operations causing it are independent of the value of *x* (e.g., in a statement directly assigning the value of *x* to *y*). It is implicit if the statement specifies a flow from some other object *z* to *y*, but execution of the statement depends on the value of *x* (e.g., in the conditional assignment *if x then y: = z;* information about the value of *x* can flow into *y* whether or not the assignment is executed).

The ***lattice model*** for security levels is widely used to describe the structure of military security levels. A lattice is a finite set together with a partial ordering on its elements such that for every pair of elements there is a least upper bound and a greatest lower bound. The simple linear ordering of sensitivity levels has already been defined. Compartment sets can be partially ordered by the subset relation: one compartment set is greater than or equal to another if the latter set is a subset of the former. Classifications, which include a sensitivity level and *a* (perhaps empty) compartment set, can then be partially ordered as follows:

For any sensitivity levels a, b and any compartment sets c, d; the relation $(a, c) \geq (b, d)$ exists if and only if $a \geq b$ and $c \supseteq d$. That each pair of classifications has a greatest lower bound and a least upper bound follows from these definitions and the facts that the classification "unclassified, no compartments" is a global lower bound and that we can postulate a classification "top secret, all compartments" as a global upper bound. Because the lattice model matches the military classification structure so closely, it is widely used.

Check Your Progress 1

1) What is computer security and what are the several forms of damages that can be done by the intruders to the computer systems?

………………………………………………………………………………………

………………………………………………………………………………………

……

2) Explain the role of Access Lists.

………………………………………………………………………………………

………………………………………………………………………………………

……

4.7 SUMMARY

Security is an important aspect of an OS. The distinction between security and protection is:

Security: Broad sense to refer all concerns about controlled access to facilities.

Protection: Mechanism to support security.

In this unit, we have discussed the concept of security and various threats to it. Also we have discussed various protection and security mechanisms to enforce security to the system.

4.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Computer security is required because most organisations can be damaged by hostile software or intruders. There may be several forms of damage which are obviously interrelated. These include:

- Damage or destruction of computer systems.
- Damage or destruction of internal data.
- Loss of sensitive information to hostile parties.
- Use of sensitive information to steal items of monetary value.
- Use of sensitive information against the organisation's customers which may result in legal action by customers against the organisation and loss of customers.
- Damage to the reputation of an organisation.
- Monetary damage due to loss of sensitive information, destruction of data, hostile use of sensitive data, or damage to the organisation's reputation.

2) The logical place to store authorisation information about a file is with the file itself or with its inode.

Each element of an Access Control List (ACL) contains a set of user names and a set of operation names. To check whether a given user is allowed to perform a given operation, the kernel searches the list until it finds an entry that applies to the user, and allows the access if the desired operation is named in that entry. The ACL usually ends with a default entry. Systems that use ACLs stop searching when they find an ACL entry that applies to the subject. If this entry denies access to the subject, it does not matter if later entries would also match and grant access; the subject will be denied access. This behaviour is useful for setting up exceptions, such as everyone can access this file except people in this group.

4.9 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.
2) H.M.Deitel, *Operating Systems*, Pearson Education Asia, New Delhi.
3) Andrew S. Tanenbaum, *Operating Systems- Design and implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.
4) Achyut S. Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.
5) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.
6) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.
7) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.

UNIT 1 MULTIPROCESSOR SYSTEMS

Structure **Page Nos.**

1.0 INTRODUCTION

A multiprocessor system is a collection of a number of standard processors put together in an innovative way to improve the performance / speed of computer hardware. The main feature of this architecture is to provide high speed at low cost in comparison to uniprocessor. In a distributed system, the high cost of multiprocessor can be offset by employing them on a computationally intensive task by making it compute server. The multiprocessor system is generally characterised by - *increased system throughput* and *application speedup* - parallel processing.

Throughput can be improved, in a *time-sharing environment*, by executing a number of unrelated user processor on different processors in parallel. As a result a large number of different tasks can be completed in a unit of time without explicit user direction. On the other hand application speedup is possible by creating a multiple processor scheduled to work on different processors.

The scheduling can be done in two ways:

1) *Automatic means*, by parallelising compiler.

2) *Explicit-tasking approach*, where each programme submitted for execution is treated by the operating system as an independent process.

Multiprocessor operating systems aim to support high performance through multiple CPUs. An important goal is to make the number of CPUs transparent to the application. Achieving such transparency is relatively easy because the communication between different (parts of) applications uses the same primitives as those in multitasking uni-processor operating systems. The idea is that all communication is done by manipulating data at shared memory locations, and that we only have to protect that data against simultaneous access. Protection is done through synchronization primitives like semaphores and monitors.

In this unit we will study multiprocessor coupling, interconnections, types of multiprocessor operating systems and synchronization.

1.1 OBJECTIVES

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

- define a multiprocessor system;
- describe the architecture of multiprocessor and distinguish among various types of architecture;
- become familiar with different types of multiprocessor operating systems;
- discuss the functions of multiprocessors operating systems;
- describe the requirement of multiprocessor in Operating System, and
- discuss the synchronization process in a multiprocessor system.

1.2 MULTIPROCESSING AND PROCESSOR COUPLING

Multiprocessing is a general term for the use of two or more CPUs within a single computer system. There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined. The term multiprocessing is sometimes used to refer to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the term multiprogramming is more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two. Processor Coupling is a type of multiprocessing. Let us see in the next section.

Processor Coupling

Tightly-coupled multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory (SMP), or may participate in a memory hierarchy with both local and shared memory (NUMA). *The IBM p690 Regatta* is an example of a high end SMP system. Chip multiprocessors, also known as multi-core computing, involve more than one processor placed on a single chip and can be thought of the most extreme form of tightly-coupled multiprocessing. Mainframe systems with multiple processors are often tightly-coupled.

Loosely-coupled multiprocessor systems often referred to as clusters are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system. *A Linux Beowulf* is an example of a loosely-coupled system.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems, but have historically required greater initial investments and may depreciate rapidly; nodes in a loosely-coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster.

Power consumption is also a consideration. Tightly-coupled systems tend to be much more energy efficient than clusters. This is due to fact that considerable economies can be realised by designing components to work together from the beginning in tightly-coupled systems, whereas loosely-coupled systems use components that were not necessarily intended specifically for use in such systems.

1.3 MULTIPROCESSOR INTERCONNECTIONS

As learnt above, a multiprocessor can speed-up and can improve throughput of the computer system architecturally. The whole architecture of multiprocessor is based on the principle of parallel processing, which needs to synchronize, after completing a stage of computation, to exchange data. For this the multiprocessor system needs an efficient mechanism to communicate. This section outlines the different architecture of multiprocessor interconnection, including:

- Bus-oriented System
- Crossbar-connected System
- Hyper cubes
- Multistage Switch-based System.

1.3.1 Bus-oriented System

Figure1 illustrates the typical architecture of a bus oriented system. As indicated, processors and memory are connected by a common bus. Communication between processors (P1, P2, P3 and P4) and with globally shared memory is possible over a shared bus. Other then illustrated many different schemes of bus-oriented system are also possible, such as:

1) Individual processors may or may not have private/cache memory.
2) Individual processors may or may not attach with input/output devices.
3) Input/output devices may be attached to shared bus.
4) Shared memory implemented in the form of multiple physical banks connected to the shared bus.

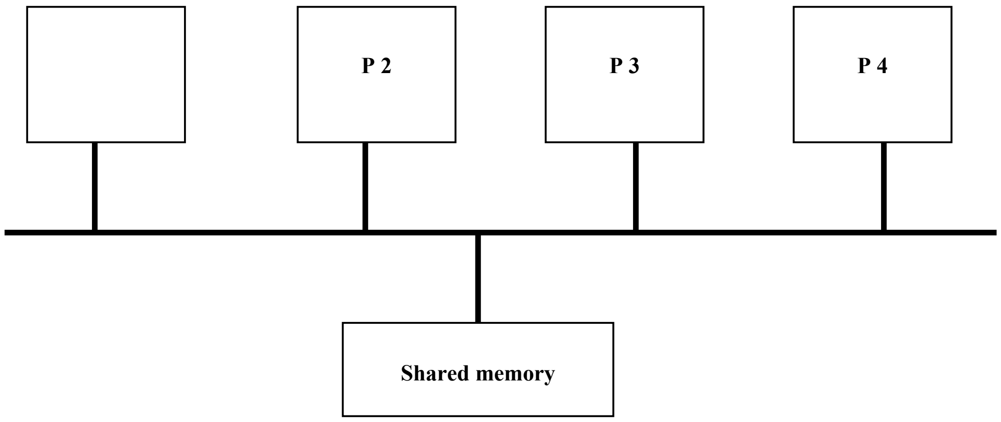


Figure 1. Bus-oriented multiprocessor interconnection

The above architecture gives rise to a problem of *contention* at two points, one is shared bus itself and the other is shared memory. Employing private/cache memory in either of two ways, explained below, the problem of contention could be reduced;

- with shared memory; and
- with cache associated with each individual processor

1) **With shared memory**

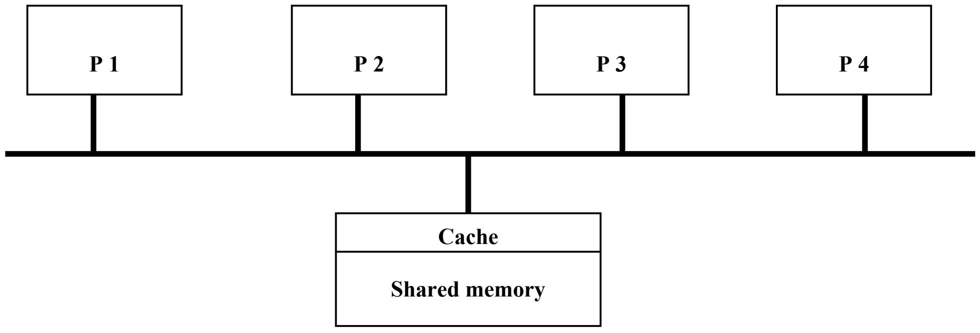


Figure 2: With shared Memory

2) **Cache associated with each individual processor.**

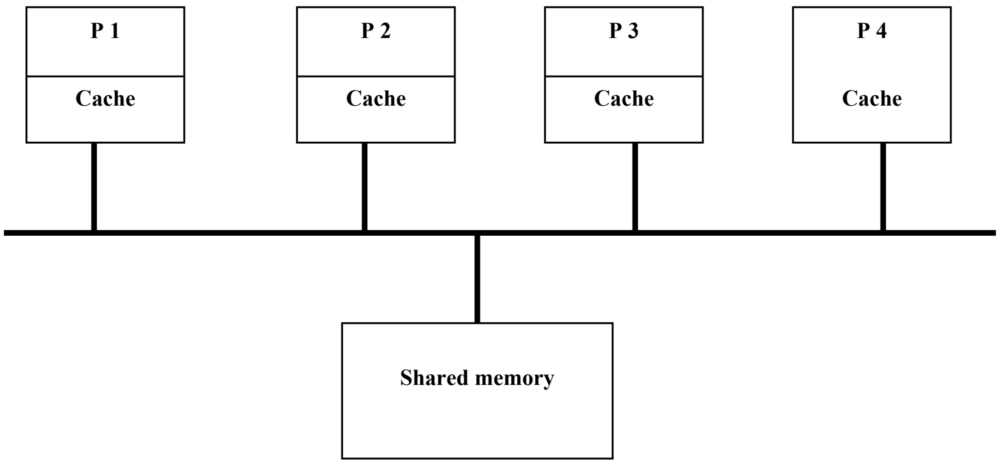


Figure 3: Individual processors with cache

The second approach where the cache is associated with each individual processor is the most popular approach because it reduces contention more effectively. Cache attached with processor an capture many of the local memory references for example, with a cache of 90% hit ratio, a processor on average needs to access the shared memory for 1 (one) out of 10 (ten) memory references because 9 (nine) references are already captured by private memory of processor. In this case where memory access is uniformly distributed a 90% cache hit ratio can support the shared bus to speed-up 10 times more than the process without cache. The negative aspect of such an arrangement arises when in the presence of multiple cache the shared writable data are cached. In this case Cache Coherence is maintained to consistency between multiple physical copies of a single logical datum with each other in the presence of update activity. Yes, the cache coherence can be maintained by attaching additional hardware or by including some specialised protocols designed for the same but unfortunately this special arrangement will increase the bus traffic and thus reduce the benefit that processor caches are designed to provide.

Cache coherence refers to the integrity of data stored in local caches of a shared resource. Cache coherence is a special case of memory coherence.

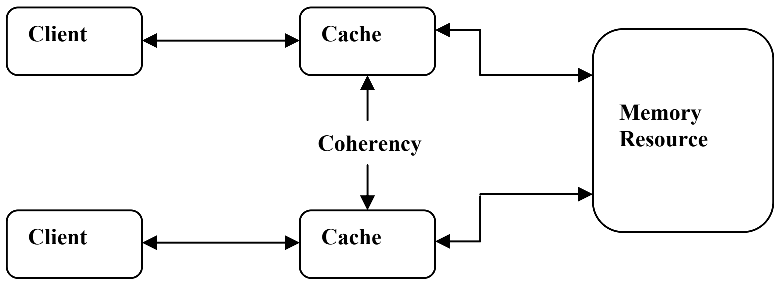


Figure 4: Multiple Caches of Common Resource

When clients in a system, particularly CPUs in a multiprocessing system, maintain caches of a common memory resource, problems arise. Referring to the *Figure 4*, if the top client has a copy of a memory block from a previous read and the bottom client changes that memory block, the top client could be left with an invalid cache of memory without it knowing any better. Cache coherence is intended to manage such conflicts and maintain consistency between cache and memory.

Let us discuss some techniques which can be employed to decrease the impact of bus and memory saturation in bus-oriented system.

1) **Wider Bus Technique**: As suggested by its name a bus is made wider so that more bytes can be transferred in a single bus cycle. In other words, a wider

parallel bus increases the bandwidth by transferring more bytes in a single bus cycle. The need of bus transaction arises when lost or missed blocks are to fetch into his processor cache. In this transaction many system supports, block-level reads and writes of main memory. In the similar way, a missing block can be transferred from the main memory to his processor cache only by a single main-memory (block) read action. The advantage of *block level access to memory* over *word-oriented busses* is the amortization of overhead addressing, acquisition and arbitration over a large number of items in a block. Moreover, it also enjoyed specialised burst bus cycles, which makes their transport more efficient.

2) **Split Request/Reply Protocols**: The memory request and reply are split into two individual works and are treated as separate bus transactions. As soon as a processor requests a block, the bus released to other user, meanwhile it takes for memory to fetch and assemble the related group of items.

The bus-oriented system is the simplest architecture of multiprocessor system. In this way it is believed that in a tightly coupled system this organisation can support on the order of 10 processors. However, the two contention points (the bus and the shared memory) limit the scalability of the system.

1.3.2 Crossbar-Connected System

Crossbar-connected System (illustrated in *Figure 5*) is a grid structure of processor and memory modules. The every cross point of grid structure is attached with switch. By looking at the *Figure* it seems to be a contention free architecture of multiprocessing system, it shows simultaneous access between processor and memory modules as *N* number of processors are provided with *N* number of memory modules. Thus each processor accesses a different memory module. Crossbar needs N^2 switches for fully connected network between processors and memory. Processors may or may not have their private memories. In the later case the system supports uniform-memory-access.

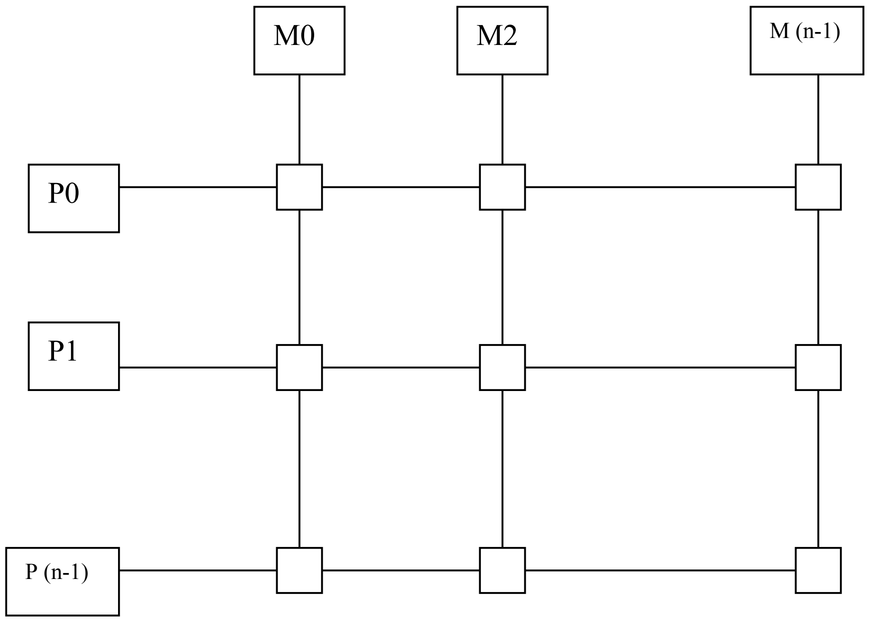


Figure 5: Crossbar-connected system

Where P= processor, M=Memory Module and =Switch

But Unfortunately this system also faces the problem of contention when,

1) More than two processors (P1 and P2) attempt to access the one memory module (M1) at the same time. In this condition, contention can be avoided by making any one processor (P1) deferred until the other one (P2) finishes this work or left the same memory module (M1) free for processor P1.

2) More than two processor attempts to access the same memory module. This problem cannot be solved by above-mentioned solution.

Thus in crossbar connection system the problem of contention occurs on at memory neither at processor nor at interconnection networks level. The solution of this problem includes quadratic growth of its switches as well as its complexity level. Not only this, it will make the whole system expensive and also limit the scalability of the system.

1.3.3 Hypercubes System

Hypercube base architecture is not an old concept of multiprocessor system. This architecture has some advantages over other architectures of multiprocessing system. As the *Figure 6* indicates, the system topology can support large number of processors by providing increasing interconnections with increasing complexity. In an *n-degree* hypercube architecture, we have:

1) 2^n nodes (Total number of processors)
2) Nodes are arranged in n-dimensional cube, i.e. each node is connected to *n* number of nodes.
3) Each node is assigned with a unique address which lies between 0 to$2^n - 1$
4) The adjacent nodes (n−1) are differing in 1 bit and the *nth* node is having maximum '*n*' internode distance.

Let us take an example of **3-degree hypercube** to understand the above structure:

1) 3-degree hypercube will have 2^n nodes i.e., $2^3 = 8$ nodes
2) Nodes are arranged in 3-dimensional cube, that is, each node is connected to 3 number of nodes.
3) Each node is assigned with a unique address, which lies between 0 to 7 ($2^n - 1$), i.e., 000, 001, 010, 011, 100, 101, 110, 111
4) Two adjacent nodes differing in 1 bit (001, 010) and the 3rd (n^{th}) node is having maximum '3'internode distance (100).

Hypercube provide a good basis for scalable system because its communication length grows logarithmically with the number of nodes. It provides a bi-directional communication between two processors. It is usually used in loosely coupled multiprocessor system because the transfer of data between two processors goes through several intermediate processors. The longest internodes delay is *n-degree*. To increase the input/output bandwidth the input/output devices can be attached with every node (processor).

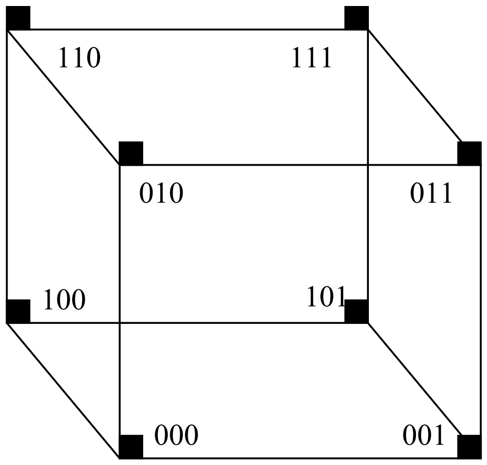


Figure 6: Hypercubes System

1.3.4 Multistage Switch Based system

Multistage Switch Based System permits simultaneous connection between several input-output pairs. It consists of several stages of switches which provide multistage interconnection network. A *N* input-output connections contains K= $\log_2 N$ stages of $^{N}/_{2}$ switches at each stage. In simple words, *N*N* processor-memory interconnection network requires $(^{N}/_{2})$ x = $\log_2 N$ switches.

For example, in a 8×8 process-memory interconnection network requires $(8/_2 * \log_2 8)$ = 4*3= 12 switches. Each switch acts as 2×2 crossbar.

This network can connect any processor to any memory module by making appropriate connection of each of the '*K*' stages. The binary address of processor and memory gives binary string routing path between module pairs of input and output. the routing path *id* decided by on the basis of destination binary address, that the sender includes with each request for connection. Various combinations of paths between sources to destination are possible by using different switch function (straight, swap, copy, etc.)

In multistage switch based system all inputs are connected to all outputs in such a way that no two-processor attempt to access the same memory at the same time. But the problem of contention, at a switch, arises when some memory modules are contested by some fixed processor. In this situation only one request is allowed to access and rest of the requests are dropped. The processor whose requests were dropped can retry the request or if buffers are attached with each switch the rejected request is forwarded by buffer automatically for transmission. This Multistage interconnection networks also called store-and-forward networks.

1.4 TYPES OF MULTIPROCESSOR OPERATING SYSTEMS

The multiprocessor operating systems are complex in comparison to multiprograms on an uniprocessor operating system because multiprocessor executes tasks concurrently.

Therefore, it must be able to support the concurrent execution of multiple tasks to increase processors performance.

Depending upon the control structure and its organisation the three basic types of multiprocessor operating system are:

1) Separate supervisor
2) Master-slave
3) Symmetric Supervision

1.4.1 Separate Supervisors

In separate supervisor system each process behaves independently. Each system has its own operating system which manages local input/output devices, file system and memory well as keeps its own copy of kernel, supervisor and data structures, whereas some common data structures also exist for communication between processors. The access protection is maintained, between processor, by using some synchronization mechanism like semaphores. Such architecture will face the following problems:

1) Little coupling among processors.
2) Parallel execution of single task.
3) During process failure it degrades.
4) Inefficient configuration as the problem of replication arises between supervisor/kernel/data structure code and each processor.

1.4.2 Master-Slave

In master-slave, out of many processors one processor behaves as a master whereas others behave as slaves. The master processor is dedicated to executing the operating system. It works as scheduler and controller over slave processors. It schedules the work and also controls the activity of the slaves. Therefore, usually data structures are stored in its private memory. Slave processors are often identified and work only as a schedulable pool of resources, in other words, the slave processors execute application programmes.

This arrangement allows the parallel execution of a single task by allocating several subtasks to multiple processors concurrently. Since the operating system is executed by only master processors this system is relatively simple to develop and efficient to use. Limited scalability is the main limitation of this system, because the master processor become a bottleneck and will consequently fail to fully utilise slave processors.

1.4.3 Symmetric

In symmetric organisation all processors configuration are identical. All processors are autonomous and are treated equally. To make all the processors functionally identical, all the resources are pooled and are available to them. This operating system is also symmetric as any processor may execute it. In other words there is one copy of kernel that can be executed by all processors concurrently. To that end, the whole process is needed to be controlled for proper interlocks for accessing scarce data structure and pooled resources.

The simplest way to achieve this is to treat the entire operating system as a critical section and allow only one processor to execute the operating system at one time. This method is called 'floating master' method because in spite of the presence of many processors only one operating system exists. The processor that executes the operating system has a special role and acts as a master. As the operating system is not bound to any specific processor, therefore, it floats from one processor to another.
Parallel execution of different applications is achieved by maintaining a queue of ready processors in shared memory. Processor allocation is then reduced to assigning the first ready process to first available processor until either all processors are busy or the queue is emptied. Therefore, each idled processor fetches the next work item from the queue.

1.5 MULTIPROCESSOR OPERATING SYSTEM FUNCTIONS AND REQUIREMENTS

A multiprocessor operating system manages all the available resources and schedule functionality to form an abstraction it will facilitates programme execution and interaction with users.

A processor is one of the important and basic types of resources that need to be manage. For effective use of multiprocessors the processor scheduling is necessary. Processors scheduling undertakes the following tasks:

(i) Allocation of processors among applications in such a manner that will be consistent with system design objectives. It affects the system throughput. Throughput can be improved by co-scheduling several applications together, thus availing fewer processors to each.

(ii) Ensure efficient use of processors allocation to an application. This primarily affects the speedup of the system.

The above two tasks are somehow conflicting each other because maximum speedup needs dedication of a large proportion of a systems processors to a single application which will decrease throughput of the system. Due to the difficulties of automating the process, the need for explicit programmer direction arises to some degree. Generally the language translators and preprocessors provide support for explicit and automatic parallelism. The two primary facets of OS support for multiprocessing are:

(i) Flexible and efficient interprocess and interprocessor synchronization mechanism, and

(ii) Efficient creation and management of a large number of threads of activity, such as processes or threads.

The latter aspect is important because parallelism is often accomplished by splitting an application into separate, individually executable subtasks that may be allocated to different processors.

The **Memory management** is the second basic type of resource that needs to be managed. In multiprocessors system memory management is highly dependent on the architecture and inter-connection scheme.

- In loosely coupled systems memory is usually handled independently on a pre-processor basis whereas in multiprocessor system shared memory may be simulated by means of a message passing mechanism.
- In shared memory systems the operating system should provide a flexible memory model that facilitates safe and efficient access to share data structures and synchronization variables.

A multiprocessor operating system should provide a hardware independent, unified model of shared memory to facilitate porting of applications between different multiprocessor environments. The designers of the mach operating system exploited the duality of memory management and inter-process communication.

The third basic resource is **Device Management** but it has received little attention in multiprocessor systems to date, because earlier the main focus point is speedup of compute intensive application, which generally do not generate much input/output after the initial loading. Now, multiprocessors are applied for more balanced general-purpose applications, therefore, the input/output requirement increases in proportion with the realised throughput and speed.

1.6 MULTIPROCESSOR SYNCHRONIZATION

Multiprocessor system facilitates parallel program execution and read/write sharing of data and thus may cause the processors to concurrently access location in the shared memory. Therefore, a correct and reliable mechanism is needed to serialize this access. This is called synchronization mechanism. The mechanism should make access to a shared data structure appear atomic with respect to each other. In this section, we introduce some new mechanism and techniques suitable for synchronization in multiprocessors.

1.6.1 Test-and-Set

The test-and-set instruction automatically reads and modifies the contents of a memory location in one memory cycle. It is as follows:

```
function test-and-set (var m: Boolean); Boolean;
begin
        test-and set:=m;
        m:=rue
end;
```

The test-and-set instruction returns the current value of variable m (memory location) and sets it to true. This instruction can be used to implement *P* and *V* operations (Primitives) on a binary semaphore, *S*, in the following way (*S* is implemented as a memory location):

```
P(S): while Test-and-Set (S) do nothing;
V(S): S:=false;
```

Initially, *S* is set to false. When a *P(S)* operation is executed for the first time, test-and-set*(S)* returns a false value (and sets *S* to true) and the "while" loop of the *P(S)* operation terminates. All subsequent executions of *P(S)* keep looping because *S* is true until a *V(S)* operation is executed.

1.6.2 Compare-and-Swap

The compare and swap instruction is used in the optimistic synchronization of concurrent updates to a memory location. This instruction is defined as follows (*r1and r2* are to registers of a processor and *m* is a memory location):

```
function test-and-set (var m: Boolean); Boolean;
var temp: integer;
begin
        temp:=m;
        if temp = r1 then  {m:= r2;z:=1}
        else [r1:= temp; z:=0}
end;
```

If the contents of *r1* and *m* are identical, this instruction assigns the contents of *r2* to *m* and sets *z* to *1*. Otherwise, it assigns the contents of *m* to *r1* and set *z* to *0*. Variable *z* is a flag that indicates the success of the execution. This instruction can be used to synchronize concurrent access to a shared variable.

1.6.3 Fetch-and-Add

The fetch and add instruction is a multiple operation memory access instruction that automatically adds a constant to a memory location and returns the previous contents of the memory location. This instruction is defined as follows:

```
Function Fetch-and-add (m: integer; c: integer);
Var temp: integer;
Begin
        Temp:= m;
        M:= m + c;
        Return (temp)
end;
```

This instruction is executed by the hardware placed in the interconnection network not by the hardware present in the memory modules. When several processors concurrently execute a fetch-and-add instruction on the same memory location, these instructions are combined in the network and are executed by the network in the following way:

- A single increment, which is the sum of the increments of all these instructions, is added to the memory location.
- A single value is returned by the network to each of the processors, which is an arbitrary serialisation of the execution of the individual instructions.
- If a number of processors simultaneously perform fetch-and-add instructions on the same memory location, the net result is as if these instructions were executed serially in some unpredictable order.

The fetch-and-add instruction is powerful and it allows the implementation of P and V operations on a general semaphore, S, in the following manner:

```
P(S): while (Fetch-add-Add(S, -1)< 0) do
begin
        Fetch-and-Add (S, 1);
        while (S<0) do nothing;
end;
```

The outer "while-do" statement ensures that only one processor succeeds in decrementing S to 0 when multiple processors try to decrement variable S. All the unsuccessful processors add 1 back to S and try again to decrement it. The "while-do" statement forces an unsuccessful processor to wait (before retrying) until S is greater then 0.

V(S): *Fetch-and-Add (S, 1).*

☞ **Check Your Progress 1**

1) What is the difference between a loosely coupled system and a tightly coupled system? Give examples.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What is the difference between symmetric and asymmetric multiprocessing?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

1.7 SUMMARY

Multiprocessors systems architecture provides higher computing power and speed. It consists of multiple processors that putting power and speed to the system. This system can execute multiple tasks on different processors concurrently. Similarly, it can also execute a single task in parallel on different processors. The design of interconnection networks includes the bus, the cross-bar switch and the multistage interconnection networks. To support parallel execution this system must effectively schedule tasks to various processors. And also it must support primitives for process synchronization and virtual memory management. The three basic configurations of multiprocessor operating systems are: Separate supervisors, Master/slave and Symmetric. A multiprocessor operating system manages all the available resources and schedule functionality to form an abstraction. It includes Process scheduling, Memory management and Device management. Different mechanism and techniques are used for synchronization in multiprocessors to serialize the access of pooled resources. In this section we have discussed Test-and-Set, Compare-and-Swap and Fetch-and-Add techniques of synchronization.

1.8 SOLUTIONS/ANSWERS

1) One feature that is commonly characterizing tightly coupled systems is that they share the clock. Therefore, multiprocessors are typically tightly coupled but distributed workstations on a network are not.

   Another difference is that: in a tightly-coupled system, the delay experienced when a message is sent from one computer to another is short, and data rate is high; that is, the number of bits per second that can be transferred is large. In a loosely-coupled system, the opposite is true: the intermachine message delay is large and the data rate is low.

   For example, two CPU chips on the same printed circuit board and connected by wires etched onto the board are likely to be tightly coupled, whereas two computers connected by a 2400 bit/sec modem over the telephone system are certain to be loosely coupled.

2) The difference between symmetric and asymmetric multiprocessing: all processors of symmetric multiprocessing are peers; the relationship between processors of asymmetric multiprocessing is a master-slave relationship. More specifically, each CPU in symmetric multiprocessing runs the same copy of the OS, while in asymmetric multiprocessing, they split responsibilities typically, therefore, each may have specialised (different) software and roles.

1.9 FURTHER READINGS

1) Singhal Mukesh and Shivaratri G. Niranjan, *Advanced Concepts in Operating Systems*, TMGH, 2003, New Delhi.

2) Hwang, K. and F Briggs, *Multiprocessors Systems Architectures*, McGraw-Hill, New York.

3) Milenkovic, M., *Operating Systems: Concepts and Design*, TMGH, New Delhi.

4) Silberschatz, A., J. Peterson, and D. Gavin, *Operating Systems Concepts*, 3rd ed., Addison Wesley, New Delhi.

UNIT 2 DISTRIBUTED OPERATING SYSTEMS

Structure **Page Nos.**

2.0 INTRODUCTION

In the earlier unit we have discussed the Multiprocessor systems. In the process coupling we have come across the tightly coupled systems and loosely coupled systems. In this unit we concentrate on the loosely coupled systems called as the distributed systems. Distributed computing is the process of aggregating the power of several computing entities to collaboratively run a computational task in a transparent and coherent way, so that it appears as a single, centralised system.
The easiest way of explaining what distributed computing is all about, is by naming a few of its properties:

- Distributed computing consists of a network if more or less independent or autonomous nodes.
- The nodes do *not* share primary storage (i.e., RAM) or secondary storage (i.e., disk) - in the sense that the nodes cannot directly access another node's disk or RAM. Think about it in contrast to a multiprocessors machine where the different "nodes" (CPUs)) share the same RAM and secondary storage by using a common bus.
- A well designed distributed system does not crash if a node goes down.
- If you are to perform a computing task which is parallel in nature, scaling your system is a lot cheaper by adding extra nodes, compared to getting a faster single machine. Of course, if your processing task is highly non-parallel (every result depends on the previous), using a distributed computing system may not be very beneficial.

Before going into the actual details of the distributed systems let us study how distributed computing evolved.

2.1 OBJECTIVES

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

- define a distributed system, key features and design goals of it;
- explain the design issues involved in the distributed systems;
- describe the models of distributed system structure;
- explain the mutual exclusion in distributed systems, and
- define RPC, its usage and limitations.

2.2 HISTORY OF DISTRIBUTED COMPUTING

Distributed computing began around 1970 with the emergence of two technologies:

- minicomputers, then workstations, and then PCs.
- computer networks (eventually Ethernet and the Internet).

With the minicomputer (e.g., Digital's PDP-11) came the timesharing operating system (e.g., MULTICS, Unix, RSX, RSTS) - many users using the same machine but it looks to the users as if they each have their own machine.

The problem with mini-computers was that they were slower than the mainframes made by IBM, Control Data, Univac, etc. As they became popular they failed to scale to large number of users as the mainframes could. The way to scale mini-computers was to buy more of them. The trend toward cheaper machines made the idea of having many minis as a feasible replacement for a single mainframe and made it possible to contemplate a future computing environment where every user had their own computer on their desk, which is a computer workstation.

Work on the first computer workstation began in 1970 at Xerox Corporation's Palo Alto Research Center (PARC). This computer was called the Alto. Over the next 10 years, the computer system's group at PARC would invent almost everything that's interesting about the computer workstations and personal computers we use today. The Alto introduced ideas like the workstation, bit-mapped displays (before that computer interfaces were strictly character-based) and the mouse.

Other innovations that came from PARC from 1970 to 1980 include Ethernet, the first local-area network, window- and icon-based computing (the inspiration for the Apple Lisa, the progenitor of the Macintosh, which in turn inspired Microsoft Windows and IBM OS/2 came from a visit to PARC), the first distributed fileserver XDFS , the first print server, one of the first distributed services: Grapevine (a messaging and authentication system), object-oriented programming (Smalltalk) and Hoare's condition variables and monitors were invented as part of the Mesa programming language used in the Cedar system.

2.2.1 Workstations−Networks

The vision of the PARC research (and the commercial systems that followed) was to replace the timesharing mini-computer with single-user workstations. The genesis of this idea is that it made computers (i.e., workstations and PCs) into a commodity item that like a TV or a car could be produced efficiently and cheaply. The main costs of a computer are the engineering costs of designing it and designing the manufacturing process to build it. If you build more units, you can amortised the engineering costs better and thus make the computers cheaper. This is a very important idea and is the main reason that distribution is an excellent way to build a cheap scalable system.

2.2.2 Wide-Scale Distributed Computing and the Internet

Another type of distributed computing that is becoming increasingly important today is the sort of wide-scale distribution possible with the Internet.

At roughly the same time as the workstation and local-area network were being invented, the U.S. Department of Defence was putting tons of U.S. taxpayer money to work to set up a world-wide communication system that could be used to support distributed science and engineering research needed to keep the Defence Dept. supplied with toys. They were greatly concerned that research assets not be centrally located, as doing so would allow one well-placed bomb to put them out of business. This is an example of another benefit of distributed systems: fault tolerance through replication.

2.3 DISTRIBUTED SYSTEMS

Multiprocessor systems have more than one processing unit sharing memory/peripheral devices. They have greater computing power, and higher reliability. Multiprocessor systems are classified into two:

- **Tightly-coupled**: Each processor is assigned a specific duty but processors work in close association, possibly sharing one memory module.
- **Loosely-coupled (distributed)**: Each processor has its own memory and copy of the OS.

A distributed computer system is a loosely coupled collection of autonomous computers connected by a network using system software to produce a single integrated computing environment.

A distributed operating system differs from a network of machines each supporting a network operating system in only one way: The machines supporting a distributed operating system are all running under a single operating system that spans the network. Thus, the print spooler might, at some instant, be running on one machine, while the file system is running on others, while other machines are running other parts of the system, and under some distributed operating systems, these software parts may at times migrate from machine to machine.

With network operating systems, each machine runs an entire operating system. In contrast, with distributed operating systems, the entire system is itself distributed across the network. As a result, distributed operating systems typically make little distinction between remote execution of a command and local execution of that same command. In theory, all commands may be executed anywhere; it is up to the system to execute commands where it is convenient.

2.4 KEY FEATURES AND ADVANTAGES OF A DISTRIBUTED SYSTEM

The following are the key features of a distributed system:

- They are Loosely coupled
  - remote access is many times slower than local access
- Nodes are autonomous
  - workstation resources are managed locally
- Network connections using system software
  - remote access requires explicit message passing between nodes
  - messages are CPU to CPU
  - protocols for reliability, flow control, failure detection, etc., implemented in software
  - the *only* way two nodes can communicate is by sending and receiving network messages−this differs from a hardware approach in which hardware signalling can be used for flow control or failure detection.

2.4.1 Advantages of Distributed Systems over Centralised Systems

- Better price/performance than mainframes
- More computing power
  - sum of the computing power of the processors in the distributed system may be greater than any single processor available (parallel processing)
- Some applications are inherently distributed
- Improved reliability because system can survive crash of one processor
- Incremental growth can be achieved by adding one processor at a time
- Shared ownership facilitated.

2.4.2 Advantages of Distributed Systems over Isolated PCs

- Shared utilisation of resources.
- Communication.
- Better performance and flexibility than isolated personal computers.
- Simpler maintenance if compared with individual PC's.

2.4.3 Disadvantages of Distributed Systems

Although we have seen several advantages of distributed systems, there are certain disadvantages also which are listed below:

- Network performance parameters.
- *Latency*: Delay that occurs after a send operation is executed before data starts to arrive at the destination computer.
- *Data Transfer Rate*: Speed at which data can be transferred between two computers once transmission has begun.
- *Total network bandwidth*: Total volume of traffic that can be transferred across the network in a give time.
- Dependency on reliability of the underlying network.
- Higher security risk due to more possible access points for intruders and possible communication with insecure systems.
- Software complexity.

2.5 DESIGN GOALS OF DISTRIBUTED SYSTEMS

In order to design a good distributed system. There are six key design goals. They are:

- Concurrency
- Scalability
- Openness
- Fault Tolerance
- Privacy and Authentication
- Transparency.

Let us discuss them one by one.

2.5.1 Concurrency

A server must handle many client requests at the same time. Distributed systems are naturally concurrent; that is, there are multiple workstations running programs independently and at the same time. Concurrency is important because any distributed service that isn't concurrent would become a bottleneck that would serialise the actions of its clients and thus reduce the natural concurrency of the system.

2.5.2 Scalability

The goal is to be able to use the same software for different size systems. A distributed software system is scalable if it can handle increased demand on any part of the system (i.e., more clients, bigger networks, faster networks, etc.) without a change to the software. In other words, we would like the engineering impact of increased demand to be proportional to that increase. Distributed systems, however, can be built for a very wide range of scales and it is thus not a good idea to try to build a system that can handle everything. A local-area network file server should be built differently from a Web server that must handle millions of requests a day from throughout the world. The key goal is to understand the target system's expected size and expected growth and to understand how the distributed system will scale as the system grows.

2.5.3 Openness

Two types of openness are important: non-proprietary and extensibility.
Public protocols are important because they make it possible for many software manufacturers to build clients and servers that will be able to talk to each other. Proprietary protocols limit the "players" to those from a single company and thus limit the success of the protocol.

A system is extensible if it permits customisations needed to meet unanticipated requirements. Extensibility is important because it aids scalability and allows a system to survive over time as the demands on it and the ways it is used change.

2.5.4 Fault Tolerance

It is critically important that a distributed system be able to tolerate "partial failures". Why is it so important? Two reasons are as follows:

- *Failures are more harmful:* Many clients are affected by the failure of a distributed service, unlike a non-distributed system in which a failure affects only a single node.

- *Failures are more likely:* A distributed service depends on many components (workstation nodes, network interfaces, networks, switches, routers, etc.) all of which must work. Furthermore, a client will often depend on multiple distributed services (e.g., multiple file systems or databases) in order to function properly. The probability that such a client will experience a failure can be approximated as the sum of the individual failure probabilities of everything that it depends on. Thus, a client that depends on N components (hardware or software) that each have failure probability P will fail with probability roughly $N*P$. (This approximation is valid for small values of P. The exact failure probability is (1−(1−P)^N).)

There are two aspects of failure tolerance to be studied as shown below:

Recovery

- A failure shouldn't cause the loss (or corruption) of critical data, or computation.
- After a failure, the system should recover critical data, even data that was being modified when the failure occurred. Data that survives failures is called "persistent" data.
- Very long-running computations must also be made recoverable in order to restart them where they left off instead of from the beginning.
- For example, if a fileserver crashes, the data in the file system it serves should be intact after the server is restarted.

Availability

- A failure shouldn't interrupt the service provided by a critical server.
- This is a bit harder to achieve than recovery. We often speak of a highly-available service as one that is almost always available even if failures occur.
- The main technique for ensuring availability is service replication.
- For example, a fileserver could be made highly available by running two copies of the server on different nodes. If one of the servers fails, the other should be able to step in without service interruption.

2.5.5 Privacy and Authentication

Privacy is achieved when the sender of a message can control what other programs (or people) can read the message. The goal is to protect against eavesdropping. For example, if you use your credit card to buy something over the Web, you will probably want to prevent anyone but the target Web server from reading the message that contains your credit card account number. Authentication is the process of ensuring that programs can know who they are talking to. This is important for both clients and servers.

For clients authentication is needed to enable a concept called trust. For example, the fact that you are willing to give your credit card number to a merchant when you buy something means that you are implicitly trusting that merchant to use your number according to the rules to which you have both agreed (to debit your account for the amount of the purchase and give the number to no one else). To make a Web purchase, you must trust the merchant's Web server just like you would trust the merchant for an in-person purchase. To establish this trust, however, you must ensure that your Web browser is really talking to the merchant's Web server and not to some other program that's just pretending to be their merchant.

For servers authentication is needed to enforce access control. For a server to control who has access to the resources it manages (your files if it is a fileserver, your money if it is a banking server), it must know who it is talking to. A Unix login is a crude example of an authentication used to provide access control. It is a crude example because a remote login sends your username and password in messages for which privacy is not guaranteed. It is thus possible, though usually difficult, for someone to eavesdrop on those messages and thus figure out your username and password.

For a distributed system, the only way to ensure privacy and authentication is by using cryptography.

2.5.6 Transparency

The final goal is transparency. We often use the term **single system image** to refer to this goal of making the distributed system look to programs like it is a tightly coupled (i.e., single) system.

This is really what a distributed system software is all about. We want the system software (operating system, runtime library, language, compiler) to deal with all of the complexities of distributed computing so that writing distributed applications is as easy as possible.

Achieving complete transparency is difficult. There are eight types, namely:

- ***Access Transparency*** enables local and remote resources to be accessed using identical operations
- ***Location Transparency*** enables resources to be accessed without knowledge of their (physical) location. Access transparency and location transparency are together referred to as network transparency.
- ***Concurrency Transparency*** enables several processes to operate concurrently using shared resources without interference between them.
- ***Replication Transparency*** enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.
- ***Failure Transparency*** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
- ***Mobility Transparency*** allows the movement of resources and clients within a system without affecting the operation of users or programs.
- ***Performance Transparency*** allows the system to be reconfigured to improve performance as loads change
- ***Scaling Transparency*** Transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms

2.6 DESIGN ISSUES INVOLVED IN DISTRIBUTED SYSTEMS

There are certain design issues to be considered for distributed systems. They are:

a) Naming
b) Communication
c) Software Structure
d) Workload Allocation
e) Consistency Maintenance

Let us discuss the issues briefly:

2.6.1 Naming

- A name is a string of characters used to identify and locate a distributed resource.
- An identifier is a special kind of name that is used directly by the computer to access the resource.

  For example the identifier for a Unix server would include at least (1) an IP address and (2) a port number. The IP address is used to find the node that runs the server and the port number identifies the server process on that node.
- Resolution is the process of turning a name into an identifier. Resolution is performed by a Name Server. It is also called “binding” (as in binding a name to a distributed service).

For example, an IP domain name (e.g., cs.ubc.ca) is turned into a IP address by the IP Domain Name Server (DNS), a distributed hierarchical server running in the Internet.

2.6.2 Communication

- Getting different processes to talk to each other
- Messages
- Remote method invocation.

2.6.3 Software Structure

- The main issues are to choose a software structure that supports our goals, particularly the goal of openness.
- We thus want structures that promote extensibility and otherwise make it easy to program the system.
- Alternatives are:
  - A monolithic structure is basically a big pile of code; it is not so desirable because it is hard to extend or reason about a system like that.
  - A modular structure divides the system into models with well-defined interfaces that define how the models interact. Modular systems are more extensible and easier to reason about than monolithic systems.
  - A layered structure is a special type of modular structure in which modules are organised into layers (one on top of the other). The interaction between layers is restricted such that a layer can communicate directly with only the layer immediately above and the layer immediately below it. In this way, each layer defines an abstraction that is used by the layer immediately above it. Clients interact only with the top layer and only the bottom layer deals with the hardware (e.g., network, disk, etc.) Network protocols have traditionally been organised as layers (as we will see in the next class) and for this reason we often refer to these protocols as "protocol stacks".
  - Operating systems can be either monolithic (e.g., UNIX and Windows) or modular (e.g., Mach and Windows NT). A modular operating system is called a micro kernel. A micro-kernel OS has a small minimalist kernel that includes as little functionality as possible. OS services such as VM, file systems, and networking are added to the system as separate servers and they reside in their own user-mode address spaces outside the kernel.

Let us study more details regarding the software structure in section 2.7 of this unit.

2.6.4 Workload Allocation

- The key issue is load balancing: The allocation of the network workload such that network resources (e.g., CPUs, memory, and disks) are used efficiently.

For CPUs, there are two key approaches. They are*:*

Processor Pools

Put all of the workstations in a closet and give the users cheap resource-poor X terminals. User processes are allocated to the workstations by a distributed operating system that controls all of the workstations. Advantage is that process scheduling is simplified and resource utilisation more efficient because workstations aren't "owned" by anyone. The OS is free to makes scheduling decisions based solely on the goal of getting the most work done. Disadvantage is that, these days, powerful workstations aren't much more expensive than cheap X terminals. So if each user has a powerful workstation instead of a cheap X terminal, then we don't need the machines in the closet and we thus have the idle-workstation model described below:

Idle Workstations

Every user has a workstation on their desk for doing their work. These workstations are powerful and are thus valuable resources. But, really people don't use their workstations all of the time. There's napping, lunches, reading, meetings, etc. - lots of times when workstations are **idle**. In addition, when people are using their workstation, they often don't need all of its power (e.g., reading mail doesn't require a 200-Mhz CPU and 64-Mbytes of RAM). The goal then is to move processes from active workstations to idle workstations to balance the load and thus make better use of network resources. But, people "own" (or at least feel like they own) their workstations. So a key issue for using idle workstations is to avoid impacting workstation users (i.e., we can't slow them down). So what happens when I come back from lunch and find your programs running on my machine? I'll want your processes moved elsewhere NOW. The ability to move an active process from one machine to another is called **process migration**. Process migration is necessary to deal with the user-returning-from-lunch issue and is useful for rebalancing network load as the load characterises a network change over time.

2.6.5 Consistency Maintenance

The final issue is consistency. There are four key aspects of consistency: atomicity, coherence, failure consistency, and clock consistency.

Atomicity

- The goal is to provide the "all-or-nothing" property for sequences of operations.
- Atomicity is important and difficult for distributed systems because operations can be messages to one or more servers. To guarantee atomicity thus requires the coordination of several network nodes.

Coherence

Coherence is the problem of maintaining the consistency of replicated data.
We have said that replication is a useful technique for (1) increasing availability and (2) improving performance.

When there are multiple copies of an object and one of them is modified, we must ensure that the other copies are updated (or invalidated) so that no one can erroneously read an out-of-date version of the object.

A key issue for providing coherence is to deal with the event-ordering problem.

Failure Consistency

- We talked last time about the importance of the goal of failure tolerance. To build systems that can handle failures, we need a model that clearly defines what a failure is. There are two key types of failures: *Fail-stop and Byzantine.*
- ***Fail-stop*** is a simplified failure model in which we assume that the only way a component will fail is by stopping. In particular, a will never fail by giving a wrong answer.
- ***Byzantine*** failure is a more encompassing model that permits failures in which a failed node might not stop but might just start giving the wrong answers. Techniques for dealing with Byzantine failure typically involve performing a computation redundantly (and often in different ways with different software). The system then compares all of the answers generated for a given computation and thus detects when one component starts giving the wrong answer. The ability to detect Byzantine failures is important for many safety-critical systems such as aircraft avionics.
- In this class we will assume failures are Fail-Stop; this is a common assumption for distributed systems and greatly simplifies failure tolerance.

Clock Consistency

- Maintaining a consistent view of time needed to order network events is critical and challenging. This becomes further difficult when the nodes of the distributed system are geographically apart. Let us study the scheme provided by Lamport on Ordering of events in a distributed environment in the next section.

2.6.6 Lamport's Scheme of Ordering of Events

Lamport proposed a scheme to provide ordering of events in a distributed environment using logical clocks. Because it is impossible to have perfectly synchronized clocks and global time in a distributed system, it is often necessary to use logical clocks instead.

Definitions:

Happened Before Relation (->): This relation captures causal dependencies between events, that is, whether or not events have a cause and effect relation. This relation (->) is defined as follows:

- a -> b, if *a* and *b* are in the same process and *a* Occurred before *b*.
- a -> b, if *a* is the event of sending a message and *b* is the receipt of that message by another process.

If a -> b and b -> c, then a -> c, that is, the relation has the property of transitivity.

Causally Related Events: If *event a -> event b*, then *a* casually affects *b*.

Concurrent Events: Two distinct events *a* and *b* are concurrent (*a* || *b*) if (not) *a -> b* and (not) b -> a. That is, the events have no causal relationship. This is equivalent to b || a.

For any two events *a and b* in a system, only one of the following is true: a -> b, b -> a, or a || b.

Lamport introduced a system of logical clocks in order to make the -> relation possible. It works like this: Each process P_i in the system has its own clock C_i. C_i can be looked at as a function that assigns *a* number, $C_i(a)$ to an event *a*. This is the timestamp of the event *a* in process P_i. These numbers are not in any way related to physical time -- that is why they are called logical clocks. These are generally implemented using counters, which increase each time an event occurs. Generally, an event's timestamp is the value of the clock at the time it occurs.

Conditions Satisfied by the Logical Clock system:

For any events *a* and *b*, if a -> b, then $C(a) < C(b)$. This is true if two conditions are met:

- If a occurs before b, then $C_i(a) < C_i(b)$.
- If a is a message sent from P_i and b is the receipt of that same message in P_j, then $C_i(a) < C_j(b)$.

Implementation Rules Required:

Clock C_i is incremented for each event: $C_i := C_i + d$ $(d > 0)$
if *a* is the event of sending a message from one process to another, then the receiver sets its clock to the max of its current clock and the sender's clock - that is, $C_j := \max(C_j, t_m + d)$ $(d > 0)$.

2.7 DISTRIBUTED SYSTEM STRUCTURE

There are three main alternative ways to structure a distributed application. Each of them defines a different **application model** for writing distributed programs.

The three distributed system application models are:

- Client / Server (and RPC)
- Distributed objects
- Distributed shared memory.

Each model requires a different set of “system” software to support applications written in that style. This system software consists of a combination of operating system and runtime-library code and support from languages and compilers. What all of this code does is translate the distributed features of the particular application model into the sending and receiving of network messages. Because, remember that the only way that nodes (workstations/PCs) can communicate in a distributed system is by sending and receiving messages.

2.7.1 Application Model 1: Client Server

- A designated node exports a “service”; this node is called the “server”.
- Nodes that import and use the service are called “clients”.
- Clients communicate with servers using a “request-response” protocol.

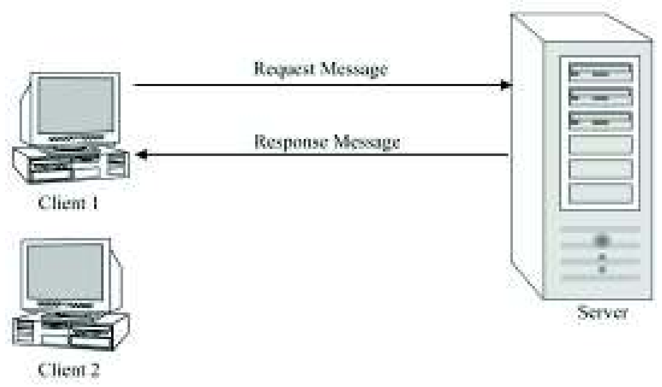


Figure 1: Client Server Model

- Request-response differs from other protocols.
    - Peer-to-peer protocols don’t have a designated server.
    - Streaming protocols such as TCP/IP work differently as we will see a couple of classes from now.
- We should really use the terms client and server to refer to the pieces of the software that implement the server and not to the nodes themselves. Why? Because a node can actually be both a client and server. Consider NFS for example. One workstation might export a local filesystem to other nodes (i.e., it is a server for the file system) and import other filesystems exported by remote nodes (i.e., it is a client for these filesystems).
- Application interface can be either send/receive or RPC
    - **Send/receive :** The Unix socket interface is an example. Clients communicate with server by building a message (a sequence of characters) that it then explictly sends to the server. Clients receive messages by issuing a “receive” call.
      We’ll talk about this on Friday.
    - **RPC :** A remote procedure call interface allows a client to request a server operation by making what looks like a procedure call. The system software translates this RPC into a message send to the server and a message receives to wait for the reply. The advantages of RPC over send/receive is that the system hides some of the details of message passing from programs: it’s easier to call a procedure than format a

message and then explicitly send it and wait for the reply. (Study more on RPC given in section 2.9 of this unit).

2.7.2 Application Model 2: Distributed Objects

- Similar to RPC-based client-server.
- Language-level objects (e.g., C++, Java, Smalltalk) are used to encapsulate data and functions of a distributed service.
- Objects can reside in either the memory of a client or a server.
- Clients communicate with servers through objects that they "share" with the server. These shared objects are located in a client's memory and look to the client just like other "local" objects. We call these objects "remote" objects because they represent a remote service (i.e., something at the server). When a client invokes a method of a remote object, the system might do something locally or it might turn the method invocation into an RPC to the server. (Recall that in the object-oriented world, procedures are called methods and procedure callls are called method invocations).
- There are two main advantages of Distributed Objects over RPC: (1) objects hide even more of the details of distribution than RPC (more next week) and (2) objects allow clients and servers to communicate using in two different ways: function shipping (like RPC) and data shipping.
  - ***function shipping*** means that a client calls the server and asks it to perform a function; this is basically like RPC.
  - ***data shipping*** means that the server sends some data to the client (stores it as part of an object) and the client then performs subsequent functions locally instead of having to call the server every time it wants to do something.

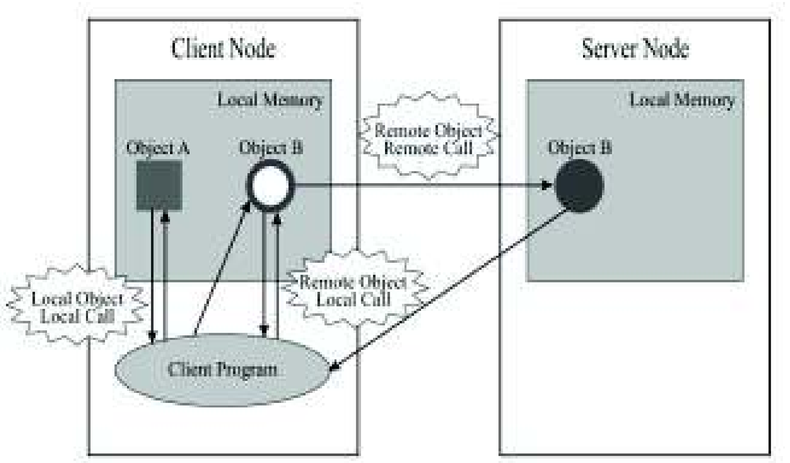


Figure 2: Distributed objects models

- For example, in the picture above, a client program running can access local object A and remote object B (controlled by the server node) in exactly the same way; the arrows show flow of a procedure call into the object and a return from it. The server can design B so that calls are turned into remote calls just like RPC (red) or it can ship some data to the client with the client's copy of the object and thus allow some calls to run locally on the client (orange).
- More on distributed objects later in the course.

2.7.3 Application Model 3: Distributed Shared Memory

- Usually used for peer-to-peer communication instead of client-server.
- Clients communicate with each other through shared variables.

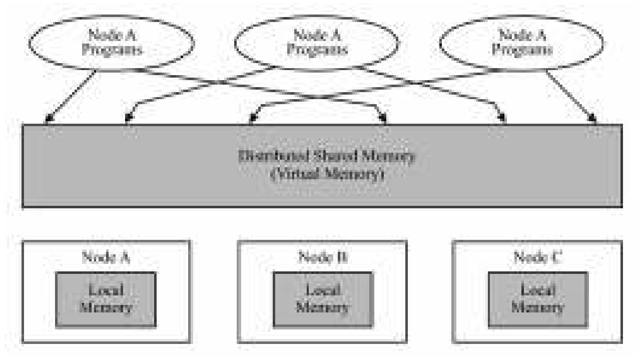


Figure 3: Distributed shared memory model

- The system implements the illusion of shared memory and translates accesses to shared data into the appropriate messages.
- To a program, the distributed system looks just like a shared-memory multiprocessor.
- The advantage is that it is really easy to program: hides all of the details of distribution.
- The disadvantage is that it often hides too much. As far as a program knows, everything in its memory is local. But really, some parts of its memory are stored on or shared with a remote node. Access to this remote data is very SLOW compared with accessing local data. For good performance, a program usually needs to know what data is local and what data is remote.
- Another disadvantage is that it is very complicated to implement a distributed-shared memory system that works correctly and performs well.
- We'll talk more about distributed shared memory towards the end of the term.

2.8 MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

When all processes sharing a resource are on the same machine, mutual exclusion is easily assured by the use of a semaphore, spin-lock or other similar shared abstraction. When the processes involved are on different machines, however, mutual exclusion becomes more difficult.

Consider the following example: A number of machines in a network are competing for access to a printer. The printer is so constructed that every line of text sent to the printer must be sent in a separate message, and thus if a process wants to print an entire file, it must obtain exclusive use of the printer, somehow, send all the lines of the file it wishes to print, and then release the printer for use by others on the network.

A trivial solution to this problem is to install a print spooler process somewhere on the net. That print spooler would gather lines of files provided by various applications processes, maintain a queue of completed files that are ready to print, and print one such file at a time. This works, but it introduces some problems: First, the spooler must have buffer capacity to hold the aggregate of all the files that have not yet been printed. Second, the spooler may become a bottleneck, limiting the performance of the entire system, and third, if the processor supporting the spooler is unreliable, the spooler may limit the reliability of the system.

2.8.1 Mutual Exclusion Servers

In the printer example being used here, the problem of storage space in the spooler typically becomes acute with graphics printers. In such a context, it is desirable to block an applications process until the printer is ready to accept data from that applications process, and then let that process directly deliver data to the printer.

For example, an applications process may be coded as follows:

```
Send request for permission to the spooler
Await reply giving permission to print

Loop

  send data directly to printer
End Loop

Send notice to spooler that printing is done
```

If all users of the spooler use this protocol, the spooler is no longer serving as a spooler, it is merely serving as a mutual exclusion mechanism! In fact, it implements exactly the same semantics as a binary semaphore, but it implements it using a client server model.

The process implementing a semaphore using message passing might have the following basic structure:

```
Loop
  Await message from client
  Case message type of
   P:
     If count > 0
                 Send immediate reply
                 count = count - 1
     Else
       Enqueue identity of client
     End if
   V:
     If queue is empty,
                 count = count + 1
     Else
       Dequeue one blocked client
       Send a delayed reply
     End if
  End case
End Loop
```

This requires a count and a queue of return addresses for each semaphore. Note that, by presenting this, we have proven that blocking message passing can be used to implement semaphores. Since we already know that semaphores plus shared memory are sufficient to implement blocking message passing, we have proven the equivalence, from a computation theory viewpoint, of these two models of interprocess communication.

The disadvantage of implementing semaphores using a server process is that server becomes a potential source of reliability problems. If we can build a mutual exclusion algorithm that avoids use of a dedicated server, for example, by having the processes that are competing for entry to a critical section negotiate directly with each other, we can potentially eliminate the reliability problem.

2.8.2 Token Based Mutual Exclusion

One alternative to the mutual-exclusion server given above is to arrange the competing processes in a ring and let them exchange a token. If a process receives the token and does not need exclusive use of the resource, it must pass the token on to the next process in the ring. If a process needs exclusive use of the resource, it waits for the token and then holds it until it is done with the resource, at which point it puts the token back in circulation.

This is the exact software analog of a token ring network. In a token ring network, only one process at a time may transmit, and the circulating token is used to assure this, exactly as described. The token ring network protocol was developed for the hardware level or the link-level of the protocol hierarchy. Here, we are proposing building a virtual ring at or above the transport layer and using essentially the same token passing protocol.

This solution is not problem free. What if the token is lost? What if a process in the ring ceases transmission? Nonetheless, it is at the root of a number of interesting and useful distributed mutual exclusion algorithms. The advantage of such distributed algorithms is that they do not rest on a central authority, and thus, they are ideal candidates for use in fault tolerant applications.

An important detail in the token-based mutual exclusion algorithm is that, on receiving a token, a process *must* immediately forward the token if it is not waiting for entry into the critical section. This may be done in a number of ways:

- Each process could periodically check to see if the token has arrived. This requires some kind of non-blocking read service to allow the process to poll the incoming network connection on the token ring. The UNIX FNDELAY flag allows non-blocking read, and the Unix select() kernel call allows testing an I/O descriptor to see if a read from that descriptor would block; either of these is sufficient to support this polling implementation of the token passing protocol. The fact that UNIX offers two such mechanisms is good evidence that these are afterthoughts added to Unix after the original implementation was complete.
- The receipt of an incoming token could cause an interrupt. Under UNIX, for example, the SIGIO signal can be attached to a socket or communications line (see the FASYNC flag set by fcntl). To await the token, the process could disable the SIGIO signal and do a blocking read on the incoming token socket. To exit the critical section, the process could first enable SIGIO and then send the token. The SIGIO handler would read the incoming token and forward it before returning.
- A thread or process could be dedicated to the job of token management. We'll refer to such a thread or process as the mutual exclusion agent of the application process. Typically, the application would communicate with its agent using shared memory, semaphores, and other uni-processor tools, while the agent speaks to other agents over the net. When the user wants entry to the critical section, it sets a variable to "let me in" and then does a wait on the entry semaphore it shares with the agent. When the user is done, the user sets the shared variable to "done" and then signals the go-on semaphore it shares with the agent. The agent always checks the shared variable when it receives the token, and only forwards it when the variable is equal to "done".

2.8.3 Lamport's Bakery Algorithm

One decentralised algorithm in common use, for example, in bakeries, is to issue numbers to each customer. When the customers want to access the scarce resource (the clerk behind the counter), they compare the numbers on their slips and the user with the lowest numbered slip wins.

The problem with this is that there must be some way to distribute numbers, but this has been solved. In bakeries, we use a very small server to distribute numbers, in the form of a roll of tickets where conflicts between two customers are solved by the fact that human hands naturally exclude each other from the critical volume of space that must be occupied to take a ticket. We cannot use this approach for solving the problem on a computer.

Before going on to more interesting implementations for distributing numbers, note that clients of such a protocol may make extensive use of their numbers! For example, if the bakery contains multiple clerks, the clients could use their number to select a clerk (number modulo number of clerks). Similarly, in a FIFO queue implemented with a bounded buffer, the number modulo the queue size could indicate the slot in the buffer to be used, allowing multiple processes to simultaneously place values in the queue.

Lamport's Bakery Algorithm provides a decentralised implementation of the "take a number" idea. As originally formulated, this requires that each competing process share access to an array, but later distributed algorithms have eliminated this shared data structure. Here is the original formulation:

For each process, *i*, there are two values, C[i] and N[i], giving the status of process I and the number it has picked. In more detail:

```
N[i] = 0 --> Process i is not in the bakery.
N[i] > 0 --> Process i has picked a number and is in the bakery.

C[i] = 0 --> Process i is not trying to pick a number.
C[i] = 1 --> Process i is trying to pick a number.

when

   N[i] = min( for all j, N[j] where N[j] > 0 )
   Process i is allowed into the critical section.
```

Here is the basic algorithm used to pick a number:

```
C[i] := 1;
N[i] := max( for all j, N[j] ) + 1;
C[i] := 0;
```

In effect, the customer walks into the bakery, checks the numbers of all the waiting customers, and then picks a number one larger than the number of any waiting customer.

If two customers each walk in at the same time, they are each likely to pick the same number. Lamport's solution allows this but then makes sure that customers notice that this has happened and break the tie in a sensible way.

To help the customers detect ties, each customer who is currently in the process of picking a number holds his hand up (by setting C[i] to 1. s/he pulls down his hand when s/he is done selecting a number -- note that selecting a number may take time, since it involves inspecting the numbers of everyone else in the waiting room.

A process does the following to wait for the baker:

```
Step 1:
while (for some j, C(j) = 1) do nothing;
```

First, wait until any process which might have tied with you has finished selecting their numbers. Since we require customers to raise their hands while they pick numbers, each customer waits until all hands are down after picking a number in order to guarantee that all ties will be cleanly recognised in the next step.

```
Step 2:

  repeat
    W := (the set of j such that N[j] > 0)
     (where W is the set of indeces of waiting processes)
    M := (the set of j in W
          such that N[j] <= N[k]
          for all k in W)
    (where M is the set of process indices with minimum numbers)
    j := min(M)
     (where  is in M and the tie is broken)
  until i = j;
```

Second, wait until your ticket number is the minimum of all tickets in the room. There may be others with this minimum number, but in inspecting all the tickets in the room, you found them! If you find a tie, see if your customer ID number is less than the ID numbers of those with whom you've tied, and only then enter the critical section and meet with the baker.

This is inefficient, because you might wait a bit too long while some other process picks a number after the number you picked, but for now, we'll accept this cost.

If you are not the person holding the smallest number, you start checking again. If you hold the smallest number, it is also possible that someone else holds the smallest number. Therefore, what you've got to do is agree with everyone else on how to break ties.

The solution shown above is simple. Instead of computing the value of the smallest number, compute the minimum process ID among the processes that hold the smallest value. In fact, we need not seek the minimum process ID, all we need to do is use any deterministic algorithm that all participants can agree on for breaking the tie. As long as all participants apply the same deterministic algorithms to the same information, they will arrive at the same conclusion.

To return its ticket, and exit the critical section, processes execute the following trivial bit of code:

```
    N[i] := 0;
```

When you return your ticket, if any other processes are waiting, then on their next scan of the set of processes, one of them will find that it is holding the winning ticket.

Moving to a Distributed Context

In the context of distributed systems, Lamport's bakery algorithm has the useful property that process *i* only modifies its own *N[i]* and *C[i]*, while it must read the entries for all others. In effect, therefore, we can implement this in a context where each process has read-only access to the data of all other processes, and read-write access only to its own data.

A distributed implementation of this algorithm can be produced directly by storing *N[i]* and *C[i]* locally with process *i*, and using message passing when any process wants to examine the values of *N* and *C* for any process other than itself. In this case, each process must be prepared to act as a server for messages from the others requesting the values of its variables; we have the same options for implementing this service as we had for the token passing approach to mutual exclusion. The service could be offered by an agent process, by an interrupt service routine, or by periodic polling of the appropriate incoming message queues.

Note that we can easily make this into a fault tolerant model by using a fault-tolerant client-server protocol for the requests. If there is no reply to a request for the values of process i after some interval and a few retries, we can simply assume that process *i* has failed.

This demonstrates that fault tolerant mutual exclusion can be done without any central authority! This direct port of Lamport's bakery algorithm is not particularly efficient, though. Each process must read the variables of all other processes a minimum of 3 times−once to select a ticket number, once to see if anyone else is in the process of selecting a number, and once to see if it holds the minimum ticket.

For each process contending for entry to the critical section, there are about *6N* messages exchanged, which is clearly not very good. Much better algorithms have been devised, but even this algorithm can be improved by taking advantage of knowledge of the network structure. On an Ethernet or on a tree-structured network, a broadcast can be done in parallel, sending one message to *N* recipients in only a few time units. On a tree-structured network, the reply messages can be merged on the way to the root (the originator of the request) so that sorting and searching for the maximum N or the minimum nonzero N can be distributed efficiently.

2.8.4 Ricart and Agrawala's Mutual Exclusion Algorithm

Another alternative is for anyone wishing to enter a critical section to broadcast their request; as each process agrees that it is OK to enter the section, they reply to the broadcaster saying that it is OK to continue; the broadcaster only continues when all replies are in.

If a process is in a critical section when it receives a request for entry, it defers its reply until it has exited the critical section, and only then does it reply. If a process is not in the critical section, it replies immediately.

This sounds like a remarkably naive algorithm, but with point-to-point communications between N processes, it takes only 2(N−1) messages for a process to enter the critical section, N−1 messages to broadcast the request and N−1 replies.

There are some subtle issues that make the result far from naive. For example, what happens if two processes each ask at the same time? What should be done with requests received while a process is waiting to enter the critical section?

Ricart and Agrawala's mutual exclusion algorithm solves these problems. In this solution, each process has 3 significant states, and its behaviour in response to messages from others depends on its state:

Outside the critical section

The process replies immediately to every entry request.

After requesting entry, awaiting permission to enter.

The process replies immediately to higher priority requests and defers all other replies until exit from the critical section.

Inside critical section.

The process defers all replies until exit from the critical section.

As with Lamport's bakery algorithm, this algorithm has no central authority. Nonetheless, the interactions between a process requesting entry to a critical section and each other process have a character similar to client-server interactions. That is, the interactions take the form of a request followed (possibly some time later) by a reply.

As such, this algorithm can be made fault tolerant by applying the same kinds of tricks as are applied in other client server applications. On receiving a request, a processor can be required to immediately send out either a reply or a negative acknowledgement. The latter says "I got your request and I can't reply yet!"

With such a requirement, the requesting process can wait for either a reply or a negative acknowledgement from every other process. If it gets neither, it can retry the request to that process. If it retries some limited number of times and still gets no answer, it can assume that the distant process has failed and give up on it.

If a process receives two consecutive requests from the same process because acknowledgements have been lost, it must resend the acknowledgement. If a process waits a long time and doesn't get an acknowledgement, it can send out a message saying "are you still there", to which the distant process would reply "I got your request but I can't reply yet". If it gets no reply, it can retry some number of times and then give up on the server as being gone.

If a process dies in its critical section, the above code solves the problem and lets one of the surviving processes in. If a process dies outside its critical section, this code also works.

Breaking Ties in Ricart and Agrawala's algorithm

There are many ways to break ties between processes that make simultaneous requests; all of these are based on including the priority of each requesting process in the request message. It is worth noting that the same alternatives apply to Lamport's bakery algorithm!

A unique process ID can be used as the priority, as was done in Lamport's bakery algorithm. This is a static priority assignment and is almost always needed to break ties in any of the more complex cases. Typically, a process will append its statically assigned process ID to any more interesting information it uses for tiebreaking, thus guaranteeing that if two processes happen to generate the same interesting information, the tie will still be broken.

The number of times the process has previously entered the same critical section can be used; if processes that have entered the critical section more frequently are given lower priority, then the system will be fair, giving the highest priority to the least frequent user of the resource.

The time since last access to the critical section offers a similar opportunity to enforce fairness if the process that used the critical section least recently is given the highest priority.

If dynamic priority assignments are used, what matters is that the priority used on any entry to the critical section is frozen prior to broadcasting the request for entry, and that it remains the same until after the process is done with that round of mutual exclusion. It is also important that each process has a unique priority, but this can be assured by appending the process ID as the least significant bits of the dynamically chosen priority.

2.9 REMOTE PROCEDURE CALLS

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC is a client/server infrastructure that increases the interoperability, portability and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces− function calls are the programmer's interface when using RPC. RPC makes the client/server model of computing more powerful and easier to program.

2.9.1 How RPC Works?

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. The *Figure 4* shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

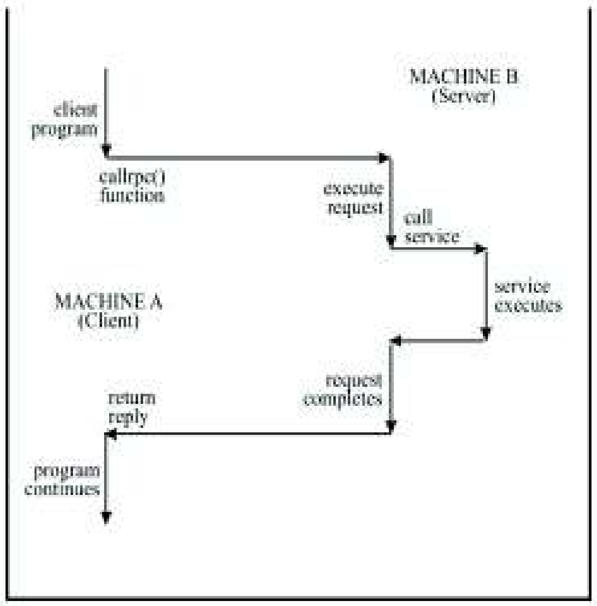


Figure 4: Flow of activity that Tables place during a RPC call between two networked system

2.9.2 Remote Procedure Calling Mechanism

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number). The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of a RPC protocol to be available simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number.

2.9.3 Implementation of RPC

RPC is typically implemented in one of two ways:

1) Within a broader, more encompassing propriety product.

2) By a programmer using a proprietary tool to create client/server RPC stubs.

2.9.4 Considerations for Usage

RPC is appropriate for client/server applications in which the client can issue a request and wait for the server's response before continuing its own processing. Because most RPC implementations do not support peer-to-peer, or asynchronous, client/server interaction, RPC is not well-suited for applications involving distributed objects or object-oriented programming.

Asynchronous and synchronous mechanisms each have strengths and weaknesses that should be considered when designing any specific application. In contrast to asynchronous mechanisms employed by Message-Oriented Middleware, the use of a synchronous request-reply mechanism in RPC requires that the client and server are always available and functioning (i.e., the client or server is not blocked). In order to allow a client/server application to recover from a blocked condition, an implementation of a RPC is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. The complexity of the application using a RPC is dependent on the sophistication of the specific RPC implementation (i.e., the more sophisticated the recovery mechanisms supported by RPC, the less complex the application utilising the RPC is required to be). RPC's that implement asynchronous mechanisms are very few and are difficult (complex) to implement.

When utilising RPC over a distributed network, the performance (or load) of the network should be considered. One of the strengths of RPC is that the synchronous, blocking mechanism of RPC guards against overloading a network, unlike the asynchronous mechanism of Message Oriented Middleware (MOM). However, when recovery mechanisms, such as retransmissions, are employed by an RPC application, the resulting load on a network may increase, making the application inappropriate for a congested network. Also, because RPC uses static routing tables established at compile-time, the ability to perform load balancing across a network is difficult and should be considered when designing a RPC-based application.

Tools are available for a programmer to use in developing RPC applications over a wide variety of platforms, including Windows (3.1, NT, 95), Macintosh, 26 variants of UNIX, OS/2, NetWare, and VMS. RPC infrastructures are implemented within the Distributed Computing Environment (DCE), and within Open Network Computing (ONC), developed by Sunsoft, Inc.

2.9.5 Limitations

RPC implementations are nominally incompatible with other RPC implementations, although some are compatible. Using a single implementation of a RPC in a system will most likely result in a dependence on the RPC vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability.

Because there is no single standard for implementing a RPC, different features may be offered by individual RPC implementations. Features that may affect the design and cost of a RPC-based application include the following:

- support of synchronous and/or asynchronous processing
- support of different networking protocols
- support for different file systems
- whether the RPC mechanism can be obtained individually, or only bundled with a server operating system.

Because of the complexity of the synchronous mechanism of RPC and the proprietary and unique nature of RPC implementations, training is essential even for the experienced programmer.

2.10 OTHER MIDDLEWARE TECHNOLOGIES

Other middleware technologies that allow the distribution of processing across multiple processors and platforms are:

- Object Request Broker(ORB)
- Distributed Computing Environment (DCE)
- Message-Oriented Middleware (MOM)
- COM/DCOM
- Transaction Processing Monitor Technology
- 3-Tier S/W Architecture.

Check Your Progress 1

1) How is a distributed OS different from Network OS? Explain.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What is a Distributed File System?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) Define Load Balancing.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

4) What is the significance of Time Consistency?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

5) What is a Remote Procedure Call and mention its use.

……………………………………………………………………………………

……………………………………………………………………………………

2.11 SUMMARY

A distributed operating system takes the abstraction to a higher level, and allows hides from the application where things are. The application can use things on any of many computers just as if it were one big computer. A distributed operating system will also provide for some sort of security across these multiple computers, as well as control the network communication paths between them. A distributed operating system can be created by merging these functions into the traditional operating system, or as another abstraction layer on top of the traditional operating system and network operating system.

Any operating system, including distributed operating systems, provides a number of services. First, they control what application gets to use the CPU and handle switching control between multiple applications. They also manage use of RAM and disk storage. Controlling who has access to which resources of the computer (or computers) is another issue that the operating system handles. In the case of distributed systems, all of these items need to be coordinated for multiple machines. As systems grow larger handling them can be complicated by the fact that not one person controls all of the machines so the security policies on one machine may not be the same as on another.

Some problems can be broken down into very tiny pieces of work that can be done in parallel. Other problems are such that you need the results of step one to do step two and the results of step two to do step three and so on. These problems cannot be broken down into as small of work units. Those things that can be broken down into very small chunks of work are called fine-grained and those that require larger chunks are called coarse-grain. When distributing the work to be done on many CPUs there is a balancing act to be followed. You don’t want the chunk of work to be done to be so small that it takes too long to send the work to another CPU because then it is quicker to just have a single CPU do the work, You also don’t want the chunk of work to be done to be too big of a chunk because then you can’t spread it out over enough machines to make the thing run quickly.

In this unit we have studied the features of the distributed operating system, architecture, algorithms relating to the distributed processing, shared memory concept and remote procedure calls.

2.12 SOLUTIONS / ANSWERS

Check Your Progress 1

1) A distributed operating system differs from a network of machines each supporting a network operating system in only one way: The machines supporting a distributed operating system are all running under a single operating system that spans the network. Thus, the print spooler might, at some instant, be running on one machine, while the file system is running on others, while other machines are running other parts of the system, and under some distributed operating systems, these parts may at times migrate from machine to machine.

   With network operating systems, each machine runs an entire operating system. In contrast, with distributed operating systems, the entire system is itself distributed across the network. As a result, distributed operating systems typically make little distinction between remote execution of a command and local execution of that same command. In theory, all commands may be executed anywhere; it is up to the system to execute commands where it is convenient.

2) Distributed File System (DFS) allows administrators to group shared folders located on different servers and present them to users as a virtual tree of folders known as a namespace. A namespace provides numerous benefits, including increased availability of data, load sharing, and simplified data migration.

3) Given a group of identical machines, it is wasteful to have some machines overloaded while others are almost idle. If each machine broadcasts its load average every few minutes, one can arrange for new processes to use whichever machine was least loaded at the time. However, one machine must not be given too many processes at once. If the system supports migration, and the load difference between two machines is great enough, a process should be migrated from one to the other.

4) Machines on a local area network have their own clocks. If these are not synchronized, strange things can happen: e.g., the modification date of a file can be in the future. All machines on a LAN should synchronize their clocks periodically, setting their own time to the "network time". However, adjusting time by sudden jumps also causes problems: it may lead to time going backward on some machines. A better way is to adjust the speed of the clock temporarily. This is done by protocols such as NTP.

5) Many distributed systems use Remote Procedure Calls (RPCs) as their main communication mechanism. It is a powerful technique for constructing distributed, client server based applications. It is based on extending the notion of conventional or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and the logical elements of the data communications mechanism and allows the application to use a variety of transports.

2.13 FURTHER READINGS

1) Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, *Applied Operating System Concepts*, 1/e, John Wiley & Sons, Inc, New Delhi.

2) Maarten van Steen and Henk Sips, *Computer and Network Organization: An Introduction*, Prentice Hall, New Delhi.

3) Andrew S. Tanenbaum and Albert S.Woodhull, *Operating Systems Design and Implementation*, 2/e, Prentice Hall, New Delhi.

4) Andrew S. Tanenbaum, *Modern Operating Systems*, 2/e, Prentice Hall, New Delhi.

5) Maarten van Steen and Andrew S. Tanenbaum, *Distributed Systems* 1/e, Prentice Hall, New Delhi.

6) Andrew S. Tanenbaum, *Distributed Operating Systems*, 1/e, Prentice Hall, 1995), New Delhi.

7) Jean Bacon, *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems*, 2/e, Addison Wesley, New Delhi.

8) Jean bacon and Tim Harris, *Operating Systems: Concurrent and distributed software design*, Addison Wesley, 2003, New Delhi.

9) Coulouris, Dolimore and Kindberg, *Distributed Systems: Concepts and Design* 3/e, Addison-Wesley, New Delhi.

10) D.M. Dhamdhere, *Operating Systems – A Concept Based Approach*, Second Edition, TMGH, 2006, New Delhi.

UNIT 3 CASE STUDY - UNIX

Structure **Page Nos.**

3.0 INTRODUCTION

Operating system is a system software, which handles the interface to system hardware (input/output devices, memory, file system, etc), schedules tasks, and provides common core services such as a basic user interface. The purpose of using an Operating System is to provide:

- **Convenience**: It transforms the raw hardware into a machine that is more agreeable to users.
- **Efficiency**: It manages the resources of the overall computer system.

UNIX is a popular operating system, and is used heavily in a large variety of scientific, engineering, and mission critical applications. Interest in UNIX has grown substantially high in recent years because of the proliferation of the Linux (a Unix look-alike) operating system. The following are the uses of the UNIX operating system:

- Universally used for high-end number crunching applications.
- Wide use in CAD/CAM arena.
- Ideally suited for scientific visualization.
- Preferred OS platform for running internet services such as WWW, DNS, DHCP, NetNews, Mail, etc., due to networking being in the kernel for a long time now.

- Easy access to the core of the OS (the kernel) via C, C++, Perl, etc.
- Stability of the operating environment.
- The availability of a network extensible window system.
- The open systems philosophy.

UNIX was founded on what could be called a “small is good” philosophy. The idea is that each program is designed to do one job efficiently. Because UNIX was developed by different people with different needs it has grown to an operating system that is both flexible and easy to adapt for specific needs.

UNIX was written in a machine independent language and C language. So UNIX and UNIX-like operating systems can run on a variety of hardware. These systems are available from many different sources, some of them at no cost. Because of this diversity and the ability to utilize the same "user-interface" on many different systems, UNIX is said to be an open system.

At the time the first UNIX was written, most operating systems developers believed that an operating system must be written in an assembly language so that it could function effectively and gain access to the hardware. The UNIX Operating System is written in C.

The C language itself operates at a level that is just high enough to be portable to a variety of computer hardware. Most publicly distributed UNIX software is written in C and must be complied before use. In practical terms this means that an understanding of C can make the life of a UNIX system administrator significantly easier.

In the earlier units, we have studied various function of OS in general. In this unit we will study a case study on UNIX and how the UNIX handles various operating system functions. In the MCSL-045, section – 1, the practical sessions are given to provide you the hands-on experience.

3.1 OBJECTIVES

After going through this unit you should be able to:

- describe the features and brief history of unix;
- describe how UNIX executes a command;
- define the use of processes;
- describe the process management, memory management handled by UNIX;
- discuss the UNIX file system and the directory structure, and
- explain the CPU scheduling in UNIX.

3.2 FEATURES OF UNIX OS

UNIX is one of the most popular OS today because of its abilities like multi-user, multi-tasking environment, stability and portability. The fundamental features which made UNIX such a phenomenally successful operating systems are:

- **Multi-user:** More than one user can use the machine at a time supported via terminals (serial or network connection).
- **Multi-tasking:** More than one program can be run at a time.
- **Hierarchical file system: T**o support file organization and maintenance in a easier manner.
- **Portability:** Only the kernel (<10%) is written in assembler. This means that the operating system could be easily converted to run on different hardware.
- **Tools for program development:** Supports a wide range of support tools (debuggers, compilers).

3.3 BRIEF HISTORY

In the late 1960s, General Electric, MIT and Bell Labs commenced a joint project to develop an ambitious multi-user, multi-tasking OS for mainframe computers known as MULTICS. MULTICS was unsuccessful, but it did motivate Ken Thompson, who was a researcher at Bell Labs, to write a simple operating system himself. He wrote a simpler version of MULTICS and called his attempt UNICS (Uniplexed Information and Computing System).

To save CPU power and memory, UNICS (finally shortened to UNIX) used short commands to lessen the space needed to store them and the time needed to decode them - hence the tradition of short UNIX commands we use today, e.g., ls, cp, rm, mv etc.

Ken Thompson then teamed up with Dennis Ritchie, the creator of the first C compiler in 1973. They rewrote the UNIX kernel in C and released the Fifth Edition of UNIX to universities in 1974. The Seventh Edition, released in 1978, marked a split in UNIX development into two main branches: SYS V (System 5) and BSD (Berkeley Software Distribution).

Linux is a free open source UNIX OS. Linux is neither pure SYS V nor pure BSD. Instead, it incorporates some features from each (e.g. SYSV-style startup files but BSD-style file system layout) and plans to conform to a set of IEEE standards called POSIX (Portable Operating System Interface). You can refer to the MCSL-045 Section – 1 for more details in a tabular form. This tabular form will give you a clear picture about the developments.

3.4 STRUCTURE OF UNIX OS

UNIX is a layered operating system. The innermost layer is the hardware that provides the services for the OS. The following are the components of the UNIX OS.

The Kernel

The operating system, referred to in UNIX as the **kernel**, interacts directly with the hardware and provides the services to the user programs. These user programs don't need to know anything about the hardware. They just need to know how to interact with the kernel and it's up to the kernel to provide the desired service. One of the big appeals of UNIX to programmers has been that most well written user programs are independent of the underlying hardware, making them readily portable to new systems.

User programs interact with the kernel through a set of standard **system calls**. These system calls request services to be provided by the kernel. Such services would include accessing a file: open close, read, write, link, or execute a file; starting or updating accounting records; changing ownership of a file or directory; changing to a new directory; creating, suspending, or killing a process; enabling access to hardware devices; and setting limits on system resources.

UNIX is a **multi-user**, **multi-tasking** operating system. You can have many users logged into a system simultaneously, each running many programs. It is the kernel's job to keep each process and user separate and to regulate access to system hardware, including CPU, memory, disk and other I/O devices.

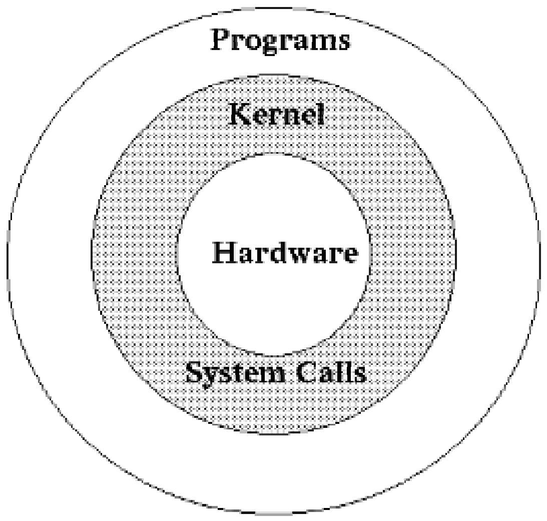


Figure 1: UNIX structure

The Shell

The shell is often called a command line shell, since it presents a single prompt for the user. The user types a command; the shell invokes that command, and then presents the prompt again when the command has finished. This is done on a line-by-line basis, hence the term “command line”.

The shell program provides a method for adapting each user’s setup requirements and storing this information for re-use. The user interacts with /bin/sh, which interprets each command typed. Internal commands are handled within the shell (set, unset), external commands are cited as programs (ls, grep, sort, ps).
There are a number of different command line shells (user interfaces).

- Bourne (sh)
- Korn (krn)
- C shell (csh)

In Windows, the command interpreter is based on a graphical user interface.
In UNIX, there is a line-orientated command interpreter. More details related to the shells are provided in MCSL-045, Section 1.

System Utilities

The system utilities are intended to be controlling tools that do a single task exceptionally well (e.g., *grep* finds text inside files while *wc* counts the number of words, lines and bytes inside a file). Users can solve problems by integrating these tools instead of writing a large monolithic application program.
Like other UNIX flavours, Linux's system utilities also embrace server programs called **daemons** that offer remote network and administration services (e.g. telnetd provides remote login facilities, httpd serves web pages).

Application Programs

Some application programs include the emacs editor, gcc (a C compiler), g++ (a C++ compiler), xfig (a drawing package), latex (a powerful typesetting language).

UNIX works very differently. Rather than having kernel tasks examine the requests of a process, the process itself enters *kernel space*. This means that rather than the process waiting “outside” the kernel, it enters the kernel itself (i.e. the process will start executing kernel code for itself).

This may sound like a formula for failure, but the ability of a process to enter kernel space is strictly prohibited (requiring hardware support). For example, on *x86*, a process enters kernel space by means of system calls - well known points that a process must invoke in order to enter the kernel.

When a process invokes a system call, the hardware is switched to the kernel settings. At this point, the process will be executing code from the kernel image. It has full powers to wreak disaster at this point, unlike when it was in user space. Furthermore, the process is no longer *pre-emptible*.

3.5 PROCESS MANAGEMENT

When the computer is switched on, the first thing it does is to activate resident on the system board in a ROM (read-only memory) chip. The operating system is not available at this stage so that the computer must "pull itself up by its own boot-straps". This procedure is thus often referred to as *bootstrapping*, also known as cold boot. Then the system initialization takes place. The system initialization usually involves the following steps. The kernel,

- tests to check the amount of memory available.
- probes and configures hardware devices. Some devices are usually compiled into the kernel and the kernel has the ability to autoprobe the hardware and load the appropriate drivers or create the appropriate entries in the /*dev* directory.
- sets up a number of lists or internal tables in RAM. These are used to keep track of running processes, memory allocation, open files, etc.

Depending on the UNIX version, the kernel now creates the first UNIX processes. A number of *dummy processes* (processes which cannot be killed) are created first to handle crucial system functions. A *ps -ef* listing on each OS shows will show you the existing processes. ***init*** is the last process created at boot time. It always has a process ID (PID) of 1. *init* is responsible for starting all subsequent processes. Consequently, it is the parent process of all (non-dummy) UNIX processes.

Don't confuse the *init* process with the system *init* command. The *init* command (usually found in /sbin or /usr/sbin) is used by root to put the system into a specific run level.

All subsequent processes are created by *init*. For example, one of the processes started by *init* is *inetd*, the internet super daemon. (*inetd*, in turn, creates many other processes, such as *telnetd*, on demand.) In the Unix process hierarchy, *init* is called the **parent** process and *inetd* the child of *init*. Any process can have any number of children (up to the kernel parameter *nproc*, the maximum allowed number of processes). If you kill the parent of a child process, it automatically becomes the child of *init*.

Each running process has associated with it a process ID or PID. In addition, each process is characterized by its parent's PID or PPID. Finally, each process runs at a default system priority (PRI). The smaller the numerical value of the PRI, the higher the priority and *vice versa*.

3.5.1 Management of the Processes by the Kernel

For each new process created, the kernel sets up an *address space* in the memory. This address space consists of the following logical segments:

- *text* - contains the program's instructions.
- *data* - contains initialized program variables.
- *bss* - contains uninitialized program variables.
- *stack* - a dynamically growable segment, it contains variables allocated locally and parameters passed to functions in the program.

Each process has two stacks: a *user stack* and a *kernel* stack. These stacks are used when the process executes in the user or kernel mode (described below).

Mode Switching

At least two different modes of operation are used by the UNIX kernel - a more privileged kernel mode, and a less privileged user mode. This is done to protect some parts of the address space from user mode access.

User Mode: Processes, created directly by the users, whose instructions are currently executing in the CPU are considered to be operating in the user-mode. Processes running in the user mode do not have access to code and data for other users or to other areas of address space protected by the kernel from user mode access.

Kernel Mode: Processes carrying out kernel instructions are said to be running in the kernel-mode. A user process can be in the kernel-mode while making a system call, while generating an exception/fault, or in case on an interrupt. Essentially, a mode switch occurs and control is transferred to the kernel when a user program makes a system call. The kernel then executes the instructions on the user's behalf.

While in the kernel-mode, a process has full privileges and may access the code and data of any process (in other words, the kernel can see the entire address space of any process).

3.5.2 The Context of a Process and Context Switching

The context of a process is essentially a snapshot of its current runtime environment, including its address space, stack space, etc. At any given time, a process can be in user-mode, kernel-mode, sleeping, waiting on I/O, and so on. The process scheduling subsystem within the kernel uses a time slice of typically 20ms to rotate among currently running processes. Each process is given its share of the CPU for 20ms, then left to sleep until its turn again at the CPU. This process of moving processes in and out of the CPU is called context switching. The kernel makes the operating system appear to be *multi-tasking* (i.e. running processes concurrently) via the use of efficient context-switching.

At each context switch, the context of the process to be swapped out of the CPU is saved to RAM. It is restored when the process is scheduled its share of the CPU again. All this happens very fast, in microseconds.

To be more precise, context switching may occur for a user process when

- a system call is made, thus causing a switch to the kernel-mode,
- a hardware interrupt, bus error, segmentation fault, floating point exception, etc. occurs,
- a process voluntarily goes to sleep waiting for a resource or for some other reason, and
- the kernel preempts the currently running process (i.e. a normal process scheduler event).

Context switching for a user process may occur also between *threads* of the same process. Extensive context switching is an indication of a CPU bottleneck.

3.5.3 Communication between the Running Processes

UNIX provides a way for a user to communicate with a running process. This is accomplished via *signals,* a facility which enables a running process to be notified about the occurrence of a) an error event generated by the executing process, or b) an asynchronous event generated by a process outside the executing process.

Signals are sent to the process ultimately by the kernel. The receiving process has to be programmed such that it can catch a signal and take a certain action depending on which signal was sent.

Here is a list of common signals and their numerical values:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGKILL	9	Kill (cannot be caught or ignore)
SIGTERM	15	Terminate (termination signal from SIGKILL).

3.6 MEMORY MANAGEMENT

One of the numerous tasks the UNIX kernel performs while the machine is up is to manage memory. In this section, we explore relevant terms (such as physical vs. virtual memory) as well as some of the basic concepts behind memory management.

3.6.1 Physical vs. Virtual Memory

UNIX, like other advanced operating systems, allows you to use all of the physical memory installed in your system as well as area(s) of the disk (called swap space) which have been designated for use by the kernel in case the physical memory is insufficient for the tasks at hand. Virtual memory is simply the sum of the physical memory (RAM) and the total swap space assigned by the system administrator at the system installation time.

Virtual Memory (VM) = Physical RAM + Swap space

3.6.2 Dividing Memory into Pages

The UNIX kernel divides the memory into manageable chunks called **pages**. A single page of memory is usually 4096 or 8192 bytes (4 or 8KB). Memory pages are laid down contiguously across the physical and the virtual memory.

3.6.3 Cache Memory

With increasing clock speeds for modern CPUs, the disparity between the CPU speed and the access speed for RAM has grown substantially. Consider the following:

- Typical CPU speed today: 250-500MHz (which translates into 4-2ns clock tick)
- Typical memory access speed (for regular DRAM): 60ns
- Typical disk access speed: 13ms

In other words, to get a piece of information from RAM, the CPU has to wait for 15-30 clock cycles, a considerable waste of time.

Fortunately, cache RAM has come to the rescue. The RAM cache is simply a small amount of very fast (and thus expensive) memory which is placed between the CPU and the (slower) RAM. When the kernel loads a page from RAM for use by the CPU, it also prefetches a number of adjacent pages and stores them in the cache. Since programs typically use sequential memory access, the next page needed by the CPU can now be supplied very rapidly from the cache. Updates of the cache are performed using an efficient algorithm which can enable cache hit rates of nearly 100% (with a 100% hit ratio being the ideal case).

CPUs today typically have hierarchical caches. The on-chip cache (usually called the L1 cache) is small but fast (being on-chip). The secondary cache (usually called the L2 cache) is often not on-chip (thus a bit slower) and can be quite large; sometimes as big as 16MB for high-end CPUs (obviously, you have to pay a hefty premium for a cache that size).

3.6.4 Memory Organisation by the Kernel

When the kernel is first loaded into memory at the boot time, it sets aside a certain amount of RAM for itself as well as for all system and user processes. Main categories in which RAM is divided are:

- *Text*: to hold the text segments of running processes.
- *Data*: to hold the data segments of running processes.
- *Stack*: to hold the stack segments of running processes.
- *Shared Memory*: This is an area of memory which is available to running programs if they need it. Consider a common use of shared memory: Let us assume you have a program which has been compiled using a shared library (libraries that look like *libxxx.so*; the C-library is a good example - all programs need it). Assume that five of these programs are running simultaneously. At run-time, the code they seek is made resident in the shared memory area. This way, a single copy of the library needs to be in memory, resulting in increased efficiency and major cost savings.
- *Buffer Cache*: All reads and writes to the file system are cached here first. You may have experienced situations where a program that is writing to a file doesn't seem to work (nothing is written to the file). You wait a while, then a *sync* occurs, and the buffer cache is dumped to disk and you see the file size increase.

3.6.5 The System and User Areas

When the kernel loads, it uses RAM to keep itself memory resident. Consequently, it has to ensure that user programs do not overwrite/corrupt the kernel data structures (or overwrite/corrupt other users' data structures). It does so by designating part of RAM as kernel or system pages (which hold kernel text and data segments) and user pages (which hold user stacks, data, and text segments). Strong memory protection is implemented in the kernel memory management code to keep the users from corrupting the system area. For example, only the kernel is allowed to switch from the user to the system area. During the normal execution of a Unix process, both system and user areas are used. A common system call when memory protection is violated is SIGSEGV.

3.6.6 Paging vs. Swapping

Paging: When a process starts in UNIX, not all its memory pages are read in from the disk at once. Instead, the kernel loads into RAM only a few pages at a time. After the CPU digests these, the next page is requested. If it is not found in RAM, a **page fault** occurs, signaling the kernel to load the next few pages from disk into RAM. This is called **demand paging** and is a perfectly normal system activity in UNIX. (Just so you know, it is possible for you, as a programmer, to read in entire processes if there is enough memory available to do so.)

The UNIX daemon which performs the paging out operation is called *pageout*. It is a long running daemon and is created at boot time. The *pageout* process cannot be killed.

Swapping: Let's say you start ten heavyweight processes (for example, five xterms, a couple netscapes, a sendmail, and a couple pines) on an old 486 box running Linux with 16MB of RAM. Basically, you do not have enough physical RAM to accommodate the text, data, and stack segments of all these processes at once. Since the kernel cannot find enough RAM to fit things in, it makes use of the available virtual memory by a process known as **swapping**. It selects the least busy process and moves it in its entirety (meaning the program's in-RAM text, stack, and data segments) to disk. As more RAM becomes available, it swaps the process back in from disk into RAM. While this use of the virtual memory system makes it possible for you to continue to use the machine, it comes at a very heavy price. Remember, disks are relatively slower (by the factor of a million) than CPUs and you can feel this disparity rather severely when the machine is swapping. Swapping is not considered a normal system activity. It is basically a sign that you need to buy more RAM.

The process handling swapping is called *sched* (in other UNIX variants, it is sometimes called *swapper*). It always runs as process 0. When the free memory falls so far below *minfree* that *pageout* is not able to recover memory by page stealing, *sched* invokes the syscall *sched()*. Syscall *swapout* is then called to free all the memory pages associated with the process chosen for being swapping out. On a later invocation of *sched()*, the process may be swapped back in from disk if there is enough memory.

3.6.7 Demand Paging

Berkeley introduced demand paging to UNIX with BSD (Berkeley System) which transferred memory pages instead of processes to and from a secondary device; recent releases of UNIX system also support demand paging. Demand paging is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and then the process is loaded into the frame by the kernel.

The advantage of demand paging policy is that it permits greater flexibility in mapping the virtual address of a process into the physical memory of a machine, usually allowing the size of a process to be greater than the amount of availability of physical memory and allowing more Processes to fit into main memory. The advantage of a swapping policy is that is easier to implement and results in less system overhead.

3.7 FILE SYSTEM IN UNIX

Physical disks are partitioned into different *file systems*. Each file system has a maximum size, and a maximum number of files and directories that it can contain. The file systems can be seen with the df command. Different systems will have their file systems laid out differently. The / directory is called the *root* of the file system.

The UNIX file system stores all the information that relates to the long-term state of the system. This state includes the operating system kernel, the executable files, configuration information, temporary work files, user data, and various special files that are used to give controlled access to system hardware and operating system functions.
The constituents of UNIX file system can be one of the following types:

3.7.1 Ordinary files

Ordinary files can contain text, data, or program information. Files cannot contain other files or directories. UNIX filenames are not broken into a name part and an extension part, instead they can contain any keyboard character except for '/' and be up to 256 characters long. However, characters such as *,?,# and & have special meaning in most shells and should not therefore be used in filenames. Putting spaces in filenames also makes them difficult to manipulate, so it is always preferred to use the underscore '_'.

3.7.2 Directories

Directories are folders that can contain other files and directories. A **directory** is a collection of files and/or other directories. Because a directory can contain other directories, we get a directory **hierarchy**. The “top level” of the hierarchy is the **root directory.**

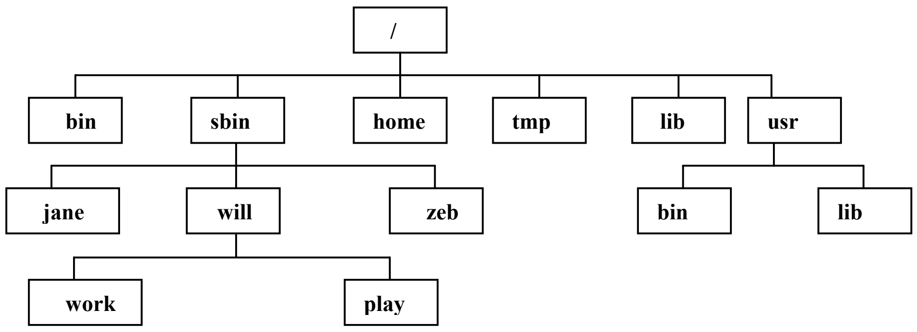


Figure 2: UNIX File System

The UNIX file system is a hierarchical tree structure with a top-level directory known as the root (designated by a slash '/'). Because of the tree structure, a directory can have many child directories, but only one parent directory. The layout is shown in Figure 2. The path to a location can be defined by an absolute path from the root /, or as a relative path from the current working directory. To specify an **absolute path,** each directory from the source to the destination must be included in the path, with each directory in the sequence being separated by a slash. To specify **a relative path,** UNIX provides the shorthand "." for the current directory and ".." for the parent directory e.g., The absolute path to the directory "play" is /sbin/will/play, while the relative path to this directory from "zeb" is ../will/play. Various UNIX directories and their contents are listed in the table given below.

Table 1: Typical UNIX directories

Directory	**Content**
/	The "root" directory
/bin	Essential low-level system utilities
/usr/bin	Higher-level system utilities and application programs
/sbin	Superuser system utilities (for performing system administration tasks)
/lib	Program libraries (collections of system calls that can be included in programs by a compiler) for low-level system utilities
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home or /homes	User home directories containing personal file space for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files
/dev	Hardware devices
/proc	A pseudo-filesystem that is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

When you log into UNIX, your current working directory is your user home directory.

You can refer to your home directory at any time as "~" and the home directory of other users as "~<login>".

3.7.3 Links

A link is a pointer to another file. There are two types of links - a **hard link** to a file cannot be distinguished from the file itself. A **soft link** or also called the **symbolic link** provides an indirect pointer or shortcut to a file.

Each file is assigned an ***inode number*** by the kernel. Attributes in a file table in the kernel include its name, permissions, ownership, time of last modification, time of last access, and whether it is a file, directory or some other type of entity.

A -i flag to *ls* shows the inode number of each entry.

A -i flag to *df* shows the number of inodes instead of the amount of space.

The *ls -l* command lists the number of links to the inode in the second column. If you remove one of the names associated with a particular inode, the other names are preserved.

The file is not removed from the filesystem until the "link count" goes to zero. All permissions and ownerships of a link are identical to the file that you have linked to. You may not link to a directory.

You can form a *symbolic link* to a file by giving *a -s* flag to *ln*. The symbolic link has its own inode number. To remove a symbolic link to a directory, use *rm*, not *rmdir*.

3.7.4 File and Directory Permissions

Each file or a directory on a UNIX system has three types of permissions, describing what operations can be performed on it by various categories of users. These permissions are read (r), write (w) and execute (x), and the three categories of users are user or owner (u), group (g) and others (o). These are summarized in the *Figure 3* given below:

Permission	**File**	**Directory**
Read	User can look at the contents of the file	User can list the files in the directory
Write	User can modify the contents of the file	User can create new files and remove existing files in the directory
Execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

Figure 3:.Interpretation of permissions for files and directories

These file and directory permissions can only be modified by:

- their owners
- by the superuser (root)

by using the chmod(change [file or directory])mode system utility.

$ chmod *options files*

chmod accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits. Each octal digit represents the access permissions for the user/owner, group and others respectively. The mapping of permissions onto their corresponding octal digits is as follows:

---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

For example the command:

```
$ chmod 600 ignou.txt
```

sets the permissions on *ignou.txt* to rw------- (i.e., only the owner can read and write to the file). chmod also supports a -R option which can be used to recursively modify file permission by using *chgrp* (change group) .

$ chgrp *group files*

This command can be used to change the group that a file or directory belongs to.

3.7.5 File Compression and Backup

UNIX systems usually support a number of utilities for backing up and compressing files. The most useful are:

tar (tape archiver)

tar backs up entire directories and files onto a tape device or into a single disk file known as an archive.

To create a disk file tar archive, use
$ tar -cvf *archivename filenames*

where *archivename* will usually have *.tar* extension. Here the *c* option means create, *v* means verbose (output filenames as they are archived), and *f* means file.

- To list the contents of a tar archive, use:
  $ tar -tvf *archivename*
- To restore files from a tar archive, use:

  $ tar -xvf *archivename*

compress, gzip

compress and gzip are utilities for compressing and decompressing individual files (which may be or may not be archive files).

- To compress files, use:
  $ compress *filename* **or** $ gzip *filename*

- To reverse the compression process, use:
  $ compress -d filename **or** $ gzip -d filename

3.7.6 Handling Removable Media

UNIX supports tools for accessing removable media such as CDROMs and floppy disks with the help of mount and umount commands.

The *mount* command serves to attach the file system found on some device to the file system tree. Conversely, the *umount* command will detach it again (it is very important to remember to do this when removing the floppy or CDROM). The file /etc/fstab contains a list of devices and the points at which they will be attached to the main filesystem:

```
$ cat /etc/fstab ↵
/dev/fd0   /mnt/floppy  auto    rw,user,noauto  0 0
/dev/hdc   /mnt/cdrom   iso9660 ro,user,noauto  0 0
```

In this case, the mount point for the floppy drive is /mnt/floppy and the mount point for the CDROM is /mnt/cdrom. To access a floppy we can use:

```
$ mount /mnt/floppy ↵
$ cd /mnt/floppy ↵
$ ls
```

To force all changed data to be written back to the floppy and to detach the floppy disk from the file system, we use:

```
$ umount /mnt/floppy
```

3.7.7 Pipes and Filters

Unix systems allow the output of one command to be used as input for another command. Complex tasks can be performed by a series of commands one piping input into the next.

Basically a pipe is written generator | consumer.

Sometimes, the use of intermediately files is undesirable. Consider an example where you are required to generate a report, but this involves a number of steps. First you have to concatenate all the log files. The next step after that is strip out comments, and then to sort the information. Once it is sorted, the results must be printed.

A typical approach is to do each operation as a separate step, writing the results to a file that then becomes an input to the next step.

Pipelining allows you to connect two programs together so that the output of one program becomes the input to the next program.

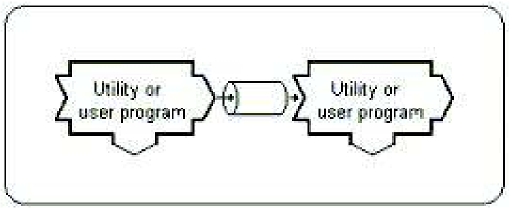


Figure 4: Pipes in UNIX

The symbol | (vertical bar) represents a pipe. Any number of commands can be connected in sequence, forming a pipeline .All programs in a pipeline execute at the same time.

The pipe (|) operator is used to create concurrently executing processes that pass data directly to one another. For example:

```
$ cat hello.txt | sort | uniq ↵
```

creates three processes (corresponding to cat, sort and uniq) that execute concurrently. As they execute, the output of the first process is passed on to the sort process which is in turn passed on to the uniq process. uniq displays its output on the screen (a sorted list of users with duplicate lines removed).

3.7.8 Redirecting Input and Output

The output from programs is usually written to the screen, while their input usually comes from the keyboard (if no file arguments are given). In technical terms, we say that processes usually write to **standard output** (the screen) and take their input from **standard input** (the keyboard). There is in fact another output channel called **standard error**, where processes write their error messages; by default error messages are also sent to the screen.

To redirect standard output to a file instead of the screen, we use the > operator:

```
$ echo hello ↵
hello
$ echo hello > output ↵
$ cat output ↵
hello
```

In this case, the contents of the file output will be destroyed if the file already exists. If instead we want to append the output of the echo command to the file, we can use the >> operator:

```
$ echo bye >> output ⮠
$ cat output ⮠
hello
bye
```

To capture standard error, prefix the > operator with a 2 (in UNIX the file numbers 0, 1 and 2 are assigned to standard input, standard output and standard error respectively), e.g.:

```
$ cat nonexistent 2>errors ⮠
$ cat errors ⮠
cat: nonexistent: No such file or directory
$
```

Standard input can also be redirected using the < operator, so that input is read from a file instead of the keyboard:

```
$ cat < output ⮠
hello
bye
```

You can combine input redirection with output redirection, but be careful not to use the same filename in both places. For example:

```
$ cat < output > output ⮠
```

will destroy the contents of the file output. This is because the first thing the shell does when it sees the > operator is to create an empty file ready for the output.

3.8 CPU SCHEDULING

CPU scheduling in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

The scheduler on UNIX system belongs to the general class of operating system schedulers known as round robin with multilevel feedback which means that the kernel allocates the CPU time to a process for small time slice, pre-empts a process that exceeds its time slice and feed it back into one of several priority queues. A process may need much iteration through the "feedback loop" before it finishes. When kernel does a context switch and restores the context of a process, the process resumes execution from the point where it had been suspended.

Each process table entry contains a priority field. There is a process table for each process which contains a priority field for process scheduling. The priority of a process is lower if they have recently used the CPU and vice versa.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa, so there is negative feedback in CPU scheduling and it is difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a one second quantum for the round- robin scheduling. 4.33SD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the time-out mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval; the subroutine to be called in this case causes the rescheduling and then resubmits a time-out to call itself again. The priority recomputation is also timed by a subroutine that resubmits a time-out for itself event. The kernel primitive used for this purpose is called sleep (not to be confused with the user-level library routine of the same name.) It takes an argument, which is by convention the address of a kernel data structure related to an event that the process wants to occur before that process is awakened. When the event occurs, the system process that knows about it calls wakeup with the address corresponding to the event, and all processes that had done a sleep on the same address are put in the ready queue to be run.

☞ **Check Your Progress 1**

1) In which programming language UNIX is written and mention the important features of UNIX OS.

………………………………………………………………………………………

………………………………………………………………………………………

2) What is a file structure in UNIX? What is its importance in UNIX?

………………………………………………………………………………………

………………………………………………………………………………………

3.9 SUMMARY

In this unit we discussed issues broadly related to features of UNIX OS, boot process, system initialization, process management, memory management, file system and CPU scheduling in UNIX operating system. In this unit we discussed several theoretical concepts of UNIX operating system in general, it is often useful to use them in your lab for practice. Refer to the Section – 1 of MCSL-045, in which we have covered the practical component of the UNIX operating system.

3.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1) UNIX is written in high-level language, C. This makes the system highly portable. It supports good programming environment. It allows complex programming to be built from smaller programs. It uses a hierarchical file system that allows easy maintenance and efficient implementation.

2) UNIX system uses a hierarchical file system structure. Any file may be located by tracing a path from the root directory. Each supported device is associated with one or more special files. Input/Output to special files is done in the same manner as with ordinary disk files, but these requests cause activation of the associated devices.

3.11 FURTHER READINGS

1) Brain W. Kernighan, Rob Pike, *The UNIX Programming Environment*, PHI, 1996

2) Sumitabha Das, *Your UNIX – The Ultimate Guide*, TMGH, 2002

3) Sumitabha Das, *UNIX Concepts and Applications*, Second Edition, TMGH, 2003

4) K.Srirengan, *Understanding UNIX*, PHI, 2002.

5) Behrouz A. Foroujan, Richard F.Gilberg, *UNIX and Shell Programming*, Thomson Press, 2003.

UNIT 4 CASE STUDY: WINDOWS 2000

Structure **Page Nos.**

4.0 INTRODUCTION

In the earlier unit we have covered the case study of one of the popular operating systems namely UNIX. In this unit we will make another case study, WINDOWS 2000.

Windows 2000 is an operating system that runs on high-end desktop PCs and servers. In this case study, we will examine various aspects of Windows 2000 operating system, starting with introduction, then exploring its architecture, processes, memory management, I/O, the file system, and security aspects.

4.1 OBJECTIVES

After studying this unit you should be able to:

- gain experience relating to various aspects on Windows 2000 operating system;
- understand the architecture of Windows 2000 OS, and
- understand processes, memory management, I/Os, the file system, and above all security management handled by the Windows 2000 OS.

4.2 WINDOWS 2000 : AN INTRODUCTION

The release of NT following NT 4.0 was originally going to be called NT 5.0. But, in the year 1999, Microsoft decided to change the name to Windows 2000. This is a single main operating system built on reliable 32-bit technology but using the Windows 98 user interface. Windows 2000 is based on the Windows NT Kernel and is sometimes referred to as Windows NT 5.0. Windows 2000 is a complex operating system consisting of over 29 million lines of C /C++ code among which eight million lines of code is written for drivers. Windows 2000 is currently by far one of the largest commercial projects ever built.

4.2.1 Significant features of Windows 2000

Some of the significant features of Windows 2000 are:

- Support for FAT16, FAT32 and NTFS
- Increased uptime of the system and significantly fewer OS reboot scenarios
- Windows Installer tracks applications and recognizes and replaces missing components
- Protects memory of individual apps and processes to avoid a single app bringing the system down
- Encrypted File Systems protect sensitive data
- Secure Virtual Private Networking (VPN) supports tunneling in to private LAN over public Internet
- Personalized menus adapt to the way you work
- Multilingual version allows for User Interface and help to switch, based on logon
- Includes broader support for high-speed networking devices, including Native ATM and cable modems
- Supports Universal Serial Bus (USB) and IEEE 1394 for greater bandwidth devices.

Windows 2000 is really a NT 5.0, as it inherits many properties from NT 4.0. It is a true 32bit / 64bit multiprogramming system with protected processes. Each process consists of 32-bit (or 64 bit) demand paged virtual address space. The user process runs in user mode and operating system runs in kernel mode resulting in complete protection, thereby eliminating the Windows 98 flaws. Processes can have greater than one thread, which are scheduled by the operating system. It provides security for all files, directories, processes, and other shareable objects. And above all it supports for running on symmetric multiprocessors with upto 32 CPUs.

Windows 2000 is better than NT 4.0 with the Windows 98 user interface. It contains a number of features that are found only in Windows 98 like complete support for plug-and-play devices, the USB bus, FireWire (IEEE 1394), IrDA (Infrared link for portable computers/printers), power management etc. The new features in Windows 2000 operating system are:

- Active directory service,
- Security using Kerberos,
- Support for smart cards,
- System monitoring tools,
- Better integration of laptop computers with desktop computers,
- System management infrastructure,
- Single instance store, and job objects.

- The file system NTFS has been extended to support encrypted files, quotas, linked files, mounted volumes, and content indexing etc.
- Internationalization.

Windows 2000 consists of a single binary that runs everywhere in the world and the language can be selected at the run time. Windows 2000 uses Unicode, but a Windows 2000 does not have MS-DOS (use command line interface).

Like earlier versions of NT, Windows 2000 comes in several product levels: Professional, Server, Advanced Server, and Datacenter server. The differences between these versions are very small and same executable binary used for all versions. The various versions of Windows 2000 are show in Table 1:

Table 1: Versions of Windows 2000

Windows 2000 Version	Max RAM	CPUs	Max clients	Cluster size	Optimized for
Professional	4GB	2	10	0	Response Time
Server	4GB	4	Unlimited	0	Throughput
Advanced Server	8GB	8	Unlimited	2	Throughput
Datacenter Server	64GB	32	Unlimited	4	Throughput

When the system is installed, version is recorded in the registry (internal database). At boot time, the operating system checks the registry to see the version. The details of various versions are given in Table 1 above. The cluster size relates to capability of Windows 2000 to make two or four servers look like a single server to the outside world.

4.3 WINDOWS 2000 PROGRAMMING

In this section we will see the details of programming interface and the registry, a small in-memory database.

4.3.1 Application Programming Interface

Windows 2000 has a set of system calls, which was not made public by Microsoft. Instead Microsoft has defined a set of function calls called the Win32 API (Application Programming Interface,) shown in Figure 1, which are publicly known and well documented. These are a set of library procedures that make system calls to perform certain jobs or do the work in the user space. These Win32 API calls do not change with new releases of Windows, although new API calls are added.

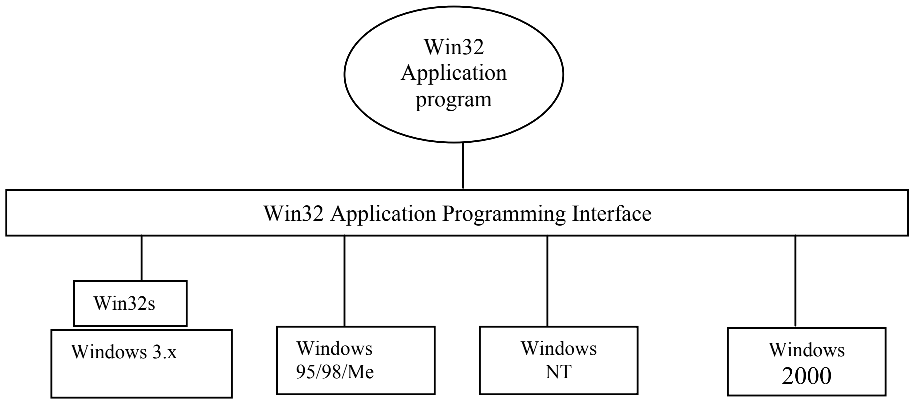

Win32 Application program
Win32 Application Programming Interface
Win32s
Windows 3.x
Windows 95/98/Me
Windows NT
Windows 2000

Figure 1: The Win32 Application Programming Interface

- Binary programs for the Intel x86 that conforms to Win32 API interface will run unmodified on all versions of Windows operating system. An extra library is required for Windows 3.x to match a subset of the 32-bit API calls to the 16-bit operating system. Windows 2000 has additional API calls, which will not function on older versions of Windows operating system.

- The Win32 API concept is totally different from the UNIX philosophy. In the case of UNIX, the system calls are all publicly known and form a minimal operating system and removing any of these would affect the functionally of the operating system. Whereas, Win32 provides a very comprehensive interface and the same thing can be performed in a number of ways.
- Most of Win32 API calls creates kernel objects including files, processes, threads, pipes, etc and returns a result called a handle to the caller. Not all system-created data structures are objects and not all objects are kernel objects. True kernel objects are those that need to be named, protected, or shared in some way, has system defined type, well-defined operations on it and occupies storage in kernel memory. This handle is used to perform operations on the object and does the handle refer to specific to the process that created the object. Handles cannot be passed directly to another process (in the same manner as UNIX file descriptors cannot be passed to another process and used there). But is possible to duplicate a handle and then it can be passed to another processes in a protected manner, allowing them controlled access to the objects of there process. Every object has a security descriptor, containing information regarding who may and may not perform what kinds of operations on the object. Windows 2000 can also create and use objects.
- Windows 2000 is partially object oriented because the only way to manipulate objects is by invoking operations on their handles by invoking Win32 calls. But there is not concept of inheritance and polymorphism.
- Memory management system is invisible to programmer (demand paging), but a significant feature is visible, that is the ability of a process to map a file onto a region of its virtual memory.
- File I/O: A file is a linear sequence of bytes, in Win32 environment. Win32 environment provides over 60 calls for creating and destroying files/directories, opening, closing, reading, writing, requesting/setting attributes of files, etc.
- Every process has a process ID telling who it is and every object has an access control list describing who can access it and which operation are allowed, thus providing a fine-grained security.
- Windows 2000 file names are case sensitive and use the Unicode character set.
- On Windows 2000, all the screen coordinates given in the graphic function are true 32-bit numbers.

4.3.2 The Registry

All the information needed for booting and configuring the system and tailoring it to the current user was gathered in a big central database called the registry. It consists of a collection of directories, each of which contains either subdirectories or entries (the files). A directory is called a key and all top level key (directories) starts with the string HKEY (handle to key). At the bottom of the hierarchy are the entries, called values. Each value consists of three parts: a name, a type, and the data. At the top level, the Windows 2000 registry has six keys, called root keys, as shown below. One can view it using one of the registry editors (regedit or regedt32).

- HKEY_LOCAL_MACHINE
- HKEY-USERS
- HKEY_PERFORMANCE_DATA
- HKEY_CLASS_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER

4.4 WINDOWS 2000 OS STRUCTURE

In this section we will see, how the system is organized internally, what are the various components and how these components interact with each others and with user programs.

Windows 2000 consists of two major components:

- Operating system itself, which runs in kernel mode

The kernel handles the process management, memory management, file systems, and so on. The lowest two software layers, the HAL (hardware abstraction layer) and the kernel, are developed in C and in assembly language and are partly machine dependent. The upper layers are written in C and are almost machine independent. The drivers are written in C, or in a few cases C++.

There are two main components: (a) Kernel Mode: the operating system itself, which runs in kernel mode, and (b) User Mode: the environment subsystems. The kernel handles process management, memory management, file systems, and so on. The environment subsystem is separate processes which help user program carry out certain specific system functions. Windows 2000 follows modular structure. The simplified of Windows 2000 is given below in *Figure 2*.

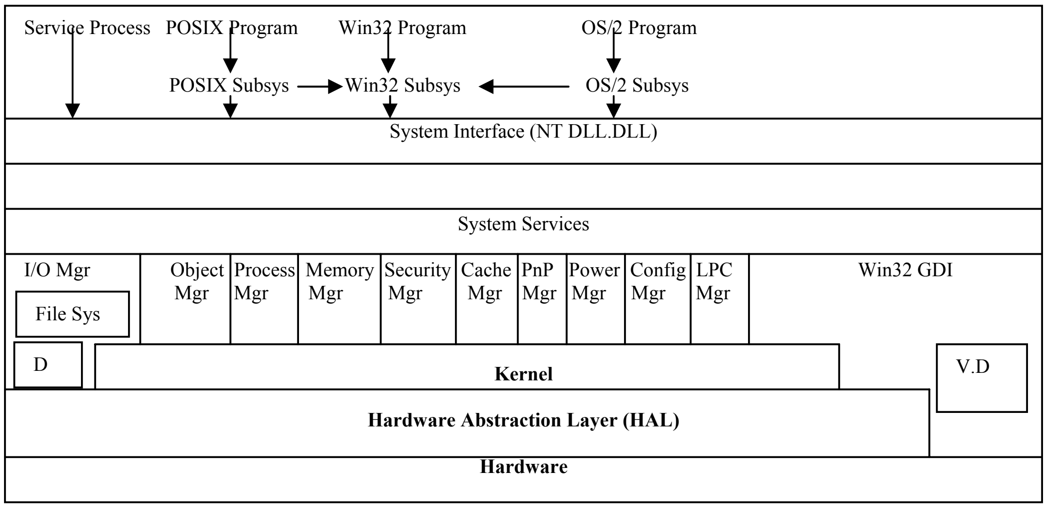


Figure 2: The structure of Windows 2000. D – Device Driver, V.D – Video Driver

12.4.1 Hardware Abstraction Layer

To make the operating system portable across platforms, Microsoft attempts to hide many of the machine dependencies in a HAL (Hardware Abstraction Layer). The job of the HAL is to present the rest of the operating system with abstract hardware devices. These devices are available in the form of machine-independent services (the procedure calls and macros). By not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to a new hardware. Furthermore, HAL services are identical on all Windows 2000 systems irrespective of the underlying hardware and porting HAL is simple because all the machine dependent code is located in one place and the goals of the port are well defined. The HAL address those services which relate to the chipset on the parent board and which vary from system to system. This hides the differences between one vendor's parent board and another one's, but not the differences between a Pentium and an Alpha. The various HAL services are given below, but HAL does not provide abstraction or services for specific I/O devices such as mouse, keyboard, or disk or for memory management unit:

- Access to device registers
- Bus-independent device addressing
- Interrupt handling
- Resetting
- DMA transfer
- Control of the times and real time clock
- Low-level spin locks and multiprocessor synchronization

- Interfacing with the BIOS and its CMOS configuration memory

4.4.2 Kernel Layer

The purpose of the kernel is to make the rest of the operating system hardware-independent. It accesses the hardware via the HAL and builds upon the extremely low level HAL services to build higher-level abstractions. In addition to providing a higher-level abstraction of the hardware and handling thread switches, the kernel also has another key function: providing low level support for control objects and dispatcher objects.

Executive

The executive is the upper portion of the operating system. It is written in C and is architecture independent. The executive consists of ten components, each of which is just a collection of procedures that work together to accomplish a certain task. These components are: object manager, I/O manager, Process Manager, Memory Manager, Security Manager, Cache Manager, Plug-and-Play manager, Power Manager, Configuration Manager, Local Procedure call manager.

On booting, Windows 2000 is loaded into memory as a collection of files and the description of important files is given below in *Table 2*:

Table 2: Important files in Windows 2000 Operating System

File Name	Consists of
ntoskrnl.exe	Kernel and executive
hal.dll	HALL
Win32k.sys	Win32 and GDI
*.sys	Driver files

Device Drivers

Device drivers are not part of *ntoskrnl.exe* binary and when a drive is installed on the system, it is added to a list of the registry and is loaded dynamically on booting. A device driver can control one or more I/O devices and can perform encryption of a data stream or just providing access to kernel data structures. The largest device drivers are Win32 GDI and video, handles system calls and most of the graphics.

Objects

Objects are most important concept in Windows 2000. Objects provide a uniform and consistent interface to all system resources and data structures such as threads, processes, semaphores, etc. An object is a data structure in RAM. A file on disk is an object, but an object is created for file when it is opened. When a system boots, there are no objects present at all (except for the idle and system processes, whose objects are hardwired into the ntoskrnl.exe file). All objects are created on the fly as the system boots up and various initialization programs run. Some of the common executive object types are shown in *Table 3* below:

Table 3: Some of the Common Executive Object Types Managed by Object Manager

Object Type	Description
Process	User Process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
Port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval

Queue	Object used for completion notification on asynchronous I/O
Open File	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Structure used for mapping files into virtual address space
Key	Registry key
Object directory	Directory for grouping objects within the object manager
Symbolic link	Pointer to another object by name
Device	I/O device object
Device driver	Each loaded device has its own object.

As objects are created and deleted during execution, the object manager maintains a name space, in which all objects in the system are located. A process uses this name space to locate and open a handle for some other process' object, provided it has been granted permission to do so. Windows 2000 maintains three name spaces: (1) object name space, (2) file system name space, and (3) the registry name space. All three-name space follows a hierarchical model with multiple levels of directories for organizing entries. The object name space is not visible to users without special viewing tools. One freely downloadable viewing tool 'winobj' is available at www.sysinternals.com.

4.4.3 Environment Subsystem

User mode components are of three kinds:

(1) DLLs,
(2) Environment subsystems, and
(3) Service processes.

These components provide each user process with an interface that is different from the Windows 2000 system call interface. Windows 2000 supports the following APIs: Win32, POSIX, and OS2. The job of the DLLs and environment subsystems is to implement the functionality of the published interface, thus hiding the true system call interface from application programs. The Win32 interface is the official interface for Windows 2000, by using DLLs and Win32 environment subsystem, a program can be coded in Win32 specifications and run unmodified on all versions of Windows, even though the system calls are not the same on different systems. Windows 2000 uses DLLs extensively for all aspects of the system. A user process generally links with a number of DLLs that collectively implements the Win32 interface.

4.5 PROCESS AND THREADS

Each process in Windows 2000 operating system contains its own independent virtual address space with both code and data, protected from other processes. Each process, in turn, contains one or more independently executing *threads*. A thread running within a process can create new threads, create new independent processes, and manage communication and synchronization between the objects.

By creating and managing processes, applications can have multiple, concurrent tasks processing files, performing computations, or communicating with other networked systems. It is even possible to exploit multiple processors to speed processing.

The following sections explain the basics of process management and also introduce the basic synchronization operations.

4.5.1 Windows Processes and Threads

Every process consists of one or more threads, and the Windows thread is the basic executable unit. Threads are scheduled on the basis of the following factors: (a) availability of resources such as CPUs and physical memory, (b) priority, (c) fairness, and so on. Windows has supported symmetric multiprocessing (*SMP*) since NT4, so threads can be allocated to separate processors within a system. Therefore, each Windows process includes resources such as the following components:

- One or more threads.
- A virtual address space that is distinct from other processes' address spaces, except where memory is explicitly shared. Note that shared memory-mapped files share physical memory, but the sharing processes will use different virtual addresses to access the mapped file.
- One or more code segments, including code in DLLs.
- One or more data segments containing global variables.
- Environment strings with environment variable information, such as the current search path.
- The process heap.
- Resources such as open handles and other heaps.

Each thread in a process shares code, global variables, environment strings, and resources. Each thread is independently scheduled, and a thread has the following elements:

- A stack for procedure calls, interrupts, exception handlers, and automatic storage.
- Thread Local Storage (*TLS*)—arrays of pointers giving each thread the ability to allocate storage to create its own unique data environment.
- An argument on the stack, from the creating thread, which is usually unique for each thread.
- A Context Structure maintained by the kernel, with machine registers values.

Windows 2000 has a number of concepts for managing the CPU and grouping resources together. In the following section we will study these concepts.

Windows 2000 supports traditional process that communicates and synchronizes with each other. Each process contains at least one thread, which contains at least one fiber (a lightweight thread). Processes combine to form a job for certain resource management purposes. And jobs, processes, threads, and fibers provides a very general set of tools for managing parallelism and resources.

- **Job:** collection of processes that share quotas and limits
- **Process:** container for holding resources
- **Thread:** Entity scheduled by the kernel
- **Fiber:** Lightweight thread managed entirely in user space.

A job in Windows 2000 is a collection of one or more processes. There are quotas (maximum number of processes, total CPU time, memory usage) and resource limits associated with each job, store in the corresponding job object.

Every process has a 4-GB address space, with the user occupying the bottom 2 GB (3 GB on advanced server and Datacenter Server) and operating system occupying the rest. Process is created using Win32 call and consist of a process ID, one or more threads, a list of handles, and an access token containing security related information. Each process starts out with one thread and new threads can be created dynamically using a win32 call. Operating system selects a thread to run for CPU scheduling. Every thread has a state like ready, running, blocked, etc. Every thread has a thread ID, which is taken from the same space as the process IDs, so an ID can never be in use for both a process and a thread at the same time. Processes and thread can be used as byte indices into kernel tables, as they are in multiple of four. A thread is normally run in user mode, but when it invokes a system call it switches to kernel mode. There are two stacks for each thread, one for use when it is in user mode and one for use when it is in kernel mode. A thread consists of the following

- Thread ID
- Two stacks
- A context (to save its register when not running)
- A private are for its own local variables and
- Possibly its own access token. If a thread has its own access token, this variable overrides the process access token in order to let client threads pass their access rights to server threads who are doing work for them.

Threads are a scheduling concept and not a resource ownership concept. In addition to normal threads that run in user processes, Windows 2000 provides a number of daemon threads (associated with the special system or idle processes) that runs in kernel space and are not linked or associated with any user process. Switching threads in Windows 2000 are relatively expensive, as they require entering and exiting kernel mode. Further, Windows 2000 provides fibers, which are similar to threads, but are scheduled in user space by the originating program (or its run time system). A thread can have multiple fibers. There are no executive objects relating to fibers, as there are for jobs, processes, and threads. In fact, the operating system knows nothing about the fibers and there are no true system calls for managing fibers. Windows 2000 is capable of running on a symmetric multiprocessor system. The relationship between jobs, processes, and threads is illustrated in *Figure 3*.

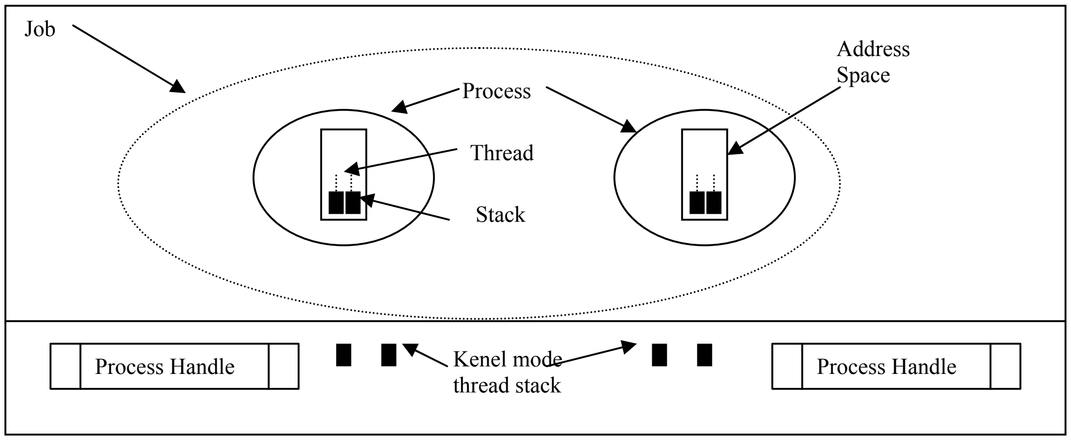


Figure 3: Relationship between jobs, processes, and threads

API Calls

Win32 API function Create Process is used to create a new process. This function consists of 10 parameters. The design is complicated that UNIX as in UNIX 'fork' has no parameters, and 'exec' has just three (pointers to the name of the file to execute, command line parameter array, and the environment strings). The ten parameters of Create Process are:

1) Pointer to the name of the executable file.
2) Command line itself
3) Pointer to a security descriptor for the process
4) Pointer to a security descriptor for the initial thread
5) A bit telling whether the new process inherits the creator's handles.
6) Miscellaneous flag (error mode, priority, debugging, consoles etc.)
7) A pointer to the environment string.
8) Pointer to the name of the new process current working directory.
9) Pointer for initial window on the screen.
10) Pointer to a structure that returns 18 values.

There is no concept of inheritance in Windows 2000 and it does not enforce any kind of parent-child or other hierarchy. All processes are created equal. But there is an implicit hierarchy in terms of who has a handle to whom. One of the eighteen parameters returns to the creating process is a handle to the new process, thus allowing control over the new process.

Initially each process is created with a one thread and process can create further threads, which is simpler than the process creation. The kernel performs the thread creation function and is not implemented purely in user space as is the case in some other operating systems. CreateThread has only six parameters:

1) Security descriptor (optional)
2) initial stack size
3) starting address
4) user defined parameter
5) initial state of the thread (ready/blocked)
6) thread ID.

4.5.2 Interprocess Communication

Threads can communicate in many ways including: pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. The various communications modes are shown in *Table 4* below:

Table 4: Thread Communication Mode

Thread communication	**Mode**
Pipes	byte and message, selected at creation time.
Name Pipes	Byte and message
Mailslots	A feature of Windows 2000 not present in UNIX. They are one-way, whereas pipes are two-way. They can also be used over a network but do not provide guaranteed delivery. They allow the sending process to broadcast a message to many receivers, instead of to just one receiver.
Sockets	Are like pipes, except that they normally connect processes on different machines. For example, one process writes to a socket and another one on a remote system reads from it.
Remote Procedures Calls	Are a mechanism for a process A to have process B call a procedure in B's address space on A's behalf and return the result to A.
Shared Files	Processes can share memory by mapping onto the same file at the same time

Besides Windows 2000 providing numerous interprocess communication mechanism, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical regions, and events. All of these mechanisms functions on threads, not process.

Semaphore

A semaphore is created using function 'CreateSemaphore' API function. As Semaphores are kernel objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated and as a result multiple process can be synchronised on the same semaphore. Calls for up (ReleaseSemaphore) and down (WaitForSingleObject) are present. A calling thread can be released eventually using WaitForSingleObject, even if the semaphore remains at 0.

Mutexes

Mutexes are kernel objects used for synchronization and are simpler than semaphores as they do not have counters. They are locks, with API functions for locking (WaitForSingleObject) and unlocking (releaseMutex). Like Semaphores, mutex handles can be duplicated.

Critical Sections or Critical Regions

This is the third synchronization mechanism which is similar to mutexes. It is pertinent to note that Critical Section are not kernel objects, they do not have handles or security descriptors and cannot be passed between the processes. Locking and unlocking is done using EnterCriticalSection and LeaveCriticalSection, respectively. As these API functions are performed initially in user space and only make kernel calls when blocking is needed, they are faster than mutexes.

Events

This synchronization mechanism uses kernel objects called events, which are of two types: manual-reset events and auto-reset events.

The number of Win32 API calls dealing with processes, threads, and fibres is nearly 100. Windows 2000 knows nothing about fibres and fibres are implemented in user space. As a result, the CreateFibre call does its work entirely in user space without making 12-13 system calls.

Scheduling

There is no central scheduling thread in Windows 2000 operating system. But when a thread cannot run any more, the thread moves to kernel mode and runs the scheduler itself to see which thread to switch to and the concurrency control is reached through the following conditions

- Thread blocks on a semaphore, mutex, event, I/O, etc.: In this situation the thread is already running in kernel mode to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it must save its own context, run the scheduler to pick its successor, and load that threat's context to start it.

- It signals an object – In this situation, thread is running in kernel mode and after signaling some object it can continue as signaling an object never blocks. Thread runs the scheduler to verify if the result of its action has created a higher priority thread. If so, a thread switch occurs because Windows 2000 is fully pre-emptive.

- The running thread's quantum expires: In this case, a trap to kernel mode occurs, thread executes the scheduler to see who runs next. The same thread may be selected again and gets a new quantum and continue running, else a thread switch takes place.

The scheduler is also called under two other conditions:

1) An I/O operation completes: In this case, a thread may have been waiting on this I/O and is now released to run. A check is to be made to verify if it should pre-empt the running thread since there is no guaranteed minimum run time. The win32 API provides two hooks (SetPriorityClass, SetThreadPriority) for process to influence thread scheduling.

2) A timed wait expires.

For SetPriorityClass, the values are: realtime, high, above normal, normal, below normal, and idle. For SetThreadPriority, the values are: time critical, highest, above normal, normal, below normal, lowest, and idle. With six process classes and seven thread classes, a thread gets any one of 42 combinations, which is an input to the scheduling algorithm.

The scheduler has 32 priorities, numbered from 0 to 31. The 42 combinations are mapped onto the 32 priority classes.

4.6 BOOTING WINDOWS 2000

To start Windows 2000 system, it must be booted first. The boot process generates the initial processes that start the system. The process is as follows:

- Boot process consists of reading in the first sector of the first disk , the master boot record, and jumping to it.
- The assembly language program reads the partition table to check which partition table contains the bootable operating system.
- On finding the O.S partition, it reads the first sector of the partition, called the boot sector.
- The program in the boot sector reads its partition's root directory, searching for a file called ntldr.
- *Ntldr* is loaded into memory and executed. Ntldr loads Windows 2000.
- *Ntldr* reads *Boot.ini*, the only configuration information that is not available in the registry. It lists of various versions of hal.dll and ntoskrnl.exe available for booting in this partition, parameters no of CPUs, RAM size, space for user processes (2 GB or 3 GB), and rate of real time clock.
- *Ntldr* then selects and loads hal.dll and ntoskrnl.exe and bootvid.dll (default video driver).
- *Ntldr* next reads the registry to find out which drivers are needed to complete the boot. It reads all drivers and passes the control to ntoskrnl.exe.
- The Operating system does some general initialization and then calls the executive components to do their own initialization. The object manager prepares its name space to allow other components call it to insert objects into the name space. Memory manager set up initial page tables and the plug-and-play manager finding out which I/O devices are present and loading their drivers. The last step is creating the first true user process, the session manager, smss.exe (*The session manager is a native Windows 2000 process, it makes true system calls and does not use the Win32 environment subsystem. It starts csrss.exe and reads registry hives from disk. Its job includes entering many objects in the object manager's name space, creating required paging files, and opening important DLLs. After this, it creates winlogon.exe).* Once this process is up and running, booting of the system is complete.
- After the service processes (user space daemons) start and allow user to log in. *Winlogon.exe* first creates the authentication manager (*lsass.exe*), and then the parent process of the services (*services.exe*). *Winlogon.exe* is responsible for user logins. A separate program in *msgina.dll* handles the actual login dialog.

Table 5: The processes starting up during system booting.

Process	Details
Idle System	Home to the idle thread
Smss.exe	Creates smss.exe & paging files; reads registry, open DLLs
Csrss.exe	First real process, creates csrss & winlogon
Winlogon.exe	Win32 process
Lsass.exe	Login daemon
Services.exe	Authentication manager
	Looks in registry and starts services
Printer server	Remote job printing
File server	Local files requests
Telnet server	Remote logins
Incoming mail handler	Accepts and stores inbound email
Incoming fax handler	Accepts and prints inbound faxes
DNS Resolver	Internet domain name system server
Event logger	Logs various system events
Plug-and-play manager	Monitors hardware

4.7 MEMORY MANAGEMENT

Windows 2000 consists of an advanced virtual memory management system. It provides a number of Win32 functions for using it and part of the executives and six dedicated kernel threads for managing it. In Windows 2000, each user process has its own virtual address space, which is 32 bit long (or 4 GB of virtual address space). The lower 2 GB minus approx 256 MB are reserved for process's code and data; the upper 2 GB map onto to kernel memory in a protected way. The virtual address space is demand paged with fixed pages size. Virtual address space for a user is shown in *Figure 4.4* below:

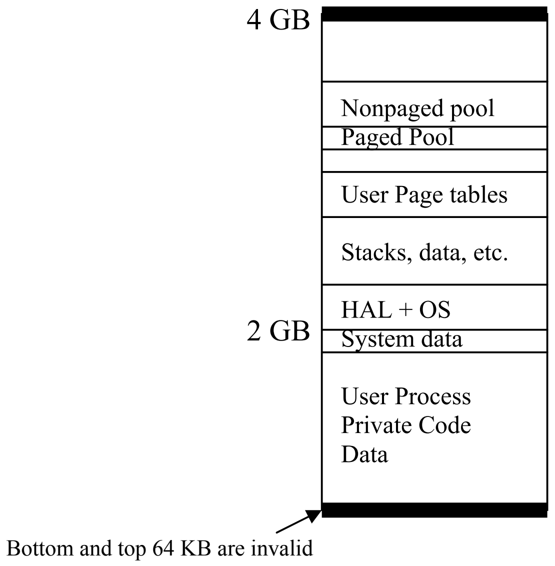


Figure 4.4: Virtual Address space layout for user processes

The bottom and top 64 Kb process virtual space is normally unmapped. This is done intentionally to catch programming errors as invalid pointers are often 0 or –1. From 64 KB onward comes the user's private code and data, which extends upto 2 GB. Some system counters and timers are available at high end of 2 GB. This makes them visible and allows processes to access them without the overhead of a system call. The next 2 GB of virtual address space contains the operating system, code, data, paged and non-paged pools. The upper 2 GB is shared by all user processes, except page tables, which are exclusively reserved for each process' own page table. The upper 2 GB of memory is not writable and mostly not even readable by user-mode processes. When a thread makes a system call, it traps into kernel mode and keeps on running in the same thread. This makes the whole OS and all of its data structure including user process visible within thread's address space by switching to thread's kernel stack. This results in less private address space per process in return for faster system calls. Large database server feel short of address space, and that is why 3-GB user space option is available on Advanced Server and Datacenter server.

There are three states available for each virtual space: free, reserved, or committed. A free page is not currently in use and reference to it results in page fault. When a process is initiated, all pages are in free state till the program and initial data are loaded onto the address space. A page is said to be committed when code or data is mapped onto a page. A committed page is referenced using virtual memory hardware and succeeds if the page is available in the main memory. A virtual page in reserved state cannot be mapped until the reservation is explicitly removed. In addition to the free, reserved, and committed states, pages also have other states like readable, writable, and executable.

In order to avoid wasting disk space, Windows 2000 committed pages that are not permanent on disk (e.g. stack pages) are not assigned a disk page till they have to be paged out. This strategy makes system more complex, but on the other hand no disk space need be allocated for pages that are never paged out.

Free and reserved pages do not consist of shadow pages on disk and a reference to these pages cause page fault. The shadow pages on the disk are arranged into one or more paging files. It is possible to have 16 paging files, spread over 16 separate disks, for higher I/O bandwidth, each created with an initial and maximum size. These files can be created at the maximum size at the time of system installation to avoid fragmentation.

Like many versions of UNIX, Windows 2000 allows files to be mapped directly onto the virtual address spaces. A mapped file can be read or written using ordinary memory references. Memory-mapped files are implemented in the same way as other committed pages, only the shadow pages are in the user's file instead of in the paging file. Due to recent writes to the virtual memory, the version in memory may not be identical to the disk version. On unmapping or explicitly flushing, the disk version is brought up to date. Windows 2000 permits two or more processes to map onto to the same portion of a file at the same time, at different virtual addresses as depicted in *Figure 5*. The processes can communicate with each and pass data back and forth at very high bandwidth, since no copying is required. The processes may have different access permissions. The change made by one process is immediately visible to all the others, even if the disk file has not yet been updated.

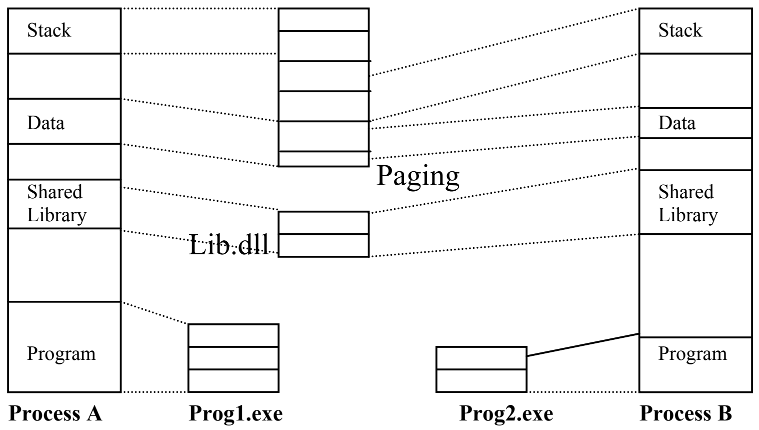


Figure 5: Two or more processes to map onto the same part of the same file at the same time, possibly at different virtual addresses in Windows 2000.

There is another technique called copy-on-write, where data can be safely written without affecting other users or the original copy on disk. This is to handle a situation where two programs share a DLL file and one of them changes the file's static data.

To allow the programs to use more physical memory than fit in the address space, there is technique to place a piece of code at any virtual address without relocation is called position independent code. There is a technique called bank switching in which a program, in which program could substitute some block of memory above the 16-bit or 20-bit limit for a block of its own memory. When 32- bit machine came, it was thought that there would be enough address space forever. Now large programs often need more than the 2 GB or 3 GB of user address space Windows 2000 allocates to them, so bank switching is back, now called address windowing extensions. This technique is only used on servers with more than 2 GB of physical memory.

System Calls

Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. Some of important functions are listed below in *Figure 6*.

Table 6: Important Win32 API functions for managing virtual memory in Windows 2000.

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it.
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space

UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

These functions operate on a region having one or a sequence of pages that are consecutive in the virtual address space. To minimize porting problems to upcoming architectures with pages larger than existing ones, allocated regions always begin on 64-KB boundaries. Actual address space is in multiple of 64 KB. Windows 2000 has a API function to allow a process to access the virtual memory of a different process over which it has been given control or handle.

Windows 2000 supports a single linear 4-GB demand-paged address space per process and segmentation is not supported in any form. Pages size is shown in *Table 7* below:

Table 7: Windows 2000 Page Size

Processor	**Page Size**
Theoretically	Any power of 2 upto 64 KB
Pentium	4 KB
Itanium	8KB or 16 KB

The operating system can use 4-MB pages to reduce page table space consumed. Whereas the scheduler selects individual threads to run and does not care much about the processes, the memory manager deals with processes and does not care much about threads. As and when virtual address space is allocated for a process, the memory manager creates a VAD (Virtual Address Descriptor). VAD consists of a range of addresses mapped, the backing store file and offset where it is mapped, and the protection code. On touching the first page, the directory of page tables is created and pointer to it added in the VAD. The address space is completely defined by the list of its VAD's.

There is no concept of pre-paging in Windows 2000. All pages are brought dynamically to the memory as page fault occurs. On each page fault:

- A trap to the kernel occurs
- Kernel builds a machine independent descriptor and passes this to memory manager
- Memory manager checks its validity
- If the faulted page falls within a committed or reserved region, then it looks for address in the list of VAD, finds/create the page table, and looks up the relevant entry.

The page-table structure is different for different architectures. Entry for a Pentium system for a mapped page is given below in *Figure 6*:

Bits	**20**	**3**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Page frame	Not used	G	L	D	A	C	Wt	U	W	V	

G: Page is global to all processes
L: Large (4-MB) page
D: Page is dirty
A: Page has been accessed
C: Caching enabled/disabled

Wt: Write through (no caching)
U: Page is accessible in user mode
W: Writing to the page permitted
V: Valid page table entry

Figure 6: A page table entry for a mapped

A page fault occurs in five of the below listed categories:

1) The page referenced is not committed.
2) A protection violation occurred
3) A shared page has been written.
4) The stack needs to grow.
5) The page referenced is committed but not currently mapped in.

It is pertinent to note that Windows 2000 does not read in isolated pages from the disk, but reads in runs of consecutive pages to minimize disk transfers. The run size is larger for code pages than for data pages.

In Windows 2000 entire paging system makes heavy use of the working set concept. Every process has a working set and this set consists of the mapped-in pages that are in memory. The size and composition of the working set fluctuates as the process' thread run. Working set is described by two parameters: minimum size and the maximum size and there are hard bounds. The default minimum is in the range 20-50 and the default initial maximum is in the range 45-345 depending on the total amount of RAM.

Windows 2000 uses a local algorithm, to prevent one process from obstructing other by hogging memory. The page is added if the working set is smaller than the minimum on occurrence of a page fault. If the working set is larger than the maximum, a page is evicted from the working set to make room for the new page. System also tries to tune to the situation and algorithm may be a mix of local and global. The last 512 pages are left for some slack for new processes.

In general, Windows 2000 resolves various conflicts through complex heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter setting. Memory management is a highly complex subsystem with many data structures, algorithms, and heuristics.

4.8 INPUT/OUTPUT IN WINDOWS 2000

The goal of the Windows 2000 I/O system is to provide a framework for efficiently handle various I/O devices. These includes keyboards, mice, touch pads, joysticks, scanners, still cameras, television cameras, bar code readers, microphones, monitors, printers, plotters, beamers, CD-records, sound cards, clocks, networks, telephones, and camcorders.

The I/O manager works in conjunction with the plug-and-play manager and closely connected with the plug and play manager. The power manager can put the computer into six different states: (1) fully operational, (2) Sleep-1: CPU power reduced, RAM and cache on; instant wake up, (3) Sleep-2: CPU and RAM on; CPU cache off; continue from current PC; (4) Sleep-3: CPU and cache off; RAM on; restart from fixed address; (5) Hibernate: CPU, cache, and RAM off; restart from saved disk file; (6) off: Everything off; full reboot required. Similarly I/O devices can be in various power states using power manager. It is pertinent to note that all the file systems are technically I/O drivers and request for any data block is initially sent to the cache manager and on failure it is directed to the I/O manager, which calls the proper file system driver to get the required block from the disk.

Windows 2000 supports dynamic disk, which spans multiple partitions and even multiple disks and may be reconfigured on fly without rebooting the system. Windows 2000 also provide support for asynchronous I/O and it is possible for a thread to start an I/O operation and then continue executing in parallel with the I/O. This feature is very important on servers.

4.8.1 Implementation of I/O

I/O manager creates a framework where different I/O devices can operate. This framework consists of a set of device-independent procedures for certain aspects of I/O plus a set of loaded device drivers for communicating with the devices.

4.8.2 Device Drivers

In Windows 2000 device driver must conform to a Windows Driver Model (defined for Microsoft). It also provides a toolkit for writing conformant drivers. The drivers must meet the following requirements:

- handle incoming I/O requests, which arrive in a standard format

- Be as object based as the rest of Windows 2000.
- Allow plug-and-play devices to be dynamically added / removed.
- Permit power management, where applicable.
- Be configurable in terms of resource usage.
- Be reentrant for use on multiprocessors.
- Be portable across Windows 98 and Windows 2000.

In UNIX, drivers are located by using their major device numbers. In Windows 2000, at boot time, or when a new hot pluggable plug-and-play device is attached to the computer, the system detects it and calls the plug-and-play manager. The plug-and-play manager detects the manufacturer and model number from the device. After that it looks for the corresponding driver in the hard disk or prompts the user to insert a floppy disk or CD-ROM with driver. Once detected the driver is loaded into the memory.

A driver may or may not be stacked as shown in *Figure 7* below. In stacked mode the request passes through a sequence of drivers. The use of driver stacking is: (a) separate out the bus management from the functional work of actually controlling device; (b) to be able to insert filter drivers into the stack.

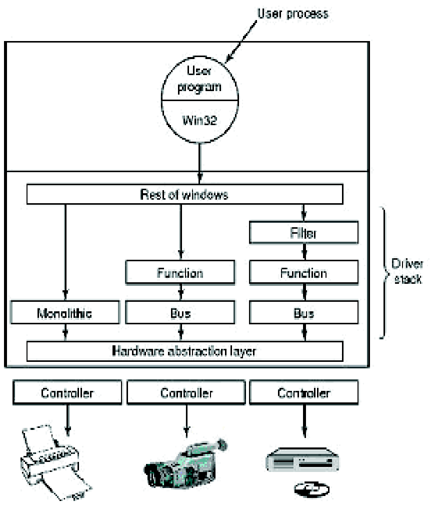


Figure 4.7: Windows 2000 allows drivers to be stacked

4.9 FILE SYSTEM MANAGEMENT

Windows 2000 supports several file systems like FAT-16, FAT-32, and NTFS. Windows 2000 also supports read-only file systems for CD-ROMs and DVDs. It is possible to have access to multiple file system types on the same running system.

4.9.1 New Technology File System (NTFS)

NTFS stands for **N**ew **T**echnology **F**ile **S**ystem. Microsoft created NTFS to compensate for the features it felt FAT was lacking. These features include increased fault tolerance and enhanced security. Individual File names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. Files are in Unicode. But Win32 API does not fully support case-sensitivity for file names. NTFS consists of multiple attributes, each of which is represented by a stream of bytes.

NTFS is a highly complex and sophisticated file system. Each NTFS volume contains files, directories, bitmaps, and other data structures. These volumes are organized as a linear sequence of blocks (512 bytes to 64 KB size), which is called clusters in Microsoft's terminology. The main data structure in each volume is master file table (MFT), which is a linear sequence of fixed size 1-KB records. Each MFT describes one file or directory.

4.9.2 Fault Tolerance

NTFS repairs hard disk errors automatically without displaying an error message. When Windows 2000 writes a file to an NTFS partition, it keeps a copy of the file in memory. It then checks the file to make sure it matches the copy stored in memory. If the copies don't match, Windows marks that section of the hard disk as bad and won't use it again (Cluster Remapping). Windows then uses the copy of the file stored in memory to rewrite the file to an alternate location on the hard disk. If the error occurred during a **read**, NTFS returns a read error to the calling program, and the data is lost.

4.9.3 Security

NTFS has many security options. You can grant various permissions to directories and to individual files. These permissions protect files and directories locally and remotely. NT operating system was designed to meet the U.S. Department of Defense's C2 (the orange book security requirements). Windows 2000 was not designed for C2 compliance, but it inherits many security properties from NT, including the following: (1) secure login with anti-spoofing mechanism, (2) Discretionary access controls, (3) Privileged access controls, (4) Address space protection per process, (5) New pages must be zeroed before being mapped in, (6) security auditing.

Every user and group in Windows 2000 is identified by a SID (security ID), which includes a binary numbers with a short header followed by a long random component. The purpose is to get a unique SID worldwide. The process and its threads run under the users' SID. And only threads with authorized SIDS can access objects. Each process has an access token which possess its SID and other information.

NTFS also includes the Encrypting File System (EFS). EFS uses public key security to encrypt files on an NTFS volume, preventing unauthorized users from accessing those files. This feature comes in quite handy on a portable compute, for example. Lose a portable, and the files on its disk are fair game to anyone who knows how to get to them. EFS use 128-bit (40-bit internationally) Data Encryption Standard (DES) encryption to encrypt individual files and folders. Encryption keys are implemented on a Windows 2000 domain or in the case of a standalone computer or locally. The operating system generates a recovery key so administrators can recover encrypted data in the event that users lose their encryption key.

4.9.4 File Compression

Another advantage to NTFS is native support for file compression. The NTFS compression offers you the chance to compress individual files and folders of your choice. An example of file compression in Windows 2000 is shown in Figure 8:

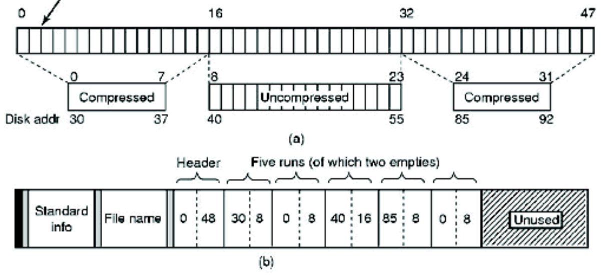


Figure 8: (a) 48-block file being compressed into 32 blocks, (b) MFT record for the file after compression

4.9.5 Disk Quotas

Disk quotas allow administrators to manage the amount of disk space allotted to individual users, charging users only for the files they own. Windows 2000 enforces quotas on a per-user and per-volume basis.

4.9.6 Caching in Windows 2000

The Windows 2000 cache manager does caching and conceptually it is similar to other systems. However, its design has some unusual properties.

Windows 2000 has single integrated cache that works for all file systems in use. Separate file systems do not need to maintain their respective cache. The operation of cache manager is shown in Figure below: operates at a higher level than the file systems.

Windows 2000 cache is organized by virtual block, not physical blocks. The traditional file caches keep track of blocks by two-part addresses of the form (partition (denotes device and partition), block (block no.)). In Windows 2000, cache manager uses (file, offset) to refers to a block.

☞ **Check Your Progress 1:**

1. Using Windows 2000 server, perform the following activities:

   a) Create a logon warning message
   b) Create an user account
   c) Create two new local groups and assign permissions
   d) Obtain a list of users without authentication

   ……………………………………………………………………………………………

   ……………………………………………………………………………………………

4.10 SUMMARY

Windows 2000 structure consists of the HAL, the kernel, the executive, and a thin system services layer that catches incoming system calls. There are also a variety of device drivers, file systems and GDI. The HAL layer provides hardware independence and hides differences in hardware from the upper layer. And the kernel hides the remaining differences from the executive to make it almost machine independent.

The user processes create Windows 2000 executive who is based on memory objects. User processes get back handles to manipulate them at later. Objects can be created by the executive components and object manager possess a name space where objects can be inserted for subsequent references.

Windows 2000 operating system supports processes, jobs, threads, and fibers. Processes consist of virtual address spaces, which are container for resources. Threads are the unit of execution, which are scheduled by the operating system. Lightweight threads or fibers are scheduled entirely in user space. Resources quotas are assigned by jobs, which is a collection of processes. Priority algorithm is used for scheduling.

Windows 2000 has a demand-paged virtual memory and the paging algorithm is based on the working set concept. There several lists of free pages. On occurrence of page fault, a free page is made available. Pages that have not been used for a long time are fed by trimming the working set using complex formulas.

Device drivers do I/O, which is based on the windows device model. Initializing a driver object those posses the addresses of the procedures that the system can call to add devices or perform I/O. Further activates a device driver; drivers can be stacked to act as filters.

The Windows 2000 NTFS file system is based on a master file table with one record per file or directory. Multiple attributes of a file can be in the MFT record or nonresident, on disk. NTFS also supports compression and encryption.

Security in Windows 2000 Operating system is based on access control lists. Each process has an access control token; each object has a security descriptor.

Windows 2000 maintains a single system wide cache for all the file systems. This is a virtual cache not the physical one.

4.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1) (i) **Creating a Logon Warning Message**

- In the run box, type Regedit to open the Registry editor.
- In the Registry Editor navigate to: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\WinIogo.
- Double-click the LegalNoticeCaption value.
- In the Value Data entry field, enter Unauthorized Access Warning!! Then press OK.
- Double-click the LegalNoticeText value.
- Type Unauthorized access to this system is punishable by a fine of Rs fifty thousands and/or one years in prison. Access to this system indicates that you have read and agree to this warning in the Value Data entry field, then press OK.
- Close the Registry Editor.
- Log off and back on again (no need to restart) to verify the changes. Verify that you warning message is present.

(ii) **Creating an User Account**

- Log on to your NT Server as an Administrator.
- Navigate to: User Manager for Domains.
- Select User > New User.
- Type TestUser in the Username Field.
- Type your first name in the Password Field. Keep in mind that passwords are case sensitive.
- Type the exact same password in the Confirm Password Field.
- Add this user to the Administrators Group.

(iii) **Assigning Permissions**

- Logon to your Windows NT server as Administrator.
- Navigate to User Manager for Domains.
- Create two New local groups, GROUP1 and GROUP2
- Create a user, Hari, and make this user belong to both the GROUP1 and GROUP2 groups.
- Close User Manager for Domains.
- Open My Computer.
- Navigate to an NTFS partition (Right-click to check properties if unsure)
- Create a new Folder called Check
- Right-click on Check, and select properties.
- Select the Sharing tab.
- Remove the Everyone group and give GROUP1 Full Control
- Select the Security tab
- Remove the Everyone group, and give GROUP2 Read
- Close the properties of the Check folder.

(iv) **Obtaining a List of users without authentication**

- Logon to your Windows 2000 machine as Administrator.

- Navigate to Active Users and Computers. Create a Global Group (Group types will be discussed shortly) named Global1.
- Create Domain five domain users.
- Add the new users to the Global1 group.
- On the second computer, logon to Windows NT as Administrator
- Navigate to User Manager for Domains
- Select Policies, and then Trust Relationships.
- Press the Add button, next to Trusted Domains.
- Enter the NetBIOS domain name of the other computer, you may leave the password blank, and press OK.
  - (In the event the name is not found, add an entry to the Lmhosts file of the NT computer with the hostname and IP address of the Windows 2000 computer)
  - Do not verify the trust.
- Stay in User Manager for Domains, and create a Local group called Local1.
- Double-click the new group to add members.
- Select the domain of the other computer as the source in the List Names From scroll box.
- Select the Global1 Group.
- Press the Members button to reveal the usernames of the remote domain.

4.12 FURTHER READINGS

1) Operating Systems, Milan Milenkovic, TMGH, 2004.
2) Operating Systems, William Stallings, Pearson Education, 2003.
3) www.microsoft.com/Windows**2000**/
4) www.**windows**itpro.com/**windows**nt**2000**2003faq/