Master of Computer Application

(Open and Distance Learning Mode)

Semester – I

Operating Systems

Centre for Distance and Online Education (CDOE)

DEVI AHILYA VISHWAVIDYALAYA, INDORE

"A+" Grade Accredited by NAAC

IET Campus, Khandwa Road, Indore - 452001

www.cdoedavv.ac.in

www.dde.dauniv.ac.in

Program Coordinator

Dr. Anand More
School of Computer Science and IT
Devi Ahilya Vishwavidyalaya, Indore – 452001

Content Design Committee

Dr. Pratosh Bansal
Centre for Distance and Online Education
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. C.P. Patidar
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. Shaligram Prajapat
International Institute of Professional Studies
Devi Ahilya Vishwavidyalaya, Indore – 452001

Language Editors

Dr. Arti Sharan
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. Ruchi Singh
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

SLM Author(s)

Mr. Ravindra Yadav
B.E., M.E.
IET, Devi Ahilya Vishwavidyalaya, Indore – 452001

Mr. Mohit Verma
M.C.A.
SCS, Devi Ahilya Vishwavidyalaya, Indore – 452001

Edition : 2022 (Restricted Circulation)
Published by : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya
Printed at : University Press, Devi Ahilya Vishwavidyalaya, Indore – 452001

Operating System

Table of Contents

Unit 1: Introduction Operating System (OS)

Unit 2: User Interface & Computing Environments

Unit 3: Types of Operating Systems

Unit 4: Process Management

Unit 5: Theards

Unit 6: Process Scheduling: Basic Concept

Unit 7: Process Synchronization

Unit 8: Deadlocks

Unit 9: Memory Management

Unit 13: I/O System

Unit 14: System Protection

Unit 15: System Security

Unit 16: Distributed Computing

Unit 17: Distributed Computing System – An Introduction

Unit 18: Distributed File System

Unit 1: Introduction Operating System (OS)

1.0 Objective

This unit provides a brief description and understanding about one of essential resource of the computer i.e. its 'Operating System'. The unit first presents several definitions of Operating System and then provides different goals and services provided by the operating system. Other topics covered within the unit are as follows,

- Operating Systems as an Extended Machine and Resource Manager
- Operating Systems Classification
- Operating Systems Goals, Functions / Services

1.1 Introduction to Operating System

An Operating Systems are so ubiquitous in computer operations that one hardly realizes its presence. Most likely you must have already interacted with one or more different operating systems. The names like DOS, UNIX etc. should not be unknown to you. These are the names of very popular operating systems.

Try to recall when you switch on a computer what all happens before you start operating on it. In a typical personal computer scenario, this is what happens. Some information appears on the

screen. This is followed by memory counting activity. Keyboard, disk drives, printers and other similar devices are verified for proper operation. These activities always occur whenever the computer is switched on or reset.

There may be some additional activities. These activities are called power-on routines. You know a computer does not do anything without properly instructed. Thus, for each one of the power-on activities also, the computer must have instructions. These instructions are stored in a non-volatile memory, usually in a ROM. The CPU of the computer takes one instruction from this ROM and executes it before taking next instruction. One by one the CPU executes these instructions. Once, these instructions are over, the CPU must obtain instructions from somewhere. Usually these instructions are stored on a secondary storage device like hard disk, floppy disk or CD-ROM disk. These instructions are collectively known as Oper- ating System and their primary function is to provide an environment in which users may execute their own instructions.

Once the operating system is loaded into the main memory, the CPU starts executing it. Operating systems run in an infinite loop, each time taking instructions in the form of commands or programs from the users and executing them. This loop continues until the user terminates the loop when the computer shuts down.

1.1.1 Some Definitions

Operating system is the most important program on a computer. It basically runs the computer and allows other programs to run as well. The operating system does all the basic things that a computer needs to do, such as recognizing inputs from the mouse or the keyboard. It keeps track of where all the files are on the computer. It allocates resources to the various programs that are running, and it prevents unauthorized access to the computer.

The most popular operating system today is Microsoft's Windows operating system. Macintosh computers have their own operating system, the most recent of which is called Mac OS X. There are also open-source operating systems such as Linux. Some of the important definitions of operating system are given as under:

1. Operating System is a software program that acts as an interface between the user and the computer. It is a software package which allows the computer to function.
2. An operating system is a computer program that manages the resources of a computer. It accepts keyboard or mouse inputs from users and displays the results of the actions and allows the user to run applications or communicate with other computers via networked connections.

3. An operating system is the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources is it needed to write sources. (Source:dictionary.com).
4. Operating system is a platform between hardware and user which is responsible for the management and coordination of activities and sharing of resources of a computer. It hosts several applications that run on a computer and handles the operations of computer hardware.
5. An operating system is the program that, after being initially loaded into the computer bya boot pro- gram, manages all the other programs in a computer. The other programs are called applications or application programs. The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface such as a command language or a Graphical User Interface (GUI).

1.1.2 Goals

An operating system is the most important program in a computer system. This is one program that runs all the time, as long as the computer is operational and exits only when the computer is shut down.

Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system.

The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Hardware of a computer is equipped with extremely capable resources – memory, CPU, I/O devices etc. All these hardware units interact with each other in a well-defined manner. Hardware alone is not enough to solve a problem, particularly easy to use, application programs are developed. These various programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

It is easier to define operating systems by their functions, i.e., by what they do than by what they are. The computer becomes easier for the users to operate, is the primary goal of an operating system. Efficient operation of the computer system is a secondary goal of an operating system. This goal is particularly important for large, shared multi-user systems. These systems are typically expensive, so it is desirable to make them as efficient as possible.

Operating systems and computer architecture have had a great deal of influence on each other. To facilitate the use of the hardware, operating systems were developed. As operating systems were designed and used, it became obvious that changes in the design of the hardware could simplify them. Operating systems are the programs that make computers operational, hence the name.

Without an operating system, the hardware of a computer is just an inactive electronic machine, possessing great computational power, but doing nothing for the user. All it can do is to execute fixed number of instructions stored into its internal memory (ROM: Read Only Memory), each time you switch the power on, and nothing else.

Operating systems are programs that act as interface between the user and the computer hardware. They sit between the user and the hardware of the computer providing an operational environment to the users and application programs. For a user, therefore, a computer is nothing but the operating system running on it. It is extended machine.

Users do not interact with the hardware of a computer directly but through the services offered by operating system. This is because the language that users employ is different from that of the hardware. Whereas users prefer to use natural language or near natural language for interaction, the hardware uses machine language. It is the operating system that does the necessary translation back and forth and lets the user interact with the hardware. The operating system speaks users' language one hand and machine language on the other. It takes instructions in form of commands from the user and translates into machine understandable instructions, gets these instructions executed by the CPU and translates the result back into user-understandableform.

A user can interact with a computer if only he/she understands the language of the resident operating system. You cannot interact with a computer running UNIX operating system, for instance, if you do not know 'UNIX language' or UNIX commands. A UNIX user can always interact with a computer running UNIX operating system, no matter what type of computer it is. Thus, for a user operating system itself is the machine – an extended machine as shown in figure 1.1.

The computer hardware is made up of physical electronic devices, viz. memory, microprocessor, magnetic disks and the like. These functional components are referred to as resources available to computers for carrying out their computations. All the hardware units interact with each other in terms of electric signals (i.e. voltage and current) usually coded into binary format (i.e. 0 and 1) in digital computers, in a very complex way.

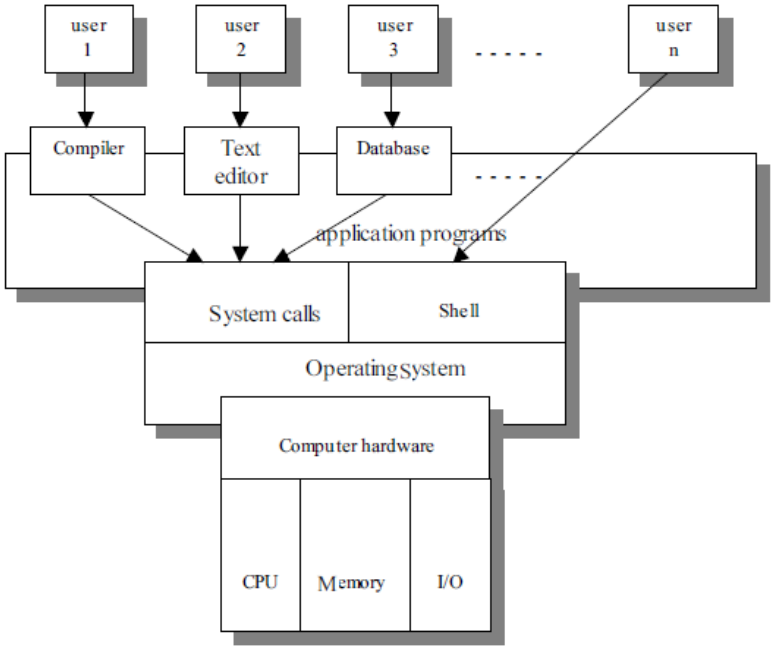


Figure 1.1: Extended-machine view of operating system.

In order to interact with the computer hardware and get a computational job executed by it, the job needs to be translated in this binary form called machine language. Thus, the instructions and data of the job must be converted into some binary form, which then must be stored into the computer's main memory. The CPU must be directed at this point, to execute the instructions loaded in the memory. A computer, being a machine after all, does not do anything by itself. Which resource is to be allocated to which program, when and how, is decided by the operating system in such a way that the resources are utilized optimally and efficiently

1.1.3 Generations

The earliest operating systems were developed in the late 1950s to manage tape storage, but programmers mostly wrote their own I/O routines. In the mid-1960s, operating systems became essential to manage disks, complex time sharing and multitasking systems.

Today, all multi-purpose computers from desktop to mainframe use an operating system. Consumer electronics devices increasingly use an OS, whereas in the past, they used custom software that provided both OS and application functionality.

Historically operating systems have been tightly related to the computer architecture, it is good idea to study the history of operating systems from the architecture of the computers on which they run. Operating systems have evolved through a number of distinct phases or generations which corresponds roughly to the decades.

(a) The 1940's – First Generations

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programs were often entered one bit at time on rows of mechanical switches (plug boards). Programming languages were unknown (not even assembly languages). Operating systems were unheard of.

(b) The 1950's – Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

(c) The 1960's – Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.

For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.

Another major feature in third-generation operating system was the technique called spooling (Simultaneous Peripheral Operations On Line). In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.

Note that spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed. Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected) terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multi-program

large number of simultaneous interactive users.

(d) Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it become possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

1.2 Classification of Operating Systems

The variations and differences in the nature of different operating systems may give the impression that all operating systems are absolutely different from each-other. But this is not true. All operating systems contain the same components whose functionalities are almost the same. For instance, all the operating systems perform the functions of storage management, process management, protection of users from one-another, etc. The procedures and methods that are used to perform these functions might be different but the fundamental concepts behind these techniques are just the same. Operating systems in general, perform similar functions but may have distinguishing features. Therefore, they can be classified into different categories on different bases. Let us quickly look at the different types of operating systems.

1.2.1 Single User– Single Processing System

The simplest of all the computer systems is a single user-single processor system. It has a single processor, runs a single program and interacts with a single user at a time. The operating system for this system is very simple to design and implement. However, the CPU is not utilized to its full potential, because it sits idle for most of the time. Figure 1.2

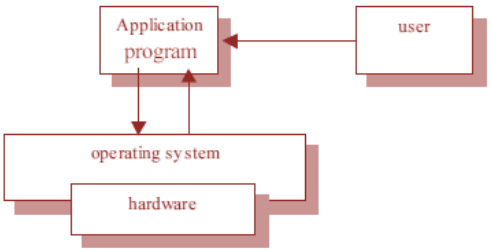


Figure 1.2 Single user – single processor system

In this configuration, all the computing resources are available to the user all the time. Therefore,

operating system has very simple responsibility. A representative example of this category of operating system is MS-DOS.

1.2.2 Batch Processing Systems

The main function of a batch processing system is to automatically keep executing one job to the next job in the batch (Figure 1.3). The main idea behind a batch processing system is to reduce the interference of the operator during the processing or execution of jobs by the computer.

1.2.3 Parallel or Multiprocessing Systems

Most systems to date are single-processor systems; that is, they have only one main CPU. However, there is a trend toward multiprocessor systems. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. These systems are referred to as tightly coupled systems.

1.2.4 Distributed Systems

A recent trend in computer systems is to distribute computation among several processors. In contrast to the tightly coupled systems, the processors do not share memory or a clock. Instead, each processor has its own memory and clock. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.

These systems are usually referred to as loosely coupled systems, or distributed systems. The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. These processors are referred to by a number of different names, such as sites, nodes, computers, and so on, depending on the context in which they arementioned.

1.2.5 Real Time Systems

Another form of a special-purpose operating system is the real-time system. A real-time system is used when there are rigid time requirements on the operation of a processor or the flow of data, and thus is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and some display systems are real-time systems. Also included are some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems.

1.3 Functions/Services of Operating Systems

As we know that operating system acts as an intermediary between the computer hardware and its users, providing a high level interface to low-level hardware resources and making it easier for the programmer and other users to access and use those resources. Some other functions/services provided by the operating system are as follows:

(a) Program Execution

The purpose of a computer system is to allow the user to execute programs. So the operating system provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems. Running a program involves the allocating and de-allocating memory, CPU scheduling in case of multiprocessing. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

(b) I/O Operations

Each program requires an input and produces output. This involves the use of I/O. Operating systems hide from the user, the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details. So, the operating system by providing I/O, makes it convenient for the users to run programs.

(c) File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating system provides this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his/her task accomplished. Thus operating system makes it easier for user programs to accomplish their task. This service involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence it is best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to provide these services but for above mentioned reasons it is best if this service s left with operating system.

(d) Communications

There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer

or running on the different computers. By providing this service, a operating system relieves the user of the worry of passing messages between processes. In case where the messages need to be passed to processes on the other computers through a network, it can be done by the user programs. The user program may be customized according to the hardware through which the message transits and provides the service interface to the operating system.

(e) Error Detection

An error occurs when one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning. This service cannot allow to be handled by user programs because it involves monitoring and in cases altering area of memory or de-allocation of memory for a faulty process. Or maybe relinquishing the CPU of a process that goes into an infinite loop. These tasks are too critical to be handed over to the user programs. A user program if given these privileges can interfere with the correct (normal) operation of the operating systems.

1.4 Summary

An Operating System is a computer program that manages the resources of a computer. It accepts inputs from users and displays the results of the actions and allows the user to run applications or communicate with other computers via networked connections. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Efficient operation of the computer system is a secondary goal of an operating system. Single user-single processor system has a single processor, runs a single program and interacts with a single user at a time. The main function of a batch processing system is to automatically keep executing one job to the next job in the batch. The objective of a multiprogramming operating system is to increase the system utilization efficiency. In a Time sharing, or multitasking system multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running. Parallel or Multiprocessing systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. Resource sharing is the main advantage of distributed systems. A real-time system is used when there are rigid time requirements on the operation of a processor or the

flow of data, and thus is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Services provided by the operating system are as follows:

1. Operating system provides an environment where the user can conveniently run programs.
2. It makes possible all Input/Output operations.
3. Operating system makes it easier for user programs to accomplish their task.
4. Operating system relieves the user of the worry of passing messages between processes.
5. Operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

Self-Assessment Questions

1. Discuss the inconveniences faced by a user interacting with a computer system without an operating system.
2. Give various definitions of operating system.
3. Discuss various types of operating systems with suitable examples.
4. Differentiate between Multiprogramming and Multiprocessing operating systems with suitable examples.
5. What are the various services provided by the operating system? Discuss.
6. Operating system acts as resource manager. What resources does it manage?
7. What are the benefits of multiprogramming?
8. What are the characteristics of real time operating systems?
9. How does a distributed system enhance resource sharing?
10. What are the constraints of a real time system?

Unit 2: User Interface & Computing Environments

2.0 Objective

This unit provides a brief description and understanding about most often terms of an Operating systems i.e. 'User Interfaces' and 'Computing Environments'. This unit first presents a brief description about Comm and Interpreter and Graphical User Interface and then after it provides information about various types of 'computing Environments' such as Client-Server Computing, Peer-to-Peer Computing, Web based Computing etc. Self answering questions and names of various reference books are also provided at the end of this unit.

2.1 Introduction

When the computer is turned on, a small 'boot program' loads the operating system. Although additional system modules may be loaded as needed, the main part, known as the 'kernel' resides in memory at all times. The operating system (OS) sets the standards for all application programs that run in the computer. Applications 'talk to' the operating system for all user interfaces and file management operations. Also called an 'executive' or 'supervisor', an operating system performs some functions like – User Interface, Job Management, Task Management, Data Management, Device Management and Security etc.

2.2 User Interfaces

Operating system works as an Interface between 'Computer Hardware' and the 'User'. Graphics based user interface includes the windows, menus and method of interaction between the user and the computer. Prior to graphical user interfaces (GUIs), all operation of the computer was performed by typing in commands. Not at all extinct, a command-line interface is included in all major operating systems, and certain highly technical operations must be executed from the command line.

Operating systems may support optional interfaces. Although the overwhelming majority of people work with the default interface, different 'shells 'offer variations of appearance and functionality. There are mainly two types of user interfaces available and known as – Command User Interface (CUI) and **Graphical User Interface** (GUI). A CUI and a GUI are the two most common interface types associated with computers. They allow the user to interact with the machine, giving it commands and viewing text or graphics.

2.2.1 Command Interpreter/ Character-Based User Interface (CUI)

CUI stands for Character-based User Interface. Early computers operated on CUIs, and they're most iconic for their typical two-colored, black and green screens with nothing but text. Users would use a keyboard to both navigate (using hotkeys) and enter commands. There was no need for a mouse in these early days, as Character-Based User Interfaces do not support such advanced hardware.

A program which reads textual commands from the user or from a file and executes them is called 'Command Interpreter'. Some commands may be executed directly within the interpreter itself (e.g. setting variables or control constructs), others may cause it to load and execute other files.

In other words, a 'Command Interpreter' is the part of a computer operating system that understands and executes commands that are entered interactively by a human being or from a program.

In some operating systems, the command interpreter is called the shell. When an IBM PC is booted BIOS loads and runs the MS-DOS command interpreter into memory from file COMMAND.COM found on a floppy disk or hard disk drive. The commands that COMMAND.COM recognizes (e.g. COPY, DIR, PRN) are called internal commands, in contrast to external commands which are executable files. According to technology website, PC Mag, the CUI was most common on older mainframe and minicomputer terminals in the early days of computing. Some early computers even featured character-based user interfaces, the most notable being Macintosh's Apple II computer. While CUIs have since fallen out of

favor for graphical user interfaces, most modern operating systems feature a modified version of a CUI called a command line interface.

2.2.2 Graphical User Interface (GUI)

A graphical user interface (GUI) is a computer environment that simplifies the user's interaction with the computer by representing programs, commands, files, and other options as visual elements, such as icons, pull-down menus, buttons, scroll bars, windows, and dialog boxes. By selecting one of these graphical elements, through either use of a mouse or a selection from a menu, the user can initiate different activities, such as starting a program or printing a document. Prior to the introduction of GUI environments, most interactive user interface programs were text oriented and required the user to learn a set of often complex commands that could be unique to a given program.

The first GUI was developed in the 1970s by Xerox Corporation, although GUIs did not become popular until the 1980s with the emergence of the Apple Macintosh computer. Today, the most familiar GUI interfaces are Apple Computer's Macintosh and Microsoft Corporation's Windows operating systems.

Computer software applications, such as word processing and spreadsheet packages, typically use the set of GUI elements built into the operating system and then add other elements of their own. The advantage of the GUI element of any software program is that it provides a standard method for performing a given task (i.e., copying a file, formatting text, printing a document) each time the user requests that option, rather than creating a set of commands unique to each potential request.

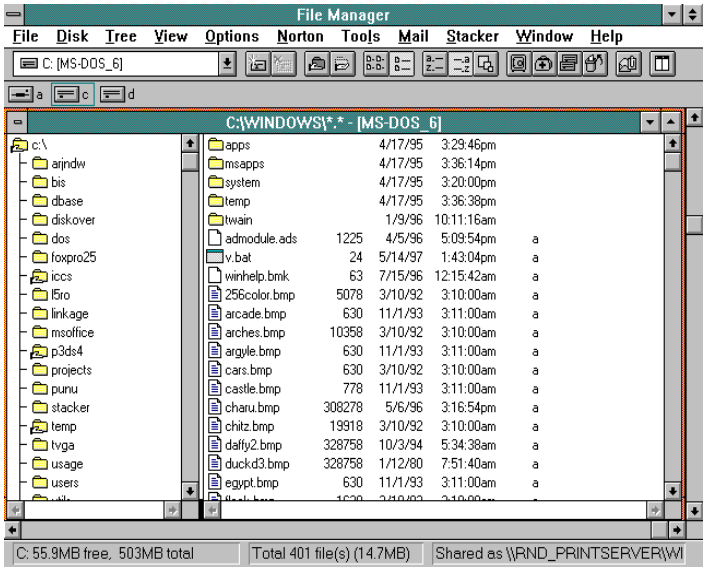


Figure 2.1: Essential components of Windows Operating system as a GUI

Many GUI elements are standard across all packages built on the same operating system, so once a user is familiar with the GUI elements of one package, it is easier to then work in other packages.

Figure 2.2: The First Commercial GUI

Generally, because of their GUI elements, any two programs even from different developers that are built on the same operating system are able to share data, thereby saving a user from having to re-key the same information for use in different programs. For example, a user can copy a graph created in a spreadsheet package and place, or 'paste,' it into a word processing document. GUI interfaces also typically offer more than one method for initiating a particular action. For example, to print a document from a program within the Windows environment, a user can select the 'Print' option from the 'File' menu, click the printer icon, or, as an alternative, use the keyboard shortcut of holding down the Ctrl key and pressing the letter 'P.' A user can then employ the option that feels most comfortable to him or her across all Windows programs. The GUI interface has also been instrumental in making the World Wide Web easily accessible to individuals through the use of the use of GUI-based 'browser' programs. Netscape Navigator, Internet Explorer, and similar programs enable a user to access and search the web using the familiar GUI format.

Today, GUIs are synonymous with personal computing. GUIs are much more advanced than their earlier counterparts because of all the computing and graphical power available today. While many operating systems allow for full control with a keyboard, these systems are much easier to operate with the addition of a mouse.

2.2.3 Difference between CUI and GUI

CUI and GUI are acronyms that stand for different kinds of user interface systems. These are terms used in reference to computers. CUI stands for Character User Interface while GUI refers to Graphical User Interface. Though both are interfaces and serve the purpose of

running the programs, they differ in their features and the control they provide to the user. CUI means you have to take help of a keyboard to type commands to interact with the computer. One can only type text to give commands to the computer as in MS-DOS on command prompt. There are no images or graphics on the screen, and it is a primitive type of interface. In the beginning, computers had to be operated through this interface and users who have seen it say that they had to contend with a black screen with white text only. In those days, there was no need of a mouse as CUI did not support the use of pointer devices. CUI's have gradually become outdated with the more advanced GUI taking their place. However, even the most modern computers have a modified version of CUI called CLI (Command Line Interface). GUI is an interface that makes use of graphics, images and other visual clues such as icons. This interface made it possible for a mouse to be used with a computer and interaction really became very easy as the user could interact with just a click of the mouse rather than having to type every time to give commands to the computer. Some other differences are given as under,

- CUI is the precursor of GUI and stands for character user interface where user has to type on key- board to proceed. On the other hand GUI stands for Graphical User Interface which makes it possible to use a mouse instead of keyboard;
- GUI is event driven in nature but CUI is sequence oriented in nature;
- GUI is much easier to navigate than CUI;
- There is only text in case of CUI whereas there are graphics and other visual clues in case of GUI;
- Most of modern computers use GUI and not CUI;

MS-DOS is an example of CUI whereas MS-Windows is an example of GUI.

2.3 Computing Environments

An operating system may process its tasks serially or simultaneously, which means that the resources of the computer system may be dedicated to a single program until its completion or they may be allocated among several programs in different stages of execution. So, there are several computing environments as discussed below.

2.3.1 Traditional Computing

Machine language was quite common for early computer systems. Instructions and data used to be fed into the computer system by means of console switches through a hexadecimal keyboard. Programs used to be started by loading the program computer register with the address of the first instruction of a program and its result used to be examined by the contents

of various registers and memory locations of the machine. This programming style caused a low utilization of users and machine. Program started being coded into programming language are first changed into object code (i.e. binary code) by translator and then automatically gets loaded into memory by a program called Loader. After transferring the control to the loaded program, the execution of program begins and its results get displayed or printed. Once in memory, the program may be re-run with a different set of input data. This type of computing environment is called 'Serial Processing' computing.

'Batch Processing' Computing is another type of traditional computing environment. The main function of a batch processing system is to automatically keep executing one job to the next job in the batch. The main idea behind a batch processing system is to reduce the interference of the operator during the processing or execution of jobs by the computer. All functions of a batch processing system are carried out by the batch monitor. The batch monitor permanently resides in the low end of the main store. The current jobs out of the whole batch are executed in the remaining storage area. In other words, a batch monitor is responsible for controlling all the environment of the system operation. The batch monitor accepts batch initiation commands from the operator, processes a job, performs the job of job termination and batch termination.

2.3.2 Client-Server Computing

Client–server computing is a distributed computing model in which client applications request services from server processes. Clients and servers typically run-on different computers interconnected by a computer network. Any use of the Internet, such as information retrieval from the World Wide Web, is an example of client–server computing.

However, the term is generally applied to systems in which an organization runs programs with multiple components distributed among computers in a network. The concept is frequently associated with enterprise computing, which makes the computing resources of an organization available to every part of its operation.

A client application is a process or program that sends messages to a server via the network. Those messages request the server to perform a specific task, such as looking up a customer record in a database or returning a portion of a file on the server's hard disk. The client manages local resources such as a display, keyboard, local disks, and other peripherals. The server process or program listens for client requests that are transmitted via the network. Servers receive those requests and perform actions. Server processes typically run on powerful PCs, workstations, or mainframe computers.

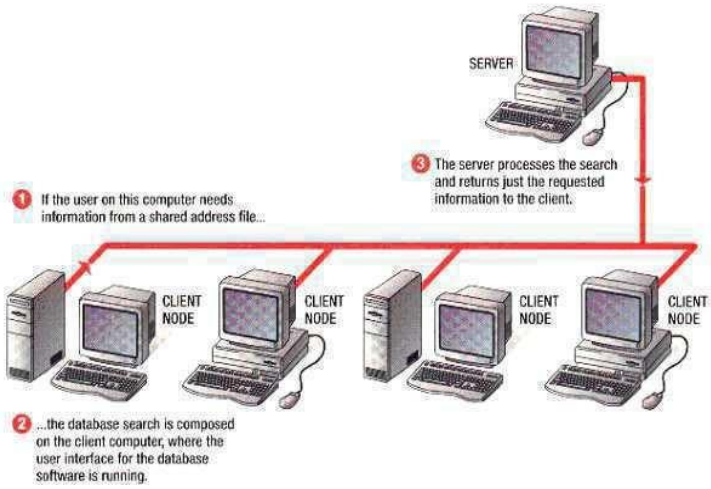


Figure 2.3: Client-Server Computing Environment

An example of a client–server system is a banking application that allows a clerk to access account information on a central database server. All access is done via a PC client that provides a graphical user interface (GUI). An account number can be entered into the GUI along with how much money is to be withdrawn or deposited, respectively. The PC client validates the data provided by the clerk, transmits the data to the database server, and displays the results that are returned by the server. The client–server model is an extension of the object based (or modular) programming model, where large pieces of software are structured into smaller components that have well defined interfaces. This decentralized approach helps to make complex programs maintainable and extensible. Components interact by exchanging messages or by Remote Procedure Calling. The calling component becomes the client and the called component the server.

A client–server environment may use a variety of operating systems and hardware from multiple vendors; standard network protocols like TCP/IP provide compatibility. Vendor independence and freedom of choice are further advantages of the model. Client–server systems can be scaled up in size more readily than centralized solutions since server functions can be distributed across more and more server computers as the number of clients increases. Server processes can thus run in parallel, each process serving its own set of clients. However, when there are multiple servers that update information, there must be some coordination mechanism to avoid inconsistencies.

The drawbacks of the client–server model are that security is more difficult to ensure in a distributed environment than it is in a centralized one, that the administration of distributed equipment can be much more expensive than the maintenance of a centralized system, that data distributed across servers needs to be kept consistent, and that the failure of one server can render a large client–server system unavailable. If a server fails, none of its clients can make

further progress, unless the system is designed to be fault tolerant.

The computer network can also become a performance or reliability bottleneck: if the network fails, all servers become unreachable. If one client produces high network traffic then all clients may suffer from long response times.

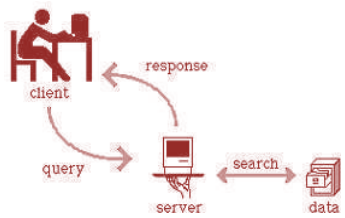


Figure 2.4: A typical Client-Server interaction

2.3.3 Peer-to-Peer Computing

The term 'peer-to-peer' (P2P) refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner. With the pervasive deployment of computers, P2P is increasingly receiving attention in research, product development, and investment circles.

In this type of computing all nodes on the network have equal relationships to all others, and all have similar types of software. Typically, each node has access to at least some of the resources on all other nodes, so the relationship is nonhierarchical. If they are set up correctly, Operating system give users access to the resources attached to other computers in the network. A peer-to-peer computing environment is shown in Figure 2.5.

In addition, some high-end peer-to-peer networks allow distributed computing, which enables users to draw on the processing power of other computers in the network. That means people can transfer tasks that take a lot of CPU power such as creating computer software to available computers, leaving their own machines free for other work.

Peer-to-peer computing environment is commonly set up in small organizations (fewer than 50 employees) or in schools, where the primary benefit of a network is shared storage, printers, and enhanced communication. Where large databases are used, LANs are more likely to include client/server relationships.

A peer-to-peer network can also include a network server. In this case, a peer-to-peer LAN is similar to a file server network. The only difference between them is that the peer-to-peer network gives users greater access to the other nodes than a file server network does.

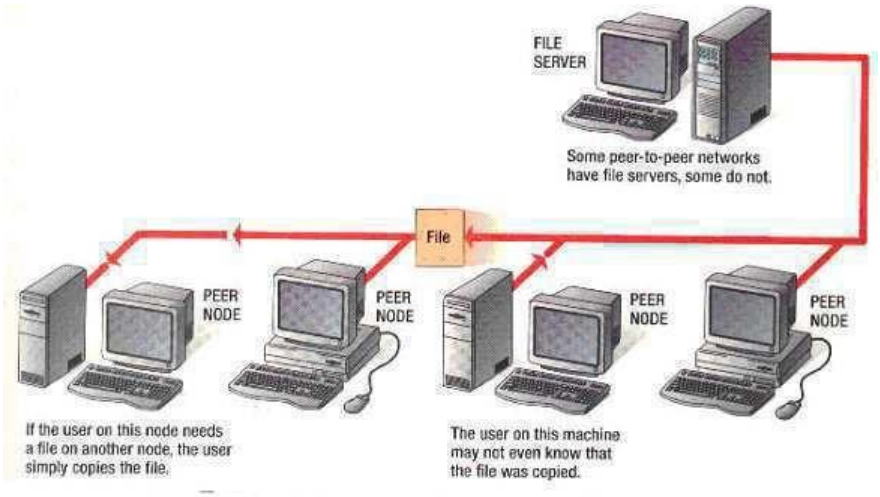


Figure 2.5: Peer-to-Peer Computing Environment

Some of the benefits of a P2P approach include improving scalability by avoiding dependency on centralized points; eliminating the need for costly infrastructure by enabling direct communication among clients; and enabling resource aggregation.

2.3.4 Web-Based Computing

Web computing can be defined as a special kind of distributed computing that involves internet-based collaboration of several remotely located applications. The idea behind Web Computing is to make distributed computing accessible to as many people as possible.

Browsers today have powerful and highly optimized JavaScript engines that in many cases are capable of providing computing capabilities comparable to native solutions. The Web Computing framework is a JavaScript library that provides client functionality required by distributed computing solutions. Web applications based on Web Computing are capable of requesting jobs from a number of work sources, downloading and executing them, uploading the results and handling various messages.

2.3.5 System Calls

System calls provides an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process. It is because of the critical nature of operations that the operating system itself does them every time they are needed. For example, for I/O a process involves a

system call telling the operating system to read or write particular area and this request is satisfied by the operating system. System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls. Modern processors provide instructions that can be used as 'System Calls'. System calls provide the interface between a process and the operating system. A system call instruction is an instruction that generates an interrupt that cause the operating system to gain control of the processor.

System Call works in the following ways,

1. First the program executes the system call instructions.
2. The hardware saves the current (instruction) and PSW register in the ii and iPSW register.
3. 0 value is loaded into PSW register by hardware. It keeps the machine in system with interrupt disabled.
4. The hardware loads the i register from the system call interrupt vector location. This completes the execution of the system call instruction by the hardware.
5. Instruction execution continues at the beginning of the system call interrupt handler.
6. The system call handler completes and executes a return from interrupt (rti) instructions. This restores the i and PSW from the ii and iPSW.

The process that executed the system call instruction continues at the instruction after the system call.

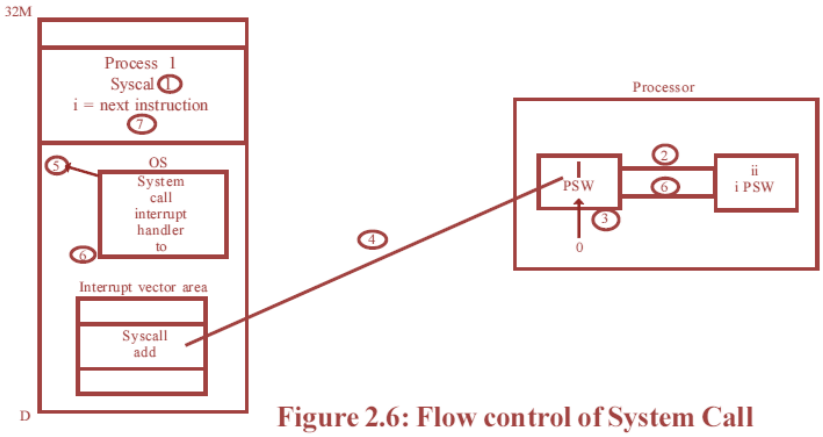


Figure 2.6: Flow control of System Call

A system call is made using the system call machine language instructions. System calls can be grouped into following five major categories-

1.	File Management

2.	Interprocess Communication
3.	Process Management
4.	I/O Device Management
5.	Information maintenance.

System calls used for create and delete the files. System calls require name of the file with file attributes for creating and deleting files. Other operations on files are read a file, write and reposition the file after it open. After all file is closed with close system call. For directories, same set of operations are performed. Sometimes, we required to reset some of the attributes on files and directory. The system call, get file attribute and set file attribute are used for this type of operations. Some OS provide many more calls, such as calls for file move and copy. System calls for 'File Management' are as follows

Create	Create a new file and Open it.
Delete	Delete a file.
Open	Open a file to read or write.
Close	Close a file, indicating that file is no longer using it.
Read	Read a byte from an open file.
Write	Write a byte to an open file.
Stat	Get information about a file.
Unlink	Remove a file from a directory.

Get/Set file attribute includes file name, file type, protection codes and accounting information. System call is used for terminating the current running process abnormally. Running program halted by tow ways: normal or abnormal. Reasons for abnormal termination of programs are – dump of memory, error message generated causes an error trap etc. debugger is used to determine the problem of dump. Dump is written to disk. OS transfer control to the command interpreter in normal or abnormal conditions. In batch OS, the command interpreter usually terminates the entire job and continues with the net job. Some operating systems allow control cards to indicate special recovery action in case an error occurs. It is possible to combine normal and abnormal termination at some error level. Error level is defined before combines command interpreter use these error level to determine the next action automatically. Load and Execute system calls are used by process to execute one program. This feature allows the command interpreter to execute a program. Get process attributes and set process attributes system calls are used to determine and reset the attributes of a job or process. It also includes job priority, its maximum allowable execution time etc. terminate

process is the system call to terminate the process or job. Wait time and wait event are the system call used in waiting condition of the process. When process is created, it may need to wait for them to finish their execution. Certain amount of time is required to wait before to complete the operation. System call of this type is dealing with the coordination of concurrent processes. Debugging a program is also provided by system calls. Microprocessors also provide a single stepping execution of program. Trap is executed by the CPU after every instruction. Debugger caught the trap, which is a system program designed to aid the programmer in finding and correcting bugs.

System calls for '**Interprocess Communication**' are as follows-

Create message queue	Create a queue to hold message.
Send message	Send a message to a message queue.
Receive message	Receive a message from a message queue.
Close connection	Terminates the communication.

System calls for 'Process Management' are as follows,

Create process	Create a new process.
Terminate process	Terminate the process making the system call.
Wait	Wait for another process to exit.
Fork	Create a duplicate of the process making a system call.
End	Halt the process execution normally
Abort	Halt the process execution abnormally.
Load	Load the process into memory.
Execute	Execute the loaded process.
Get process attributes and set process attributes	To Get and Set process attributes
Allocate and free memory.	To allocate and to release the memory

For accessing device, system calls are used. Many of the system calls for files are also needed for devices. In multi-user environment, request is made before to sue of that device. After using any device, it must be release. So using release system call, device is free to use by another user. These functions are similar to the open and close system call of files. System calls for 'I/O Device Management' are as follows-

Request device	To ensure exclusive use of device.
Release device	Release the device after finished with the device.
Read, write	Same as file system call.
State	Get information about an I/O device.

System calls for 'Information maintenance' are as follows-

Get time and date
Set time and date
Set process, file or device attributes.
Get process, file or device attributes
Get system data
Set system data.

The operating system provides a set of operations which are called system calls. A system call interface is the description of the set of system calls implemented by the operating system.

2.5 Summary

Operating system works as an Interface between 'Computer Hardware' and the 'User'. There are mainly two types of user interfaces available and known as 'Command User Interface' (CUI) and 'Graphical User Interface' (GUI). A program which reads textual commands from the user or from a file and executes them is called 'Command Interpreter'. A graphical user interface (GUI) is a computer environment that simplifies the user's interaction with the computer by representing programs, commands, files, and other options as visual elements, such as icons, pull-down menus, buttons, scroll bars, windows, and dialog boxes. DOS is an example of CUI whereas Windows is an example of GUI. GUI is event driven in nature, but CUI is sequence oriented in nature. Client–server computing is a distributed computing model in which client applications request services from server processes. Clients and servers typically run-on different computers interconnected by a computer network. A client application is a process or program that sends messages to a server via the network. The term 'peer-to-peer' (P2P) refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized man- ner. In this type of computing all nodes on the network have equal relationships to all others, and all have similar types of software. Web computing can be defined as a special kind of distributed computing that involves internet-

based collaboration of several remotely located applications. The idea behind Web Computing is to make distributed computing accessible to as many people as possible. System calls provide the interface between a process and the operating system. A system call instruction is an instruction that generates an interrupt that cause the operating system to gain control of the processor.

Self Assesment Questions

1. What is a system call? Give some examples of system calls.
2. Discuss the following with proper illustration,
   - Peer-to-Peer computing environment
   - Client-Server Computing
3. What do you understand by 'Web-based' computing?
4. Write a short note on various types of 'User Interfaces'.
5. What is GUI? Give any three examples of GUI.
6. What is Command interpreter? Give any one example of CUI.

Unit 3: Types of Operating Systems

3.0 Objective

After studying, this unit you will be able to understand the various types of operating systems and their evolution. Also, you will be able to differentiate between the different types of operating systems according to their features and working.

3.1 Introduction

An operating system is an integrated set of programs that controls and coordinates the use of hardware among various application programs for different users. It acts as an interface between the users and computer hardware (see Fig 3.1).

The operating system provides varieties of facilities and services to assist a programmer in creating programs. These facilities are in the form of utilities such as editors, compilers, interpreters, etc. Besides this the operating system performs function such as hiding details of the hardware, resource management, etc. Due to the complexity of an operating system, it must be created piece by piece. Each of these pieces must be well delineated portion of the system, with clearly defined inputs, outputs, and functions.

The most commonly used operating systems are Windows 95, Windows 98, WindowsNT,

Windows XP, OS/2, Unix, Mac OS, Linux, Windows 7.0, etc. These operating systems simultaneously manage information measuring 16 bits, 32 bits, 64 bits or more.

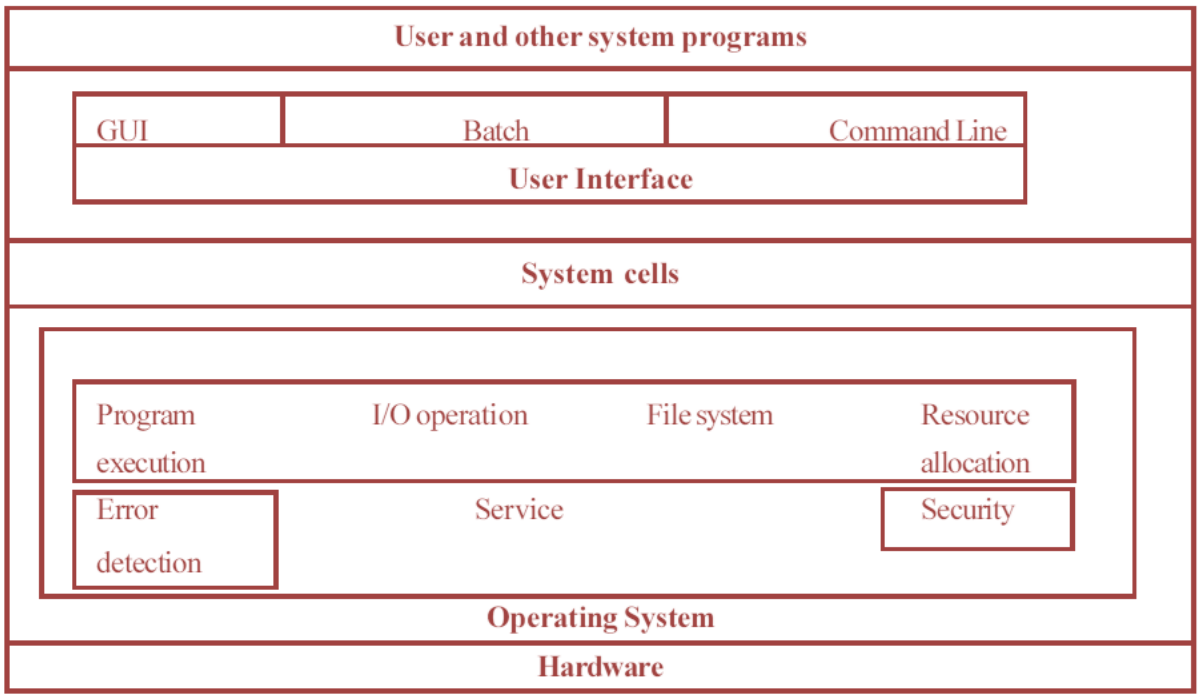


Figure 3.1: Operating System as an Interface

3.2 Evolution of Operating System

In early computer system, after writing program on the paper, the programmer or a data entry operator punches the program and its data on the cards or paper taps. Then onward, the programmer submits the deck of cards or the paper taps containing the program and the data at the computer centre, where an operator loads it manually in the system. Before loading it, the operator clears the data from previous job, remaining in the main memory. Finally, the operator prints the result of the execution of the job at the computer centre, which is collected by the programmer, later on. The same process was repeated for every job. This whole process was known as Manual Loading Mechanism. In manual Loading mechanism, job to job transition was not automatic, due to which computer remained idle, while an operator load and unloads jobs and prepared the system for a new job. These actions wasted a lot of central processing unit time.

In order to reduce this idle time and speed up the processing, automatic job to job transition method was devised. This method was called Batch Processing (BP), where, jobs with similar need were batched together and were seen as a group. In this method when one job finishes, system control is transferred backed automatically to the operating system that performs jobs needed to load and run the next job (as shown in Figure 3.2).

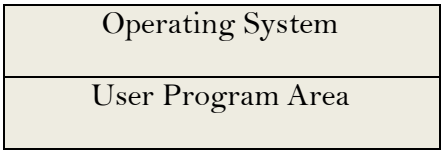


Figure 3.2: Simple Batch System

Batch Processing helped in reducing, not only the idle time of a computer system, but also, the set up time required by the operator to batch similar jobs together. For example, if all FORTRAN compilation jobs are batched together, the FORTRAN Compiler needs to be loaded once only for processing of all these jobs. Control statements were used by the operating system to identify new job and to determine the resources needed by it during its execution. These control statements were written in a language which was known as Job Control Language (JCL). Typical JCL commands included, making of job beginning and end, command for loading and execution of the program and commands to announce resource needs.

However, the delay between job submission and completion increased in batch processing system, where, a large number of programs were put in a batch and processed. In order to improve the performance of the system, two approaches were developed - Buffering and Spooling. Buffering is a method of overlapping input, output and processing of a single job whereas Spooling allows central processing unit to overlap the input of one job with the computation and output of the other jobs. Spooling uses the disk as a very buffer for reading and for storing output files. Although, disks were faster than the card reader and printers, they had their limitation.

Performance of the system improved with the help of buffering and spooling, but, both of them have their limitations. An efficient approach was developed to increase the system performance and resource utilization, so that central processing unit always has a job to execute. This approach known as Multiprogramming approach interleaved execution of two or more different and independent program by a computer (as shown in Figure 3.3).

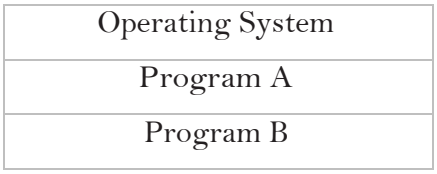


Figure 3.3: Multi programmed Batch System

Thus, in this approach multiple program are available to the CPU and a portion of one is executed, then a portion of another and so on.

3.3 Types of Operating System

The operating system is considered as a backbone of a computer managing, both, hardware and software resources. The operating system may be classified as single- user system and multi-

user system. In a single-user system, the operating system acts as a interface for only one user whereas in multi-user system, it act as an interface for more the one user. We will know discuss different types of operating system.

3.3.1 Batch Processing Operating System

This operating system requires grouping of similar jobs which consist of programs, data and system commands. In batch processing, jobs are loaded into a system and processed one at a time. Once loaded, a job remains in the main memory until its execution is over and next job is loaded automatically, only, after the completion of current job. This type of processing is suitable for those programs which require large computation time. In this processing, there is little or no need of user interaction. The user prepares their program and data and submits them to the operator. The operator gives a command to the system to start executing the batched jobs. When all jobs in the submitted batch were processed, the operator takes out the printout and keeps them. These printouts are collected by the user later on. Some example of such programs includes payroll, forecasting, etc.

In the Batch Processing, the Process Scheduling, Memory Management, File Management and Input/ output Management are simple. As only one program is executive at a time, there is no competition for Input/output devices. Also, there is hardly a need for the file access control mechanism in this processing. Here the memory is divided into two parts - one part contains operating system routines while the other part contain user program to be executed.

Limitation:

(1) Time taken between job submission and job completion is very high

(2) As user interaction is very limited, they have no control over the intermediate results of a program.

(3) The programmer cannot correct bugs, the moment it occurs in this approach. Thus programs must be debugged.

3.3.2 Multiprogramming Operating System

Suppose there is a single program resident in the main memory, and it is being executed by the CPU. If the program is I/O bound, then the program would run for a moment and then pass into the wait state (see Figure 3.4). During the wait period, the CPU remains idle. Moreover, irrespective of the type of program, whenever, a program finishes its job, the operating system loads new program from the hard disk into the memory and hand over the control to CPU. The

Hard Disk being a slower device as compared to Central Processing Unit, a lot of time is wasted during loading of the new program and obviously the CPU remains idle in the mean time.

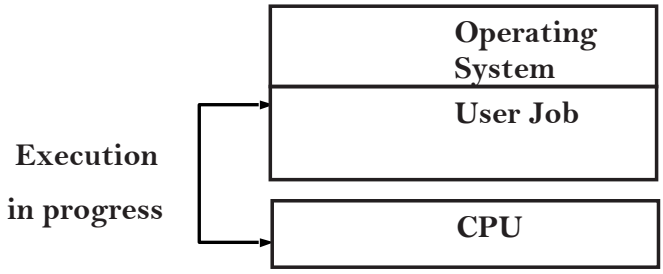


Figure 3.4: Uniprogramming System

The remedy to the above cited problem is multiprogramming. This type of operating system allows con- current residency of many programs in the main memory of the computer. As there is more than one program resident in the main memory, another program is available for the execution in a situation where the current executing program enters in the wait state. Thus the CPU will spend less time idle. When a program finishes its job, the Central Processing Unit is immediately allotted to another program residing in the main memory and thus, no time is wasted. The vacancy created by the ongoing program is filled by the operating system who loads a new program from the hard disk to the vacant area in the main memory.

One of the unique features of multiproamming system is that storage is allocated for each program. The areas of primary storage allocated for the individual programs are called partitions. Each partition must have some form of storage and priority protection to ensure that a program in one portion will not accidently writes over and destroys the instruction of another partition and priority because both, programs will need access to the central processing unit facilities. Figure 3.5 gives illustration of multiprogramming system where there three jobs (1, 2 and 3) residing in the memory, out of which job1 is performing I/O operation, job2 is executing and job3 is waiting for the CPU to become free.

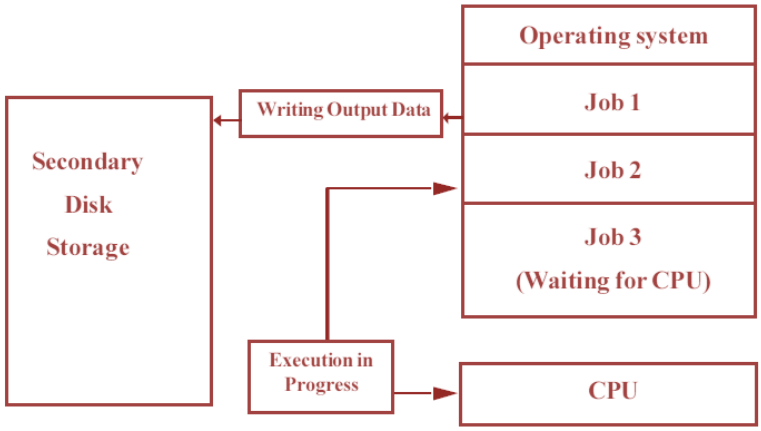


Figure 3.5: Multiprogramming System

The Multiprogramming Operating System results in greater memory efficiency and lesser Central Processing Unit idle time. In a Multiprogramming system, often there are multiple jobs in ready state. Hence when CPU becomes free, the operating system must decide which of these ready jobs should be allocated to the CPU for execution. CPU Scheduler is that part of the operating system which takes such a decision.

Although this operating system is complex but still it is preferred due to its higher processor efficiency.

3.3.3 Multitasking Operating System

Multitasking Operating System has the ability to execute two or more of a single user's task concurrently. Thus in Multitasking, a single job may contain two or more task that can execute concurrently in multiprogramming mode. On a single CPU system, Multitasking allows number of processes to cooperate in achieving an activity that can be parcelled into smaller concurrent activities. An internet browser that search for some information is an example of a task. The computer user can switch back and forth between active task to see results, enter a new request or data, etc. For microcomputer, Multitasking OperatingSystem provide single user with multiprogramming capabilities. This is often accomplished through foreground / background processing. Most modern operating system like Windows, OS /2. UNIA, Macintosh System 7, etc support Multitasking. It is a fact that Multitasking is possible when the operating system supports multiprogramming.

In UNIX operating system, the user may specify multitasking by fallowing a command with an ampersand (&). In this case shell does not work for the execution of the command to finish. It immediately prompt for a new command while the previous command continue execution in the background.

In order to differentiate between Multiprogramming and Multitasking, we can say that Multiprogramming is interleaved execution of multiple jobs in a multi-user system whereas Multitasking is interleaved execution of multiple jobs after referred as task in a single-user system.

3.3.4 Time Sharing Operating System

This operating system works in an interactive mode with a quick response time. Time Sharing System allows simultaneous interactive use of a computer resource by many users in such a way that each one feels that he / she is the sole user of the system. It uses multiprogramming with a special CPU Scheduling Algorithm to achieve this. In Time sharing System, the executive time is divided into small slots called 'time slice'. Each process is process

for a time slice and then the other process is taken for processing by the processor. This process goes on till all the jobs are processed.

This system contains many user terminals connected to the same computer, simultaneously. Using these terminals, multiple users can work on the same system simultaneously. Multiprogramming feature allow multiple user program to reside simultaneously in the main memory and CPU scheduling algorithm allo- cates time slice, one by one to each user process. Figure 3.6 illustrate process state diagram of Time Sharing System.

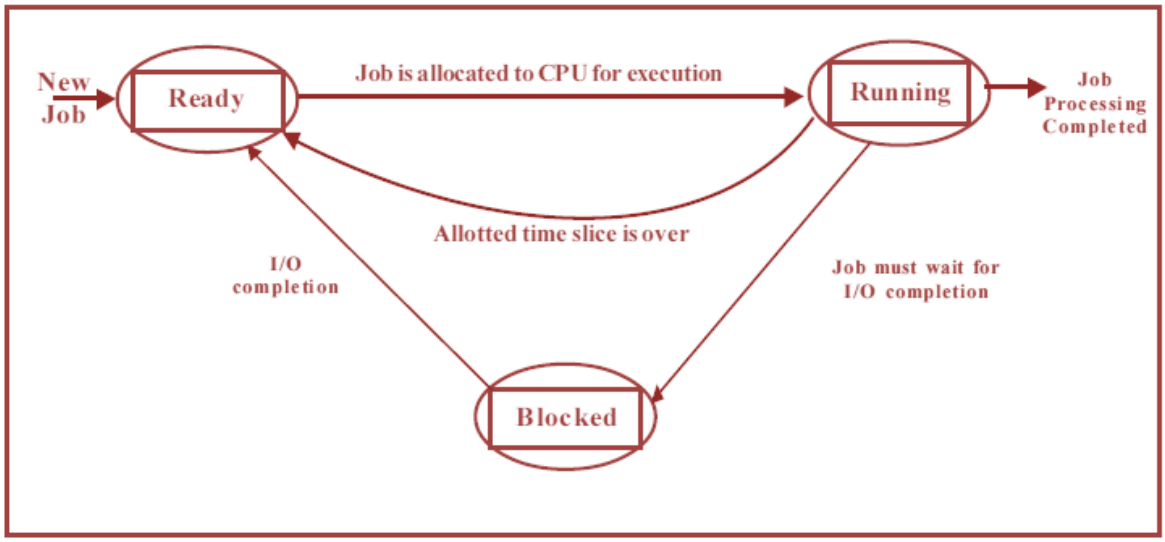


Figure 3.6: Process state diagram for a Time Sharing System

As in Time Sharing System, time slice scheduling of CPU is used, the programs are executed with rotating priority which increases during waiting and drops, after the service is granted. The operating system interrupts the programs which are in execution, longer than system defined time slice, to prevent a program monopolizing the processor. Time Sharing provides Memory Protection Mechanism to prevent a job's interaction and data from other job in a multiprogramming environment. It has relatively large memory to support multiprogramming. File Management in this system provides protection and access control. Job's state Preservation Mechanism and CPU Scheduling Algorithm are unique features of this operating system.

Although Time Sharing System are complex due to their large memory requirement, still they have an edge over other system. This system helps in reducing CPU idle time. Its special CPU scheduling Algorithm ensures quick response time to all users. This feature helps improving the programmer's efficiency by making interactive programming and debugging much simpler and quicker.

3.3.5 Real Time Operating System

Real Time Operating System is designed to respond to events that happen in real time. They are used in environments, where a large number of events, mostly external to the computer system, must be accepted and processed in short time. It is a computer system that require not only that computing results be correct but, also, that the results be produced within a specific deadline. Results produced after the deadline has passed, even if correct, may be of no real value. Real Time system finds application in automobiles, aeroplane, home appliance like microwave oven and dishware's, in consumers digital devices like cameras and MP3 player, etc.

A soft real time system is less restrictive and in it critical real time task receives priority over other tasks and it retains that priority until it completes. For example, Linux provides soft real time support.

A hard real time system has stringent requirement and in it critical real time task is to be completed within their deadlines. Safety critical systems are hard real time system.

The design of the Real Time Operating System reflects its single purpose nature and is often quite simple. Their microprocessors are inexpensively mass produced. The main objective of R.T.O.S. is to provide quick response time. Accordingly, the defining characteristics of the system are to support the time requirements of the real time tasks. Real Time System meet the time requirement by using Scheduling algorithm that gives real time processes the highest scheduling priorities. Also, the schedulers must ensure that the priority of a real time task does not degrade over time. The processor is normally allocated the highest priority process among those which are ready to execute, user convince and resource utilization are of secondary concern. As there is a little swapping of program between primary and secondary memory, the memory management is less demanding in this system, however, the processes tend to provide protection and sharing of the memory. Real Time System provides sophisticated form of interrupt management and I / O buffering. As compare to file management, concern is on the speed of access rather than efficient utilization of the secondary storage.

3.3.6 Multiprocessor Operating System

Multiprocessor system is that system which is composed of several independent processors. These processors operate in parallel, thereby allowing, simultaneous execution of several programs or of several parts of the same program. Basic organisation of Multiprocessing system is given in Figure 3.7.

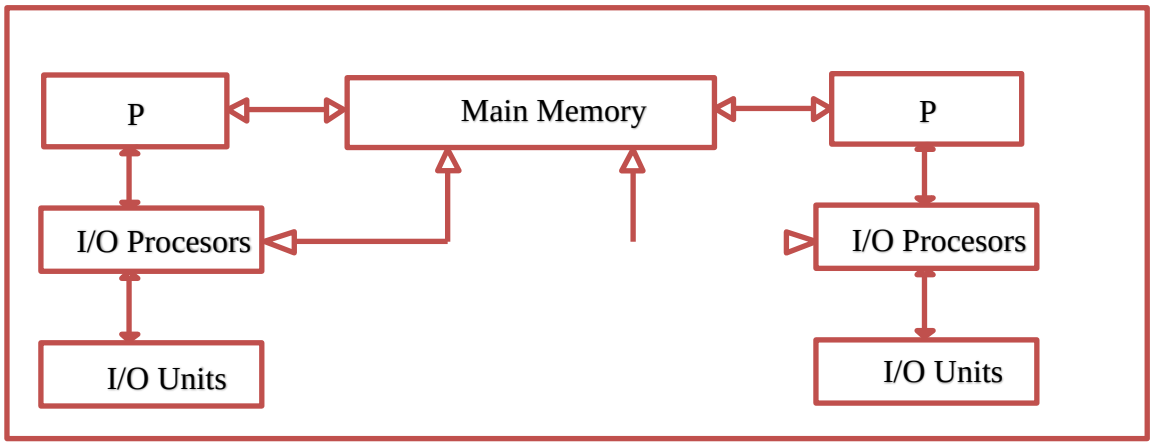


Figure 3.7: Basic Organisation of Multiprocessing system

Multiprocessor System are of two types-

(1) **Loosely Coupled Multiprocessor System -** In this system, the memory and I/O units are attached to an individual processor. Thus, for each processor there is a local memory and an I/O unit module. Hence, in this system no sharing of memory and I/O are permitted. The interprocessor communication is achieved through messages.

(2) **Tightly Coupled Multiprocessor System -** In this system, there is a single-wide primary memory shared by all processors. Hence shared memory access is permitted and a word-by-word interaction among communicating process is possible. Thus tightly coupled multiprocessor system provides means to share information interchange and synchronization among processes through a shared location.

Multiprocessing offers data processing capabilities that are not present when only one CPU is used.

Multiprocessing System have better performance due to shorter response time and higher throughput. They have better reliability, that is, if one of the processor break down, the other processor(s) takes over the system workload automatically until the faulty processor is repaired. Thus in this system many complex operations can be performed at the same time. Multiprocessing finds its application in rail control, traffic control, airways, etc.

Multiprocessing System has its limitation. These systems have high initial cost and their regular operation and maintenance is costlier than single processor system. Also, designing of such system is very complex and time taking.

3.3.7 Distributed Operating System

A Distributed Operating System is a collection of loosely coupled processors that do not share memory. These processors communicate with one another through various communication networks. Thus, this system allows users to share resources on geographically dispersed hosts connected via a computer network. The processors in a Distributed System may vary in size and functions. They may include small microprocessor, workstation, minicomputer and large general purpose computer system. A general structure of a Distributed System is shown in Figure 3.8.

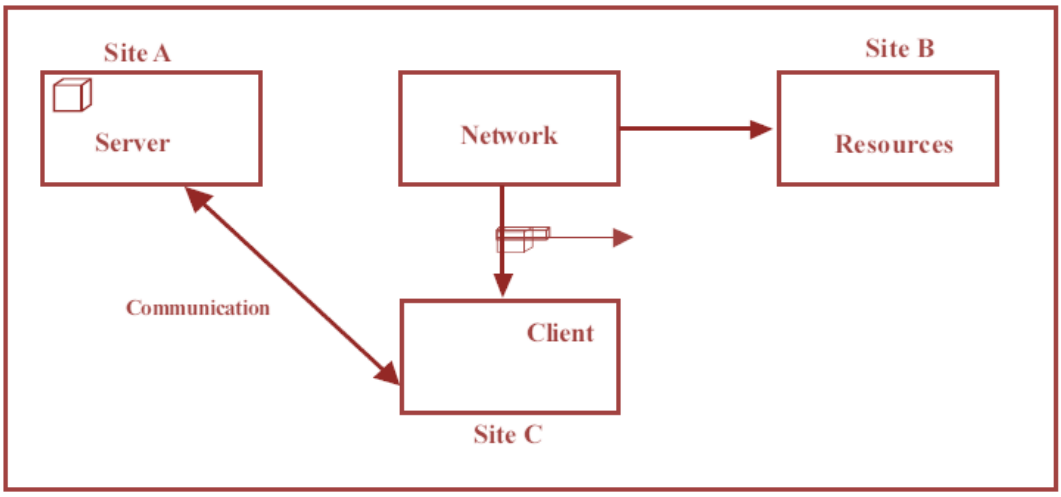


Figure 3.8: A Distributed System

Resource sharing in a Distributed System provides mechanism for sharing files at the remote site, processing information in a distributed database, printing files at the remote sites and performing various other operations. This system allows us to distribute the sub computations among the various sites, which run concurrently, thereby providing computations speedup. Distributed System gives better reliability, that is, if one site fails in this system, the remaining sites can continue operating. As several sites are connected to one another by a communication network, the users at different sites have the opportunity to exchange information's.

The Distributed Systems are widely accepted by the Industry because of better user interface, more flexibility in locating resources and expanding facilities and easier maintenance.

3.3.8 Special Operating System

(1) Embedded Operating System:

Embedded Systems are the most prevalent form of the computer system in existence. These systems tend to have very specific tasks. In these systems there is a little or no user interface,

preferring to spend their time monitoring and managing hardware devices such as automobile engines and robotic arms. These Embedded Systems vary considerably. Some are general purpose computers, running standard operating system, like UNIX, with special purpose application to implement the functionality. Others are hardware devices with a special purpose embedded operating system. Embedded System find its use in controlling heating and lighting, alarm systems, etc of a house through a central computer which may be general purpose computer or an embedded one. Embedded System, almost, always run real time operatingsystems.

(2)Hand Mobile O.S. or Handheld O.S.

This type of system includes personal digital assistants (PDAs), such as Palm and Pocket-PCs and cellular telephones, many of which are using special purpose embedded operating system. Devices using handheld system are small in size due to which they have small amount of memory, slow processor and small display screens. On using virtual memory techniques, the devices making use of handheld system can manage the memory efficiently. Also, most of the device using this system uses small battery due to which speed of the processor is slow. Some handheld devices use wireless technology such as Blue tooth allowing remote access to e-mail and web browsing. Cellular Telephones with connectivity to the internet fall in this category.

The Mobile Operating System or Handheld System controls a mobile device or information appliance similar in principle to an operating system such as Windows, Mac OS X, Linux distributed system that control a desktop computer or a Laptop.

The popular mobile operating systems running on smart phones, PDAs, tablet computer etc. are

(1) Android from Google, which is a Linux derived operating system

(2) Symbian from Nokia which supports multiple user interfaces.

(3) Blackberry operating system from RIM.

(4) Web operating system from HP, initially developed by Palm.

3.4 Summary

In this unit we have trace the evolution of the operating systems. We have discussed the different types of operating system on the basis of their design, process management, file management and memory management. The main objective of all the discussed operating systems was to reduce the CPU idle time and enhances the efficiency of the system. Besides this the operating system provided its user with an interface that is easier to use than the bare machine. Efficiency of an operating system and overall performance of a computer

system are usually measured in terms of its throughput, turnaround time and response time.

Self Assessment Questions

1. How was a job typically executed in early computer system?
2. What are control statements? Why they are needed ?
3. What is Batch Processing?
4. Differentiate between uniprogramming and multiprogramming system.
5. Draw basic organization diagram of multiprocessing system.
6. Define Time Slice.
8. What is Time Sharing?
9. Differentiate between Multiprogramming and Multiprocessing processing.
10. What is Multitasking Processing?
11. Define the essential properties of the following types of operating system:
    (a) Real Time
    (b) Time Sharing
    (c) Distributed
    (d) Handheld
    (e) Multiprogrmming
    (f) Multitasking
    (g) Multiprocessing
    (h) Embedded

Unit 4: ProcessManagement

4.0 Objective

After studying this unit you will be able to define the concept of a process, the process state, control switching and process control block. You will be able to understand how a process is created and terminated. Also, you will go through the basics of inter-process communication and Client -Server System.

4.1 Introduction

In early computer systems only one program was executed at a time. This program had the access to the system's all resources. Today, the computer system allows the multiple programs to run concurrently. This evolution along with the complexity of the operating system requires firmer control and more compartmentalization of various programs. Hence, a separate module of Process Management was developed. This module takes care of creation and termination of processes, scheduling of system resources to different processes requesting them and providing mechanism for synchronization and communication among various processes. We may define a process as a program in execution. In other words, process is a running program with some specific tasks to perform. For example, a word processing program being run by an individual user on computer is a process

4.2 Introduction to Processes

A process is defined as a program in execution performing certain task allotted to it. For example a time sharing user program such as compiler is a process. Shell or command interpreter, in UNIX operating system, is a process, performing the task of listening to whatever is typed on a terminal. A process needs resources, including CPU time, memory files and input/output devices to perform its task. These resources are either given to the process when it is created or allocate to it while running. During creation, various initialization data may be passed. When the process terminates, the operating system will reclaim any reusable resources. In a system, a process is unit of work. As system is collection of processes, including both, user processes and operating system processes, all these processes can be executed concurrently.

A process is an 'active' entity whereas a program is a 'passive' entity. Process, includes current activity along with the contents of the processor's registers. A process, also include, the process stack containing the primary data and a data section containing global variables. Also, a process may contain a memory known as heap which is allocated during process run time. More than one process associated with the same program is considered separate execution sequences. For example, multiple users running different copies of the mail program.

4.3 Process State

The state of a process is defined in part by the current activity of that process. During execution, a process changes its state. Depending on the implementation, the operating systems may differ in the number of states a process goes through. Each process may be in one of the following five states:

01	New State	The process being created
02	Running State	The instructions are being executed.
03	Waiting State	The process is waiting for some event to occur, for instance an I/O completion or reception of a signal. In this state a process is unable to proceed until some external events happen.
04	Ready State	The process is waiting to be assigned to a processor.
05	Terminated State	The process has finished execution.

It should be remembered that at any instance, only one process can be running on any processor. The five state process model is given below in Figure 4.1.

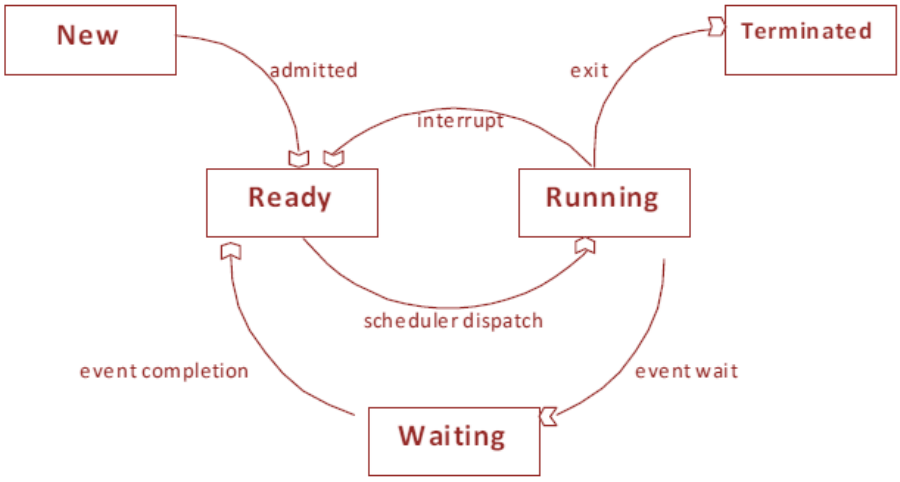


Figure 4.1: Process State Model

4.4 Process Control Block

Process control block is the representation of a process in the operating system. This block, also known as task control block, is collection of process attributes needed by the operating system to control a process. A schematic diagram of the Process Control Block is shown in Figure 4.2.

Process state
Process number
Program counter
Registers
Memory limits
List of open files
● ● ●

Figure 4.2: Process Control Block

The following are the various attributes associated with a specific process in the Process Control Block :

1	**Program Counter**	**It depicts the address of the next instruction to be executed for this process.**
2	Process State	It may be new, waiting ready, running and so on.
3	CPU Registers	They include accumulator, index registers, stack pointers and the general purpose registers.
4	CPU Scheduling information	It includes a process priority, pointers to scheduling queues, and other scheduling parameters.

5	Memory Management	Depending on the memory used by the Information operating system it includes value of the base & limit registers, page table or segment table.
6	Accounting Information	It includes time limit, account numbers, amount of CPU and real time used, and so on.
7	I/O Status Information	It includes list of I/O devices allocated to the process, list of open files, and so on.

4.5 Context Switching

The operating system changes the current task of a CPU if interrupt occurs. This allows the system to schedule all processes in the main memory to run on the CPU at equal intervals. When an interrupts occurs, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its process is done. The context is represented in the Process Control Box of the process. It includes the value of the CPU registers, the process state and the memory management information.

The switching of the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as Context Switching. During context switching, the kernel saves the context of the old process in its PCB and loads the context of the new process scheduled to run.

Let us understand with the help of an example. Suppose if two processes P_0 and P_1 are in ready queue. If CPU is executing Process P_0 and Process P_1 is in wait state. If an interrupt occurs for Process P_0, the operating system suspends the execution of the first process, and stores the current information of Process P_0 in its PCB and switch to the second process namely Process P_1. In doing so, the program counter from the PCB of Process P_1 is loaded, and thus execution can continue with the new process. The switching between two processes, Process P_0 and Process P_1 is illustrated in the Figure 4.3 :

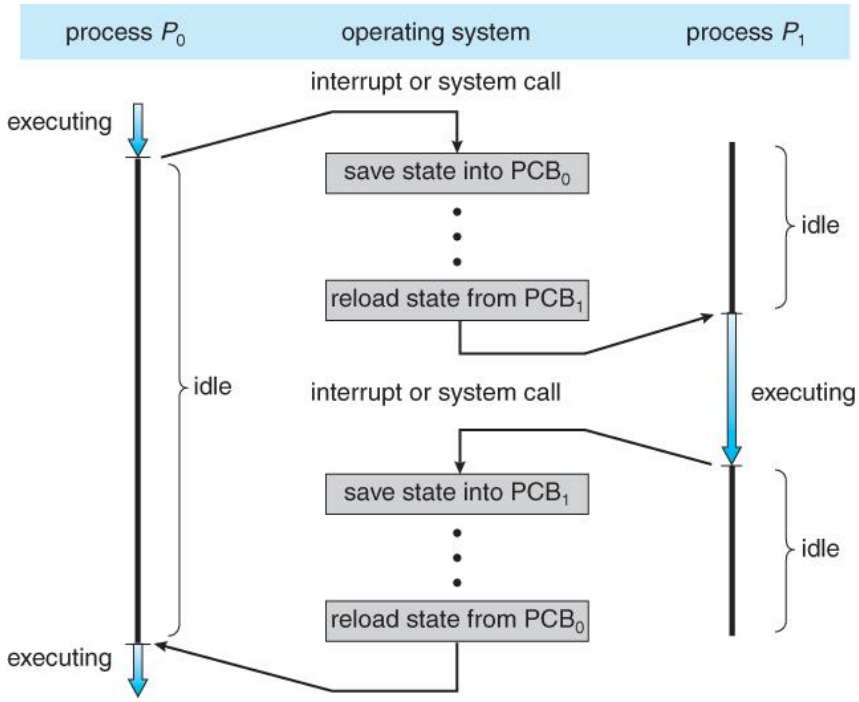


Figure 4.3: Diagram showing Process Switching

The context switching time depends on the memory speed and the number of registers that must be copied. They are dependent on hardware support, also.

4.6 Process Creation and Termination

There are four principal events that cause a process to be created:

- System initialization.
- Execution of process creation system call by running a process.
- A user request to create a new process.
- Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that interact with a (human) user and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming e-mails, sleeping most of the day but suddenly springing to life when an incoming e-mail arrives. Another background process may be designed to accept an incoming request for web pages hosted on the machine, waking up when a request arrives to service that request.

Process creation in UNIX and Linux are done through fork() or clone() system calls. There are several steps involved in process creation. The first step is the validation of whether the

parent process has sufficient authorization to create a process. Upon successful validation, the parent process is copied almost entirely, with changes only to the unique process-id, parent process, and user-space. Each new process gets its own user space.

During the execution, a process may create several new processes. The process which creates several new processes is called Parent process while the new ones are called children of that process. Each of these new processes in turn may create other processes, thereby, forming a Tree of processes. A typical process tree for the Solaris Operating System is illustrated in the Figure 4.4. Processes are identified by a unique Process Identifier (PID), which is an integer number.

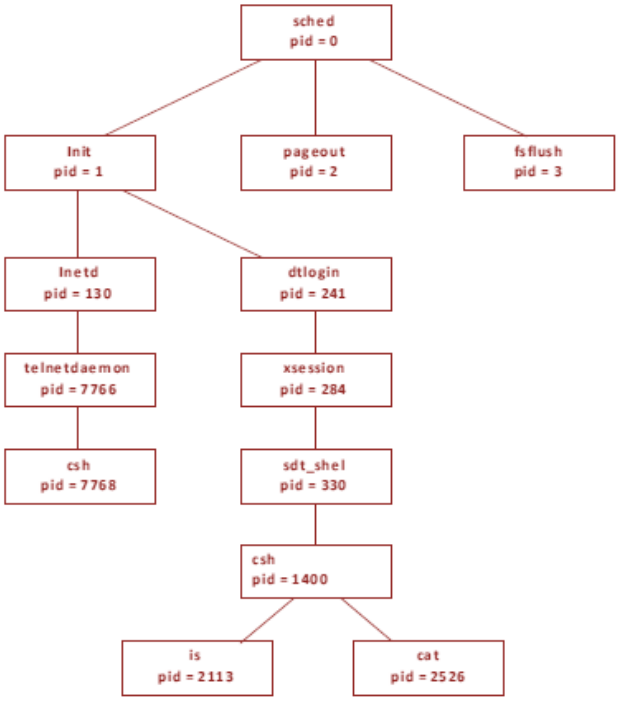


Figure 4.4: A tree of processes on a Solaris system

In Solaris operating system, the process at the top of the tree is called sched process with pid 0. The shed process creates several children processes that are responsible for managing memory and file systems. The child process includes pageout and fsflush. The sched process creates the init process which serves as the root parent process for all user processes. Inetd and dlogin are the two children of init. inetd is responsible for networking services, whereas dlogin is the process representing a user login screen. When a user logs in, dtlogin creates an X-Window session, which in turns creates the sdt_shel process. Below sdt_shel, a user's command line shell - csh is created. It is this command-line interface where the user then invokes various child processes such as the ls and cat commands. csh process with pid of 7768 represent a user who has logged onto the system using telnet.

When a process creates a sub process, that sub process can either obtain its resources directly from the operating system or it may be constrained to a subset of the resources of the parent process. Overloading of the system due to processes is prevented by restricting a child process to a subset of the parent's resources.

When the process finishes executing its final statement and asks the system to delete it by using the exit () system call, a process terminates. At that point the process may return a status value – an integer to its parent process. All the resources of the process, whether physical or virtual memory, open files, and input/ output devices are de-allocated by the operating system. A process can terminate another process through appropriate system call, for example, Terminate Process( ) in Win32. The parent of the process can only invoke a system call to terminate the process. Otherwise, users could arbitrary kill each other's job. When a process creates a new process the identity of the newly created process is passed to the parent. A parent may terminate the execution of one of its children if the child has exceeded its usage of some of its resource or the task assigned to the child is no longer required. There are some systems who do not allow child to exist if its parent has terminated. In such systems, if a process terminates than all its children must also be terminated. This type of termination is called Cascading Termination.

4.7 Basics of Interprocess Communication (IPC)

There are two ways in which the processes execute concurrently – independent processes or cooperating processes. Independent process is the process which cannot affect or be affected by the other processes. Thus independent process does not share its data with any other processes. Cooperating process is the process which can affect or be affected by the other processes. In other words, we can say that cooperating process are those which share their data with other processes. Various salient features of cooperating process are Information sharing, computation speedup, modularity and convenience.

Interprocess Communication Mechanism is generally required by cooperating process as it allows it to exchange information and data. Shared Memory and Message Passing are the two fundamental systems used in the interprocess communication.

In Share Memory System, after establishing the memory region to be shared by the cooperating process, exchange of information takes place by means of reading and writing data to the shared region. For example, the producer-consumer problem, where, a producer process produces information which is consumed by the consumer process.

In Message Passing System, communication takes place by means of messages exchanged between the cooperating processes. For example, a chat program used on the world wide web

could be designed so that chat participants communicate with one another by exchanging messages. Generally, two operations are provided in the message passing model - send (message) and receive (message).

Shared memory allows maximum speed and convenience of communication whereas message passing is useful for exchange of smaller data. In shared memory model, system calls are required to establish the shared- memory region. Once it is established, all accesses are treated as routine memory accesses and no help from kernel is required. In message passing model, the message passing system are implemented using system call, which results in more time consuming task of kernel intervention.

4.8 Basics of Communication in Client-Server Systems

Communication in the client - server systems may use sockets, remote procedure calls (RPCs) and java's remote method invocation (RMI). We will be giving here brief out lines of the three.

Sockets

A socket is defined as an endpoint for communication. A pair of sockets - one for each process, is employed for a pair of processes communicating over a network. A socket is identified by an IP address concatenated with a port number. When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For instance, if a client on the host A with IP address 136.86.5.20 wishes to establish a connection with a web server (which is listening on port 70) at address 151.25.18.4, host A may be assigned port 1525. The connection will consist of a pair of sockets: (136.86.5.20:1525) on the host A and (151.25.18.4:70) on the web server. The information travelling between the hosts are delivered to appropriate process based on the destination port number. Care is to be taken to ensure that all connections consist of a unique pair of sockets.

Remote Procedure Call (RPCs)

RPCs are another form of distributed communication. An RPC occurs when a process calls a procedure on a remote application. Here message based communication scheme is used to provide remote service.

The important characteristics of RPCs are :

- They provide a very familiar interface for the application developer
- Way of implementing the commonplace request-reply primitive
- The format of the messages is standard, not application dependent

- They make it easier to reuse code, since the RPCs have a standard interface and are separate from any one application-proper.

The messages exchanged in RPC communication are well structured. Each message is address to an RPC daemon listening to a port on the remote system and each contains an identifier of the function to execute and the parameters to pass to that function. The execution of the function takes place according to the request. After that, in a separate message, an output is sent back to the requester. A system generally has one network address. It can have many ports to differentiate the many network services it supports. If a remote process needs a service, it sends a message to the proper port. Suppose a system allows the other system to share information about its current user, then it would have a daemon supporting such a RPC attached to a port, say 3125. Any remote system could obtain the needed information by sending an RPC message to the concerned port, that is, 3125 on the server. The desired information would be received in the reply message. The RPC system provides stub on the client side which hides the details that allow communication to take place. Each separate remote procedure has a separate stub. RPC system calls the appropriate stub when a client invokes a remote procedure. This particular stub locates the port on the server and packages the parameters into a form which can be transmitted over a network. After that, the stub transmits a message to the server using message passing. The counter stub on the server side receives this message and invokes the procedure on the server. If required, the return value in the form of output is passed back to the client using same procedure. In the Figure 4.5 there is illustration of Remote Procedure Calls.

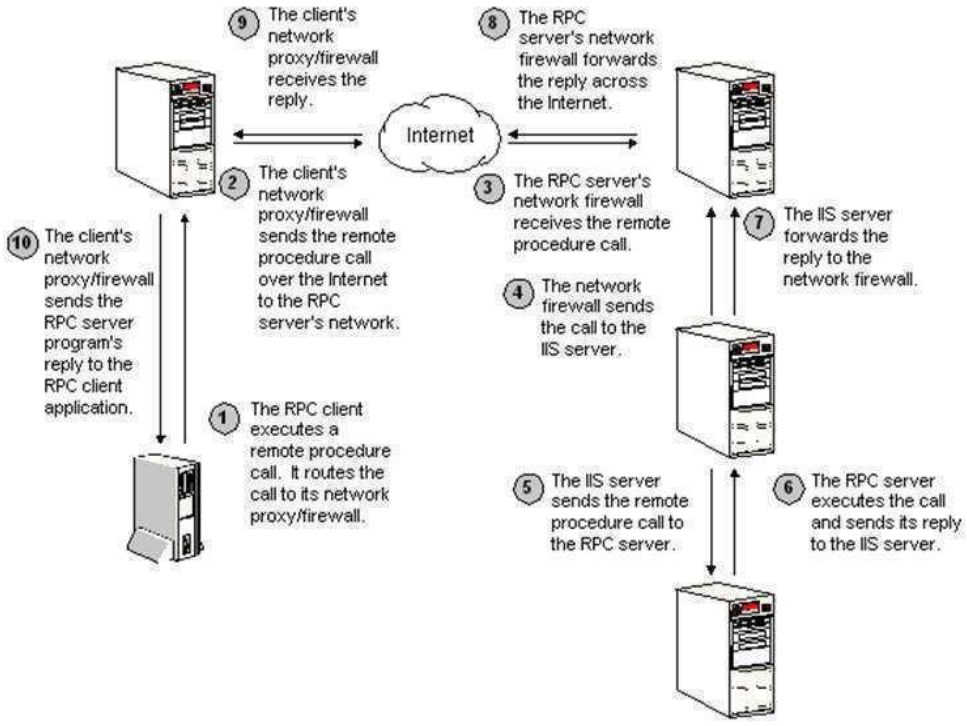


Figure 4.5: Illustration of Remote Procedure Call

It must be ensured that messages are acted on 'exactly once' rather than 'at most once' because if local procedure call fail, RPC can fail or be duplicated and executed more than once as a result of common network errors.

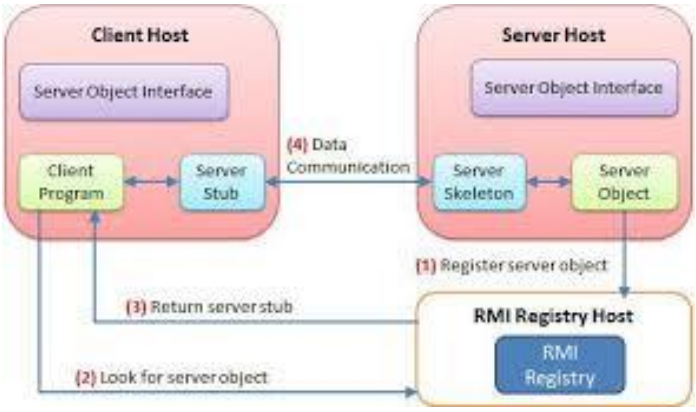
Remote Method Invocation (RMI)

Remote method invocation is java version of remote procedure call. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Re- mote Method Protocol (JRMP). In order to support code running in a non-JVM context, a CORBA version was later developed. RMI allows a thread to invoke a method on a remote object. The objects are considered remote if they reside in a different java virtual machine. Hence, on the same computer or on the remote host connected by network, these objects may reside on different java virtual machine.

Two basic differences between RPC and RMI are:

1. RMI is object based and it supports invocation of methods on remote objects, whereas, RPCs support procedural programming, in which only remote procedures or functions can be called.
2. In RMI, it is possible to pass objects as parameters to remote methods, whereas, in RPCs, the parameters to remote procedures are ordinary data structures

Remote Method Invocation implements the remote object using stubs and skeletons. When a client invokes a remote method, the stub - a proxy for the remote object, on the client side, is called. This particular stub creates a parcel containing the name of the method to be invoked on the server and marshalled the parameters for the method. The stub then sends this parcel to the server where skeleton for the remote object receives it. This skeleton unmarshalled the parameters and invokes the desired method on the server. The skeleton, then marshal the return value into a parcel and return it to the client where the stub on the client side unmarshals the return value and pass it to the client. In the figure 4.6 illustration of Remote Method Invocation is given.



4.9 Summary

Process Management takes care of creation and termination of processes, scheduling of system resources to different processes requesting them and providing mechanism for synchronization and communication among various processes. A process is a program in execution. A process changes its state on execution.

The state of a process is defined by that process's current activity. The various state of a process are running, new, ready, waiting or terminated. Each process is represented in the operating system by its own Process Control Box. The PCBs can be linked together to form a ready queue.

The processes executing in the operating system are of two types - Independent processes and Cooperation processes. Shared Memory and Message Passing are the two fundamental models used in the interprocess communication.

Communication in the client - server systems may use sockets, remote procedure calls (RPCs) and java's remote method invocation (RMI).

Self Assessment Questions

1. Define the following:
    a. Process
    b. Process State
    c. Process Control Box
2. How a process is created and terminated?
3. Briefly discuss the following:
    a. Shared Memory System
    b. Message Passing System
    c. Sockets
    d. Remote Method Invocation (RMI)

e. Remote Procedure Calls (RPCs)

4. Differentiate between independent process and cooperation process.
5. Differentiate between RMI and RPCs.
6. Briefly discuss various process states.
7. escribe the actions taken by the kernel to context switch between processes.
8. Explain the major components of a process

Unit 5: Theards

5.0 Objective

After reading this unit, you will be able to understand general overview of thread, which is the fundamental unit of CPU utilization, different thread libraries and issues related to multithreaded programming.

5.1 Introduction

Traditionally, programs are single-path execution, hence a single thread. This practice would have made the production of today's software production impossible as the need of speed required programs to perform multiple tasks and events at the same time. With traditional turn-by-turn game, such as tic-tac-toe or chess, the traditional approach works fine, however with new age multitasking programs where multiple events need to run in parallel, the traditional approach proves useless.

5.2 Threads

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated). Since thread will generally call different procedures and thus will have a different execution history. This is why thread needs its own stack.

A thread has a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes and as a result threads share with other their code section, data section, OS resources also known as task, such as open files and signals with other threads.

5.3 Processes Vs Threads

Processes are used to group resources together and threads are the entities scheduled for execution on the CPU. We mentioned earlier that in many respect threads operate in the same way as processes. Some of the similarities and differences are:

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a processes, is execute sequentially.
- Like processes, thread can create children.
- Like processes, if one thread is blocked, another thread can run.

Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task .
- Unlike processes, thread is design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Why Threads?

Threads are cheap because

1. They only need a stack and storage for registers therefore, threads are cheap to create.
2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
3. Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

Biggest drawback of threads is that there is no protection between threads.

5.3.1 User-Level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact,

the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Advantages: The most obvious advantage of this technique is that user-level threads package can be implemented on an operating system that does not support threads. Some other advantages are:

- User-level threads do not require modification to operating systems.
- Simple Representation : Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient: Thread switching is not much expensive than a procedure call.

Disadvantages:

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads require non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

5.3.2 Kernel-Level Threads

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating system kernel provides system call to create and manage threads.

Advantages:

- Because kernel has full knowledge of all threads, scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient.
- Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increase in kernel complexity.

Advantages of Threads over Multiple Processes

- **Context Switching** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general- purpose registers, but they do not require space to share memory information, information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.
- **Sharing** Threads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, operating system resources like open files etc.

Disadvantages of Threads over Multiprocesses

- **Blocking** The major disadvantage is that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.
- **Security** Since there is, an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

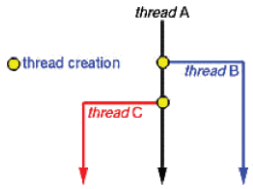
Application that benefits from Threads

A proxy server satisfying the requests for a number of computers on a LAN would be benefited by a multi- threaded process. In general, any program that has to do more than one task at a time could benefit from multitasking. For example, a program that reads input, process it, and outputs could have three threads, one for each task.

Application that cannot Benefit from Threads

Any sequential process that cannot be divided into parallel task will not benefit from thread, as they would block until the previous one completes. For example, a program that displays the time of the day would not benefit from multiple threads.

Resources used in Thread Creation and Process Creation



When a new thread is created it shares its code section, data section and operating system resources like open files with other threads. But it is allocated its own stack, register set and a program counter.

The creation of a new process differs from that of a thread mainly in the fact that all the shared resources of a thread are needed explicitly for each process. So though two processes may be running the same piece of code they need to have their own copy of the code in the main memory to be able to run. Two processes also do not share other resources with each other. This makes the creation of a new process very costly compared to that of a new thread.

Context Switch

To give each process on a multiprogrammed machine a fair share of the CPU, a hardware clock generates interrupts periodically. This allows the operating system to schedule all processes in main memory (using scheduling algorithm) to run on the CPU at equal intervals. Each time a clock interrupt occurs, the interrupt handler checks how much time the current running process has used. If it has used up its entire time slice, then the CPU scheduling algorithm (in kernel) picks a different process to run. Each switch of the CPU from one process to another is called a context switch.

Major Steps of Context Switching

- The values of the CPU registers are saved in the process table of the process that was running just before the clock interrupt occurred.
- The registers are loaded from the process picked by the CPU scheduler to run next.

In a multiprogrammed uniprocessor computing system, context switches occur frequently enough that all processes appear to be running concurrently. If a process has more than one thread, the Operating System can use the context switching technique to schedule the threads so they appear to execute in parallel. This is the case if threads are implemented at the kernel level. Threads can also be implemented entirely at the user level in run-time libraries. Since in this case no thread scheduling is provided by the Operating System, it is the responsibility of the

programmer to yield the CPU frequently enough in each thread so all threads in the process can make progress.

Action of Kernel to Context Switch among Threads

The threads share a lot of resources with other peer threads belonging to the same process. So, a context switch among threads for the same process is easy. It involves switch of register set, the program counter and the stack. It is relatively easy for the kernel to accomplished this task.

Action of Kernel to Context Switch among Processes

Context switches among processes are expensive. Before a process can be switched its process control block (PCB) must be saved by the operating system. The PCB consists of the following information:

- The process state.
- The program counter, PC.
- The values of the different registers.
- The CPU scheduling information for the process.
- Memory management information regarding the process.
- Possible accounting information for this process.
- I/O status information of the process.

When the PCB of the currently executing process is saved, the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time. Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.

- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a

port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)

Benefits

There are four major categories of benefits to multi-threading:

1. Responsiveness - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
2. Resource sharing - By default; threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. Economy - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
4. Scalability, i.e. utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure (5.1). On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in Figure (5.2).

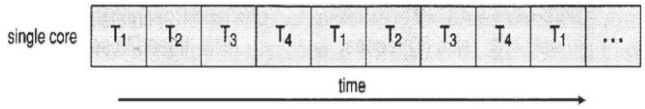


Figure 5.1: Concurrent Execution on a single core system

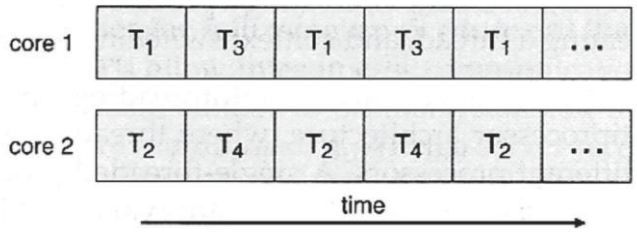


Figure 5.2 : Concurrent Execution on a multicore system

- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- For application programmers, there are five areas where multi-core chips present new challenges:

1. Dividing activities - Examining applications to find activities that can be performed concurrently.
2. Balance - Finding tasks to run concurrently that provide equal value, i.e. don't waste a thread on trivial tasks.
3. Data splitting - To prevent the threads from interfering with one another.
4. Data dependency - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
5. Testing and debugging - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

5.4 Multi Threading Models

- There are two types of threads to be managed in a modern system: user threads and kernel threads.
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

5.4.1 Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one

model does not allow individual processes to be split across multiple CPUs.

- Green threads for Solaris and GNU Portable Threads implement the many-to-one model.

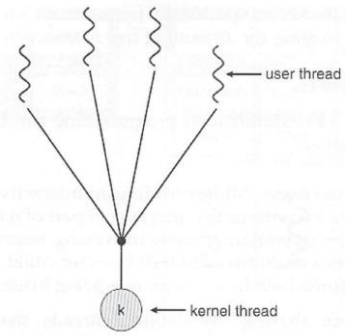


Figure 5.3: Many-To-One Model

5.4.2 One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

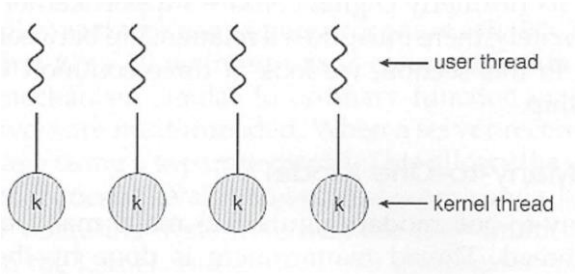


Figure 5.4: One-To-One Model

5.4.3 Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller num ber of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.

- Individual processes may be allocated variable numbers of kernel threads, depending on the num ber of CPUs present and other factors.

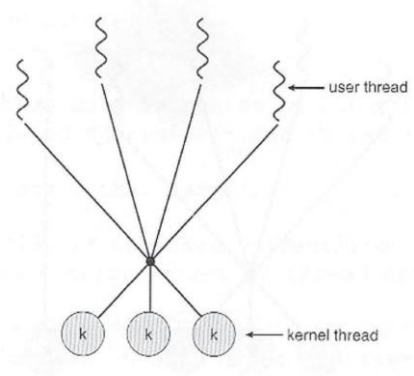


Figure 5.5 : Many-To-Many Model

- One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation.
- IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9.

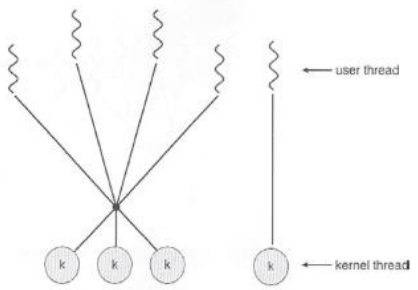


Figure 5.6 : Two Level Model

5.5 Thread Libraries

Thread libraries provide programmers with an API for creating and managing threads. Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls and requires a kernel with thread library support. There are three main thread libraries in use today:

1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
2. Windows 2000 threads - provided as a kernel-level library on Windows systems.
3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

- The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

5.5.1 Pthreads

- The POSIX standard (IEEE 1003.1c) defines the *specification* for pThreads, not the *implementation*.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function:

5.5.2 Java Threads

- All Java programs use threads - even "common" single-threaded ones.
- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the runnable interface
- The creation of new threads requires objects that implement the runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
- Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)

Because Java does not support global variables, threads must pass a reference to a shared object in order to share data, in this example the "Sum" Object.

Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or any of the other

models discussed previously.

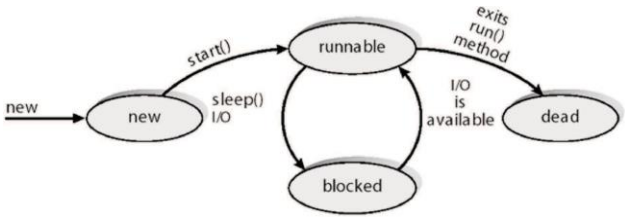


Fig. 5.7 Operating System Examples

5.5.3 Windows 2000 Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

5.5.4 Linux Threads

- Linux does not distinguish between processes and threads - It uses the more generic term "tasks".
- The traditional fork( ) system call completely duplicates a process (task).

An alternative system call, clone( ) allows for varying degrees of sharing between the parent and child tasks, controlled by flags such as those shown in the following table:

flag	Meaning
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGHAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

- Calling clone( )with no flags set is equivalent to fork( ). Calling clone( ) with CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES is equivalent to creating a thread, as all of these data structures will be shared.

- Linux implements this using a structure task_struct, which essentially provides a level of indirection to task resources. When the flags are not set, then the resources pointed to by the structure are copied, but if the flags are set, then only the pointers to the resources are copied, and hence the resources are shared.
- clone() allows a child task to share the address space of the parent task (process)
- Several distributions of Linux now support the NPTL (Native POSIX Thread Library)
- POSIX compliant.
- Support for SMP (symmetric multiprocessing), NUMA (non-uniform memory access), and multicore processors.
- Support for hundreds to thousands of threads.

5.6 Thread Issues

5.6.1 The fork( ) and exec( ) System Calls

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?
- A: System dependant.
- A: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.
- A: Many versions of UNIX provide multiple versions of the fork call for this purpose.

5.6.2 Cancellation

- Threads that are no longer needed may be cancelled by another thread in one of two ways:
  - **Asynchronous Cancellation** cancels the thread immediately.
  - **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

5.6.3 Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?
- A: There are four major options:
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.

- Deliver the signal to certain threads in the process.
- Assign a specific thread to receive all signals in a process.

- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However, the signal can only be delivered to one thread, which is generally the first thread that is accepting that particular signal.
- Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

5.6.4 Thread Pools

- Creating new threads every time one is needed and then deleting it when it is done can be inefficient and can also lead to a very large (unlimited) number of threads being created.
- An alternative solution is to create a number of threads when the process first starts and put those threads into a ***thread pool***.
- Threads are allocated from the pool as needed and returned to the pool when no longer needed.
- When no threads are available in the pool, the process may have to wait until one becomes available.
- The (maximum) number of threads available in a thread pool may be determined by adjustable parameters, possibly dynamically in response to changing system loads.
- Win32 provides thread pools through the "Pool Function" function. Java also provides support for thread pools.

5.6.5 Thread-Specific Data

- Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
- However sometimes threads need thread-specific data also.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data.

5.6.6 Scheduler Activations

- Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier

models.

- This virtual processor is known as a "Lightweight Process", LWP.
- There is a one-to-one correspondence between LWPs and kernel threads.
- The number of kernel threads available, (and hence the number of LWPs) may change dynamically.
- The application (user level thread library) maps user threads onto available LWPs.
- kernel threads are scheduled onto the real processor(s) by the OS.
- The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an ***upcall***, which is handled in the thread library by an ***upcall handler***. The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.
- If the kernel thread blocks, then the LWP blocks, which blocks the user thread.
- Ideally there should be at least as many LWPs available as there could be concurrently blocked kernel threads. Otherwise if all LWPs are blocked, then user threads will have to wait for one to become available.

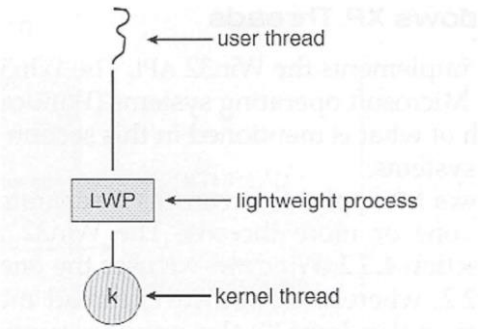


Figure 5.8: Light Weight Process (LWP)

5.7 Thread Scheduling

Scheduling Threads

Threads can be scheduled, and the threads library provides several facilities to handle and control the scheduling of threads. It also provides facilities to control the scheduling of threads during synchronization operations such as locking a mutex. Each thread has its own set of scheduling parameters. These parameters can be set using the thread attributes object before the thread is created. The parameters can also be dynamically set during the thread's execution. Controlling the scheduling of a thread can be a complicated task. Because the scheduler handles

all threads systemwide, the scheduling parameters of a thread interact with those of all other threads in the process and in the other processes. The following facilities are the first to be used if you want to control the scheduling of a thread.

The threads library allows the programmer to control the execution scheduling of the threads in the following ways:

- By setting scheduling attributes when creating a thread
- By dynamically changing the scheduling attributes of a created thread
- By defining the effect of a mutex on the thread's scheduling when creating a mutex (known as synchronization scheduling)
- By dynamically changing the scheduling of a thread during synchronization operations (known as synchronization scheduling)

Scheduling Parameters

A thread has the following scheduling parameters:

Scope The contention scope of a thread is defined by the thread model used in the threads library.

Policy The scheduling policy of a thread defines how the scheduler treats the thread after it gains control of the CPU.

Priority The scheduling priority of a thread defines the relative importance of the work being done by each thread.

The scheduling parameters can be set before the thread's creation or during the thread's execution. In general, controlling the scheduling parameters of threads is important only for threads that are CPU- intensive. Thus, the threads library provides default values that are sufficient for most cases.

Contention Scope and Concurrency Level

The *contention scope* of a user thread defines how it is mapped to a kernel thread. The threads library defines the following contention scopes:

PTHREAD_SCOPE_PROCESS	Process contention scope, sometimes called local contention scope. Specifies that the thread will be scheduled against all other local contention scope threads in the process. A process-contention-scope user thread is a user thread that shares a kernel thread with other process-contention-scope user threads in the process. All user threads in an M:1 thread model have process contention scope.

PTHREAD_SCOPE_SYSTEM	System contention scope, sometimes called global contention scope. Specifies that the thread will be scheduled against all other threads in the system and is directly mapped to one kernel thread. All user threads in a 1:1 thread model have system contention scope.

In an M:N thread model, user threads can have either system or process contention scope. Therefore, an M:N thread model is often referred as a mixed-scope model.

The concurrency level is a property of M:N threads libraries. It defines the number of virtual processors used to run the process-contention scope user threads. This number cannot exceed the number of process-contention-scope user threads and is usually dynamically set by the threads library. The system also sets a limit to the number of available kernel threads.

Setting the Contention Scope

The contention scope can only be set before a thread is created by setting the contention-scope attribute of a thread attributes object. The **pthread_attr_setscope** subroutine sets the value of the attribute; the **pthread_attr_getscope** returns it.

The contention scope is only meaningful in a mixed-scope M:N library implementation. A

TestImplementation routine could be written as follows:

```
int TestImplementation()
{
    pthre
    ad_at
    tr_t
    a; int
    result
    ;
    pthread_attr_init(&a);
    switch (pthread_attr_setscope(&a, PTHREAD_SCOPE_PROCESS))
     {
        case 0:        result =
        LIB_MN; break; case
        ENOTSUP: result = LIB_11;
        break;
```

```
            case ENOSYS:     result =
            NO_PRIO_OPTION; break; default:
                        result = ERROR; break;
        }
      pthread_attr_d
      estroy(&a);
      return result;
}
```

Impacts of Contention Scope on Scheduling

The contention scope of a thread influences its scheduling. Each contention-scope thread is bound to one kernel thread. Thus, changing the scheduling policy and priority of a global user thread results in changing the scheduling policy and priority of the underlying kernel thread.

In AIX®, only kernel threads with root authority can use a fixed-priority scheduling policy (FIFO or round-robin). The following code will always return the **EPERM** error code if the calling thread has system contention scope but does not have root authority. This code would not fail, if the calling thread had process contention scope.

```
schedparam.sched_priority = 3;
pthread_setschedparam(pthread_self(), SCHED_FIFO,
schedparam);
```

Using the inheritsched Attribute

The **inheritsched** attribute of the thread attributes object specifies how the thread's scheduling attributes will be defined. The following values are valid:

PTHREAD_INHERIT_SCHED	Specifies that the new thread will get the scheduling attributes (schedpolicy and schedparam attributes) of its creating thread. Scheduling attributes defined in the attributes object are ignored.
PTHREAD_EXPLICIT_SCHED	Specifies that the new thread will get the scheduling attributes defined in this attributes object.

The default value of the **inheritsched** attribute is PTHREAD_INHERIT_SCHED. The attribute is set by calling the **pthread_attr_setinheritsched** subroutine. The current value of the attribute is returned by calling the **pthread_attr_getinheritsched** subroutine.

To set the scheduling attributes of a thread in the thread attributes object, the **inheritsched** attribute must first be set to PTHREAD_EXPLICIT_SCHED. Otherwise, the attributes-object scheduling attributes are ignored.

Scheduling Policy and Priority

The threads library provides the following scheduling policies:

SCHED_FIFO	First-in first-out (FIFO) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run to completion in FIFO order.
SCHED_RR	Round-robin (RR) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run for a fixed time slice in FIFO order.
SCHED_OTHER	Default AIX® scheduling. Each thread has an initial priority that is dynamically modified by the scheduler, according to the thread's activity; thread execution is time-sliced. On other systems, this scheduling policy may be different.

The threads library handles the priority through a sched_param structure, defined in the sys/ sched.h header file. This structure contains the following fields:

sched_priority	Specifies the priority.
sched_policy	This field is ignored by the threads library. Do not use.

Setting the Scheduling Attributes at Execution Time

The **pthread_getschedparam** subroutine returns the **schedpolicy** and **schedparam** attributes of a thread. These attributes can be set by calling the **pthread_setschedparam** subroutine. If the target thread is currently running on a processor, the new scheduling policy and priority will be implemented the next time the thread is scheduled. If the target thread is not running, it can be scheduled immediately at the end of the subroutine call.

For example, consider a thread T that is currently running with round-robin policy at the moment the **schedpolicy** attribute of T is changed to FIFO. T will run until the end of its time slice, at which time its scheduling attributes are then re-evaluated. If no threads have higher priority, T will be rescheduled, even before other threads having the same priority. Consider a second example where a low-priority thread is not running. If this thread's priority is raised by another thread calling the **pthread_setschedparam** subroutine, the target thread will be scheduled immediately if it is the highest priority runnable thread.

Note: Both subroutines use a *policy* parameter and a **sched_param** structure. Although this structure contains a sched_policy field, programs should not use it. The subroutines use the *policy* parameter to pass the scheduling policy, and the subroutines then ignore the sched_policy field.

Scheduling-Policy Considerations

Applications should use the default scheduling policy, unless a specific application requires the use of a fixed-priority scheduling policy. Consider the following points about using the nondefault policies:

- Using the round-robin policy ensures that all threads having the same priority level will be scheduled equally, regardless of their activity. This can be useful in programs where threads must read sensors or write actuators.
- Using the FIFO policy should be done with great care. A thread running with FIFO policy runs to completion, unless it is blocked by some calls, such as performing input and output operations. Ahigh-priority FIFO thread may not be preempted and can affect the global performance of the system. For example, threads doing intensive calculations, such as inverting a large matrix, should never run with FIFO policy.

The setting of scheduling policy and priority is also influenced by the contention scope of threads. Using the FIFO or the round-robin policy may not always be allowed.

sched_yield Subroutine

The **sched_yield** subroutine is the equivalent for threads of the **yield** subroutine. The **sched_yield** subroutine forces the calling thread to relinquish the use of its processor and gives other threads an opportunity to be scheduled. The next scheduled thread may belong to the same process as the calling thread or to another process. Do not use the **yield** subroutine in a multi-threaded program.

The interface **pthread_yield** subroutine is not available in Single UNIX Specification, Version 2.

5.8 Summary

Threads are an inherit part of software products as a fundamental unit of CPU utilization as a basic building block of multithreaded systems. The use of threads has evolved over the years from each program consisting of a single thread as the path of execution of it. The notion of multithreading is the expansion of the original application thread to multiple threads running in parallel handling multiple events and performing multiple tasks concurrently. Today's modern operating systems foster the ability of multiple threads controlled by a single process all within the same address space.

Multithreading brings a higher level of responsiveness to the user as a thread can run while other threads are on hold awaiting instructions. As all threads are contained within a parent process, they share the resources and memory allocated to the process working

within the same address space making it less costly to generate multiple threads vs. Processes. These benefits increase even further when executed on a multiprocessor architechture as multiple threads can run in parallel across multiple processors as only one process may execute on one processor.

Threads divide into two types: **user-level threads** – visible to developers but unknown to the kernel, and **kernel-level threads** – managed by the operating system's kernel.

Three models identify the relationships between user-level and kernel-level threads: **one-to-one, many- to-one,** and **many-to-many**.

This chapter explores various notions related to systems with multithreading capability, including POSIX, Java, Windows 2000, Linux thread libraries. Challenges associated to multithreaded program development explored in this report are those of thread cancellation, signal handling, thread-specific data, and semantics of necessary system calls.

Self Assessment Questions

1. List two reasons why a context switch between threads may be faster than a context switch between processes.
2. List three resources that are typically shared by all of the threads of a process.
3. What Components of a program state are shared across threads in a multithreaded process?
4. List differences between user level threads and kernel level threads.
5. What resources are used when a thread is created?
6. Can a multithreaded system using many user-level threads achieve better performance on a multi processor system than on a single processor system?

Unit 6: Process Scheduling: Basic Concept

6.0 Objective

This chapter provides a general overview of CPU Scheduling, which is the basis for multiprogrammed operating system and describes various concepts related to scheduling and different CPU scheduling algorithms.

6.1 Introduction

Scheduling is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance a system effectively or achieve a target quality of service.

The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously).

The scheduler is concerned mainly with:

- Throughput - number of processes that complete their execution per time unit.
- Latency, specifically:
  - Turnaround - total time between submission of a process and its completion.
  - Response time - amount of time it takes from when a request was submitted until the first response is produced.
- Fairness / Waiting Time - Equal CPU time to each process (or more generally appropriate times according to each process' priority).

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above mentioned concerns depending upon the user's needs and objectives.

6.2 Types of schedulers

Operating systems may feature up to 3 distinct types of schedulers, a *long-term scheduler* (high-level scheduler), a *mid-term or medium-term scheduler* and a *short-term scheduler*. The names suggest the relative frequency with which these functions are performed. The Scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run.

Long-term scheduling

The long-term scheduler decides which jobs or processes are to be admitted to the ready queue (in the Main Memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e., whether a high or low amount of processes are to be executed concurrently, and how the split between IO intensive and CPU intensive processes is to be handled.

Medium-term scheduling

The medium-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.

In many systems today, the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand.

Short-term scheduling

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-

memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision willat a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU. In most cases short-term scheduler is written in assembler because it is a critical part of the operating system.

6.3 CPU-I/O Burst Cycle

A process will run for a while (the CPU burst), perform some I/O (the I/O burst), then run for a while more (the next CPU burst). How long between I/O operations, depends on the process.

I/O Bound processes are the processes that perform lots of I/O operations. Each I/O operation is followed by a short CPU burst to process the I/O, then more I/O happens.

CPU bound processes are the processes that perform lots of computation and do little I/O. They tend to have a few long CPU bursts.

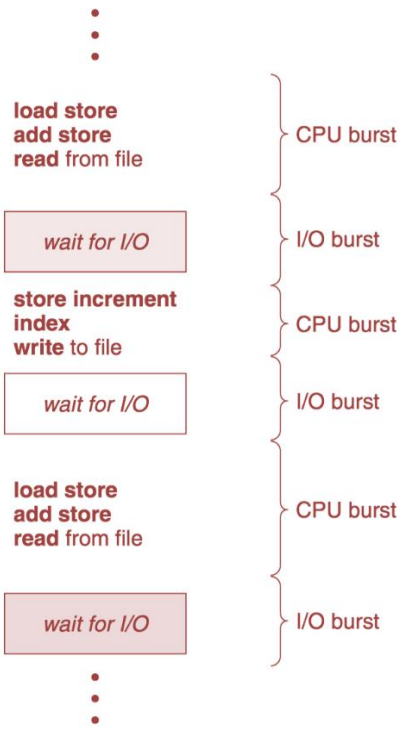


Figure 6.1: Process execution consists of a *cycle* of CPU execution and I/O wait

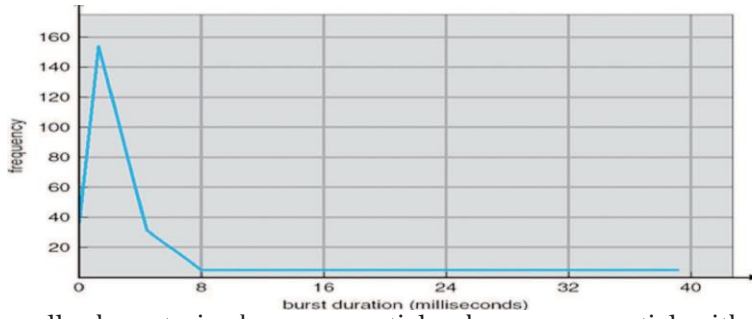
- The success of CPU scheduling depends on an observed property of processes:

- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.

➢ Eventually, the final CPU burst ends with a system request to terminate execution (see the Fig).

The durations of CPU bursts have been measured extensively. They tend to have a frequency curve as in the figure 6.2.

Figure 6.2 : Frequency Curve



The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.

- ➢ An I/O-bound program typically has many short CPU bursts.
- ➢ A CPU-bound program might have a few long CPU bursts.

This distribution can be important in the selection of an appropriate CPU-scheduling algorithm. Nearly all processes alternate bursts of computing with (disk) I/O requests, as shown in Figure 6.3.

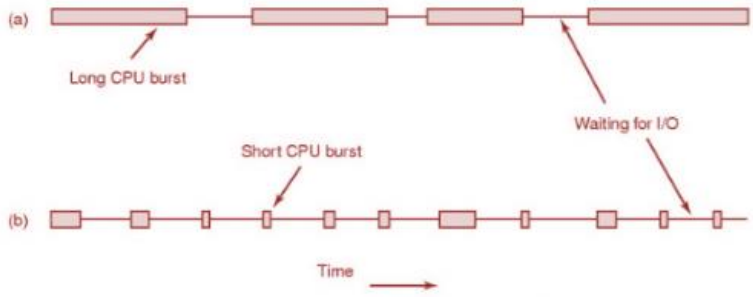


Figure 6.3: I/O Bursts

➢ Some processes, such as the one in Fig . (a), spend most of their time computing (CPU-bound), while others, such as the one in Fig. (b), spend most of their time waiting for I/O (I/O-bound).

➢ Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and then when they are finished loading and running all the I/O-bound jobs (a careful mix of

processes).

Pre-emptive/Non Pre-emptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O, on a semaphore, or for some other reason).
4. When a process terminates. If no process is ready, a system-supplied idle process is normally run. For situations 1 and 4, there is no choice in terms of scheduling. Anew process (if one exists in the ready queue) must be selected for execution.

There is a choice, however, for situations 2 and 3. When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is pre-emptive.

- Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- Unfortunately, pre-emptive scheduling incurs a cost associated with access to shared data.
    - Consider the case of two processes that share data.
    - While one is updating the data, it is preempted so that the second process can run.
    - The second process then tries to read the data, which are in an inconsistent state.
    - In such situations, we need new mechanisms to coordinate access to shared data.
- A non-preemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. First-Come-First-Served (FCFS), Shortest Job first (SJF).
- In contrast, a pre-emptive scheduling algorithm picks a process and lets it run for a

maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run. Round-Robin (RR), Priority Scheduling.

Dispatcher

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves thefollowing:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

6.4 Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In selection of any algorithm to use in a particular situation, we must consider following criterias of the various algorithms.

1. **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
2. **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
3. **Turnaround time**. The interval from the time of submission of a process to the time of completion is the turnaround time.

   Turnaround time (T_r) = T_s + T_w where T_s : Execution time. T_w : Waiting time.
4. **Waiting time**. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
5. **Response time.** In an interactive system, turnaround time may not be the best

criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced.

A typical scheduler is designed to select one or more primary performance criteria and rank them in order of importance. One problem in selecting a set of performance criteria is that they often conflict with each other. For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time decreases.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

A scheduling algorithm that maximizes throughput may not necessarily minimize turnaround time. Given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs. If short jobs kept arriving at a steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.

6.5 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different algorithms for scheduling CPU. Following are some of them:

6.5.1 First-Come, First-Served(FCFS) Scheduling

By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this algorithm, processes are assigned the CPU in the order they request it. Basically, there is a single queue of ready processes. Relative importance of jobs measured only by arrival time (poor choice). The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue.

The average waiting time under the FCFS policy, however, is often quite long. Consider the

following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	*Burst Time(ms)*	*Waiting Time(ms)*	*Turnaround Time(ms)*
$\mathbf{P_1}$	*24*	*0*	*24*
$\mathbf{P_2}$	*3*	*24*	*27*
$\mathbf{P_3}$	*3*	*27*	*30*
Average	-	*17*	*27*

Table 6.1

If the processes arrive in the order P_1,P_2,P_3 and are served in FCFS order, we get the result as

The average waiting time is (0 + 24 + 27) / 3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, the results will be

Waiting time for P2 = 0ms Waiting time for P3 = 3ms Waiting time for P1 = 6ms

So the average waiting time is now (6 + 0 + 3) / 3 = 3 milliseconds.

This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system (dynamic system), the following scenario may result.

- The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/0 and will move into the ready queue, waiting for the CPU.
- While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/0 device.
- All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues.
- At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU.

- Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done.
- There is a convoy effect as all the other processes wait for the one big process to get off the CPU. A long CPU-bound job may take the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods.

This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Another Example: Suppose that there is one CPU-bound process that runs for 1 sec at a time and many I/ O-bound processes that use little CPU time but each have to perform 1000 disk reads to complete.

- The CPU-bound process runs for 1 sec, then it reads a disk block.
- All the I/O processes now run and start disk reads.
- When the CPU-bound process gets its disk block, it runs for another 1 sec, followed by all the I/ O-bound processes in quick succession.
- The net result is that each I/O-bound process gets to read 1 block per second and will take 1000 sec to finish.
- With a scheduling algorithm that preempted the CPU-bound process every 10 msec, the I/O- bound processes would finish in 10 sec instead of 1000 sec, and without slowing down the CPU- bound process very much.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0.

6.5.2 Shortest-Job-First Scheduling

- A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst

Process	*Burst Time(ms)*	*Waiting Time(ms)*	*Turnaround Time(ms)*
P_1	*6*	*3*	*9*
P_2	*8*	*16*	*24*
P_3	*7*	*9*	*16*
P_4	*3*	*0*	*3*
Average	-	*7*	*13*

Table 6.2

Using SJF scheduling, we would schedule these processes according to the following Figure:

By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm gives the minimum average waiting time for a given set of processes.

- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the average waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.

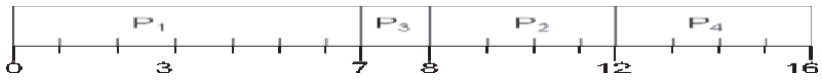
We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones.

Also, long running jobs may starve for the CPU when there is a steady supply of short jobs.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	**0.0**	**7**
P_2	**2.0**	**4**
P_3	**4.0**	**1**
P_4	**5.0**	**4**

Table 6.3



Waiting time for P1 = 0ms

Waiting time for P2 = (8 – 2)= 6ms

Waiting time for P3 = (7 - 4)= 3ms

Waiting time for P4 = (12 - 5)= 7ms

So average waiting time of above scenario is (0 + 6 + 3 + 7) / 4 = 4

For FCFS scheduling scheme the Average waiting time is (0 + 5 + 7 + 7)/4 = 4.75

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Table 6.4

Waiting time for P1 = (11 – 2)= 9ms

Waiting time for P2 = (5 - 4)= 6ms

Waiting time for P3 = 0ms

Waiting time for P4 = (7 - 5)= 2ms

Average waiting time of above scenario is (9 + 1 + 0 + 2) / 4 = 3

In the case of preemptive SJF which is also known as Shortest-Remaining-Time-First (SRTF), there is less average waiting time, because a long job is pre-empted whenever small job arrives in the system. After two minutes when Job P_2 arrives, resources are pre-empted by process P_1 .In similar manner all the longer processes are pre-empted to execute shorter jobs first.

6.5.3 Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but pre-emption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10

to 100 milliseconds. The ready queue is treated as a circular queue.

To implement RR scheduling,

- We keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1time quantum, and dispatches the process.
- The process may have a CPU burst of less than 1 time quantum.
    - In this case, the process itself will release the CPU voluntarily.
    - The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum,
    - the timer will go off and will cause an interrupt to the OS.
    - A context switch will be executed, and the process will be put at the tail of the ready queue.

The CPU scheduler will then select the next process in the ready queue.

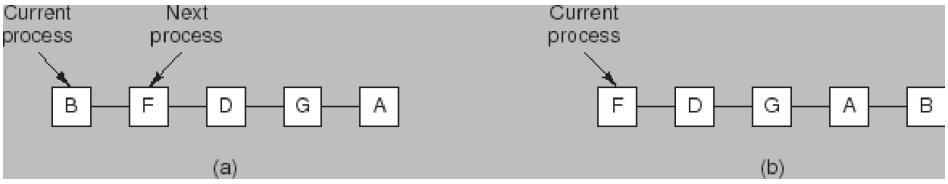


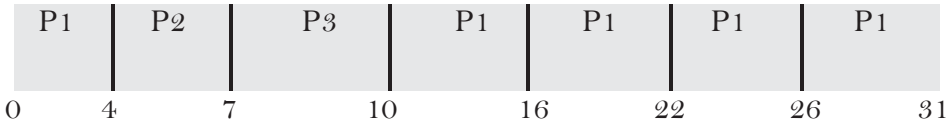
Figure 6.4 : Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: (a time quantum of 4 milliseconds)

Process	Burst Time	Waiting Time	Turnaround Time
P_1	2 4	6	30
P_2	3	4	7
P_3	3	7	10
Average	-	5.66	15 .6 6

Table 6.5

Using round-robin scheduling, we would schedule these processes according to the following figure:



In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus pre-emptive.

- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units.
- Each process must wait no longer than $(n-1) * q$ time units until its next time quantum.
- For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum.

- If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at 1 / n speed of the real processor.

We need also to consider the effect of context switching on the performance of RR scheduling. Switching from one process to another requires a certain amount of time for doing the administration' saving and loading registers and memory maps, updating various tables and lists, flushing and reloading the memory cache, etc.

- Let us assume that we have only one process of 10 time units.
- If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly
    - Thus, we want the time quantum to be large with respect to the context-switch time.
- If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
- In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
- The time required for a context switch is typically less than 10 microseconds; thus, the

context- switch time is a small fraction of the time quantum.

- Setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.
- Poor average waiting time when job lengths are identical; Imagine 10 jobs each requiring 10 time slices, all complete after about 100 time slices, even FCFS is better!
- In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. If context-switch time is added in, the average turnaround time increases for a smaller time quantum, since more context switches are re quired.

Although the time quantum should be large compared with the context-switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to FCFS policy.

Scheduling algorithm	***CPU Overhead***	***Throughput***	***Turnaround time***	***Response time***
First In First Out	*Low*	*Low*	*High*	*Low*
Shortest Job First	*Medium*	*High*	*Medium*	*Medium*
Priority based scheduling	*Medium*	*Low*	*High*	*High*
Round-robin scheduling	*High*	*Medium*	*Medium*	*High*

Table 6.6

6.6 Operating System Examples

6.6.1 Linux Scheduling

- The Linux scheduler is a pre-emptive, priority-based algorithm with two separate priority ranges:
  - a real-time range from 0 to 99
  - and a nice value ranging from 100 to 140.
- These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.
- Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta.

  A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered

expired and is not eligible for execution again until all other tasks have also exhausted their time quanta.

- The kernel maintains a list of all runnable tasks in a run queue data structure. Because of its support for SMP, each processor maintains its own run queue and schedules itself independently.
- Each run queue contains two priority arrays -active and expired.
    - The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks.
    - The scheduler chooses the task with the highest priority from the active array for **execution** on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own run queue structure.
    - When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged; the expired array becomes the active array, and vice versa.

6.6.2 Windows 2000 Scheduling

- Windows 2000 is designed as responsive to the need of a user
- A preemptive scheduler is implemented
- Scheduler has flexible priority levels that include RR scheduling within each level
- Some levels also have dynamic priority variation based on current thread activity

Process and Thread Priorities

- Priorities are organized in two bands or classes: real time and variable
- Each band has 16 levels
- Threads requiring immediate attention are in real time class
- Overall W2K uses a priority driven preemptive scheduler so real time threads have preference over other threads

Priorities in two classes are handled differently

- All real time class threads have fixed priority that never changes (RR queue is used)
- In the variable class a threads priority begins with some initial value which may then change up/down during threads life time
- There is a FIFO queue at each priority level, but a process may migrate to one of the other queues within the variable priority class
- No process can be promoted to the upper class

- Initial priority of a thread in variable class is determined by:

Process Based Priority: Can have any value between 0 – 15

Thread Based Priority: This could be equal or within two levels above or below that of the process. For example if a process has a base priority 4, one of its thread has -1, then initial priority of that thread is 3

Multi-processor Scheduling

For a single processor

- Highest priority thread is always active unless it is blocked for some reason
- If more that one thread has the same priority, processor is shared based on RR

For Multi-processor

- A system with N processor, (N – 1) highest priority threads are always active, running exclusively on (N – 1) processors.
- The remaining lower priority threads run on the single remaining processor

Very early MS-DOS and Microsoft Windows systems were non-multitasking, and as such did not feature a scheduler. Windows 3.1x used a non-preemptive scheduler, meaning that it did not interrupt programs. It relied on the program to end or tell the OS that it didn't need the processor so that it could move on to another process. This is usually called cooperative multitasking. Windows 95 introduced a rudimentary preemptive scheduler; however, for legacy support opted to let 16 bit applications run without preemption.

Windows NT-based operating systems use a multilevel feedback queue. 32 priority levels are defined, 0 through to 31, with priorities 0 through 15 being "normal" priorities and priorities 16 through 31 being soft real-time priorities, requiring privileges to assign. 0 is reserved for the Operating System. Users can select 5 of these priorities to assign to a running application from the Task Manager application, or through thread management APIs. The kernel may change the priority level of a thread depending on its I/O and CPU usage and whether it is interactive (i.e. accepts and responds to input from humans), raising the priority of interactive and I/O bounded processes and lowering that of CPU bound processes, to increase the responsiveness of interactive applications. The scheduler was modified in Windows Vista to use the cycle counter register of modern processors to keep track of exactly how many CPU cycles a thread has executed, rather than just using an interval-timer interrupt routine. Vista also uses a priority scheduler for the I/O queue so that disk defragmenters and other such

programs don't interfere with foreground operations.

6.7 Summary

What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can can have a big effect on resource utilization and the overall performance of the system.

CPU/IO burst cycle. A process will run for a while (the CPU burst), perform some IO (the IO burst), then run for a while more (the next CPU burst). How long between IO operations? Depends on the process.

- IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
- CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

Preemptive vs. Non-preemptive SJF scheduler. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.

Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.

Basic assumptions behind most scheduling algorithms:

- There is a pool of runnable processes contending for the CPU.
- The processes are independent and compete for resources.
- The job of the scheduler is to distribute the scarce resource of the CPU to the different processes "fairly" (according to some definition of fairness) and in a way that optimizes some performance criteria.

First-Come, First-Served. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.

Shortest-Job-First (SJF) can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?

Implementing round-robin requires timer interrupts. When schedule a process, set the timer

to go off after the time quantum amount of time expires. If process does IO before timer goes off, no problem - just run next process. But if process expires its quantum, do a context switch. Save the state of the running process and run the next process.

Self Assessment Questions

1. Difference between Turnaround time and response time.

Given following information.

Process no.	Arrival time	CPU Burst
1	0	10
2	1	2
3	2	3
4	3	1
5	4	5

a. Compute waiting & turnaround time for FCFS and SJF scheduling algorithms.

b. Which of the schedules in part (a) results in the minimal average waiting time (over all processes)

3. Rank the following scheduling algorithms on a scale of 1 to 4 (*1 being the highest*) in terms of the extent to which they facilitate **low average waiting time**:
   - First-Come-First-Served
   - Shortest Job First (Non-Preemptive)
   - Priority (Preemptive)
   - Round-Robin

4. Explain the differences in the degree to which the following scheduling algorithms discriminate in favour of short processes:
   - First-Come-First-Serve
   - Round-Robin

5. Explain various scheduling criterias. Why we need scheduling Algorithms. Explain in brief all the scheduling algorithms.

6. What are CPU Bound And I/O bound Jobs.

7. What do you understand by CPU-I/O burst?

8. Differentiate between preemptive and nonpreemptive scheduling

Unit 7: Process Synchronization

7.0 Objective

After studying this unit, you will be able to understand the concept of synchronizing different process running within a computer. You will learn about race condition and need to synchronization, critical section issues. You will also learn about simaphores and monitors.

7.1 Meaning of Synchronization

In operating system, **synchronization** refers to one of two distinct but related concepts: synchronization of processes, and synchronization of data. **Process synchronization** refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action. **Data synchronization** refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization.

Process Synchronization Problem

Resource sharing is not the only area of concern in multiprogramming systems.

Synchronization is an important problem in Inter-Process Communication.

Consider the following example, in which two processes, running concurrently, are sharing a bounded buffer. One process is producing items to place in the buffer, while another process is consuming the items in the buffer.

```
buffer[10]          // buffer of size
10 Producer process-
while (true) // loop forever
produce (item)     // create a new
item enter_item (item) // place
item in buffer Consumer process-
while (true)        // loop forever
remove_item (item) // remove an item from the buffer
```

Here, the producer process must not be allowed to place an item in the buffer unless an empty slot in the buffer exists. Conversely, the consumer must not be allowed to remove an item from the buffer unless an item exists. This example illustrates the need for **synchronization** between processes. This means that a certain sequence of events must not be allowed to happen. Synchronization is different from mutual exclusion, in that synchronization would not prevent the producer and consumer from accessing the buffer at the same time.

7.2 Need of Synchronization

In computing, a **process** is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

7.2.1 Thread and process synchronization

Thread synchronization or serialization, strictly defined, is the application of particular mechanisms to ensure that two concurrently-executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes. Synchronization is used to control access to state both in small scale multiprocessing systems in multithreaded environments and multiprocessor computers and in distributed computers consisting of thousands of units in banking and database systems, in web servers, and so on.

A **thread of execution** is the smallest unit of processing that can be scheduled by an

operating system. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

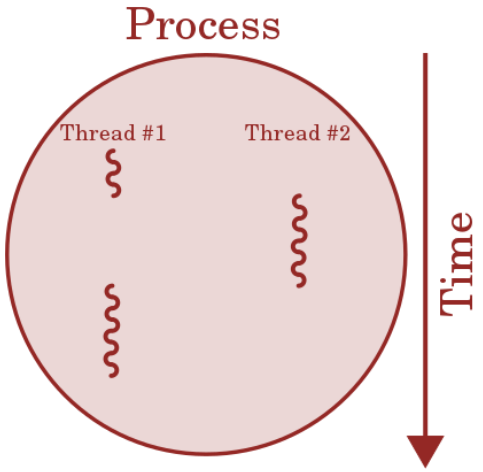


Figure 7.1: A process with two threads of execution on a single processor.

On a single processor, **multithreading** generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor (including multi-core system), the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task. Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The kernel of an operating system allows programmers to manipulate threads via the system call interface. Some implementations are called a *kernel thread*, whereas a *lightweight process* (LWP) is a specific type of kernel thread that shares the same state and information.

7.2.2 Data Synchronization

Data synchronization is the process of establishing consistency among data from a source to a target data storage and vice versa and the continuous harmonization of the data over time.

7.2.3 File-Based Solutions

There are tools available for file synchronization, version control (CVS, Subversion, etc.), distributed file systems (Coda, etc.), and mirroring, in that these entire attempt to keep sets of files synchronized. However, only version control and file synchronization tools can deal

with modifications to more than one copy of the files.

File synchronization is commonly used for home backups on external hard drives or updating for transport on USB flash drives. The automatic process prevents copying already identical files and thus can save considerable time from a manual copy, also being faster and less error prone. Version control tools are intended to deal with situations where more than one person wants to simultaneously modify the same file, while file synchronizers are optimized for situations where only one copy of the file will be edited at a time. For this reason, although version control tools can be used for file synchronization, dedicated programs require less overhead.

Distributed file systems may also be seen as ensuring multiple versions of a file are synchronized. This normally requires that the devices storing the files are always connected, but some distributed file systems like Coda allow disconnected operation followed by reconciliation. The merging facilities of a distributed file system are typically more limited than those of a version control system because most file systems do not keep a version graph.

Mirroring: A mirror is an exact copy of a data set. On the Internet, a mirror site is an exact copy of another Internet site. Mirror sites are most commonly used to provide multiple sources of the same information and are of particular value as a way of providing reliable access to large downloads.

Synchronization can also be useful in encryption for synchronizing Public Key Servers.

7.3 Race Condition

When two processes sharing a common variable try to update it simultaneously, one cannot predict the output of it, this is the race condition. A thread while updating the variable can be preempted by another thread and update it differently This is why synchronization mechanisms are used.

When different computational results (e.g., output, values of variables) occur depending. On the particular timing and resulting order of execution of statements across separate

Threads or processes

Example:

X=5

Process 1: X=X+1

Process 2: X=X+2

Machine code is

LOAD EAX,MEMORY_X

```
ADD EAX , 1
MOV MEMORY_X,EAX
```

While process1 is executing second line it could be preempted. And process2 takes turn so, MEMORY_X contains value 7. Now Process1 comes back and starts the remaining lines of code. The memory then contains value 6. Which is the final answer But we should have got a value of 8 instead.

7.3.1 Race Condition Properties

There are three properties that are necessary for a race condition to exist:

1. Concurrency Property. There must be at least two control flows executing concurrently.
2. Shared Object Property. A shared race object must be accessed by both of the concurrent flows.
3. Change State Property. At least one of the control flows must alter the state of the race object.

Solution strategy

1. Need to ensure that only one process or thread accesses a variable or I/O device until it has completed its required sequence of operations.
2. In general, a thread needs to perform some sequence of operations on I/O device or data Structure to leave it in a consistent state, before the next thread can access the I/O Device or data structure.

7.4 Critical-Section Problem

The producer-consumer problem is a specific example of a more general situation known as the ***critical section*** problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:

- Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
- The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
- The code following the critical section is termed the exit section. It generally releases

the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.

The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

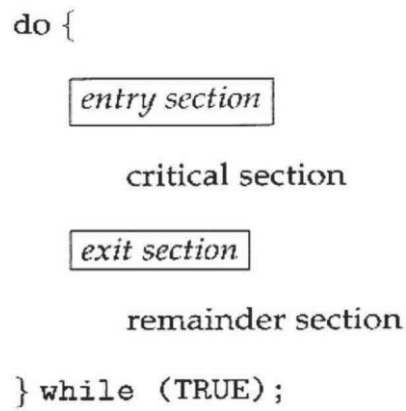


Figure 7.2: General structure of a typical process P_i

A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion -** Only one process at a time can be executing in their critical section.
2. **Progress** – If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (i.e. processes cannot be blocked forever waiting to get into their critical sections.)
3. **Bounded Waiting -** There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (i.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

   We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the ***relative*** speed of one process versus another. Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly, kernels can take on one of two forms:

   a. Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be

problematic for real-time systems, because timing cannot be guaranteed.

b. Preemptive kernels allow for real-time operations but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

Non-preemptive kernels include Windows XP, 2000, traditional UNIX, and Linux prior to 2.6. Preemptive kernels include Linux 2.6 and later, and some commercial UNIXes such as Solaris and IRIX.

7.5 Synchronization Hardware

In this section, we present some simple hardware instructions that are available on many Systems and show how they can be used effectively in solving the critical-section problem. The Critical-section problem could be solved simply in a uniprocessor environment if we could disallow interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically. We can use these special instructions to solve the critical section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, let us abstract the main concepts behind these types of instructions.

The *Test-and-Set instruction* can executed atomically—that is, as one uninterruptible unit. Thus, if two *Test-and-Set instructions* are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

If the machine supports the *Test-and-Set instruction*, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi, is shown in Figure.

The Swap instruction, defined as as shown in Figure operates on the contents of two words like the *Test- and-Set instruction* it is executed atomically.

If the machine supports the *Swap instruction*, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false.

```
Function Test and Set (var target:boolean):boolean Begin
        Test-and-Set:=target;
        Target:=true;
    End;
```

Figure 7.3: the definition of the Test-and-Set instruction.

```
Repeat
While Test-and-Set (lock) do no-op; Critical
section
Lock: =false;
Reminder section Until
false;
```

Figure 7.4: Mutual-exclusion implementation with *Test-and-Set.*

- In addition, each process also has a local Boolean variable *key*. The structure of process PJ is shown in Figure. These algorithms do not satisfy the bounded-wait in requirement. We present an algorithm that uses the *Test-and-Set* instruction in Figure.

This algorithm satisfies all the critical-section requirements.

```
Procedure Swap(var a,b:boolean); var
temp:boolean:
Begin temp:=a;
a:=b; b:=temp;
End;
```

Figure 7.5: The definition of the Swap instruction

```
Repeat
        Key:= true;
Repeat
        Swep (lock , key);
        Until key:= false;
        Critical section
        Lock: =false
        Reminder section
        Until false;
```

Figure 7.6: Mutual-exclusion implementation with the *Swap* instruction.

```
Repeat
 Waiting[i]=true;
    Key=true;
 While=waiting[i]and key do key:=Test-and-Set(lock);
  Waiting[i]=false;
 Critical section J:=i+1
   mod n;
 While (j=i) and(not waiting[j])do j:=j+1 mod n; if
  j=i then lock:=false
 else waiting[j]:=false;
 reminder section
    until false;
```

Figure 7.7: Bounded-waiting mutual exclusion with *Test-and-Set.*

7.6 Introduction to Semaphore & Monitor

Sometimes a process may need to wait for some other process to finish before it can continue. In this instance, the two processes need to be synchronized together. There are a number of ways in which this can be done. A common method in operating systems is to use a variable called a **semaphore** that only one process can own at a time. There are two calls associated with a semaphore, one to lock it and one to unlock it. When a process attempts to lock a semaphore, it will be successful if the semaphore is free. If the semaphore is already locked, the process requesting the lock will be blocked and remain blocked till the process that has the semaphore unlocks it. When that happens, the process that was blocked will be unblocked and the semaphore can then be locked by it.

System semaphores are used by the operating system to control system resources. A program can be assigned a resource by getting a semaphore (via a system call to the operating system). When the resource is no longer needed, the semaphore is returned to the operating system, which can then allocate it to another program.

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using the semaphore techniques.

Types of semaphore:

A semaphore is a variable. There are 2 types of semaphores:

Binary semaphores

Counting semaphores

Binary semaphores have two methods associated with it. (up, down / lock, unlock) Binary semaphores can take only 2 values (0/1). They are used to acquire locks. When a resource is available, the process in charge set the semaphore to 1 else 0.

Counting Semaphore may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semaphore initislize'.

Binary Semaphores can assume only the value 0 or the value 1 counting semaphores also called general semaphores can assume only nonnegative values.

The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S), operates as follows:

P(S): **IF** S > 0
THEN S := S - 1
ELSE (wait on S)

The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S), operates as follows:

V(S): **IF** (one or more process are waiting
on S) **THEN** (let one of these
processes proceed) **ELSE** S := S +1

Operations P and V are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operation has stared, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within **P(S)** and **V(S)**.

If several processes attempt a P(S) simultaneously, only process will be allowed to proceed. The other processes will be kept waiting, but the implementation of P and V guarantees that processes will not suffer indefinite postponement.

Semaphores solve the lost-wakeup problem

7.6.1 Producer-Consumer Problem Using Semaphores

The Solution to producer-consumer problem uses three semaphores, namely, full, empty and mutex.

The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

Initialization

* Set full buffer slots to 0.
  i.e., semaphore Full = 0.
* Set empty buffer slots to N. i.e., semaphore empty = N.
* For control access to critical section set mutex to 1. i.e., semaphore mutex = 1.

```
Producer (
) WHILE
(true)
        produce-Item
        ( ); P (empty);
        P (mutex);
        enter-
        Item ( ) V
        (mutex)
        V (full);
Consumer ( )
WHILE (true)
        P (full)
        P (mutex);
```

```
        remove-Item
        ( ); V (mutex);
        V (empty);
consume-Item (Item)
```

7.6.2 What is a Monitor?

A monitor is a **set of multiple** routines which are protected by a mutual exclusion lock. None of the routines in the monitor can be executed by a thread until that thread acquires the lock. This means that only ONE thread can execute within the monitor at a time. Any other threads must wait for the thread that's currently executing to give up control of the lock.

However, a thread can actually suspend itself inside a monitor and then wait for an event to occur. If this happens, then another thread is given the opportunity to enter the monitor. The thread that was suspended will eventually be notified that the event it was waiting for has now occurred, which means it can wake up and reacquire the lock.

7.6.3 Differences between Monitors and Semaphores

Both Monitors and Semaphores are used for the same purpose – thread synchronization. But, monitors are simpler to use than semaphores because they handle all of the details of lock acquisition and release. An application using semaphores has to release any locks a thread has acquired when the application terminates – this must be done by the application itself. If the application does not do this, then any other thread that needs the shared resource will not be able to proceed.

Another difference when using semaphores is that every routine accessing a shared resource has to explicitly acquire a a lock before using the resource. This can be easily forgotten when coding the routines dealing with multithreading. Monitors, unlike semaphores, automatically acquire the necessary locks.

7.7 Summary

Process synchronization mean the coordination of simultaneious threads or processes to complete a task in order to get correct runtime order and avoid unexpected race conditions. On the other hand data synchronization is to keep multiple copies of dataset in coherence with one another.

A critical section is a piece of code that accesses a shared resource. And critical section problem is to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Semaphores are used to help with synchronization. If multiple processes share a common resources, they need a way to be able to use that resource without disrupting each other. You want each process to be able to read from and write to that resource uninterrupted. A semaphore will either allow or disallow access to the resource, depending on how it is set up.

Self-Assessment Questions

1. Explain Process Synchronization. How it is different from Data Synchronization?
2. What is Critical Section Problem. Explain its general structure.
3. Write some hardware instruction used for solving critical section problem.
4. Explain Semaphore and its different types.

8.0 Objective

After studying this unit you will be able to understand about deadlocks and conditions for deadlocks like mutual exclusion, hold and wait, no-preemption and circular wait. You will be able to understand about algorithms like bankers algorithm to avoid deadlocks. Also, you will get ideas about ways to detect deadlocks and recover from them.

8.1 Introduction

A **deadlock** is a situation wherein two or more competing actions are each waiting for the other to finish. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software lock* or *soft lock*. Computers intended for the *time-sharing* and/or *real-time* markets are often equipped with a *hardware lock* (or *hard lock*) which guarantees *exclusive access* to processes, forcing serialized access. Deadlocks are particularly troubling because there is no *general* solution to avoid (soft) deadlocks.

Examples: This situation may be like, two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler,

a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil to finish his work with the ruler. Neither request can be satisfied, so a deadlock occurs.

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

An example of a deadlock which may occur in database products is the following. Client applications using the database may require exclusive access to a table, and in order to gain exclusive access they ask for a *lock*. If one client application holds a lock on a table and attempts to obtain the lock on a second table that is already held by a second client application, this may lead to deadlock if the second application then attempts to obtain the lock that is held by the first application. (This particular type of deadlock could be prevented, by using an *all-or-none* resource allocation algorithm.)

8.2 Necessary Conditions for Deadlocks

There are four necessary conditions for a deadlock to occur

1. **Mutual Exclusion**: A resource that cannot be used by more than one process at a time
2. **Hold and Wait**: Processes already holding resources may request new resources held by other processes
3. **No Preemption:** No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.

Circular Wait: Two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds. When circular waiting is triggered by mutual exclusion operations it is sometimes called *lock inversion*.

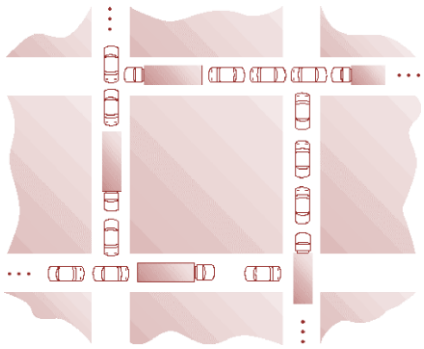


Figure 8.1: The traffic deadlock

Consider each section of the street as a resource.

1. Mutual exclusion condition applies, since only one vehicle can be on a section of

the street at a time.

2. 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.
3. No-preemptive condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.
4. Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of 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.

8.3 Prevention

All four conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions

8.3.1 Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-sharable 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.

8.3.2 Elimination of "Hold and Wait" Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of 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 of 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 derives 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.

8.3.3 Elimination of "No-preemption" Condition

The no preemption 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. 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 second process may hold the resources needed by the first process. This is a deadlock. This strategy require 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.

8.3.4 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 impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.

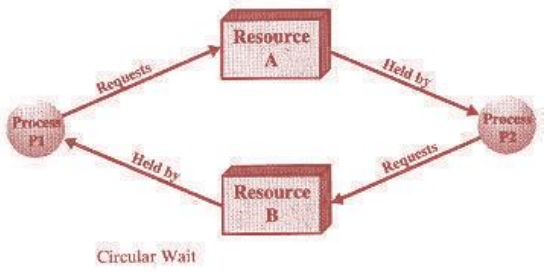


Figure 8.2 : Provide a global numbering of all the resources

1 = Card reader

2 = Printer

3 = Plotter

4 = Tape drive

5 = Card punch

Now the rule is this: 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.

8.4 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra, is the Banker's algorithm. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

8.4.1 Banker's Algorithm

In this analogy, customers are processes, Units are reprocess, say tape drive and Bauker in the operating system.

Customers	Used	Max	
A	*0*	*6*	
B	*0*	*5*	*Available*
C	*0*	*4*	*Units = 10*
D	0	7	

Figure 8.3

In the above figure, we see four customers each of whom has been granted a number of credit units. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	Used	Max
A	*1*	*6*
B	*1*	*5*
C	*2*	*4*
D	*4*	*7*

Figure 8.4

Safe State The key to a state being safe is that there is at least one way for all users to finish.

In other analogy, the state of figure 2 is safe because with 2 units left, the banker can delay any request except *C's*, thus letting *C* finish and release all four resources. With four units in hand, the banker can let either *D* or *B* have the necessary units and so on.

Unsafe State Consider what would happen if a request from B for one more unit were granted in above figure 2.

We would have following situation

Customers	Used	Max	
A	*1*	*6*	
B	*2*	*5*	*Available*
C	*2*	*4*	*Units = 1*
D	*4*	*7*	

Figure 8.5

This is an unsafe state.

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker 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 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 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 postponed until later. Haberman [1969] has shown that executing of the algorithm has 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.

8.5 Deadlock Detection

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.

- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

8.6 Recovery from Deadlock

- There are two options for breaking a deadlock:
  - To abort one or more processes to break the circular wait (Process Termination).
  - To preempt some resources from one or more of deadlock processes (Resource Preemption).

8.6.1 Recovery from Deadlock: Process Termination

- Two methods to eliminate deadlocks by aborting a process. In both methods, the system reclaims all resources allocated to the terminated processes:
  - Abort all deadlocked processes: It will break the deadlock cycle, but a great expense.
  - Abort one process at a time until the deadlock cycle is eliminated: Overhead, since, after each process aborted a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated?

8.6.2 Recovery from Deadlock: Resource Preemption

If preemption is required to deal with deadlocks, then three issues need to be addressed:

Selecting a victim – which resources and which processes are to be preempted? (Minimize cost)

Rollback – If we preempt a resource from a process, what should be done with that process? We must roll back the process to some safe state and restart it from that

state.

Starvation – That is same process may always be picked as victim, include number of rollback is cost factor. Insure that a starvation will not occur. That is Guarantee that resources will not always be preempted from the same process.

8.7 Summary

In a multiprogramming environment, several process compete for resources. A situation may arise where a process is waiting for a resource that is held by other waiting processes. This situation is called a deadlock.

A system has a finite set of resources such as memory, I/O devices, etc. It also has a finite set of processes that need to used these resources. A process that wishesh to use any of these resources, makes a request to use that resource. If the resource is free the process gets it. If it is being used by another process, it waits for it to become free. The assumption is that the resource will eventually become free and the waiting process will then used the resources. But in some situations, the other process may also be waiting for some resource.

Self Assessment Questions

1. How can we deal with deadlock?
2. What is the difference between prevention and avoidance?
3. What are examples of strategies for prevention?
4. What happens to a process if resources it is holding are preempted?
5. Is it safe for the process to just go on executing?
6. Short of killing the process that is preempted, how might this problem be addressed?
7. What is the difference between a safe state and a deadlock-free state?
8. What is a sufficient condition for deadlock in each of these models?
9. What you understand by safe and unsafe state? Explain banker's algorithm. with necessary data structure for deadlock avoidance.
10. Explain how can we eliminating deadlock by aborting deadlock.
11. How can we recover from deadlock?
    a) Process termination.
    b) Resource preemption.
12. Explain how we eliminating deadlock by resource preemption can.

a) Selecting victim.

b) Rollback.

c) Starvation.

13. Explain the algorithm that examines whether the deadlock has accured.

a) Single instance of each resource type.

b) Several instance of each resource type.

14. Explain the two deadlock avoidance algorithms.

a) Safe state.

b) Resource allocation graph algorithm.

15. Explain how we can prevent deadlock.

Unit 9: Memory Management

9.0 Objective

After studying this unit you will be able to understand basic hierarchy of different types of memories available in a computer system on the basis of speed and capacity. You will be able to know the concepts of loading, linking and memory allocation for different processes. You will learn about physical and logical address space and paging algorithms.

9.1 Introduction

The Memory Management is the part of the operating system that must solve the above issues. In other words memory management is about sharing memory so that the largest number of processes can run in the most efficient way. Memory management is a collection of techniques for providing sufficient memory to one or more processes in a computer system, especially when the system does not have enough memory to satisfy all processes' requirements simultaneously. Techniques include swapping, paging and virtual memory. Memory management is usually performed mostly by a hardware memory management unit.

The von Neumann principle for the design and operation of computers requires that a program has to be primary memory resident to execute. Also, a user requires revisiting his programs often during its evolution. However, due to the fact that primary memory is volatile, a user needs to store his program in some non-volatile store. All computers provide a non-volatile secondary memory available as an online storage. Programs and files may be disk resident and downloaded whenever their execution is required.

Therefore, some form of memory management is needed at both primary and secondary memory levels.

Secondary memory may store program scripts, executable process images and data files. It may store applications, as well as, system programs. In fact, a good part of all OS, the system

programs which provide services (the utilities for instance) are stored in the secondary memory. These are requisitioned as needed.

The main motivation for management of main memory comes from the support for multiprogramming. Several executables processes reside in main memory at any given time. In other words, there are several programs using the main memory as their address space.

Also, programs move into, and out of, the main memory as they terminate, or get suspended for some IO, or new executables are required to be loaded in main memory. So, the OS has to have some strategy for main memory management. In this chapter we shall discuss the management issues and strategies for both main memory and secondary memory. Let us begin by examining the issues that prompt the main memory management.

Allocation: First of all the processes that are scheduled to run must be resident in the memory. These processes must be allocated space in main memory. Swapping, fragmentation and compaction: If a pro- gram is moved out or terminates, it creates a hole, (i.e. a contiguous unused area) in main memory. When a new process is to be moved in, it may be allocated one of the available holes. It is quite possible that main memory has far too many small holes at a certain time. In such a situation none of these holes is really large enough to be allocated to a new process that may be moving in. The main memory is too Operating Systems/Memory management fragmented. It is, therefore, essential to attempt compaction. Compaction means OS re-allocates the existing programs in contiguous regions and creates a large enough free area for allocation to a new process.

Garbage collection: Some programs use dynamic data structures. These programs dynamically use and discard memory space. Technically, the deleted data items (from a dynamic data structure) release memory locations. However, in practice the OS does not collect such free space immediately for allocation. This is because that affects performance. Such areas, therefore, are called garbage. When such garbage exceeds a certain threshold, the OS would not have enough memory available for any further allocation. This entails compaction (or garbage collection), without severely affecting performance.

Protection: With many programs residing in main memory it can happen that due to a programming error (or with malice) some process writes into data or instruction area of some other process. The OS ensures that each process accesses only to its own allocated area, i.e. each process is protected from other processes. Virtual memory: Often a processor sees a large logical storage space (a virtual storage space) though the actual main memory may not be that large. So some facility needs to be provided to translate a logical address available to a processor into a physical address to access the desired data or instruction.

IO support: Most of the block-oriented devices are recognized as specialized files. Their buffers

need to be managed within main memory alongside the other processes. The considerations stated above motivate the study of main memory management. One of the important considerations in locating an executable program is that it should be possible to relocate it any where in the main memory.

9.2 Memory Hierarchy

There is main memory that communicates directly with the C.P.U as shown in below figure. There is secondary or auxiliary memory which indirectly communicates with the main memory through input /output processor.

Cache memory is placed between main-memory and C.P.U.

1. Cache Memory-it is less than or equal to 4MB.Acess time is 3-10 nano seconds (10^{-9}), managed by hardware and backed by main-memory.
2. Main-Memory-It is greater than or equal to 1GB.Access time is 80-400 nano seconds, managed by operating system and back up by hard disk.
3. Disk storage-It is greater than 1 GB. Access time is 5, 000, 000, managed by operating system and user and backed by magnetic tape.

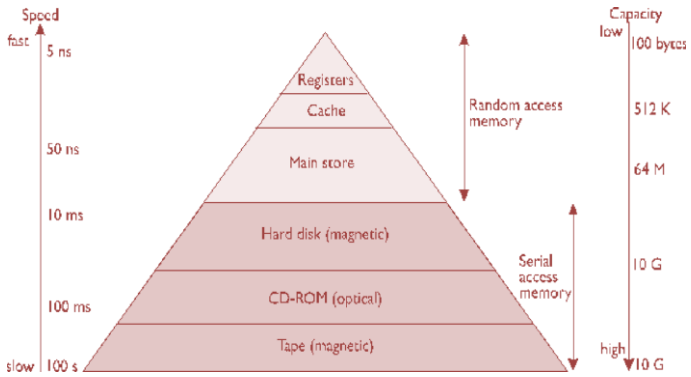


Figure 1: Diagram of hierarchy of memory organization in a computer

Address Binding

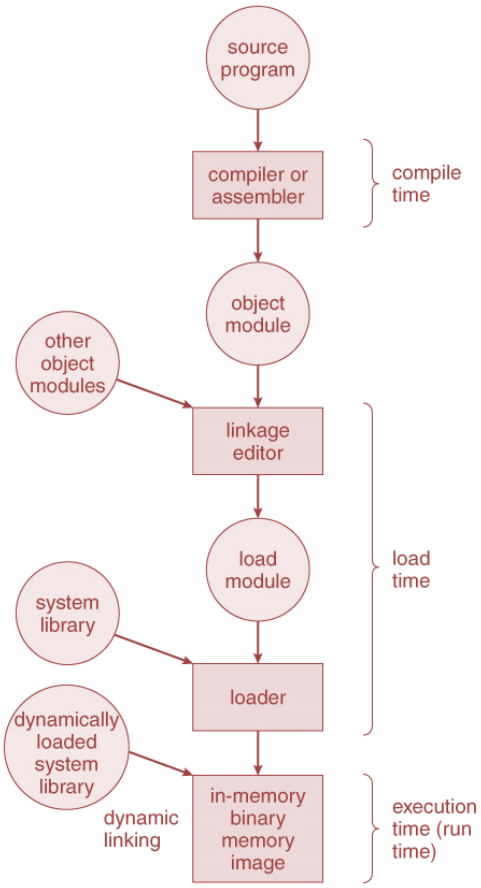
User programs typically refer to memory addresses with symbolic names such as "i", "count", and "aver- age Temperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:

- **Compile Time -** If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.
- **Load Time -** If the location at which a program will be loaded is not known at compile

time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.

- **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.

Figure 2: shows the various stages of the binding processes and the units involved in each stage:



Logical Versus Physical Address Space

- The address generated by the CPU is a ***logical address***, whereas the address actually seen by the memory hardware is a ***physical address***.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, have different logical and physical addresses.
    - In this case the logical address is also known as a ***virtual address***, and the two terms are used interchangeably by our text.

- The set of all logical addresses used by a program composes the ***logical address space***, and the set of all corresponding physical addresses composes the ***physical address space.***

➢ The run time mapping of logical to physical addresses is handled by the ***memory-management unit, MMU***.

- The MMU can take on many forms. One of the simplest is a modification of the base- register scheme described earlier.
- The base register is now termed a ***relocation register***, whose value is added to every memory request at the hardware level.

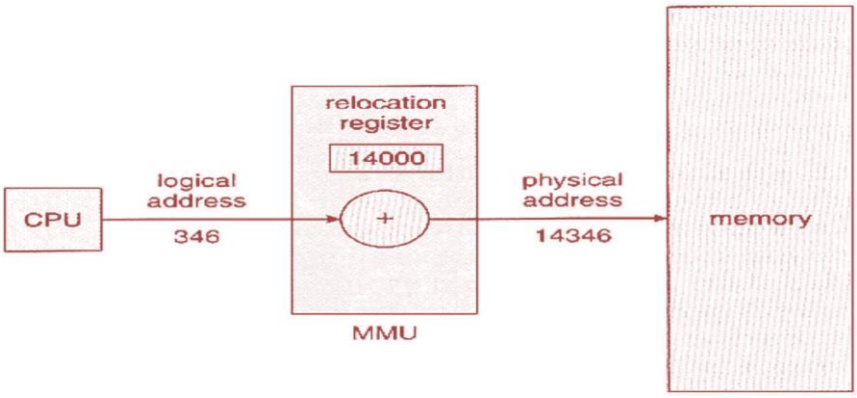


Figure 3: Dynamic relocation using a relocation register

Dynamic loading & linking

Dynamic Loading

➢ Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

Dynamic Linking and Shared Libraries

➢ With ***static linking*** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.

➢ With ***dynamic linking***, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

- This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.

- We will also learn that if the code section of the library routines is ***reentrant***, (meaning it does not modify the code while it runs, making it safe to re-enter it), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it. (Each process would have their own copy of the ***data*** section of the routines, but that may be small relative to the code segments.) Obviously the OS must manage shared routines in memory.
- An added benefit of ***dynamically linked libraries*** (DLLs, also known as ***shared libraries*** or ***shared objects*** on UNIX systems) involves easy upgrades and updates. When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes. However, if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.
- In practice, the first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes which are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***
- If compile-time or load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.
- Swapping is a very slow process compared to other operations. For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take 1/4 second (250 milliseconds) just to do the data transfer. Adding in a latency lag of 8 milliseconds and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well

as new data in, the overall transfer time required for this swap is 512 milliseconds, or over half a second. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.

- To reduce swapping transfer overhead, it is desired to transfer as little information as possible, which requires that the system know how much memory a process ***is*** using, as opposed to how much it ***might*** use. Programmers can help with this by freeing up dynamic memory that they are no longer using.
- It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise, the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (E.g. Paging.) However, some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again.

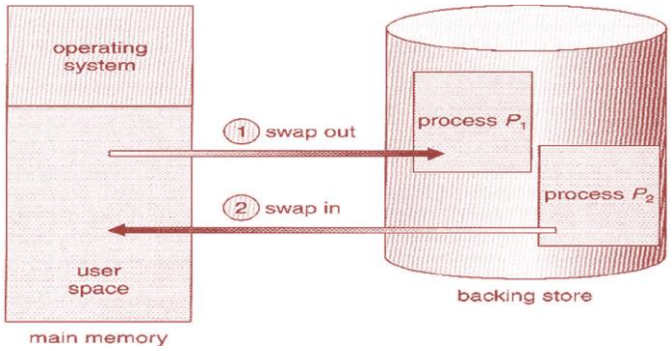


Figure 4: swapping of two processes using a disk as a backing store

Contiguous Memory Allocation

One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.)

Memory Mapping and Protection

- The system shown in Figure 5 below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

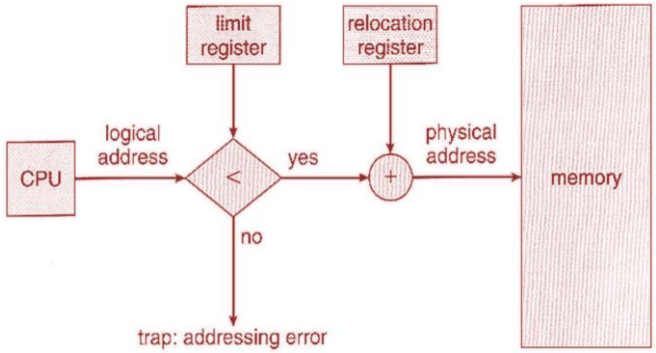


Figure 5: Hardware Support for Relocation and Limit

Registers Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. **First fit -** Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
2. **Best fit -** Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
3. **Worst fit -** Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

9.3 Fragmentation

- All the memory allocation strategies suffer from ***external fragmentation.*** External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement.
- The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.
- Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation.
    - ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average 1/2 block will be wasted per memory request, because on the average the last allocated block will be only half full.
    - Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.
    - Some systems use variable size blocks to minimize losses due to internal fragmentation.
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.

9.4 Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation

by allocating memory in equal sized blocks known as ***pages***.

- Paging eliminates most of the problems of the other methods discussed previously and is the predominant memory management technique used today.

Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

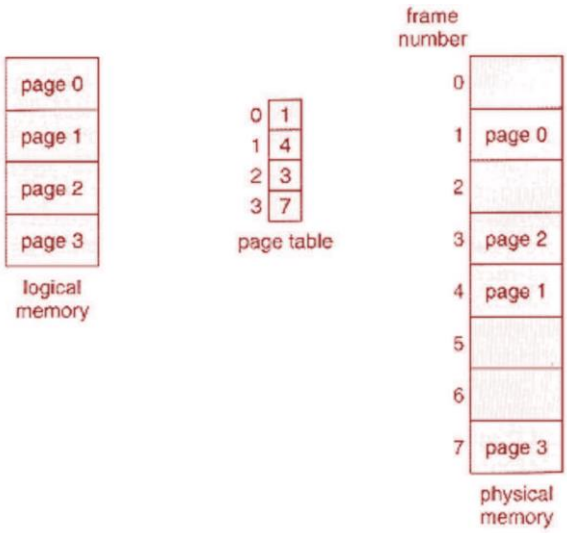
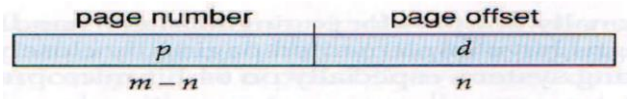


Figure 6: Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a

certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.



- (DOS used to use an addressing scheme with 16-bit frame numbers and 16-bit offsets, on hardware that only supported 24-bit hardware addresses. The result was a resolution of starting frame addresses finer than the size of a single frame, and multiple frame-offset combinations that mapped to the same physical hardware address.)

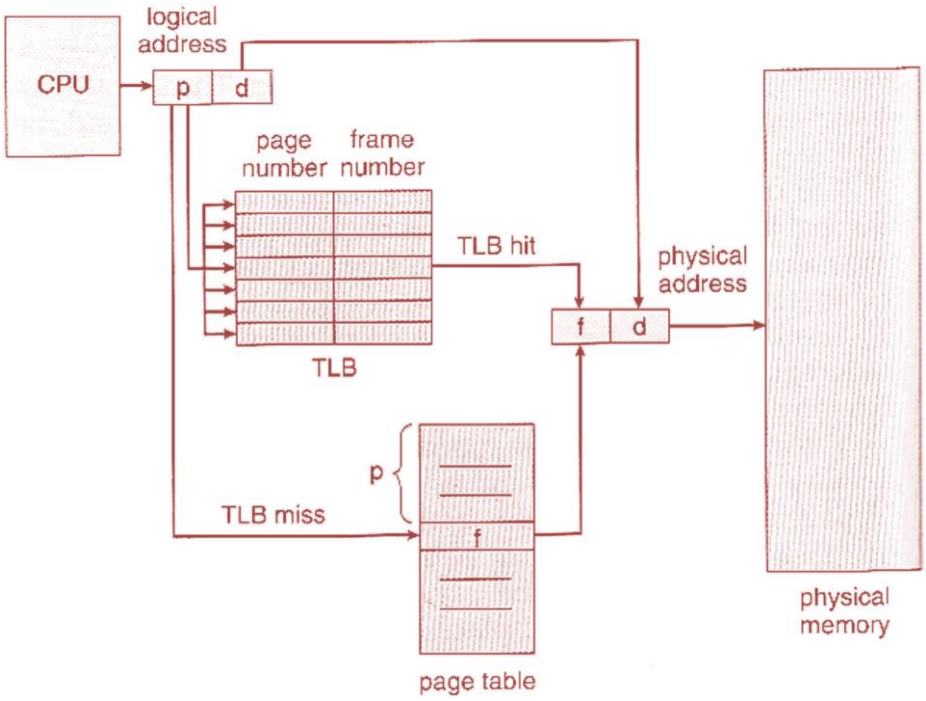


Figure 7: Paging Hardware

- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

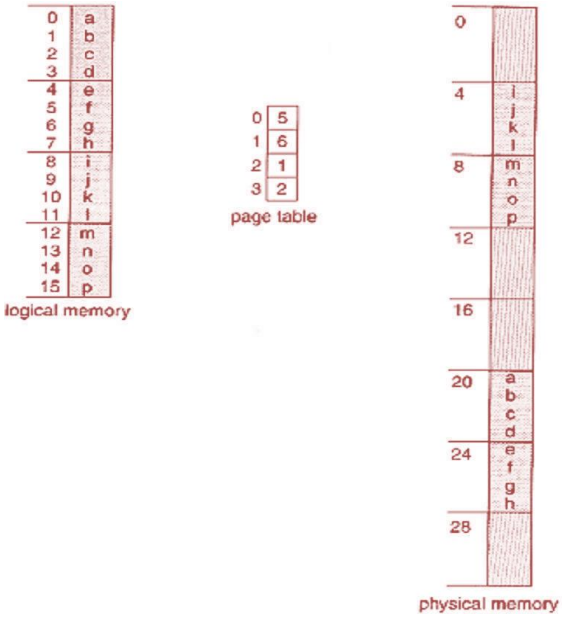


Figure 8: Paging example for a 32-byte memory with 4-byte pages

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process. (Possibly more, if processes keep their code and data in separate pages.)
- Larger page sizes waste more memory but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.
- Page table entries (frame numbers) are typically 32-bit numbers, allowing access to 2^32 physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. (32 + 12 = 44 bits of physical address space.)
- When a process requests memory (e.g., when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.

- The operating system must keep track of each individual process's page table, updating it when- ever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

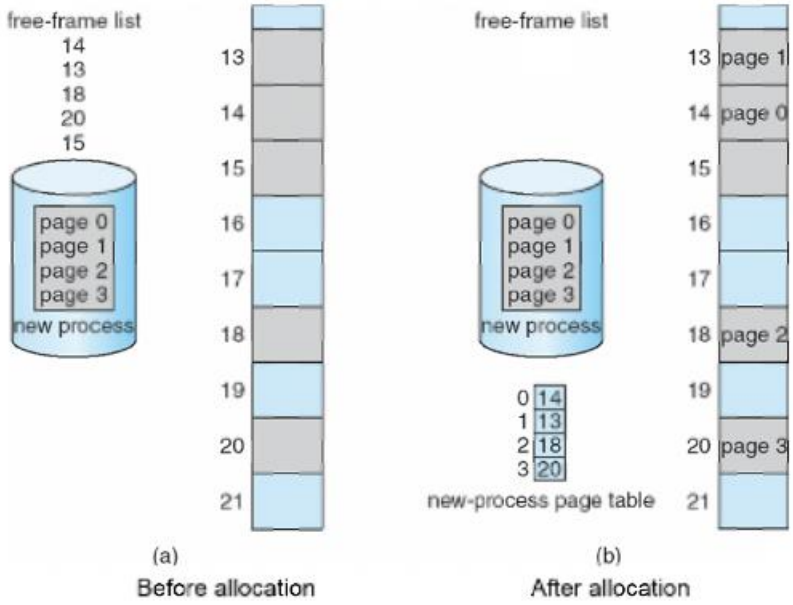


Figure 9: Free frames

Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use a set of registers for the page table. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)
- An alternate option is to store the page table in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record where in memory the page table is located.
    - Process switching is fast, because only the single register needs to be changed.

- However, memory access just got half as fast, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memorylocation.
- The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***

- The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.

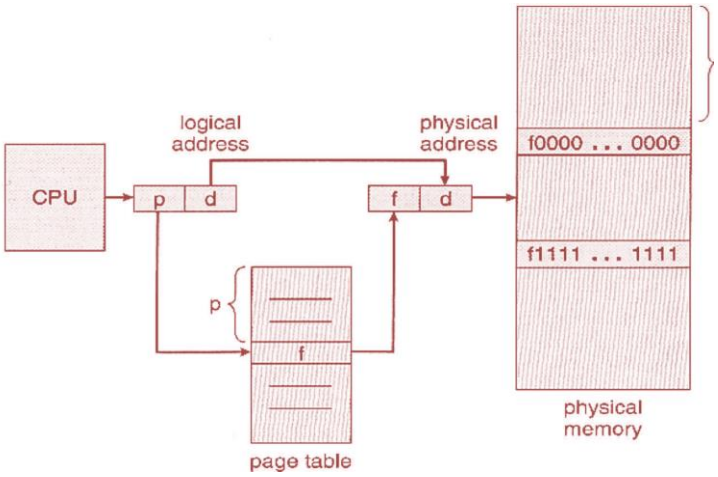


Figure 10: Paging hardware with TLB

- The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device.
- Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
- If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
- Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically, these would be kernel frames.
- Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.

- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nano- seconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So, with an 80% TLB hit ratio, the average memory access time would be:

  0.80 * 120 + 0.20 * 220 = 140 nanoseconds for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

Protection

- The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read- write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.
- Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process, as shown by example in Figure 8.12 below.
- Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. (Areas of memory in the last page that are not entirely filled by the process and may contain data left over by whoever used that frame last.)
- Many processes do not use all of the page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a ***page-table length register, PTLR***, to specify the length of the page table.

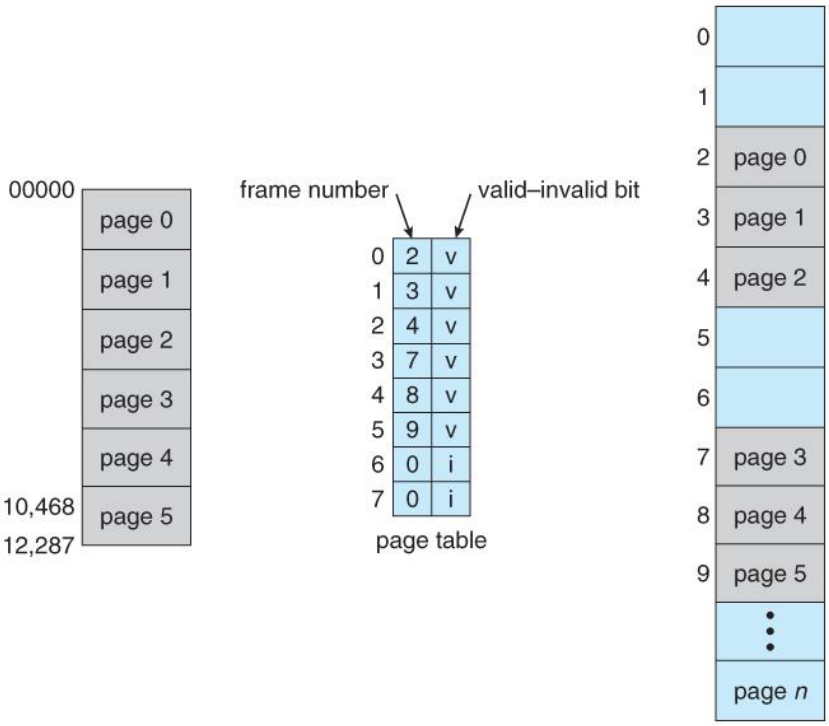


Figure 11: Valid(v) or invalid(i) bit in a page table

9.5 Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- If code is ***reentrant***, that means that it does not write to or change the code in any way (it is non self-modifying), and it is therefore safe to re-enter it. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.
- Some systems also implement shared memory in this fashion.

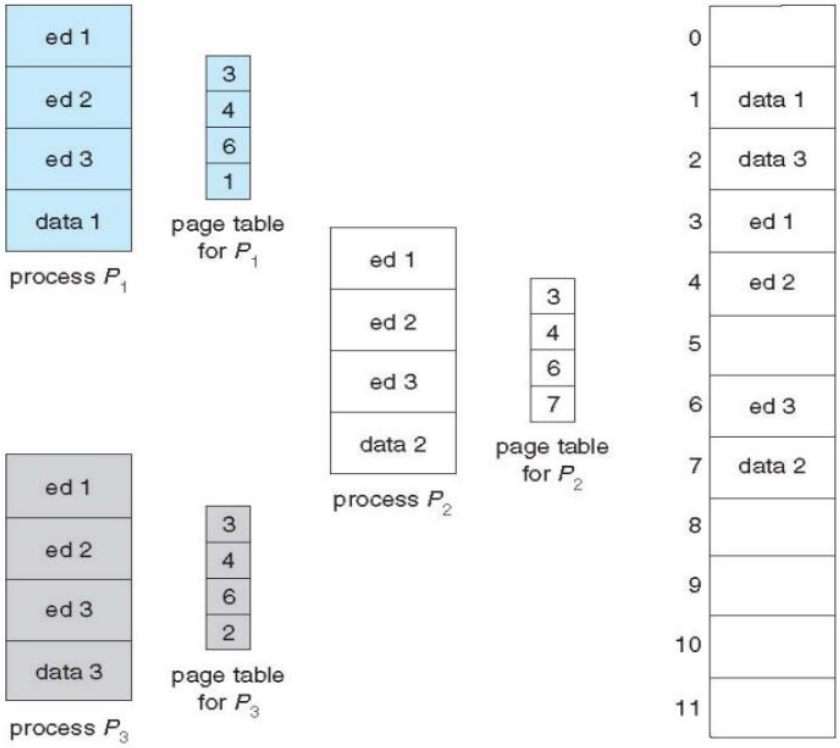


Figure12: Sharing of code in a paging environment

Basic method

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Fig.

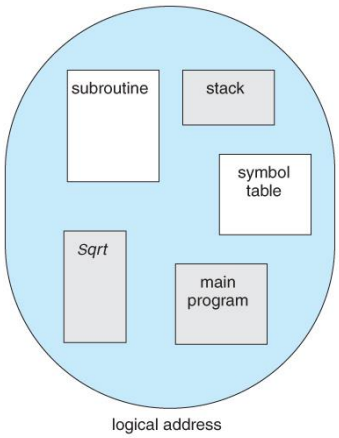


Figure13: User's view of a program

Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously. (Note that at this point in the discussion of segmentation, each segment is kept in contiguous memory and may be of different sizes, but that segmentation can also be combined with paging as we shall see shortly.)

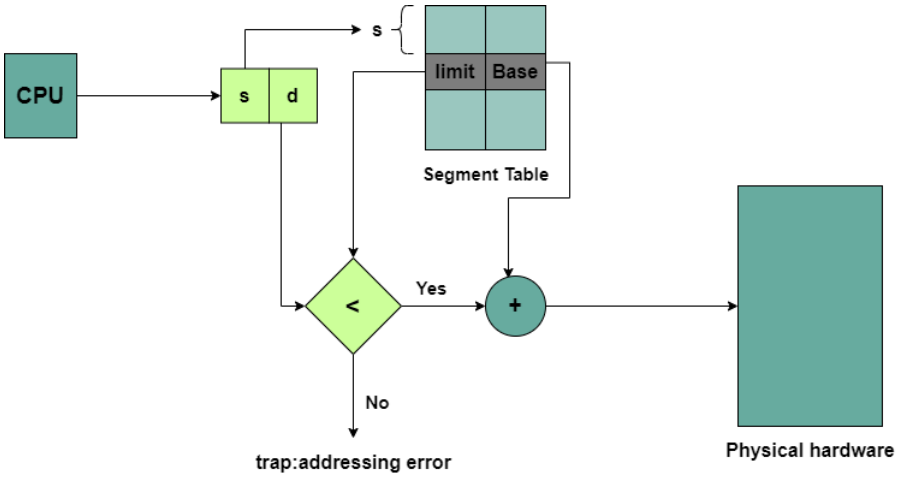


Figure14: Segmentation Hardware

9.6 Kernel memory Allocation

Kernel memory Allocation is treated differently from user memory. They are often allocated from a free memory pool. Kernel required memory for structures of varying size, some Kernel memory needs to be contiguous. Kernel Memory Allocation in Unix and Solaris is through Buddy System and Slab Allocator.

Buddy Allocator

In the buddy system, the memory is broken down into power-of-two sized naturally aligned blocks. These blocks are organized in an array of lists, in which the list with index i contains all unallocated blocks of size 2^i. The index i is called the order of block. There should be two

adjacent equally sized blocks in the list i (i.e. buddies), the buddy allocator would coalesce them and put the resulting block in list $i + 1$, provided that the resulting block would be naturally aligned. Similarly, when the allocator is asked to allocate a block of size 2^i, it first tries to satisfy the request from the list with index i. If the request cannot be satisfied (i.e. the list i is empty), the buddy allocator will try to allocate and split a larger block from the list with index $i + 1$. Both of these algorithms are recursive. The recursion ends either when there are no blocks to coalesce in the former case or when there are no blocks that can be split in the latter case.

This approach greatly reduces external fragmentation of memory and helps in allocating bigger continuous blocks of memory aligned to their size. On the other hand, the buddy allocator suffers increased internal fragmentation of memory and is not suitable for general kernel allocations. This purpose is better ad- dressed by the slab allocator.

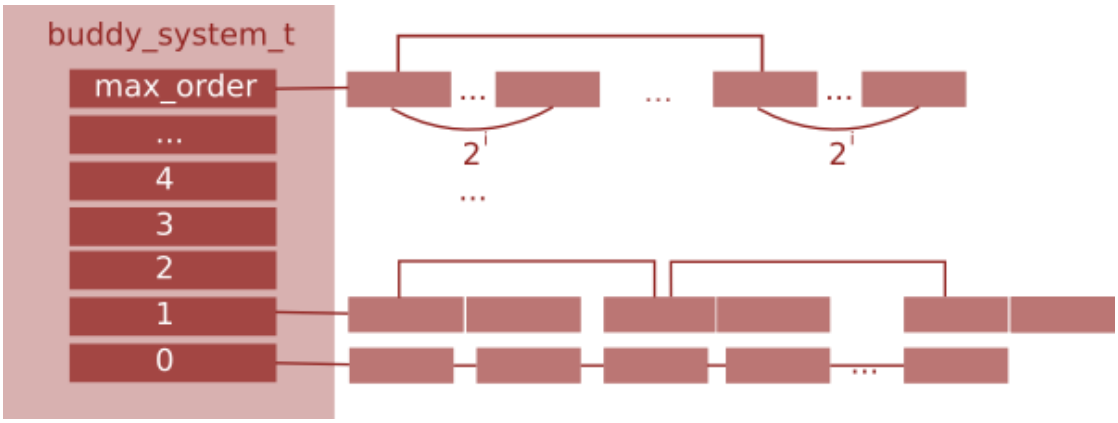


Figure 15: Buddy system scheme.

Implementation

The buddy allocator is, in fact, an abstract framework which can be easily specialized to serve one particular task. It knows nothing about the nature of memory it helps to allocate. In order to beat the lack of this knowledge, the buddy allocator exports an interface that each of its clients is required to implement. When supplied with an implementation of this interface, the buddy allocator can use specialized external functions to find a buddy for a block, split and coalesce blocks, manipulate block order and mark blocks busy or available.

Data organization. Each entity allocable by the buddy allocator is required to contain space for storing block order number and a link variable used to interconnect blocks within the same order. Whatever entities are allocated by the buddy allocator, the first entity within a block is used to represent the entire block. The first entity keeps the order of the whole block. Other entities within the block are assigned the magic value BUDDY_INNER_BLOCK. This is especially important for effective identification of buddies in a one-dimensional array because

the entity that represents a potential buddy cannot be associated with BUDDY_INNER_BLOCK(i.e., if it is associated with BUDDY_INNER_BLOCKthen it is not a buddy).

Slab allocator

The majority of memory allocation requests in the kernel is for small, frequently used data structures. The basic idea behind the slab allocator is that commonly used objects are preallocated in continuous areas of physical memory called slabs. Whenever an object is to be allocated, the slab allocator returns the first available item from a suitable slab corresponding to the object type. Due to the fact that the sizes of the requested and allocated object match, the slab allocator significantly reduces internal fragmentation.

Slab of one object type are organized in a structure called slab cache. There are ususally more slabs in the slab cache, depending on previous allocations. If the slab cache runs out of available slabs, new slabs are allocated. In order to exploit parallelism and to avoid locking of shared spinlocks, slab caches can have variants of processor-private slabs called magazines. On each processor, there is a two-magazine cache. Full magazines that are not part of any per-processor magazine cache are stored in a global list of full magazines.

Each object begins its life in a slab. When it is allocated from there, the slab allocator calls a constructor that is registered in the respective slab cache. The constructor initializes and brings the object into a known state. The object is then used by the user. When the user later frees the object, the slab allocator puts it into a processor private magazine cache, from where it can be precedently allocated again. Note that allocations satisfied from a magazine are already initialized by the constructor. When both of the processor cached magazines get full, the allocator will move one of the magazines to the list of full magazines. Similarily, when allocating from an empty processor magazine cache, the kernel will reload only one magazine from the list of full magazines. In other words, the slab allocator tries to keep the processor magazine cache only half-full in order to prevent thrashing when allocations and deallocations interleave on magazine boundaries. The advantage of this setup is that during most of the allocations, no global spinlock needs to be held.

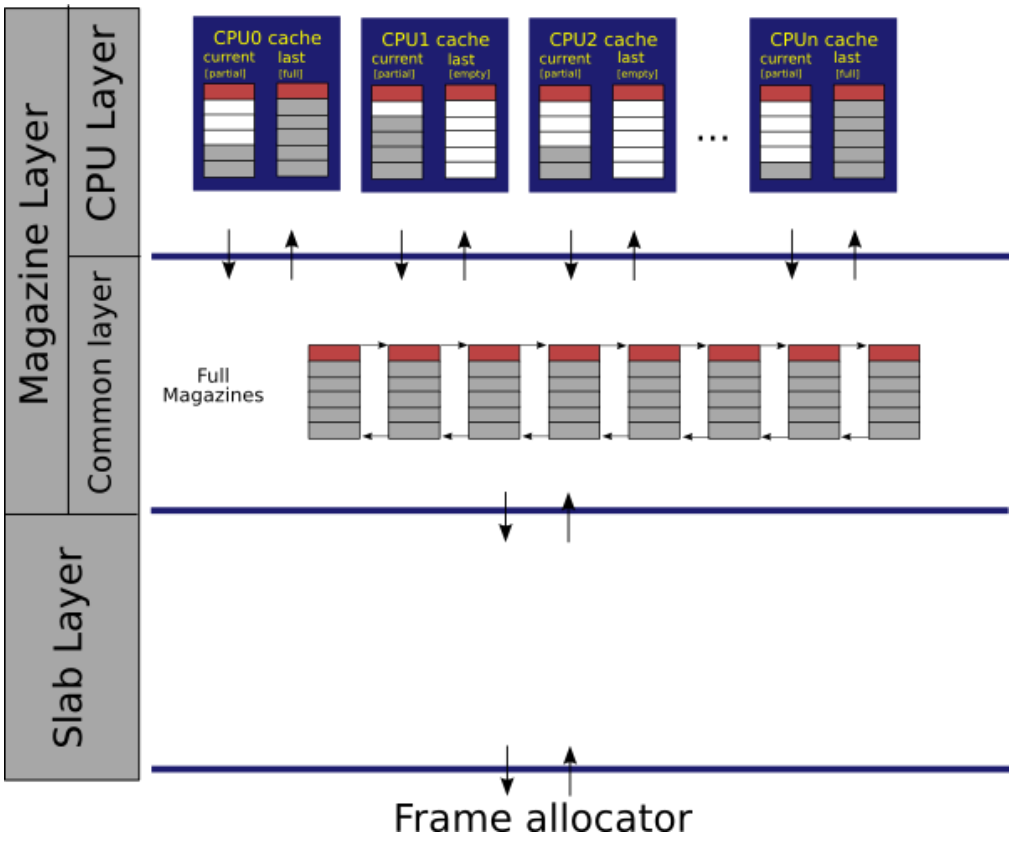


Figure 16: Slab allocator scheme.

Implementation

The slab allocator is closely modelled after with the following exceptions:

- empty slabs are immediately deallocated and
- empty magazines are deallocated when not needed.

The following features are not currently supported but would be easy to do:

- cache coloring and
- dynamic magazine grow (different magazine sizes are already supported, but the allocation strategy would need to be adjusted).

Allocation/deallocation

The following two paragraphs summarize and complete the description of the slab allocator operation (i.e. slab_alloc() and slab_free() functions).

Allocation. *Step 1.* When an allocation request comes, the slab allocator checks availability of memory in the current magazine of the local processor magazine cache. If the available memory is there, the allocator just pops the object from magazine and returns it.*Step 2.* If the current magazine in the processor magazine cache is empty, the allocator will attempt to swap it with the last magazine from the cache and return to the first step. If also the last magazine is empty, the algorithm will fall through to Step 3.*Step 3.* Now the allocator is in the situation when both

magazines in the processor magazine cache are empty. The allocator reloads one magazine from the shared list of full magazines. If the reload is successful (i.e. there are full magazines in the list), the algorithm continues with Step 1.*Step 4*. In this fail-safe step, an object is allocated from the conventional slab layer and a pointer to it is returned. If also the last magazine is full, **Deallocation.** *Step 1*. During a deallocation request, the slab allocator checks if the current magazine of the local processor magazine cache is not full. If it is, the pointer to the objects is just pushed into the magazine and the algorithm returns. *Step 2*. If the current magazine is full, the allocator will attempt to swap it with the last magazine from the cache and return to the first step. If also the last magazine is empty, the algorithm will fall through to Step *3*. *Step 3*. Now the allocator is in the situation when both magazines in the processor magazine cache are full. The allocator tries to allocate a new empty magazine and flush one of the full magazines to the shared list of full magazines. If it is successfully, the algoritm continues with Step 1. *Step 4*. In case of low memory condition when the allocation of empty magazine fails, the object is moved directly into slab. In the worst-case object deallocation does not need to allocate any additional memory.

9.7 Summary

Computer has main memory of RAM. Various architectures enable various uses of such memory, internally, memory could be accessed in different ways. Processes cannot run unless their code and data structures are in the RAM. It is in the main memory where instructions reside and are interpreted by the processor.

Some important issues related to the need for memory management include :

- Many times an operating system manages many process (multi-programming).
- The code and data for a process must be in RAM before it could be run.
- Process must not be able to access the code and data of other processes without permission. It means that the processes must be protected.
- Processes must be able to access and share the code and data of other processes if they have permission.
- There is usually not enough RAM to hold the code and data for all the currently running processes in RAM.

Self-Assessment Questions

1) What is address space? Difference logical and physical spaces.
2) What is the difference between static relocation and dynamic relocation?

3) What is segmentation? Explain address translation in segmentation.
4) What is memory fragmentation? What are the internal and External memory fragmentations.
5) In the paging scheme, what are pages, frame and page table?
6) Explain how page table is used in address translation.
7) Can we use compaction to solve internal fragmentation problem? Justify your answer.
8) Explain the buddy system for space allocation. What kind of memory fragmentation does it induce?
9) Explain briefly the role of the compiler, loader, and memory management hardware in the following address binding schemes:
    a. compile time binding
    b. load time binding
    c. runtime binding
10) What is dynamic memory management? Why is it used? Which part of the Kernel is stored there?
11) Can we precisely restrict a process within its address space in segmentation, paging and paged segmentation schemes? justify your answer.
12) Explain slab allocator and how it is implemented.
13) What kind of fragmentations (external or internal) do the following memory management schemes have? In each case. justify your answer.
    a. Paging
    b. Segmentation with (first fit, best fit and buddy system)
    c. Paged segmentation
14) What is Swapping and how it is implemented by virtual memory.
15) Explain memory hierarchy in terms of memory management in operating system.

Unit 10: Introduction to Paging, Segmentation and Segmentation with Paging

10.0 Objective

- Introduction to paging
- To provide a detailed description of the segmentation and segmentation with paging.
- Structure of page table etc.

10.1 Introduction

Virtual Memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory for programmers when physical memory is smaller in size. This makes the task of programmers very easy. In addition to this, VM also allows sharing of files and memory between several processes and makes implementation of memory protection much easier. Virtual memory can be implemented by one of the following techniques:

1. Paging
2. Segmentation
3. Segmentation with Paging

Paging

Paging is a memory management scheme in which memories as well as processes are divided into fixed size of blocks. Paging is the schema that permits the logical address space of a process to be noncontiguous. Physical memory broken into fixed sized blocks is called Page frames. Logical memory address space/ process is also broken into blocks of same size as page frame called Pages. When a process is to be executed, its pages are loaded into any available

memory frames from secondary storage. The secondary storage is also divided into fixed size blocks that are of the same size as the memory frames.

The page size is defined by the hardware and typically a power of two. The size of page typically lies between 512 bytes and 16 MB, depending upon computer architecture.

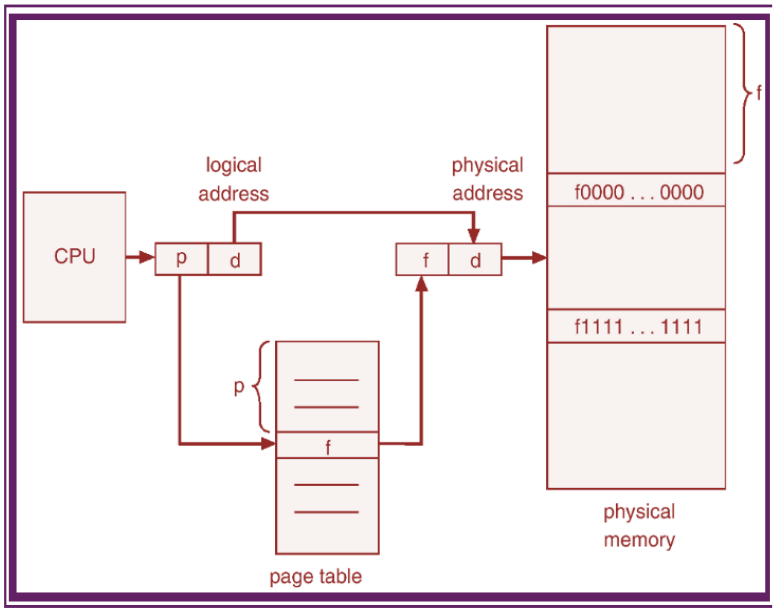


Figure 10.1: Address Translation Architecture

Page Table

The operating system stores the address translation tables for mapping virtual address to physical address in a data structure known as a page table. Figure 10.1 : Explain the address translation.

- Page table is kept in main memory.
- Page-table base register (PTBR) points to the page table.
- Page-table length register (PTLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory
- Address generated by CPU is divided into: Page number (p) - used as an index into a page table which contains base address of each page in physical memory. Page offset (d) - combined with base address to define the physical memory address that is sent to the memory unit.

If the size f the logical address space is 2m, and a page size is 2n addressing units (bytes or words), then the high order m-n bits of a logical address designate the page number, and the

n low order bits designate the page offset.

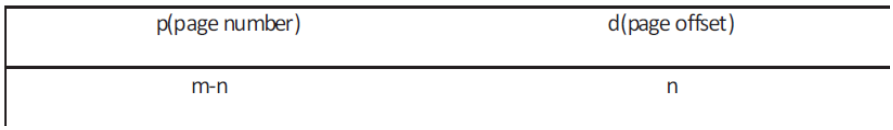


Figure 10.2: Address Generation

10.2 Segmentation

Segment is a logical entity about which the programmer is aware of A segment may contain a procedure or an array, stack or a collection of scalar variables. Usually it does not contain a mixture of different type of entities. Each segment has a name and length. To specify address in this segmented memory or two-dimensional memories, the program needs to supply a two part address, a segment number and address within the segment called offset. The number of segments present in the system is computer architecture dependent.

Some important properties of segments are

- Each segment consists of a linear sequence of addresses, from zero to maximum.
- The length of each segment may be anything from 0 to maximum.
- Different segment may have different length.
- The segment length may change during execution.
- The OS maintains a free list and allocate segments to memory holes.
- Simplifies modification and recompilation of procedures and facilitate sharing of procedure and data.

A segment table is very similar to the page table. A segment table entry must have fields for storing the segment's starting address in main memory and the size of the segment. If the maximum segment size is m bits, then last m bits of the logical address specify the segment offset. The remaining bits specify the segment number. The logical address is translated into a physical address by extracting the segment number and offset from the logical address. The segment number is used as an index into the segment table. The offset is compared to the segment size. If the offset size is greater than the segment size, invalid address faults is generated, and abort the program. Otherwise, the offset is added to the segment's starting address to generate the physical address. To increase speed, the size check and physical address generation can be performed concurrently.

Because segments are user defined, it is possible to define certain segments to be read only. By

adding a read only bit into a segment table entry, the memory management system can check for write operations to read only segments and generate a fault if such an operation is detected.

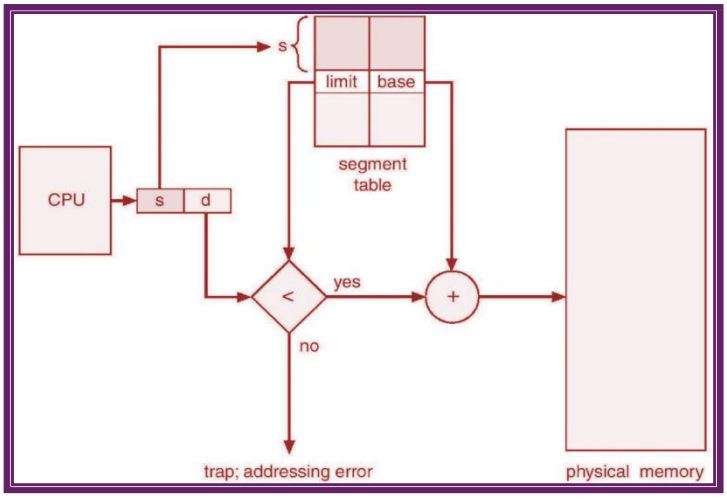


Figure 10.3: Segmentation

Shared segments may also be used by processes executing different programs but using the same subroutine library. In this situation, care must be taken that addresses within the shared segment will work with both programs. For addresses to locations within the segment, the simplest solution is to use relative addressing. Relative addressing uses the offset from the current value of the program counter to generate the destination address. For all addresses, indirect addressing through a register pointing to the appropriate segment is also a possibility. Direct addressing, which specifies the segment and offset of the destination address, is possible only if all programs use the same segment number for the segment being accessed.

10.3 Segmentation with Paging

Segmentation can be combined with paging to provide the efficiency of paging with the protection and sharing capabilities of segmentation. When paging is added in the segmentation, the segment offset is further divided into a page number and a page offset. The segment table entry contains the address of the segment's page table. The hardware adds the logical address's page number bits to the page table address to locate the page table entry. The physical address is formed by appending the page offset to the page frame number specified in the page table entry.

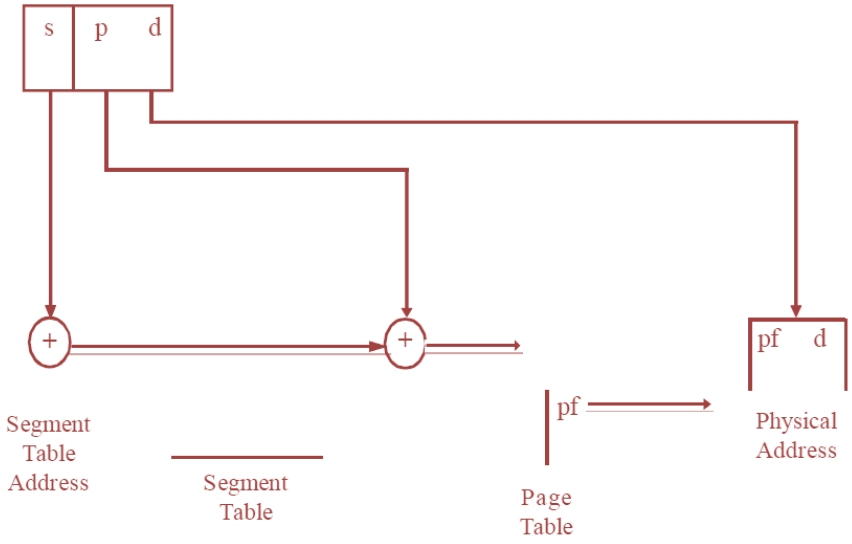


Figure 10.4: Segmentation with paging

10.4 Basic Hardware Support

In the simplest case page table is implemented with the help of dedicated registers. Page table is kept in main memory and a page table base register (PTBR) points to the page table. The problem is there with this approach is much time require to access a user memory location. If we want to access some location i, we must get index into the page table from the PTBR offset. This task requires a memory access. Now we get the page number from the page table and combine with the page offset to produce the actual address. With this scheme two memory accesses are needed to access a byte.

The most common solution to this problem is to use a special small fast lookup hardware cache called Translation Look-aside Buffer (TLB). The TLB is associative high-speed memory. Each entry in the TLB consists of two parts a key and a value. If the item is found the corresponding value field is returned. TLB search is fast but hardware is expensive. Generally, the number f entries in TLB are small.

TLB contains only few of the page table entries. When a logical address is generated by the CPU, its page number is given to the TLB. If the page number is found its frame number is immediately available and used to access required memory. If the page number is not in the TLB (TLB miss) a memory reference to the page table is used to get the frame number from the page table. This frame can be used to access the main memory. This page number and frame number is now added to the TLB for the future reference. If the TLB is full then OS must replace one of the entry from the TLB. Some entries from the TLB cannot be removed are wired down.

Some TLBs stores Address Space Identifiers (ASIDs) in each entry of the TLB. An ASID uniquely identifies each process and is used to provide address space protection for that

process.

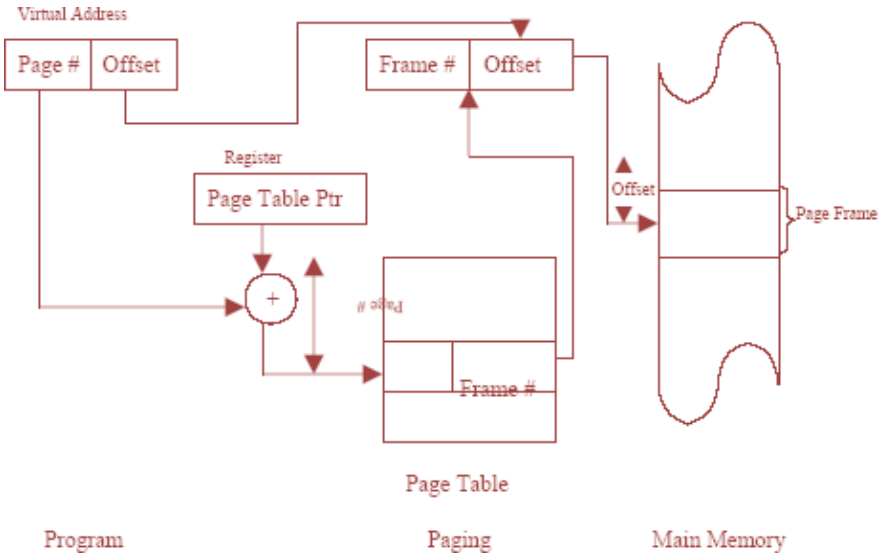


Figure 10.5: Paging with Translation Look-aside Buffer

HIT Ratio: The percentage of time that a particular page number is fond in the TLB is called the hit ratio. Let

Hit Ratio=h => Miss Ratio=(1-h)

TLB search takes ts seconds

Memory access time tm

seconds

If page is in TLB (Hit)

Memory access time = (ts +

tm) If page is not in TLB (miss)

Memory access time=(ts + 2tm)

(tm to read page from memory ts to read data)

Effective access time= (ts + tm)h + (ts + 2tm)(1-h)

10.5 Structure of Page Table

The basic mechanism for reading a word from memory involves the translation by using a page table of a virtual, or logical, address that consists of page number and offset, into a physical address that consists of a frame number and offset. Because the page table is of variable length, depending on the size of the process, it cannot be stored in registers. Instead, it must be in main memory to be accessed. Figure 10.6 suggests a hardware implementation of this scheme. When a particular process is running, a register holds the starting address of the page

table for that process. The page number of a virtual address is used to index that table loop up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address. Let us consider the number of page table entries required. In most systems, there is one page table per process. The entire page table may take up too much main memory. So Page tables are also stored in virtual memory. When a process is running, part of its page table is in main memory.

An alternative approach is the use of an Inverted Page Table structure. In this approach, the page number portion of a virtual address is mapped into a hash table by using a simple hashing function. The hash table contains a pointer to the inverted page table, which contains the page table entries. With this structure, there is one entry in the hash table and inverted page table for each real memory rather than for one per virtual page. Thus, a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages. Since more than one virtual address may map into the same hash table entry, a chaining technique is used for managing overflow.

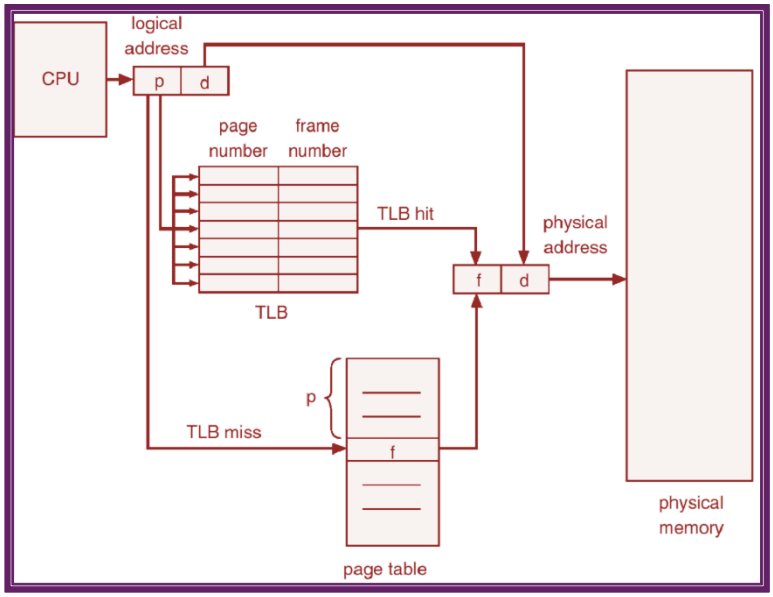


Figure 10.6: Address Translation in a Paging System

Page Size

An important hardware design is the size of page to be used. Several factors to be included

- Small page size, less amount of internal fragmentation
- Small page size, more pages required per process
- More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory.
- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better.

Based on the principle of locality

- Small page size, large number of pages will be found in main memory.
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

10.6 Hierarchical Paging

Modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 NIB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

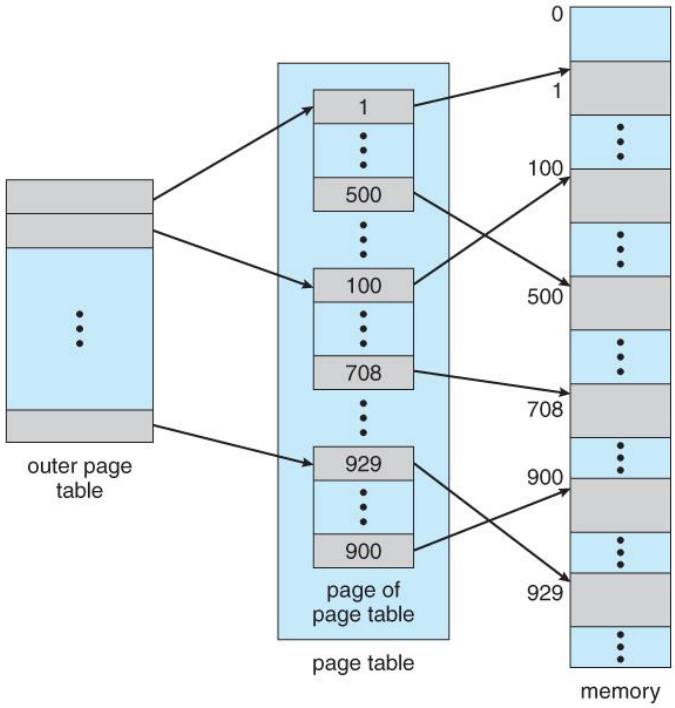


Figure 10.7: A two level page table scheme

One way is to use a two-level paging algorithm, in which the page table itself is also paged Remember our example of a 32-bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.

Because we page the page table, the page number is further divided into a la-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

Page number		Page offset
P_1	P_2	P_3
10	10	12

Figure 10.8: Two Level Page Structure

where P1 is an index into the outer page table and P2 is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure 10.8 Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

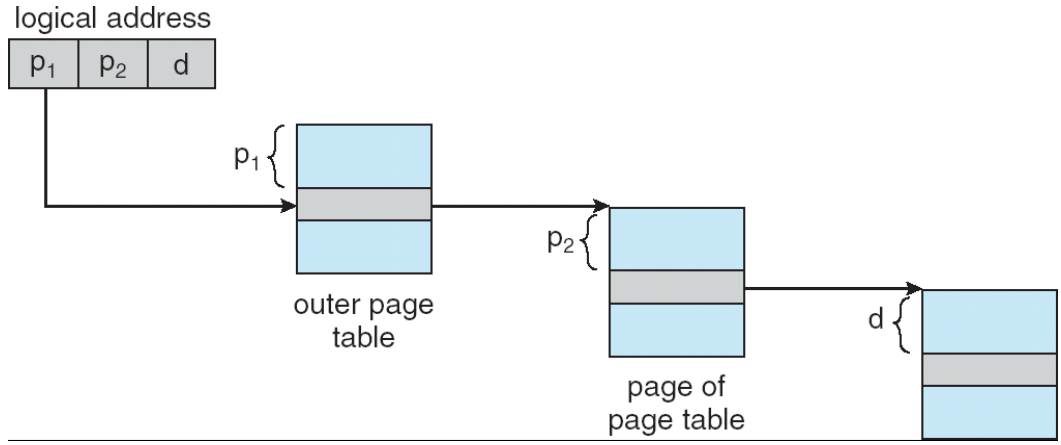


Figure 10.9: Address Translation for a two level 32 bit paging architecture

10.7 Hashed Paging

A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired

physical address.

If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 8.16. A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation 1JSeS clustered page tables, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.

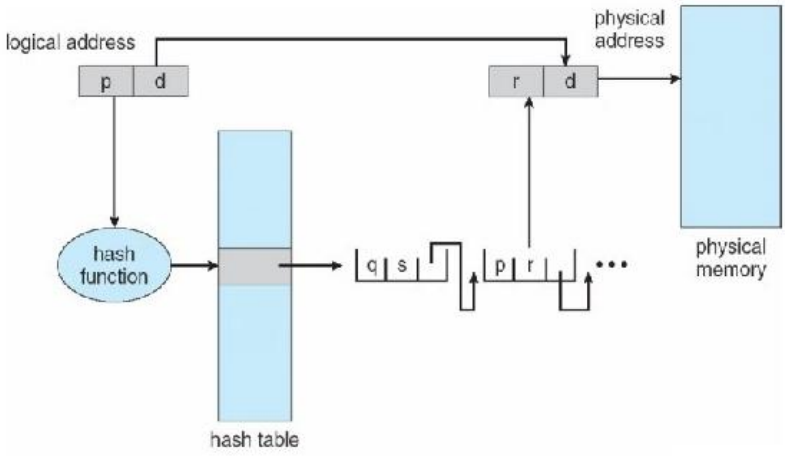


Figure 10.10: Hashed page table

10.8 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used. To solve this problem, we can use an inverted page table. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location; with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Below figure the operation of an inverted page table. **Compare**

it with Figure, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

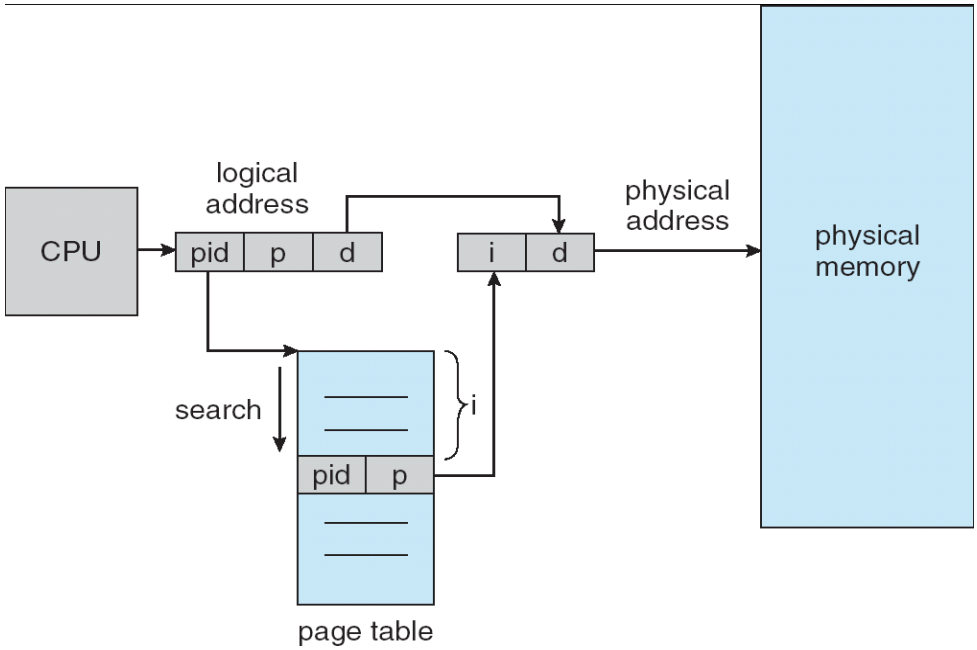


Figure 10.11: Inverted page table architecture

10.9 Summary

One of the most important, and most complex, tasks of an operating system is Memory Management. Memory management involves treating of main memory as a resource to be allocated to and shared among a number of active processes.

To efficiently use the processor and the I/O facilities, it is desirable to maintain as many processes in main memory as possible. In addition, it is desirable to free programmers from size restrictions in program development. The way to address both of these concerns is to use Virtual Memory. With Virtual memory all address references are logical references that are translated at run time to real address.

This use allows a process to be allocated anywhere in main memory and for that location to change over time. Virtual memory also allows a process to be broken up into pieces. These pieces need not be contiguously located in main memory during execution, and indeed all pieces of the process need not be in main memory.

Two basic approaches to provide virtual memory are Paging and Segmentation. With Paging, each process is divided into relatively small, fixed-size pages. Segmentation provides for the use

of pieces of varying size. It is also possible to combine segmentation and paging in a single memory management scheme.

Self Assessment Question

1. Define Memory Management.
2. List the requirements of memory management.
3. State the advantages of logical organization.
4. Define Overlaying.
5. What do you mean by Virtual memory?
6. Differentiate Internal fragmentation and External fragmentation.
7. State the technique used to overcome external fragmentation.
8. Bring out the difference between First fit, Best fit and Next fit.
9. Differentiate Logical address and Physical address.
10. Define: pages and frames.
11. State the advantages and disadvantages of Paging.
12. Define Segmentation.
13. Write down the steps for address translation in segmentation.

Unit 11: Virtual Memory Management

11.0 Objective

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

11.1 Introduction

As we know that a computer is designed for performing the multiple tasks at a time and for this some memory is also used by the computer for executing the instructions those are given by the user. But when there is a situation when the memory which is required by the user is high from the available memory. So that the Logical Memory will be treat as the Permanent Memory or from the Physical Memory and when we wants to display the size of Logical Memory big enough which is not actually exists. So at that situation we will use the concept of Virtual Memory.

In virtual memory, the physical memory is treated as the logical memory. Thus with the help of virtual memory we can increase the size of logical memory as from the physical memory. A user will see or feels that all the programs are running into the Logical Memory of the computer. With the help of virtual memory all the space of hard disk can be used as the Logical Memory So that a user can execute any number of programs.

The various benefits of the virtual Memory are :-

1. Unused Address space: With the help of unused address space a user can execute any number of programs because all the actual addresses will be treated as the logical

addresses. All the programs those are given by the user will be stored into the disk space and all the programs will be stored into the physical address space but they will treat as they are stored into the logical address space.

2. Increased degree of multiprogramming: With the help of virtual memory we can execute many programs at a time because many programs can be fit in the physical memory so that more programs can be stored into the memory but this will not increase the response time of the CPU means this will not affect the execution of the programs.
3. Decrease Number of I/O Operations: There will be less operations those are to be used for performing the swapping of the processes. All the programs will be automatically loaded into the memory when they are needed.

But always remember that the whole programs are never to be copied into the memory, it means all the programs are copied in form of pages or parts.

11.2 Pre-Paging and Demand Paging

When we execute a program it might be loaded from disk into memory. One option is to load the entire program in physical memory at the time of execution. However, a problem with this approach is that we may not initially need the entire program in memory. Consider a program that starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to initially load pages only as they are needed.

This technique is known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution, pages that are never accessed are thus never loaded into physical memory.

When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed.

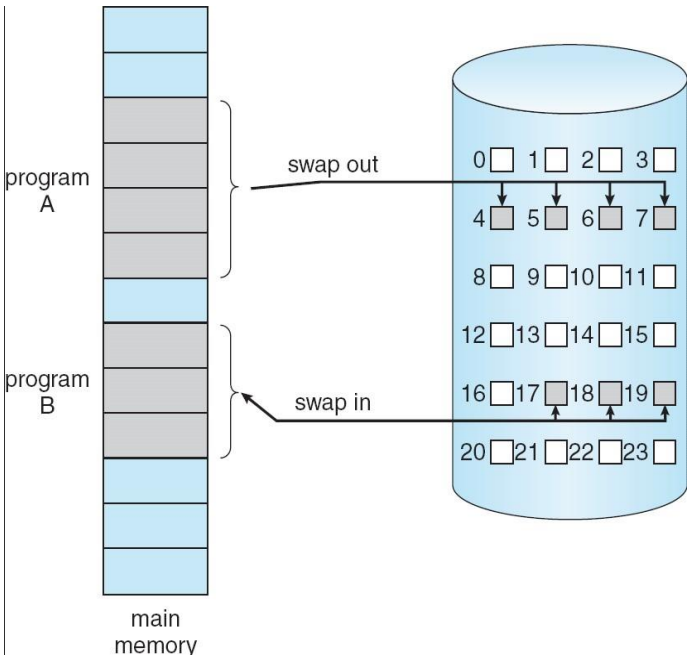


Figure 11.1: Transfer of a paged memory to contiguous disk space

Demand paging Concepts

When a process is to be swapped in, the pager guesses which pages is be used before the process is swapped out again. Instead of swapping in the whole process, the pager brings only necessary pages into memory. Thus it avoids reading into memory pages that will not be used any way, decreasing the swap time and the amount of physical memory needed. With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid -invalid bit scheme can be used for this purpose.

When this bit is set to "valid", the associated page is both legal and in memory. If the bit is set to "invalid", the page either is not valid or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is shown in figure 11.1

Marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

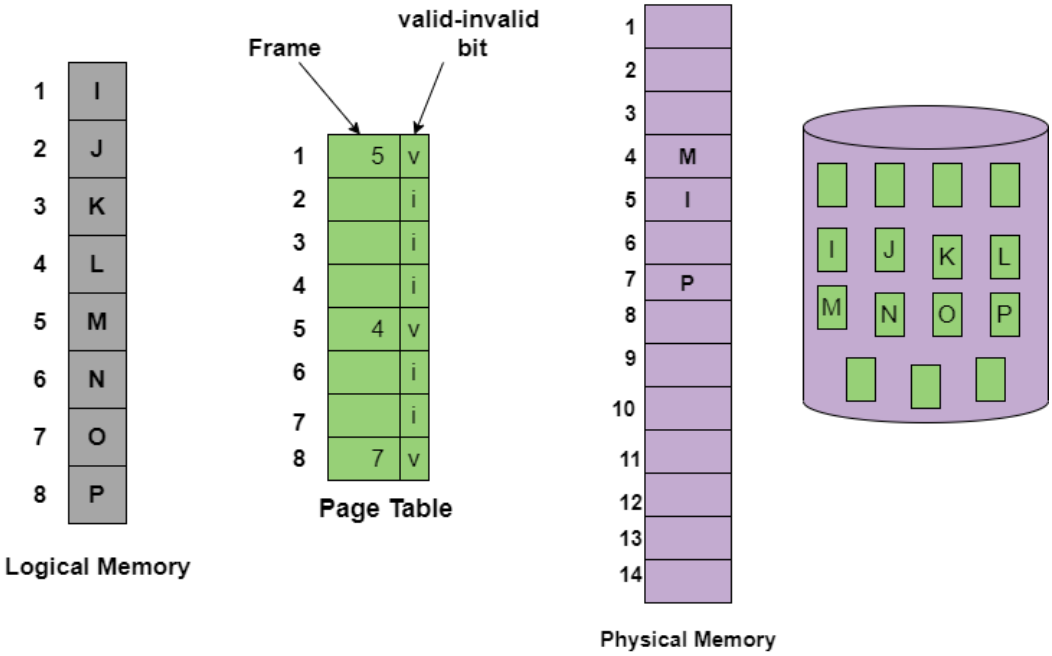


Figure 11.2: Page table when some pages are not in main memory

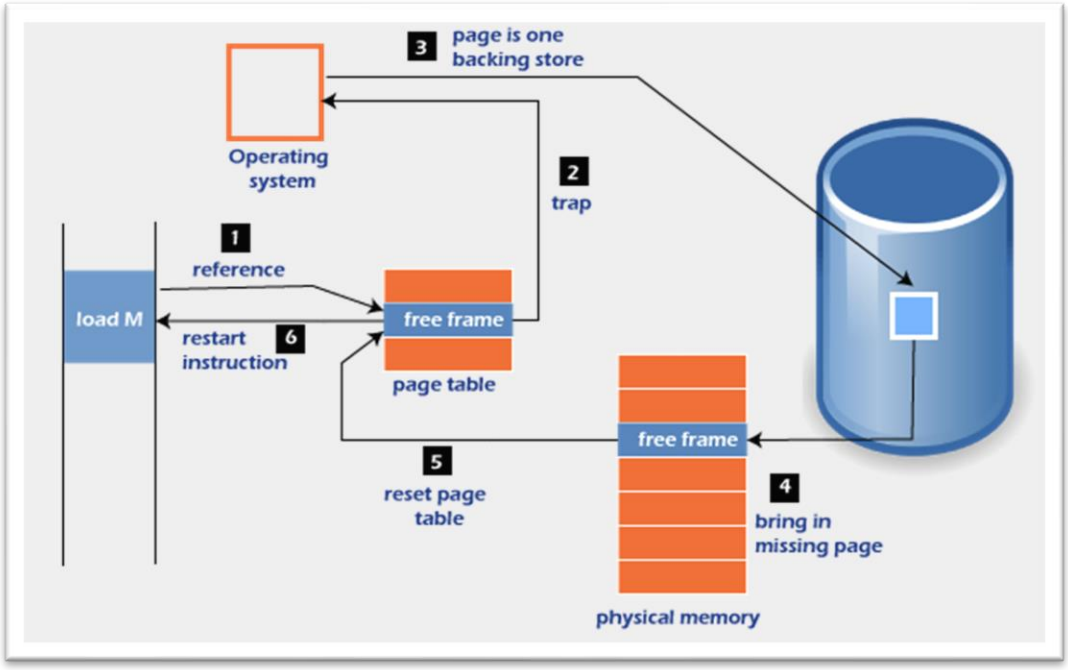


Figure 11.3: Steps in handling a page fault

11.3 Copy-on-write

Copy-on-write is a technique which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-an-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

Copy-on-write is shown in below figures 11.4, which show the contents of the physical memory

before and after process 1 modifies page C.

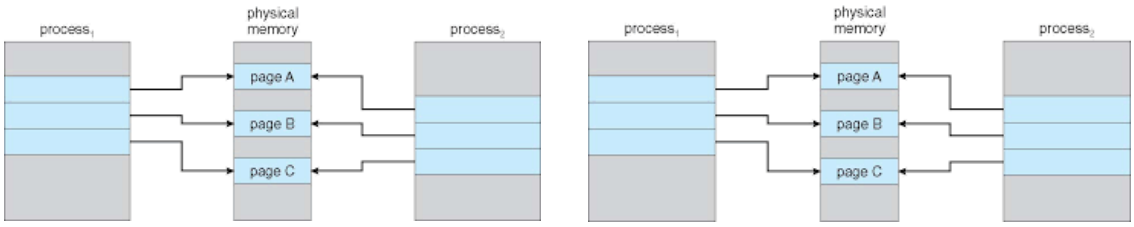


Figure 11.4(a) Before process 1modifies page C (b) after process1modifies page C

When child process attempts to modify a page containing portions of the stack, with the pages set to be copy-an-write. The operating system will then create a copy of this page, mapping it to the address space of the child process.

The child process will then modify its copied page and not the page belonging to the parent process.

When the copy-on-write technique is used, only the pages that are modified by either process are copied, all unmodified pages can be shared by the parent and child processes. Only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified can be shared by the parent and child. Copy-on-write is a technique used by several operating systems including windows XP, Linux, and Solaris.

Operating systems typically allocate these pages using a technique known as zero-fill-on-demand. Zero- fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

11.4 Page Replacement Basics

Consider a situation where a user process is under execution and a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use.

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput.

Page replacement takes the following approach. If no frame is free, we find one that is currently not being used and free it. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process fault occurred.

1. Find the location of the desired page on the disk.

2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    c. Write the Victorian frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, user addresses are mapped into physical addresses, so the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

11.5 Page Replacement Policies

A reference to a page, not present in main memory, is called page fault. When page fault occurs, it is necessary for OS to read in the required page from the secondary memory. Removing the page randomly will not be a good idea hence we need to have page replacement algorithm.

Page replacement algorithms decide what pages to page out (swap out) when a page needs to be allocated. This happens when a page fault occurs and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in, and this usually involves waiting for I/O completion. This determines the quality of the page replacement algorithm: the less time wasted by waiting for page-ins, the better the algorithm. A page replacement algorithm tries, by looking at the limited information about accesses to the pages provided by hardware, to guess what pages should be replaced in order to minimize the total number of page misses, while balancing this with the costs (primary storage

and processor time) of the algorithm itself.

There are various page replacement algorithms available.

Some of the very important and common algorithms are discussed:

1. FIFO: The first in first out page replacement algorithm selects the page that has been in memory the longest .To implement this algorithm page table entries must include a field for the swap in time. When a page is swapped in the OS loads the field with current time. The page selected for replacement will be the one with the earliest swap in time. Although easy to implement and inexpensive FIFO is not very efficient. Frequently used pages even though they are in memory should not be swapped out. FIFO doesn't consider the amount they have been used and swaps them out anyway.

Example:

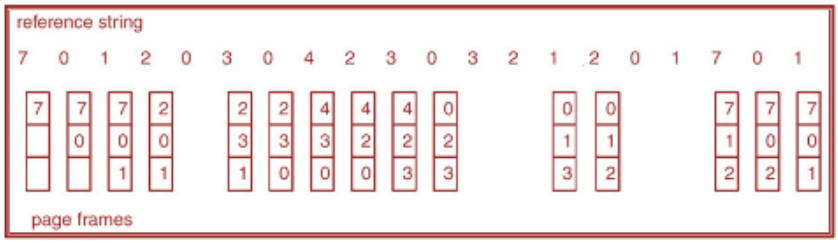


Figure 11.5 FIFO Page - replacement algorithm

2. LRU: The least recently used algorithm keep tracks of the last time each page was used not when it was swapped in. The memory management hardware uses a counter which is incremented during each memory reference. Each page table entry has a field that stores a value of this counter. When a page is referenced the value of the counter is updates in the page table entry for that page.

LRU replacement associates with each page the time of that page's last used. When a page must be replaced LRU chooses that page which has not been used for the longest period of time. This strategy is the optimal page replacement algorithm looking backward in time rather than forward.

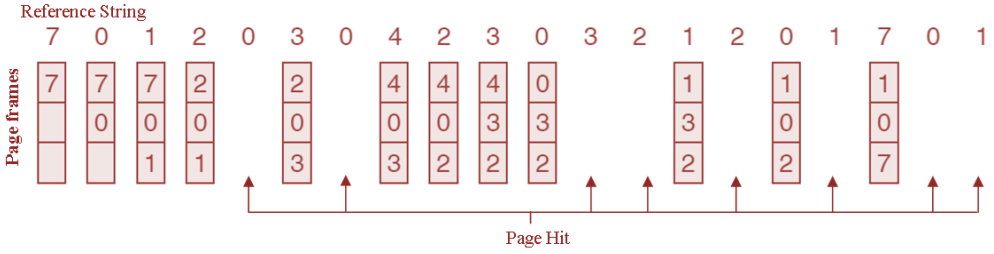


Figure 11.6

3. Second chance page replacement algorithm: The basic algorithm of second chance

replacement is a FIFO replacement algorithm. When page has been selected, we inspect its reference bits. If the value is 0, we proceed to replace this page and the reference bit is set to 1, however we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance its referenced bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced. In addition if a page is used often enough to keepits reference bit set, it will never be replaced.

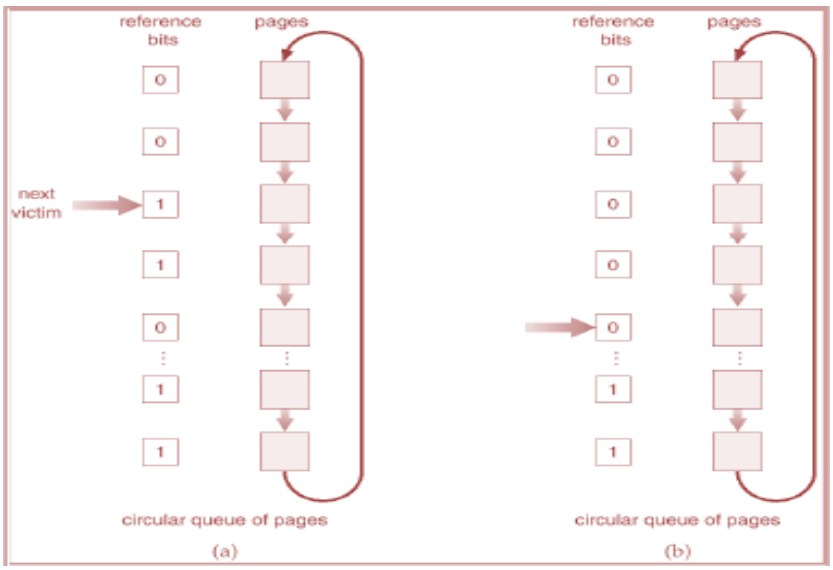


Figure 11.7: Second chance page replacement algorithm

11.6 Thrashing cause

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling. In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page.

However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

11.6.1 Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is

based on the actual behavior of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their tilrle paging.

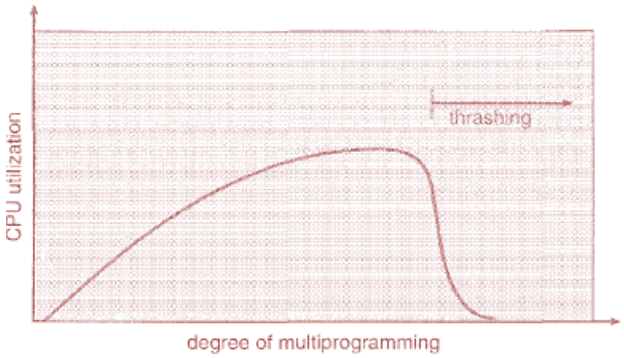


Figure 11.8

This phenomenon is illustrated in Figure 11.8, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the

paging device HlOSt of the time. The average service time for a page fault will increase because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which mayoverlap.

For example, when a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

11.7 Summary

- Virtual memory is commonly implemented by demand paging. Pure demand paging never brings in a page until that page is referenced. The first reference causes a page fault to the operating system. The operating-system kernel consults an internal table to determine where the page is located on the backing store. It then finds a free frame and reads the page in from the backing store.
- The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even tll0ugh its entire memory image is not in main memory at once. As long as the page-fault rate is reasonably low, performance is acceptable.
- We can use demand paging to reduce the number of frames allocated to a process. This arrangement can increase the degree of multiprogramming (allowing more processes to be available for execution at one time) and -in theory, at least-the CPU utilization of the system. It also allows processes to be run even though their memory requirements exceed the total available physical memory. Such processes run in

virtual memory.

- If total memory requirements exceed the physical memory, then it may be necessary to replace pages from memory to free frames for new pages. Various page-replacement algorithms are used. Optimal page replacement requires future knowledge. LRU replacement is an approximation of optimal page replacement, but even it may be difficult to implement. Most page-replacement algorithms, such as the second-chance algorithm, are approximations of LRU replacement.
- Most operating systems provide features for memory mapping files, thus allowing file I/O to be treated as routine memory access. The Win32 API implements shared memory through memory mapping files.

Self-Assessment Questions

1. What do you mean by virtual memory?
2. What is demand paging
3. What is page table, how you will determine size of a page
4. What is Thrashing.
5. Write any two-page replacement policies

Unit 12: File Management System

12.0 Objective

- To explain the function of file systems.
- To describe the interfaces to file systems.
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- To explore file-system protection.

12.1 Introduction

File system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, 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. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

12.2 File Attribute

Each file is stored in a directory and uses a directory entry that describes its characteristics such as its name and size. The directory entry also contains a pointer to where the file is stored on disk. One of the characteristics stored for each file is a set of file attributes that give the operating system and application software more information about the file and how it is intended to be used.

Any software program can look in the directory entry to determine the attributes of a file, and based on them, make decisions about how to treat the file.

A file can have more than one attribute attached to it they may be vary from one operating system to another but typically consist of these:

A file can have following attributes.

- **NAME:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This is a number or tag which is used to identified the file with in the file system.
- **Type:** this information is needed for system to support different types of file.
- **Size:** The size of file in bytes.
- **Read-Only:** Most software, when seeing a file marked read-only, will refuse to delete or modify it. This is pretty straight-forward. For example, DOS will say "Access denied" if you try to delete a read-only file. On the other hand, Windows Explorer will happily munch it. Some will choose the middle ground: they will let you modify or delete the file, but only after asking for confirmation.
- **Hidden:** if the file is marked hidden then under normal circumstances it is hidden from view. DOS will not display the file when you type "DIR" unless a special flag is used, as shown in the earlier example.
- **System:** This flag is used to tag important files that are used by the system and should not be altered or removed from the disk. In essence, this is like a "more serious" read-only flag and is for the most part treated in this manner.
- **Volume Label:** Every disk volume can be assigned an identifying label, either when it is formatted, or later through various tools such as the DOS command "LABEL". The volume label is stored in the root directory as a file entry with the label attribute set.
- **Directory:** This is the bit that differentiates between entries that describe files and those that describe subdirectories within the current directory. In theory you can convert a file to a directory by changing this bit. of course in practice, trying to do this would result in a mess--the entry for a directory has to be in a specific format.

- **Archive:** This is a special bit that is used as a "communications link" between software applications that modify files, and those that are used for backup. Most backup software allows the user to do an incremental backup, which only selects for backup any files that have changed since the last backup. This bit is used for this purpose. When the backup software backs up ("archives") the file, it clears the archive bit (makes it zero). Any software that modifies the file subsequently, is supposed to set the archive bit. Then, the next time that the backup software is run, it knows by looking at the archive bits which files have been modified, and therefore which need to be backed up. Again, this use of the bit is "voluntary"; the backup software relies on other software to use the archive bit properly; some programs could modify the file without setting the archive attribute, but fortunately most software is "well-behaved" and uses the bit properly. Still, you should not rely on this mechanism absolutely to ensure that your critical files are backed up.

12.3 File Operations

We can perform various operation on a file. a operation system have various system call to create, write, read, move, delete a file.

- **Creating a file.** Two steps are necessary to create a file. Space in the file system must be found for the file. An entry for the new file must be made in the directory.
- **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. The system needs to keep a read pointer to the location in the file where the next read is to take place.
  Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position pointer.
  Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file.** The directory is searched for the appropriate entry, and the current-file- position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file seek.
- **Deleting a file.** To delete a file, we search the directory for the named file. Having

found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

- **Truncating a file.** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged (except for file length) but lets the file be reset to length zero and its file space released.

12.4 File Types

The types of files recognized by the system are either regular, directory, or special. However, the operating system uses many variations of these basic types.

The following basic types of files exist:

regular	Stores data (text, binary, and executable)
directory	Contains information used to access other files
special	Defines a FIFO (first-in, first-out) pipe file or a physical device

All file types recognized by the system fall into one of these categories. However, the operating system uses many variations of these basic types.

- **Regular files**

  Regular files are the most common files and are used to contain data. Regular files are in the form of text files or binary files.

- **Text files**

  Text files are regular files that contain information stored in ASCII format text and are readable by the user. You can display and print these files. The lines of a text file must not contain NULL characters, and none can exceed {LINE_MAX} bytes in length, including the newline character.

  The term text file does not prevent the inclusion of control or other nonprintable characters (other than NULL). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters or they explicitly describe their limitations within their individual sections.

- **Binary files**

  Binary files are regular files that contain information readable by the computer. Binary files might be executable files that instruct the system to accomplish a job. Commands and programs are stored in executable, binary files. Special compiling programs translate ASCII text into binary code.

Text and binary files differ only in that text files have lines of less than {LINE_MAX} bytes, with no NULL characters, each terminated by a newline character.

- **Directory files**

  Directory files contain information that the system needs to access all types of files, but directory files do not contain the actual file data. As a result, directories occupy less space than a regular file and give the file system structure flexibility and depth. Each directory entry represents either a file or a subdirectory with reference to Unix each entry contains the name of the file and the file's index node reference number (i-node number). The i-node number points to the unique index node assigned to the file. The i-node number describes the location of the data associated with the file. Directories are created and controlled by a separate set of commands.

- **Special files**

  Special files define devices for the system or are temporary files created by processes. The basic types of special files are FIFO (first-in, first-out), block, and character. FIFO files are also called pipes. Pipes are created by one process to temporarily allow communication with another process. These files cease to exist when the first process finishes. Block and character files define devices.

  Every file has a set of permissions (called access modes) that determines who can read, modify, or execute the file.

12.5 File Structure

File types also can be used to indicate the internal structure of file.

The operating system support multiple file structures: The resulting size of the operating system is large. If the operating system defines five different file structures, it needs to contain the code to support these file structures.

Every file may need to be define as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may occur.

Some operating systems require a minimal number of file structures. This approach has been adopted in UNIX, MS-DOS, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support.

Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure that of an executable file so that the system is able to load and run programs.

12.6 Internal File Structure

Files can be structured in any of several ways. Three common possibilities are Stream of Bytes, Records, Tree of Records.

Stream of Bytes. The file is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Both UNIX and Windows use this approach.

Records. The first step up in structure is a file is a sequence of fixed-length records, each with some internal structure.

Tree of Records. The third kind of file structure organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record.

Internally, locating an offset within a file can be complicated for the OS.

Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Packing a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX OS defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks -say, 512 bytes per block- as necessary. The file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks. Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted.

The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

12.7 Accessing Method

Information is kept in files. Files reside on secondary storage. When this information is to be used, it has to be accessed and brought into primary main memory. Information in files could be accessed in many ways. It is usually dependent on an application.

Sequential Access: A simple access method, information in a file is accessed sequentially one

record after, another. To process the next record records previous to it must be accessed. Sequential access is based on the tape model that is inherently a sequential access device. Sequential access is best suited where most of the records in a file are to be processed. For example, transaction files.

Direct Access: Sometimes it is not necessary to process every record in a file. It may not be necessary to process records in the order in which they are present. Information present in a record of a file is to be accessed only if some key value in that record is known. In all such cases, direct access is used. Direct access is based on the disk that is a direct access device and allows random access of any file block. Since a file is a collection of physical blocks, any block and hence the records in that block are accessed.

For example, master files. Databases are often of this type since they allow query processing that involves immediate access to large amounts of information. All reservation systems fall into this category. Not all operating systems support direct access files. Usually, files are to be defined as sequential or direct at the time of creation and accessed accordingly later.

Sequential access of a direct access file is possible but direct access of a sequential file is not.

Indexed Sequential Access: This access method is a slight modification of the direct access method. It is in fact a combination of both the sequential access as well as direct access. The main concept is to access a file direct first and then sequentially from that point onwards. This access method involves maintaining an index. The index is a pointer to a block. To access a record in a file, a direct access of the index is made. The information obtained from this access is used to access the file. For example, the direct access to a file will give the block address and within the block the record is accessed sequentially. Sometimes indexes may be big. So hierarchies of indexes are built in which one direct access of an index leads to info to access another index directly and so on till the actual file is accessed sequentially for the particular record. The main advantage in this type of access is that both direct and sequential access of files is possible.

12.8 Directory Structure

File systems typically have directories (sometimes called folders) which allow the user to group files. This may be implemented by connecting the file name to an index in a table of contents or an inode in a Unix-like file system. Directory structures may be flat (i.e. linear), or allow hierarchies where directories may contain subdirectories. The first file system to support arbitrary

hierarchies of directories was the file system in the UNIX operating system. The native file systems of Unix-like systems also support arbitrary directory hierarchies, as do, for example, the FAT file system in MS-DOS 2.0 and later and Microsoft Windows, the NTFS file system in the Windows NT family of operating systems.

12.9 File Access and Access Control

Many operating systems, including Unix, are multiuser. As such, it is necessary to have mechanisms which protect a user's file from unwanted access. An integral part of an access control mechanism in a multiuser operating system is the concept of user ownership of a file and, very often, the concept of group ownership. However, many operating systems, such as those for personal computers, are single-user operating systems. In a single-user environment, there is no need for file access mechanisms which protect files from unwanted access by others because, conceptually, there is only one user. Nonetheless, some single-user operating systems permit a file to be marked as read-only to protect the file from inadvertent modification or removal.

Both single-user and multiuser operating systems have been used as servers providing transparent file access. Most multiuser operating systems are already well suited for use as a file server since access protection is already an integral part of its design. For a single-user operating system, it is necessary that some sort of access control mechanism be implemented when such an operating system is used to provide file service. This is usually accomplished by creating a separate partition for each user's files. Thus, user identification is not associated with an individual file but with a partition on the server's disk. File servers which have single-user operating systems may also have partitions that are accessible by anyone.

Network environments include both servers based on multiuser operating systems and servers based on single-user operating systems. Application programs on clients may have transparent access to servers which are based on single-user operating systems which may neither provide owner information nor group ownership information for a file but are capable of imposing some level of file access control. That file access control may be no more than a read/write permission that applies to any user. Avery large group of applications are able to function in such an environment.

12.10 Summary

A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line, or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.

The major task for the operating system is to map the logical file concept onto physical storage devices such as magnetic tape or disk. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.

Each device in a file system keeps a volume table of contents or device directory listing the location of the files on the device. In addition, it is useful to create directories to allow files to be organized.

A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.

Self-Assessment Questions

1. What do you mean by file attributes?
2. What are the different operation can be perform on a file?
3. Explain various file accessing methods.
4. Write a short note on file access control.

Unit 13: I/O System

13.0 Objective

After studying this unit, you will be able to understand the input-output subsystem. You will learn the principles of I/O hardware and their complexities. You will also study about working of input-output scheduling, spooling and error handling etc.

- Explore the structure of an operating system's I/O subsystem.
- Discuss the principles of I/O hardware and its complexity.
- Provide details of the performance aspects of I/O hardware and software.

13.1 Introduction

The control of devices connected to the computer is a major concern of operating-system designers. Because input output devices vary so widely in their function and speed (consider a mouse, a hard disk, and a cdrom), varied methods are needed to control them. These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the

complexities of managing I/O devices.

The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The device drivers present a uniform device access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

13.2 Overview I/O Hardware

Computers operate a with many kinds of I/O devices. like general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Here we will learn concepts to understand how the devices are attached and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point i.e. port. If devices use a common set of wires, the connection is called a bus. A bus is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.

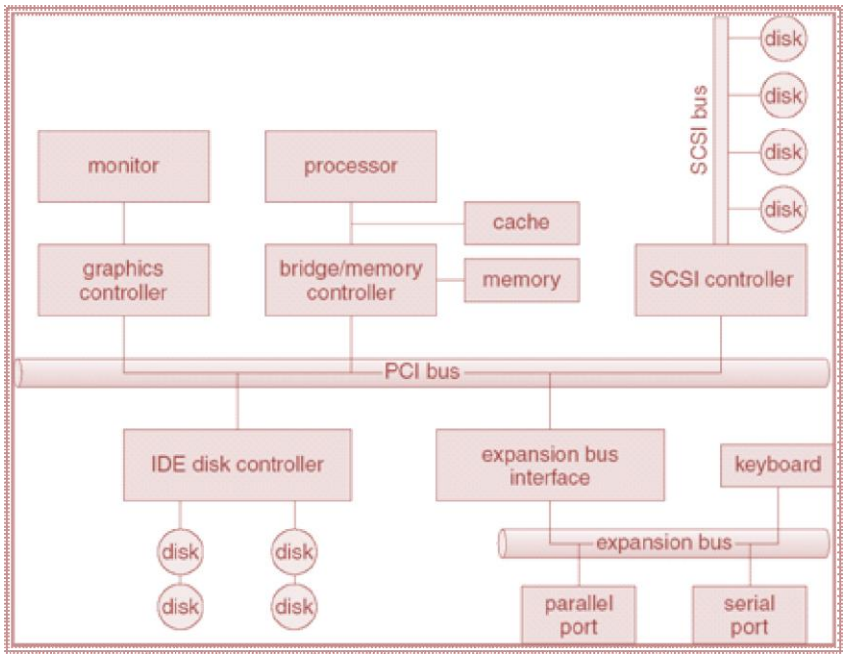


Figure 13.1: Bus Structure

Buses are used widely in computer architecture. A typical PC bus structure in the figure 13.1 shows a PCI bus (the common PC system bus) that connects the processor-memory subsystem to the fast devices and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are

connected together on a SCSI (Small Computer System Interface) bus plugged into a SCSI controller.

A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port.

The SCSI bus controller is often implemented as a separate circuit board that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controller.

The disk controller implements the disk side of the protocol for some kind of connection-SCSI or ATA (Advanced Technology Attachment), for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

The controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers.

The CPU executes I/O requests using the standard data-transfer instructions to read and write the device- control registers. An I/O port typically consists of four registers, called the (1) status, (2) control, (3) data-in, and (4) data-out registers.

- The status register contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex comn1unication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.
- The data-in register is read by the host to get input.
- The data-out register is written by the host to send output.

The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

13.2.1 Polling

Device polling refers to a technique that lets the operating system periodically poll devices, instead of relying on the devices to generate interrupts when they need attention. This might

seem inefficient and counterintuitive, but when done properly, polling gives more control to the operating system on when and how to handle devices, with a number of advantages in terms of system responsiveness and performance.

In particular, polling reduces the overhead for context switches which is incurred when servicing interrupts and gives more control on the scheduling of the CPU between various tasks (user processes, software interrupts, device handling) which ultimately reduces the chances of live lock in the system.

13.2.2 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the interrupt- request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the

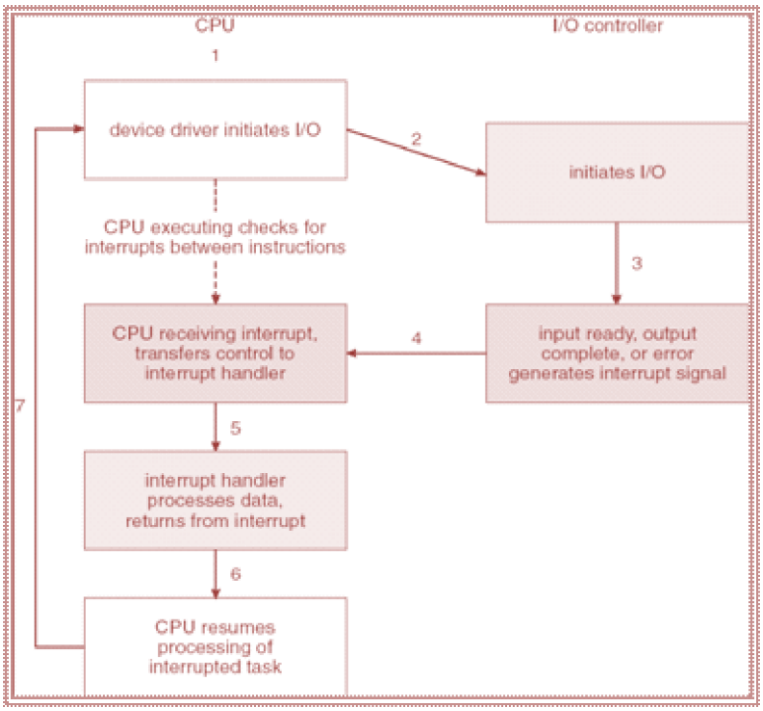


Figure 13.2: Interrupt I/O cycle

Interrupt request line, the CPU performs a state save and jumps to the interrupt handler routine at a fixed address in memory. Title interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device. Figure summarizes the interrupt-driven I/O cycle.

13.2.3 Direct Memory Access

I/O devices such as hard disks write to memory all the time, and with a normal system setup, their requests go through to the CPU first and then the CPU reads/writes the data from/to the memory sequentially. When such a request involves polling/busy wait loops, it is known as programmed I/O. The problem with polling is that it wastes a lot of CPU resources as it sits in a tight loop checking for changes in values. The second method for I/O devices to access memory is called interrupt-driven. The problem now is that servicing interrupts is expensive, in the sense that when an interrupt signal is sent by an I/O device, before or after a memory access, the following has to be done every time:

- The data for the currently running process in the CPU is to be saved to the stack.
- An Interrupt Service Routine is allowed to handle the interrupt
- Data is sent back to the I/O controller
- The data for the previously running process is restored from the stack.

Such a method has a very large overhead. The state information for the current process has to be saved before the I/O operation can use the CPU, and then restored after it has finished.
A much better way of doing things would be to take away the CPU from the picture all together and have the I/O devices talking directly to the memory. This is where DMA (Direct Memory Access) comes in. The DMA controller sits on the shared system bus containing the memory and CPU and allows 7 I/O devices to connect to it. Once it has all the information it needs about a particular I/O device, such as the number of words (word = 16 bits) to transfer and the memory address of the first word of input or output (Memory Address Counter), the DMA controller then transmits the data directly to the memory via the shared system bus.
The advantages of DMA The most important is that the processor does not have to worry about I/O operations between computer peripherals and memory. Another advantage is the fact that transfers are much simpler since they do not require the CPU to execute specific instructions to do the transfer or have to deal with interrupts being signaled in from I/O devices. All the DMA controller needs is a start indicator to let it know when it can start transferring data to memory, a counter for how many words are left and a stop indicator to clean up at the end.

13.3 Application I/O Interface

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into device drivers, while application layers are presented with a common interface for all (or at least large general categories of)

devices.

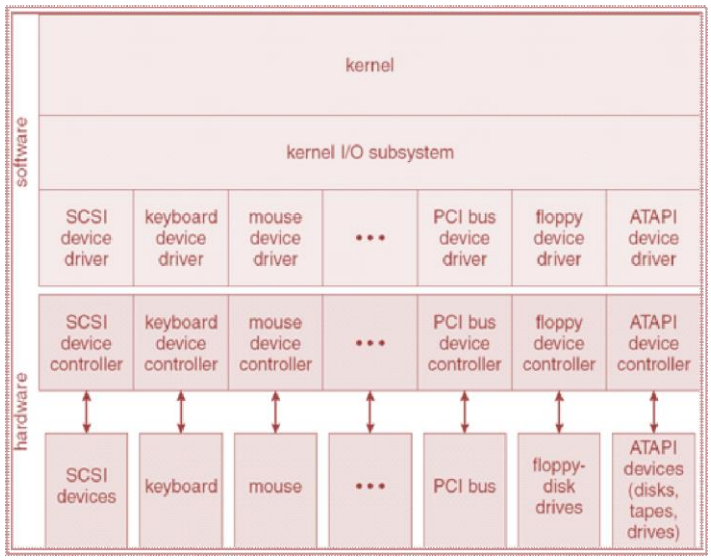


Figure 13.3 : Kernel I/O Structure

Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.

Most OS also have an escape, or back door, which allows applications to send commands directly to device drivers if needed. In UNIX this is the ioctl( ) system call (I/O Control). ioctl( ) takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.

13.3.1 Blocked Character Device

Block devices are accessed a block at a time and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read( ), write( ), and seek( ).

Accessing blocks on a hard drive directly (without going through the file system structure) is called raw I/O, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)

A new alternative is direct I/O, which uses the normal file system access, but which disables buffering and locking operations.

Memory-mapped file I/O can be layered on top of block-device drivers. Rather than reading in the entire file, it is mapped to a range of memory addresses, and then paged into memory as needed using the virtual memory system. Access to the file is then accomplished through

normal memory accesses, rather than through read( ) and write( ) system calls. This approach is commonly used for executable program code.

Character devices are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

13.3.2 Blocking & Non Blocking Input Output

With blocking I/O a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.

With non-blocking I/O the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.

One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen or perform other tasks.

A subtle variation of the non-blocking I/O is the asynchronous I/O, in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has completed, and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available but does not complete the operation and notify the process later.)

13.4 Kernel I/O Sub System

Kernels provide many services related to I/O. Several services-scheduling, buffering, caching, spooling, device reservation, and error handling-are provided by the kernel's I/O subsystem and build on the hardware and device driver infrastructure. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

13.4.1 Input Output Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete.

Operating-system developers implement scheduling by maintaining a wait queue of requests for

each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests.

One way in which the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations. Another way is by using storage space in main memory or on disk via techniques called buffering, caching, and spooling.

13.4.2 Buffering

A buffer is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk.

So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This double buffering decouples the producer of data from the consumer, thus relaxing timing requirements between them.

13.4.3 Caching

A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data.

These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory.

13.4.4 Spooling and Device Reservation

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.

The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In others, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, to remove unwanted jobs before those jobs print, to suspend printing while the printer is serviced, and so on.

13.4.5 Error handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons! as when a network becomes overloaded! or for "permanent" reasons! as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance! a disk read () failure results in a read () retry! And a network send () error results in a resend ()! If the protocol so specifies. Unfortunately! if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return one bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named error number is used to return an error code-one of about a hundred values-indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application. For instance, a failure of a SCSI device is reported by the SCSI protocol in three levels of detail: a sense key that identifies the general nature of the failure, such as a hardware error or an

illegal request; an additional sense code that states the category of failure, such as a bad command parameter or a self-test failure; and an additional sense-code qualifier that gives even more detail, such as which command parameter was in error or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host-but that seldomare.

13.4.6 I/O Protection

Errors are closely related to the issue of protection. A user process may accidentally or purposefully attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various mechanisms to ensure that such disruptions cannot take place in the system. To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system.

13.4.7 Kernel Data Structure

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file table structure from Section 11.1. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a read () operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perforn1 a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique.

13.5 Summary

The I/O system is modular; you can easily expand or customize it. The OS I/O system consists of the following software components:

- The Kernel
- File Managers
- Device drivers
- The device descriptor

The kernel, file managers, and device drivers process I/O service requests at different levels. The device descriptor contains information used to assemble the elements of a particular I/O subsystem. The file manager, device driver, and device descriptor modules are standard memory modules.

- Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
- I/O subsystems must contend with two (conflicting?) trends (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.
- Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

Self-Assessment Questions

1. What do you mean by I/O subsystem?
2. Explain Following.
   (i) Polling
   (ii) Interrupts
   (iii) Direct Memory Access
3. What do you mean by Kernel input output sub system? Explain following:
   (i) Buffering
   (ii) Spooling
   (iii) Catching

Unit 14: System Protection

14.0 Objective

After studying this unit, you will be able to understand the concept of protecting a system and its need and domain. You will also learn about different methods of enforcing protecting mechanisms like access rights, access matrix. This chapter provides a general overview of

- Protection concept & goal
- Principles of protection
- Domain of protection
- Protection mechanism : Access right, Access matrix

14.1 Introduction

Protection is strictly an internal problem. How do we provide controlled access to program & data stored in computer system. The various processes (user and system processes) must be protected from one other's activities. Protection mechanisms deal with controlling the access of programs, processes or users to the resources of a computer system. Security however, an issue that requires a control over the external environment within which the system operates, Internal protection is of no use if the computer system is accessed by an unauthorized individual who can remove some important files or insert a virus into the system and make it non-functional.

14.2 Goals of Protection

The protection mechanism to be enforced into a system should have the following goals:

- In multi-propgramming systems, the operating system should enable the users to safely share a common logical address space such as files etc. It should also enable the users to share a common physical address space i.e., memory. In such systems, therefore, the goal should be to prevent accidental and intentional destructive behavior.
- Ensure fair and reliable resource usage i.e., each program component active in a system should use the system resources only in accordance with certain policies (stated for the use of these resources). Since the policies for a resource usage may vary dependency upon an application, therefore, protection mechanisms should be provided as tools for the enforcement of these varying policies.

14.3 Principles of Protection

Frequently, a guiding principle can be used in the design of an operating system. This principle simplifies design decisions and keeps the system consistent and easy to understand. A key time-tested guiding principle for protection is the principle of least privilege. It dictates that programs, users and even systems be given just enough privileges to perform their tasks.

Consider the analogy of a security guard with a passkey. It this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. If however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied or otherwise compromised will be much greater.

An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the minimum damage to be done. The overflow of a buffer in a system daemon might cause the daemon to fail, for example, but should not allow the execution of code from the process's stack that would enable a remote user to gain maximum privileges and access to the entire system (as happens too often today.)

Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs. An operator who needs to mount tapes and backup files on the system has access to just those commands and files needed to accomplish the job. Some systems implement role- based access control (RBAC) to provide this functionality.

Computers implemented in a computing facility under the principle of least privilege can be limited to running specific services, accessing specific remote hosts via specific services, and doing so during specific time.

The principle of least privilege can help produce a more secure computing e environment. Unfortunately, it frequently does not. For example, Windows 2000 has a complex protection scheme at its core and yet has many security holes. By comparison, Solaris is considered relatively secure, even though it is a variant of UNIX, which historically was designed with little protection in mind. One reason for the difference may be that Windows 2000 has more lines of code and more services than Solaris and thus has more to secure and protect. Another reason could be that the protection scheme in Windows 2000 is incomplete or protects the wrong aspects of the operating system, leaving other area vulnerable.

14.4 Domain of Protection

A computer system is a collection of processes and objects. By objects, we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

A process should be allowed to access only those resources for which it has authorization, Furthermore, at any time, a process should be able to access only those resources that it currently requires to complete its task. This second requirement, commonly referred to as the need-to-know principle, is useful in limiting the amount of damage a faulty process can cause in the system. For example, when process p invokes procedure A(), the procedure should be allowed to access only its own variables and the formal parameters passed to it, it should not be able to access all the variables of process p. Similarly, consider the case where process p invokes a compiler to compile a particular file. The compiler should not be able to access files arbitrarily but should have access only to a well-defined subset of files (such as the source file, listing file, and so on) related to the file to be compiled. Conversely, the compiler may have private files used for accounting or optimization purposes that process p should not be able to access. The need to-know principle is similar to the principle of least privilege, in that the goals of protection are to minimize the risks of possible security violations.

14.4.1 Domain Structure

To facilitate this scheme, a process operates within a protection domain which specifies the resources that the process may access. Each domain defines a set of objects and the types of operation that may be invoked on each object. The ability to execute an operation on an object is an access right. A domain is a collection of access rights, each of which is an ordered pair <object-name, rights-set>. For example, if domain D has the access right <file F, {read, write}>, then a process executing in domain D can both read and write file F; it cannot, however, perform any other operation on that object.

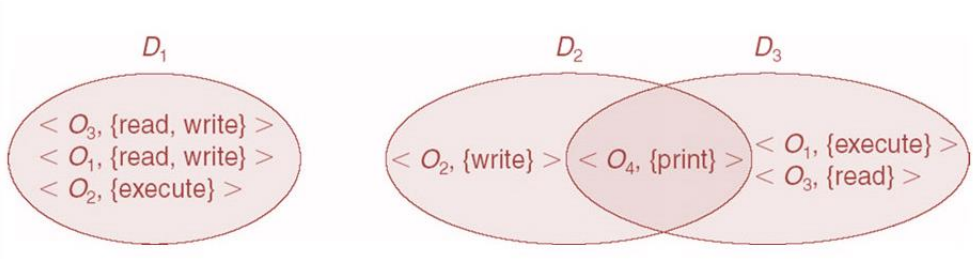


Figure 14.1 : System with three protection

Domains do not need to be disjoint; they may share access rights. For example, in Figure, we have three domains: D1, D2 and D3. The access right <O4, {print}" > is shared by D2 and D3, implying that a process executing in either of these two domains can print object O4. Note that a process must be executing in domain D2 to read and write object O1, while only processes in domain D3 may execute object O1.

The association between a process and a domain may be either static, if the set of resources available to the process is fixed throughout the process's lifetime, or dynamic. As might be expected, establishing dynamic protection domains is more complicated than establishing static protection domains.

If the association between process and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. The reason stems from the fact that a process may execute in two different phases and may, for example, need read access in one phase and write access in another. If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than are needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa. Thus, the need- to-know principle is violated. We must allow the contents of a domain to be modified so that it always reflects the minimum necessary access rights.

If the association is dynamic, a mechanism is available to allow domain switching, enabling the

process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

A domain can be realized in a variety of ways:

- Each user may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed-generally when one user logs out and another user logs in.
- Each process may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each procedure may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

14.5 Methods for Enforcement of Protection Mechanisms

As we know that a computer system is a collection of processes and objects (resources). Objects means hardware objects like CPU, memory, disk, printer, etc., and software objects like files, programs, semaphores, etc. Each object has a specific kind of operation applicable to it. For example, memory can be read or written, CPU can be only executed on, disk can be read and written, data files can be created, deleted, opened, read, written and closed. Thus, if these operations are known, a control mechanism can be implemented for each resource. A process should be allowed to access only those resources it has been authorized to access. A process should follow "Need-to-know" principle so that it should be able to access the resources, which it currently needs. This can be one way of implementing protection because only those resources, which are currently required, will need to be protected and others will be totally safe.

14.5.1 Access right

At any instance, protection should be provided to those resources, which may be currently used by a process. Thus, a domain known as protection domain can be defined for a process. Such a domain will contain the set of objects and the types of operations that can be invoked on each object. This means that domain is a set of <object, rights> pair. 'Rights' refer to the operation that can be invoked by a process on the corresponding object. These rights are also known as access rights of a particular process on the corresponding object e.g. - if a process is executing in domain D = < data file F, {read, write}>; it can perform only read and write

operations on the data file named 'F'.

The association of a process with a domain may be static of dynamic. The association can be static only when the set of resources available to a process is fixed throughout its lifetime. In case of static association, protection can be implemented by strictly adhering to the "need-to-know" principle. This implies that the domain should keep changing at different instances of a process' execution, as and when there is a change in the use of resources by this process. For example, consider a process P that needs read access (on a disk) in one phase and write access in another phase. Then in the first phase the domain should include only read access right i.e. < disk, read > and in the later phase, the domain should be modified to contain < disk, write >.

In case of dynamic association, a process can switch form one domain to another. Thou, here instead of changing the contents of a domain (to which a process may be statically associated as in case of static association), a new domain may be created with the changed content and the process may be switched to this new domain.

Examples of implementation of protection mechanisms using domain are:

1. Dual mode (Monitor User mode) model of operating system execution is an example of protection mechanism because it ensures that if a process is in monitor mode, it can execute privileged instructions only and if it executes in user mode, it can invoke only non-privileged instructions. Memory protection can be implemented in multiprogramming environment where more than one user program may be executing in the user mode and may sometimes, interfere with each other.
2. In UNIX, a domain is actually associated with a user. In this case, the sets of objects that can be accessed depend upon the identity user. Domain switching occurs when the user is changed generally, this happens when one user logs out and other logs in or in a multiuser system when more than one user may be trying to access the same object. This can help in safe sharing of objects among different users. Therefore, the domain will contain < user id, domain-bit >. This domain bit is also known as stupid bit. When a user (with user-id = A) starts executing a file (owned by user B), whose domain bit is off, the user-id of the process is set to A. This temporary user-id change ends when the process accessing the file exits or when the user logs off, whichever is earlier. When the domain bit is on, the user-id is not changed to A and remains B i.e., owner of the file. This implies that file is used by B and thus, should not be used by A.

14.5.2 Access Matrix

The access matrix model is the policy for user authentication and has several implementations such as access control lists (ACLs) and capability lists. It is used to describe

which users have access to what object (resources). Thus, basically it checks the access rights of users on system resources. The access matrix model consists of four major parts :

- A list of objects
- A list of subjects
- A function T that returns n object's type

The matrix itself, with the objects (resources) making the columns and the subjects making the rows.

In the cells where a subject and object meet, lie the rights the subject has on that object. Some example access fights (as discussed in previous sub-section) are read, write, execute, list and delete. Figure 14.2 shows an example access matix.

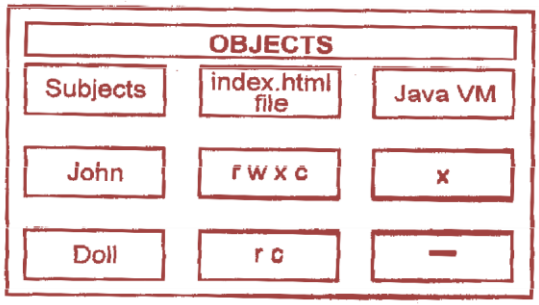


Figure 14.2: Example Access Matrix

An access matrix has several standard operations associated with it:

- Entry of a right onto a specified cell
- Removal of a right from a specified cell
- Creation of a subject
- Creation of an object
- Removal of a subject
- Removal of an object

The access matrix shown is Figure 14.2 considers users as subject and states that john can read. write, execute and copy the index.html file and that Doll can only read and copy this file. The subjects can also be processes and procedures. This implies that access matrix model can be used to define the access fights for processes also. It means that subjects are in fact the domains, which can be realized either as a user, or a process a procedure. The generalized access

matrix model is shown in figure 14.3

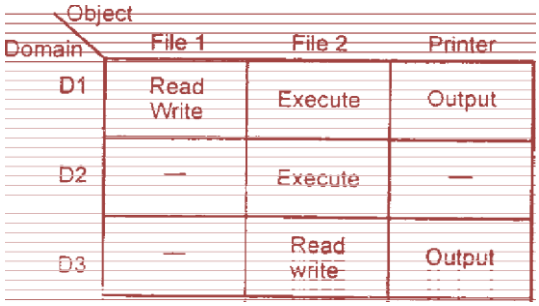


Figure 14.3 : Generalized Access Matrix

Advantages of access matrix as a mechanism for and implementing protection are:

It provides an appropriate mechanism for defining and implementing strict control for both the static and dynamic association between processes and domains. When a process is switched from one domain to another, n operation called switch is used. Thus, domain switching by processes can be controlled by including domains among the objects of the access matrix. Figure 14.3 shows an access matrix with domains as objects.

This figure shows that a process executing in domain D1 can switch to domain D2. Similarly, a process executing in domain D2 can switch to domain D1 or domain D3.

It is thus an appropriate method of implementing protection mechanism.

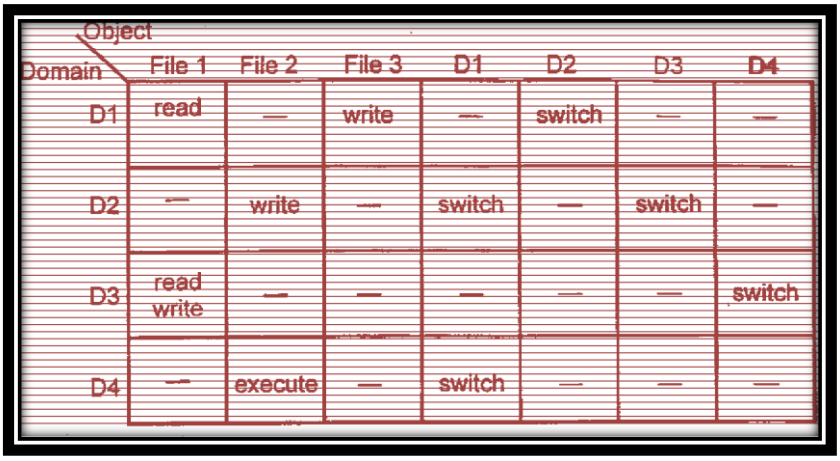


Fig. 14.4: Generalized Access matrix with Domain as Object

Disadvantages of this method are:

1. Although a useful model, access matrices are inefficient for storage of access rights in a computer system because they tend to be large and sparse.
2. Mechanisms must be enforced to protect the access matrices themselves from change.

14.5.3 Implementation of access matrices

The two most commonly used implementations are access control lists (ACLs) and Capability lists.

1. Access Control Lists (ACLs)

Access Control Lists (ACLs) are created by placing on each object a list of users and their rights to access that object.

2. Capability Lists

Capability Lists are implemented by storing on each subject a list o right the subject has for every object. This effectively gives each user, a sort of key ring. To remove access to a particular object, every user (subject) that has access to it must be "touched". A touch is an examination of a user's right to that object. Figure 14.4 shows an example of implementation of the access matrix using capability lists.

14.5.4 Comparison of Access list and capabilities

	Access Lists	*Capabilities List*
01	Each object (resource has a list of pairs of the form <subject, access rights>	Each subject (user, process or procedure has a list of pairs of the form <object, access rights>
02	It would be tedious to have separate Listing for each subject (user), therefore, they are grouped into classes. For example, in UNIX, there are three Classes: self, group, anybody else.	Here capabilities are the names of the objects. The objects not referred to in a capability list cannot be even named.
03	The default is: Everyone should be also to access a file.	The default is: No one should be able to access a file unless they have been given a capability
04	Access lists are simple and are used In almost all file systems.	Capabilities are used in systems that Need to be very secure as if prohibits sharing of information unless access is given to a subject.

14.6 Revocation of Access Rights

In a dynamic protection system, we may sometimes need to revoke access right to objects shared

by different users. Various questions about revocation may arise.

The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

- Reacquisition: Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. It access has been revoked, the process will not be able to reacquire the capability.
- Back-pointers: A list of pointers is maintained with each object, pointing to all capabilities associated with that object.
- Indirection: The capabilities point indirectly, not directly, to the objects Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desire entry and deleting it. Then, when an access is attempted, the capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.
- Keys: A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning the capability. A master key is associated with each object; it can be defined or replace with the set-key operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. Revocation replaces the master key with a new value via the set-key operation, invalidating all previous capabilities for the object.

This scheme does not allow selective revocation, since only one master key is associated with each object. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one global table of keys. A capability is valid only if its key matches some key in the global table. We implement revocation by removing the matching key from the table. With this scheme, a key can be associated with several objects, and several keys can be associated with each object, providing maximum flexibility.

In key-based schemes, the operations of defining keys, inserting them into lists and deleting them from lists should not be available to all users in particular, it would be reasonable to allow only the owner of an object to set the keys for that object. This choice, however, is a policy decision that the protection system can implement but should not define.

14.7 Summary

Computer system objects need to be protected from misuse. Objects may be hardware (memory, CPU & I/O devices) or software (files, programs & semaphores) An access right is permission to perform an operation on an object. A domain is a set of access rights. Processes execute in domains. The access matrix is a general model of protection that provides a mechanism for protection without imposing a particular policy on the system or its users. Revocation of access rights in a dynamic protection model is typically easier to implement with an access list scheme then with a capability list.

Self Assessment Questions

1. Discuss the strengths and weakness of implementing an access matrix using access list that are associated with objects.
2. Compare capability lists and access lists.
3. Differentiate between static and dynamic association of a process with a domain.
4. What is the "need-to-know" principle? Why is it important for protection system to adhere to this principle?
5. Suppose that you share a file on a network with fellow student so that he can copy this file from your system. What right should he be given on this file? What will be the contents of cell of access matrix corresponding to this grant?
6. Discuss access control mechanism in context to data files.
7. What is a domain? What are access hierarchies and protection rings.

Unit 15: System Security

15.0 Objective

After studying this unit, you will be able to understand concept of security and its principles and need. You will learn about methods of authentication, threats to computer system and classification of security system. This chapter provides a general overview of

- Security concepts
- Authentication and Encryption
- Program and System Threats
- Computer Security Classification

15.1 Introduction

Security is an important aspect of any OS. In general secure system will control, through use of specific security features, access of to information that only properly authorized individual or processes operating on their behalf will have access to read, write, create or delete.

Security Violation (or misuse) of the system can be categorized as intentional or accidental. It is easier to protect against accidental misuse than against malicious misuse.

15.1.1 Need for Security

There in no or very little security in computer application, until the importance of data was truly realized. The computer applications for financial and personal data were developed, real need for security was felt & people realized that data on computers are extremely important aspect of modern life. Two typical examples of such security mechanism were as follows:

- Provide a user id & Password to every user and use that information to authenticate a user.
- Encode information stored in fuel databases in some fashion so that it is not visible to users who do not haveright permission.

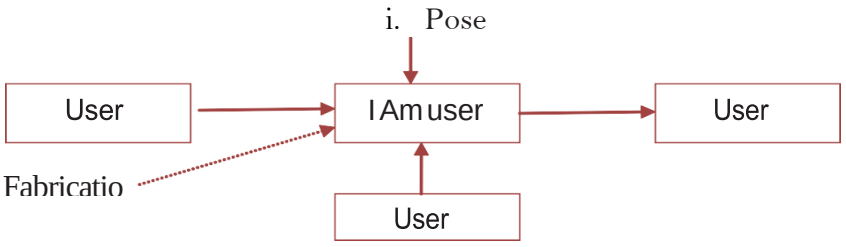
As technology improved, fuel communication infrastructure became extremely mature and newer and newer application behaves to be developed for various user demands and needs. People realized that basic security, measurers are not quite enough.

15.1.2 Principles ofSecurity

These are four chief principles of security Authentication, Confidentiality, Non-Repudiation and Integrity. There are two more, Access control & Availability which are not related to particular message but are linked to overall system security as whole.

a) Authentication:

This mechanism helps to establish proof of identity. The origin of an electronic message or document is correctly identified.



a. **Figure 15.1: Absence of authentication**

"Fabrication is possible in the absence of proper authentication mechanism" means absence of authentication leads to Fabrication.

b) **Confidentiality:**

This specifies that only sender and intended recipients should be able to access the contents of message. Confidentiality gets compromised if an unauthorized person is able to access a message

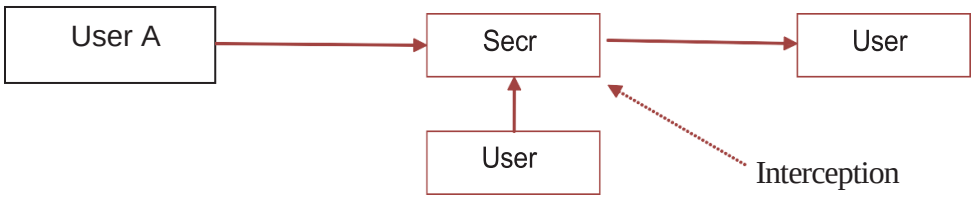


Figure 15.2: Loss of Confidentiality

Here confidential message sent by A to B, which is accessed by C without permission or knowledge of A and B. This type of attack is interception. Interception causes loss (Absence) of message confidentiality.

c) **Non-repudiation:** There are situations where a user sends a message and later on refuses. That he had sent that message.

a. I never sent, that message which you claim to have received

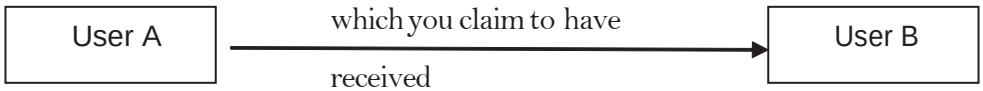


Figure 15.3: Establishing non-repudiation

"Nonrepudiation does not allow the sender of a message to refuse the claim of not sending that message".

d) **Integrity:** When the contents of a message are changed after the sender sends it, but before it reaches the intended recipients. Then integrity of message is Lost.

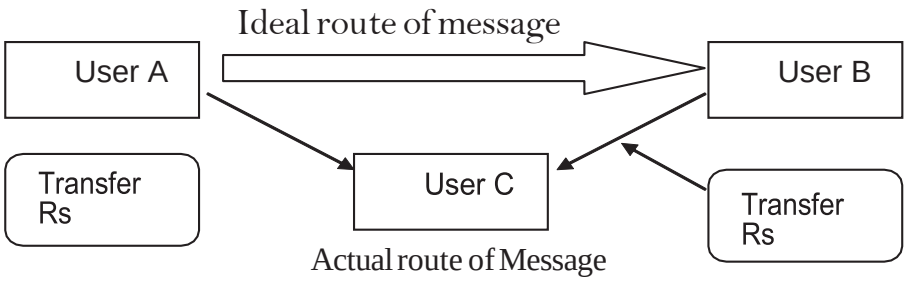


Figure 15.4: Loss of integrity

Here user C tempers with message and send changed message to user B (Bank). User B has no way of knowing that contents charged. This type of attack is modification. Modification Courses Loss of message integrity.

e) **Access Control:** This determines who should be able to access what. For example user A can read to file X, Write to Y, Z but can only update P, Q .

   a. This is broadly related in two areas :
   b. Role management (user side) : Which user can do what?
   c. Rule management (resource side) : which resource is available in what circumstances/

"Access Control specified who can access what"

f) **Availability:** This states that resources (i.e. information) should be available to authorize parties at all times.

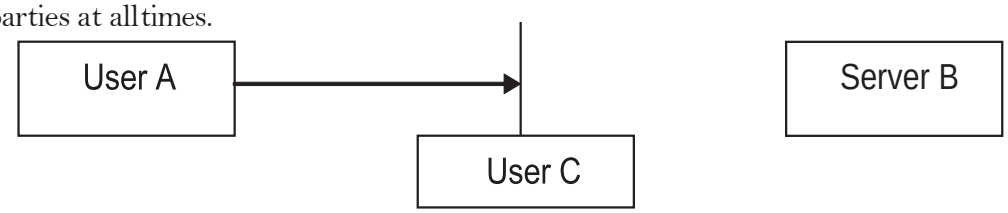


Figure 15.5: Attack on availability

Attack on availability Due to intentional action of unauthorized user C, an authorized user A may not be able to contact server B. Such attach is called Interruption "Interruption puts availability of resource in danger"

15.2 Authentication

Authentication of the user of a system is one of the major security issues associated with an

operating system. This authentication is different from what we discussed in Section as it deals with authenticating the user who tries to access a system a system resource. Generally, authentication of a user is based on:

1. User identifier and password (Passwords)
2. Badge card possessed by a user (Artifacts)
3. Finger prints, retina pattern or signature (Biometrics techniques) Let us discuss these techniques in detail.

15.2.1 Passwords

The password is the most common authentication mechanism based on sharing of a secret. In a password-based system each user has a password, which may initially be assigned by the system administrator. Many systems allow users to subsequently change their passwords. The system stores all user passwords and uses them to authenticate the users. When logging in, the system requests and the user supplies a presumably secret, user-specific passwords.

Passwords are popular because they require no special hardware and are relatively easy to implement. On the negative side, passwords offer limited protection, as they may be relatively easy to obtain or guess. Unencrypted passwords files stored in a system are obviously an easy prey. User-chosen passwords are frequently dictionary words or proper names. This makes them easy to remember and easy to guess. For example, user Ids, names or surnames, and their backward spellings typically account for a significant percentage of passwords.

System-chosen passwords, on the other hand, are usually random combinations of letters and numbers that are hard to guess but are also hard to remember. As a result, the users tend to write them down and store them in a handy place near the terminal. This can be easily located by someone and thus may no longer be a secret.

Various techniques have been proposed to strengthen the level of protection availed by the password mechanism. Unfortunately, most of these have drawbacks that reduce either their effectiveness or user acceptance. For example, password schemes may be multilevel, and users may be required to supply additional password schemes may be multilevel, and users may be required to supply additional passwords at the system's request at random intervals during computer use. This tends to annoy the actual authorized users. Another technique is to have the system issue a dynamic challenge to the user after log-in. This challenge can be in the form of a random number generated by the computer, to which the user is supposed to apply a secret

transformation, such as squaring and incrementing the value. Failure to do so may be used to detect unauthorized users.

15.2.2 Aritifact-Based Authentication

The artifacts commonly used for user authentication include machine-readable badges (usually with magnetic stripes) and electronic smart cards. Badge or card readers may be installed in or near the terminals, and users are required to supply the artifact for authentication. In many systems, artifact identification is coupled with the use of a password. That is, the user must insert the card and then supply his or her password. This form of authentication is common with automated teller banking machines. The artifact-based systems work especially well in environments where the artifact is also used for other purposes.

For example, in some companies' badges are required for employees to gain access to the organization's gate. The use of such a badge as an artifact for computer access and authentication can reduce the likelihood of the loss of an artifact. Smart cards can augment this scheme by keeping even the user's password within the card itself, which allows authentication without storage of passwords in the computer system. This makes it more difficult for mischievous users to uncover user passwords but he loss of such cards can be hazardous.

15.2.3 Biometrics Techniques

The third major group of authentication mechanisms is based on the unique characteristics of each user. Some user characteristics can be established by means of biometrics techniques. These fall in two basic categories:

1. Physiological characteristics, such as fingerprints, capillary patterns in the retina, had geometry, and facial characteristics.
2. Behavioural characteristics, such as signature dynamics, voice pattern, and timing of keystrokes.

In general, behavioural characteristics can vary with a user's state and thus may be susceptible to higher false acceptance or rejection rates. For example, signature dynamics and keystroke patterns may vary with a user's stress level and fatigue.

The biometrics detection devices are usually self-contained and independent of the computer system, which increases their resistance to common computer penetration methods and improves the potential for tamper proofing.

The primary advantages of biometrics authentication are the largely increased accuracy of user

authentication and reduction of errors in security-conscious environments. For example, some retinal-scan devices claim an error rate as slow as 1 in 3 million (1 in trillion when both eyes are scanned).

The drawbacks of biometrics authentication include increased cost, potential invasion of privacy, and reluctance of some users.

15.3 Encryption

As computer networks gain popularity, more sensitive information is being transmitted over channels where eavesdropping and message interception is possible. Thus, the operating system should have some provision to fight such situations i.e. to protect the data that are transferred over the network.

Encryption is one such mechanism, which allows such data to be scrambled so that even if some one intercepts it on the network, it is not readable to him/her. Thus, the basic purpose of encryption is to make the data transfer secure over the network.

Encryption works as:

1. It transforms information from ("Clear" or "plan taxt") to coded information ("Cipher text"), which cannot be read by outside parties.
2. This transformation process is controlled by an algorithm and a key.
3. This process should be reversible so that the intended recipient can read the information transmitted to him in the form of cipher text. But for this a decryption mechanism which decrypts (decodes) the cipher text to plaintext is must. Figure shows a general encryption and decryption mechanism.

The main challenge in using this approach is the development of encryption schemes that are impossible to break.

These are two kinds of encryption:

1. 'Symmetrical Encryption' or secret key which uses a single key to encrypt and decrypt the transmitted data.
2. 'Asymmetrical Encryption' which uses 'Private Key', in which one key is used to encrypt and another to decrypt the transmitted data.

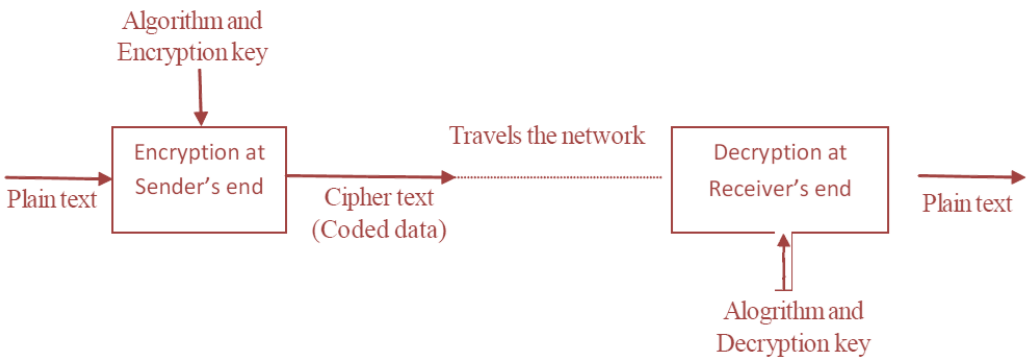


Figure 15.6: Encryption & Decryption Mechanism

15.4 Program and System Threats

15.4.1 Virus

Attacker can launch application level or network level attack using virus. A virus is a computer program that attaches itself to another authorized program and causes damage to the computer system or to the network. A virus can be repaired (antivirus used) and its damage can be controlled by using good backup and recovery procedures.

During its lifetime, A virus goes through four phases:

a. Dormant Phase: Here virus idle, activated based on certain action / event / date-time / key stroke occurs this is optional phase.

b. Propagation Phase: In this phase virus copies itself and each copy starts creating more copies of self, them propagating virus.

c. Triggering Phase: A dormant virus moves into this phase when action / event for which it was waiting is initiated.

d. Execution Phase: this is actual work of virus, which could be destructive (delete files) or harmless (display some message).

Virus can be classified into following categories:

a. Parasitic Virus: Common type, attaches itself to executable files and keeps replicating.

b. Memory resident Virus: Attaches itself to area of main memory then infect every executable.

c. Boot sector Virus infects master boot record of disk and spreads when Operating System starts booting computer.

d. Stealth Virus: Built in intelligence, prevents antivirus software from detecting it.

e. Polymorphic Virus: Virus keeps on changing its signature (identity) on very execution;

Hence difficult to detect.

f. Metamorphic Virus: In addition to changing signature, this virus keeps rewriting itself every time detection much harder.

15.4.2 Worm

Worm is a piece of code which gets replicated. It is similar to virus, but its implementation is different. A virus modifies a program / data while warm does not modify. It replicates again & again.

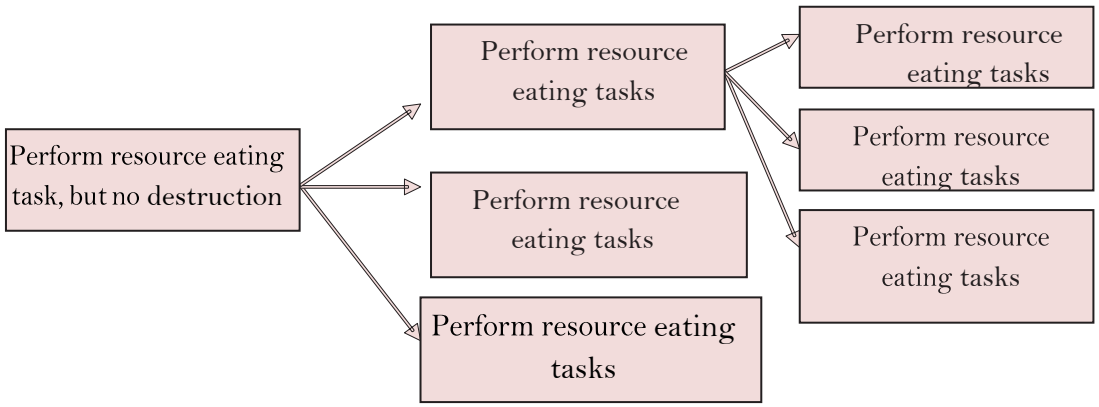


Figure: 15.7: Worm

Hence network or Computer becomes very slow. Hence "A Worm does not perform any destructive action only consumes system resources to bring it down."

15.4.3 Trojan Horse

It is a hidden piece of code, like virus but it attempts to reveal confidential information to an attacker. The name (Trojan Horse) is due to the greck soldiers who hide inside a large hollow horse which was pulled by Troy citiren un ware of the contents. Once greek soldiers entered the city of troy they opened the gates for the rest & greek soldiers.

In similar fashion, Trojan horse could silently sit in the code for a login screen by attaching itself to it. When user enters the user id & password the Trojan horse captures these details & sends this information to the attacker without the knowledge of the user. Attackes can misuse this information.

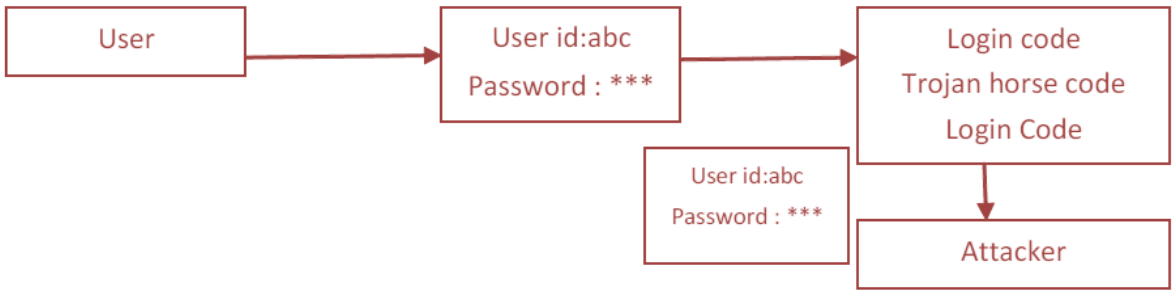


Figure 15.8: Trojan Horse

"ATrojan Horse allows an attacker to obtain some confidential. To obtain some confidential information about a computer or network".

15.4.4. Trapdoors

Sometimes, software designers may want to be able to modify their programs after their installation and even after they have gone in production. To assist them in this task, the programmers leave some secret entry points which do not require authorization to access certain objects. Essentially, they bypass certain validation checks. Only the software designers know how to make use of these shortcuts. These are called trap doors. At times such shortcuts may be necessary for coping with emergency situations, but then these trap doors can also be abused by some others to penetrate into the system.

15.4.5 Logic Bomb

Consider a program that initiates a security incident only under certain circumstances. It would be hard to detect because under normal operations there would be no security hole. However, when a predefined set of parameters were met, the security hole would be created. This scenario is known as a logic bomb. A programmer, for example, might write code to detect if she is still employed; if that check failed, a daemon could be spawned to allow remote access or code could be launched to cause damage to the site.

15.4.6 Port Scanning

Port scanning is not an attack but rather is a means for a cracker to detect a system's vulnerabilities to attack. Port scanning typically is automated, involving a tool that attempts to create a TCP/IP connection to a specific port or a range of ports. For example, suppose there is

a known vulnerability (or bug) in send mail. A cracker could launch a port scanner to try to connect to say, port 25 of a particular system or a range of systems. If the connection was successful, the cracker (or tool) could attempt to communicate with the answering service to determine if it was indeed send mail and, if so, if it was the version with the bug.

There is no such tool, but there are tools that perform subsets of that functionality. For example, nmap (from http://www.insecure.org/nmap/) is a very versatile open-source utility for network exploration and security auditing.

15.4.7 Stack & Buffer Overflow

The stack-or buffer-overflow attack is the most common way for an attacker outside the system, on a network or dial-up connection, to gain unauthorized access to the target system. An authorized user of the system may also use this exploit for privilege escalation.

Essentially, the attack exploits a bug in a program. The bug can be a simple case of poor programming, in which the programmer neglected to code bounds checking on an input field. In this case, the attacker sends more data than the program was expecting. Using trial and error, or by examining the source code of the attacked program if it is available, the attacker determines the vulnerability and writes a program to do the following:

1. Overflow an input field, command-line argument, or input buffer-for example, on a network daemon-until it writes into the stack.
2. Overwrite the current return address on the stack with the address of the exploit code loaded in step 3.
3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute-for instance, spawn a shell.

The result of this attack program's execution will be a root shell or other privileged command execution.

15.4.8 Denial of Service

DOS attacks make an attempt to prevent legitimate users from accessing some services, which they are eligible for e.g. for instance an unauthorized user might sand too many login requests to a serves using random user id's one after the other in quick succession, so as to flood the network and deny other legitimate users to access the system.

15.5 Computer Security Classification

The U.S. Department of Defense Trusted Computer System Evaluation Criteria specify four security classifications in systems. A,B, C, and D. This specification is widely used to determine the security of a facility and to model security solutions, so we explore it here. The lowest-level classification is division D, or minimal protection. Division D includes only one class and is used for systems that have failed to meet the requirements of any of the other security classes For instance, MS-DOS and Windows 3.1 are in division D.

Division C, the next level of security, provides discretionary protection and accountability of users and their actions through the use of audit capabilities. Division C has two levels: C1 and C2. AC1-class system incorporates some form of controls that allow users to protect private information and to keep other users from accidentally reading or destroying their data. A C1 environment is one in which cooperating users access data at the same levels of sensitivity. Most versions of UNIX and C1 class.

The sum total of all protection systems within a computer system (hardware, software, firmware) that correctly enforce a security policy is known as a trusted computer base (TCB). The TCB of a C1 system controls access between users and files by allowing the user to specify and control sharing of objects by named individuals or defined groups. In addition, the TCB requires that the users identify themselves before they start any activities that the TCB is expected to mediate. This identification is accomplished via a protected mechanism or password; the TCB protects the authentication data so that they are inaccessible to unauthorized users.

A C2-class system adds an individual-level access control to the requirements of a C system. For example, access rights of a file can be specified to the level of a single individual. In addition, the system administrator can selectively audit the actions of any one or more users based on individual identity. The TCB also protects itself from modification of its code or data structures. In addition, no information produced by a prior user is available to another user who accesses a storage object that has been released back to the system. Some special, secure versions of UNIX have been certified at the C2 level.

Division-B mandatory-protection systems have all the properties of a class C2 system in addition, they attach a sensitivity label to each object. The B1-class TCB maintains the security label of each object in the system; the label is used for decisions pertaining to mandatory access control. For example, a user at the confidential level could not access a file at the more sensitive secret level. The TCB also denotes the sensitivity level at the top and bottom of each page of any human-readable

output. In addition to the normal user name password authentication information, the TCB also maintains the clearancy and authorizations of individual users and will support at least two levels of security. These levels are hierarchical, so that a user may access any objects that carry sensitivity lables equal to or lower than his security clearance. For example, a secret-level user could access a file at the confidential level in the absence of other access controls. Processes are also isolated through the use of distinct address spaces.

A B2-class system extends the sensitivity labels to each system resource, such as storage objects. Physical devices are assigned minimum and maximum security levels that the system uses to enforce constraints imposed by the physical environments in which the devices are located. In addition, a B2 system supports covert channels and the auditing of events that could lead to the exploitation of a covert channel.

A B3-class system allows the creation of access-control lists that denote users or groups not granted access to a given named object. The TCB also contains a mechanism to monitor events that may indicate a violation of security policy. The mechanism notifies the security administrator and security policy. The mechanism notifies the security administrator and. If necessary, terminates the event in the least disruptive manner.

The highest-level classification is division A. Architecturally, a class A1 system is functionally equivalent to a B3 system, but it uses formal design specifications and verification techniques, granting a high degree of assurance that the TCB has been implemented correctly. A system beyond class A1 might be designed and developed in a trusted facility by trusted personnel.

The use of a TCB merely ensures that the system can enforce aspects of a security policy, the TCB does not specify what the policy should be. Typically, a given computing environment develops a security policy for certification and has the plan accredited by a security agency, such as the National Computer Security Center. Certain computing environments may require other certification, such as that supplied by TEMPEST, which guards against electronic eavesdropping. For example, a TEMPEST certified system has terminals that are shielded to prevent electromagnetic field from escaping. The shielding ensures that equipment outside the room or building where the terminal is housed cannot detect what information is being displayed by the terminal.

15.6 Summary

Security has gained immense prominence as all business using computer system. The principal of

any security mechanism are confidentiality, authentication integrity, non-repudiation, access control & availability. Authentication is consumed with establishing identity of user or system, Encryption limits the domain of receivers of data. While authentication limits the domain of senders. Severed type of attacks can be launched against program and against individual computers or the masses. Stack & buffer overflow techniques allow successful attackers to change their level of system access. Virus & Worms are self perpetuating, sometimes infecting thousand of computers Denial of service attacks prevent legitimate use of target systems.

The four security classification in system A,B,C & D. the specifications are widely used to determine the security of a facility and to model security solutions.

Self-Assessment Questions

1. What are the key principles of security?
2. Discuss reasons behind the significant of authentication?
3. What is a worm ? Give the significant difference b/w Worm and Virus?
4. Discuss the principle behind Trojan horse?
5. Describe the program& System threats like port scanning, trapdoor's, Logic bomb, and denial of service.
6. Explain the different computer system classification in detail.
7. What are two advantages of encrypting data stored in the computer system?

Unit 16: Distributed Computing

16.0 Objective

This chapter provides a general overview about

- Distributed Computing
- Distributed Process Management
- Message Passing
- Remote Procedure Calls
- Distributed Memory Management

16.1 Introduction to Distributed Computing

The high volume of networked computers, workstations, LANs has prompted users to move from a simple end user computing to a complex distributed computing environment. This transition is not just networking the computers, but also involves the issues of scalability, security etc. A Distributed Computing Environment herein referred to, as DCE is essentially an integration of all the services necessary to develop, support and manage a distributed computing environment. Despite the advances in processor design, users still demand more performance. Eventually, single CPU technologies must give way to multiple processors parallel Computers: it is less expensive to run 10 inexpensive processors cooperatively than it is to buy a new computer 10 times as fast. This change is inevitable and has been realized to some extent in the specialization of subsystems like bus mastering drive controllers. However, the need for additional computational power has thus far rested solely on advances in CPU technologies.

The present day computing industry depends on the efficient usage of resources. So instead of duplicating the resources at every node of computing, a remote method of accessing the resources is more efficient and saves costs. This gave rise to the field of distributed computing, where not only physical resources, but also processing power was distributed.

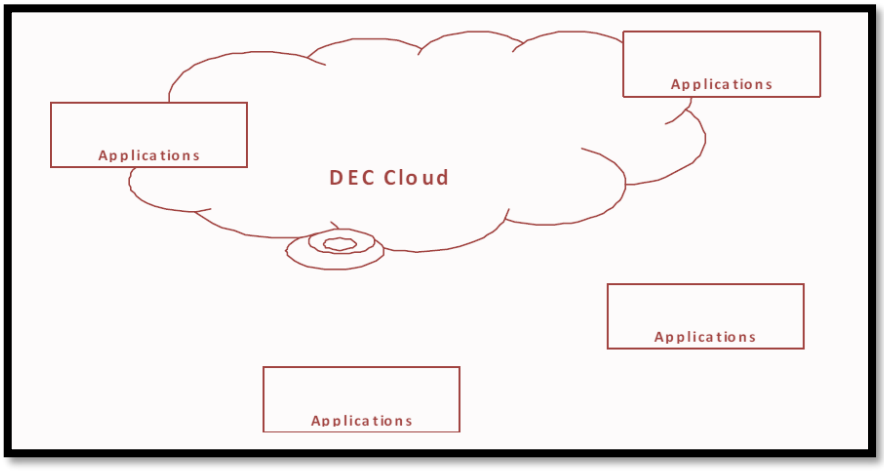
Distributed computing was driven by the following factors,

a) Desire to share data and resources
b) Minimize duplication of functionality
c) Increase cost efficiency
d) Increase reliability and availability of resources.

When an organization migrates from networked computing to Distributed Computing a lot of factors are to be taken into consideration. For example, replication of files gives rise to consistency problems, clock synchronization becomes important, and security is a bigger consideration.

A Distributed Computing Environment addresses all these issues by providing an integrated set of cross platform, comprehensive services which aids in the development and application of distributed applications.

The following diagram gives a simple view of the DCE architecture,



The DCE cloud refers to the distributed computing environment tools that facilitate distributed computing.

A distributed system is a collection of independent computers that appear to its users as a single coherent system.

– Andrew Tannenbaum

This certainly is the ideal form of a distributed system, where the "implementation detail" of building a powerful system out of many simpler systems is entirely hidden from the user.

Unfortunately, when we look at the reality of networked computers, we find that the multiplicity of system components usually shines through the abstractions provided by the operating system and other software. In other words, when we work with a collection of independent computers, we are almost always made painfully aware of this. For example, some applications require us to identify and distinguish the individual computers by name while in others our computer hangs due to an error that occurred on a machine that we have never heard of before.

16.1.1 Examples of Distributed Systems

Probably the simplest and most well known example of a distributed system is the collection of Web servers—or more precisely, servers implementing the HTTP protocol—that jointly provide the distributed database of hypertext and multimedia documents that we know as the World-Wide Web.

The alternative to using a distributed system is to have a huge centralized system, such as a mainframe. For many applications there are a number of economic and technical reasons that make distributed systems much more attractive than their centralized counterparts.

- **Cost:** Better price/performance as long as commodity hardware is used for the component

computers.

- **Performance:** By using the combined processing and storage capacity of many nodes, performance levels can be reached that are beyond the range of centralized machines.
- **Scalability:** Resources such as processing and storage capacity can be increased incrementally.
- **Reliability:** By having redundant components the impact of hardware and software faults on users can be reduced.
- **Inherent Distribution:** Some applications, such as email and the Web (where users are spread out over the whole world), are naturally distributed. This includes cases where users are geographically dispersed as well as when single resources (e.g., printers, data) need to be shared.

However, these advantages are often offset by the following problems encountered during the use and development of distributed systems:

New component: Network: Networks are needed to connect independent nodes and are subject to performance limitations. Besides these limitations, networks also constitute new potential points of failure.

Security: Because a distributed system consists of multiple components there are more elements that can be compromised and must, therefore, be secured. This makes it easier to compromise distributed systems.

Software Complexity: As will become clear throughout this course distributed software is more complex and harder to develop than conventional software; hence, it is more expensive to develop and there is a greater chance of introducing errors.

16.1.2 Hardware and Software Architectures

A key characteristic of our definition of distributed systems is that it includes both a hardware aspect (independent computers) and a software aspect (performing a task and providing a service). From a hardware point of view distributed systems are generally implemented on multicomputers. From a software point of view they are generally implemented as distributed operating systems or

middleware.

16.1.3 Multi-computers

A multicomputer consists of separate computing nodes connected to each other over a network.

Multi-computers generally differ from each other in three ways:

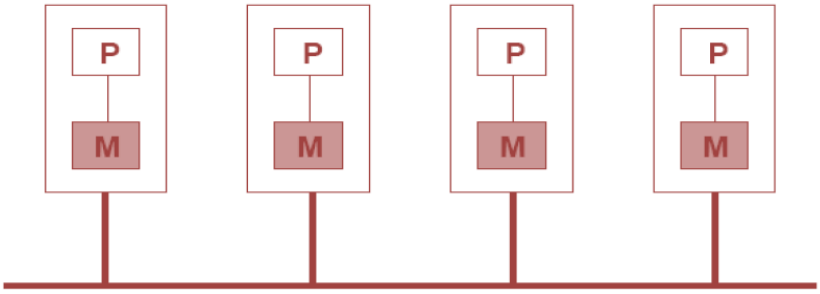
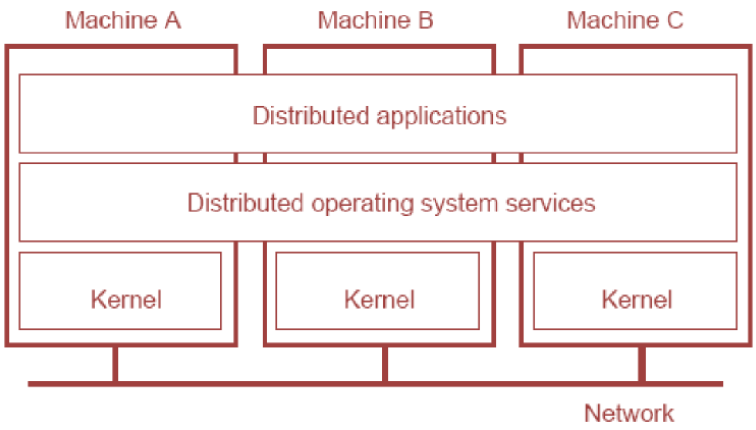


Figure 1: A multicomputer.

1. **Node resources:** This includes the processors, amount of memory, amount of secondary storage, etc. available on each node.
2. **Network connection:** The network connection between the various nodes can have a large impact on the functionality and applications that such a system can be used for. A multi computer with a very high bandwidth network is more suitable for applications that actively share data over the nodes and modify large amounts of that shared data. A lower bandwidth network, however, is sufficient for applications where there is less intense sharing of data.
3. **Homogeneity:** A homogeneous multicomputer is one where all the nodes are the same, that is they are based on the same physical architecture (e.g. processor, system bus, memory, etc.). A heterogeneous multicomputer is one where the nodes are not expected to be the same.

One common characteristic of all types of multicomputers is that the resources on any particular node cannot be directly accessed by any other node. All access to remote resources ultimately takes the form of requests sent over the network to the node where that resource resides.

16.1.4 Distributed Operating System



A Distributed Operating System

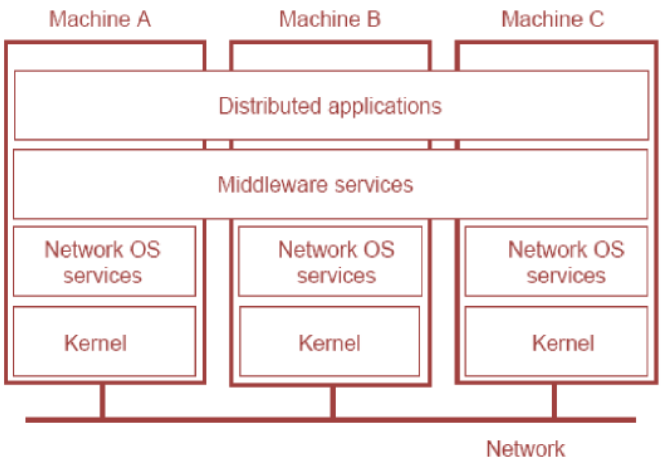
A distributed operating system (DOS) is a an operating system that is built, from the ground up, to provide distributed services. As such, a DOS integrates key distributed services into its architecture (Figure 2). These services may include distributed shared memory, assignment of tasks to processors, masking of failures, distributed storage, interprocess communication, transparent sharing of resources, distributed resource management, etc.

A key property of a distributed operating system is that it strives for a very high level of transparency, ideally providing a single system image. That is, with an ideal DOS users would not be aware that they are, in fact, working on a distributed system. Distributed operating systems generally assume a homogeneous multicomputer. They are also generally more suited to LAN environments than to wide-area network environments.

In the earlier days of distributed systems research, distributed operating systems where the main topic of interest. Most research focused on ways of integrating distributed services into the operating system, or on ways of distributing traditional operating system services. Currently, however, the emphasis has shifted more toward middleware systems. The main reason for this is that middleware is more flexible (i.e., it does not require that users install and run a particular operating system), and is more suitable for heterogeneous and wide-area multicomputers.

16.1.5 Middleware

Whereas a DOS attempts to create a specific system for distributed applications, the goal of middleware is to create system independent interfaces for distributed applications.



A Middleware System

As shown in Figure 3 middleware consists of a layer of services added between those of a regular network OS1 and the actual applications. These services facilitate the implementation of distributed applications and attempt to hide the heterogeneity (both hardware and software) of the underlying system architectures.

The principle aim of middleware, namely raising the level of abstraction for distributed programming, is achieved in three ways:

(1) communication mechanisms that are more convenient and less error prone than basic message passing;

(2) independence from OS, network protocol, programming language, etc. and

(3) standard services (such as a naming service, transaction service, security service, etc.).

To make the integration of these various services easier, and to improve transparency and system independence, middleware is usually based on a particular paradigm, or model, for describing distribution and communication. Since a paradigm is an overall approach to how a distributed system should be developed, this often manifests itself in a particular programming model such as 'everything is a file', remote procedure call, and distributed objects. Providing such a paradigm automatically provides an abstraction for programmers to follow and provides direction for how to design and set up the distributed applications. Paradigms will be discussed in more detail later on in the course.

Although some forms of middleware focus on adding support for distributed computing directly into a language (e.g., Erlang, Ada, Limbo, etc.), middleware is generally implemented as a set

of libraries and tools that enable retrofitting of distributed computing capabilities to existing programming languages. Such systems typically use a central mechanism of the host language (such as the procedure call or method invocation) and dress remote operations up such that they use the same syntax as that mechanism resulting, for example, in remote procedure calls and remote method invocation.

Since an important goal of middleware is to hide the heterogeneity of the underlying systems (and in particular of the services offered by the underlying OS), middleware systems often try to offer a complete set of services so that clients do not have to rely on underlying OS services directly. This provides transparency for programmers writing distributed applications using the given middleware. Unfortunately this 'everything but the kitchen sink' approach often leads to highly bloated systems. As such, current systems exhibit an unhealthy tendency to include more and more functionality in basic middleware and its extensions, which leads to a jungle of bloated interfaces. This problem has been recognised and an important topic of research is investigating adaptive and reflective middleware that can be tailored to provide only what is necessary for particular applications.

With regards to the common paradigms of remote procedure call and remote method invocations, Waldo et al. [WWWK94] have eloquently argued that there is also a danger in confusing local and remote operations and that initial application design already has to take the differences between these two types of operations into account. We shall return to this point later.

16.1.6 Distributed Systems and Parallel Computing

Parallel computing systems aim for improved performance by employing multiple processors to execute a single application. They come in two flavours: shared-memory systems and distributed memory systems. The former use multiple processors that share a single bus and memory subsystem. The latter are distributed systems in the sense of the systems that we are discussing here and use independent computing nodes connected via a network (i.e., a multicomputer). Despite the promise of improved performance, parallel programming remains difficult and if care is not taken performance may end up decreasing rather than increasing.

16.1.7 Distributed Systems in Context

The study of distributed systems is closely related to two other fields: Networking and Operating

Systems. The relationship to networking should be pretty obvious, distributed systems rely on networks to connect the individual computers together. There is a fine and fuzzy line between when one talks about developing networks and developing distributed systems. As we will discuss later the development (and study) of distributed systems concerns itself with the issues that arise when systems are built out of interconnected networked components, rather than the details of communication and networking protocols.

The relationship to operating systems may be less clear. To make a broad generalization operating systems are responsible for managing the resources of a computer system, and providing access to those resources in an application independent way (and dealing with the issues such as synchronization, security, etc. that arise). The study of distributed systems can be seen as trying to provide the same sort of generalized access to distributed resources (and likewise dealing with the issues that arise).

16.1.8 The DCE Cloud

It Consists of the Following Components,

a) Distributed File Service

b) Distributed Time Service

c) Security Service

d) Cell Directory Service

e) Threads Service

All these services are achieved by the use of Remote Procedure calls (RPC).

Properties of DCE

A DCE Provides a Global Computing Environment, which can interoperate with other services like DNS and X.500. This sort of global interoperability provides the much-needed interface for Write Once Run Anywhere Applications. Also the suite of components is completely integrated and interoperable, which facilitates the networking of two systems for processing even though they have different hardware and software configurations.

DCE Cells

A collection of machines, users and resources that are a part of a group and having their own directory service and security service can be called a DCE Cell. In an organization there may be

a large number of cells, say one for each department.

DCE Remote Procedure Calls (RPC)

The Remote Procedure Call (RPC) in a DCE is the facility that lets users make remote procedure calls and connect to another system on the DCE. The application programmer is essentially hidden from the fact that it is a remote procedure call, by the components of RPC.

16.2 Distributed Process Management

Most processes are created and managed by a command interpreter, but any other process may also create new ones. All that is required is the capability that allows communication with a Kernel. Most users will have access to the cluster creation capability for the Kernel running on their own workstation; that is, users can create new processes on their own workstation.

The capability for creating processes on poolprocessors is typically kept by a "Processor Pool" service that acts as an agent for running programs on behalf of user processes. Load balancing can be achieved by the Processor Pool service when it allocates pool processors judiciously. Although clusters rarely move to a new host after being started up, migration is a central concept in the process management mechanisms. This is became loading new clusters into memory, taking core dumps, making check- points, and doing remote debugging are all similar to migrating a cluster. In fact, if we can migrate a cluster from one machine to another, downloading, check-pointing, debugging, etc., should be simple.

Load balancing by migrating cluster is a poorly understood area and it is dubious whether it is very useful with the current sort of workstations and networks. Migrating a five megabyte cluster, for instance, will take at least seven seconds, because that is how long it takes a fast transport protocol to copy the memory contents over a 10 Mbit Ethernet; five megabyte programs are not at all uncommon, especially as candidates for migration: long-lived clusters are usually large too. Migration is thus rather expensive and the gain of a migrate operation must be big in order to merit one. In spite of this, migration can be useful. When a workstation's owner logs off in the evening, the workstation can turn itself into a Pool Processor and provide process execution service to the rest of the system. When the owner returns in the morning, however, and logs back on, the guest clusters running there could be nudged off by migrating them away to some other workstation.

16.3 Message Passing

In distributed systems, there are two kinds of fundamental inter-process communication models:

a) Shared Memory and

b) Message Passing.

From a programmer's perspective, shared memory computers, while easy to program, are difficult to build and aren't scalable to beyond a few processors. Message passing computers, while easy to build and scale, are difficult to program. In some sense, shared memory model and message passing model are equivalent. One of the solutions to parallel system communication is Distributed Shared Memory(DSM), where memory is physically distributed but logically shared. DSM appears as shared memory to the applications programmer but relies on message passing between independent CPUs to access the global virtual address space.

The **message-passing** is a common paradigm model for distributed computing, in the sense that it mimics the behavior in human communications. It is an appropriate paradigm for network services where processes interact with each other through the exchanges of messages. Message passing requires the participating processes to be tightly-coupled: throughout their interaction, the processes must be in direct communication with each other. If communication is lost between the processes (due to failures in the communication link, in the systems, or in one of the processes), the collaboration fails. The message-passing paradigm is data- oriented. Each message contains data marshaled in a mutually agreed upon format, and is interpreted as a request or response according to the protocol. The receiving of each message triggers an action in the receiving process. It is inadequate for complex applications involving a large mix of requests and responses. In such an application, the task of interpreting the messages can become overwhelming. A distributed object is one whose methods can be invoked by a **remote process**, a process running on a computer connected via a network to the computer on which the object exists.

16.4 Remote Procedure Calls

16.4.1 Definitions

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.

16.4.2 Components of RPC

The RPC components are

a) The Interface Definition Language and its Compiler: The skeletons and stubs are created by the IDL and then compiled by the IDL compiler. The server stubs replace the remote part of the procedure call, and at the server the skeleton replaces the client.

b) Runtime RPC Library: The RPC runtime library is actively involved in the sending and receiving of remote procedure calls and finding the necessary server services and communicating between the client and the server.

c) Secure RPC Components: The Secure RPC components work along with the security APIs to provide authentication and authorization for the remote procedure calls.

d) Name Service Independent APIs: The Name Service Independent (NSI) APIs help in locating the right server to process the request. It is integrated into the directory services Component to facilitate the Association and Binding of the Client to the Server.

e) UUID Facilities: This UUID Stands for Universal Unique Identifiers. This is useful to generate UUIDs, to uniquely identify each server and client on the DCE.

16.4.3 Specifications Conformance

The DCE Architecture conforms to the Network Computing Architecture (NCA) [IRPC] specifications. Transport independence and hence the OS independence is achieved as the NCA supports both connection oriented as well as connection independent protocols.

16.4.4 Facilities Supplied by the RPC

The following are the facilities that are supplied by the DCE RPC which are shielded from the application programmer [IRPC]

a) Security services

b) Use of the Directory Service to Find the right server for remote calls

c) Managing the data formats which are different in each cell of the DCE.

d) Management of messages for example its fragmenting and reassembly.

e) The communication protocols used. RPC can communicate over TCP/IP and UDP.

16.4.5 Communication Methodology Using RPC

The Following are the commonly used communication methodologies in RPC.

Creation of the IDL File:

The interface for RPC is defined in the IDL file and not the actual procedures. The IDL file advertises the input and output of the Services offered by the remote server. The IDL file is written based on the server's procedure and then compiled using the IDL compiler. Compilation of the IDL file produce client and server stubs.

Client's View of the RPC:

The client is then provided with the Stubs generated by the compilation of the IDL file and it is incorporated into its procedure calls. A simple procedure call is now converted to a complex RPC over the network.

Server's View of RPC:

The server side has the subroutine to perform the function as given in the IDL. The server receives the parameters passed through the IDL and performs the procedure execution and sends back the results as published in the IDL to the client.

Binding:

The client finds the appropriate server to send the remote call by looking up the server's services. This is called Binding. The server when it starts must advertise the services it provides by registering with the directory services. The client then accesses the directory service to find about the server, which offers its services and then addresses that server.

16.4.6 Advantages of Using DCE RPC

The following are the advantages that are obtained by using the DCE RPC

Operating System Independence:

The RPC calls do not depend on the underlying OS's network calls mechanism

Machine Independence:

Even if the machines connecting through RPC are different, RPC can be successfully used as it provides the instructions in native format for both the client and the server.

Language Independence:

Any modern programming language can access the stubs and the skeletons that are produced by the IDL compiler.

Protocol Independence:

The server when registering with the DCE directory service explicitly states the protocol that it uses. Hence the clients can use that protocol or access a different server. The connection oriented and connection free protocols can be interchangeably used.

16.4.7 Security Inbuilt in RPC

The secure RPC is called the Authenticated RPC. There are various levels of authentication,

a) None – No Authentication
b) Connection – Authentication through encryption occurs at the first connection or handshake
c) Call Authentication – The first data packet which is sent to the server is authenticated
d) Packet Authentication – Each packet of data sent through the RPC Interface is authenticated

In addition to these levels packet integrity and privacy can be protected by the use of Cryptographic Checksums.

16.4.8 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. Figure below 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.

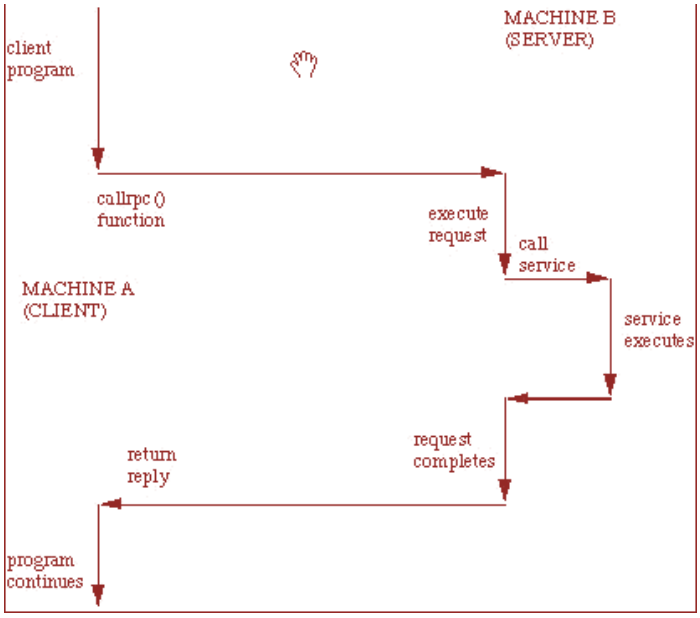


Figure 16.5 Remote Procedure Calling Mechanism

16.5 Distributed Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

Advantages:

- Shields programmer from Send/Receive primitives
- Single address space; simplifies passing-by-reference and passing complex data structures
- Exploit locality-of-reference when a block is moved

- No memory access bottleneck, as no single bus
- Large virtual memory space
- DSM programs portable as they use common DSM programming interface

Disadvantages:

- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than message-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own message- passing solutions.

Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:

- Large Address Spaces
- Protection
- Memory Mapping
- Fair Physical Memory Allocation
- Shared Virtual Memory

Threads can allocate and de-allocate blocks of memory, called **Segments**. These segments can be read and written and can be mapped into and out of the address space of the process. A process owns at least one segment but may have many more of them. Segments can be used for text, data, stack, or any other purpose the process desires. The operating system does not enforce any particular pattern on segment usage.

16.6 Summary

Let us sum up the different concepts we have studied till here.

- A distributed computing system is a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals and communication between any two processors of the system takes place by message passing over the communication network.

- A distributed system includes both a hardware aspect and a software aspect. From a hardware point distributed systems are multi-computers and from a software view point they are distributed operating systems or middleware.
- A distributed operating system (DOS) is built, from the ground up, to provide distributed services.
- The message-passing is a common paradigm model for distributed computing, in the sense that it mimics the behavior in human communications.
- RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler clients transparently make remote calls through a local procedure interface.
- DSM uses simpler software interfaces, and cheaper o_-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems

Self - Assessment Exercise

1. Explain the various reasons for designing applications in Distributed Processing system
2. Explain the features of Concurrency Control in Distributed Computing Environment.
3. List down various application areas where distributed computing is used

Unit 17: Distributed Computing System – An Introduction

17.0 Objective

This unit covers a new task execution strategy called "Distributed Computing" and the following related terminologies.

- Distributed Computing System (DCS)
- Distributed Computing models
- Advantages of Distributed computing
- Security

17.1 Introduction

Distributed computing is a method of computer processing in which different parts of a program are run simultaneously on two or more computers that are communicating with each other over a network. Distributed computing is a type of **segmented** or parallel computing, but the latter term is most commonly used to refer to processing in which different parts of a program run simultaneously on two or more processors that are part of the same computer. While both types of processing

require that a program be segmented—divided into sections that can run simultaneously, distributed computing also requires that the division of the program take into account the different environments on which the different sections of the program will be running. For example, two computers are likely to have different file systems and different hardware components.

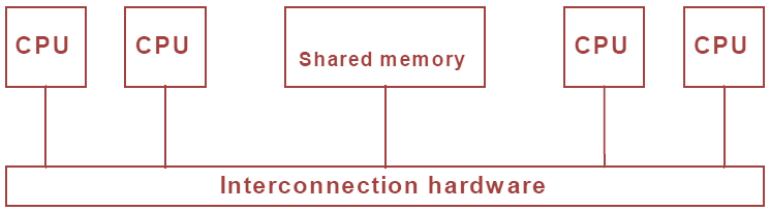
An example of distributed computing is BOINC, a framework in which large problems can be divided into many small problems which are distributed to many computers. Later, the small results are reassembled into a larger solution. Distributed computing is a natural result of using networks to enable computers to communicate efficiently. But distributed computing is distinct from computer networking or **fragmented** computing. The latter refers to two or more computers interacting with each other, but not, typically, sharing the processing of a single program. The World Wide Web is an example of a network, but not an example of distributed computing.

Advancements in microelectronic technology have resulted in the availability of fast, inexpensive processors, and advancements in communication technology have resulted in the availability of cost-effective and highly efficient computer networks. The net result of the advancements in these two technologies is that the price performance ratio has now changed to favor the use of inter-connected, multiple processors in place of a single, high-speed processor.

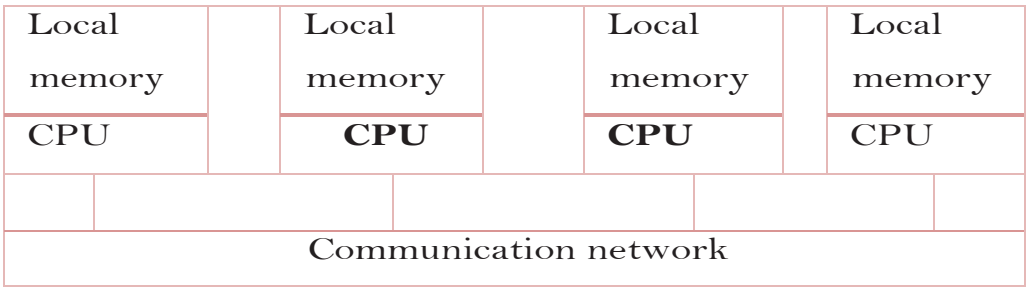
17.1.1 Basic Multiprocessor Systems

Two basic types of computer architectures consisting of interconnected, multiple processors can be distinguished as

1. ***Tightly Coupled Systems*** - systems with a single system wide primary memory (address space) that is shared by all the processors (also referred to as parallel processing systems, multiprocessors, SMMP - shared memory multiprocessors, SMS - shared memory systems, SMP – symmetric multiprocessors).



2. ***Loosely Coupled Systems*** - the systems where processors do not share memory and each processor has its own local memory (also referred to as distributed computing systems, multicomputer, DMS- distributed memory systems, MPP – massively parallel processors).



Let us see some points with respect to both tightly coupled multiprocessor systems and loosely coupled multiprocessor.

- Tightly coupled systems are referred to as parallel processing systems, and loosely coupled systems are referred to as distributed computing systems, or simply distributed systems.
- In case of tightly coupled systems, the processors of distributed computing systems can be located far from each other to cover a wider geographical area.
- In tightly coupled systems, the number of processors that can be usefully deployed is usually small and limited by the bandwidth of the shared memory.
- The Distributed computing systems are more freely expandable and can have an almost unlimited number of processors.

This has provided us with the basic idea of designing distributed operating systems. Although the field is still immature, with ongoing active research activities, commercial distributed operating systems have already started to emerge. These systems are based on already established basic concepts.

17.2 Distributed Computing System – An Outline

It is a collection of independent computers (nodes, sites) interconnected by transmission channels, that appear to the users of the system as a single computer.

Each node of distributed computing system is equipped with a processor, a local memory, and interfaces. Communication between any pair of nodes is realized only by message passing as no common memory is available. Usually, distributed systems are asynchronous, i.e., they do not use a common clock and do not impose any bounds on relative processor speeds or message transfer times.

17.3 Evolution of Distributed Computing System

Early computers were very expensive (they cost millions of dollars) and very large in size (they occupied a big room). These computers were run from a console by an operator and were not accessible to ordinary users. The job setup time was a real problem in early computers and wasted most of the valuable central processing unit (CPU) time. To increase CPU utilization several new concepts were introduced in the 1950s and 1960s like the batching together of jobs with similar needs before processing them, automatic sequencing of jobs, off-line processing by using the concepts of buffering and spooling and multiprogramming. Finally, multiprogramming improved CPU utilization by organizing jobs so that the CPU always had something to execute.

However, none of these ideas allowed multiple users to directly interact with a computer system and to share its resources simultaneously. It was not until the early 1970s that computers started to use the concept of time-sharing to overcome this hurdle. Parallel advancements in hardware technology allowed reduction in the size and increase in the processing speed of computers, causing large-sized computers to be gradually replaced by smaller and cheaper ones that had more processing capability than their predecessors.

The advent of time-sharing systems **was the first step was** distributed computing systems **because it provided us with two important concepts used in distributed computing systems-**

- The sharing of computer resources simultaneously by many users
- The accessing of computers from a place different from the main computer room.

However, in parallel, there were advancements in computer networking technology in the late 1960s and early 1970s that emerged as two key networking technologies-

- **LAN (Local Area Network)**: The LAN technology allowed several computers located within a building or a campus to be interconnected in such a way that these machines could exchange information with each other at data rates of about 10 megabits per second (Mbps). The first high- speed LAN was the Ethernet developed at Xerox PARC in 1973
- **WAN Technology:** allowed computers located far from each other (may be in different cities or countries or continents) to be interconnected in a such a way that these machines could exchange information with each other at data rates of about 56 kilobits per second (Kbps). The first WAN was the ARPANET (Advanced Research Projects Agency Network) developed by the U.S. Department of Defense in 1969.

- **ATM Technology**: The data rates of networks continued to improve gradually in the 1980s providing data rates of up to 100 Mbps for LANs and data rates of up to 64 Kbps for WANs. Recently (early 1990s) there have been another major advancements in networking technology – the ATM (Asynchronous Transfer Mode) technology. The ATM technology is an emerging technology that is still not very well established. It will make very high speed networking possible, providing data transmission rates up to 1.2 gigabits per second (Gbps) in both LAN and WAN environments. The availability of such high-bandwidth networks will allow future distributed computing systems to support a completely new class of distributed applications, called multimedia applications, that deal with the handling of a mixture of information, including voice, video and ordinary data. The merging of computer and networking technologies gave birth to Distributed computing systems in the late 1970s.

17.4 Distributed Computing System Models

Various models are used for building distributed computing system. These models can be broadly classifies into five categories-minicomputer, workstation, workstation-server, processor-pool and hybrid**. They are briefly described below.**

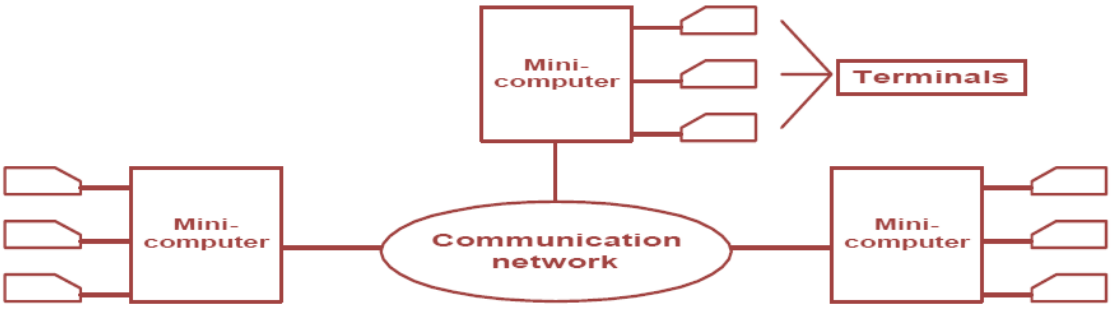
Distributed System

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

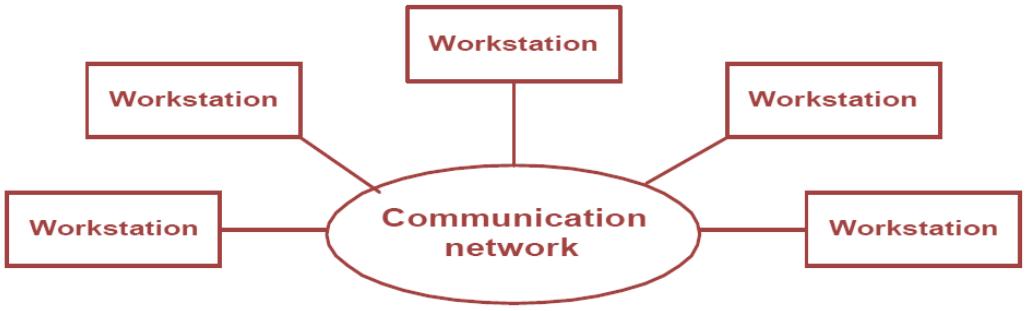
Comparison of three Kinds of Multiple CPU Systems

17.4.1 Minicomputer Model

Distributed computing system based on this model consists of a few minicomputers (or supercomputers) interconnected by a communication network. Each minicomputer is connected by several interactive terminals. Each user is logged on to one specific minicomputer, with access to remote resources available on other minicomputers. There is a simple extension of the centralized time sharing system. The example of distributed computing system based on minicomputers is the early ARPA net.



Workstation Model (NOW - Network of Workstations, P2P - Peer-to Peer)



General Characteristic:

- System consists of several workstations interconnected by a communication network.
- Every workstation may be equipped with its own disk and serving as a single-user computer.
- In such environment like company's office or a university department, at any one time (especially at night), a significant portion of the workstation are idle, resulting in the waste of large amount of CPU time.
- Main idea: interconnect all workstations by a high-speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.
- User logs onto one of the workstations and submits job for execution.
- If the user's workstation does not have sufficient processing power for executing the processes of the submitted job efficiently, it transfers one or more of the processes from the user's workstation to some other workstation that is currently idle and gets the process executed there.

- The result of execution is returned to the user's workstation.
- Implementation issues:
  - How does the system find an idle workstation?
  - How is the process transferred from one workstation to get it executed on another workstation?
  - What happens to a remote process if a user logs onto a workstation that was idle until now and was being executed a process of another workstation?
- Examples of distributed computing systems based on the workstation model:
  - Sprite system; experimental system developed at Xerox PARC.

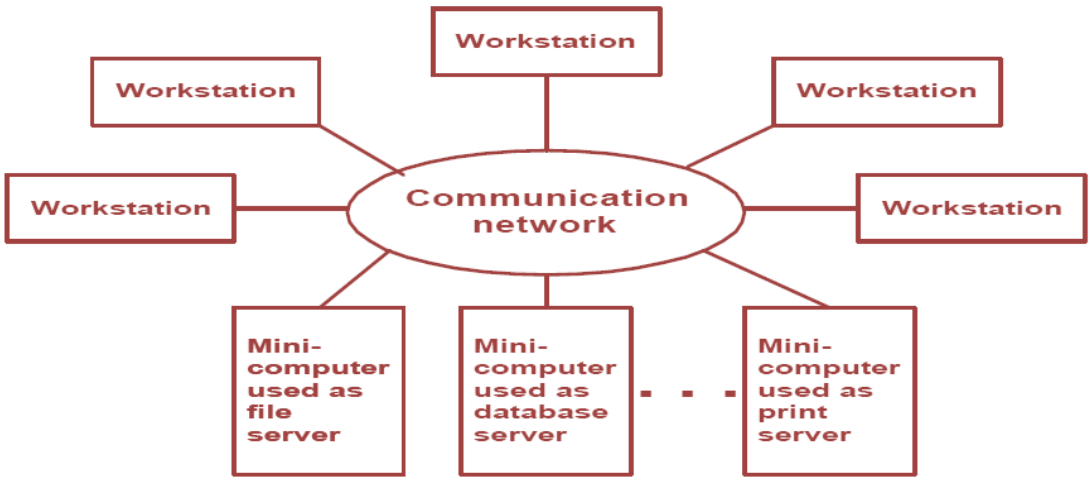
17.4.2 Workstation-Server Model

System consists of a few minicomputers and several workstations (diskless or diskful) interconnected by a communication network. In addition to the workstation, there are specialized machines running server processes (*servers*) for managing and providing access to shared resources.

Each minicomputer is used as a server machine to provide one or more types of service:

- implementing the file system;
- database service;
- print service;
- other types of service.

User logs onto a workstation called his home workstation. Normal computation activities required by the user's processes are performed at the user's home workstation. Requests for services provided by special servers are sent to a server providing that type of service that performs the user's requested activity and returns of requested processing to the user's workstation. User's processes need not be migrated to the server machines for getting the work done by those machines.

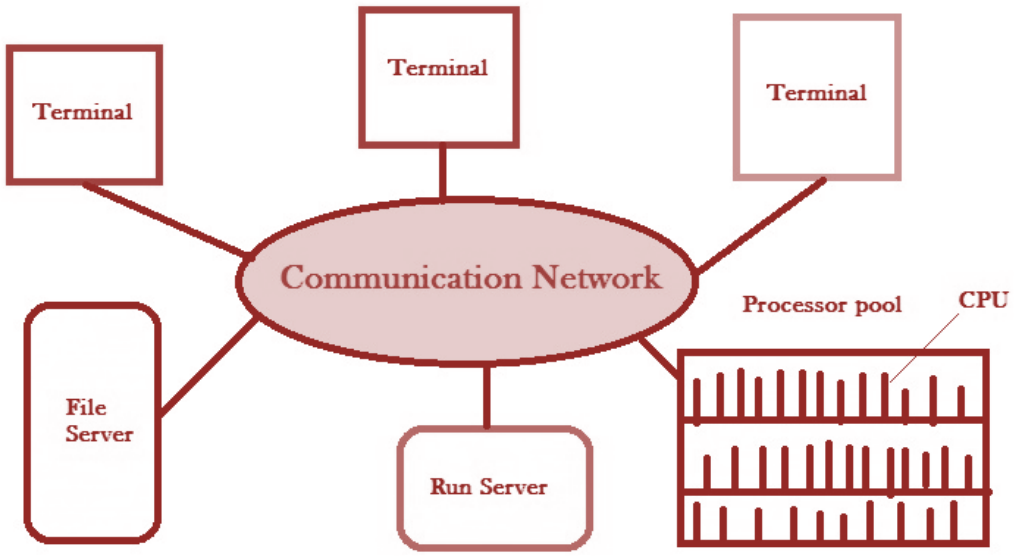


The workstation-server model has several advantages:

1. In general, it is much cheaper to use a few minicomputers equipped with large, fast disk that are accessed over the network than a large number of diskful workstations, with each workstation having a small, slow disk.
2. Diskless workstations are also preferred to diskful workstations from a system maintenance point of view. Backup and hardware maintenance are easier to perform with a few large disks than with many small disks scattered all over a building or campus. Furthermore, installing new releases of software (such as file server with new functionalities) is easier when the software is to be installed on a few file server machines than on every workstation.
3. In the workstation-server model, since the file servers manage all files, users have the flexibility to use any workstation and access the files in the same manner irrespective of which workstation the user is currently logged on. Note that this is not the true with the workstation model, in which the workstation model, in which each workstation has its local file system, because different mechanisms are needed to access local and remote files.
4. In the workstation-server model, the request-response protocol is mainly used to access the services of the server machines. Therefore, unlike the workstation model, this model does not need a process migration facility, which is difficult to implement. The request-response protocol is known as the client-server model of communication. In this model, a client process (which in this case resides on a workstation) sends a request to server process (which in this case resides on a minicomputer) for getting some services such as reading a unit of a file. The server executes the request and sends back a reply to the client that contains the result of processing.
5. A user has guaranteed response time because workstations are not used for executing

remote processes. However, the model does not utilize the processing capability of idle workstations.

17.4.3 Processor-Pool Model



The model is based on the observation that most of the time a user does not need any computer power but once in a while he may need a very large amount of computing power for a short time.

- The processors are pooled together to be shared by the users as needed.
- The pool of processors consists of a large number of microcomputers and minicomputers attached to the network.
- Each processor in the pool has its own memory to load and run a system program or an application program.
- The processors in the pool have not terminals attached directly to them, and users access the system from terminals that are attached to the network via special devices.
- A special server (*run server*) manages and allocates the processors in the pool to different users on a demand basis.
- Appropriate number of processors are temporary assigned to user's job by the run server.
- When the computation is completed, the processors are returned to the pool.

When a user submits a job for computation, the run server temporarily assigns an appropriate

number of processors to his or her job. For example, if the user's computation job is the compilation of a program having n segments, in which each of the segments can be compiled independently to produce separate relocatable object files, n processors from the pool can be allocated to this job to compile all the n segments in parallel. When the computation is completed, the processors are returned to the pool for use by other users.

In the processor-pool model there is no concept of a home machines. That is, a user does not log onto a particular machine but to the system as a whole. This is in contrast to other models in which each user has a home machine (e.g., a workstation or minicomputer) onto which he or she logs and runs most of his or her programs there by default. Amoeba and the Cambridge Distributed Computing Systems are examples of distributed computing systems based on the processor-pool model.

17.5 Security in Distributed Environment

Computing security is, at its core, more than a technical issue: It's a fundamental business challenge. Managers have plenty of security alternatives, but little real guidance on making intelligent decisions about them. And today's distributed, multivendor, Internet-connected environments encompass more insecure systems and networks than ever before.

Security in these systems is multilayered and can be tailored to meet company and user-specific needs Security in Distributed Computing offers the manager of distributed systems a thorough, common-sense framework for cost-effective computer security.

Security Policy determines precisely which actions the entities in a system are allowed to take and which ones are prohibited. The security policy can be enforced by following techniques:

ENCRYPTION: Transforming data into coded text that an attacker cannot understand.

AUTHENTICATION: Verifying the claimed identity of different entities.

AUTHORIZATION: Verifying whether the client is allowed to perform the requested service..

AUDITING: Tracing each and every client's activity.

The security in distributed systems requires use of a unique, assigned login name for each user. This name must be used in conjunction with the system provided or created password unique to the username to gain access.

The security protocol of distributed systems requires it check the login name and password against its files along with the access point to authenticate login. The system is unique in that it can accomplish this without an active server.

Each user has a personally constructed security profile. This profile only allows them access to certain areas of the files and programs located within the distributed system. This security protocol helps to keep information confidential by only allowing limited access.

17.5.1 Architecture

Various hardware and software architectures are used for distributed computing. At a lower level, it is necessary to interconnect multiple CPUs with some sort of network, regardless of whether that network is printed onto a circuit board or made up of loosely coupled devices and cables. At a higher level, it is necessary to interconnect processes running on those CPUs with some sort of communication system.

Distributed programming typically falls into one of several basic architectures or categories: Client-server, 3-tier architecture, N-tier architecture, Distributed objects, loose coupling, or tight coupling.

- **Client-server**

  Smart client code contacts the server for data, then formats and displays it to the user. Input at the client is committed back to the server when it represents a permanent change.

- **3-tier architecture**

  Three tier systems move the client intelligence to a middle tier so that stateless clients can be used. This simplifies application deployment. Most web applications are 3- Tier.

- **N-tier architecture**

  N-Tier refers typically to web applications which further forward their requests to other enterprise services. This type of application is the one most responsible for the success of application servers.

- **Tightly coupled (clustered)**

  refers typically to a set of highly integrated machines that run the same process in parallel, subdividing the task in parts that are made individually by each one, and then put back together to make the final result.

- **Peer-to-peer**

  an architecture where there is no special machine or machines that provide a service or manage the network resources. Instead, all responsibilities are uniformly divided among all machines, known as peers. Peers can serve both as clients and servers.

- **Space based**

  refers to an infrastructure that creates the illusion (virtualization) of one single address-space. Data are transparently replicated according to application needs. Decoupling in time, space and reference is achieved.

Another basic aspect of distributed computing architecture is the method of communicating and coordinating work among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master/slave relationship. Alternatively, a "database-centric" architecture can enable distributed computing to be done without any form of direct inter-process communication, by utilizing a shared database.

17.6 Advantages of Distributed System Over Centralized System

The distributed computing systems are much more complex and difficult to build than traditional centralized systems (those consisting of a single CPU, its memory, peripherals, and one or more terminals). The increased complexity is mainly due to the fact that in addition to being capable of effectively using and managing a very large number of distributed resources, the system software of a distributed computing system should also be capable of handling the communication and security problems that are very different from those of centralized systems. For example, the performance and reliability of a distributed computing system depends to a great extent on the performance and reliability of the underlying communication network. Special software is usually needed to handle loss of messages, during transmission across the network or to prevent overloading of the network that degrades the performance and responsiveness to the users. Similarly, special software security measures are needed to protect the widely distributed shared resources and services against intentional or accidental violation of access control and privacy constraints.

Despite the increased complexity and the difficulty of building distributed computing systems, the installation and use of distributed computing systems overweigh their disadvantages. The technical needs, the economic pressures, and the major advantages that have led to the emergence and popularity of distributed computing systems are described here.

- **Inherently Distributed Applications:** Distributed computing systems come into existence in some very natural easy. For example, several applications are inherently distributed in nature and require a distributed computing system for their realization. For instance, in

an employee database of a nationwide organization, the data pertaining to a particular employee are generated at the employee's branch office, and in addition to the global need to view the entire database; there is a local need for frequent and immediate access to locally generated data at each branch office. Such applications require that some processing power be available at the many distributed locations for collecting, preprocessing, and accessing data, resulting in the need for distributed application are a computerized worldwide airline reservation system, a computerized banking system in which a customer can deposit/withdraw money from his or her account from any branch of the bank, and a factory automation system controlling robots and machines all along an assembly line.

- **Information Sharing Among Distributed Users:** Efficient person-to-person communication facility by sharing information over great distances is the one more advantage. In a distributed computing system, the users working at other nodes of the system can easily and efficiently share information generated by one of the users. This facility may be useful in many ways. For example, two or more users who are geographically far off from each other can perform a project but whose computers are the parts of the same distributed computing system.

  The use of distributed computing systems by a group of users to work cooperatively is known as computer-supported cooperative working (CSCW), or groupware.

- **Resource Sharing:** Information is not the only thing that can be shared in a distributed computing system. Sharing of software resources such as software libraries and databases as well as hardware resources such as printers, hard disks, and plotters can also be done in a very effective way among all the computers and users of a single distributed computing system

- **Better Price Performance Ration:** This is one of the most important reasons for the growing popularity of distributed computing system. With the rapidly increasing power and reduction in the price of microprocessors, combined with the increasing speed of communication networks, distributed computing systems potentially have a much better price-performance ratio than a single large centralized system. Another reason for distributed computing systems to be more cost effective than centralized systems is that

they facilitate resource sharing among multiple computers.

- **Shorter Response Times and Higher Throughput:** Due to multiplicity of processors, distributed computing systems are expected to have better performance than single-processor centralized systems. The two most commonly used performance metrics are response time and throughput of user processes. That is, the multiple processors of distributed computing systems can be utilized properly for providing shorter response times and higher throughput than a single processor centralized system. Another method often used in distributed computing systems for achieving better overall performance is to distribute the load more evenly among the multiple processors by moving jobs from currently overloaded processors to lightly loaded ones.

- **Higher Reliability:** *Reliability* refers to the degree of tolerance against errors and component failures in a system. A reliable system prevents loss of information even in the event of component failures. The multiplicity of storage devices and processors in a distributed computing system allows the maintenance of multiple copies of critical information within the system. With this approach, if one of the processors fails, the computation can be successfully completed at the other processor, and if one of the storage devices fails, the computations can be successfully completed at the other processors, and if one of the storage devices fails, the information can still be used from the other storage device.

- **Availability:** An important aspect of reliability is *availability*, which refers to the fraction of time for which a system is available for use. In comparison to a centralized system, a distributed computing system also enjoys the advantage of increased availability.

- **Extensibility and Incremental Growth:** Another major advantage of distributed computing systems is that they are capable of **incremental growth**. That is, it is possible to gradually extend the power and functionality of a distributed computing system by simply adding additional resources (both hardware and software) to the system as and when the need arises. For example, additional processors can be easily added to the system to handle the increased workload of an organization that might have resulted from its expansion. **Extensibility** is also easier on a distributed computing system because addition of new

resources to an existing system can be performed without significant disruption of the normal functioning of the system. Properly designed distributed computing systems that have the property of extensibility and incremental growth are called ***open distributed systems***.

- **Better Flexibility in Meeting User's Needs:** Different types of computers are usually more suitable for performing different types of computations. For example, computers with ordinary power are suitable for ordinary data processing jobs, whereas high-performance computers are more suitable for complex mathematical computations. In a centralized system, the users have to perform all types of computations on the only available computer.

17.7 Disadvantages of Distributed System Over Centralized System

Technical Issues

If not planned properly, a distributed system can decrease the overall reliability of computations if the unavailability of a node can cause disruption of the other nodes. Leslie Lamport famously quipped that: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." Troubleshooting and diagnosing problems in a distributed system can also become more difficult, because the analysis may require connecting to remote nodes or inspecting communication between nodes. Many types of computation are not well suited for distributed environments, typically owing to the amount of network communication or synchronization that would be required between nodes. If bandwidth, latency, or communication requirements are too significant, then the benefits of distributed computing may be negated, and the performance may be worse than a non-distributed environment.

Project-related Problems

Distributed computing projects may generate data that is proprietary to private industry, even though the process of generating that data involves the resources of volunteers. This may result in controversy as private industry profits from the data which is generated with the aid of volunteers. In addition, some distributed computing projects, such as biology projects that aim to develop thousands or millions of "candidate molecules" for solving various medical problems, may create vast amounts of raw data. This raw data may be useless by itself without refinement of the raw

data or testing of candidate results in real-world experiments. Such refinement and experimentation may be so expensive and time-consuming that it may literally take decades to sift through the data. Until the data is refined, no benefits can be acquired from the computing work.

Other projects suffer from lack of planning on behalf of their well-meaning originators. These poorly planned projects may not generate results that are palpable or may not generate data that ultimately result in finished, innovative scientific papers. Sensing that a project may not be generating useful data, the project managers may decide to abruptly terminate the project without definitive results, resulting in wastage of the electricity and computing resources used in the project. Volunteers may feel disappointed and abused by such outcomes. There is an obvious opportunity cost of devoting time and energy to a project that ultimately is useless, when that computing power could have been devoted to a better planned distributed computing project generating useful, concrete results.

Another problem with distributed computing projects is that they may devote resources to problems that may not ultimately be soluble, or to problems that are best pursued later in the future, when desktop computing power becomes fast enough to make pursuit of such solutions practical. Some distributed computing projects may also attempt to use computers to find solutions by number-crunching mathematical or physical models. With such projects there is the risk that the model may not be designed well enough to efficiently generate concrete solutions. The effectiveness of a distributed computing project is therefore determined largely by the sophistication of the project creators.

17.7.1 Network Operating Systems

There are two Types of Distributed operating systems:

- Network Operating Systems
- Distributed Operating Systems

In the Network operating systems Users are aware of multiplicity of machines. User's access to resources of various machines is done explicitly by:

- Remote logging into the appropriate remote machine (telnet, ssh)
- Remote Desktop (Microsoft Windows)
- Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism

17.8 Issues in Designing a Distributed Operating System

In general, designing operating system is more difficult than designing a centralized operating system for several reasons. In the design of a centralized operating system, it is assumed that the operating system has access to complete and accurate information about the environment in which it is functioning. In a distributed system, the resources are physically separated, there is no common clock among the multiple processors, delivery of messages is delayed, and messages could even be lost. Due to all these reasons, a distributed operating system doest not have up-to-data, consistent knowledge about the state of the various components of the underlying distributed system. Despite these complexities and difficulties, a distributed operating system must be designed to provide all the advantage of a distributed system to its users. That is, the users should be able to view a distributed system as virtual centralized system that is flexible, efficient, reliable, secure and easy to use. To meet this requirement, the designers of a distributed operating system must deal with several design issues. Some of the key design issues are described below.

- **Transparency:** We saw that one of the main goals of a distributed operating system is to make the existence of multiple *computers invisible (transparent)* and provide a single system image to its users. That is, a distributed operating system must be designed in such a way that a collection of distinct machines connected by a communication subsystem appears to its users as a *virtual uniprocessor*. The eight forms of transparency identified by the International Standards Organization's Reference Model for Open Distributed Processing [ISO 1992] are ***access transparency, location transparency, replication transparency, failure transparency, migration transparency, location transparency, concurrency transparency performance transparency, and scaling transparency***.

- **Reliability:** In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources. However, the existence of multiple instances of the resources alone cannot increase the systems reliability. Rather, the distributed operating system, which manages these resources, must be designed properly to increase the systems reliability by taking full advantage of this characteristic feature of a distributed system.

  For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover from faults. Commonly used methods for dealing with these issues are briefly described here.

- **Flexibility:** Another important issue in the design of distributed operating systems is flexibility. The design of a distributed operating system should be flexible due to the following reasons:

- ***Ease of modification:*** From the experience of system designers, it has been found that some parts of the design often need to be replaced/modified either because some bug is detected in the design or because the design is no longer suitable for the changed system environment or new-user requirements. Therefore, it should be easy to incorporate changes in the system in a user-transparent manner or with minimum interruption caused to the users.
- ***Ease of enhancement:*** In every system, new functionalities have to be added from time to time to make it more powerful and easy to use. Therefore, it should be easy to add new services to the system.

  The most important design factor that influences the flexibility of a distributed operating system is the model used for designing its kernel. The *kernel* of an operating system is its central controlling part that provided basic system facilities. It operates in a separate address space that a user cannot replace or modify. The two commonly used models for kernels design in distributed operating systems are **monolithic kernel and the micro kernel.**
- **In *monolithic kernel*** model, the kernel provides most operating system services such as process management, and inter-process communication. As a result, the kernel has a large, monolithic structure. Many distributed operating systems that are extensions or imitations of the UNIX operating system use the monolithic kernel model. This is mainly because UNIX itself has a large, monolithic kernel.
- ***In the micro kernel*** model, the main goal is to keep the kernel as small as possible. Therefore, in this model, the kernel is a very small nucleus of software that provides only the minimal facilities necessary for implementing additional operating system services. The only services provided by the kernel in this model are inter-process communication, low–level device management and some memory management. All other operating system services, such as file-management, name management, additional process and memory management activities, and much system call handling are implemented as a user-level server processes.

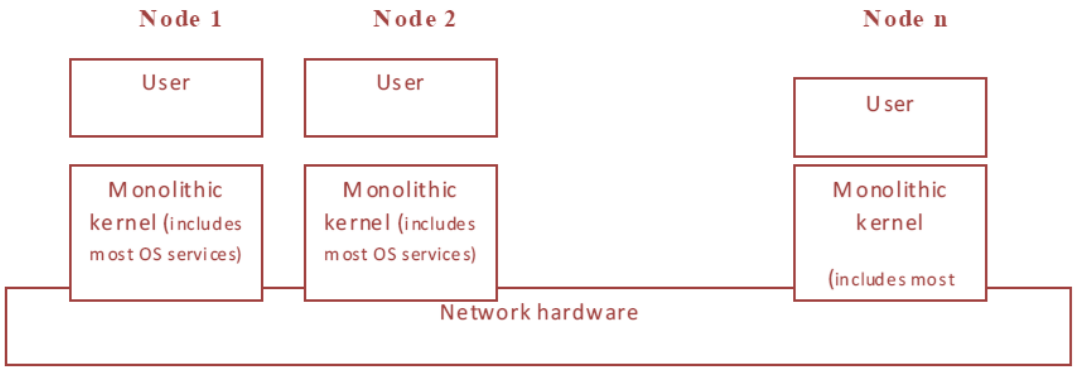


Fig 17.6(a) The Monolithic Kernel Model

- **Performance:** If a distributed system is to be used, its performance must be at least as good as a centralized system. That is, when a particular application is run on a distributed system, its overall performance should be better than or at least equal to that of running the same applications on a single-processor system. However, to achieve this goal, it is important that the various components of the operating system of a distributed system be designed properly; otherwise, the overall performance of the distributed system may turn out to be worse than a centralized system.
- **Scalability:** Scalability refers to the capability of a system to adapt to increased service load. It is inevitable that a distributed system will grow with time since it is very common to add new machines or an entire sub-network to the system to take care of increased workload or organizational changes in a company. Therefore, a distributed operating system should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users.
- **Heterogeneity:** A heterogeneous distributed system consists of interconnected sets of dissimilar hardware or software systems. Because of the diversity, designing heterogeneous distributed systems is far more difficult than designing homogenous distributed systems in which each system is based on the same, or closely related, hardware and software. However, as a consequence of large scale, heterogeneity is often inevitable in distributed systems. Furthermore, many users prefer often heterogeneity because heterogeneous distributed systems provide the flexibility to their users of different computer platforms for different applications.
- **Security:** In order that the users can trust the system and rely on it, the various

resources of a computer system must be protected against destruction and unauthorized access. Enforcing security in a distributed system is more difficult than in a centralized system because of the lack of a single point of control and the use of insecure networks for data communication. In a centralized system, all users are authenticated by the system at login time, and the system can easily check whether a user is authorized to perform the requested operation on an accessed resource. In a distributed system, however, since the client-server model is often used for requesting and providing services, when a client sends a request message to a server, the server must have some way of knowing who is the client. This is not so simple as it might appear because any client identification field in the message cannot be trusted. This is because an intruder (a person or program trying to obtain unauthorized access to system resources) may pretend to be an authorized client or may change the message contents during transmission. Therefore, as compared to a centralized system, enforcement of security in a distributed system has the following additional requirements:

1. It should be possible for the sender of a message to know that the intended receiver received the message.
2. It should be possible for receiver of a message to know that the message was sent by the genuine sender
3. It should be possible for both the sender and receiver of a message to be guaranteed that the contents of the message were not changed while it was in transfer.

- **Emulation of Existing Operating Systems:** For commercial success, it is important that a newly designed distributed operating system be able to emulate existing poplar operating systems such as UNIX. With this property, new software can be written using the system call interface of the new operating system to take full advantage of its special features of distribution, but a vast amount to already existing old software can also be run on the same system without the need to rewrite them. Therefore, moving to the new distributed operating system will allow both types of software to be run side by side.

17.9 Summary

Let us sum up the different concepts we have studied till here.

- The existing models for distributed computing systems can be broadly classified into five categories, minicomputer, workstation-server, processor-pool and hybrid.
- Distributed computing system is much more complex and difficult to build than the traditional centralized systems. Despite the increased complexity and the difficulty of buildings, the installation and the use of distributed computing system are rapidly increasing. This is mainly because the advantages of distributed computing systems outweigh its disadvantages.
- The main advantages of distributed computing systems are (a) suitability for inherently distributed applications. (b) Sharing of information among distributed users and sharing of resources (d) better price performance ratio (e) shorter response times and higher throughout

  (f) higher reliability (g) extensibility and incremental growth and (h) better flexibility in meeting user's needs.
- The operating systems commonly used for distributed computing systems can be broadly classified into two types: *network operating systems and distributed operating systems.* As compared to a network operating system, a distributed operating system has better transparency and fault capability and provides the image of a virtual uniprocessor to the users.
- The main issue involved in the design of a distributed operating system is transparency, reliability, flexibility, performance, scalability, heterogeneity, security and emulation of existing operating systems.

Self - Assessment Exercise

1. Differentiate between Centralised approach and Fully Distributed Approach
2. Identify the disadvantages of distributed approach in comparison to Centralised approach.
3. What are the main issues involve in the design of a distributed system?
4. Explain the different models of Distributed Computing.
5. What are the security issues related to a system with distributed approach?

Unit 18: Distributed File System

18.0 Objective

This unit covers following aspects:

- Features of Good Distributed File System
- File Models & File Accessing Models
- File- Sharing Semantics
- System Design Issue & Design Principle

18.1 Introduction

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. A Distributed File System is a network file system where a single file system can be distributed across several physical computer nodes. Separate nodes have direct access to only a part of the entire file system, in contrast to shared disk file systems where all nodes have uniform direct access to the entire storage. Example: Google file system, CODA, Hadoop.

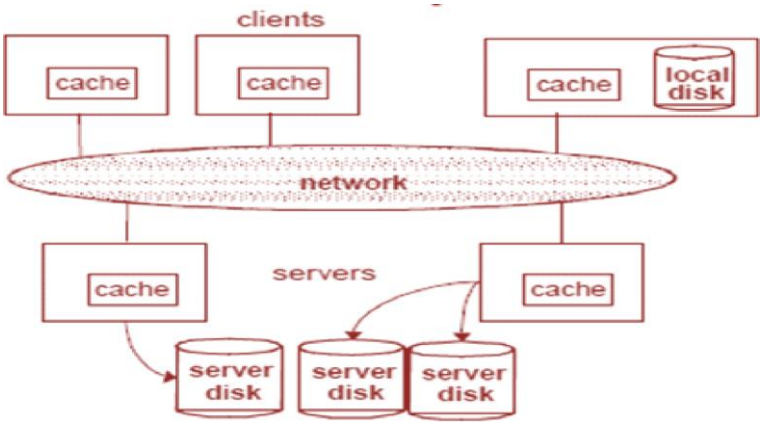


Figure 18.1: Distributed File System

18.2 Features of Good DFS

1) **Fault tolerance**: Distributed storage is composed of a large number of distributed storage components (rather than a single storage component). Increasing the number of components affects fault tolerance. Distributing the components makes the system more faults prone (because of network disruptions).

2) **Scalability:** The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).

3) **Transparency:** The distributed systems should be perceived as a single entity by the users or the application programmers rather than as a collection of autonomous systems, which are cooperating. The users should be unaware of where the services are located and also the transferring from a local machine to a remote one should also be transparent.

   a. **Access transparency** Clients are unaware that files are distributed and can access them in the same way as local files are accessed.

   b. **Location transparency** A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.

   c. **Concurrency transparency** All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.

   d. **Failure transparency** The client and client programs should operate correctly after a server failure.

   e. **Heterogeneity** File service should be provided across different hardware

and operating system platforms.

f. **Replication transparency** To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.

g. **Migration transparency** Files should be able to move around without the client's knowledge.

Other Characteristics Include

1) **Network Transparency:** Same access operation as if they are local files.
2) **Location Independence:** The file name should not be changed when the physical location of the file changes.
3) **User Mobility:** User should be able to access the file from anywhere.
4) **File Mobility:** Moves files from one place to the other in a running system.

18.3 File models & File Accessing Models

18.3.1 Distributed File System Concepts

A file service is a specification of what the file system offers to clients. A file server is the implementation of a file service and runs on one or more machines. A file itself contains a name, data, and attributes (such as owner, size, creation time, access rights). An immutable file is one that, once created, cannot be changed. Immutable files are easy to cache and to replicate across servers since their contents are guaranteed to remain unchanged. Two forms of protection are generally used in distributed file systems, and they are essentially the same techniques that are used in single-processor non-networked systems:

Capabilities

Each user is granted a ticket (capability) from some trusted source for each object to which it has access. The capability specifies what kinds of access are allowed.

Access Control Lists

Each file has a list of users associated with it and access permissions per user. Multiple users may be organized into an entity known as a group.

18.3.2 File Service Types

To provide a remote system with file service, we will have to select one of two models of operation. One of these is the upload/download model. In this model, there are two fundamental operations: read file transfers an entire file from the server to the requesting client and write file copies the

file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file? The second model is a remote access model. The file service provides remote operations such as open, close, read bytes, write bytes, get attributes, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.

Another important distinction in providing file service is that of understanding the difference between directory service and file service. A directory service, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the file service. The file service itself provides the file interface (this is mentioned above). Another component of file distributed file systems is the client module. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the vnode file system layer of a UNIX kernel).

18.3.3 Naming Issues

In designing a distributed file service, we should consider whether all machines (and processes) should have the exact same view of the directory hierarchy. We might also wish to consider whether the name space on all machines should have a global root directory (a.k.a. super root) so that files can be accessed as, for example, //server/path. This is a model that was adopted by the Apollo Domain System, an early distributed file system, and more recently by the web community in the construction of a uniform resource locator (URL).

In considering our goals in name resolution, we must distinguish between location transparency and location independence. By location transparency we mean that the path name of a file gives no hint to where the file is located. For instance, we may refer to a file as //server1/dir/file. The server (server) can move anywhere without the client caring, so we have location transparency. However, if the file moves to server2 things will not work. If we have location independence, the files can be moved without their names changing. Hence, if machine or server names are embedded into path names, we do not achieve location independence. It is desirable to have access transparency, so that applications and users can access remote files just as they access local files. To facilitate this, the remote file system name space should be syntactically consistent with the local name space. One way of accomplishing this is by redefining the way files are named and require an explicit syntax for identifying remote files. This can cause legacy applications to fail and user discontent (users will have to learn a new way of naming their files). An alternate

solution is to use a file system mounting mechanism to overlay portions of another file system over a node in a local directory structure. Mounting is used in the local environment to construct a uniform name space from separate file systems (which reside on different disks or partitions) as well as incorporating special-purpose file systems into the name space (e.g., /proc on many UNIX systems allows file system access to processes). A remote file system can be mounted at a particular point in the local directory tree. Attempts to access files and directories under that node will be directed to the driver for that file system.

To summarize, our naming options are:

- machine and path naming (machine: path, ./machine/path).
- mount remote file systems onto the local directory hierarchy (merging the two-name spaces).
- provide a single name space which looks the same on all machines. The first two of these options are relatively easy to implement.

Types of Names

When we talk about file names, we refer to symbolic names (for example, server.c). These names are used by people (users or programmers) to refer to files. Another "name" is the identifier used by the system internally to refer to a file. We can think of this as a binary name (more precisely, as an address). On most UNIX file systems, this would be the device number and inode number. On MS-DOS systems, this would be the drive letter and FAT index. Directories provide a mapping from symbolic names to file addresses (binary names).

Typically, one symbolic name maps to one file address. If multiple symbolic names map onto one binary name, these are called hard links. On inode-based file systems (e.g., most UNIX systems), hard links must exist within the same device since the address (inode) is unique only on that device. On MS-DOS systems, they are not supported because file attributes are stored with the name of the file. Having two symbolic names refer to the same data will cause problems in synchronizing file attributes (how would you locate other files that point to this data?). A hack to allow multiple names to refer to the same file (whether its on the same device or a different device) is to have the symbolic name refer to a single file address but that file may have an attribute to tell the system that its contents contain a symbolic file name that should be dereferenced. Essentially, this adds a level of indirection: access a file which contains another file name, which references the file attributes and data. These files are known as symbolic links. Finally, it is possible for one symbolic name to refer to multiple file addresses. This doesn't make much sense on a local system1 but can be useful on a networked file system to provide fault tolerance or enable the system to use the file address which is most efficient.

18.4 File- Sharing Semantics

The analysis of file sharing semantics is that of understanding how files behave. For instance, on most systems, if a read follows a write, the read of that location will return the values just written. If two writes occur in succession, the following read will return the results of the last write. File systems that behave this way are said to observe sequential semantics. Sequential semantics can be achieved in a distributed system if there is only one server and clients do not cache data. This can cause performance problems since clients will be going to the server for every file operation (such as single-byte reads). The performance problems can be alleviated with client caching. However, now if the client modifies its cache and another client reads data from the server, it will get obsolete data. Sequential semantics no longer hold. One solution is to make all the writes write-through to the server. This is inefficient and does not solve the problem of clients having invalid copies in their cache. To solve this, the server would have to notify all clients holding copies of the data.

Another solution is to relax the semantics. We will simply tell the users that things do not work the same way on the distributed file system as they did on the local file system. The new rule can be "changes to an open file are initially visible only to the process (or machine) that modified it." These are known as session semantics. Yet another solution is to make all the files immutable2. That is, a file cannot be open for modification, only for reading or creating. If we need to modify a file, we'll create a completely new file under theold name.

Immutable files are an aid to replication, but they do not help with changes to the file's contents (or, more precisely, that the old file is obsolete because a new one with modified contents succeeded it). We still have to contend with the issue that there may be another process reading the old file. It's possible to detect that a file has changed and start failing requests from other processes. A final alternative is to use atomic transactions. To access a file or a group of files, a process first executes a begin transaction primitive to signal that all future operations will be executed indivisibly. When the work is completed, an end transaction primitive is executed. If two or more transactions start at the same time, the system ensures that the end result is as if they were run in some sequential order. All changes have an all or nothing property.

18.5 System Design Issue

18.5.1 Name Resolution

In looking up the pathname of a file (e.g., via the namei function in the UNIX kernel), we may choose to evaluate a pathname a component at a time. For example, for a pathname aaa/bbb/ccc, we would perform a remote lookup of aaa, then another one of bbb, and finally one of ccc). Alternatively, we may pass the rest of the pathname to the remote

machine as one lookup request once we find that a component is remote. The drawback of the latter scheme is

a) The remote server may be asked to walk up the tree by processing .. (parent node) components and reveal more of its file system than it wants and
b) Other components cannot be mounted underneath the remote tree on the local system. Because of this, component at a time evaluation is generally favored but it has performance problems (a lot more messages). We may choose to keep a local cache of component resolutions.

18.5.2 Should Servers Maintain State?

This issue is a topic of passionate debate. A stateless system is one in which the client sends a request to a server, the server carries it out, and returns the result. Between these requests, no client-specific information is stored on the server. A stateful system is one where information about client connections is maintained on the server. In a stateless system:

- Each request must be complete – the file has to be fully identified and any offsets specified.
- Fault tolerance: if a server crashes and then recovers, no state was lost about client connections because there was no state to maintain.
- No remote open/close calls are needed (they only serve to establish state).
- No wasted server space per client.
- No limit on the number of open files on the server; they aren't "open" – the server maintains no per- client state.
- No problems if the client crashes. The server does not have any state to clean up. On a stateful system:
- requests are shorter (less info to send).
- better performance in processing the requests.
- idempotency works; cache coherence is possible.
- file locking is possible; the server can keep state that a certain client is locking a file (or portion thereof)

18.5.3 File Caching Schemes

We can employ caching to improve system performance. There are four places in a distributed system where we can hold data:

1. on the server's disk
2. in a cache in the server's memory

3. in the client's memory
4. on the client's disk

The first two places are not an issue since any interface to the server can check the centralized cache. It is in the last two places that problems arise, and we have to consider the issue of cache consistency. Several approaches may be taken: write-through What if another client reads its own cached copy? All accesses would require checking with the server first (adds network congestion) or require the server to maintain state on who has what files cached. Write-through also does not alleviate congestion on writes. delayed writes Data can be buffered locally (where consistency suffers) but files can be updated periodically. A single bulk write is far more efficient than lots of little writes every time any file contents are modified. Unfortunately, the semantics become ambiguous.

Write on Close

This is admitting that the file system uses session semantics.

Centralized Control

Server keeps track of who has what open in which mode. We would have to support a stateful system and deal with signaling traffic.

18.5.4 Fault Tolerance

In brief, we can say that in a computational system data are processed to produce information. Once produced, usually this information are stored in a media by the file system for further accesses Their necessity of being accessible implies to provide safe mechanisms for storing and accessing data/information which claims fault tolerance issues. Besides, in distributed file systems, not only local failures (e.g., due storage devices) should be dealt with, but also other failures inherent to the distributed environment. Following, we address some issues that are strict related with fault tolerance in distributed ule systems.

Stateful and Stateless Services

To better understand how fault tolerance can be employed in a distributed file system, it is useful to understand how the services are provided. Distributed file systems services can be implemented by using two different service designs: stateful service or stateless service. These paradigms have contradictory concepts; however both supply the file operations.

Stateful Service

In this type of service, information about file operations is kept in the server during all the file session. A communication channel is established between the client and the server when the client explicitly solicits the file opening. A number (identifier) is used to define the communication channel then this identifier will be used to perform file operations. To attend its clients, the server copies data from the storage devices to memory and let them there till the file closing.

Stateless Service

On the other hand, the stateless service does not establish a communication channel. Moreover, there is no necessity for explicit file opening and closing: before executing a file operation the server will automatically open and close the file. Each request sent to the server must define the desired file likewise, if a read or write operation is requested, it must contain the position in the file referring to the respective operation.

The usage of the main memory can improve performance whereas memory access is faster than disk one. The memory can be used for caching in the stateless service but its usage is not obligatory as in stateful service. As a result to this, stateful service presents an advantage when compared to the stateless one. On the contrary, the advantage of using the main memory becomes a disadvantage with respect to the fault tolerance context. If the server crashes, information stored in memory will probably be lost or at least harder to be recovered.

The consequences of servers and clients failures will depend of the used service. If a stateless server crashes, the previous file sessions will not be disturbed. On the other hand, if the server crashes in a stateful service, it should be able to recover the file session state, likewise it has inconveniences as mentioned earlier. Focusing on the client failures, in a stateful service the server should be able to realize when it happens to free the allocated memory. On the contrary, stateless service servers do not need to handle client faults. However, stateless service clients can experience a situation in which they cannot distinguish a slow server from a recovering one.

Finally, we can also compare both approaches referring to service overhead. Due to the communication channel established on the stateful service its overhead is significant lower than the stateless service. The reason is that in a stateful service it is not necessary to send details about the file operation each time a request is sent. Additionally, such a service can be understood as a centralized and coupled service. In contrast, a stateless service is decentralized and decoupled that delegates service tasks to clients. The great decentralized participation of clients in stateless service allows to better providing fault tolerance mechanisms once it is possible to avoid single points of failures.

18.5.5 File Replication

In distributed systems, replication techniques can be employed with different goals. Targeting consistency, for example, replication is useful for performance issues, e.g., accessing data at the same time or accessing the data copy whose network communication has a low latency. Replication can also be used for availability and fault tolerance. moreover, replication means file replication in distributed file system on text. Furthermore, replication mechanisms should keep replicas consistent even if they are used to provide fault tolerance. It implies a fault tolerance trade-off between consistency and performance. The techniques used in replication should choose one of these characteristics to prioritize.

- **Consistency vs. Performance**. When a client performs a file operation the file changes have to be updated to all its replicas. Consistency protocols manage them, providing atomicity to replica updates. In other words, these protocols ensure that all replicas of a file will correspond to the last change done in it. To achieve this, consistency protocols must avoid that clients open outdated replicas. This task is not trivial for distributed file systems that deal with mutable data (files). By that very fact, such protocols increase the system overhead which can degrade system performance. Consequently, the harder the consistency protocol is, the lower the performance that the system will acquire.
- **File Replication Location**. Another interesting aspect of file replication is the place where it will be replicated. Basically, it is possible to explore two approaches. The first one relies on replicating them on the same machine - be replicas on the same storage device or on different ones. This approach can profit a RAID scheme if relied on different medias (hard disks). In opposition, the second approach concerns replicas in different machines likewise using the network infrastructure to manage them.

It is important to remember that despite the approach chosen, clients should not care about file replication, i.e., it should be transparent for them. However, details about file replication (e.g., number of replicas) can be exposed to clients if the distributed file system allows them to tune it by themselves. Yet, there is the possibility to provide a hybrid approach that relies on file replication on different machines and also taking advantage of different Medias (in some servers or in all of them). This scenario may increase the system throughput however, if well-tuned, it can offer an environment to build robust fault tolerance services without degrading the system performance. Moreover, fault tolerance consistency protocols could improve their performance when relying on this hybrid approach. Fault tolerant mechanisms would profit

from different machines to avoid single points of failure and the consistency protocol could be better performed relying on faster accesses to local storage devices.

18.6 Design Principle: Andrew File System (AFS)

AFS was developed in the late 80s at CMU. It uses the following design principles:

1. Callbacks: The server records that has the copy of a file.
2. Write-back cache on file close: If a file is modified, the update is propagated to server when the file is closed. The server then immediately tells all clients who own an old copy.
3. Files are cached on each client's disk. NFS caches only in clients' memory.
4. Session semantics: Updates are only visible on close.

In UNIX (single machine), updates are visible immediately to other processes that have the file open. In AFS, everyone who has the file open sees the old version; anyone who opens the file again will see the new version.

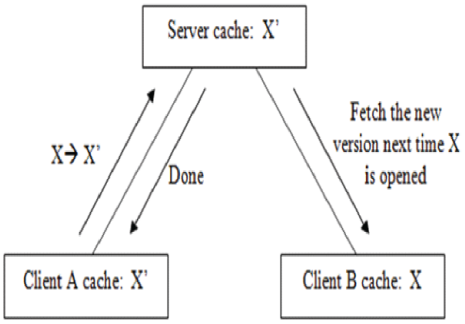


Figure 18.6: AFS Process

When a client opens a file and the file is not on the local disk, the client gets the file from the server and adds itself on the callback list. When a client closes a file, the client sends the updated copy back to the server and tells all clients to get the new version on the next open.

If the server crashes, the server loses all the callback states and needs to ask all clients to reconstruct the callback states.

18.7 Case Study

18.7.1 DCE Distributed File Service.

Distributed Computing Environment

DCE

A vendor-independent distributed computing environment, DCE was defined by the Open

Software Foundation (OSF), a consortium of computer manufacturers, including IBM, DEC, and Hewlett- Packard. It is not an operating system, nor is it an application. Rather, it is an integrated set of services and tools that can be installed as a coherent environment on top of existing operating systems and serve as a platform for building and running distributed applications.

A primary goal of DCE is vendor independence. It runs on many different kinds of computers, operating systems, and networks produced by different vendors. For example, some operating systems to which DCE can be easily ported include OSF/1, AIX, DOMAIN OS, ULTRIX, HP-UX, SINIX, SunOS, UNIX System V, VMS, WINDOWS, and OS/2. On the other hand, it can be used with any network hardware and transport software, including TCP/IP, X.25, as well as other similar products. As shown in Figure 10.7, DCE is middleware software layered between the DCE applications layer and the operating system and networking layer. The basic idea is to take a collection of existing machines (possibly from different vendors), interconnect them by a communication network, add the DCE software platform on top of the native operating systems of the machines, and then be able to build and run distributed applications. Each machine has its own local operating system, which may be different from that of other machines. The DCE software layer on top of the operating system and networking layer hides the differences between machines by automatically performing data-type conversions when necessary. Therefore, the heterogeneous nature of the system is transparent to the applications programmers, making their job of writing distributed applications much simpler.

DCE Application
DCE Software
Operating System & Networking

Table 18.7(a): Position of DCE software in a DCE-based distributed system

DCE Creation?

The OSF did not create DCE from scratch. Instead, it created DCE by taking advantage of work already done at universities and industries in the area of distributed computing. For this, OSF issued a request for technology (RFT), asking for tools and services needed to build a coherent distributed computing environment. To be a contender, a primary requirement was that actual working code must ultimately be provided. The submitted bids were carefully evaluated by OSF employees and a team of outside experts. Finally, those tools and services were selected that the members of the evaluation committee believed provided the best solutions. The code

comprising the selected tools and services, almost entirely written in C, was then further developed by OSF to produce a single integrated package that was made available to the world as DCE. Version 1.0 of DCE was released by OSF in January 1992.

DCE Components

As mentioned above, DCE is a blend of various technologies developed independently and nicely integrated by OSF. Each of these technologies forms a component of DCE. The main components of DCE are as follows:

1. **Threads Package:** It provides a simple programming model for building concurrent applications. It includes operations to create and control multiple threads of execution in a single process and to synchronize access to global data within an application.
2. **Remote Procedure Call (RPC) Facility:** It provides programmers with a number of powerful tools necessary to build client-server applications. In fact, the DCE RPC facility is the basis for all communication in DCE because the programming model underlying all of DCE is the client-server model. It is easy to use, is network- and protocol-independent, provides secure communication between a client and a server, and hides differences in data requirements by automatically converting data to the appropriate forms needed by clients and servers.
3. **Distributed Time Service (DTS):** It closely synchronizes the clocks of all the computers in the system. It also permits the use of time values from external time sources, such as those of the U.S. National Institute for Standards and Technology (NIST), to synchronize the clocks of the computers in the system with external time. This facility can also be used to synchronize the clocks of the computers of one distributed environment with the clocks of the computers of another distributed environment.
4. **Name Services:** The name services of DCE include the Cell Directory Service (CDS), the Global Directory Service (GDS), and the Global Directory Agent (GDA). These services allow resources such as servers, files, devices, and so on, to be uniquely named and accessed in a location-transparent manner.
5. **Security Service:** It provides the tools needed for authentication and authorization to protect system resources against illegitimate access.
6. **Distributed File Service (DFS):** It provides a system wide file system that has such characteristics as location transparency, high performance, and high availability. A unique feature of DCE DFS is that it can also provide file services to clients of other file systems.

The DCE components listed above are tightly integrated. It is difficult to give a pictorial representation of their interdependencies because they are recursive. For example, the name services use RPC facility for internal communication among its various servers, but the RPC facility uses the name services to locate the destination. Therefore, the interdependencies of the various DCE components can be best depicted in tabular form, as shown in Figure.

Component name	Other Component used by it
Threads	None
RPC	Threads, name, security
DTS	Threads, RPC, name, security
Name	Threads, RPC, DTS, security
Security	Threads, RPC, DTS, name
DFS	Threads, RPC, DTS, name, security

Table 18.7(b): Interdependencies of DCE components

DCE Cells

The DCE system is highly scalable in the sense that a system running DCE can have thousands of computers and millions of users spread over a worldwide geographic area. To accommodate such large systems, DCE uses the concept of cells. This concept helps break down a large system into smaller, manageable units called cells. In a DCE system, a cell is a group of users, machines, or other resources that typically have a common purpose and share common DCE services. The minimum cell configuration requires a cell directory server, a security server, a distributed time server, and one or more client machines. Each DCE client machine has client processes for security service, cell directory service, distributed time service, RPC facility, and threads facility. ADCE client machine may also have a process for distributed file service if a cell configuration has a DCE distributed file server. Due to the use of the method of intersection for clock synchronization, it is recommended that each cell in a DCE system should have at least three distributed time servers. An important decision to be made while setting up a DCE system is to decide the cell boundaries. The following four factors should be taken into consideration for making this decision.

1. **Purpose:** The machines of users working on a common goal should be put in the same cell, as they need easy access to a common set of system resources. That is, users of machines in the same cell have closer interaction with each other than with users of machines in different cells. For example, if a company manufactures and sells various types of products, depending on the manner in which the company functions, either a

product-oriented or a function-oriented approach may be taken to decide cell boundaries [Tanenbaum 1995]. In the product-oriented approach, separate cells are formed for each product, with the users of the machines belonging to the same cell being responsible for all types of activities (design, manufacturing, marketing, and support services) related to one particular product. On the other hand, in the function-oriented approach, separate cells are formed for each type of activity, with the users belonging to the same cell being responsible for a particular activity, such as design, of all types of products.

2. **Administration:** Each system needs an administrator to register new users in the system and to decide their access rights to the system's resources. To perform his or her job properly, an administrator must know the users and the resources of the system. Therefore, to simplify administration jobs, all the machines and their users that are known to and manageable by an administrator should be put in a single cell. For example, all machines belonging to the same department of a company or a university can belong to a single cell. From an administration point of view, each cell has a different administrator.
3. **Security:** Machines of those users who have greater trust in each other should be put in the same cell. That is, users of machines of a cell trust each other more than they trust the users of machines of other cells. In such a design, cell boundaries act like firewalls in the sense that accessing a resource that belongs to another cell requires more sophisticated authentication than accessing a resource that belongs to a user's own cell.
4. **Overhead:** Several DCE operations, such as name resolution and user authentication, incur more overhead when they are performed between cells than when they are performed within the same cell. Therefore, machines of users who frequently interact with each other and the resources frequently accessed by them should be placed in the same cell The need to access a resource of another cell should arise infrequently for better overall system performance Notice from the above discussion that in determining cell boundaries the emphasis is on purpose, administration, security, and performance. Geographical considerations can, but do not have to, play a part in cell design. For better performance, it is desirable to have as few cells as possible to minimize the number of operations that need to cross cell boundaries. However, subject to security and administration constraints, it is desirable to have smaller cells with fewer machines and users. Therefore, it is important to properly balance the requirements imposed by the four factors mentioned above while deciding cell boundaries in a DCE

system.

18.7.2 Sun Network File System (NFS)

Sun's NFS is one of the most popular and widespread distributed file systems in use today. The design goals of NFS were:

- Any machine can be a client and/or a server.
- NFS must support diskless workstations (that are booted from the network). Diskless workstations were Sun's major product line.
- Heterogeneous systems should be supported: clients and servers may have different hardware and/ or operating systems. Interfaces for NFS were published to encourage the widespread adoption of NFS.
- High performance: try to make remote access as comparable to local access through caching and read-ahead.

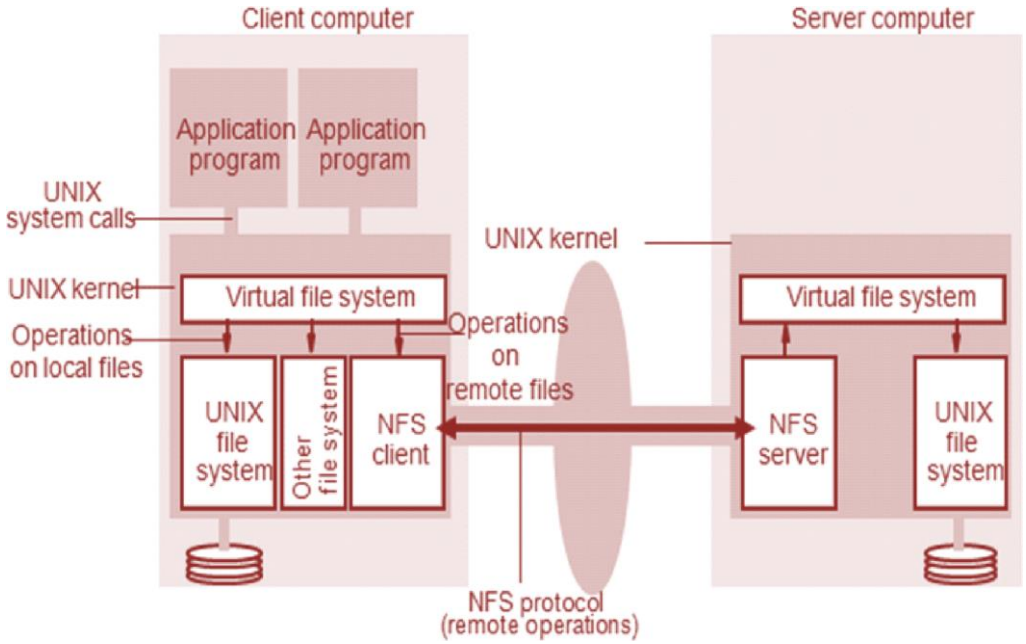


Figure 18.7(c): NFS Architecture

From a transparency point of view NFS offers:

Access Transparency

Remote (NFS) files are accessed through normal system calls; the protocol is implemented under the VFS (vnode) layer in UNIX.

Location Transparency

The client adds remote file systems to its local name space via mount. File systems must be exported at the server. The user is unaware of which directories are local and which are remote.

The location of the mount point in the local system is up to the client's administrator.

Failure Transparency

NFS is stateless; UDP is used as a transport. If a server fails, the client retries.

Performance Transparency

Caching at the client will be used to improve performance

No migration Transparency

The client mounts machines from a server. If the resource moves to another server, the client must know about the move.

No support for Unix Semantics

NFS is stateless, so stateful operations such as file locking are a problem. All UNIX file system controls may not be available.

Devices

Since NFS had to support diskless workstations, where every file is remote, remote device files had to refer to the client's local devices. Otherwise there would be no way to access local devices in a diskless environment.

NFS Protocols

The NFS client and server communicate over remote procedure calls (Sun's RPC) using two protocols: the mounting protocol and the directory and file access protocol. The mounting protocol is used to request a access to an exported directory (and the files and directories within that file system under that directory). The directory and file access protocol is used for accessing the files and directories (e.g. read/write bytes, create files, etc.). The use of RPC's external data representation (XDR) allows NFS to communicate with heterogeneous machines. The initial design of NFS ran only with remote procedure calls over UDP. This was done for two reasons. The first reason is that UDP is somewhat faster than TCP but does not provide error correction (the UDP header provides a checksum of the data and headers). The second reason is that UDP does not require a connection to be present. This means that the server does not need to keep per- client connection state and there is no need to re-establish a connection if a server was rebooted.

The lack of UDP error correction is remedied in the fact that remote procedure calls have

built-in retry logic. The client can specify the maximum number of retries (default is 5) and a timeout period. If a valid response is not received within the timeout period the request is re-sent. To avoid server overload, the timeout period is then doubled. The retry continues until the limit has been reached. This same logic keeps NFS clients fault-tolerant in the presence of server failures: a client will keep retrying until the server responds.

Mounting Protocol

The client sends the pathname to the server and requests permission to access the contents of that directory. If the name is valid and exported (listed in /etc/dfs/sharetab on System V release 4 versions of UNIX, and /etc/exports on many other versions) the server returns a file handle to the client. This file handle contains all the information needed to identify the file on the server: {file system type, disk ID, inode number, and security info}. Mounting an NFS file system is accomplished by parsing the path name, contacting the remote machine for a file handle, and creating an in-core vnode at the mount point. A vnode points to an inode for a local UNIX file or, in the case of NFS, an rnode. The rnode contains specific information about the state of the file from the point of view of the client.

Directory and File Access Protocol

Clients send RPC messages to the server to manipulate files and directories. A file is accessed by performing a lookup remote procedure call. This returns a file handle and attributes. It is not like an open in that no information is stored in any system tables on the server. After that, the handle may be passed as a parameter for other functions. For example, a read (handle, offset, count) function will read count bytes from location offset in the file referred to by handle. The entire directory and file access protocol is encapsulated in sixteen functions.

These are:

null	no-operation but ensure that connectivity exists
lookup	lookup the file name in a directory
create	create a file or a symbolic link
remove	remove a file from a directory
rename	rename a file or directory
read	read bytes from a file
write	write bytes to a file
link	create a link to a file
symlink	create a symbolic link to a file
read link	read the data in a symbolic link (do not follow the link)

mkdir	create a directory
rmdir	remove a directory reader read from a directory
getattr	get attributes about a file or directory (type, access and modify times, and access permissions)
setattr	set file attributes
statfs	get information about the remote file system

Accessing Files

Files are accessed through conventional system calls (thus providing access transparency). If you recall conventional UNIX systems; a hierarchical pathname is dereference to the file location with a kernel function called name. This function maintains a reference to a current directory looks at one component and finds it in the directory, changes the reference to that directory, and continues until the entire path is resolved. At each point in traversing this pathname, it checks to see whether the component is a mount point, meaning that name resolution should continue on another file system. In the case of NFS, it continues with remote procedure calls to the server hosting that file system.

Upon realizing that the rest of the pathname is remote, name will continue to parse one component of the pathname at a time to ensure that references to and to symbolic links become local if necessary. Each component is retrieved via a remote procedure call which performs an NFS lookup. This procedure returns a file handle. An in-core rnode is created and the VFS layer in the file system creates a vnode to point to it.

The application can now issue read and write system calls. The file descriptor in the user's process will reference the in-core vnode at the VFS layer, which in turn will reference the in core rnode at the NFS level which contains NFS-specific information, such as the file handle. At the NFS level, NFS read, write, etc. operations may now be performed, passing the file handle and local state (such as file offset) as parameters. No information is maintained on the server between requests; it is a stateless system. The RPC requests have the user ID and group ID number sent with them. This is a security hole that may be stopped by turning on RPC encryption.

Problems

The biggest problem with NFS is file consistency. The caching and validation policies do not guarantee session semantics. NFS assumes that clocks between machines are synchronized and performs no clock synchronization between client and server. One place where this hurts is in distributed software development environments. A program such as make, which compares times of files (such as object and source) to determine whether to regenerate them, can either

fail or give confusing results.

Because of its stateless design, open with append mode cannot be guaranteed to work. You can open a file, get the attributes (size), and then write at that offset, but you'll have no assurance that somebody else did not write to that location after you received the attributes. In that case your write will overwrite the other once since it will go to the old end-of-file byte offset. Also because of its stateless nature, file locking cannot work. File locking implies that the server keeps track of which processes have locks on the file. Sun's solution to this was to provide a separate process (a lock manager) that does keep state.

One common programming practice under UNIX file systems for manipulating temporary data in files is to open a temporary file and then remove it from the directory. The name is gone, but the data persists because you still have the file open. Under NFS, the server maintains no state about remotely opened files and removing a file will cause the file to disappear. Since legacy applications depended on this, Sun's solution was to create a special hack for UNIX: if the same process that has a file open attempts to delete it, it is instead moved to a temporary name and deleted on close. It's not a perfect solution, but it works well. Permission bits might change on the server and disallow future access to a file. Since NFS is stateless, it has to check access permissions each time it receives an NFS request. With local file systems, once access is granted initially, a process can continue accessing the file even if permissions change. By default, no data is encrypted, and Unix-style authentication is used (used ID, group ID). NFS supports two additional forms of authentication: Diffie-Hellman and Kerberos. However, data is never encrypted, and user-level software should be used to encrypt files if this is necessary.

18.7.3 OSF

Distributed file systems are an important component of an overall plan for distributed. Two such plans are currently being promulgated, one by OSF and the other by UI. A high-level view of these plans is shown in Figure.

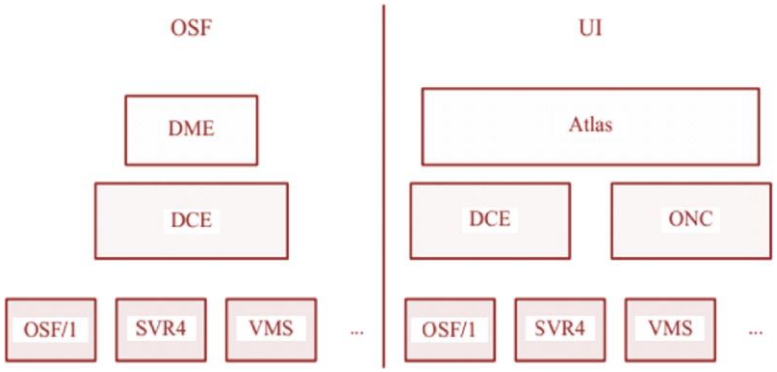


Figure 18.7(d): Competing Distributed Architecture Plans

UNIX International (UI), with the aid of (a portion of) the computer industry, has devised an overall framework for an industry-standard distributing computing architecture. The Open Software Foundation (OSF), in the meantime, has been developing an actual distributed computing architecture, known as DCE (for distributed computing environment). Fortunately, as it turns out, OSF's DCE fits within the UI Atlas view of the world. However, Sun's ONC (Open Network Computing) also fits within this scheme. The intent of both groups is that, whatever distributed architecture is adopted, it will support (or be supportable by) most existing operating systems, not just UNIX. DCE is currently well into development. Two initial, "functionality" versions of it have been released, and a production-quality release is expected within a year. As a separate project, OSF is working on DME (distributed management environment), whose concern is the management of services within a distributed environment.

18.8 Summary

In this unit, we discussed the various distributed file system. A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. A Distributed File System is a network file system where a single file system can be distributed across several physical computer nodes. Fault tolerance, scalability and transparency etc are the features of good distributed file system. Each file has a list of users associated with it and access permissions per user. Multiple users may be organized into an entity known as a group. These are managed by the Access Control List. In distributed file systems, not only local failures (e.g., due storage devices) should be dealt with, but also other failures inherent to the distributed environment. Stateless and stateful services are generally considered in it. At last, we compared various file system like AFS, NFS etc. in case study.

Self - Assessment Exercise

1. Briefly explain the features of Good DFS.
2. What do you mean by File System? Explain different File System issues.
3. Explain DCE.
4. Explain the NFS and its protocols.
5. Explain OSF with suitable diagram.