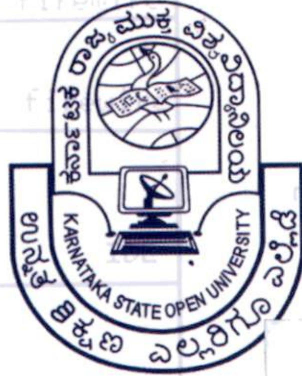
KARNATAKA STATE OPEN UNIVERSITY

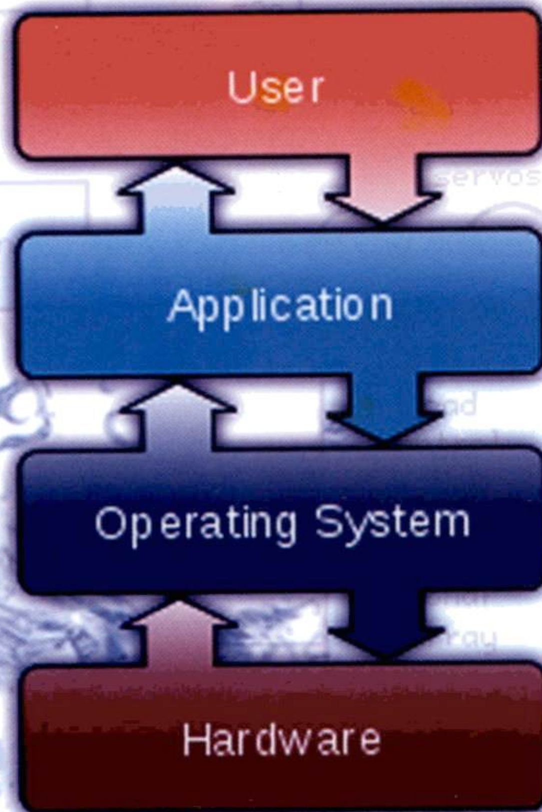
MANASAGANGOTRI, MYSORE- 570 006

DEPERTAMENT OF STUDIES IN INFORMATION TECHNOLOGY



M.SC IN INFORMATION SCIENCE

I SEMESTER



OPERATING SYSTEM

IS 1.3

KSOU NATIONAL INTERNATIONAL RECOGNITION

Karnataka State Open University (KSOU) was established on 1st June 1996 with the assent of H.E. Governor of Karnataka as a full fledged University in the Academic year 1996 vide Government notification No./EDI/UOV/dated 12th February 1996 (Karnataka State Open University Act – 1992). The Act was promulgated with the object to incorporate an Open University at the State Level for the introduction and promotion of Open University and Distance Education Systems in the education pattern of the State and the Country for the Co-ordination and determination of standard of such systems.

- With the virtue of KSOU Act of 1992, Karnataka State Open University is empowered to establish, maintain or recognize Institutions, Colleges, Regional Centres and Study Centres at such places in Karnataka and also open outside Karnataka at such places as it deems fit.
- All Academic Programmes offered by Karnataka State Open University are recognized by the Distance Education Council (DEC), Ministry of Human Resource Development (MHRD), New Delhi.
- Karnataka State Open University is a regular member of the Association of Indian Universities (AIU), New Delhi, since 1999.
- Karnataka State Open University is a permanent member of Association of Commonwealth Universities (ACU), London, United Kingdom since 1999. Its member code number: ZKASOPENUINI.
- Karnataka State Open University is a permanent member of Asian Association of Open Universities (AAOU), Beijing, CHINA, since 1999.
- Karnataka State Open University has association with Commonwealth of Learning (COL), Vancouver, CANADA, since 2003. COL is an intergovernmental organization created by commonwealth Heads of Government to encourage the development and sharing of open learning distance education knowledge, resources and technologies.

Higher Education To Everyone Everywhere

Module	Unit No.	Page No.
1	Unit - 1	5 – 18
	Unit - 2	19 – 32
	Unit - 3	33 – 45
	Unit - 4	46 – 59
2	Unit - 5	60 – 74
	Unit - 6	75 – 102
	Unit - 7	103 – 120
	Unit - 8	121 – 134
3	Unit - 9	135 – 148
	Unit - 10	149 – 160
	Unit - 11	161 – 169
	Unit - 12	170 – 181
4	Unit - 13	182 – 201
	Unit - 14	202 – 218
	Unit - 15	219 – 228
	Unit - 16	229 - 241

Course Design and Editorial Committee

Prof. K. S. Rangappa Vice Chancellor & Chairperson Karnataka State Open University Manasagangotri, Mysore – 570 006	**Prof. Vikram Raj Urs** Dean (Academic) & Convener Karnataka State Open University Manasagangotri, Mysore – 570 006
Head of the Department – Incharge	**Course Co-Ordinator**
Prof. Kamalesh DIRECTOR IT&Technology Karnataka State Open University Manasagangotri, Mysore – 570 006	**Mr. Mahesha DM** Lecturer, DOS in Information Technology Karnataka State Open University Manasagangotri, Mysore – 570 006

Course Writers

Dr. G. Hemanth Kumar Professor DOS in Computer Science University of Mysore Manasagangotri, Mysore – 570 006	**Modules 1 and 3**	**Units 1-4 and 9-12**
And		
Mr. Chandrajit Assistant Professor MIT college of Enginering Mysore – 570 006	**Modules 2 and 4**	**Units 5-8 and 13-16**

Publisher

Registrar
Karnataka State Open University
Manasagangotri, Mysore – 570 006

Developed by Academic Section, KSOU, Mysore
Karnataka State Open University, 2012

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Karnataka State Open University.

Further information on the Karnataka State Open University Programmes may be obtained from the University's Office at Manasagangotri, Mysore – 6.

Printed and Published on behalf of Karnataka State Open University, Mysore-6 by the **Registrar (Administration)**

MSIT – 103 Operating system

Preface

Operating systems bridge the gap between the hardware of a computer system and the user. Consequently, they are strongly influenced by hardware technology and architecture, both of which have advanced at a breathtaking pace since the first computers emerged in the 1940s. Many changes have been quantitative: the speed of processors, memories, and devices has been increasing continuously, whereas their size, cost, and power consumption has been decreasing. But many qualitative changes also have occurred. For example, personal computers with sophisticated input, output, and storage devices are now omnipresent; most also are connected to local area networks or the Internet. These advances have dramatically reshaped the world within which operating systems must exist and cooperate. Instead of managing a single processor controlling a collection of local memories and I/O devices, contemporary operating systems are required to manage highly parallel, distributed, and increasingly more heterogeneous configurations.

This book is an introduction to operating systems, appropriate for computer science or computer engineering majors at the junior or senior level. One objective is to respond to a major paradigm shift from single-processor to distributed and parallel computer systems, especially in a world where it is no longer possible to draw a clear line between operating systems for centralized environments and those for distributed ones. Although most of the book is devoted to traditional topics, we extend and integrate these with basic ideas in distributed computing.

After the introductory chapter, the book is organized into four main sections: Process Management and Coordination, Memory Management, File and I/O Management, andProtection and Security. At the end of each chapter, there is a summary of unit, exercise questions and key words

UNIT-1:

Structure

1.0 Objectives

1.1 Introduction

1.2 Operating System

1.3 I/O System Management

1.4 Evolution of Operating Systems

1.5 Mainframe Systems

1.6 Desktop Systems

1.7 Microprocessor System

1.8 Distributed Systems

1.9 Clustered Systems

1.10 Real Time Embedded Systems

1.11 Handheld Systems

1.12 Unit Summary

1.13 Keywords

1.14 Exercise

1.15 Reference

1.Objectives

After going through this unit, you will be able to: Describe Basic Organization of Computer Systems

- Describe Basic Organization of Computer Systems
- Define Operating system, functions, history and Evolution
- Define assembler, linker, loader, compiler

1.1Introduction

An operating system act as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system

1.2OperatingSystem

Definition of Operating System:

An Operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.

A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being applications programs.

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

Functions of Operating System

Operating system performs three functions:

- Convenience: An OS makes a computer more convenient to use.
- Efficiency: An OS allows the computer system resources to be used in an efficient manner.
- Ability to Evolve: An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without at the same time interfering with service.

Operating System as User Interface

Every general purpose computer consists of the hardware, operating system, system programs, and application programs. The hardware consists of memory, CPU, ALU, I/O devices, peripheral device and storage device. System program consists of compilers, loaders, editors, OS etc. The application program consists of business program, database program. The fig. 1.1 shows the conceptual view of a computer system

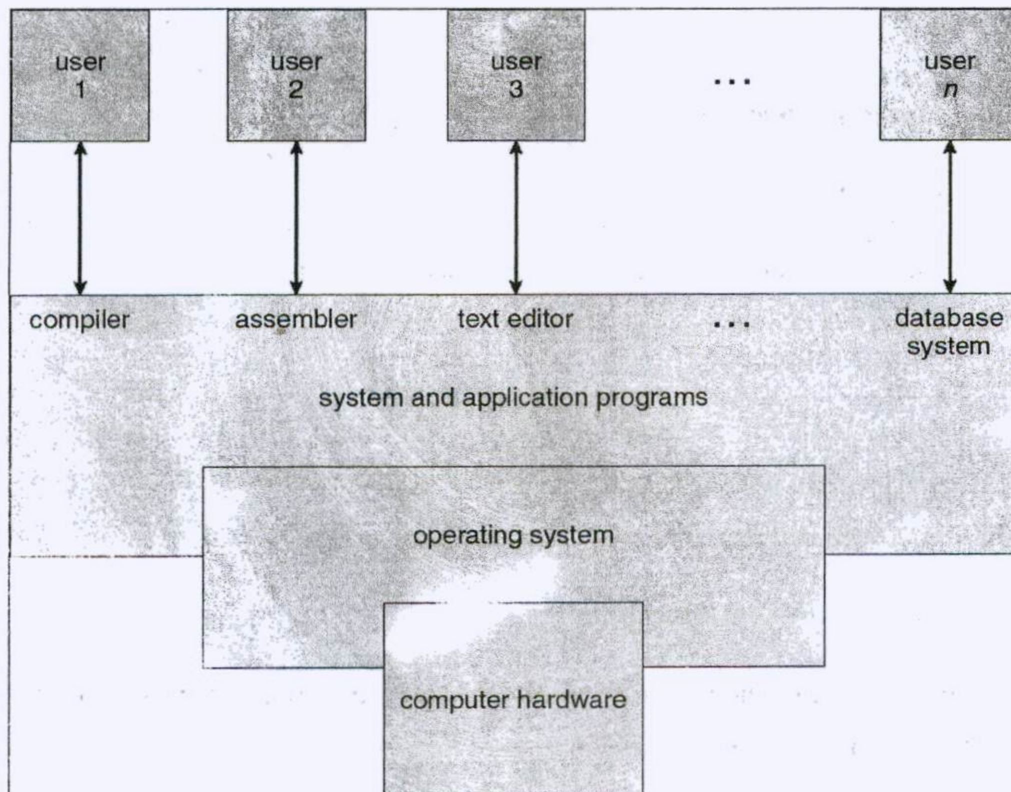


Fig1.1 Conceptualviewof a computer system

Every computer must have an operating system to run other programs. The operating system and coordinates the use of the hardware among the various system programs and application program for a various users. It simply provides an environment within which other programs can do useful work.

The operating system is a set of special programs that run on a computer system that allow it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling peripheral devices.

OS is designed to serve two basic purposes:

a. It controls the allocation and use of the computing system's resources among the various user and tasks.

b. It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.

The operating system must support the following tasks. The tasks are:

1. Provides the facilities to create, modification of program and data files using and editor.
2. Access to the compiler for translating the user program from high level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

1.3 I/OSystemManagement

I/OSystemManagement

- ThemodulethatkeepstrackofthestatusofdevicesiscalledtheI/Otraffic controller.EachI/Odevicehas adevicehandlerthatresidesinaseparate processassociatedwiththatdevice.
- The I/O subsystemconsistsof

1. Amemorymanagementcomponentthatincludesbuffering,cachingand spooling.
2. Ageneraldevice driver interface.

Driversfor specific hardwaredevices.

1.4 Evolution of Operating Systems

The Operating System Zoo: Mainframe OSs, Server OSs, Multiprocessor OSs, Personal computer OSs, Handheld OSs, Embedded OSs, Sensor node OSs, Real-time OSs, Smart card Oss.

Generations:

First generation (1945 - 1955), vacuum tubes, plug boards

Second generation (1955 - 1965), transistors, batch systems

Third generation (1965 - 1980), ICs and multiprogramming

Fourth generation (1980 - present), personal computers

Next generation??, personal digital assistants (PDA), information appliances

1.5 Mainframe Systems

1. The earliest computers, developed in the 1940s, were programmed in machine language and they used front panel switches for input.
2. The programmer was also the operator interacting with the computer directly from the system console. First commercial systems: Enormous, expensive and slow. I/O: Punch cards and line printers.

3. The term probably had originated from the early mainframes, as they were housed in enormous, room-sized metal boxes or frames (See Figure). Later the term was used to distinguish high-end commercial machines from less powerful units.

4. Single operator/programmer/user runs and debugs interactively:
   - ❖ Standard library with no resource coordination. Monitor that is always resident
- initial control in monitor,
- control transfers to job,
- When job completes control transfers back to monitor.

5. Poor throughput (like amount of useful work done per hour) and poor utilization (keeping all devices busy). Inefficient use of hardware.

1.6 Desktop Systems

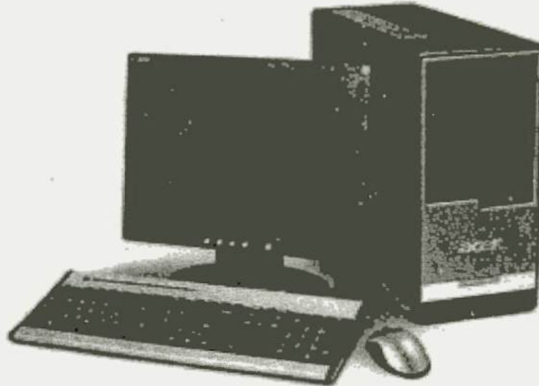
1. Single-user, dedicated. Previously thought as individuals have sole use of computer, do not need advanced CPU utilization, protection features (see Fig. 1.3).

2. Not still true. May run several different types of OS (Windows, Mac OS X, UNIX, and Linux) which offer multitasking and virtual memory on PC hardware.

3. Most systems use a single processor. On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all systems have other special-purpose processors as well.

4. They may come in the form of device-specific processors, such as disk, keyboard, and

graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.



5. All of these special-purpose processors run a limited instruction set and do not run user processes. Sometimes they are managed by the OS, in that the OS sends them information about their next task and monitors their status.

6. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling.

7. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU.

8. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

1.7 Microprocessor System

Intel developed and delivered the first commercially viable microprocessor way back in the early 1970's: the 4004 and 4040 devices. The 4004 was not very powerful and all it could do was add

and subtract with 4-bit data only at a time. But it was amazing those days that everything was on one chip. Prior to the 4004, engineers built computers either from collections of chips or from discrete components (Transistor wired one at a time). The machines then, were not portable and were very bulky, and power hungry. The 4004 changed the scene with all its circuitry on a single chip. The 4004 powered one of the first portable electronic calculators named 'Busicom'.These 4-bit microprocessors, intended for use in calculators, required very little power Nevertheless, they demonstrated the future potential of the microprocessor - an entire CPU on a single piece of silicon. Intel rapidly followed their 4-bit offerings with their 8008 and 8080 eight-bit CPUs.

A small outfit in Santa Fe, New Mexico, incorporated the 8080 CPU into a box they called the Altair 8800. Although this was not the world's first "personal computer" (there were some limited distribution machines built around the 8008 prior to this), the Altair was the device that sparked the imaginations of hobbyists of the world and the personal computer revolution was born.

The trends in processor design had impact of historical development of microprocessors from different manufacturers. Intel started facing competition from Motorola, MOS Technology, and an upstart company formed by disgruntled Intel employees, Zilog. To compete, Intel produced the 8085 microprocessor. To the software engineer, the 8085 was essentially the same as the 8080. However, the 8085 had lots of hardware improvements that made it easier to design into a circuit. Unfortunately, from software perspective the other manufacturer's offerings were better. Motorola's 6800 series was easier to program, MOS Technologies' 65xx family was also easier to program but very inexpensive, and Zilog's Z80 chip was upward compatible with the 8080 with lots of additional instructions and other features. By 1978 most personal computers were using the 6502 or Z80 chips, not the Intel offerings

The first microprocessor to make a real splash in the market was the Intel 8088, introduced in 1979 and incorporated into the IBM PC (which appeared around 1982 for the first time). If we are familiar with the PC market and its history, we know that the PC market moved from the 8088 to the 80286 to the 80386 to the 80486 to the Pentium to the Pentium II to the Pentium III to the Pentium 4. Intel makes all of these microprocessors and all of them are improvements of design base of the 8088. The Pentium 4 can execute any piece of code that ran on the original 8088, but it does it about 5,000 times faster!

Sometime between 1976 and 1978 Intel decided that they needed to leap-frog the competition

and produced a 16-bit microprocessor that offered substantially more power than their competitor's eight-bit offerings. This initiative led to the design of the 8086 microprocessor. The 8086 microprocessor was not the world's first 16-bit microprocessor (there were some oddball 16-bit microprocessors prior to this point) but it was certainly the highest performance single-chip 16-bit microprocessor when it was first introduced.

A microprocessor is the chip containing some control and logic circuits that is capable of making arithmetic and logical decisions based on input data and produce the corresponding arithmetic or logical output. The word 'processor' is the derivative of the word 'process' that means to carry out systematic operations on data. The computer we are using to write this page of the manuscript uses a microprocessor to do its work. The microprocessor is the heart of any computer, whether it is a desktop machine, a server or a laptop. The microprocessor we are using might be a Pentium, a K6, a PowerPC, a Spark or any of the many other brands and types of microprocessors, but they all do approximately the same thing in approximately the same way. No logically enabled device can do anything without it. The microprocessor not only forms the very basis of computers, but also many other devices such as cell phones, satellites, and many other hand held devices. They are also present in modern day cars in the form of microcontrollers.

A microprocessor is also known as a CPU or central processing unit, which is a complete computational engine that is fabricated on a single chip. Here we will discuss the history of the 80x86 CPU family and the major improvements occurring along the line. The historical background will help us to better understand the design compromises they made as well as to understand the legacy issues surrounding the CPU's design. We are discussing the major advances in computer architecture that Intel employed while improving the x86.

1.8 Distributed Systems

In the late 1970s Xerox PARC was already using Alto computers on Ethernets as servers providing printing and ¯le services (Lampson 1988). In the 1980s universities also developed experimental systems for distributed personal computing. It is difficult to evaluate the significance of this recent work:

- By 1980 the major concepts of operating systems had already been discovered.

- Many distributed systems were built on top of the old time-sharing system Unix, which was designed for central rather than distributedcomputing (Pike 1995).

- In most distributed systems, process communication was based on a complicated, unreliable programming technique, known as remote procedure calls.

- Only a handful of distributed systems, including Locus (Popek 1981) and the Apollo Domain (Leach 1983), were developed into commercial products.

- There seems to be no consensus in the literature about the fundamentalcontributions and relative merits of these systems.

Under these circumstances, the best I could do was to select a handful of readable papers that I hope are representative of early and more recent distributed systems.

1.9 Clustered Systems

Server clustering is essential in today's business environment, in order to provide the high availability/scalability of services required to support 24x7x365 production operations. This high availability/scalability requirement encompasses network operating systems, application services and LAN/WAN network resilience. Today's computing architectures must be designed within an infrastructure combining High Availability, Scalability, Manageability, Flexibility and Cost Effectiveness.

Evolution provides technical consultancy resources to effectively implement Novell or Microsoft Cluster Services, Citrix Server Farms or Check Point server load balancing to provide a high availability computing architecture within your organization.

What is a cluster? A cluster is two or more interconnected servers [nodes] that create a solution to provide higher availability, higher scalability or both. Clustering servers for high availability is seen if one node fails, another node in the cluster assumes the workload of the failed node, and users see no interruption of access. Clustering servers for high scalability includes increased application performance and support for a greater number of concurrent users. Clustering can be

implemented at different levels of the system, including operating systems, middleware and application level. The more layers that incorporate clustering technology, the more reliable, scalable and manageable the cluster.

Cluster technology requires the necessary server hardware [nodes], shared storage devices, interconnects and cluster services software/systems management and cluster resources [applications and services].

1.10 Real Time Embedded Systems

- Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens. They tend to have very specific tasks.

- The systems they run on are usually primitive, and so the OSs provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

- Embedded systems almost always run real-time OSs. A real-time system is used when rigid time requirements (time critical) have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application.

- A real-time system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. Real-Time systems may be either;
    - hard (must react in time): the real-time system absolutely must complete critical tasks within a guaranteed time. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.
    - soft real-time (deal with failure to react in time): the real-time system can satisfy its performance criteria by running any critical task at a higher priority (of CPU access).Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

1.11 Handheld Systems

- Handheld systems include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones, many of which use special-purpose embedded OSs.
- Hand-held systems must also deal with limited resources although their screens have recently become more substantial. Because of their size, most handheld devices have a small amount of memory, slow processors, and small display screens.
- Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability.

1.12 Summary

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

The evolution of operating systems went through seven major phases (Table). Six of them significantly changed the ways in which users accessed computers through the open shop, batch processing, multiprogramming, timesharing, personal computing, and distributed systems. In the seventh phase the foundations of concurrent programming were developed and demonstrated in model operating systems.

Major Phases	Operating Systems
Open Shop	IBM 701 open shop (1954)
Multiprogramming	Atlas supervisor (1961) B5000 system (1964) Exec II system (1966) Egdon system (1966)

Timesharing	CTSS (1962) Multics file system (1965) Titan file system (1972) Unix (1974)
Concurrent Programming	THE system (1968) RC 4000 system (1969) Venus system (1972) Boss 2 system (1975) Solo system (1976) Solo program text (1976)
Personal Computing	OS 6 (1972) Alto system (1979) Pilot system (1980) Star user interface (1982)
Distributed Systems	WFS ¯le server (1979) Unix United RPC (1982) Unix United system (1982) Amoeba system (1990)

1.13 Keywords

Operating System, Timesharing, Distributed, Multiprogramming

1.14 Exercises

1. Define Operating System?
2. Explain various function of operating system?
3. Explain I/O system Management?
4. Discuss evolution of different types of operating systems

1.5 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-2:

Structure

2.0 Objectives

2.1 Introduction

2.2 Feature Migration

2.3 Computing Environment

2.4 System Components

2.5 Operating Systems Services

2.6 System Calls and System Programs

2.7 System structure

2.8 Virtual Machine

2.9 Unit Summary

2.10 Keywords

2.11 Exercise

2.12 Reference

2.0Objectives

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

- Describe Basic Structure Systems
- Express the importance of Feature migration
- Define System call, System structure and virtual machines

2.1Introduction

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

2.2Feature Migration

One reason to study early architectures and operating systems is that a feature that once ran only on huge systems may eventually have made its way into very small systems. Indeed, an examination of operating systems for mainframes and microcomputers shows that many features once available only on mainframes have been adopted for microcomputers. The same operating-system concepts are thus appropriate for various classes of computers: mainframes, minicomputers, microcomputers, and handhelds. To understand modern operating systems, then, you need to recognize the theme of feature migration and the long history of many operating-system features.

A good example of feature migration started with the Multiplexed Information and Computing Services (MULTICS) operating system. MULTICS was developed from 1965 to 1970 at the Massachusetts Institute of Technology () as a computing utility. It ran on a large, complex mainframe computer (the GE 645). Many of the ideas that were developed for MULTICS were

subsequently used at Bell Laboratories (one of the original partners in the development of MULTICS) in the design of UNIX. The UNIX operating system was designed around 1970 for a PDP-11 minicomputer. Around 1980, the features of UNIX became the basis for UNIX-like operating systems on microcomputers; and these features are included in several more recent operating systems for microcomputers, such as Microsoft Windows, Windows XP, and the Mac OS X operating system. Linux includes some of these same features, and they can now be found on PDAs.

2.3Computing Environment

Computing Environment is a collection of computers / machines, software, and networks that support the processing and exchange of electronic information meant to support various types of computing solutions.

High Performance Computing

The High Performance Computing environment consists of high-end systems used for executing complex number crunching applications for research it has two such machines and they are,

- HP Cluster.
- VEGA Supercluster.

The machines in the above list belong to cluster computing category.

HP Cluster

The HP Cluster is a 120 Processor machine with 15 Nodes. Each node has 8 processors and named as LEO. One Head Node having DL 360 HP Proliant and other 14 Clients With DL 140 HP Proliant.

VEGA Supercluster

The HP Proliant Rack server DL160 based Cluster options are built with, eight HP 10642 G2 Racks hosting 2048 core Compute Server infrastructure, One separate rack for Infiniband Switch option, One separate rack for NAS and Tape Library and three separate racks HP SFS parallel file system solution.

2.4 System Components

Even though, not all systems have the same structure many modern operating systems share the same goal of supporting the following types of system components.

Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, registers, OS resources in use etc.).

It is important to note that a process is not a program. A process is only ONE instant of a program in execution. There are many processes can be running the same program. The five major activities of an operating system in regard to process management are

- Creation and deletion of user and system processes.
- Suspension and resumption of processes.
- A mechanism for process synchronization.
- A mechanism for process communication.
- A mechanism for deadlock handling.

Main-Memory Management

Primary-Memory or Main-Memory is a large array of words or bytes. Each word or byte has its own address. Main-memory provides storage that can be access directly by the CPU. That is to say for a program to be executed, it must in the main memory.

The major activities of an operating in regard to memory-management are:

- Keep track of which part of memory are currently being used and by whom.
- Decide which processes are loaded into memory when memory space becomes available.

- Allocate and deallocate memory space as needed.

File Management

A file is a collected of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, and data transfer rate and access methods.

File systems normally organized into directories to ease their use. These directories may contain files and other directions.

The five main major activities of an operating system in regard to file management are

- The creation and deletion of files.
- The creation and deletion of directions.
- The support of primitives for manipulating files and directions.
- The mapping of files onto secondary storage.
- The backup of files on stable storage media.

I/O System Management

I/O subsystem hides the peculiarities of specific hardware devices from the user. Only the device driver knows the peculiarities of the specific device to whom it is assigned.

Secondary-Storage Management

Generally speaking, systems have several levels of storage, including primary storage, secondary storage and cache storage. Instructions and data must be placed in primary storage or cache to be referenced by a running program. Because main memory is too small to accommodate all data and programs, and its data are lost when power is lost, the computer system must provide secondary storage to back up main memory. Secondary storage consists of tapes, disks, and other media designed to hold information that will eventually be accessed in primary storage (primary, secondary, cache) is ordinarily divided into bytes or words consisting of a fixed number of bytes. Each location in storage has an address; the set of all addresses available to a program is called

an address space.

The three major activities of an operating system in regard to secondary storage management are:

- Managing the free space available on the secondary-storage device.
- Allocation of storage space when new files have to be written.
- Scheduling the requests for memory access.

Networking

A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock. The processors communicate with one another through communication lines called network. The communication-network design must consider routing and connection strategies, and the problems of contention and security.

Protection System

If computer systems has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by computer systems.

Command Interpreter System

A command interpreter is an interface of the operating system with the user. The user gives commands with are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Command-Interpreter is usually not part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be support by an operating system, and they do not really need to run in kernel mode. There are two main advantages to separating the command interpreter from the kernel.

2.5 Operating Systems Services

Following are the five services provided by an operating systems to the convenience of the users.

Program Execution

The purpose of a computer system is to allow the user to execute programs. So the operating systems provide 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 deallocating memory, CPU scheduling in case of multiprocess. 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.

I/O Operations

Each program requires an input and produces output. This involves the use of I/O. The operating systems hides 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 systems by providing I/O make it convenient for the users to run programs.

For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating systems provide 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 systems make it easier for user programs to accomplished 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 I think 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.

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 the 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 to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

Error Detection

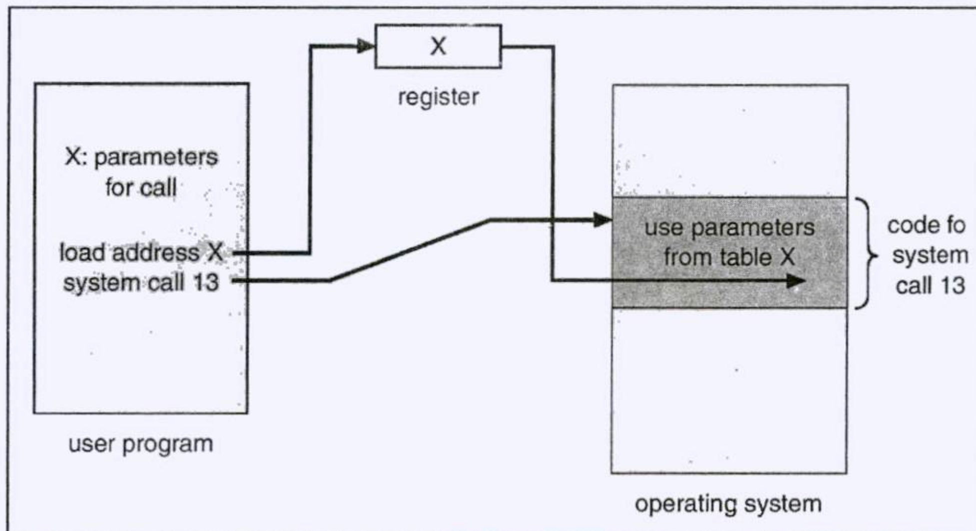
An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

This service cannot allow to be handled by user programs because it involves monitoring and in cases altering area of memory or deallocation of memory for a faulty process. Or may be 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.

2.6 System Calls and System Programs

System calls provide an interface between the process an 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.



2.7 System Structure

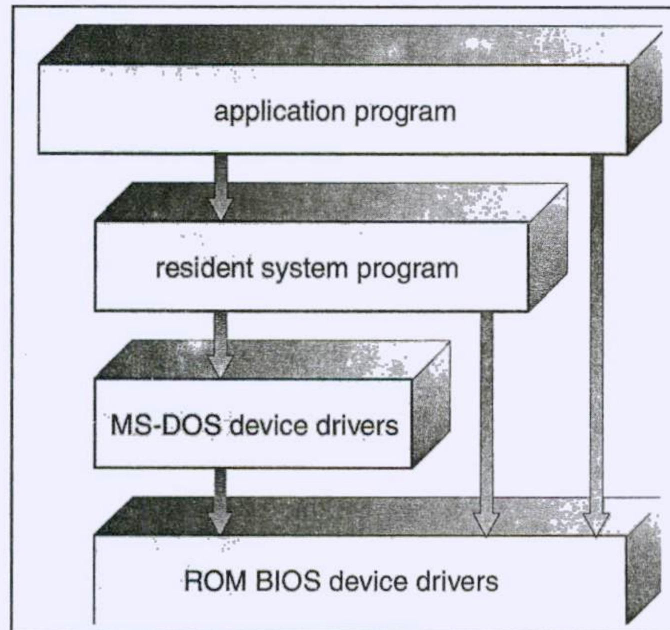
- Modern OS should be developed carefully due to their size and complexity.
- A common approach is to divide the systems into small components.
- We can view an OS from several points.
    - One view focuses on the services that the system provides;
    - Another, on the interface that it makes available to users and programmers;
    - A third, on its components and their interconnections.
- We consider what services an OS provides, how they are provided, and what the various methodologies are for designing such systems.

MS-DOS System Structure

MS-DOS – written to provide the most functionality in the least space

Not divided into modules

Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

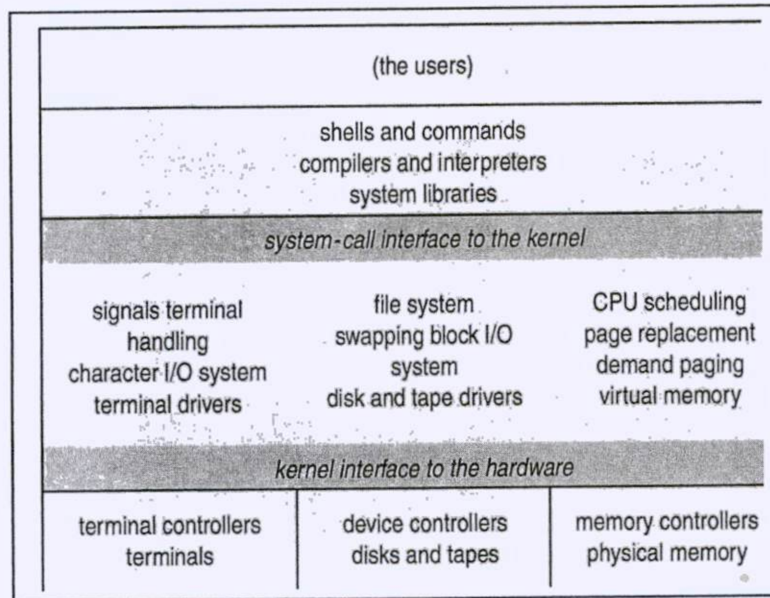


Unix System Structure

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.
  - Systems programs – use kernel supported system calls to provide useful functions such as compilation and file manipulation.
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.



- An OS layer is an implementation of an abstract object that is the encapsulation of data and operations that can manipulate those data. These operations (routines) can be invoked by higher-level layers. The layer itself can invoke operations on lower-level layers.
- Layered approach provides modularity. With modularity, layers are selected such that each layer uses functions (operations) and services of only lower-level layers.
- Each layer is implemented by using only those operations that are provided lower level layers.
- The major difficulty is appropriate definition of various layers.

Microkernel System Structure

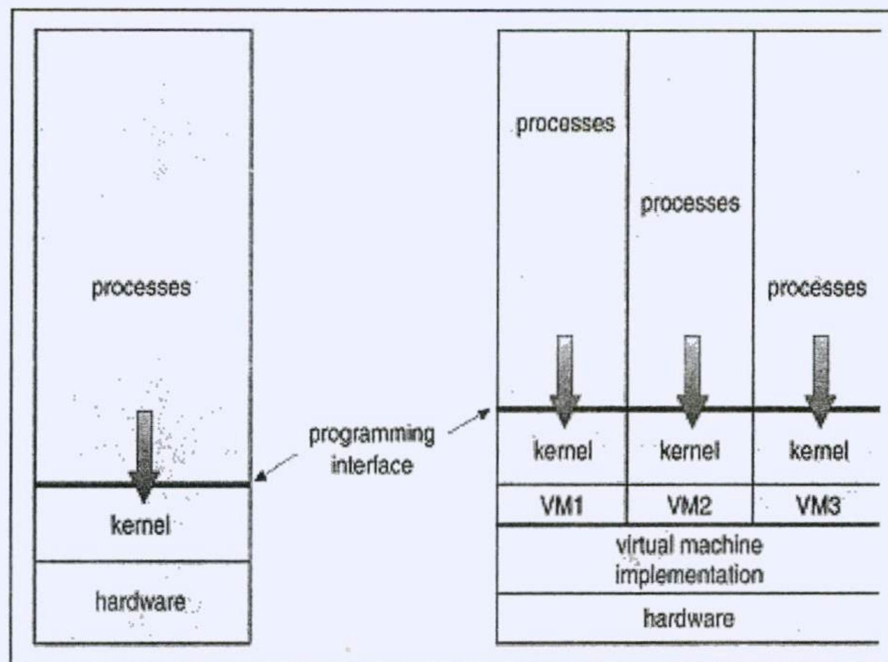
- Moves as much from the kernel into "*user*" space.
- Communication takes place between user modules using message passing.
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure

2.8 Virtual Machine

- The layered approach described in Section above is taken to its logical conclusion in the concept of a virtual machine. The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

- By using CPU scheduling and virtual-memory techniques, an OS can create the illusion that a process has its own processor with its own (virtual) memory. Normally, a process has additional features, such as system calls and a file system that are not provided by the bare hardware.

- The virtual-machine approach does not provide any such additional functionality but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (see Figure).



- There are several reasons for creating a virtual machine, all of which are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different OSs) concurrently.

- Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming back into fashion as a means of solving system compatibility problems.

Advantages/Disadvantages of Virtual Machines

The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources. A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation. The virtual machine concept is difficult to implement due to the effort required to provide an exact duplicate to the underlying machine.

2.9 Summary

Operating systems provide a number of services. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mechanism for a user to issue a request without writing a program. Commands may come from files during batch-mode execution or directly from a terminal when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.

The types of requests vary according to level. The system-call level must provide the basic functions, such as process control and file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control, status requests, and I/O requests. Program errors can be considered implicit requests for service.

2.10 Keywords

Virtual machine, Kernel, File Management, System call.

2.11 Exercises

1. Explain various system components
2. Explain applications of virtual machines
3. What are the five major activities of an operating system with regard to file management?
4. What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?
5. Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

2.12 Reference

1. Operating System Concepts, 8th Edition by Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, John Wiley and Sons, 2008.
2. Modern Operating Systems, 3rd Edition by Andrew S. Tanenbaum, Prentice Hall, 2008.
3. Beginning Linux Programming, 3rd Edition by Neil Matthew and Richard Stones, Wiley Publishing, 2004.

UNIT-3:

Structure

3.0 Objectives

3.1 Introduction

3.2 Memory Management

3.3 Contiguous Allocation

3.4 Partitioning

3.5 Paging

3.6 Segmentation

3.7 Segmentation and Paging

3.8 Demand Paging

3.9 Unit Summary

3.10 Keywords

3.11 Exercise

3.12 Reference

3.0Objectives

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

- Define memory management.
- Explain allocation of contiguous memory.
- To explain the segmentation and paging.
- To describe the demand paging.

3.1Introduction

Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory. Modern computer systems use disks as the primary on-line storage medium for information (both programs and data). The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. A file is a collection of related information defined by its creator. The files are mapped by the operating system onto physical devices. Files are normally f organized into directories for ease of use.

The devices that attach to a computer vary in many aspects. Some devices transfer a character or a block of characters at a time. Some can be accessed only sequentially, others randomly. Some transfer data synchronously, others asynchronously. Some are dedicated, some shared. They can be read-only or read-write. They vary greatly in speed. In many ways, they are also the slowest major component of the computer.

Because of all this device variation, the operating system needs to provide a wide range of functionality to applications, to allow them to control all aspects of the devices. One key goal of an operating system's I/O subsystem is to provide the simplest interface possible to the rest of the system. Because devices are a performance bottleneck, another key is to optimize I/O for maximum concurrency.

3.2Memory Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory (at least partially) during execution.

To improve both the utilization of the CPU and the speed of its response to users, the computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the hardware design of the system. Each algorithm requires its own hardware support.

- In a very simple OS, only one program at a time is in the memory. To run second program, the first one has to be removed and the second one placed in memory. Single Tasking System.

- More sophisticated OSs allows multiple programs to be in memory at the same time. To keep them from interfering with one another (and with OS), some kind of protection mechanism is needed. Multi-Tasking System.

- Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has its own address.
- The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on Von Neumann architecture).

- For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

- The OS is responsible for the following activities in connection with memory management:

  - Keeping track of which parts of memory are currently being used and by whom,

- Deciding which processes (or parts thereof) and data to move into and out of memory.
- Allocating and de-allocating memory space as needed.

3.3Contiguous Allocation

Contiguous memory allocation is one of the efficient ways of allocating main memory to the processes. The memory is divided into two partitions. One for the Operating System and another for the user processes. Operating System is placed in low or high memory depending on the interrupt vector placed. In contiguous memory allocation each process is contained in a single contiguous section of memory.

Memory protection

Memory protection is required to protect Operating System from the user processes and user processes from one another. A relocation register contains the value of the smallest physical address for example say 100040. The limit register contains the range of logical address for example say 74600. Each logical address must be less than limit register. If a logical address is greater than the limit register, then there is an addressing error and it is trapped. The limit register hence offers memory protection.

The MMU, that is, Memory Management Unit maps the logical address dynamically, that is at run time, by adding the logical address to the value in relocation register. This added value is the physical memory address which is sent to the memory. The CPU scheduler selects a process for execution and a dispatcher loads the limit and relocation registers with correct values. The advantage of relocation register is that it provides an efficient way to allow the Operating System size to change dynamically.

Memory allocation

There are two methods namely, multiple partition method and a general fixed partition method. In multiple partition method, the memory is divided into several fixed size partitions. One process occupies each partition. This scheme is rarely used nowadays. Degree of multiprogramming depends on the number of partitions. Degree of multiprogramming is the number of programs that are in the main memory. The CPU is never left idle in multiprogramming. This was used by IBM OS/360 called MFT. MFT stands for

Multiprogramming with a fixed number of Tasks.

Generalization of fixed partition scheme is used in MVT. MVT stands for Multiprogramming with a Variable number of Tasks. The Operating System keeps track of which parts of memory are available and which is occupied. This is done with the help of a table that is maintained by the Operating System. Initially the whole of the available memory is treated as one large block of memory called a hole. The programs that enter a system are maintained in an input queue. From the hole, blocks of main memory are allocated to the programs in the input queue. If the hole is large, then it is split into two, and one half is allocated to the arriving process and the other half is returned. As and when memory is allocated, a set of holes in scattered. If holes are adjacent, they can be merged.

Now there comes a general dynamic storage allocation problem. The following are the solutions to the dynamic storage allocation problem.

- First fit: The first hole that is large enough is allocated. Searching for the holes starts from the beginning of the set of holes or from where the previous first fit search ended.

- Best fit: The smallest hole that is big enough to accommodate the incoming process is allocated. If the available holes are ordered, then the searching can be reduced.

- Worst fit: The largest of the available holes is allocated.

First and best fits decrease time and storage utilization. First fit is generally faster.

Fragmentation

The disadvantage of contiguous memory allocation is fragmentation. There are two types of fragmentation, namely, internal fragmentation and External fragmentation.

Internal fragmentation

When memory is free internally, that is inside a process but it cannot be used, we call that fragment as internal fragment. For example say a hole of size 18464 bytes is available. Let the size of the process be 18462. If the hole is allocated to this process, then two bytes are left which is not used. These two bytes which cannot be used forms the internal fragmentation. The worst part of it is that the overhead to maintain these two bytes is more than two bytes.

External fragmentation

All the three dynamic storage allocation methods discussed above suffer external fragmentation. When the total memory space that is got by adding the scattered holes is sufficient to satisfy a request but it is not available contiguously, then this type of fragmentation is called external fragmentation.

The solution to this kind of external fragmentation is compaction. Compaction is a method by which all free memory that are scattered are placed together in one large memory block. It is to be noted that compaction cannot be done if relocation is done at compile time or assembly time. It is possible only if dynamic relocation is done, that is relocation at execution time.

One more solution to external fragmentation is to have the logical address space and physical address space to be noncontiguous. Paging and Segmentation are popular noncontiguous allocation methods.

3.4Partitioning

In a partitioning operating system, memory (and possibly CPU time as well) is divided among statically allocated partitions in a fixed manner. The idea is to take a processor and make it pretend it is several processors by completely isolating the subsystems.

Hard partitions are set up for each part of the system and each has certain amount of memory (and potentially a time slice) allocated to it. Each partition is forever limited to its initial fixed memory allocation, which can neither be increased nor decreased after the initial system configuration.

Within each partition may be multiple threads or processes, or both, if the operating system supports them. How these threads are scheduled depends on the implementation of the OS. A partition will generally support a separate namespace to enable multi-programming by mapping the program into the partition.

If the operating system supports time partitioning, it too is fixed. For example, in an ARINC 653 partitioning system with just three partitions and a total major allocation of 100ms per cycle, a

fixed cyclic scheduler could be set to run the first partition for 20 ms, then the second partition for 30 ms, and then the third for 50 ms.

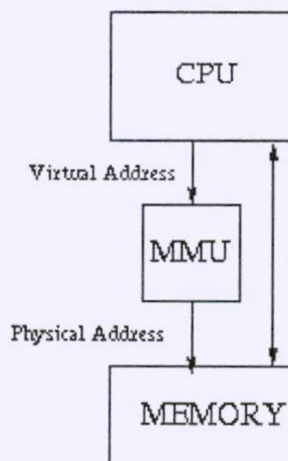
3.5 Paging

When a program is selected for execution, the system brings it into virtual storage, divides it into pages of four kilobytes, and transfers the pages into central storage for execution. To the programmer, the entire program appears to occupy contiguous space in storage at all times. Actually, not all pages of a program are necessarily in central storage, and the pages that are in central storage do not necessarily occupy contiguous space.

The pieces of a program executing in virtual storage must be moved between real and auxiliary storage. To allow this, z/OS® manages storage in units, or blocks, of four kilobytes. The following blocks are defined:

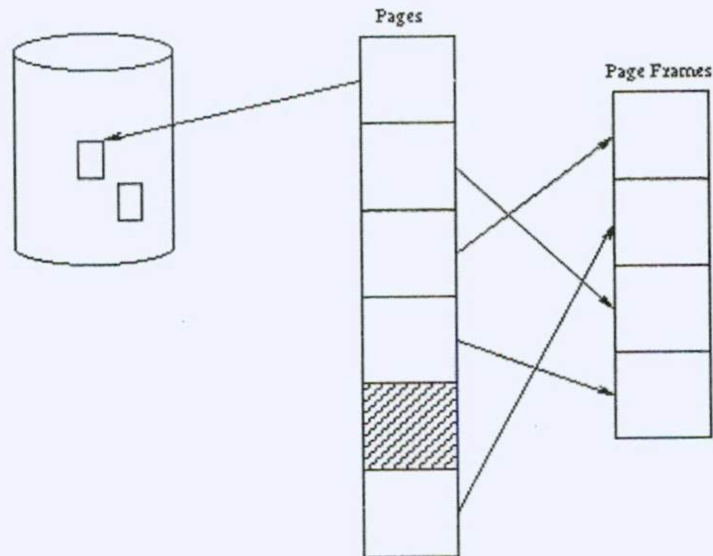
- A block of central storage is a **frame**.
- A block of virtual storage is a **page**.
- A block of auxiliary storage is a **slot**.

Most modern computers have special hardware called a memory management unit (MMU). This unit sits between the CPU and the memory unit. Whenever the CPU wants to access memory (whether it is to load an instruction or load or store data), it sends the desired memory address to the MMU, which translates it to another address before passing it on the memory unit. The address generated by the CPU, after any indexing or other addressing-mode arithmetic, is called a virtual address, and the address it gets translated to by the MMU is called a physical address.



Normally, the translation is done ; . Each page is a power of 2 bytes

long, usually between 1024 and 8192 bytes. If virtual address p is mapped to physical address f (where p is a multiple of the page size), then address p+o is mapped to physical address f+o for any offset o less than the page size. In other words, each page is mapped to a contiguous region of physical memory called a page frame.



The MMU allows a contiguous region of virtual memory to be mapped to page frames scattered around physical memory making life much easier for the OS when allocating memory. Much more importantly, however, it allows infrequently-used pages to be stored on disk. Here's how it works: The tables used by the MMU have a valid bit for each page in the virtual address space. If this bit is set, the translation of virtual addresses on a page proceeds as normal. If it is clear, any attempt by the CPU to access an address on the page generates an interrupt called a page fault trap. The OS has an interrupt handler for page faults, just as it has a handler for any other kind of interrupt. It is the job of this handler to get the requested page into memory.

In somewhat more detail, when a page fault is generated for page p1, the interrupt handler does the following:

- Find out where the contents of page p1 are stored on disk. The OS keeps this information in a table. It is possible that this page isn't anywhere at all, in which case the memory reference is simply a bug. In this case, the OS takes some corrective action such as killing the process that made the reference (this is source of the notorious message "memory fault -- core dumped"). Assuming the page is on disk:
- Find another page p2 mapped to some frame f of physical memory that is not used much.
- Copy the contents of frame f out to disk.

- Clear page p2's valid bit so that any subsequent references to page p2 will cause a page fault.
- Copy page p1's data from disk to frame f.
- Update the MMU's tables so that page p1 is mapped to frame f.
- Return from the interrupt, allowing the CPU to retry the instruction that caused the interrupt.

3.6 Segmentation

Memory segmentation is the division of computer's primary memory into segments or sections. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset within that segment. Segments or sections are also used in object files of compiled programs when they are linked together into a program image and when the image is loaded into memory.

Different segments may be created for different program modules, or for different classes of memory usage such as code and data segments. Certain segments may even be shared between programs

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory.

- The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory.

- This mapping allows differentiation between logical memory and physical memory.

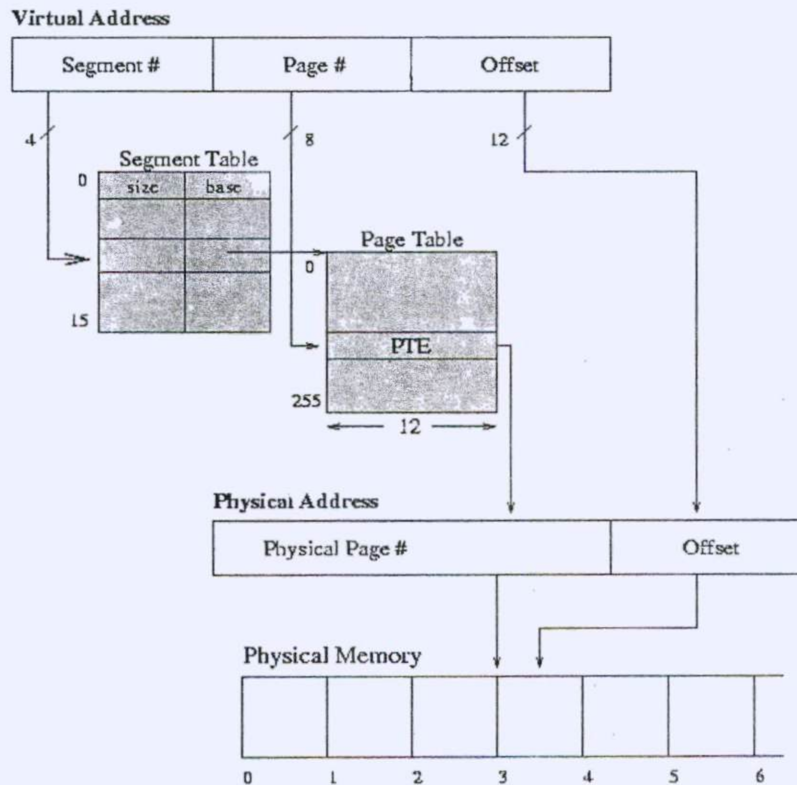
3.7 Segmentation and Paging

Use two levels of mapping, with logical sizes for objects, to make tables manageable.

- Each segment contains one or more pages.
- Segment corresponds to logical units: code, data, and stack. Segments vary in size and are often large. Pages are for the use of the OS; they are fixed size to make it easy to

manage memory.

- Going from paging to P+S is like going from single segment to multiple segments, except at a higher level. Instead of having a single page table, have many page tables with a base and bound for each. Call the stuff associated with each page table a segment.



System 370 example: 24-bit virtual address space, 4 bits of segment number, 8 bits of page number, and 12 bits of offset. Segment table contains real address of page table along with the length of the page table (a sort of bounds register for the segment). Page table entries are only 12 bits, real addresses are 24 bits.

- If a segment is not used, then there is no need to even have a page table for it.
- Can share at two levels: single page, or single segment (whole page table).

Pages eliminate external fragmentation, and make it possible for segments to grow without any reshuffling.

If page size is small compared to most segments, then internal fragmentation is not too bad.

The user is not given access to the paging tables.

If translation tables are kept in main memory, overheads could be very high: 1 or 2 overhead references for every real reference.

Another example: VAX.

- Address is 32 bits, top two select segments. Three base-bound pairs define page tables (system, P0, P1).
- Pages are 512 bytes long.
- Read-write protection information is contained in the page table entries. not in the segment table.
- One segment contains operating system stuff; two contain stuff of current user process.
- Potential problem: page tables can get big. Do not want to have to allocate them contiguously, especially for large user processes. Solution:
  - System base-bounds pairs are physical addresses, system tables must be contiguous.
  - User base-bounds pairs are virtual addresses in the system space. This allows the user page tables to be scattered in non-contiguous pages of physical memory.
  - The result is a two-level scheme.

In current systems, you will see three and even four-level schemes to handle 64-bit address spaces.

3.8 Demand Paging

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

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

3.9 Summary

This chapter has introduced the term operating system and its primary goals. They are

- Efficient use of computer resources
- To provide a good user-friendly interface

We have also discussed the Von-Neumann concept of a stored program and the need for secondary storage. Since different secondary storage devices differ with respect to speed, cost and permanence of storage, a hierarchy of storage exists and the design of a computer system makes the right balance of the above factors.

3.10 Keywords

Fragmentation, Memory protection, Memory Management, Paging.

3.11 Exercises

1. What are the five major activities of an operating system in regard to memory management?
2. Disscus the working of Paging
3. Explain segementation with paging
4. Explore the demand paging

3.12 Reference

1. Silberschatz and Galvin, Operating system concepts, Addison-Wesley publication, 5th edition.
2. Achyut S Godbole, Operating systems, Tata McGraw-Hill publication.
3. Milan Milankovic, Operating systems concepts and design, Tata McGraw-Hill publication, 2nd edition.

UNIT-4:

Structure

4.0 Objectives

4.1 Introduction

4.2 Page Replacement Algorithms

4.3 Allocation of Frames

4.4 Thrashing

4.5 Disk Structure

4.6 Disk Scheduling

4.7 Disk Management

4.8 Swap Space Management

4.9 Unit Summary

4.10 Keywords

4.11 Exercise

4.12 Reference

4.0Objectives

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

- Describe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices.
- Explain the performance characteristics of mass-storage devices.
- Discuss operating-system services provided for mass storage.

4.1Introduction

Almost all modern general purpose operating systems use virtual memory to solve the overlay problem. In virtual memory the combined size of program code, data and stack may exceed the amount of main memory available in the system. This is made possible by using secondary memory, in addition to main memory. The operating system tries to keep the part of the memory in active use in main memory and the rest in secondary memory. When memory located in secondary memory is needed, it can be retrieved back to main memory

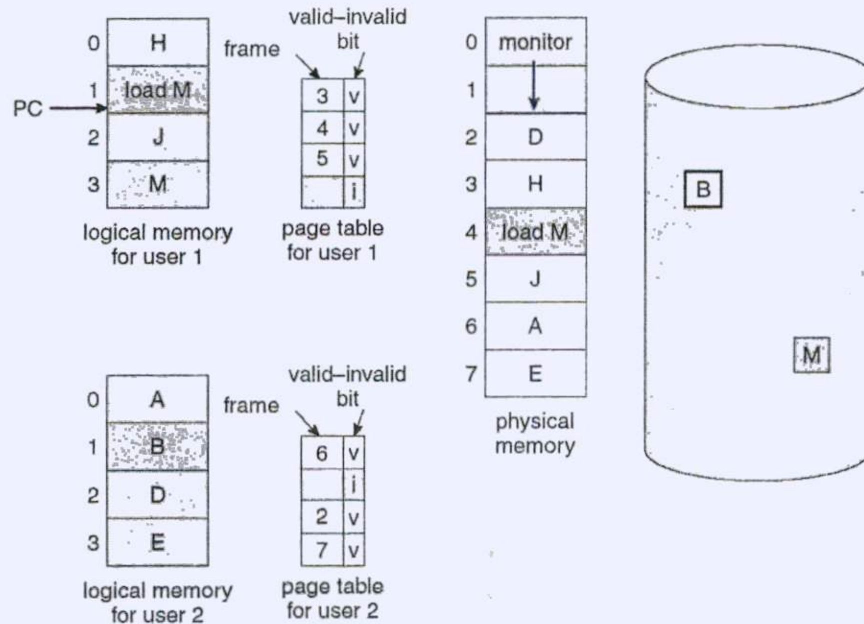
4.2Page Replacement Algorithms

Certain operating systems use paging to get virtual memory. This means that a part of the hard disk or a file is used so that the applications or the operating system see more memory that is actually there. A Page replacement algorithm is an algorithm that decides which pages should be written to disk or file, when a new page needs to be allocated.

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

- Page replacement takes the following approach (see Fig. 9.10).
    - If no frame is free, we find one that is not currently being used and free it.
    - We can free a frame by writing its contents to swap space and changing the page table (and all other tables) 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 faulted.



- We modify the page-fault service routine to include page replacement:

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 victim 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.

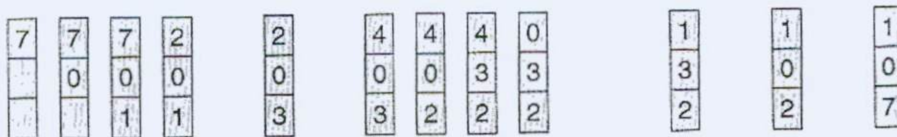
- Notice that, if no frames are free, two page transfers (one out and one in) are required.
- We can reduce this overhead by using a modify bit (or dirty bit).
- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
  - ❖ When we select a page for replacement, we examine its modify bit.
  - ❖ If the bit is set, we know that the page has been modified since it was read in from the disk (write that page to the disk).
- If the modify bit is not set, the page has not been modified since it was read into memory (not write the memory page to the disk: It is already there).
- This technique also applies to read-only pages (for example, pages of binary code).
- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.
- Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.
- We must solve two major problems to implement demand paging:
  - ❖ develop a frame-allocation algorithm. If we have multiple processes in memory, we must decide how many frames to allocate to each process.
  - ❖ develop a page-replacement algorithm. When page replacement is required, we must select the frames that are to be replaced.
- Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.
- Second major problem will be discussed firstly.
- For a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address.
- If we have a reference to a page , then any immediately following references to page will never cause a page fault (page will be in memory after the first reference).

FIFO Page Replacement

- If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string (similar situation with the SJF CPU-scheduling algorithm). As a result, the optimal algorithm is used mainly for comparison studies.

LRU Page Replacement

- The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that
  - ❖ The FIFO algorithm uses the time when a page was brought into memory,
  - ❖ Whereas the OPT algorithm uses the time when a page is to be used.
- If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time



- This approach is the least-recently-used (LRU) algorithm. The result of applying LRU replacement to our example reference string is shown in Fig. 9.14. The LRU algorithm produces 12 faults.
  - ❖ Notice that the first 5 faults are the same as those for optimal replacement.
  - ❖ When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
  - ❖ Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
  - ❖ When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

- Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

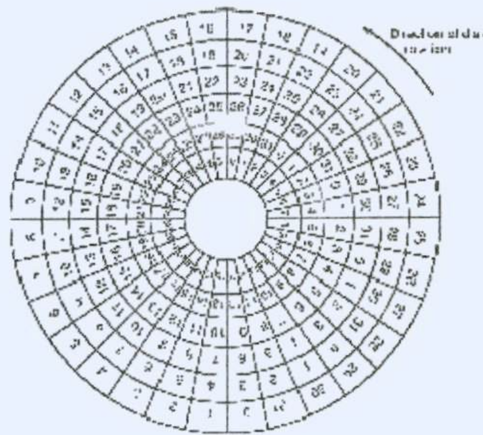
4.3Allocation of Frames

- Now, first major problem mentioned in above section will be discussed. How do we allocate the fixed amount of free memory among the various processes?
- If we have 93 free frames and two processes, how many frames does each process get?
- The simplest case is the single-user system.
    - Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames.
    - The OS may take 35 K8, leaving 93 frames for the user process.
- Under pure demand paging, all 93 frames would initially be put on the free-frame list.
    - When a user process started execution, it would generate a sequence of page faults.
    - The first 93 page faults would all get free frames from the free-frame list.
    - When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on.
    - When the process terminated, the 93 frames would once again be placed on the free-frame list.
- There are many variations on this simple strategy. We can require that the OS allocate all its buffer and table space from the free-frame list.
- When this space is not in use by the OS, it can be used to support user paging. The user process is allocated any free frame.

4.4Thrashing

Disk Formatting

- A new magnetic disk is a blank slate: It is just a platter of a magnetic recording material.
- Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called **low-level formatting**, or **physical formatting**.
- Low-level formatting fills the disk with a special data structure for each sector.
    - The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer.
    - The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code** (ECC).
- When the controller **writes** a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.
- When the sector is **read**, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.
- The controller automatically does the ECC processing whenever a sector is read or written.
- To use a disk to hold files, the OS still needs to record its own data structures on the disk. It does so in two steps.
    - The first step is to partition the disk into one or more groups of cylinders.
        - The OS can treat each partition as though it were a separate disk.
        - For instance, one partition can hold a copy of the OS's executable code, while another holds user files.
    - After partitioning, the second step is **logical formatting** (or creation of a file system).
        - In this step, the OS stores the initial file-system data structures onto the disk.
        - These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory.
- When reading sequential blocks, the seek time can result in missing block 0 in the next **track**. Disk can be formatted using a cylinder skew to avoid this



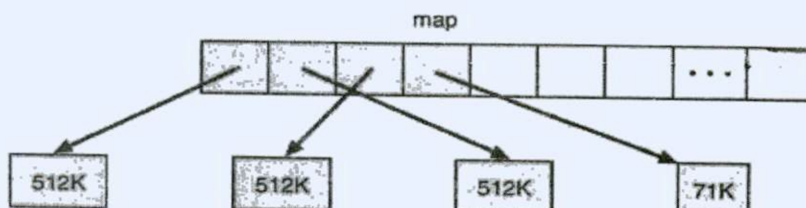
- To increase efficiency, most file systems group blocks together into larger chunks, frequently called clusters. Disk I/O is done via blocks, but file system I/O is done via clusters; effectively assuring that I/O has more sequential-access and fewer random-access characteristics.

4.8 Swap Space Management

Swapping is moving entire processes between disk and main memory. Swapping in that setting occurs when the amount of physical memory reaches a critically low point and processes (which are usually selected because they are the least active) are moved from memory to swap space to free available memory. In practice, very few modern operating systems implement swapping in this fashion. Rather, systems now combine swapping with virtual memory techniques and swap pages, not necessarily entire processes. In fact, some systems now use the terms swapping and paging interchangeably, reflecting the merging of these two concepts.

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

4.3 BSD Text-Segment Swap Map



4.9 Summary

Disk drives are the major secondary-storage I/O devices on most computers. Most secondary storage devices are either magnetic disks or magnetic tapes. Modern disk drives are structured as a large one-dimensional array of logical disk blocks which is usually 512 bytes.

Disks may be attached to a computer system in one of two ways: (1) using the local I/O ports on the host computer or (2) using a network connection such as storage area networks.

4.10 Keywords

Thrashing, Disk Structure, Disk Scheduling, Disk Management.

4.11 Exercises

1. Explain various Page Replacement Algorithms.
2. Discuss Allocation of Frames
3. Define thrashing?

4.12 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill

1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-5:

Structure

5.0 Objectives

5.1 Introduction

5.2 Process Concept

5.3 Process operations

5.4 Process Scheduling Queues

5.5 Inter-process Communication (IPC)

5.6 Unit Summary

5.7 Keywords

5.8 Exercise

5.9 Reference

5.0Objectives

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

- Describe Process Concepts and Creation
- Express the importance of Process Scheduling methods
- Define Interprocess Communication

5.1Introduction

A process can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently. Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

5.2Process Concept

Processes & Programs

- Process is a dynamic entity. A process is a sequence of instruction execution process exists in a limited span of time. Two or more process may execute the same program by using its own data & resources.
- A program is a static entity which is made up of program statement. Program contains the instruction. A program exists in a single space. A program does not execute by itself.

- A process generally consists of a process stack which consists of temporary data & data section which consists of global variables.
- It also contains program counter which represents the current activities.
- A process is more than the program code which is also called text section.

Process State

The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily. The process state consists of at least following:

- Code for the program.
- Program's static data.
- Program's dynamic data.
- Program's procedure call stack.
- Contents of general purpose register.
- Contents of program counter (PC)
- Contents of program status word (PSW).
- Operating Systems resource in use.

5.3Process operations

Process Creation

In general-purpose systems, some way is needed to create processes as needed during operation. There are four principal events led to processes creation.

- System initialization.
- Execution of a process Creation System calls by a running process.
- A user request to create a new process.
- Initialization of a batch job.

Foreground processes interact with users. Background processes that stay in background sleeping but suddenly springing to life to handle activity such as email, webpage, printing, and so on. Background processes are called daemons. This call creates an exact clone of the calling process.

A process may create a new process by some create process such as 'fork'. It choose to does so, creating process is called parent process and the created one is called the child processes. Only one parent is needed to create a child process. Note that unlike plants and animals that use sexual representation, a process has only one parent. This creation of process (processes) yields a hierarchical structure of processes like one in the figure. Notice that each child has only one parent but each parent may have many children. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings and the same open files. After a process is created, both the parent and child have their own distinct address space. If either process changes a word in its address space, the change is not visible to the other process.

Following are some reasons for creation of a process

- User logs on.
- User starts a program.
- Operating systems creates process to provide service, e.g., to manage printer.
- Some program starts another process, e.g., Netscape calls *xv* to display a picture.

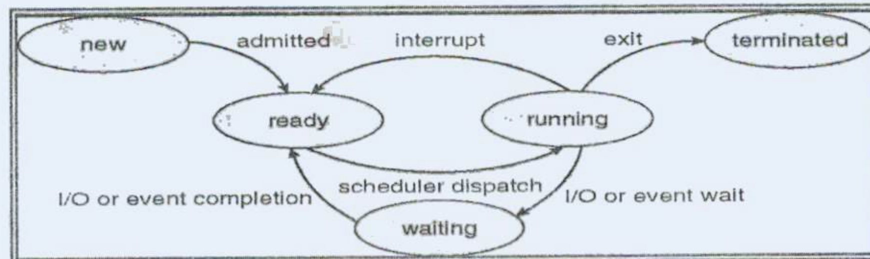
Process Termination

A process terminates when it finishes executing its last statement. Its resources are returned to the system, it is purged from any system lists or tables, and its process control block (PCB) is erased i.e., the PCB's memory space is returned to a free memory pool. The new process terminates the existing process, usually due to following reasons:

- **Normal Exist** Most processes terminates because they have done their job. This call is exist in UNIX.
- **Error Exist** When process discovers a fatal error. For example, a user tries to compile a program that does not exist.
- **Fatal Error** An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.
- **Killed by another Process** A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is kill. In

- some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

Process States :A process goes through a series of discrete process states.



- **New State** The process being created.
- **Terminated State** The process has finished execution.
- **Blocked (waiting) State** When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an I/O completion) before it can proceed. In this state a process is unable to run until some external event happens.
- **Running State** A process is said t be running if it currently has the CPU, that is, actually using the CPU at that particular instant.
- **Ready State** A process is said to be ready if it use a CPU if one were available. It is runable but temporarily stopped to let another process run.

Logically, the 'Running' and 'Ready' states are similar. In both cases the process is willing to run, only in the case of 'Ready' state, there is temporarily no CPU available for it. The 'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

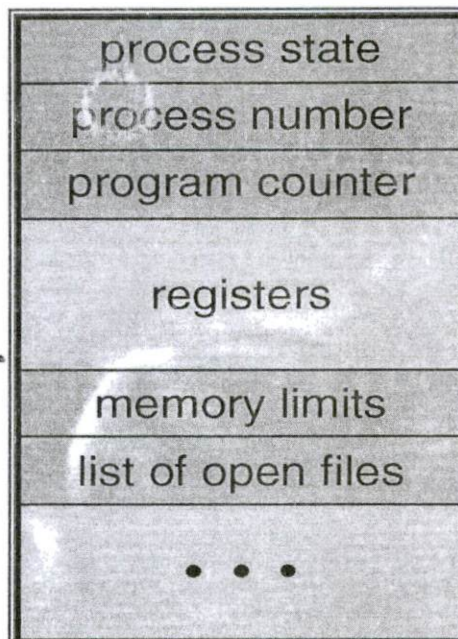
Process Control Block

A process in an operating system is represented by a data structure known as a process control block (PCB) or process descriptor. The PCB contains important information about the specific process including

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on.

The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

The following figure shows the process control block.



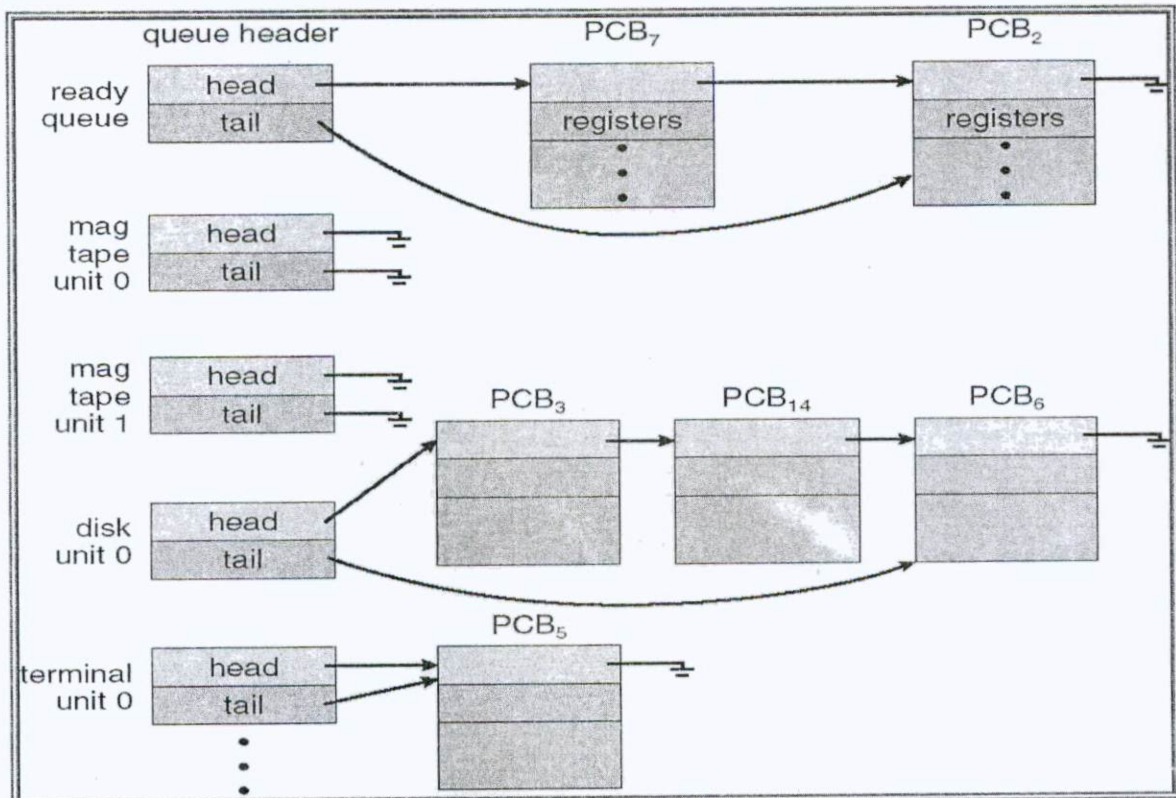
5.4 Process Scheduling Queues

The following are the different types of process scheduling queues.

1. Job queue – set of all processes in the system

2. Ready queue – set of all processes residing in main memory, ready and waiting to execute
3. Device queues – set of processes waiting for an I/O device
4. Processes migrate among the various queues

Ready Queue and Various I/O Device Queues



Ready Queue

The process that are placed in main m/y and are already and waiting to executes are placed in a list called the ready queue. This is in the form of linked list. Ready queue header contains pointer to the first & final PCB in the list. Each PCB contains a pointer field that points next PCB in ready queue.

Device Queue

The list of processes waiting for a particular I/O device is called device. When the CPU is allocated to a process it may execute for some time & may quit or interrupted or wait for the occurrence of a particular event like completion of an I/O request but the I/O may be busy with some other processes. In this case the process must wait for I/O. This will be placed in device queue. Each device will have its own queue.

The process scheduling is represented using a queuing diagram. Queues are represented by the rectangular box & resources they need are represented by circles. It contains two queues ready queue & device queues.

Once the process is assigned to CPU and is executing the following events can occur,

a. It can execute an I/O request and is placed in I/O queue.
b. The process can create a sub process & wait for its termination.
c. The process may be removed from the CPU as a result of interrupt and can be put back into ready queue.

Schedulers

The following are the different type of schedulers

1. **Long-term scheduler (or job scheduler)** – selects which processes should be brought into the ready queue.
2. **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU.
3. **Medium-term schedulers**

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
    - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
    - CPU-bound process – spends more time doing computations; few very long CPU bursts

Context Switch

1. When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
2. Context-switch time is overhead; the system does no useful work while switching.
3. Time dependent on hardware support

Cooperating Processes & Independent Processes

Independent process: one that is independent of the rest of the universe.

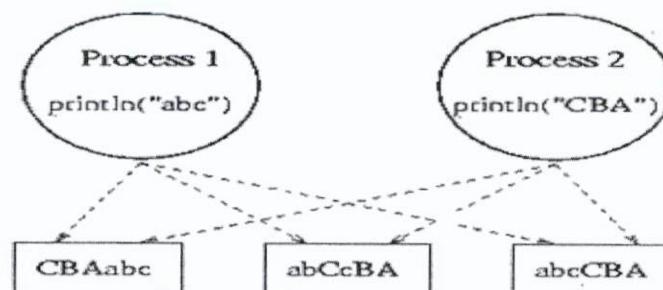
- Its state is not shared in any way by any other process.
- Deterministic: input state alone determines results.
- Reproducible.
- Can stop and restart with no bad effects (only time varies). Example: program that sums the integers from 1 to i (input).

There are many different ways in which a collection of independent processes might be executed on a processor:

- Uniprogramming: a single process is run to completion before anything else can be run on the processor.
- Multiprogramming: share one processor among several processes. If no shared state, then order of dispatching is irrelevant.
- Multiprocessing: if multiprogramming works, then it should also be ok to run processes in parallel on separate processors.
    - A given process runs on only one processor at a time.
    - A process may run on different processors at different times (move state, assume processors are identical).
    - Cannot distinguish multiprocessing from multiprogramming on a very fine grain.

Cooperating processes:

- Machine must model the social structures of the people that use it. People cooperate, so machine must support that cooperation. Cooperation means shared state, e.g. a single file system.
- Cooperating processes are those that share state. (May or may not actually be "cooperating")
- Behavior is nondeterministic: depends on relative execution sequence and cannot be predicted a priori.
- Behavior is irreproducible.
- Example: one process writes "ABC", another writes "CBA". Can get different outputs, cannot tell what comes from which. E.g. which process output first "C" in "ABCCBA"? Note the subtle state sharing that occurs here via the terminal. Not just anything can happen, though. For example, "AABBCC" cannot occur.



1. Independent process cannot affect or be affected by the execution of another process
2. Cooperating process can affect or be affected by the execution of another process
3. Advantages of process cooperation

    - Information sharing
    - Computation speed-up
    - Modularity
    - Convenience

5.5 Inter-process Communication (IPC)

1. Mechanism for processes to communicate and to synchronize their actions
2. Message system – processes communicate with each other without resorting to shared variables
3. IPC facility provides two operations:

    - send(*message*) – message size fixed or variable
    - receive(*message*)

4. If *P* and *Q* wish to communicate, they need to:

    - establish a *communicationlink* between them
    - exchange messages via send/receive

5. Implementation of communication link

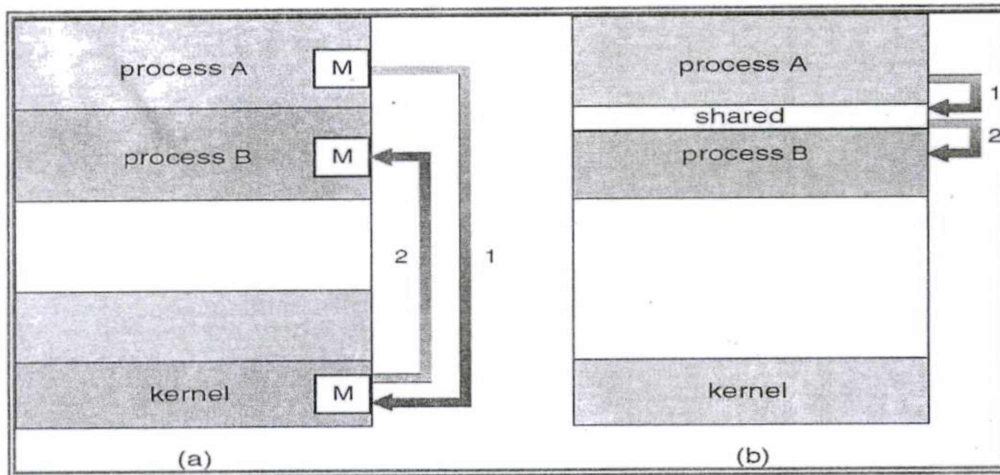
    - physical (e.g., shared memory, hardware bus)
    - logical (e.g., logical properties)

Communications Models

There are two types of communication models

1. Multi programming

2. Shared Memory



Direct Communication

1. Processes must name each other explicitly:

   - send (*P, message*) – send a message to process P
   - receive(*Q, message*) – receive a message from process Q

2. Properties of communication link

   - Links are established automatically
   - A link is associated with exactly one pair of communicating processes
   - Between each pair there exists exactly one link
   - The link may be unidirectional, but is usually bi-directional

Indirect Communication

1. Messages are directed and received from mailboxes (also referred to as ports)

   - Each mailbox has a unique id
   - Processes can communicate only if they share a mailbox

2. Properties of communication link

- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional

3. Operations

- o Create a new mailbox
- o Send and receive messages through mailbox
- o Destroy a mailbox

4. Primitives are defined as:

send(*A, message*) – send a message to mailbox A

receive(*A, message*) – receive a message from mailbox A

5. Mailbox sharing

P1, P2, and *P3* share mailbox A

P1, sends; *P2* and *P3* receive

Who gets the message?

6. Solutions

Allow a link to be associated with at most two processes

Allow only one process at a time to execute a receive operation

Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

1. Message passing may be either blocking or non-blocking
2. Blocking is considered synchronous

   Blocking send has the sender block until the message is received.

   Blocking receive has the receiver block until a message is available.

3. Non-blocking is considered asynchronous

   Non-blocking send has the sender send the message and continue.

   Non-blocking receive has the receiver receive a valid message or null.

Buffering

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   sender must wait for receiver (rendezvous)

2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full

3. Unbounded capacity – infinite length
   sender never waits

5.6 Summary

A process is a program in execution. As a process executes, it changes state. The state of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process-control block (PCB).

A process, when it is not executing, is placed in some waiting queue. There are two major classes of queues in an operating system: I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCBs can be linked together to form a ready queue. Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory management. Short-term (CPU) scheduling is the selection of one process from the ready queue.

5.7 Keywords

Process Creation, Process Scheduling, Interprocess, Process Management.

5.8 Exercises

1. Explain various Process Scheduling methods.
2. Explain interprocess communication.
3. What are the major activities of an operating system with regard to Process management?

5.9 Reference

1. Operating System Concepts, 8th Edition by Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, John Wiley and Sons, 2008.
2. Modern Operating Systems, 3rd Edition by Andrew S. Tanenbaum, Prentice Hall, 2008.
3. Beginning Linux Programming, 3rd Edition by Neil Matthew and Richard Stones, Wiley Publishing, 2004.

UNIT-6:

Structure

6.0 Objectives

6.1 Introduction

6.2 Threads

6.3 Multithreading Models

6.4 CPU/Process Scheduling

6.5 Scheduling Algorithms

6.6 Real Time Scheduling

6.7 Unit Summary

6.8 Keywords

6.9 Exercise

6.10 Reference

6.0Objectives

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

- To introduce the notion of a thread
- To explore CPU utilization that forms the basis of multithreaded computer systems.
- To discuss the APIs for Phtreads, Win32, and Java thread libraries.

6.1Introduction

The process model introduced in previous assumed that a process was an executing program with a single thread of control. Most modern operating systems now provide features enabling a process to contain multiple threads of control. This chapter introduces many concepts associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Win32, and Java thread libraries. We look at many issues related to multithreaded programming and how it affects the design of operating systems. Finally, we explore how the Windows XP and Linux operating systems support threads at the kernel level.

6.2Threads

Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

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). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has

thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Processes Vs Threads

As we mentioned earlier that in many respect threads operate in the same way as that of 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 processes, threads within processes execute sequentially.
- Like processes, thread can create children.
- And like process, 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?

Following are some reasons why we use threads in designing operating systems.

1. A process with multiple threads makes a great server for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.
4. Responsiveness
5. Resource Sharing

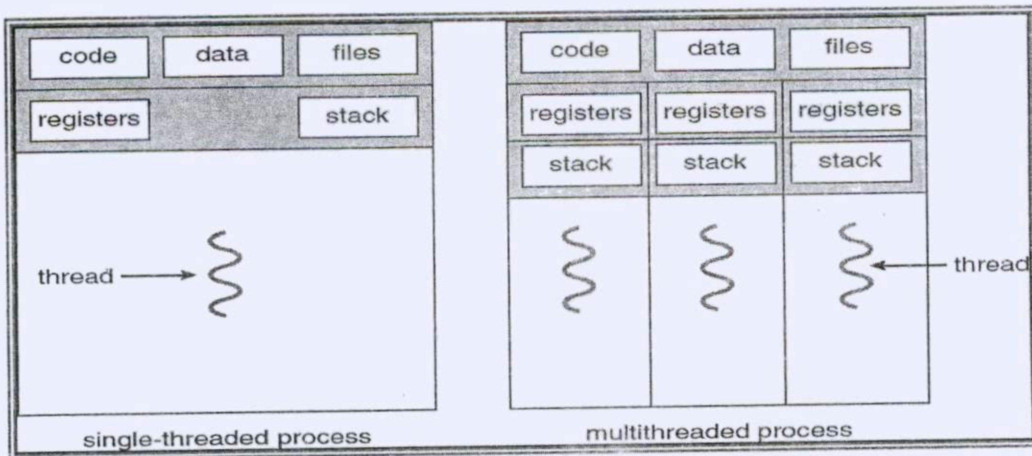
6. Economy
7. Utilization of MP Architectures

Threads are cheap in the sense that

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 is fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

But this cheapness does not come free - the biggest drawback is that there is no protection between threads.

Single and Multithreaded Processes



User-Level Threads

1. Thread management done by user-level threads library
2. Three primary thread libraries:

- POSIX Pthreads
- Win32 threads
- Java 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 a 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 more 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 irrespect 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.

Kernel-Level Threads

1. Supported by the Kernel
2. Examples

    - ❖ Windows XP/2000
    - ❖ Solaris
    - ❖ Linux
    - ❖ Tru64 UNIX
    - ❖ Mac OS X

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 Systems 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. For instance, threads operations are hundreds of times slower than that of user-level threads.
- 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 increased 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** Treads 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 file etc.

Disadvantages of Threads over Multiprocesses

- **Blocking** The major disadvantage if 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.

6.3 Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

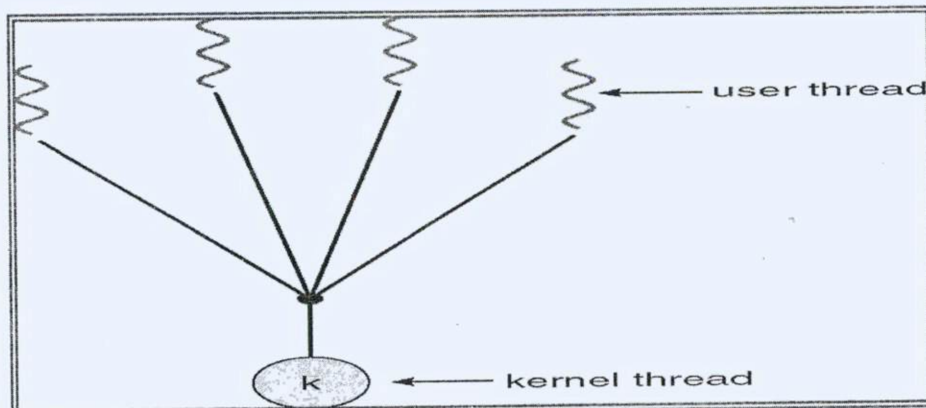
Many-to-One

Many user-level threads mapped to single kernel thread

-Examples:

-Solaris Green Threads

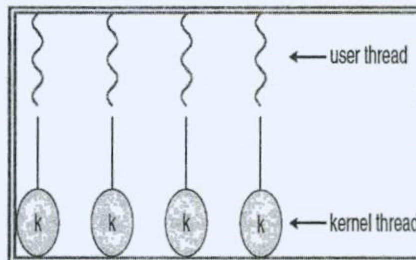
-GNU Portable Threads



One-to-One

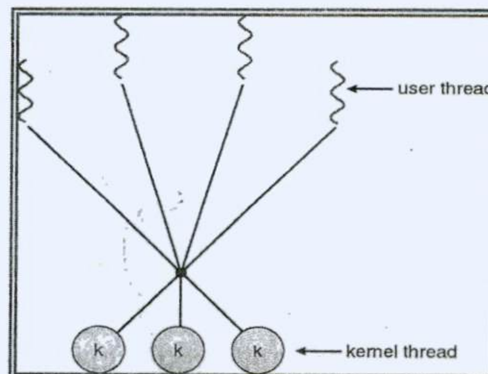
1. Each user-level thread maps to kernel thread
2. Examples

    - Windows NT/XP/2000
    - Linux
    - Solaris 9 and later

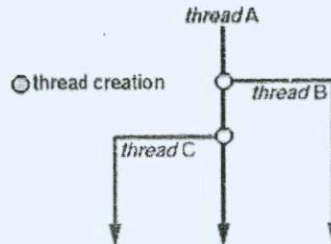


Many-to-Many Model

1. Allows many user level threads to be mapped to many kernel threads.
2. Allows the operating system to create a sufficient number of kernel threads.
3. Solaris prior to version 9.
4. Windows NT/2000 with the *ThreadFiber* package.



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.

Thread Pools

1. Create a number of threads in a pool where they await work
2. Advantages:

    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool

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.

6.4 CPU/Process Scheduling

The assignment of physical processors to processes allows processors to accomplish work. The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling.

When more than one process is runnable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm.

CPU Scheduler

a. Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
b. CPU scheduling decisions may take place when a process:

    1. Switches from running to waiting state
    2. Switches from running to ready state
    3. Switches from waiting to ready
    4. Terminates

- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

1. Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

    - switching context
    - switching to user mode
    - jumping to the proper location in the user program to restart that program

2. Dispatch latency – time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

1. CPU utilization – keep the CPU as busy as possible
2. Throughput – # of processes that complete their execution per time unit
3. Turnaround time – amount of time to execute a particular process
4. Waiting time – amount of time a process has been waiting in the ready queue
5. Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

General Goals

Fairness

Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

Policy Enforcement

The scheduler has to make sure that system's policy is enforced. For example, if the local

policy is safety then the *safety control processes* must be able to run whenever they want to, even if it means delay in *payroll processes*.

Efficiency

Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time

A scheduler should minimize the response time for interactive user.

Turnaround

A scheduler should minimize the time batch users must wait for an output.

Throughput

A scheduler should maximize the number of jobs processed per unit time.

A little thought will show that some of these goals are contradictory. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

Preemptive VsNonpreemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Nonpreemptive Scheduling

A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of nonpreemptive scheduling

1. In nonpreemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In nonpreemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.
3. In nonpreemptive scheduling, a schedular executes jobs in the following two situations.
   a. When a process switches from running state to the waiting state.
   b. When a process terminates.

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can takeaway.The strategy of allowing processes that are logically runable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

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.

Following are some scheduling algorithms we will study

- FCFS Scheduling.
- Round Robin Scheduling.
- SJF Scheduling.
- SRT Scheduling.
- Priority Scheduling.
- Multilevel Queue Scheduling.
- Multilevel Feedback Queue Scheduling.

A. First-Come-First-Served (FCFS) Scheduling

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

Perhaps, First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawbacks of this scheme is that the average time is often quite long.

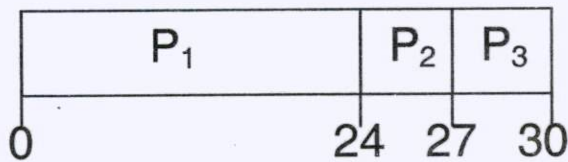
The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

Example:-

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: $P1$,$P2$, $P3$

The Gantt Chart for the schedule is:

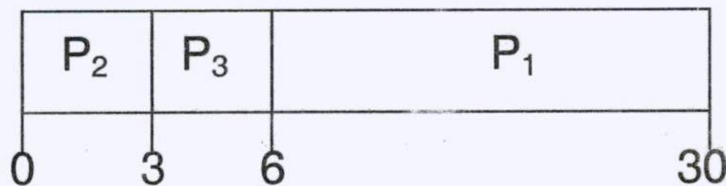


Waiting time for $P1 = 0$; $P2 = 24$; $P3 = 27$

Average waiting time: $(0 + 24 + 27)/3 = 17$

Suppose that the processes arrive in the order $P2$, $P3$, $P1$

The Gantt chart for the schedule is:



Waiting time for $P1 = 6$; $P2 = 0$; $P3 = 3$

Average waiting time: $(6 + 0 + 3)/3 = 3$

Much better than previous case

Convoy effect short process behind long process

B. Round Robin Scheduling

One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).

In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.

If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and appoximates FCFS.

In any event, the average waiting time under round robin scheduling is often quite long.

1. Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
2. 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 at once. No process waits more than $(n-1)q$time units.
3. Performance

    ->q large $\Rightarrow$ FIFO

    ->q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high.

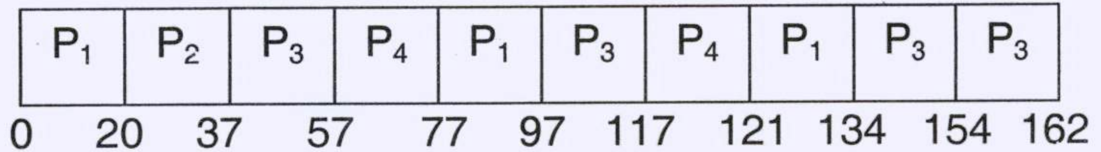
    Example:-

Process	Burst Time
P1	53
P2	17

P3 68

P4 24

The Gantt chart is:



->Typically, higher average turnaround than SJF, but better *response*

C. Shortest-Job-First (SJF) Scheduling

Other name of this algorithm is Shortest-Process-Next (SPN).

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.

The SJF algorithm favors short jobs (or processors) at the expense of longer ones.

The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

The best SJF algorithm can do is to rely on user estimates of run times.

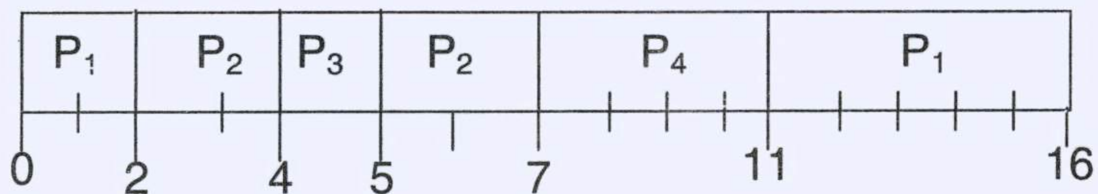
In the production environment where the same jobs run regularly, it may be possible to provide reasonable estimate of run time, based on the past performance of the process. But in the development environment users rarely know how their program will execute.

Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

1. Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
2. Two schemes:
   - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
   - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
3. SJF is optimal – gives minimum average waiting time for a given set of processes

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

->SJF (preemptive)



->Average waiting time = (9 + 1 + 0 +2)/4 = 3

D. Shortest-Remaining-Time (SRT) Scheduling

- The SRT is the preemtive counterpart of SJF and useful in time-sharing environment.
- In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
- In SJF scheme, once a job begins executing, it run to completion.
- In SJF scheme, a running process may be preempted by a new arrival process with shortest estimated run-time.
- The algorithm SRT has higher overhead than its counterpart SJF.
- The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.
- In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

E. Priority Scheduling

1. A priority number (integer) is associated with each process
2. The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)

   ->Preemptive

   ->nonpreemptive

3. SJF is a priority scheduling where priority is the predicted next CPU burst time
4. Problem ≡ Starvation – low priority processes may never execute
5. Solution ≡ Aging – as time progresses increase the priority of the process

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.

An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa.

Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are

- Time limits.
- Memory requirements.
- File requirements,
  - for example, number of open files.
- CPU Vs I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as

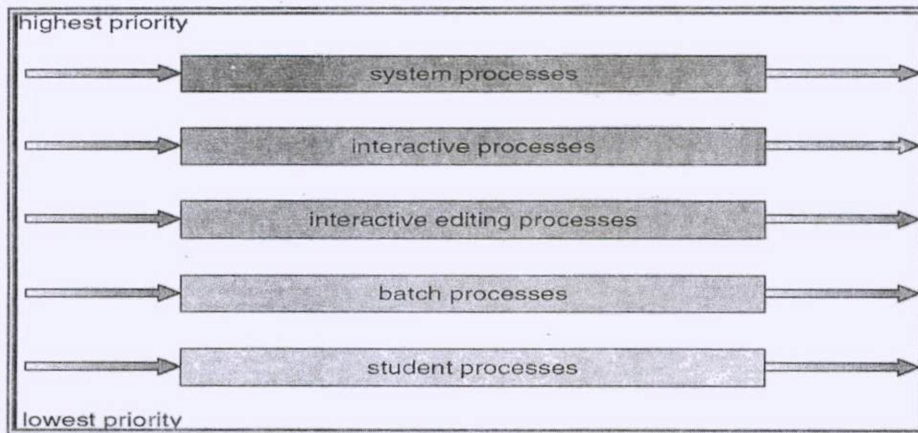
- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Priority scheduling can be either preemptive or non preemptive

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is *aging*. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

F. Multilevel Queue Scheduling



A multilevel queue scheduling algorithm partitions the ready queue in several separate queues, for instance

In a multilevel queue scheduling processes are permanently assigned to one queues.

The processes are permanently assigned to one another, based on some property of the process, such as

- Memory size
- Process priority
- Process type

Algorithm chooses the process from the occupied queue that has the highest priority, and run that process either

- Preemptive or
- Non-preemptively

Each queue has its own scheduling algorithm or policy.

Possibility I

If each queue has absolute priority over lower-priority queues then no process in the queue could run unless the queue for the highest-priority processes were all empty.

For example, in the above figure no process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes will all empty.

Possibility II

If there is a time slice between the queues then each queue gets a certain amount of CPU times, which it can then schedule among the processes in its queue. For instance;

- 80% of the CPU time to foreground queue using RR.
- 20% of the CPU time to background queues using FCFS.

Since processes do not move between queues so, this policy has the advantage of low scheduling overhead, but it is inflexible.

G. Multilevel Feedback Queue Scheduling

Multilevel feedback queue-scheduling algorithm allows a process to move between queues. It uses many ready queues and associate a different priority with each queue.

The Algorithm chooses to process with highest priority from the occupied queue and run that process either preemptively or unpreemptively. If the process uses too much CPU time it will moved to a lower-priority queue. Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue. Note that this form of aging prevents starvation.

- A process entering the ready queue is placed in queue 0.
- If it does not finish within 8 milliseconds time, it is moved to the tail of queue 1.
- If it does not complete, it is preempted and placed into queue 2.
- Processes in queue 2 run on a FCFS basis, only when 2 run on a FCFS basis queue, only when queue 0 and queue 1 are empty.

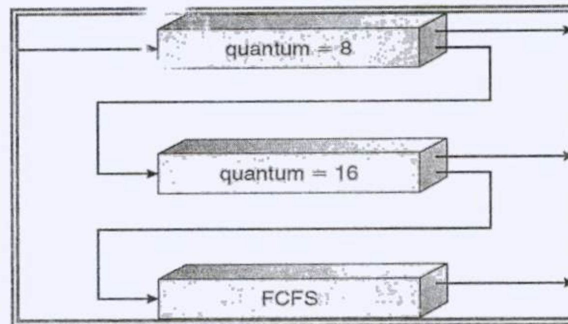
Example:-

1. Three queues:

   - $Q0$ – RR with time quantum 8 milliseconds
   - $Q1$ – RR time quantum 16 milliseconds
   - $Q2$ – FCFS

2. Scheduling

   - A new job enters queue $Q0$ which is servedFCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q1$.
   - At $Q1$ job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q2$.



6.6 Real Time Scheduling

Different classes of scheduling algorithm used in real-time systems:

• Clock-driven

– Primarily used for hard real-time systems where all properties of all jobs are known at design time, such that offline scheduling techniques can be used

• Weighted round-robin

– Primarily used for scheduling real-time traffic in high-speed, switched networks

• Priority-driven

– Primarily used for more dynamic real-time systems with a mix of time based and event-based activities, where the system must adapt to changing conditions and events

Clock-Driven Scheduling

• Decisions about what jobs execute when are made at specific time instants

– These instants are chosen before the system begins execution

– Usually regularly spaced, implemented using a periodic timer interrupt

• Scheduler awakes after each interrupt, schedules the job to execute for the next period, and then blocks itself until the next interrupt

• E.g. the helicopter example with an interrupt every $1/180^{th}$ of a second

• E.g. the furnace control example, with an interrupt every 100ms

• Typically in clock-driven systems:

– All parameters of the real-time jobs are fixed and known

– A schedule of the jobs is computed off-line and is stored for use at runtime; as a result, scheduling overhead at run-time can be minimized

– Simple and straight-forward, not flexible

Weighted Round-Robin Scheduling

Regular round-robin scheduling is commonly used for scheduling time-shared applications

– Every job joins a FIFO queue when it is ready for execution

– When the scheduler runs, it schedules the job at the head of the queue to execute for at most one time slice

• Sometimes called a quantum – typically O(tens of ms)

– If the job has not completed by the end of its quantum, it is preempted and placed at the end of the queue

– When there are n ready jobs in the queue, each job gets one slice every n time slices (n time Slices is called a round)

– Only limited use in real-time systems

Weighted Round-Robin Scheduling

• In weighted round robin each job J_i is assigned a weight w_i; the job will receive w_i Consecutive time slices each round, and the duration of a round is $\sum W_i$

– Equivalent to regular round robin if all weights equal 1

– Simple to implement, since it doesn't require a sorted priority queue

- Partitions capacity between jobs according to some ratio
- Offers throughput guarantees
  - Each job makes a certain amount of progress each round

6.7 Summary

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of multiprocessor architectures.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required. Three different types of models relate user and kernel threads: The many-to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

6.8 Keywords

Threads, CPU Scheduling, Real Time Scheduling, Multithreading.

6.9 Exercises

1. Explain various CPU Scheduling algorithms
2. Describe the actions taken by a thread
3. Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Describe.

6.10 Reference

1. Operating System Concepts, 8th Edition by Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, John Wiley and Sons, 2008.
2. Modern Operating Systems, 3rd Edition by Andrew S. Tanenbaum, Prentice Hall, 2008.
3. Beginning Linux Programming, 3rd Edition by Neil Matthew and Richard Stones, Wiley Publishing, 2004.

UNIT-7:

Structure

7.0 Objectives

7.1 Introduction

7.2 Race Conditions

7.3 The Critical-Section Problem

7.4 Synchronization Hardware

7.5 Semaphores

7.6 Classical Problems of Synchronization

7.7 Critical region

7.8 Monitors

7.9 Unit Summary

7.10 Keywords

7.11 Exercise

7.12 Reference

7.0 Objectives

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

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- To present both software and hardware solutions of the critical-section problem.
- To introduce the concept of atomic transaction and describe mechanisms to ensure atomicity.

7.1 Introduction

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of lightweight processes or threads, which we discussed in previous Chapter. Concurrent access to shared data may result in data inconsistency. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

7.2 Race Conditions

In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one 'customer' thread at a time should be allowed to examine and update the shared variable. Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

1. count++ could be implemented as

   register1 = count

   register1 = register1 + 1

   count = register1

2. count-- could be implemented as

   register2 = count

   register2 = register2 – 1

   count = register2

3. Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

A potential problem; the order of instructions of cooperating processes

Process A	Process B	concurrent access
X = 1;	Y = 2;	does not matter
X = Y + 1;	Y = Y * 2;	important!

- Producer-consumer problem. It is described that how a bounded buffer could be used to enable processes to share memory
    - **Bounded buffer problem**. The solution allows at most

      $$BUFFER_SIZE - 1$$

      items in the buffer at the same time.
    - An integer variable counter, initialized to 0. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

The code for the producer process:

```
while (true)
  {
    /* produce an item in next Produced */
while (counter == BUFFER_SIZE)
      ; /* do nothing */
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
counter++;
  }
```

The code for the consumer process:

```
while (true)
  {
while (counter == 0)
      ; /* do nothing */
nextConsumed = buffer [out] ;
out = (out + 1) % BUFFER_SIZE;
counter--j;
    /* consume the item in nextConsumed */
  }
```

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- We would arrive at incorrect state because we allowed both processes to manipulate the variable counter concurrently.

- A **race condition** is a situation where two or more processes access shared data concurrently and final value of shared data depends on timing (race to access and modify data)
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter (process synchronization).

7.3 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

General structure of a typical process P_i.

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

- A solution to the CS problem must satisfy the following requirements:

  1. **Mutual exclusion**. If process P_i is executing in its CS, then no other processes can be executing in their CSs.
  2. **Progress**. If no process is executing in its CS and some processes wish to enter their CSs, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its CS next, and this selection cannot be postponed indefinitely. (No process should have to wait forever to enter its CS.)
  3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted.
  4. **Fault tolerance**. Process running outside its CR should not block other processes accessing the CR.
  5. **No assumptions** may be made about speeds or the number of CPUs.

- **Atomic operation**. Atomic means either an operation happens in its entirely or NOT at all (it cannot be interrupted in the middle). Atomic operations are used to ensure that cooperating processes execute correctly.
- Machine instructions are atomic, high level instructions are not (count++; this is actually 3 machine level instructions, an interrupt can occur in the middle of instructions).
- Various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its CS, no other process will enter its CS and cause trouble.
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson's Solution
  - The TSL instructions (Hardware approach)

Solution to Critical-Section Problem

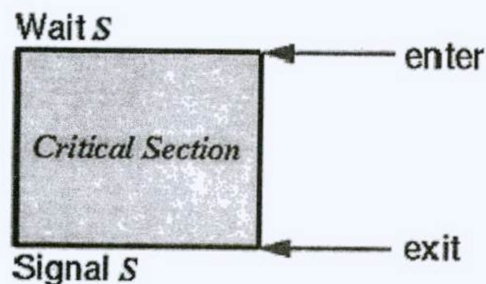
1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

Critical Section



The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the *Critical Section*. To avoid race conditions and flawed results, one must identify codes in *Critical Sections* in each thread. The characteristic properties of the code that form a *Critical Section* are

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

Here, the important point is that when one process is executing shared modifiable data in

its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

7.4 Synchronization Hardware

Hardware support can make some of this a little easier. Problems can arise when a process is preempted within a single high-level language line. But we can't preempt in the middle of a machine instruction.

If we have a single machine instruction that checks the value of a variable and sets it atomically, we can use that to our advantage. This is often called a Test-and-Set or Test and Set Lock instruction, and does this, atomically:

```
booleanTestAndSet(boolean *target)
{
booleanorig_val = *target;
*target = TRUE;
returnorig_val;
}
```

So it sets the variable passed in to true, and tells us if it was true or false before we called it. So if two processes do this operation, both will set the value of target to true, but only one will get a return value of false.

This is the functionality we want, but how does the instruction actually work?

Really, this would be an instruction that takes two operands:

```
TAS R, X
```

Where R is a CPU register and X is a memory location. After the instruction completes (atomically), the value that was in memory at X is copied into R, and the value of X is set to 1. R contains a 0 if X previously contained a 0. If two processes do this operation concurrently, only one will actually get the 0.

The Intel x86 BTS instruction sets a single bit in memory and sets the carry bit of the status word

to the previous value. This is equivalent. Think about how you might implement an atomic test and set on, for example, a micro programmed architecture. Any of these can be used to implement a "test and set" function as described above. Armed with this atomic test-and-set, we can make a simple mutual exclusion solution for any number of processes, with just a single shared variable:

• **Shared data**

```
boolean lock = false;
```

• **Process Pi**

```
while (1) {
while (TestAndSet(&lock)); /* busy wait */
/* critical section */
lock = false;
/* non-critical section */
}
```

This satisfies mutual exclusion and progress, but not bounded waiting (a process can leave the CS and come back around and grab the lock again before others who may be waiting ever get a chanceto look).

A solution that does satisfy bounded waiting is still fairly complicated:

• **Shared data**

```
boolean lock=false;
boolean waiting[n]; /* all initialized to false */
```

• **Process Pi and its local data**

```
int j;
boolean key;
while (1) {
waiting[i]=true;
key=true;
while (waiting[i] && key)
key = TestAndSet(&lock);
waiting[i]=false;
/* critical section */
```

```
j=(i+1)%n;
while ((j!=i) && !waiting[j])
j=(j+1)%n;
if (j==i) lock=false;
else waiting[j]=false;
/* non-critical section */
}
```

Another hardware instruction that might be available is the atomic swap operation:

```
void swap(boolean *a, boolean *b) {
boolean temp = *a;
*a = *b;
*b = temp;
}
```

Different architectures have variations on this one, including the x86 XCHG instruction.

An algorithm to use this, minus the bounded wait again, is straightforward:

• **Shared data**

```
boolean lock = false;
```

• **Process Pi**

```
boolean key = true;
while (1) {
while (key == true) swap(&key,&lock); /* busy wait */
/* critical section */
lock = false;
/* non-critical section */
}
```

It's pretty similar to what we saw before withTestAndSet().

7.5 Semaphores

All that busy waiting in all of our algorithms for mutual exclusion is pretty annoying. It's just wasted time on the CPU. If we have just one CPU, it doesn't make sense for that process to take up its whole quantum spinning away waiting for a shared variable to change that can't change

until the current process relinquishes the CPU!

This inspired the development of the semaphore. The name comes from old-style railroad traffic control signals where mechanical arms swing down to block a train from a section of track that another train is currently using. When the track was free, the arm would swing up, and the waiting train could now proceed. A semaphore S is basically an integer variable, with two atomic operations:

```
wait(S):
while (S <= 0); /* wait */
S--;
signal(S):
S++;
```

wait and signal are also often called down and up (from the railroad semaphore analogy) and occasionally are called P and V (because Dijkstra, who invented them, was Dutch, and these are the first letters of the Dutch words proberen (to test) and verhogen (to increment)).

Important!!! Processes using a semaphore in its most pure form are not allowed to set or examine its value. They can use the semaphore only through the wait and signal operations.

Note, however, that we don't want to do a busy-wait. A process that has to wait should be put to sleep, and should wake up only when a corresponding signal occurs, as that is the only time the process has any chance to proceed.

Semaphores are built using hardware support, or using software techniques such as the ones we discussed for critical section management. Since the best approach is just to take the process out of the ready queue, some operating systems provide semaphores through system calls. We will examine their implementation in this context soon.
Given semaphores, we can create a much simpler solution to the critical section problem for n processes:

- **Shared data**

```
semaphoremutex=1;
• Process Pi
while (1) {
wait(mutex);
/* critical section */
signal(mutex);
/* non-critical section */
}
```

The semaphore provides the mutual exclusion for sure, and should satify progress, but depending on the implementation of semaphores, may or may not provide bounded waiting. A semaphore implementation might look like this:

```
struct semaphore
{
int value;
proclist L;
};
```

• blockoperation suspends the calling process, removes it from consideration by the scheduler

• wakeup(P) resumes execution of suspended process P, puts it back into consideration

```
• wait(S):
S.value--;
if (S.value< 0) {
add this process to S.L;
block;
}

• signal(S):
S.value++;
if (S.value<= 0) {
remove a process P from S.L;
wakeup(P);
}
```

There is a fairly standard implementation of semaphores on many Unix systems: POSIX semaphores

• create a shared variable of typesem t
• initialize it with seminit(3)
• wait operation issemwait(3)
• signal operation issem post(3)
• deallocate withsemdestroy(3)

Examples using POSIX semaphores and a semaphore-like construct called a mutex provided by pthreads to provide mutual exclusion in the bounded buffer problem:

7.6 Classical Problems of Synchronization

We will use semaphores to consider some synchronization problems. While some actual implementations provide the ability to "try to wait", or to examine the value of a semaphore's counter, we restrict ourselves to initialization, wait, and signal.

Bounded buffer using semaphores

First, we revisit our friend the bounded buffer.

• **Shared data:**

semaphorefullslots, emptyslots, mutex;
full=0; empty=n; mutex=1;

• Producer process:

```
while (1)
{
produce item;
wait(emptyslots);
wait(mutex);
add item to buffer;
signal(mutex);
signal(fullslots);
}
```

• Consumer process:

```
while (1)
{
wait(fullslots);
```

```
wait(mutex);
remove item from buffer;
signal(mutex);
signal(emptyslots);
consume item;
}
```

mutexprovides mutual exclusion for the modification of the buffer (not shown in detail). The others make sure that the consumer doesn't try to remove from an empty buffer (fullslots is > 0) or that the producer doesn't try to add to a full buffer (emptyslots is > 0).

7.7 Critical region

A critical region is a simple mechanism that prevents multiple threads from accessing at once code protected by the same critical region.

The code fragments could be different, and in completely different modules, but as long as the critical region is the same, no two threads should call the protected code at the same time. If one thread is inside a critical region, and another thread wants to execute code protected by the same critical region, the second thread must wait for the first thread to exit the critical region. In some implementations a critical region can be set so that if it takes too long for the first thread to exit said critical region, the second thread is allowed to execute, dangerously and potentially causing crashes. This is the case for the critical regions exposed by Max and the default upper limit for a given thread to remain inside a critical region is two seconds. Despite the fact that there are two seconds of leeway provided before two threads can dangerously enter a critical region, it is important to only protect as small a portion of code as necessary with a critical region.

Under Max 4.1 and earlier there was a simple protective mechanism called "lockout" that would prevent the scheduler from interrupting the low priority thread during sensitive operations such as sending data out an outlet or modifying members of a linked list. This lockout mechanism has been deprecated, and under the Mac OS X and Windows XP versions (Max 4.2 and later) does nothing. So how do you protect thread sensitive operations? Use critical regions (also known as critical sections). However, it is very important to mention that all outlet calls are now thread safe and should never be contained inside a critical region. Otherwise, this could result in serious timing problems. For other tasks which are not thread safe, such as accessing a linked list,

critical regions or some other thread protection mechanism are appropriate.

In Max, the critical_enter() function is used to enter a critical region, and the critical_exit() function is used to exit a critical region. It is important that in any function which uses critical regions, all control paths protected by the critical region, exit the critical region (watch out for goto or return statements). The critical_enter() and critical_exit() functions take a critical region as an argument. However, for almost all purposes, we recommend using the global critical region in which case this argument is zero. The use of multiple critical regions can cause problems such as deadlock, i.e. when thread #1 is inside critical region A waiting on critical region B, but thread #2 is inside critical region B and is waiting on critical region A. In a flexible programming environment such as Max, deadlock conditions are easier to generate than you might think. So unless you are completely sure of what you are doing, and absolutely need to make use of multiple critical regions to protect your code, we suggest you use the global critical region.

In the following example code we show how one might use critical regions to protect the traversal of a linked list, testing to find the first element whose values is equal to "val". If this code were not protected, anotner thread which was modifying the linked list could invalidate assumptions in the traversal code.

```
critical_enter(0);
for (p = head; p; p = p->next) {
if (p->value == val)
break;
  }
critical_exit(0);
return p;
```

And just to illustrate how to ensure a critical region is exited when multiple control paths are protected by a critical region, here's a slight variant.

```
critical_enter(0);
for (p = head; p; p = p->next) {
if (p->value == val) {
critical_exit(0);
return p;
    }
```

```
  }
critical_exit(0);
return NULL;
```

7.8 Monitors

Monitors were developed in the 1970s to make it easier to avoid deadlocks.

- A monitor is a collection of procedures, variables, and data structures grouped together.
- Processes can call the monitor procedures but cannot access the internal data structures.
- Only one process at a time may be be **active** in a monitor.

  Active in a monitor means in ready queue or CPU with the program counter somewhere in a monitor method.
- A monitor is a language construct.

  Compare this with semaphores, which are usually an OS construct.
- The compiler usually enforces mutual exclusion.
- Condition variables allow for blocking and unblocking.
  - cv.wait() blocks a process.

    The process is said to be waiting for (or waiting on) the condition variable cv.
  - cv.signal() (also called cv.notify) unblocks a process waiting for the condition variable cv.

    When this occurs, we need to still require that only one process is active in the monitor. This can be done in several ways:
    - on some systems the old process (the one executing the signal) leaves the monitor and the new one enters
    - on some systems the signal must be the last statement executed inside the monitor.
    - on some systems the old process will block until the monitor is available again.
    - on some systems the new process (the one unblocked by the signal) will remain blocked until the monitor is available again

- If a condition variable is signaled with nobody waiting, the signal is lost.

  Compare this with semaphores, in which a signal will allow a process that executes a wait in the future to no block.
- You should not think of a condition variable as a variable in the traditional sense.

  It does not have a value.

  Think of it as an object in the OOP sense.

  It has two methods, wait and signal that manipulate the calling process.

7.9 Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section of code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user-coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

7.10 Keywords

Critical Section, Synchronization Hardware, Semaphores, Critical Regions, Monitors.

7.11 Exercises

1. Discuss the critical section problem.
2. Explain semaphores with example
3. Explore Classical Problems of Synchronization

7.12 Reference

1. Silberschatz and Galvin, Operating system concepts, Addison-Wesiey publication, 5th edition.
2. Achyut S Godbole, Operating systems, Tata McGraw-Hill publication.
3. Milan Milankovic, Operating systems concepts and design, Tata McGraw-Hill publication, 2nd edition.

UNIT-8:

Structure

8.0 Objectives

8.1 Introduction

8.2 System Model

8.3 Deadlock Characterization

8.4 Methods for Handling Deadlocks

8.5 Deadlock Prevention

8.6 Deadlock Avoidance

8.7 Deadlock Detection

8.8 Recovery from Deadlock

8.9 Unit Summary

8.10 Keywords

8.11 Exercise

8.12 Reference

8.0 Objectives

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

- Describe the Deadlock Characterization
- Explain the Methods for Handling Deadlocks.
- Discuss Deadlock Avoidance and Deadlock Detection.

8.1 Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. We discussed this issue briefly in Chapter 6 in connection with semaphores.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: "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.

8.2 System Model

When processes request a resource and if the resources are not available at that time the process enters into waiting state. Waiting process may not change its state because the resources they are requested are held by other process. This situation is called deadlock.

The situation where the process waiting for the resource i.e., not available is called deadlock.

- A system may consist of finite number of resources and is distributed among number of processes. There resources are partitioned into several instances each with identical instances.

- A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.
- A process may utilize the resources in only the following sequences:

1. Request:- If the request is not granted immediately then the requesting process must wait it can acquire the resources.

2. Use:- The process can operate on the resource.

3. Release:- The process releases the resource after using it.

Deadlock may involve different types of resources.

For eg:- Consider a system with one printer and one tape drive. If a process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs.

Multithread programs are good candidates for deadlock because they compete for shared resources.

8.3 Deadlock Characterization

Necessary Conditions:-

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-

1. Mutual Exclusion:- Only one process must hold the resource at a time. If any other process requests for the resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait:-A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.

3. No Preemption:- Resources can't be preempted i.e., only the process holding the resources must release it after the process has completed its task.

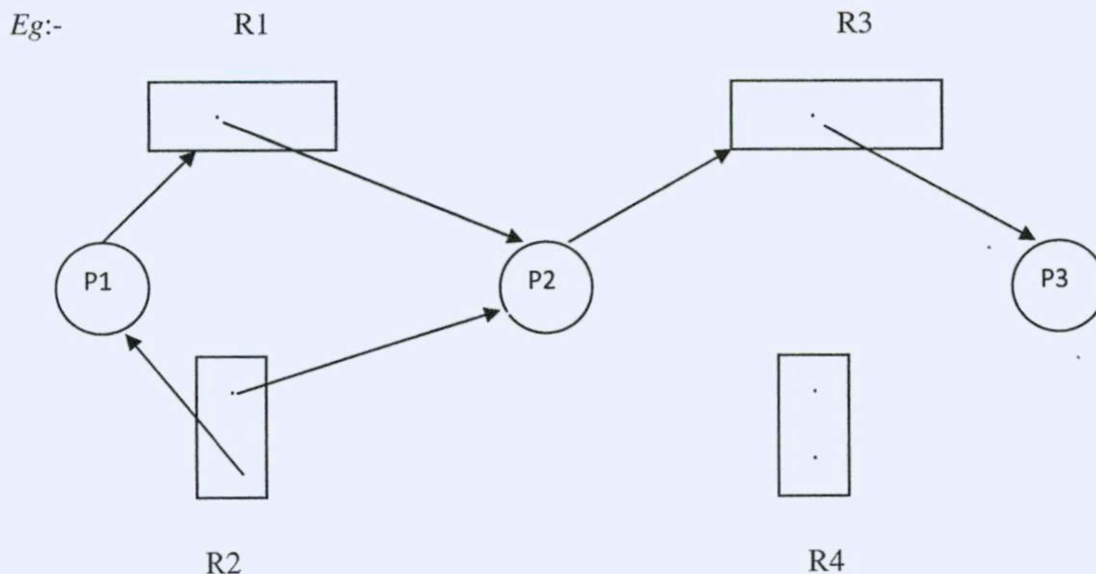
4. Circular Wait:- A set {P0,P1……..Pn} of waiting process must exist such that P0 is waiting for a resource i.e., held by P1, P1 is waiting for a resource i.e., held by P2. Pn-1 is waiting for resource held by process Pn and Pn is waiting for the resource i.e., held by P1.

All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:-

- Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).
- The set of vertices (v) can be described into two different types of nodes P={P1,P2……..Pn} i.e., set consisting of all active processes and R={R1,R2……….Rn}i.e., set consisting of all resource types in the system.
- A directed edge from process Pi to resource type Rj denoted by Pi→Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called Request edge.
- A directed edge Ri→Pj signifies that resource Rj is held by process Pi. This is called Assignment edge.

Eg:-



- If the graph contain no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.
- If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.

8.4 Methods for Handling Deadlocks

There are three ways to deal with deadlock problem

- We can use a protocol to prevent deadlocks ensuring that the system will never enter into the deadlock state.
- We allow a system to enter into deadlock state, detect it and recover from it.
- We ignore the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.
    - ❖ To ensure that the deadlock never occur the system can use either deadlock avoidance or a deadlock prevention.
    - ❖ Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.
    - ❖ Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.
    - ❖ If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. During this it can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and algorithm to recover from deadlock.
    - ❖ Undetected deadlock will result in deterioration of the system performance.

8.5 Deadlock Prevention

For a deadlock to occur each of the four necessary conditions must hold. If at least one of the three condition does not hold then we can prevent occurrence of deadlock.

Mutual Exclusion:-This holds for non-sharable resources.

Eg:- A printer can be used by only one process at a time. Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

Hold and Wait:- This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available.
One protocol can be used is that each process is allocated with all of its resources before its start execution.

Eg:- consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

Another protocol that can be used is to allow a process to request a resource when the process has none. i.e., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible.

No Preemption:- To ensure that this condition never occurs the resources must be preempted. The following protocol can be used.

- If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.
- When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting

process and allocate them to the requesting process. The requesting process must wait.

Circular Wait:- The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order.

Let R={R1,R2,.........Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering.

Eg:-we can define a one to one function

F:R→N as follows :- F(disk drive)=5

F(printer)=12

F(tape drive)=1

Deadlock can be prevented by using the following protocol:-

- Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri).
- Alternatively when a process requests an instance of resource type Rj, it has released any resource Ri such that F(Ri) >= F(Rj).

- If these two protocol are used then the circular wait can't hold.

8.6 Deadlock Avoidance

- Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.

- Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- For each requests it requires to check the resources currently available, resources that are currently allocated to each processes future requests and release of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.
- A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

Safe State:-

- A state is a safe state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.
- A system is in safe state if there exists a safe sequence.
- A sequence of processes <P1,P2,..........Pn> is a safe sequence for the current allocation state if for each Pi the resources that Pi can request can be satisfied by the currently available resources.
- If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- A safe state is not a deadlock state.
- Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.
- In this, if a process requests a resource i.e., currently available it must still have to wait. Thus resource utilization may be lower than it would be without a deadlock avoidance algorithm.

Resource Allocation Graph Algorithm:-

➔ This algorithm is used only if we have one instance of a resource type. In addition to the request edge and the assignment edge a new edge called claim edge is used.
For eg:- A claim edge Pi➔Rj indicates that process Pi may request Rj in future.

The claim edge is represented by a dotted line.

- When a process Pi requests the resource Rj, the claim edge is converted to a request edge.
- When resource Rj is released by process Pi, the assignment edge Rj➔Pi is replaced by the claim edge Pi➔Rj.

➔ When a process Pi requests resource Rj the request is granted only if converting the request edge Pi➔Rj to as assignment edge Rj➔Pi do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state

Banker's Algorithm:-

➔ This algorithm is applicable to the system with multiple instances of each resource types, but this is less efficient then the resource allocation graph algorithm.

➔ When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.

➔ Several data structures are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types. We need the following data structures:-

- **Available**:- A vector of length m indicates the number of available resources. If Available[i]=k, then k instances of resource type Rj is available.
- **Max**:- An n*m matrix defines the maximum demand of each process if Max[i,j]=k, then Pi may request at most k instances of resource type Rj.

- **Allocation**:- An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j]=k, then Pi is currently k instances of resource type Rj.
- **Need**:- An n*m matrix indicates the remaining resources need of each process. If Need[i,j]=k, then Pi may need k more instances of resource type Rj to compute its task. So Need[i,j]=Max[i,j]-Allocation[i]

Safety Algorithm:-

This algorithm is used to find out whether or not a system is in safe state or not.

Step 1. Let work and finish be two vectors of length M and N respectively.

Initialize work = available and

Finish[i]=false for i=1,2,3,.......n

Step 2. Find i such that both

Finish[i]=false

Need i <= work

If no such i exist then go to step 4

Step 3. Work = work + Allocation

Finish[i]=true

Go to step 2

Step 4. If finish[i]=true for all i, then the system is in safe state.

This algorithm may require an order of m*n*n operation to decide whether a state is safe.

Resource Request Algorithm:-

Let Request(i) be the request vector of process Pi. If Request(i)[j]=k, then process Pi wants K instances of the resource type Rj. When a request for resources is made by process Pi the following actions are taken.

- If Request(i) <= Need(i) go to step 2 otherwise raise an error condition since the process has exceeded its maximum claim.
- If Request(i) <= Available go to step 3 otherwise Pi must wait. Since the resources are not available.
- If the system want to allocate the requested resources to process Pi then modify the state as follows.

  Available = Available – Request(i)

  Allocation(i) = Allocation(i) + Request(i)

  Need(i) = Need(i) – Request(i)

- If the resulting resource allocation state is safe, the transaction is complete and Pi is allocated its resources. If the new state is unsafe then Pi must wait for Request(i) and old resource allocation state is restored.

8.7 Deadlock Detection

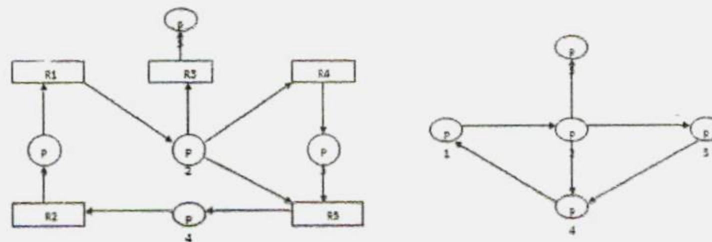
If a system does not employ either deadlock prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment the system may provide

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

Single Instances of each Resource Type:-

- If all the resources have only a single instance then we can define deadlock detection algorithm that uses a variant of resource allocation graph called a wait for graph. This graph is obtained by removing the nodes of type resources and removing appropriate edges.

- An edge from Pi to Pj in wait for graph implies that Pi is waiting for Pj to release a resource that Pi needs.
- An edge from Pi to Pj exists in wait for graph if and only if the corresponding resource allocation graph contains the edges Pi→Rq and Rq→Pj.
- Deadlock exists within the system if and only if there is a cycle. To detect deadlock the system needs an algorithm that searches for cycle in a graph.



8.8 Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - One is simply to abort one or more processes to break the circular wait.
  - The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

- Abort all deadlocked processes. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Irrecoverable losses or erroneous results may occur, even if this is the least important process.
- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. We should abort those processes whose termination will incur the minimum cost.

Resource Preemption

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. Highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible.

1. Selecting a victim: Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.

2. Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource.

   - ❖ Check pointing: means that its state is written to a file so that it can be restarted later.
   - ❖ The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process.

   - ❖ When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints.

   - ❖ Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it.

- ❖ Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

8.9 Summary

A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. There are three principal methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state, detect it, and then recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

8.10 Keywords

Deadlock, Banker's Algorithm, Safe State, Mutual Exclusion.

8.11 Exercises

1. Explain characterization of deadlock.
2. Explain different methods to recover from deadlock?
3. Write advantage and disadvantage of deadlock avoidance and deadlock prevention

8.12 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT 9

Structure

9.0 Objectives

9.1 Introduction

9.2 File Concept

9.3 Access Methods

9.4 Directory Structure

9.5 Unit Summary

9.6 Keywords

9.7 Exercise

9.8 Reference

9.0Objectives

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

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore file-system protection.

9.1Introduction

For most users, the 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 the 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. File systems live on devices, which we explore fully in the following chapters but touch upon here. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and we want to control who may access files and how files may be accessed.

9.2File Concept

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.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

File Attributes

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, usually as a sequence of bytes, with two views:

- ❖ Logical (programmer) view, as the users see it (how they are used and what properties they have.).
- ❖ Physical (OS) view, as it actually resides on secondary storage.

- The information in a file is defined by its creator. Commonly, files represent programs (both source and object forms) and data.
    - ❖ Data files may be numeric, alphabetic, alphanumeric, or binary.
    - ❖ Files may be free form, such as text files, or may be formatted rigidly.
- In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.
- A file has a certain defined structure, which depends on its **type**.
    - ❖ A text file is a sequence of characters organized into lines (and possibly pages).
    - ❖ A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.
    - ❖ An object file is a sequence of bytes organized into blocks understandable by the system's linker.
    - ❖ An executable file is a series of code sections that the loader can bring into memory and execute.

File Operations

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
    - o **Creating a file**. Two steps are necessary to create a file.
        1. Space in the file system must be found for the file.
        2. An entry for the new file must be made in the directory.
    - o **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.

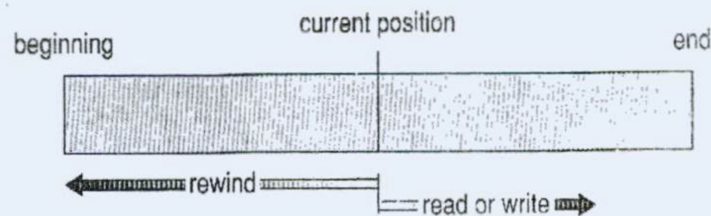
9.3 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

Sequential Access

- The simplest access method is sequential access. Information in the file is processed in order, one record after the other.
- This mode of access is by far the beginning current position most common; for example, editors and compilers usually access files in this fashion.
- Reads and writes make up the bulk of the operations on a file.
    - ❖ A read operation read next reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
    - ❖ Similarly, the write operation write next appends to the end of the file and advances to the end of the newly written material (the new end of file).



- Sequential access, which is depicted in Figure, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.

- For the direct-access method, the file operations must be modified to include the block number as a parameter.
- The block number provided by the user to the OS is normally a relative block number.
    - A relative block number is an index relative to the beginning of the file.
    - Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.
- The use of relative block numbers allows the OS to decide where the file should be placed (called the allocation problem) and helps to prevent the user from accessing portions of the file system that may not be part of her file.

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

- Modern OSs has all their files are automatically random access.

Other Access Methods

- Other access methods can be built on top of a direct-access method.
- These methods generally involve the construction of an index for the file.
    - To find a record in the file, we first search the index to learn exactly which block contains the desired record
    - and then use the pointer to access the file directly and to find the desired record.
- This structure allows us to search a large file doing little I/O. But, with large files, the index file itself may become too large to be kept in memory.
- One solution is to create an index for the index file.
    - The primary index file would contain pointers to secondary index files, which would point to the actual data items.

- For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The file is kept sorted on a defined key.
- To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index.
- The secondary index blocks point to the actual file blocks. This block is read in, and again a binary search is used to find the block containing the desired record.
- Finally, this block is searched sequentially.

- In this way any record can be located from its key by at most two direct-access reads

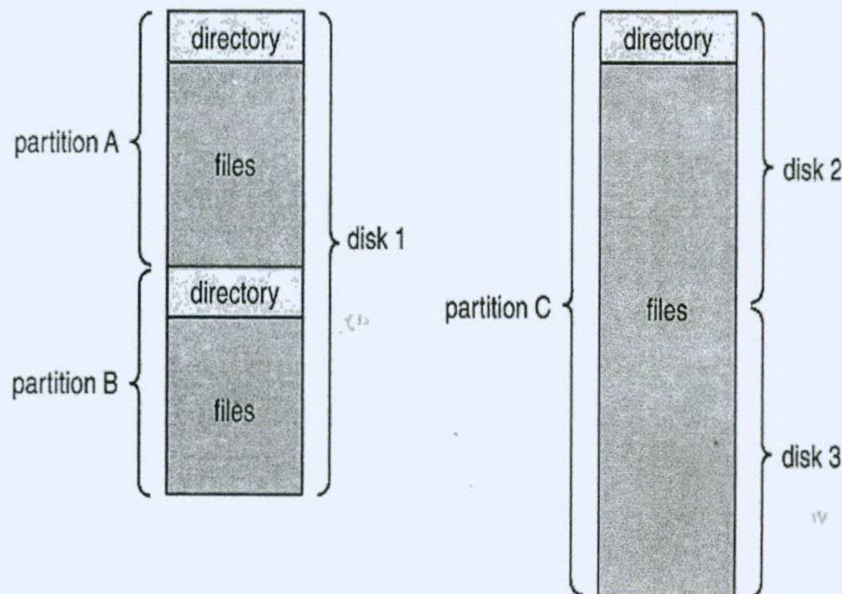
last name
logical record number
Adams
Arthur
Asher
Smith
Smith, John
social-security
age
index file
relative file

9.4 Directory Structure

The file systems of computers, then, can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization involves the use of directories.

Storage Structure

- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions**, **slices**, or (in the IBM world) **minidisks**.
- A file system can be created on each of these parts of the disk. We simply refer to a chunk of storage that holds a file system as a **volume**.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**.
- The device directory (more commonly known simply as a directory) records information-such as name, location, size, and type-for all files on that volume. Figure shows a typical file-system organization.



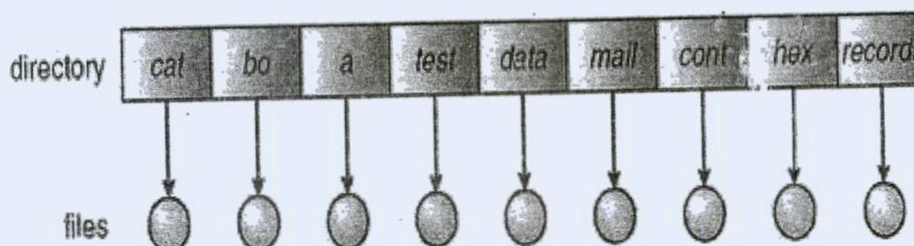
Directory Overview

- To keep track of files, file systems normally have directories or folders. Usually, a directory is itself a file.

- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- We want to be able
    - to insert entries,
    - to delete entries,
    - to search for a named entry,
    - to list all the entries in the directory.
- When considering a particular directory structure! we need to keep in mind the operations that are to be performed on a directory:
    - **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file.
    - **Create a file**. New files need to be created and added to the directory.
    - **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.
    - **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
    - **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.
    - **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals (backup copy).

Single-Level Directory

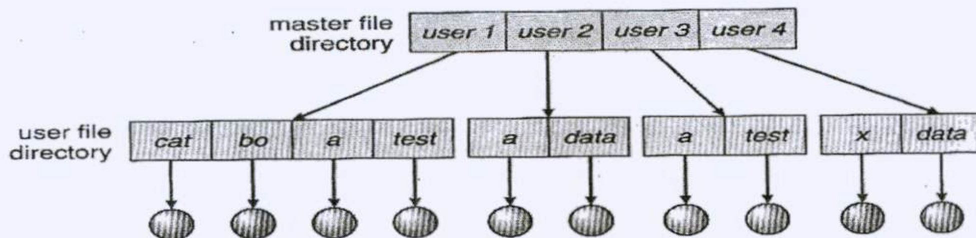
- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand



- On early personal computers, this system was common, in part because there was only one user. The world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once.
- A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data file *test*, then the unique-name rule is violated.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

Two-Level Directory

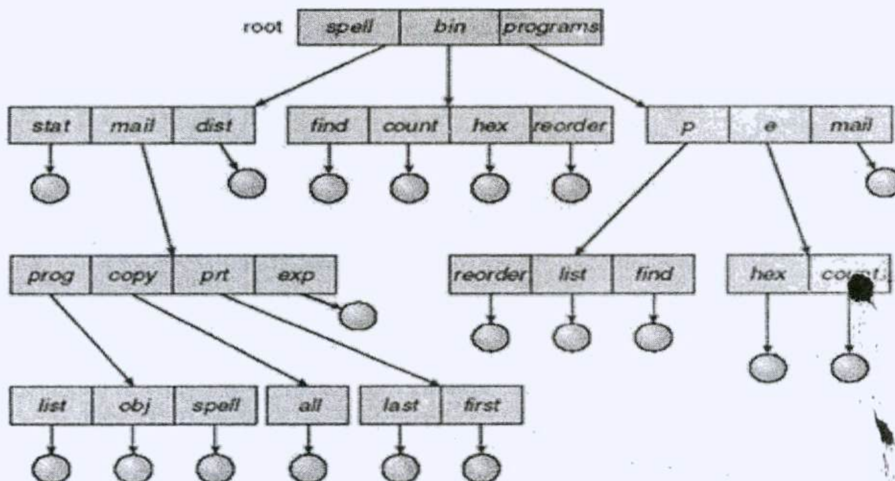
- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user



- When a user refers to a particular file, only his own UFD is searched (create a file, delete a file?).
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages.
- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2.
    - The root of the tree is the MFD.
    - Its direct descendants are the UFDs.
    - The descendants of the UFDs are the files themselves. The files are the leaves of the tree.
- Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).
- Thus, a user name and a file name define a **Path** name. To name a file uniquely, a user must know the path name of the file desired.
- Additional syntax is needed to specify the volume of a file. For instance, in MS-DOS a volume is specified by a letter followed by a colon. Thus, a file specification might be C:\userb\test

Tree-Structured Directories

- Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height



- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way. All directories have the same internal format.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**

1. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
2. A relative path name defines a path from the current directory.

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.

- For example, user B can access a file of user A by specifying its path names.
- User B can specify either an absolute or a relative path name.
- Alternatively, user B can change her current directory to be user A's directory and access the file by its file names.

9.5 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 (of fixed or variable length), 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.

9.6 Keywords

Access methods, file structure, file concepts.

9.7 Exercises

1. Explain briefly about the file concept.
2. What are access methods? Briefly explain
3. What does directory structure mean?

9.8 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-10:

Structure

10.0 Objectives

10.1 Introduction

10.2 File System mounting

10.3 File Structure

10.4 Directory implementations

10.5 Allocation methods

10.6 Free Space Managements

10.7 Unit Summary

10.8 Keywords

10.9 Exercise

10.10 Reference

10.0Objectives

After going through this unit, you will be able to: Describe File System

- Describe Mounting
- Define Directory implementation
- Describe Free Space management

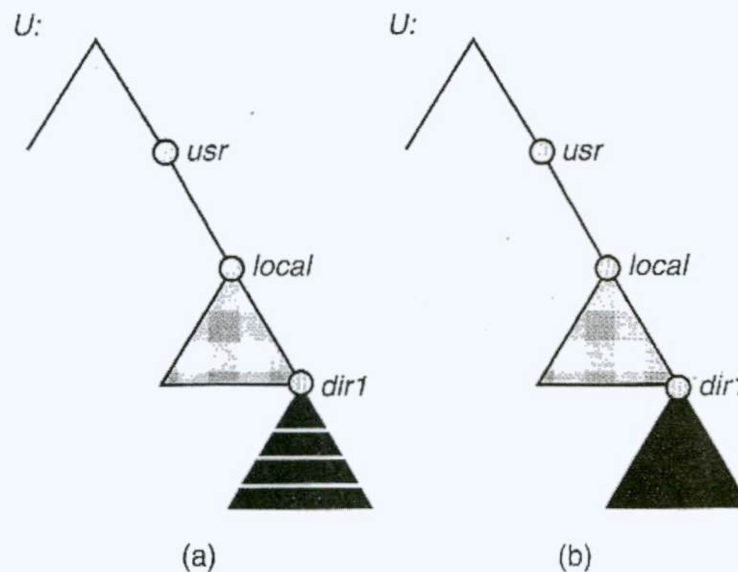
10.1Introduction

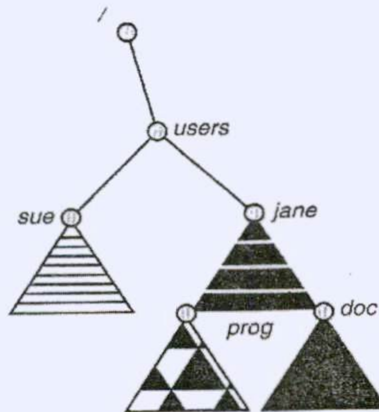
A **file system** is a means to organize data expected to be retained after a program terminates by providing procedures to store, retrieve and update data, as well as manage the available space on the device(s) which contain it. A file system organizes data in an efficient manner and is tuned to the specific characteristics of the device. A tight coupling usually exists between the operating system and the file system. Some file systems provide mechanisms to control access to the data and metadata. Ensuring reliability is a major responsibility of a file system. Some file systems allow multiple programs to update the same file at nearly the same time.

File systems are used on data storage devices, such as hard disk drives, floppy disks, optical discs, or flash memory storage devices, to maintain the physical locations of the computer files. They may provide access to data on a file server by acting as clients for a network protocol or 9P clients), or they may be virtual and exist only as an access method for virtual data. This is distinguished from a directory service and registry.

10.2File System mounting

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point** -the location within the file structure where the file system is to be attached.
- Typically, a mount point is an empty directory. Next, the OS verifies that the device contains a valid file system.
- Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point.
- This scheme enables the OS to traverse its directory structure, switching among file systems as appropriate.
- To illustrate file mounting, consider the file system depicted in Figure, where the triangles represent sub-trees of directories that are of interest. At this point, only the files on the existing file system can be accessed.



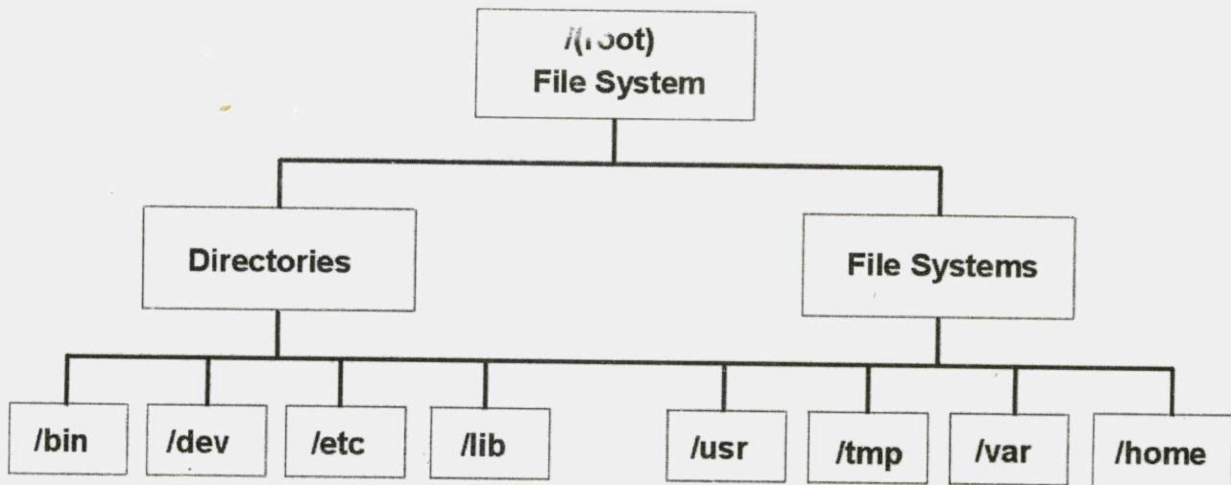


10.3 File Structure

It is important to understand the difference between a file system and a directory. A file system is a section of hard disk that has been allocated to contain files. This section of hard disk is accessed by mounting the file system over a directory. After the file system is mounted, it looks just like any other directory to the end user. However, because of the structural differences between the file systems and directories, the data within these entities can be managed separately.

When the operating system is installed for the first time, it is loaded into a directory structure, as shown in the following illustration.

Figure 1. / (root) File System Tree. This tree chart shows a directory structure with the / (root) file system at the top, branching downward to directories and file systems. Directories branch to /bin,/dev, /etc, and /lib. File systems branch to /usr, /tmp, /var, and /home.



The directories on the right (/usr, /tmp, /var, and /home) are all file systems so they have separate sections of the hard disk allocated for their use. These file systems are mounted automatically when the system is started, so the end user does not see the difference between these file systems and the directories listed on the left (/bin, /dev, /etc, and /lib).

On standalone machines, the following file systems reside on the associated devices by default:

/File System	/Device
/dev/hd1	/home
/dev/hd2	/usr
/dev/hd3	/tmp
/dev/hd4	/(root)
/dev/hd9var	/var
/proc	/proc
/dev/hd10opt	/opt

The file tree has the following characteristics:

- Files that can be shared by machines of the same hardware architecture are located in the /usr file system.
- Variable per-client files, for example, spool and mail files, are located in the /var file system.
- The /(root) file system contains files and directories critical for system operation. For example, it contains
- A device directory (/dev)
- Mount points where file systems can be mounted onto the root file system, for example, /mnt
- The /home file system is the mount point for users' home directories.
- For servers, the /export directory contains paging-space files, per-client (unshared) root file systems, dump, home, and /usr/share directories for diskless clients, as well as exported /usr directories.
- The /proc file system contains information about the state of processes and threads in the system.
- The /opt file system contains optional software, such as applications.

The following list provides information about the contents of some of the subdirectories of the /(root)file system.

/bin	Symbolic link to the /usr/bin directory.
/dev	Contains device nodes for special files for local devices. The /dev directory contains special files for tape drives, printers, disk partitions, and terminals.
/etc	Contains configuration files that vary for each machine. Examples include: /etc/hosts /etc/passwd

/export	Contains the directories and files on a server that are for remote clients.
/home	Serves as a mount point for a file system containing user home directories. The /home file system contains per-user files and directories. In a standalone machine, a separate local file system is mounted over the /homedirectory. In a network, a server might contain user files that should be accessible from several machines. In this case, the server's copy of the /home directory is remotely mounted onto a local /home file system.
/lib	Symbolic link to the /usr/lib directory, which contains architecture-independent libraries with names in the form lib*.a.
/sbin	Contains files needed to boot the machine and mount the /usr file system. Most of the commands used during booting come from the boot image's RAM disk file system; therefore, very few commands reside in the /sbin directory.
/tmp	Serves as a mount point for a file system that contains system-generated temporary files.
/u	Symbolic link to the /home directory.
/usr	Serves as a mount point for a file system containing files that do not change and can be shared by machines (such as executable programs and ASCII documentation). Standalone machines mount a separate local file system over the /usr directory. Diskless and disk-poor machines mount a directory from a remote server over the /usr file system.
/var	Serves as a mount point for files that vary on each machine. The /var file system is configured as a file system because the files that it contains tend to grow. For example, it is a symbolic link to the /usr/tmp directory, which contains temporary work files.

10.4. Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss the trade-offs involved in choosing one of these algorithms.

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency performance, and reliability of the file system.\

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- This method is simple to program but time-consuming to execute.
  - To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.
  - To delete a file, we search the directory for the named file, then release the space allocated to it.
- The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.
- Directory information is used frequently, and users will notice if access to it is slow. In fact, many OSs implement a software cache to store the most recently used directory information.
- A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory.

Hash Table

- Another data structure used for a file directory is a **hash table.**
- With this method, a linear list stores the directory entries, but a hash data structure is also used
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time.

10.5. Allocation Methods

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems (such as Data General's RDOS for its Nova line of computers) support all three. More commonly, a system uses one method for all files within a file system type.

- The direct-access nature of disks allows us flexibility in the implementation of files.
- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- Three major methods of allocating disk space are in wide use: **contiguous, linked,** and **indexed.**

Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b + 1 after block b normally requires no head movement.

When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed. The IBM VM/CMS operating system uses contiguous allocation because it provides such good performance.

10.6 Free space management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically possible.) To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we discuss next.

Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a free-space list which records all disk blocks that are free.

- Free-space list can be implemented as:

Bit-Vector:

Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated ,the bitis1.

Linked-List:

This approach link all the free disk blocks together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on.

Grouping:

This approach stores the addresses of n free blocks in the first free block. The first n-1 of these are actually free. The last one is the disk address of another block containing addresses of other

'n' free blocks. The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

Counting:

This approach takes advantage of the fact that several contiguous blocks may be allocated or freed simultaneously. Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number n of free contiguous blocks that follow the first block.

10.7 Summary

The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk.

Physical disks may be segmented into partitions to control media use and to allow multiple, possibly varying, file systems on a single spindle. These file systems are mounted onto a logical file system architecture to make them available for use. File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices. Upper levels deal with symbolic file names and logical properties of files. Intermediate levels map the logical file concepts into physical device properties.

10.8 Keywords

Hash Table, File Structure, Mounting, Counting.

10.9 Exercises

1. What is linked allocation?
2. Describe free space management
3. Briefly explain file structure
4. Explain file system mounting

10.10 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-11:

Structure

11.0 Objectives

11.1 Introduction

11.2 Protection and security

11.3 Goals of Protection

11.4 Domain protection

11.5 Implementation of domain

11.6 Access matrix

11.7 Implementation of access matrix

11.8 Revoking access rights

11.9 Keywords

11.10 Exercise

11.11 Reference

11.0Objectives

After going through this unit, you will be able to: Describe Protection

- Describe security, their importance
- Define access matrix and implementation
- Describe revoking access rights

11.1Introduction

Computer security is a branch of computer technology known as information security as applied to computers and networks. The objective of computer security includes protection of information and property from theft, corruption, or natural disaster, while allowing the information and property to remain accessible and productive to its intended users. The term computer system security means the collective processes and mechanisms by which sensitive and valuable information and services are protected from publication, tampering or collapse by unauthorized activities or untrustworthy individuals and unplanned events respectively. The strategies and methodologies of computer security often differ from most other computer technologies because of its somewhat elusive objective of preventing unwanted computer behavior instead of enabling wanted computer behavior.

11.2Protection and Security

The operating system is (in part) responsible for protection – to ensure that each object (hardware or software) is accessed correctly and only by those processes that are allowed to access it.

We have seen protection in a few contexts:

- Memory protection
- file protection
- CPU protection

We want to protect against both malicious and unintentional problems.Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.

Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls I to be imposed, together with a means of enforcing them.

Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.

Protection

- **File Protection.** It is up to the OS to manage the system security so that files are only accessible to authorized users. Files in UNIX are protected by assigning each one a 9-bit binary protection code.
    - Three bit fields, one for owner, one for other members of the owner's group, and one for everyone else.
    - Each field has a bit for **read** access, a bit for **write** access, and a bit for **execute** access.

      $(d)rwxr-x--x$
    -
- **OS Protection**. OS must protect itself from users; reserved memory only accessible by OS. The OS is responsible for allocating access to memory space and CPU time and peripherals etc., and it will control dedicated hardware facilities:

- The memory controller, control to detect and prevent unauthorized access.
  - A timer will also be under OS control to manage CPU time allocation to programs competing for resources.
- **User Protection**. OS may protect users from another user. A fundamental requirement of multiple users of a shared computer system is that they do not interfere with each other. This gives rise to the need for separation of the programs in terms of their resource use and access:
  - If one program attempts to access main memory allocated to some other program, the access should be denied and an exception raised.
  - If one program attempts to gain a larger proportion of the shared CPU time, this should be prevented.
- One approach to implementing resource allocation is to have at least two modes of CPU operation

Security

How can we define a system to be secure?

One possibility: all resources are used only exactly as intended

A secure system should not allow:

- Unauthorized reading of information
- Unauthorized modification of information
- Unauthorized destruction of data
- Unauthorized use of resources
- Denial of service for authorized uses

11.3 Goals of Protection

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of

any complex system that makes use of shared resources.

We need to provide protection for several reasons. The most obvious is the need to prevent mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each program component active in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system.

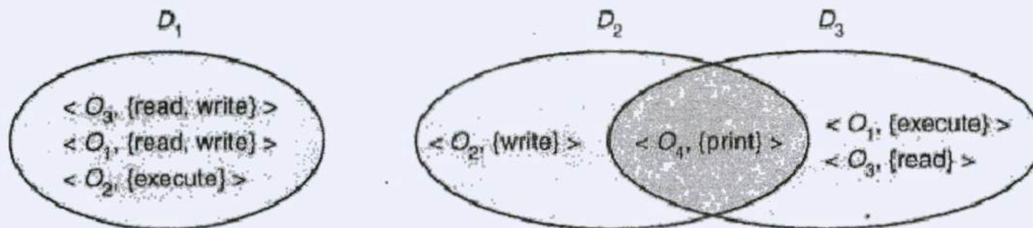
- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system policies, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the mechanisms for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

11.4 Domain of Protection

A computer can be viewed as a collection of processes and objects (both HW & SW).
The need to know principle states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
The modes available for a particular object may depend upon its type.

Domain Structure

- A protection domain specifies the resources that a process may access.



- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An access right is the ability to execute an operation on an object.
- A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.

- The association between a process and a domain may be static or dynamic.
  - ❖ If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
  - ❖ If the association is dynamic, then there needs to be a mechanism for domain switching.
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

11.5.Acess Matrix

According to the model, the protection state of a computer system can be abstracted as a set of objects O, that is the set of entities that needs to be protected (e.g. processes, files, memory pages) and a set of subjects S, that consists of all active entities (e.g. users, processes). Further there exists a set of rights R of the form $r(s,o)$, where $s \in S$, $o \in O$ and $r(s,o) \subseteq R$. A right thereby specifies the kind of access a subject is allowed to process object.

In this matrix example there exists two processes, a file and a device. The first process has the ability to execute the second, read the file and write some information to the device, while the second process can only send information to the first.

	Acess 1	Access 2	File	Device
Role 1	Read, write,execute,own	execute	write	read
Role 2	read	Read write execute own		

11.6 Implementation to access matrix

• Each column = Access-control list for one object

Defines who can perform what operation.

Domain 1 = Read, Write

Domain 2 = Read

Domain 3 = Read

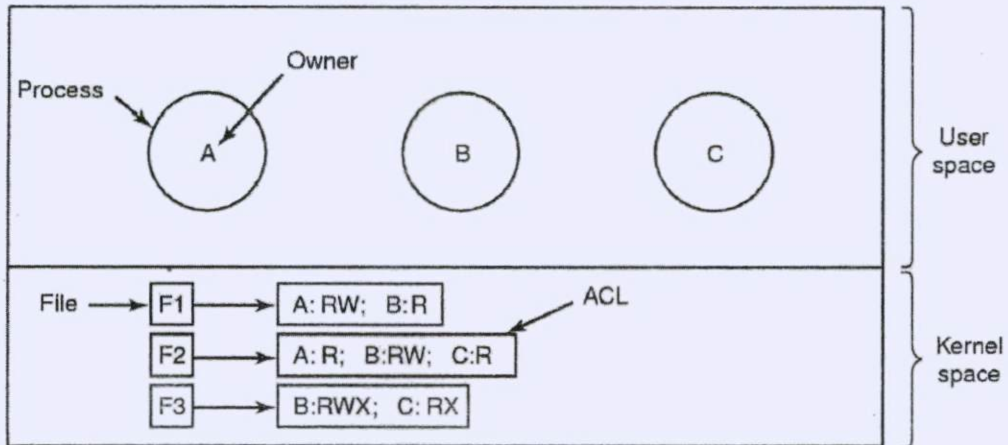
• Each Row = Capability List (like a key)

Fore each domain, what operations allowed on what objects.

Object 1 – Read

Object 4 – Read, Write, Execute

Object 5 – Read, Write, Delete, Copy



11.7. Revoking Access Rights

- Simple with access list if we revoke by object
- More difficult with capabilities. Use
- Reacquisition
- Back-pointers
- Indirections (not selective)
- Keys (not selective if one key per object)

Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.
- Language implementation can provide software for protection enforcement when automatic hardware supported checking is unavailable.
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

11.8 Summary

Computer systems contain many objects, and they need to be protected from misuse. Objects may be hardware (such as memory, CPU time, and I/O devices) or software (such as files, programs, and 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 and may use any of the access rights in the domain to access and manipulate objects. During its lifetime, a process may be either bound to a protection domain or allowed to switch from one domain to another.

The access matrix is a general model of protection that provides a mechanism for protection without imposing a particular protection policy on the system or its users. The separation of policy and mechanism is an important design property.

11.9 Keywords

Operating System, Access Matrix, Security, Protection

11.10 Exercises

1. Define Protection and security?
2. Explain access matrix?
3. Disscuss implementation of access matrix

11.11 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-12:

Structure

12.0 Objectives

12.1 Introduction

12.2 Security Problems

12.3 Authentication

12.4 Program threads and system threads

12.5 Intrusion Detection

12.6. Cryptography

12.7 Unit Summary

12.8 Keywords

12.9 Exercises

12.10 References

12.0 Objectives

After going through this unit, you will be able to: Describe Computer security

- Describe system threads and program threads
- Define cryptography.
- Explain problems in security

12.1 Introduction

How do we provide controlled access to programs and data stored in a computer system? **Security**, on the other hand, requires not only an adequate protection system but also consideration of the *external* environment within which the system operates. A protection system is ineffective if user authentication is compromised or a program is run by an unauthorized user.

Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. These resources include information stored in the system (both data and code), as well as the CPU, memory, disks, tapes and networking that are the computer. In this chapter, we start by examining ways in which resources may be accidentally or purposefully misused. We then explore a key security enabler — cryptography. Finally, we look at mechanisms to guard against or detect attacks.

12.2 Security Problems

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function.

In Pervious Chapter, we discussed mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources. We say that a system is secure if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Protecting data against unauthorized use is a major concern for operating systems designers and administrators.

- Data confidentiality – prevent unauthorized access
- Data integrity – prevent unauthorized tampering
- System availability – prevent denial of service

Issues to consider (at design stage)

- Casual users browsing information that's not for them
- Snooping by skilled insiders
- Organized and determined attempts to make money
- Espionage etc.

System "bloat" is definitely bad for security

Accidental data loss probably more common than determined security evasion

12.3 Authentication

The discussion of authentication above involves messages and sessions. But what of users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is user authentication. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. A user normally identifies herself. How do we determine whether a user's identity is authentic? Generally, user authentication is based on one or more of three things: the user's possession of

something (a key or card), the user's knowledge of something (a user identifier and password), and/or an attribute of the user (fingerprint, retina pattern, or signature).

The most common approach to authenticating a user identity is the use of passwords. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Some simple problems

– Should you be able to see the number of letters in a password?

– When will authentication fail (id/password)?

– Easy to guess passwords (e.g. "baby names books")

– But people won't try my computer…

• War dialers / IP scanning

– Start off by collecting machines that accept logins

• Dialing, systematic scanning of ip addresses in ic.ac.* etc

– Then systematically try logins and passwords

– Boring? That's what computers are good at.

Unix Passwords

One-way-function **f** is applied to password at login

• "Encrypted", i.e. f(p), passwords stored on disk, in older systems usually in publically readable file

• No-one can tell a user what their password is

• Problem(s):

– What if 2 people choose the same password?

– What if someone chooses "hello" or "password"?

• To guard against pre-computed password dictionaries:

– Add salt…

– Password entry goes from f("doggie") -> f("doggie1234")

– Need to store the random number unencrypted

– f("doggie1234") and f("doggie5678") no obvious relation

• Still possible to copy actual password files and search

– Make password file unreadable

One Time Passwords

• Sequence of passwords, each used only once

– What is the idea behind this?

– Do not lose the book where the sequence is written down

• Better plan (Lamport 1981):

– One-way function f

– Suppose we need 4 passwords: start with f(f(f(f(p)))), then f(f(f(p))) etc

– The point is that although the previous password is very easy to calculate from the current one, the next one is impossible / very hard to computer.

Authentication Using Physical Objects

An example we all know: ATM card plus PIN to show that it is us who are using the card

– Magnetic strip: stores about 140 bytes, cost $0.10 - $0.50

– Real problems if this stores the PIN

• Smart cards

– Small CPU (maybe 4MHz, 8-bit, some RAM and ROM), small EEPROM (memory that doesn't need power to keep its value)

– Cost more like $5…

– Challenge-response authentication

• Server sends random string

• Smart card adds user's password, encrypts, sends back part

• Server does the same and compares

– Even better to run a small Java VM on the card – allows substituting the encryption algorithm on the fly.

Biometric Authentication

Authentication via palm- / finger-print reader

– Some concerns regarding use in criminal cases

• Alternative: Iris recognition tools.

Need to make sure there is a live person there!

12.4. Program threads and system threads

Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

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). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Processes Vs Threads

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within processes, threads within processes execute sequentially.

- Like processes, thread can create children.
- And like process, 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.

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 a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are

- A user-level thread does 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 more 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.

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 Systems kernel provides system call to create and manage threads.

The implementation of general structure of kernel-level thread is

<DIAGRAM>

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. For instance, threads operations are hundreds of times slower than that of user-level threads.
- 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 increased in kernel complexity.

12.5. Intrusion Detection

An **intrusion detection system (IDS)** is a device or software application that monitors network or system activities for malicious activities or policy violations and produces reports to a Management Station. Some systems may attempt to stop an intrusion attempt but this is neither required nor expected of a monitoring system.[1] Intrusion detection and prevention systems (IDPS) are primarily focused on identifying possible incidents, logging information about them, and reporting attempts.[1] In addition, organizations use IDPSes for other purposes, such as identifying problems with security policies, documenting existing threats, and deterring individuals from violating security policies.[1] IDPSes have become a necessary addition to the security infrastructure of nearly every organization.[1]

IDPSes typically record information related to observed events, notify security administrators of important observed events, and produce reports.[1] Many IDPSes can also respond to a detected threat by attempting to prevent it from succeeding.[1] They use several response techniques, which involve the IDPS stopping the attack itself, changing the security environment (e.g., reconfiguring a firewall), or changing the attack's content.[1]

Types

For the purpose of dealing with IT, there are three main types of IDS:

Network intrusion detection system (NIDS) is an independent platform that identifies intrusions by examining network traffic and monitors multiple hosts, developed in 1986 by Pete R. Network intrusion detection systems gain access to network traffic by connecting to a network hub, network switch configured for port mirroring, or network tap. In a NIDS, sensors are located at choke points in the network to be monitored, often in the demilitarized zone (DMZ) or

at network borders. A sensor captures all network traffic and analyzes the content of individual packets for malicious traffic. An example of a NIDS is Snort.

Host-based intrusion detection system (HIDS)

It consists of an agent on a host that identifies intrusions by analyzing system calls, application logs, file-system modifications (binaries, password files, capability databases, Access control lists, etc.) and other host activities and state. In a HIDS, sensors usually consist of a software agent. Some application-based IDS are also part of this category. Examples of HIDS are Tripwire andOSSEC.

Stack-based intrusion detection system (SIDS)

This type of system consists of an evolution to the HIDS systems. The packets are examined as they go through the TCP/IP stack and, therefore, it is not necessary for them to work with the network interface in promiscuous mode. This fact makes its implementation to be dependent on the Operating System that is being used. Intrusion detection systems can also be system-specific using custom tools and honey pots

12.6. Cryptography

The art of protecting information by transforming it (*encrypting* it) into an unreadable format, called cipher text. Only those who possess a secret *key* can decipher (or *decrypt*) the message into plain text. Encrypted messages can sometimes be broken by cryptanalysis, also called *code breaking*, although modern cryptography techniques are virtually unbreakable.

As the Internet and other forms of electronic communication become more prevalent, electronic security is becoming increasingly important. Cryptography is used to protect e-mail messages, credit card information, and corporate data. One of the most popular cryptography systems used on the Internet is *Pretty Good Privacy* because it's effective and free. Cryptography systems can be broadly classified into symmetric-key systems that use a single key that both the sender and recipient have, and *public-key* systems that use two keys, a public key known to everyone and a private key that only the recipient of messages uses. Cryptographic

transformations are a fundamental building block in many security applications and protocols. To improve performance, several vendors market hardware accelerator cards. However, until now no operating system provided a mechanism that allowed both uniform and efficient use of this new type of resource. We present the Open BSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the operating system kernel that provides uniform access to accelerator functionality by hiding card-specific details behind a carefully designed API. We evaluate the impact of the OCF in a variety of benchmarks, measuring overall system performance, application throughput and latency, and aggregate throughput when multiple applications make use of it. We conclude that the OCF is extremely efficient in utilizing cryptographic accelerator functionality, attaining 95% of the theoretical peak device performance and over 800 Mbps aggregate throughput using 3DES. We believe that this validates our decision to opt for ease of use by applications and kernel components through a uniform API and for seamless support for new accelerators. Furthermore, our evaluation points to several bottlenecks in system and operating system design: data copying between user and kernel modes, PCI bus signaling inefficiency, protocols that use small data units, and single-threaded applications. We identify some of these limitations through a set of measurements focusing on application-layer cryptographic protocols such as SSL. We offer several suggestions for improvements and directions for future work. We provide experimental evidence of the effectiveness of a new approach which we call *operating system shortcutting*. Shortcutting can improve the performance of application-layer cryptographic protocols by 27% with very small changes to the kernel.

12.7 Summary

Protection is an internal problem. Security, in contrast, must consider both the computer system and the environment—people, buildings, businesses, valuable objects, and threats—within which the system is used.

The data stored in the computer system must be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data.

Absolute protection of the information stored in a computer system from malicious abuse is not possible; but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority.

12.8 Keywords

Cryptography, System threads, Program threads, Security

12.9 Exercises

1. Describe system security.

2. What are program threads?

3. What are system threads?

4. Write about cryptography.

12.10 References

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT 13

Structure

13.0 Objectives

13.1 Introduction

13.2 History

13.3 Design Principles

13.4 General Purpose Utilities

13.5 Navigating the file system

13.6 Handling Ordinary Files

13.7 Editors

13.8 Summary

13.9 Keywords

13.10 Exercises

13.11 References

13.0 Objectives

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

- Explain history of Unix Operating System
- Describe Unix system design principles
- Describe the handling files in Unix

13.1 Introduction

All operating systems provide services for programs they run. Typical services include executing a new program, opening a file, reading a file, allocating a region of memory, getting the current time of day, and so on. The focus of this text is to describe the services provided by various versions of the UNIX operating system.

Describing the UNIX System in a strictly linear fashion, without any forward references to terms that haven't been described yet, is nearly impossible (and would probably be boring). This chapter provides a whirlwind tour of the UNIX System from a programmer's perspective. We'll give some brief descriptions and examples of terms and concepts that appear throughout the text. We describe these features in much more detail in later chapters. This chapter also provides an introduction and overview of the services provided by the UNIX System, for programmers new to this environment.

13.2 History

There are numerous events in the computer industry that have occurred since UNIX started life as a small project in Bell Labs in 1969. UNIX history has been\largely influenced by Bell Labs' Research Editions of UNIX, AT&T's System V UNIX, Berkeley's Software Distribution (BSD), and Sun Microsystems' SunOS and Solaris operating systems. The following list shows the major events that have happened throughout the history of UNIX.

1969. Development on UNIX starts in AT&T's Bell Labs.

1971. 1st Edition UNIX is released.

1973. 4th Edition UNIX is released. This is the first version of UNIX that had the kernel written in C.

1974. Ken Thompson and Dennis Ritchie publish their classic paper, "The UNIX Timesharing System" [RITC74].

1975. 6th Edition, also called V6 UNIX, becomes the first version of UNIX to be used outside Bell Labs. The University of California at Berkeley starts development on the *Berkeley Software Distribution* or more commonly called BSD.

1977. At this stage there were 500 sites running UNIX. Universities accounted for about 20 percent of those sites.

1979. 7th Edition UNIX was rewritten to make it more portable. Microsoft licenses 7th Edition and starts development of Xenix.

1980. Microsoft releases Xenix, a PC-based version of UNIX.

1982. AT&T's UNIX Systems Group releases System III UNIX. The Santa Cruz Operation (SCO) licenses Xenix from Microsoft.

1983. AT&T's UNIX System Development Labs release System V Release 1 UNIX.

1984. 4.2BSD is released including TCP/IP. System V Release 2 is released and the number of installations of UNIX worldwide exceeds 100,000. Digital Equipment Corporation's (DEC's) 4.2BSD-based Ultrix is released.

1986. 4.3BSD is released. 4.2BSD-based HP-UX first appears. IBM releases AIX 2 for the RT server.

1987. AT&T releases System V Release 3, which includes STREAMS, the Network File System (NFS), and the Transport Level Interface (TLI).

1989. As a joint venture between AT&T's Unix System Laboratories (USL) and Sun Microsystems, System V Release 4.0 is released.

1990. Based on SVR2 with enhancements from 4.2BSD and 4.3BSD, IBM releases AIX 3.1.

1991. Linus Torvalds announces Linux 0.0.1.

1992. USL releases System V Release 4.2 that includes the VERITAS file system VxFS and Volume Manager VxVM.

1993. 4.4BSD, the last release from Berkeley, is released. SVR4.2MP is released by Novell following their purchase of USL from AT&T.

1994. 4.4BSD Lite, which was free of copyrighted UNIX source code, is released.

1995. SCO buys Novell's UNIX business.

1996. The Linux 2.0 kernel is released.

1997. UnixWare 7, a merge of SVR4.2MP and SCO OpenServer, is released.

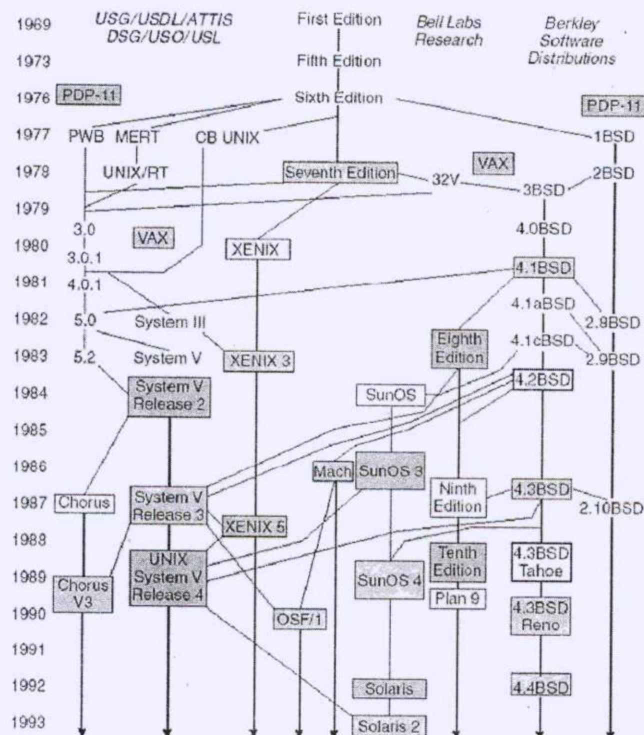
2001. SCO's UNIX business is sold to Linux distributor Caldera. The Linux 2.4 kernel emerges after many delays.

2003. The SCO Group started legal action against various users and vendors of Linux. SCO had alleged that Linux contained copyrighted Unix code now owned by The SCO Group.

2005. Sun Microsystems released the bulk of its Solaris system code (based on UNIX System V Release 4) into an open source project called OpenSolaris.

2007. A major portion of the case was decided in Novell's favor. (that Novell had the copyright to UNIX, and that the SCO Group had improperly kept money that was due to Novell).

2010. Following a jury trial, Novell, and not The SCO Group, was "unanimously [found]" to be the owner of the UNIX and UnixWare copyrights.



13.3 Design Principles

- Designed to be a time-sharing system
- Has a simple standard user interface (shell) that can be replaced
- File system with multilevel tree-structured directories
- Files are supported by the kernel as unstructured sequences of bytes
- Supports multiple processes; a process can easily create new processes

- High priority given to making system interactive, and providing facilities for program development

13.4General Purpose

The general-purpose utilities of the system can be broadly divided into two categories:

1.Some commands tell you the state of the system.

2.Others can aid you directly in your work.

The passwd command to change your password. When used without arguments, it sets the user's own password:

passwd: Change Your Password

The passwd command to change your password. When used without arguments, it sets the user's own password:

$ passwd

Changing password for gates

(current) UNIX password:

Enter new UNIX password:

Retype new UNIX password:

- When invoked by an ordinary user, passwd asks for the old password, then it demands the new password twice
- If everything goes smoothly, the new password is registered by the system, and the prompt is returned.

- Depending on the way they are configured, many systems conduct certain checks on the string that you enter as password.
- They may either disallow you from framing easy-to-remember passwords or advise you against choosing a bad password.
- These are good practice when handling your own password:
    - Don't choose a password similar to your old one.
    - Don't use commonly used names.
    - Use a mix of alphabetic or numeric characters.
    - Make sure the password is unmeaningful enough to prevent other from guessing it.
    - Don't write down the password in an easily accessible document.
    - Change the password regularly.
- When you enter a password, the string is **encrypted** by the system.
- Encryption generates a string of seemingly random characters that UNIX uses subsequently to determine the authenticity of a password.
- This encryption is stored in a file named shadow in the /etc directory.

who and w: Know the Users

- UNIX maintains an account of all users who are logged on to the system.
- There are two commands which display an informative listing of users – who and w.
- Who produces a simple three or four-columnar output.

```
$ who

A  pts/2   Aug 31 12:00 (sisko.cs.twsu.edu)

B  pts/5   Sep  2  15:10 (156.26.169.34)
```

- The first column show the user-ids of the users currently working on the system.
- The second column shows the device names of their respective terminals. These terminals are actually special files representing devices.
- The third column shows the date and time of logging in.
- The fourth column show the location (domain name or IP address) of logging in.
- The option –H prints line of column headings.

- The option –u adds user idle time.

```
$ who –Hu

USER LINE  LOGIN-TIME  IDLE FROM

A     pts/3   Aug 31 12:57   00:40  (kira.cs.twsu.edu)

B     pts/5   Sep  2 15:10     .      (156.26.169.34)
```

- shows A has activity in the last one minute. B seems to be idling for the last 40 minutes.
- The w command produces a more detailed output of users' activities, and additionally displays many details of the system:

```
$ w

4:46pm  up 124 days, 19:10,  5 users,  load average: 0.00, 0.00, 0.00

USER   TTY    FROM             LOGIN@  IDLE   JCPU   PCPU   WHAT

A    pts/1   ge.cs.twsu.edu      3:02pm   40:45   0.04s   0.04s   -bash

B    pts/2   alonzo.cs.twsu.e   Fri12pm   2days   2.03s   1.98s   pine
```

- The first line of output shows the command was executed at 4:46 p.m. on a system having two users.
- The system has been up and running for 124 days and 19 hours and 10 minutes.
- The system load averages for the past one, five and fifteen minutes are virtually negligible.
- The remaining output is who-like except for the last three columns.
- The output shown under JCPU shows the total CPU time consumed by all processes at that terminal.
- The command the user is currently executing is shown in the last column.
- PCPU refers to the time consumed by that process.
- Because the multitasking nature of UNIX permits a user to run more than one job at the same time, the JCPU and PCPU times can often be different.
- who and w are regularly used by the system administrator to monitor whether terminals are being properly utilized.

tty: Know Your Terminal

The tty command tells you the filename of the terminal you are using.

$ tty

/dev/pts/6

- This shows the user is using the number 6 of pts terminal.
- pts is known as pseudo terminal system.
- The lock command lets you lock your terminal. But it is not available in Linux.

stty: Set Terminal Characteristics

- The terminal is the device with which a user communicates with the system.
- Different terminals are configured differently. It's possible that your terminal may not behave in the way it's meant to.
- The stty command helps straighten these things out.
- stty uses an enormous number of keywords.
- The –a (all) option displays the current settings.

$ stty -a

speed 9600 baud; rows 25; columns 80; line = 0;

intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;

eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;

lnext = ^V; flush = ^O; min = 1; time = 0;

-parenb -parodd cs8 -hupcl -cstopbcread -clocal -crtscts

-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncricrnlixon -ixoff

-iuclc -ixany -imaxbel

opost -olcuc -ocrnlonlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0

isigicanoniexten echo echoeechok -echonl -noflsh -xcase -tostop -echoprt

echoctlechoke

- The output shows the baud rate (the speed) of the terminal is 9600.
- In this system, the [Ctrl-c] key interrupts a program. The erase character is [Ctrl-?], and the kill character is [Ctrl-u].
- Of special significance is the eof (end-of-file) character which is set to [Ctrl-d]. You can use this key with the cat command to end input from the keyboard.
- A series of keywords with a – prefixed are options which are turned off. The ones without a – prefixed are options which are turned on.
- echoe decides whether backspacing should erase character. If it is set (without – prefix to it), backspacing removes the character from display.
- This setting can be reversed in this way:

stty -echoe

- The echo option decides if the input is echoed. By default, the option is turned on. You can turn it off and on as follows respectively:

stty –echo

stty echo

- If you like to use use[Ctrl-c] as the interrupt key instead of [Delete], you can use

sttyintr \^c

- When you insert control characters into a file, you'll see a ^ symbol prefixed to the character. For example, [Ctrl-l] is seen as ^l. It's actually a single character occupying two slots on the terminal.
- When creating files with cat, you can use [Ctrl-d] to terminate input. Instead of [Ctrl-d], you can use [Ctrl-a] as the eof character:

sttyeof \^a

- To set the terminal characteristics to values that will work on most terminals, use the following command:

stty sane

13.5Navigating the file system

The Unix File System

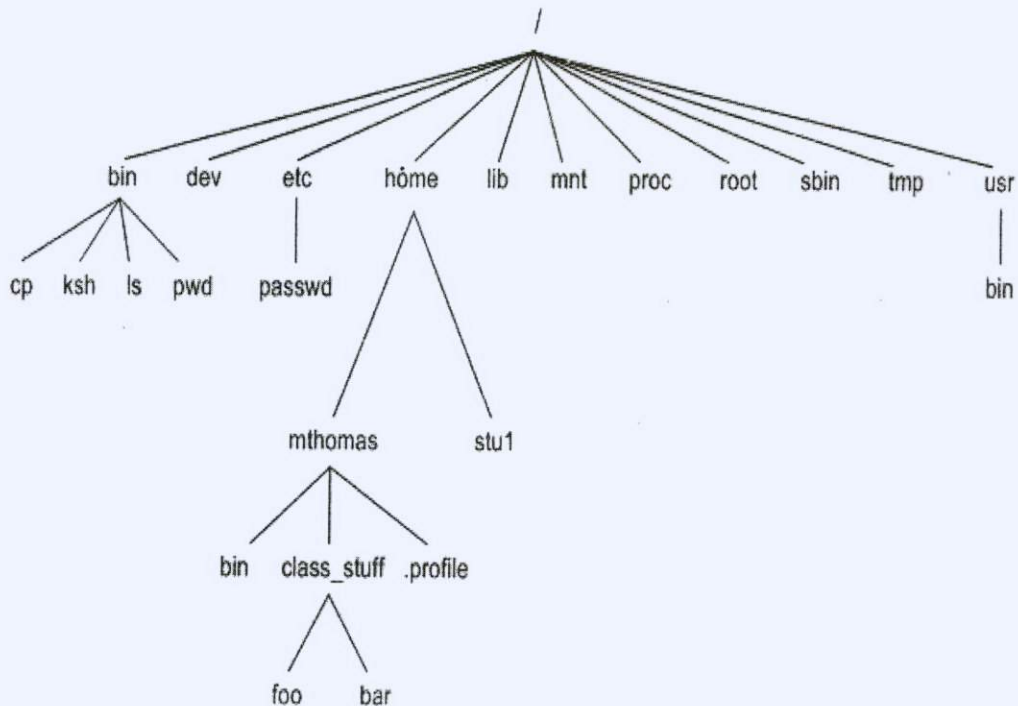
The Unix file system is a methodology for logically organizing and storing large quantities of data such that the system is easy to manage. A **file** can be informally defined as a collection of (typically related) data, which can be logically viewed as a stream of bytes (i.e. characters). A file is the smallest unit of storage in the Unix file system. By contrast, a **file system** consists of files, relationships to other files, as well as the attributes of each file. File attributes are information relating to the file, but do not include the data contained within a file. File attributes for a generic operating system might include (but are not limited to): a file type (i.e. what kind of data is in the file)

- a file name (which may or may not include an extension)
- a physical file size
- a file owner file protection/privacy capability
- file time stamp (time and date created/modified)

Additionally, file systems provide tools which allow the manipulation of files, provide a logical organization as well as provide services which map the logical organization of files to physical devices.From the beginner's perspective, the Unix file system is essentially composed of files and **directories**. Directories are special files that may contain other files.

The Unix file system has a hierarchical (or tree-like) structure with its highest level directory called root (denoted by /, pronounced *slash*). Immediately below the root level directory are several subdirectories, most of which contain system files. Below this can exist system files, application files, and/or user data files. Similar to the concept of the process parent-child relationship, all files on a Unix system are related to one another. That is, files also have a parent-child existence. Thus, all files (except one) share a common parental link, the top-most file (i.e. /) being the exception.

Below is a diagram (slice) of a "typical" Unix file system. As you can see, the top-most directory is / (slash), with the directories directly beneath being system directories. Note that as Uniximplementaions and vendors vary, so will this file system hierarchy. However, the organization of most file systems is similar.



While this diagram is not all inclusive, the following system files (i.e. directories) are present in most Unixfilesystems:

- *bin* - short for binaries, this is the directory where many commonly used executable commands reside
- *dev* - contains device specific files
- *etc* - contains system configuration files
- *home* - contains user directories and files
- *lib* - contains all library files
- *mnt* - contains device files related to mounted devices
- *proc* - contains files related to system processes
- *root* - the root users' home directory (note this is different than /)
- *sbin* - system binary files reside here. If there is no sbin directory on your system, these files most likely reside in etc
- *tmp* - storage for temporary files which are periodically removed from the file system
- *usr* - also contains executable commands

File Types

From a user perspective in a Unix system, everything is treated as a file. Even such devices such as printers and disk drives. Since all data is essentially a stream of bytes, each device can be viewed logically as a file. All files in the Unix file system can be loosely categorized into 4 types, specifically:

1. ordinary files
2. directory files
3. device files
4. Link files

Ordinary files are comprised of streams of data (bytes) stored on some physical device. Examples of ordinary files include simple text files, application data files, files containing high-level source code, executable text files, and binary image files. Note that unlike some other OS implementations, files do not have to be binary Images to be executable (more on this to come).

The second type of file listed above is a special file called a directory .Directory files act as a container for other files, of any category. Thus we can have a directory file contained within a directory file (this is commonly referred to as a subdirectory). Directory files don't contain data in the user sense of data, they merely contain references to the files contained within them. It is perhaps noteworthy at this point to mention that any "file" that has files directly below (contained within) it in the hierarchy **must** be a directory, and any "file" that does not have files below it in the hierarchy can be an ordinary file, or a directory, albeit empty.

The third category of file mentioned above is a device file. This is another special file that is used to describe a physical device, such as a printer or a zip drive. This file contains no data whatsoever; it merely maps any data coming its way to the physical device it describes. Device file types typically include: character device files, block device files, Unix domain sockets, named pipes and symoblic links. However, not all of these file types may be present across various Unix implementations.

Link file is a pointer to another file. There are two types of links - a hard link to a file is indistinguishable from the file itself. A soft link(or symbolic link) provides an indirect pointer or shortcut to a file. A soft link is implemented as a directory file entry containing a pathname

File System Navigation

To begin our discussion of navigating or moving around the file system, the concept of **file names** must be introduced. It is, after all, the name of a file that allows for its manipulation. In simplest terms, a file name is a named descriptor for a file. It must consist of one or more characters (with the maximum of 255), where the characters can be almost any character that can be interpreted by the shell (except for / since this has a special meaning). However it is strongly advised to use file names that are descriptive of the function of the file in question.

By rule, Unix file names do not have to have ending extensions (such as .txt or .exe) as do some other operating systems. However, certain applications with which we interact may require extensions, such as Adobe's Acrobat Reader (.pdf) or a web browser (.html). And as always character case matters .Thus the following are all valid Unix file names (note these may be any file type):

My_Stuff	a file called My_Stuff
my_stuff	a different file than above
mortgage.c	a C language program...can you guess what it does?
a.out	a C language binary executable
.profile	ksh default startup file

While file names are certainly important, there is another important related concept, and that is the concept of a **file specification**(or file spec for short). A file spec may simply consist of a file name, or it might also include more information about a file, such as where is resides in the overall file system. There are 2 techniques for describing file specifications, absolute and relative.

With **absolute** file specifications, the file specification always begins from the root directory, complete and unambiguous. Thus, absolute file specs always begin with /. For example, the following are all absolute file specs from the diagram above:

/etc/passwd

/bin

/usr/bin

/home/mthomas/bin

/home/mthomas/class_stuff/foo

Note the first slash indicates the top of the tree (root), but each succeeding slash in the file spec acts merely as a separator. Also note the files named bin in the file specifications of /bin, /usr/bin, and /home/mthomas/bin are different bin files, due to the differing locations in the file system hierarchy.

With **relative** file specifications, the file specification always is related to the users current position or location in the file system. Thus, the beginning (left-most part) of a relative file spec describes either:

- an ordinary file, which implies the file is contained within the current directory
- a directory, which implies a child of the current directory (i.e. one level down)
- a reference to the parent of the current directory (i.e. one level up)

What this means then is that a relative file specification that is valid from one file system position is probably not valid from another location. Beginning users often ask "How do I know where I am?" The command to use to find this is the **pwd** (**p**rint **w**orking **d**irectory) command, which will indicate the users current position (in absolute form) in the file system.

As mentioned above, part of a relative file specification can be a reference to a parent directory. The way one references a parent (of the current) directory is with the characters **..** (pronounced dot dot). These characters (with no separating spaces) describe the parent directory relative to the current directory, again, one directory level up in the file system.

The following are examples referencing the diagram above:

To identify where we are, we type and the system returns the following:

$pwd*[Enter]*

/home/mthomas/class_stuff

Thus the parent of this directory is:

/home/mthomas # in absolute form

.. # in relative form

Looking at another example:

$pwd*[Enter]*

/home/mthomas

Thus the parent of this directory is:

/home # in absolute form

.. # in relative form

And one (note there could be many) child of the /home/mthomas directory is:

/home/mthomas/bin # in absolute form

bin # in relative form

So you ask "How the heck do we use this?" One uses this to navigate or move about the file system. Moving about the file system is accomplished by using the **cd** command, which allows a user to **c**hange **d**irectories. In the simplest usage of this command, entering

$cd*[Enter]*

will move the user to their "home" or login directory. If the user wishes to change to another directory, the user enters

$cdfile_spec*[Enter]*

and assuming file_spec is a valid directory, the users current working directory will now be this directory location. Remember, the file specification can always be a relative or an absolute specification.

As before, we type and the system returns the following:

```
$pwd[Enter]
/home/mthomas/class_stuff
```

If we wish to change directories to the /home/mthomas/bin directory, we can type

```
$cd /home/mthomas/bin [Enter]          # absolute, will work from anywhere
```

or

```
$cd ..[Enter]          # relative, move up one directory, i.e. to the parent
$cd bin [Enter]        # relative, move down to the bin directory
```

or

```
$cd ../bin [Enter]  # relative, both steps in one file spec
```

Novice users sometimes ask which file specification method should they use, or which one is better. The simple and open ended answer is "it depends." That is it depends upon how long the file specification is; or how easy it is to type, including any special characters; or how familiar one is with the current location in the file system hierarchy, etc.

13.6 Handling Ordinary files

Users actually do most of their work with ordinary files and its natural that the unix system should feature a host of commands to handle them. Some of the commands and their functionalities are given below.

- View text files with **cat** and **more**(or **less**)
- Use **cat** to create a file.
- The essential file functions- copy with **cp** , remove with **rm** an rename with **mv**
- Print a file with **lp**
- Classify files with **file**
- Count the number of lines, words and characters with **wc**
- Display the ASCII octal value of text with **od**
- Compare two files with **cmp, comm, diff.**
- Compress and decompress files with **gzip** and **gunzip**.
- Create an archive comprising multiple files with **tar**.
- Perform both functions(compressing and archiving) with **zip** and **unzip**

13.7 Editors

Unix text editors are quite unlike the word processors found on Windows and Macintosh systems. They do not support WYSIWYG editing, auto-formatting, line-wrapping, multiple fonts, or multiple font sizes. Instead they are oriented toward plain text.

On the plus side, Unix text editors have lots of features, such as auto-indentation and syntax highlighting, that make them idea for writing scripts, programs, and HTML pages.

Text Editors Available

vi Non-graphical (terminal-based) editor. Guaranteed to be available on any system.Requires knowledge of arcane keystroke commands.Distinctly unfriendly to novices.

emacs

Window-based editor. Menus make it friendlier to novices, but you still need to know keystroke commands to use many of the advanced functions. Installed on all Linux distributions and on most other UNIX systems.

pico

Simple terminal-based editor available on most versions of Unix. Uses keystroke commands, but they are listed in logical fashion at bottom of screen.

13.8 Summary

Unix (officially trademarked as **UNIX**, sometimes also written as **UNIX**) is a multitasking, multi-user computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, Michael Lesk and Joe Ossanna.

There are numerous events in the computer industry that have occurred since UNIX started life as a small project in Bell Labs in 1969. UNIX history has been\largely influenced by Bell Labs' Research Editions of UNIX, AT&T's System V UNIX, Berkeley's Software Distribution (BSD), and Sun Microsystems' SunOS and Solaris operating systems.

Unix is designed to be a time-sharing system and has a simple standard user interface can be broadly divided into two categories: Some commands tell you the state of the system and others can (shell) that can be replaced. The general-purpose utilities of the system aid you directly in your work. Handling of ordinary files and navigating around such file systems are done through various commands. Text editors available editors on unix are **vi**, **emacs** and **pico.**

13.9 Keywords

Multitasking, Multi-user computer operating system

13.10Exercises

1. What are the designing principles of UNIX operating system?
2. Explain the different file types in UNIX operating system?

3. Which are the different commands used for handling of ordinary files in UNIX?
4. Write a short note on UNIX Editors.

13.11References

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.
4. Unix concepts and applications by Das, Sumitabha

UNIT-14:

Structure

14.0 Objectives

14.1 Introduction

14.2 Basic File Attributes

14.3 Process

14.4 Process Termination

14.5 Process Identifiers

14.6 Process Accounting

14.7 Process Relationship

14.8 Command-Line Arguments

14.9 User Identification

14.10 User Identification

14.11 Job Control

14.12 Unit Summary

14.13 Keywords

14.14 Exercise

14.15 Reference

14.0Objectives

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

- File attributes in more detail and how these can be changed.
- Ways to Process Unix
- How to use Unix Command line arguments.

14.1Introduction

Before looking at the process control primitives in the next chapter, we need to examine the environment of a single process. In this chapter, we'll see how the main function is called when the program is executed, how command-line arguments are passed to the new program, what the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and various ways for the process to terminate. Additionally, we'll look at the longjmp and setjmp functions and their interaction with the stack. We finish the chapter by examining the resource limits of a process.

14.2Basic File Attributes

There are three basic attributes for plain file permissions: read, write, and execute.

Read Permission of a file

If you have read permission of a file, you can see the contents. That means you can use more(1), cat(1), etc.

Write Permission of a file

If you have write permission of a file, you can change the file. This means you can add to a file, or overwrite a file. You can empty a file called "yourfile" by copying the empty (/dev/null) file on top of it

```
cat /dev/null yourfile
```

Execute Permission of a file

If the file has execute permission, then you can ask the operating system to run the file as if it were a program. If it's a binary file/program, you can execute it like any other program. In other words, of there is a file called "xyz", and it is in your searchpath, and the file is executable, all you need to do to run the program is type

```
xyz
```

If the file is a shell script, then the execute attribute says you can treat it as if it were a program. To put it another way, you can create a file using your favorite editor, add the execute attribute to it, and it "becomes" a program. However, since a shell has to read the file, a shell script has to be readable and executable. A compiled program does not need to be readable.

The basic permission characters, "r", "w", and "x"

r means read **w** means write, and **x** means e**X**ecute.

Simple, eh?

Using chmod to change permissions

The chmod(1) command is used to change permission. The simplist way to use the chmod command is to add or subtract the permission to a file. A simple plus or minus is used to add or sumtract the permission.

You may want to prevent yourself from changing an important file. Remove the write permission of the file "myfile" with the command

```
chmod -w myfile
```

If you want to make file "myscript" executable, type

```
chmod +x myscript
```

You can add or remove more than one of these attributes at a time

```
chmod -rwx file
chmod +wx file
```

You can also use the "=" to set the permission to an exact combination This command removes the write and execute permisison, while adding the read permission:

```
chmod =r myfile
```

Note that you can change permissons of files you own. That is, you can remove all permissions of a file, and then add them back again. You can make a file "read only" to protect it. However, making a file read only does not prevent you from deleting the file. That's because the file is in a directory, and directories also have read, write and execute permission. And the rules are different. Read on.

Directories use these same permissions, but they have a different meaning. Yes, very different meanings. This is classic Unix: terse, flexible and very subtle. Okay - let's cover the basic directory permissions.

Read permission on a directory

If a directory has read permission, you can see what files are in the directory. That is, you can do an "ls" command and see the files inside the directory. However, read permission of a directory does **not** mean you can read the **contents** of files in the directory.

Write permission on a directory

Write permission means you can add a new file to the directory. It also means you can **rename** or **move** files in the directory.

Execute permission on a directory

Execute allows you to **use** the directory name when accessing files inside that directory. The "x" permission means the directory is "searchable" when searching for executable. If it's a program, you can execute the program.

14.3Process

A process is a program in execution. Every time you invoke a system utility or an application program from a shell, one or more "child" processes are created by the shell in response to your

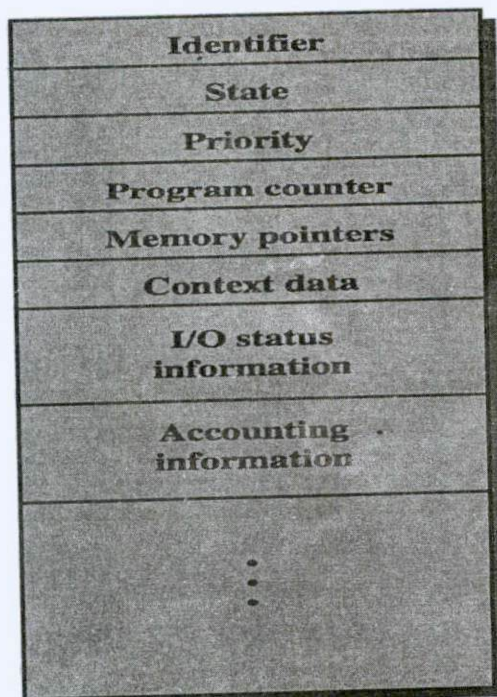
command. All UNIX processes are identified by a unique process identifier or PID. An important process that is always present is the init process. This is the first process to be created when a UNIX system starts up and usually has a PID of 1. All other processes are said to be "descendants" of init.

Process control block

For the duration of its existence, each process is represented by a *process control block* (PCB) which contains information such as:

- state of the process (ready, running, blocked)
- register values
- priorities and scheduling information
- memory management information
- accounting information such as open files
- allocated resources

Specifics of a PCB depend on the system and techniques used.



14.4Process Termination

There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from `main`
2. Calling `exit`
3. Calling `_exit` or `_Exit`
4. Return of the last thread from its start routine
5. Calling `pthread_exit` from the last thread

Abnormal termination occurs in three ways:

6. Calling `abort`
7. Receipt of a signal
8. Response of the last thread to a cancellation request

For now, we'll ignore the three termination methods specific to threads until we discuss threads .

Exit Functions

Three functions terminate a program normally: `_exit` and `_Exit`, which return to the kernel immediately, and `exit`, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>

void exit(intstatus);

void _Exit(intstatus);

#include <unistd.h>

void _exit(intstatus);
```

We'll discuss the effect of these three functions on other processes, such as the children and the parent of the terminating process

The reason for the different headers is that `exit` and `_Exit` are specified by ISO C, whereas `_exit` is specified by POSIX.1.

Historically, the `exit` function has always performed a clean shutdown of the standard I/O library: the `fclose` function is called for all open streams. Recall that this causes all buffered output data to be flushed (written to the file).

All three exit functions expect a single integer argument, which we call the exit status. Most UNIX System shells provide a way to examine the exit status of a process. If (a) any of these functions is called without an exit status, (b) `main` does a `return` without a return value, or (c) the `main` function is not declared to return an integer, the exit status of the process is undefined. However, if the return type of `main` is an integer and `main` "falls off the end" (an implicit return), the exit status of the process is 0.

This behavior is new with the 1999 version of the ISO C standard. Historically, the exit status was undefined if the end of the `main` function was reached without an explicit `return` statement or call to the `exit` function.

Returning an integer value from the `main` function is equivalent to calling `exit` with the same value. Thus

```
exit(0);
```

is the same as

```
return(0);
```

from the `main` function.

Terminating a Process

POSIX and C89 both define a standard function for terminating the current process:

```
#include <stdlib.h>
void exit (int status);
```

A call to exit( ) performs some basic shutdown steps, and then instructs the kernel to terminate the process. This function has no way of returning an error—in fact, it never returns at all. Therefore, it does not make sense for any instructions to follow the `exit()` call.

The status parameter is used to denote the process' exit status. Other programs—as well as the user at the shell—can check this value. Specifically, status & 0377 is returned to the parent. We will look at retrieving the return value later in this chapter.

EXIT_SUCCESS and EXIT_FAILURE are defined as portable ways to represent success and failure. On Linux, 0 typically represents success; a nonzero value, such as 1 or -1, corresponds to failure.

Consequently, a successful exit is as simple as this one-liner:

```
exit (EXIT_SUCCESS);
```

Before terminating the process, the C library performs the following shutdown steps, in order:

1. Call any functions registered with atexit( ) or on_exit( ), in the reverse order of their registration. (We will discuss these functions later in this page itself.)
2. Flush all open standard I/O streams.
3. Remove any temporary files created with the tmpfile( ) function.

These steps finish all the work the process needs to do in user space, so exit( ) invokes the system call _exit( ) to let the kernel handle the rest of the termination process:

```
#include<unistd.h>
void _exit (int status);
```

When a process exits, the kernel cleans up all of the resources that it created on the process' behalf that are no longer in use. This includes, but is not limited to, allocated memory, open files, and System V semaphores. After cleanup, the kernel destroys the process and notifies the parent of its child's demise.

Applications can call _exit( ) directly, but such a move seldom makes sense: most applications need to do some of the cleanup provided by a full exit, such as flushing the stdout stream. Note, however, that vfork( ) users should call _exit( ), and not exit( ), after a fork.

In a brilliant stroke of redundancy, the ISO C99 standard added the _Exit( ) function, which has identical behavior to _exit( ):

```
#include<stdlib.h>
void _Exit (int status);
```

14.5Process Identifiers

- `getuid`, `getgid`, `geteuid`, `getegid`
- process id, parent process id
- `getpid` and `getppid`
- real vs. effective user and group ids
- cast results of these functions to `long`
- output of `ps` command provides many of these details, including process state; experiment with the `-a`, `-A`, `-l`, and `-o` options
- `top`

14.6Process Accounting

- In general, UNIX provides user-based process accounting. That is, the operating system tracks system usage by recording statistics about each process that is run, including its UID. In addition, records are kept of the image that was run by the process and the

system resources (such as memory, CPU time, and I/O operations) that it used.

- The accounting system is designed for tracking system resource usage, primarily so that users can be charged money for them. The data collected by the accounting system can also be used for some types of system performance monitoring.
- The accounting systems under BSD and System V are quite different, but they are both based on the same raw data. Therefore, the sort of information that may be obtained from them is essentially identical, although output methods and formats are not.
- Accounting capabilities also need to be present in the UNIX kernel, and many systems make this configurable as well. On Linux systems, you will need to obtain the accounting specific kernel patches as they have not yet been merged into the kernel tree. They are found combined with the patches that enable disk quotas. You will also need to get the GNU accounting utilities and build them.

Standard Accounting Files

When accounting is enabled, the UNIX kernel writes a record to a binary data file as each process terminates. These files are stored in the home directory of the standard user adm, which is /var/admon most systems, although the previous location, /usr/adm, is still used by HP-UX and SCO UNIX.

Records written by both accounting systems to the raw accounting file contain the same data. It is only the ordering of the fields within each record that varies between System V and BSD (refer to the /usr/include/sys/acct.h file for details). Accounting records contain the following data about each process that runs on the system:

- Command image name (for example, grep).
- CPU time used (separated into user and system time).
- Elapsed time taken for the process to complete.
- Time the process began.
- Associated user and group IDs.
- Lifetime memory usage (BSD: average use of the process' lifetime; System V:

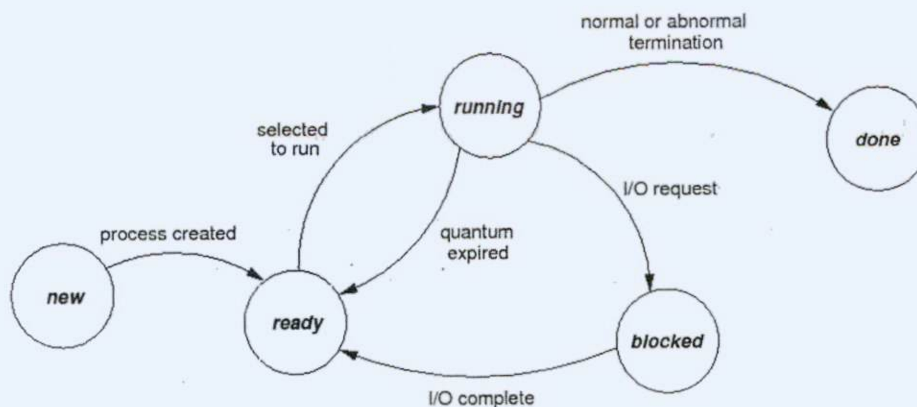
aggregate sum of the memory in use at each clock tick).

- Number of characters read and written.
- Number of disk I/O blocks read and written.
- Initiating TTY.
- Accounting flag associated with the process.
- Process' exit status.

14.7 Process Relationship

A process moves through various states during its life:

- new,
- ready,
- running,
- blocked, and
- done states, and
- transitions between them



Many processes may be ready or waiting at the same time, but only one can be running at a time in a uniprocessor system.

Process and process lifecycle nomenclature

- *batch process*: a process which is not connected to a terminal, or a non-interactive process

- *resident monitor*: first software support for an operating system; transferred CPU control from process to process; provided automatic process sequencing

- *multiprogramming*: multiple processes are kept in main memory and contend for the CPU with the goal of reducing the time the processor is idle

- *timesharing* (or *preemptive multi-tasking*): an extension of *multiprogramming* in which the CPU is switched between processes very quickly; a small amount of CPU time is given to each process (called a *time slice* or *quantum*); gives rise to interactive systems and enables communication between a user and a process

- non-preemptive multi-tasking = multiprogramming without timesharing

- *job scheduling*: policy by which processes are loaded into main memory or, in other words, the ready queue

- *ready queue*: contains all processes in main memory ready to execute; a process is dispatched from the ready queue to the CPU (called *process scheduling*); its processing may cause it to put on a device queue

- *process scheduling* (or *CPU scheduling*): policy by which processes are granted access to the CPU

- *context switch*: each process change requires a *context switch* which is purely overhead (i.e., no useful work is being accomplished); switching the CPU to another process requires i) saving the state of the old process and ii) loading the state for the new process

- *context switch time*: time required to perform a *context switch*

- *system call*: call to a core OS service, such as a read or write, and typically triggers a

  Context switch; a system call is a request to the operating system for a service

14.8 Command-Line Arguments

When a program is executed, the process that does the exec can pass command-line arguments to the new program. This is part of the normal operation of the UNIX system shells. We have already seen this in many of the examples from earlier chapters.

Example

The program in echoes all its command-line arguments to standard output. Note that the normalecho(1) program doesn't echo the zeroth argument.

If we compile this program and name the executable echoarg, we have

```
    $ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

We are guaranteed by both ISO C and POSIX.1 that argv[argc] is a null pointer. This lets us alternatively code the argument-processing loop as

```
    for (i = 0; argv[i] != NULL; i++)
```

14.9 Unix Kernel

The kernel runs the show, i.e. it manages all the operations in a Unix flavored environment. The kernel architecture must support the primary Unix requirements. These requirements fall in two categories namely, functions for process management and functions for file management (files include device files). Process management entails allocation of resources including CPU, memory, and offers services that processes may need. The file management in itself involves handling all the files required by processes, communication with device drives and regulating transmission of data to and from peripherals.

The kernel operation gives the user processes a feel of synchronous operation, hiding all underlying asynchronism in peripheral and hardware operations (like the time slicing by clock). In summary, we can say that the kernel handles the following operations:

1. It is responsible for scheduling running of user and other processes.
2. It is responsible for allocating memory.
3. It is responsible for managing the swapping between memory and disk.
4. It is responsible for moving data to and from the peripherals.
5. It receives service requests from the processes and honors them.

All these services are provided by the kernel through a call to a system utility. As a result, kernel by itself is rather a small program that just maintains enough data structures to pass arguments, receive the results from a call and then pass them on to the calling process. Most of the data structure is tables. The chore of management involves keeping the tables updated. Implementing such software architecture in actual lines of code would be very small. The order of code for kernel is only 10000 lines of C and 1000 lines of assembly code.

Kernel also aids in carrying out system generation which ensures that Unix is aware of all the peripherals and resources in its environment. For instance, when a new disk is attached, right from its formatting to mounting it within the file system is a part of system generation.

14.10 User Identification

Although not necessarily the case, in most cases a user on a UNIX system means a particular individual who can log in, edit files, run programs, and otherwise make use of the system. Each user has a username (login name) that identifies them. When adding a new user account to the system, the administrator assigns it a unique user identification number (UID). The UID is the system's way of identifying the user. The administrator also assigns each new user to one or more groups. A group is a collection of users who generally share a similar function. Each group also has a group identification number (GID). It is the system's way of identifying a group. Together, any user's UID and GID determine what kinds of access rights they have to files and other system resources.

User account information is stored primarily in the password file, /etc/passwd. This is an ASCII text file containing the complete list of system users, together with their usr and group ID's, and a coded form of their passwords. The /etc/group file lists defined groups. Both /etc/passwd and /etc/group are public information. All users may read them, but only the superuser is allowed to modify them.

Adding New Users

To add a new user to the system, you must:

- Assign the user a username, a user ID number, and a primary group and decide which other groups they should be a member of (if any).
- Enter this data into /etc/passwd, any secondary password file. And /etc/group.
- Assign a password to the new account.
- Set other user account parameters in use on the system (possibly including password aging, an account expiration date, resource limits, and system privileges).
- Create a home directory for the user.
- Place initialization files in the user's directory.
- Use chown and/or chgrp to give the new user ownership of their home directory and initialization files.
- Add the user to any other facilities in use such as the disk quota system, the mail system, and the printing system.
- Perform any other site specific initialization tasks.
- Test the new account.

14.11 Job Control

Methods of controlling processes are shown in the table below:

Form (Command or Action)	Meaning
&	Run command in background.
^Z	Stop foreground process.
jobs	List background processes.
%n	Refers to background jobs number *n*.
fg	Brings background process to foreground.
%?str	Refers to the background job command containing the characters in *str*.
bg	Restart stopped background process.
kill	Shell version of UNIX command.
~^Z	Suspend rlogin session.
~~^Z	Suspend 2nd level rlogin session. Useful for nested rlogin's. each additional tilde says to pop back to the next highest level of the rlogin. Thus, one tilde pops all the way back to the lowest level job, two tildes pops back to the first rlogin session, and so on.

14.12 Summary

Understanding the environment of a C program in a UNIX system's environment is a prerequisite to understanding the process control features of the UNIX System. In this chapter, we've looked at how a process is started, how it can terminate, and how it's passed an argument list and an environment. Although both are uninterpreted by the kernel, it is the kernel that passes both from the caller of exec to the new process.

We've also examined the typical memory layout of a C program and how a process can dynamically allocate and free memory. It is worthwhile to look in detail at the functions available for manipulating the environment, since they involve memory allocation. The functions setjmp and longjmp were presented, providing a way to perform nonlocal branching within a process. We finished the chapter by describing the resource limits that various implementations provide.

14.13 Keywords

Process Termination, Process Identifiers, Process Accounting, Process Timers, Process Relationships.

14.14 Exercises

1. Describe explain working of Unix Kernel

2. Briefly explain different file attributes
3. Explain Process termination.

14.15 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harve_ M Deitel, Addison-Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-15:

Structure

15.0 Objectives

15.1 Introduction

15.2 History of Linux

15.3 Design Principles

15.4 Kernel Modules

15.5 Process Management

15.6 Unit Summary

15.7 Keywords

15.8 Exercise

15.9 Reference

15.0Objectives

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

- Work with Linux operating system
- Ways to Process Linux
- Explore Linux Kernel.

15.1Introduction

Linux is a modem, free operating system based on UNIX standards. First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility. Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet. It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms.

The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code.

15.2 History of Linux

The Linux Kernel

Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system.

Linux 1.0 (March 1994) included these new features:

1. Support for UNIX's standard TCP/IP networking protocols

2. BSD-compatible socket interface for networking programming

3. Device-driver support for running IP over an Ethernet

4. Enhanced file system

5. Support for a range of SCSI controllers for high-performance disk access

6. Extra hardware support

Version 1.2 (March 1995) was the final PC-only Linux kernel.

Linux 2.0

Released in June 1996, 2.0 added two major new capabilities:

1. Support for multiple architectures, including a fully 64-bit native Alpha port.

2. Support for multiprocessor architectures _ other new features included:

3. Improved memory-management code

4. Improved TCP/IP performance

5. Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand.

6. Standardized configuration interface

Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems.

The Linux System

Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.

The min system libraries were started by the GNU project, with improvements provided by the Linuxcommunity. Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return. The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories.

Linux Distributions

Standard, precompiled sets of packages, or *distributions*, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools. The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management.

Early distributions included SLS and Slack ware. *Red Hat* and*Debian*are popular distributions from commercial and noncommercial sources, respectively. The RPM Package file format permits compatibility among the various Linux distributions.

Linux Licensing

The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation. Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product.

15.3Design Principles

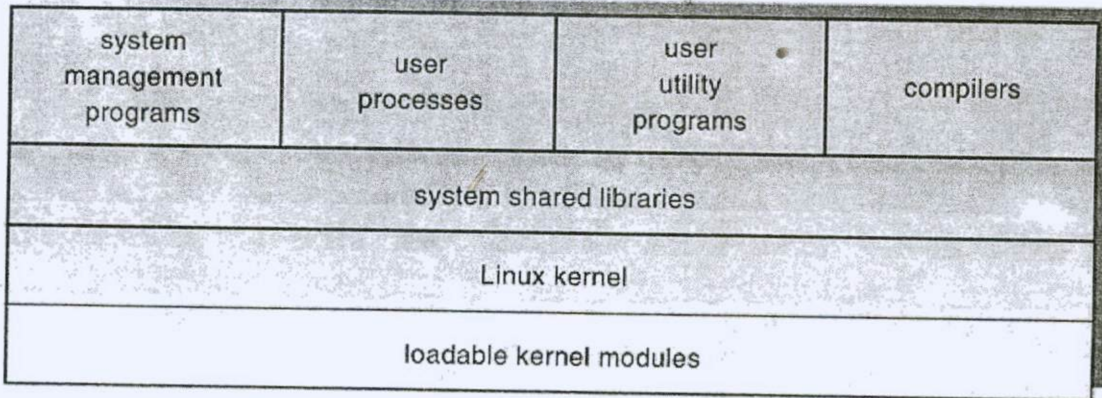
Linux is a multi-user, multitasking system with a full set of UNIX-compatible tools. Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.

Main design goals are speed, efficiency, and standardization.

- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior.

Components of a Linux System

Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components.



The kernel is responsible for maintaining the important abstractions of the operating system.

1. Kernel code executes in *kernel mode* with full access to all the physical resources of the computer.

2. All kernel code and data structures are kept in the same single address space.

The system libraries define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code. The system utilities perform individual specialized management tasks.

15.4Kernel Modules

~~Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of~~ the kernel. A kernel module may typically implement a device driver, a file system, or a networking protocol.

- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.

Three components to Linux module support:

1. Module management
2. Driver registration
3. Conflict resolution

Module Management

Supports loading modules into memory and letting them talk to the rest of the kernel.

Module loading is split into two separate sections:

1. Managing sections of module code in kernel memory
2. Handling symbols that modules are allowed to reference

The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

Driver Registration

Allows modules to tell the rest of the kernel that a new driver has become available.

The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.

Registration tables include the following items:

1. Device drivers
2. File systems
3. Network protocols
4. Binary format

Conflict Resolution

A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver

The conflict resolution module aims to:

1. Prevent modules from clashing over access to hardware resources
2. Prevent *auto probes* from interfering with existing device drivers
3. Resolve conflicts with multiple drivers trying to access the same hardware

15.5Process Management

UNIX process management separates the creation of processes and the running of a new program into two distinct operations.

1. The fork system call creates a new process.
2. A new program is run after a call to execve.

Under UNIX, a process encompasses all the information that the operating system must maintain *t* track the context of a single execution of a single program. Under Linux, process properties fall into three groups: the process's identity, environment, and context.

Process Identity

Process ID (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process.

- Credentials. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files.
- Personality. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls.

- Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX.

Process Environment

The process's environment is inherited from its parent, and is composed of two null- terminated vectors:

1. The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
2. The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.

Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user modesystem software. The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

Process Context

- The (constantly changing) state of a running program at any point in time.
- The scheduling context is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.
- The kernel maintains accounting information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.
- The file table is an array of pointers to kernel file structures. When making file I/O system calls, processes refer to files by their index into this table.
- Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
- The signal-handler table defines the routine in the process's address space to be called when specific signals arrive.
- The virtual-memory context of a process describes the full contents of the its private address space.

Processes and Threads

Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.

A distinction is only made when a new thread is created by the clone system call.

1. fork creates a new process with its own entirely new process context
2. clone creates a new process with its own identity, but that is allowed to share the data structures of its parent

Using clone gives an application fine-grained control over exactly what is shared between two threads.

15.6 Summary

Microsoft Windows and Linux are both growing in terms of server operating system market share. Windows, which only a few years ago was not considered up to the job of supporting critical system requirements, is now the primary server operating system for many companies--and not just small firms. Linux, which in the late 1990's was considered a hobbyist toy, now is the dominant operating system in some applications such as web servers and is part of the strategic technology platform for major vendors such as IBM and Oracle.

15.7 Keywords

Linux, Kernel module, Process management

15.8 Exercises

1. Describe explain working of Linux Kernel
2. Briefly explain different design principles of Linux operating system
3. Explain Process management.

15.9 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

UNIT-16:

Structure

16.0 Objectives

16.1 Introduction

16.2 Scheduling in Linux

16.3 Memory Management in Linux

16.4 Linux file system

16.5 Input and Output

16.6 Interprocess Communication

16.7 Unit Summary

16.8 Keywords

16.9 Exercise

16.10 Reference

16.0 Objectives

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

- Work with Linux operating system
- Ways to Process Linux
- Explore Linux Kernel.

16.1 Introduction

Multitasking operating systems come in two flavors: cooperative multitasking and preemptive multitasking. Linux, like all Unix variants and most modern operating systems, provides preemptive multitasking. In preemptive multitasking, the scheduler decides when a process is to cease running and a new process is to resume running. The act of involuntarily suspending a running process is called preemption. The time a process runs before it is preempted is predetermined, and is called the time slice of the process. The time slice, in effect, gives each process a slice of the processor's time. Managing the time slice enables the scheduler to make global scheduling decisions for the system. It also prevents any one process from monopolizing the system. As we will see, this time slice is dynamically calculated in the Linux scheduler to provide some interesting benefits.

16.2 Scheduling in Linux

We will now look at the Linux scheduling algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes. Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard (''real-time'' extensions to UNIX) which uses those names. The real time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level.

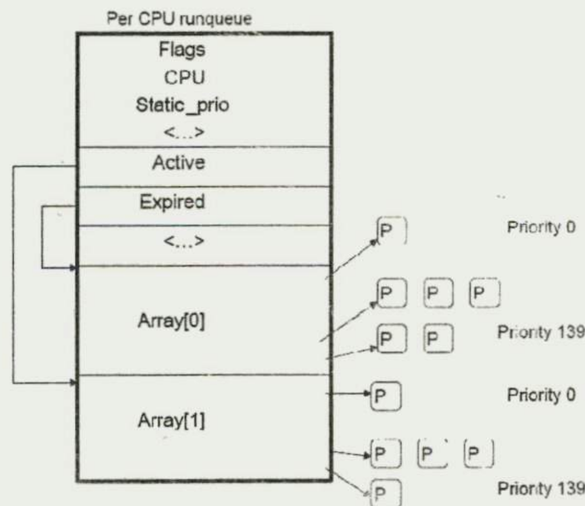
Like most UNIX systems, Linux associates a nice value with each thread. The default is 0 but this can be changed using the nice(value) system call, where value ranges from -20 to +19. This value determines the static priority of each thread. A user computing p to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from -20 to -1). Deducing the reason for this rule is left as an exercise for the reader.

- A key data structure used by the Linux scheduler is a **runqueue**. A runqueue is associated with each CPU in the system, and among other information, it maintains two arrays, *active* and *expired*. As shown in Fig. 10-10, each of these fields is a pointer to an array of 140 list heads, each corresponding to a different priority.The list head points to a doubly-linked list of processes at a given priority. The basic operation of the scheduler can be described as follows.
- The scheduler selects a task from the highest priority active array. If that task's timeslice (quantum) expires, it is moved to an expired list (potentially at a different priority level). If the task blocks, for instance to wait on an I/O event, before its timeslice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to reflect the CPU time it already consumed. Once its timeslice is fully exhausted, it too will be placed on an expired array. When there are

no more tasks in any of the active arrays, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa.

- This method ensures that low priority tasks will not starve (except when real-time FIFO threads completely hog the CPU, which is unlikely to happen).
- Different priority levels are assigned different timeslice values. Linux assigns higher quanta to higher priority processes. For instance, tasks running at priority level 100 will receive time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec.

The idea behind this scheme is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait a second between read calls will slow it down enormously. It is far better to let it run immediately after each request is completed, so it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that over when all the I/O bound and interactive processes are blocked.



- Since Linux (or any other OS) does not know apriori whether a task is I/O- or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is

continuously recalculated, so as to (1) reward interactive threads, and (2) punish CPU-hogging threads.

- The maximum priority bonus is ˉ5, since lower priority values correspond to higher priority received by the scheduler. The maximum priority penalty is +5. More specifically, the scheduler maintains a *sleep_avg* variable associated with each task. Whenever a task is awaken, this variable is incremented, whenever a task is preempted or its quantum expires, this variable is decremented by the corresponding value. This value is used to dynamically map the task's bonus to values from ˉ5 to +5.
- The Linux scheduler recalculates the new priority level as a thread is moved from the active to the expired list. The scheduling algorithm described in this section refers to the 2.6 kernel, and was first introduced in the unstable 2.5 kernel. Earlier algorithms exhibited poor performance in multiprocessor settings and did not scale well with an increased number of tasks.
- Since the description presented in the above paragraphs indicates that a scheduling decision can be made through access to the appropriate active list, it can be done in constant O(1) time, independent of the number of processes in the system.

In addition, the scheduler includes features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specify or modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock.

The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

In fact, the kernel code contains synchronization variables in numerous locations. Earlier Linux kernels had just one **big kernel lock (BLK)**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs to execute kernel code concurrently. Hence, many new synchonization points were introduced at much finer granularity.

16.5 Memory Management in Linux

Rather than describing the theory of memory management in operating systems, this section tries to pinpoint the main features of the Linux implementation. Although you do not need to be a Linux virtual memory guru to implement mmap, a basic overview of how things work is useful. What follows is a fairly lengthy description of the data structures used by the kernel to manage memory. Once the necessary background has been covered, we can get into working with these structures.

Address Types

Linux is, of course, a virtual memory system, meaning that the addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Virtual memory introduces a layer of indirection that allows a number of nice things. With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system's physical memory. Virtual memory also allows the program to play a number of tricks with the process's address space, including mapping the program's memory to device memory.

Thus far, we have talked about virtual and physical addresses, but a number of the details have been glossed over. The Linux system deals with several types of addresses, each with its own

semantics. Unfortunately, the kernel code is not always very clear on exactly which type of address is being used in each situation, so the programmer must be careful.

The following is a list of address types used in Linux. Figure shows how these address types relate to physical memory.

User virtual addresses

These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.

Physical addresses

The addresses used between the processor and the system's memory. Physical addresses are 32- or 64-bit quantities; even 32-bit systems can use larger physical addresses in some situations.

Bus addresses

The addresses used between peripheral buses and memory. Often, they are the same as the physical addresses used by the processor, but that is not necessarily the case. Some architectures can provide an I/O memory management unit (IOMMU) that remaps addresses between a bus and main memory. An IOMMU can make life easier in a number of ways (making a buffer scattered in memory appear contiguous to the device, for example), but programming the IOMMU is an extra step that must be performed when setting up DMA operations. Bus addresses are highly architecture dependent, of course.

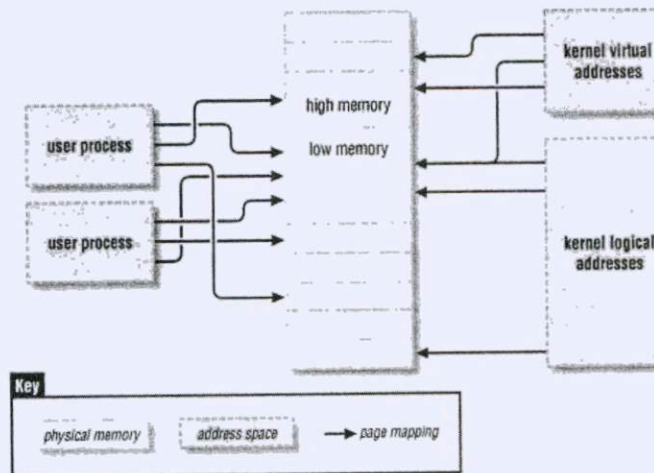
Kernel logical addresses

These make up the normal address space of the kernel. These addresses map some portion (perhaps all) of main memory and are often treated as if they were physical addresses. On most architectures, logical addresses and their associated physical

addresses differ only by a constant offset. Logical addresses use the hardware's native pointer size and, therefore, may be unable to address all of physical memory on heavily equipped 32-bit systems. Logical addresses are usually stored in variables of type unsigned long or void *. Memory returned fromkmalloc has a kernel logical address.

Kernel virtual addresses

Kernel virtual addresses are similar to logical addresses in that they are a mapping from a kernel-space address to a physical address. Kernel virtual addresses do not necessarily have the linear, one-to-one mapping to physical addresses that characterize the logical address space, however. All logical addresses are kernel virtual addresses, but many kernel virtual addresses are not logical addresses. For example, memory allocated by vmalloc has a virtual address (but no direct physical mapping). The kmap function (described later in this chapter) also returns virtual addresses. Virtual addresses are usually stored in pointer variables.



16.4 Linux file system

A simple description of the UNIX system, also applicable to Linux, is this:

"On a UNIX system, everything is a file; if something is not a file, it is a process."

This statement is true because there are special files that are more than just files (named pipes and sockets, for instance), but to keep things simple, saying that everything is a file is an acceptable generalization. A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

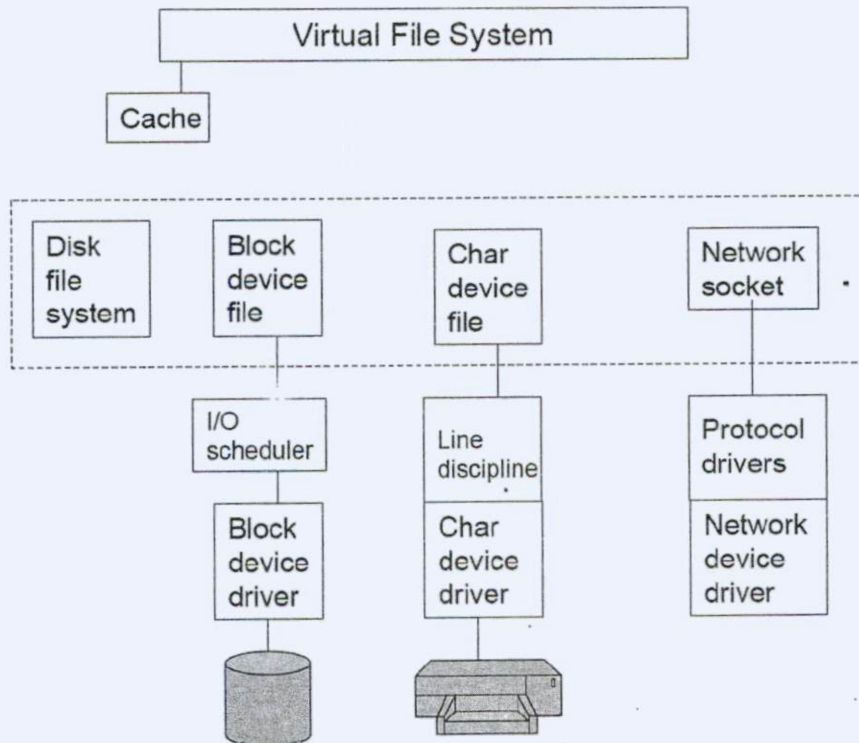
In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files. For now we will use this image of the tree, but we will find out later why this is not a fully accurate image.

Sorts of files

Most files are just files, called ***regular*** files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.

While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in /dev, we will discuss them later.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree. We will talk about links in detail.
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.



The -l option to ls displays the file type, using the first character of each input line:

```
jaime:~/Documents>ls -l
total 80
-rw-rw-r--   1 jaimejaime   31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r--   1 jaimejaime   41472 Feb 21 17:56 Linux.doc
drwxrwxr-x   2 jaimejaime    4096 Feb 25 11:50 course
```

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Special file
s	Socket
p	Named pipe

Symbol	Meaning
b	Block device

n order not to always have to perform a long listing for seeing the file type, a lot of systems by default don't issue just **ls**, but **ls -F**, which suffixes file names with one of the characters "/=*|@" to indicate the file type. To make it extra easy on the beginning user, both the -F and --color options are usually combined. As a user, you only need to deal directly with plain files, executable files, directories and links. The special file types are there for making your system do what you demand from it and are dealt with by system administrators and programmers. Now, before we look at the important files and directories, we need to know more about partitions.

16.5 Input and Output

Each I/O device in a Linux system generally has a special file associated with it. Most I/O can be done by just using the proper file, eliminating the need for special system calls. Nevertheless, sometimes there is a need for something that is device specific. Prior to POSIX most UNIX systems had a system call ioctl that performed a large number of device-specific actions on special files. Over the course of the years, it had gotten to be quite a mess. POSIX cleaned it up by splitting its functions into separate function calls primarily for terminal devices. In Linux, and modern UNIX systems in general, whether each one is a separate system call or they share a single system call or something else is implementation dependent.

- The first four listed in Fig. 10-20 are used to set and get the terminal speed. Different calls are provided for input and output because some modems operate at split speed. For example, old videotex systems allowed people to access public databases with short requests from the home to the server at 75 bits/sec with replies coming back at 1200 bits/sec.
- This standard was adopted at a time when 1200 bits/sec both ways was too expensive for home use. Times change in the networking world. This asymmetry still persists, with some telephone companies offering inbound service at 8 Mbps and outbound service at 512 kbps, often under the name of **ADSL** (**Asymmetric Digital Subscriber Line**).

- The last two calls in the list are for setting and reading back all the special characters used for erasing characters and lines, interrupting processes, and so on. In addition, they enable and disable echoing, handle flow control, and other related functions. Additional I/O function calls also exist, but they are somewhat specialized so we will not discuss them further. In addition, ioctl is still available.

Function call	Description
s = cfsetospeed(&termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(&termios, speed)	Get the output speed
s = cfgtetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, &termios)	Set the attributes
s = tcgetattr(fd, &termios)	Get the attributes

16.6 Interprocess Communication

The types of inter process communication are:

1. Signals - Sent by other processes or the kernel to a specific process to indicate various conditions.
2. Pipes - Unnamed pipes set up by the shell normally with the "|" character to route output from one program to the input of another.
3. FIFOS - Named pipes operating on the basis of first data in, first data out.
4. Message queues - Message queues are a mechanism set up to allow one or more processes to write messages that can be read by one or more other processes.
5. Semaphores - Counters that are used to control access to shared resources. These counters are used as a locking mechanism to prevent more than one process from using the resource at a time.
6. Shared memory - The mapping of a memory area to be shared by multiple processes.

Message queues, semaphores, and shared memory can be accessed by the processes if they have access permission to the resource as set up by the object's creator. The process must pass an identifier to the kernel to be able to get the access.

16.7 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), and the three forms of IPC commonly called XSI IPC (message queues, semaphores, and shared memory). Semaphores are really synchronization primitive, not true IPC, and are often used to synchronize access to a shared resource, such as a shared memory segment. With pipes, we looked at the implementation of the popen function, at co-processes, and the pitfalls that can be encountered with the standard I/O library's buffering.

16.8 Keywords

Scheduling, Memory management, Linux file system. Interprocess Communication.

16.9 Exercises

1. Describe explain working of Linux file system
2. Briefly explain different memory management of Linux
3. Explain Process interprocess communication.

16.10 Reference

1. Operating System Concepts and Design by Milan Milenkovic, II Edition McGraw Hill 1992.
2. Operating Systems by Harvey M Deitel, Addison Wesley-1990.
3. The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall of India.

ಆದೇಶ ಸಂಖ್ಯೆ : ಕರಾಮುವಿ/ಅಸಾವಿ/2-1614/2012-13 ದಿನಾಂಕ : 02-08-2012
ಒಳಪುಟ : 60 GSM ಮಾಪ್ಲಿತೊ ಮತ್ತು ರಕ್ಷಾ ಪುಟ : 220 GSM 'ಆರ್ಟ್ ಕಾರ್ಡ್'
ಮುದ್ರಕರು : ಶ್ರೀ ಸುಬ್ರಮಣ್ಯೇಶ್ವರ ಬುಕ್ ಡಿಪೋ, ಬೆಂಗಳೂರು - 560 002, ಪ್ರತಿಗಳು - 200

Karnataka State Open University

Manasagangotri Mysore - 570 006

The Open University system has been initiated in order to augment opportunities for higher education and as an instrument of democratizing education.

National Education Policy 1986

♣ **REGIONAL CENTRES**

Bangalore
Davanagere
Gulbarga
Dharwad
Shimoga
Mangalore
Tumkur
Hassan
Chamarajanagar
Bellary
Mandya
Kolar
Bijapur
Belagaum
Ramanagar
Bangalore (another one)
Chikmagalur
Udupi
Karwar
Bidar
Mysore

● **HEAD QUARTERS**

★ **Total Study Centres : 123**

♣ **Regional Centres : 21**

❄ **B.Ed Study Centres : 10**

✦ **M.Ed Study Centres : 06**

KSOU

Higher Education to everyone everywhere

ಉನ್ನತ ಶಿಕ್ಷಣ ಎಲ್ಲರಿಗೂ ಎಲ್ಲೆಡೆ

ಕರ್ನಾಟಕ ರಾಜ್ಯ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ

ಮಾನಸಗಂಗೋತ್ರಿ, ಮೈಸೂರು – 570 006

Karnataka State Open University

Manasagangotri, Mysore - 570 006 Website : www.ksoumysore.edu.in