edition 2021

FUNDAMENTAL OF
OPERATING SYSTEM

TECHNOLOGY

Hairi Alias & Saiful Azizi Abdullah

Preface

This book can facilitate a learner with a basic knowledge and information regarding to Operating System and all of it architecture, concept and mechanisms. As a start, in Chapter 1 will describing an Operating System overview according to OS history and generation, while in Chapter 2 reader will be exposed about foundation of Operating System management. In Chapter 3, discussion according to the resources that support the Operating System functions while in the last chapter the explanation and elaboration about the File Management in Operating System.

Copyright © 2023

Fundamental of Operating System Technology

All right reserved. It is not permitted to be reproduced, stored in retrieval system, or transmitted by any means whether electronic, mechanical, photocopying, recording or otherwise without written permission from publisher of Polytechnic Sultan Mizan Zainal Abidin.

© Polytechnic Sultan Mizan Zainal Abidin

Writer and Editor by Hairi bin Alias and Saiful Azizi bin Abdullah

Edition 2021

Publisher

Politeknik Sultan Mizan Zainal Abidin
KM.08 Jalan Paka,
23000 Dungun Terengganu
Tel : 09-8400800 / Fax : 09-8458781

Synopsis

This book covers the fundamental issue of Operating System. Learner will get certain knowledge according to the operating system technology in fundamental level. This is including about Operating System Overview, Basic in Management of Operating System and the Management of Files and Resources in Operating System environment. Each chapter contains a suitable diagram that can give readers a conceptual view of the discussed topic.

Contents

Chapter	Topic	Page
1	OPERATING SYSTEM OVERVIEW	2 - 19
2	BASIC of OPERATING CONCEPTS MANAGEMENT	30 - 39
3	RESOURCE MANAGEMENT	41 - 64
4	FILE MANAGEMENT	66 - 76
REFERENCES		77

CHAPTER ONE

OPERATING SYSTEM OVERVIEW

Introduction to Operating System

Operating System Generation

Operating System types

Various product of an Operating System

Open Source versus Closed Source

Shell Program: Menu-Driven and Graphical-based Interface

Terminologies in Operating System

Operating System Structure

Operating systems have been evolving through the years. In the following sections we will briefly look at this development. Since operating systems history call have been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none.

The first true digital computer was designed by the English mathematician Charles Babbage (1792-1871). Although Babbage spent most of his life and fortune trying to build his "analytical engine," he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed.
Needless to say, the analytical engine did not have an operating system.

Operating System Generation

The First Generation (1945-55) Vacuum Tubes and Plugboards

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until World War 11. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J.Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines using vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but were much slower than even the chest personal computer available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, often by wiring up plugboards to control the machine's basic functions. Programming languages were unknown (not even assembly language).

Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come

down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would bum out during the run. Virtually all the problems were straightforward numerical, calculations, such as grinding out tables of sinus and cosines.

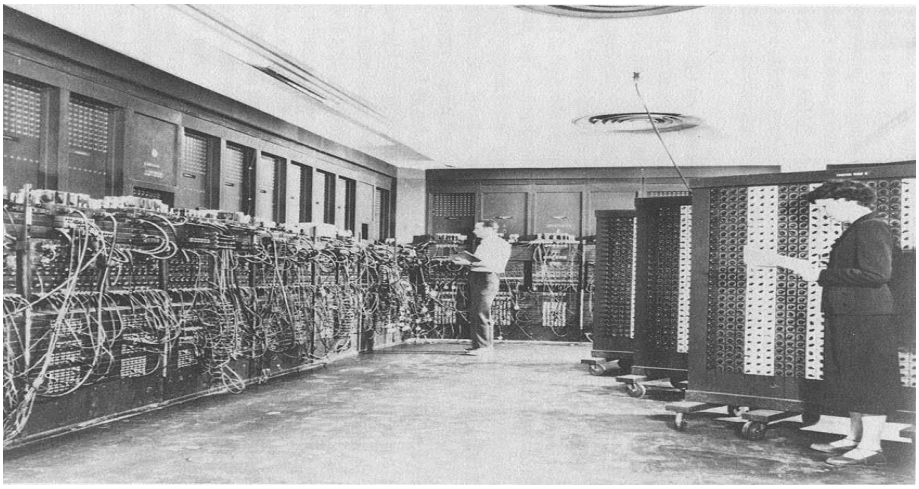


Figure 1.0 : room full of armoires: mechanical relays, then vacuum tubes

The Second Generation *(1955-65)* Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines were locked away in specially air conditioned computer rooms, with staffs of professional operators to run them. Only big corporations, or major government agencies or universities could afford the multimillion dollar price tag. To run a job (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and *carry* it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in, Much computer time was wasted while operators were walking around the machine room. Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the batch system. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small, (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations.

Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-2.

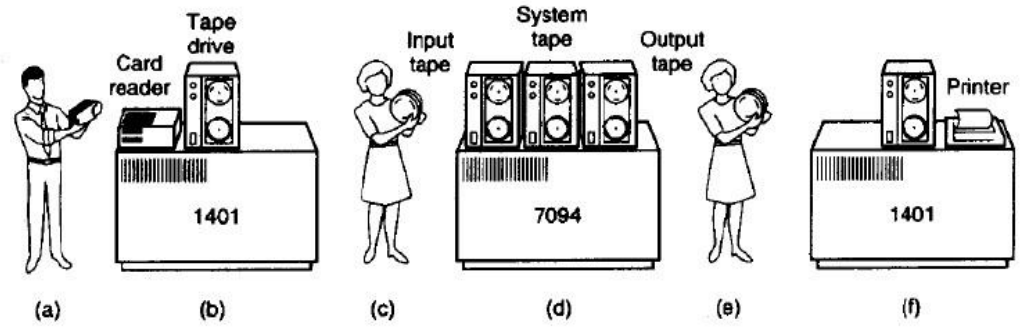


Figure 1-1. An early batch system.

(a) Programmers bring cards to 1401.

(b) 1401 reads batch of jobs onto tape.

(c) Operator carries input tape to 7094.

(d) 7094 does computing.

(e) Operator carries output tape to 1401.

(f) 1401 prints output.

After about an hour of collecting a batch of jobs, the tape was rewound and brought into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing off line (i.e., not connected to the main computer).

The Third Generation (1!hS1980): ICs and Multiprogramming

By the early **1960**s, most computer manufacturers had two distinct, and totally incompatible, product lines. On the one hand there **were** the word-oriented, large-scale scientific computers, such as the 7094, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which **were** widely used for tape, sorting and printing by banks and insurance companies.

Developing and maintaining two completely different product lines was an expensive proposition for the manufacturers. In addition, many new computer customers initially needed a small machine but later outgrew it and wanted a bigger machine that would run all their old programs, but faster. IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized to much more powerful than the 7094, The machines differed only in price and performance (maximum memory, processor speed, number of *I/O* devices permitted, and so forth.). The 360 was the first major computer line to use (small-scale) **Integrated Circuits (ICs)**, thus providing a major prior/performance advantage over the second generation machines, which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at scattered computer centers today, but their use is declining rapidly.

Figure 1.2 : IBM 7094 at Columbia University

Despite its enormous size and problems, *OS/360* and the similar third generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming**. On the 7094, when the current job paused to *wait* for a tape or other I/O operation to complete, the CPU simply set idle until the I/o finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the *110* wait time can often be 80 or 90 percent of the total time, so something had to be done to avoid having the CPU be idle so much.

The solution that evolved was to partition memory into several pieces, with a different job in each partition, as shown in Fig. 1-4. While one job was waiting for I/*O* to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100 percent of the time. Having multiple jobs in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

Another major feature present in third-generation operating systems was the ability to mid jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This technique is called spooling (from

Simultaneous Peripheral Operation On Line) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

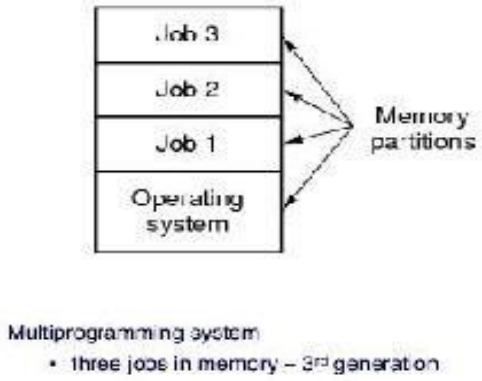


Figure 1-3. A multiprogramming system with three jobs in memory.

The Fourth Generation (1980-Present): Personal Computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned. In terms of architecture, personal computers were not that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

The most powerful personal computers used by businesses, universities, and government installations are usually called workstations, but they are really just large personal computers. Usually, they are connected together by a network. The widespread availability of computing power, especially highly interactive computing power usually with excellent graphics, led to the growth of a major industry producing software for personal computers. Much of this software was user-friendly meaning that it was intended for users who not only knew nothing *about computers* but furthermore had absolutely no intention whatsoever of learning. Another Microsoft operating system is WINDOWS m, which is compatible with WINDOWS 95 at a certain level, but a complete *rewrite* from scratch internally.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running network operating systems and distributed operating systems. In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users).

Operating System Types

Surely, the greatest leap of imagination in the history of operating systems was the idea that computers might be able to schedule their own workload by means of software.

BATCH OPERATING SYSTEM

The early operating systems took drastic measures to reduce idle computer time: the users were simply removed from the computer room! They were now asked to prepare their programs and data on punched cards and submit them to a computing center for execution. Now, card readers and line printers were too slow to keep up with fast computers. This bottleneck was removed by using fast tape stations and small satellite computers to perform *batch processing*.

Operators collected decks of punched cards from users and used a satellite computer to input a batch of jobs from punched cards to a magnetic tape. This tape was then mounted on a tape station connected to a main computer. The jobs were now input and run one at a time in their order of appearance on the tape. The running jobs output data on another tape. Finally, the output tape was moved to a satellite computer and printed on a line printer.

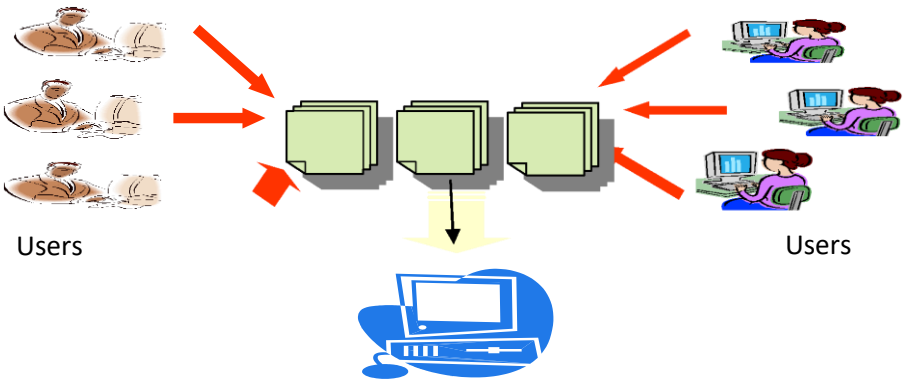


Figure 1.4 : Batch Operating System concept

In figure 1.4,

a. Users submit jobs to a central place where these jobs are collected into a batch

b. Subsequently placed on an input *queue* at the computer where they will be run.

c. The user has no interaction with the job during its processing

While the main computer executed a batch of jobs, the satellite computers simultaneously printed a previous output tape and produced the next input tape.

Batch processing was severely limited by the sequential nature of magnetic tapes and early computers. Although tapes could be rewound, they were only efficient when accessed sequentially. And the first computers could only execute one program at a time. It was therefore necessary to run a complete batch of jobs at a time and print the output in first-come, first-served order.

To reduce the overhead of tape changing, it was essential to batch many jobs on the same tape. Unfortunately, large batches greatly increased service times from the users' point of view. It would typically take hours (or even a day or two) before you received the output of a single job. If the job involved a program compilation, the only output for that day might be an error message caused by a misplaced semicolon!

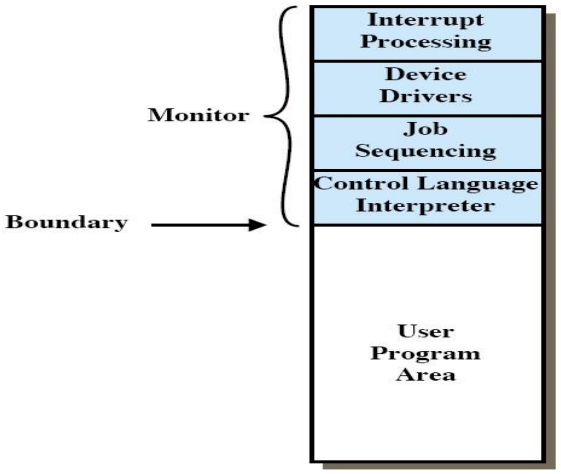


Figure 1.5 : Monitor's structure- a special software system in batch OS that cohabitates in memory with the job being executed

MULTIPROGRAMMING OPERATING SYSTEM

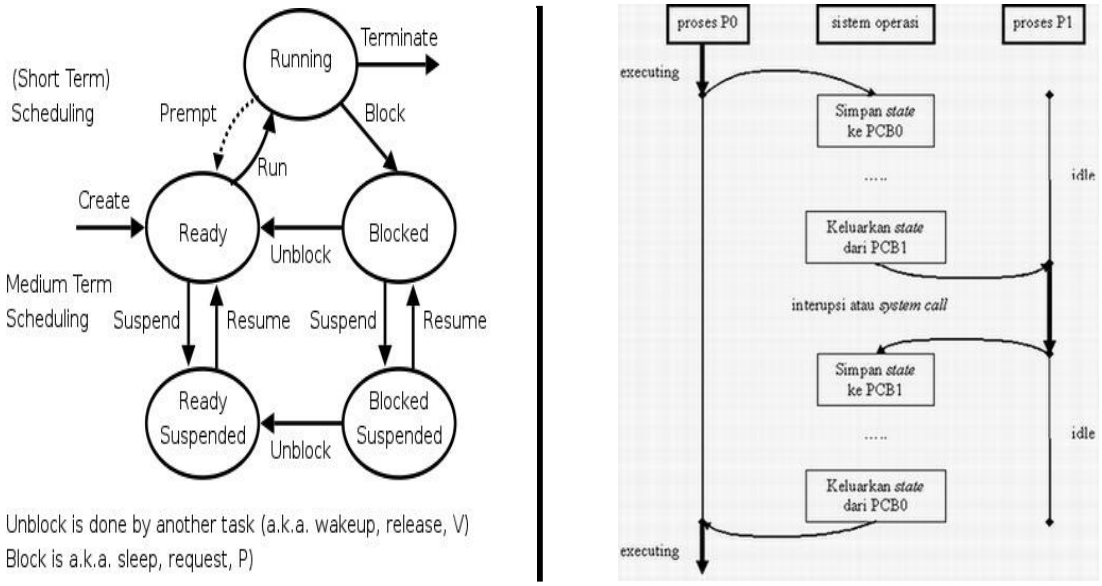
Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time. A concepts of multiprogramming OS as shown in figure 1.3

If the machine has the capability of causing an interrupt after a specified time interval, then the operating system will execute each program for a given length of time, regain control, and then execute another program for a given length of time, and so on. In the absence of this mechanism, the operating system has no choice but to begin to execute a program with the expectation, but not the certainty, that the program will eventually return control to the operating system.

In multiprogramming OS, there are two algorithms that were used to manipulate the task for being process by the system. They are called as Task Scheduling and Context Switching.

Task scheduling is an **Operation** to identify which task to be run by the CPU at the given task and which to be next. These tasks will be queued for processing .*Context Switching* is

a **Process** to store and restore the state of a CPU. When a process is executed, state of the CPU with respect to that process will be restored and while completing the time slice, the present state will be stored.



(a) (b)

Figure 1.6: The mechanism of task scheduling(a) and context switching(b) in CPU

DISTRIBUTED OPERATING SYSTEM

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system. True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized system differ in critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism.

1.2. Service and Support

Service is one of the key issues regarding open source software. Open source software relies on its online community network to deliver support via forums and blogs. While there are massive, loyal and engaged online communities that users can turn to, time- poor consumers of today are familiar with the immediate service and support that enables issues to be resolved in a timely manner, and these communities cannot guarantee the high level of responsive service and support proprietary software can offer.

1.3. Innovation

Open source software enables innovation by providing users with the freedom and flexibility to adapt the software to suit, without restriction. However, innovation may or may not be passed on to all users of the software. It is a user's prerogative whether they wish to share their innovation

with any online communities, and users must be actively participating in these communities to become aware of such innovations. It has been debated whether customized changes to the original source code limit the future support and growth of the software, as these can potentially result in a limited ability to apply future updates, fixes or modules aimed at improving the software, leaving the user with a version that may have irresolvable issues. It is relevant to note that open source software providers generally struggle to attract large scale R&D.

1.4. Usability

Open source software has been highly criticized for its lack of usability, as generally, the technology is not reviewed by usability experts and does not cater to the vast majority of computer users. Open source software is generally developer-centric, and without system administration experience or the knowledge required to manipulate programming language, use of the software and ability to fix errors as they arise is often limited to those with technical expertise.

1.5. Security

Open source software is often viewed as having security issues. Open source software is not necessarily developed in a controlled environment. While big players often have a concentrated development team, oftentimes the software is being developed by

individuals all over the world who may not work on the software for the duration of its developing lifetime. This lack of continuity and common direction can lead to barriers to effective communication surrounding the software. Furthermore, open source software is not always peer reviewed or validated for use. While users are free to examine and verify source code, the level of expertise required means that it is entirely possible for a programmer to embed back door Trojans to capture private and confidential information without the user ever knowing. Adopting a reputable brand with a concentrated development team supported by a strong online community will reduce the potential risk.

CLOSED SOURCE SOFTWARE

Closed source software can be defined as proprietary software distributed under a licensing agreement to authorized users with private modification, copying and republishing restrictions.

2.1. Cost

The cost of proprietary software will vary from a few thousand to a few hundred thousand dollars, depending on the complexity of the system required. This cost is made up of a base fee for software, integration and services and annual licensing/support fees. This cost may be prohibitive for some; however what the user is paying for is a more customized product from a trusted brand that includes higher levels of security and functionality, continuous innovation, a greater ability to scale, ongoing training and support and a lower requirement for technical skills.

2.2. Service and Support

If the internet is an important channel for an organisation, software is often a secondary concern, with service level and support structure requirements taking precedent in favour of maximising uptime and minimising downtime. Service is probably the greatest advantage of using proprietary software. Proprietary software providers offer ongoing support to users, a key selling point for users without technical expertise. If the user manual or guide is not enough, or if a user experiences a problem with the software, there is an immediate point of call to turn to for assistance. There is a certain reduction in the risk undertaken with proprietary software because users are working with companies that are viable, and people with intimate knowledge of the products and services being used should any questions arise.

2.3. Innovation

Proprietary software providers do not allow users to view or alter the source code. While this may be viewed as a disadvantage to some, it ensures the security and reliability of the software. Furthermore, many proprietary software providers customize software for specific users to provide more flexibility while investing in R&D in order to regularly offer new products and upgrades. Moreover, proprietary software providers have online user communities that create value by sharing ideas, strategies and best practices through feedback mechanisms such as forums and surveys, which also foster innovation and allow the product to adapt with changing needs. This innovation comes fully tested, and is available to all users of the software.

2.4. Usability

Proprietary software generally employs expert usability testing, and as the software is normally aimed at a more targeted audience, and therefore more tailored, usability is generally ranked quite high. In addition, detailed user manuals and guides are provided. This enables faster training and provides an immediate reference, allowing users to move along the learning curve more quickly. Supporting services include seminars, targeted training courses and extensive support to help maximise use of the software.

2.5. Security

Proprietary software is viewed as more secure because it is developed in a controlled environment by a concentrated team with a common direction. Moreover, the source code may be viewed and edited by this team alone, and is heavily audited, eliminating the risk of back door Trojans and reducing the risk of any bugs or issues with the software.

When deciding between open source or closed source (proprietary) software, it is critical to first consider the organization's business internal (resources and capabilities) and external (stable or evolving) environment, and the level of risk the organization is willing to take.

Shell Program : Menu-Driven and Graphical-based Interface

Menu-Driven shell

A user interface that uses menus to communicate with the computer. Rather than having a single line where a command must be typed in, the user has a list of items to choose from, and can make selections by highlighting one. This kind of interface is easier to use than a command-line interface, but does not have all the visual elements of a graphical user interface.

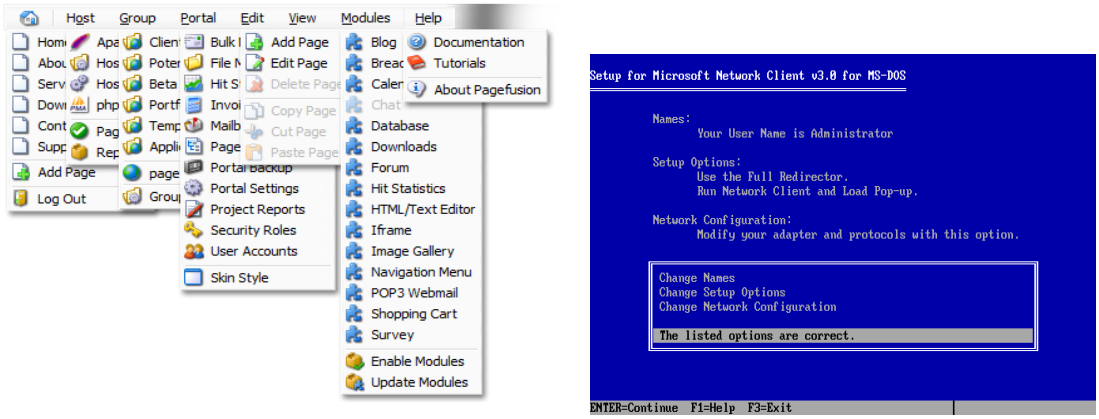
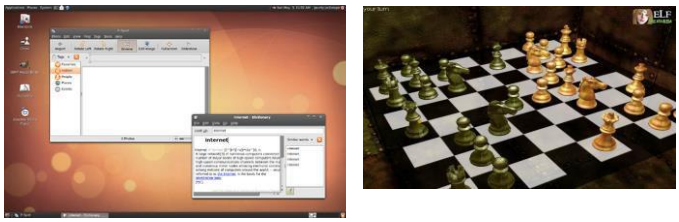


Figure 1.11 : Menu-driven interface

Graphical-Driven shell

A **graphical user interface (GUI)** (sometimes pronounced **gooey**) is a type of user interface item that allows people to interact with programs in more ways than typing such as computers. A *GUI* offers graphical icons, and visual indicators, as opposed to text-based interfaces, typed command labels or text navigation to fully represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements.

Figure 1.12 : Graphical-Driven Interface



Network Operating System

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

Terminologies in Operating System

Multitasking

Multitasking is a method of running several jobs at a time, now jobs can be either in the form of programs, processes, threads , user performing multi tasks(playing songs, working on word-processor etc..) at a time in single pc only etc. The main idea behind to do so is better CPU utilization. Several jobs are kept in the memory at a time such that when CPU is busy in execution of one job or task then OS switches over to other job and make it ready to get its next turn for execution by CPU. In this way CPU is used efficiently. Moreover, there are two algorithms which are support the multitasking process such as preemptive multitasking and cooperative multitasking.

Preemptive Multitasking

Preemptive multitasking is task in which a computer operating system uses some criteria to decide how long to allocate to any one task before giving another task a turn to use the operating system. The act of taking control of the operating system from one task and giving it to another task is called *preempting*. A common criterion for preempting is simply elapsed time (this kind of system is sometimes called *time sharing* or *time slicing*). In some operating systems, some applications can be given higher priority than other applications, giving the higher priority programs control as soon as they are initiated and perhaps longer time slices.

Cooperative Multitasking

A type of multitasking which is the process currently controlling the CPU must offer control to other processes. It is called *cooperative* because all programs must cooperate

for it to work. If one program does not cooperate, it can hog the CPU. In contrast, *preemptive multitasking* forces applications to share the CPU whether they want to or not.

Operating System Structures

The OS structure is a container for a collection of structures for interacting with the operating system's file system, directory paths, processes, and I/O subsystem. The types and functions provided by the OS substructures are meant to present a model for handling these resources that is largely independent of the operating system.

Monolithic Structure

The components of monolithic operating system are organized haphazardly and any module can call any other module without any reservation. Similar to the other operating systems, applications in monolithic OS are separated from the operating system itself. That is, the operating system code runs in a privileged processor mode (referred to as kernel mode), with access to system data and to the hardware; applications run in a non-privileged processor mode (called as user mode), with a limited set of interfaces available and with limited access to system data.

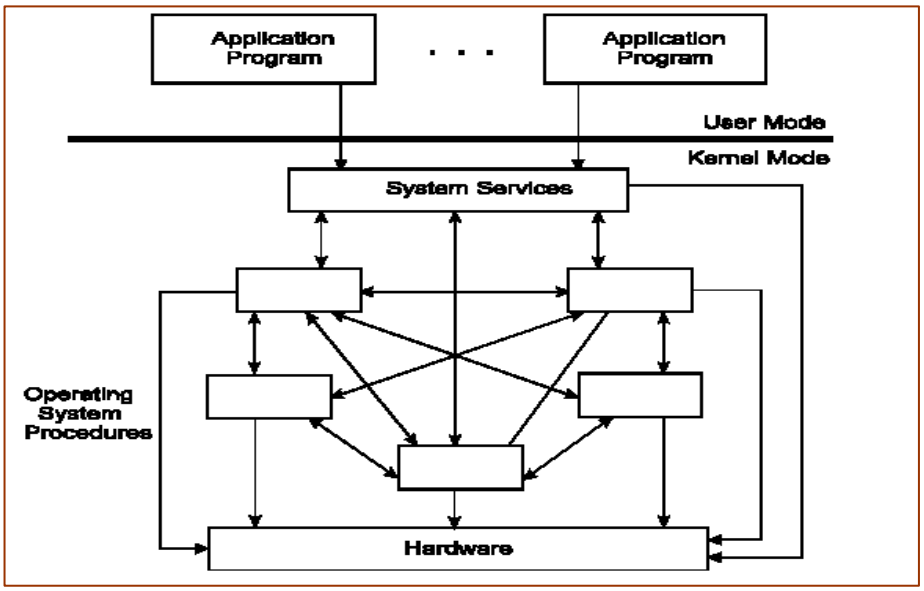


Figure 1.13 : Monolithic Structure

When a user-mode program calls a system service, the processor traps the call and then switches the calling thread to kernel mode. Upon completion, thread switched back to the user mode, by the operating system and allows the caller to continue. The monolithic structure does not enforce data hiding in the operating system. It delivers better application performance, but extending such a system can be difficult work because modifying a procedure can introduce bugs in seemingly unrelated of the system. The example of this structure like a CP/M and MS-DOS.

Layered Structure

The components of layered operating system are organized into modules and layers them one on top of the other. Each module provides a set of functions that other module can call. Interface functions at any particular level can invoke services provided by lower layers but not the other way around.

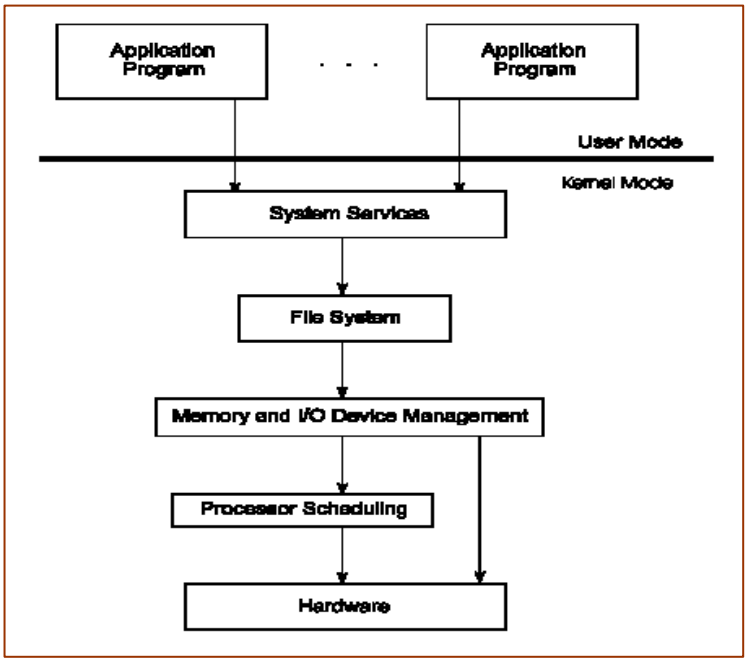


Figure 1.14 : Layered Structure

One advantage of a layered operating system structure is that each layer of code is given access to only lower-level interfaces(and data structures) it requires, thus limiting the amount of code that will unlimited power. That is in this approach, the N^{th} layer can access provided by the $(N-1)^{th}$ layer and provide services to $(N+1)^{th}$ layer. This will cause

the operating system to be debugged starting at the lowest layer, adding one layer at a time until the whole system worked correctly. This can enhance the OS with one layer can be replaced without affecting other parts of the system. Layered OS structure delivers low in performance in comparison to monolithic structure but good in stability.

Client-Server or Microkernel Structure

The idea in this structure is to divide the operating system into several processes, each of which implements a single set of services such as I/O servers, memory server, thread server, process server and interface system. Each server runs in user mode, provide services to the requested client. The client, which can be either another operating system component or application program, requests a service by sending a message to the server. An OS kernel running in kernel mode delivers the message to the appropriate server then perform the operation and kernel delivers the result to the client in another message which shown in figure 1.15

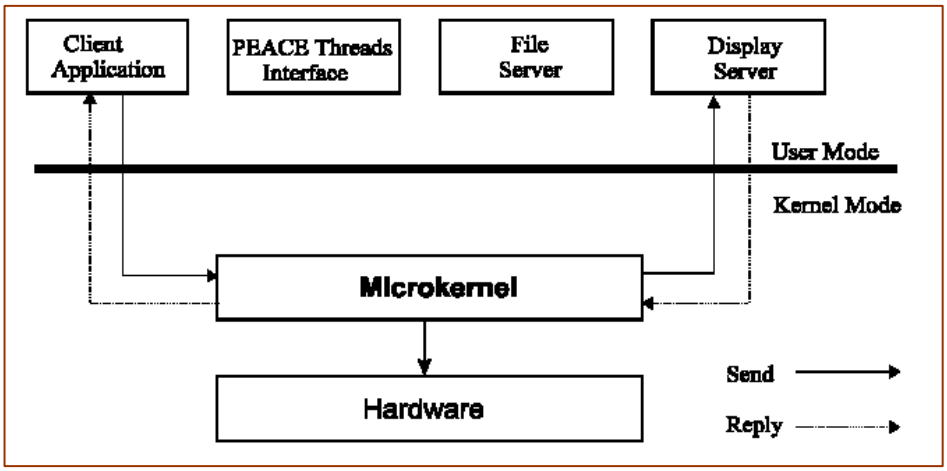


Figure 1.15 : A Client-server structure

Virtual Machines structure

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

everything else the real machine has. For reason of Each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the hard ware. Different virtual machines can, and usually do, run different operating systems. The example OS which used this structure are IBM VM/370, JAVA, VMWare. The structure of virtual-machine concept can be likely as figure 1.16.

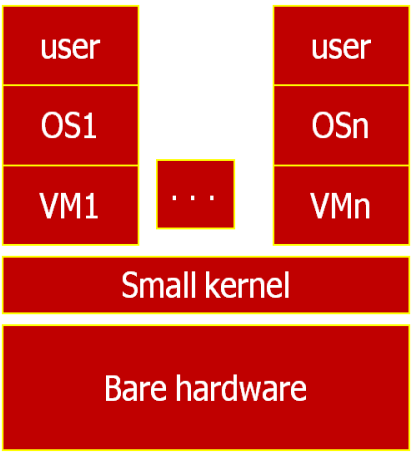


Figure 1.16 : Concept of Virtual-Machines structure

CHAPTER TWO

OPERATING SYSTEM BASIC CONCEPT

User Interface Management

- Command line Interface (CLI)
- Graphical User Interface (GUI)
- Voice User Interface (VUI)
- Menu User Interface (MUI)
- Web User Interface (WUI)

Basic function of OS component

- User Interface function
- File System function
- Logical IO and Physical IO
- Directory Management
- Disk Space Management

IO Control System (IOCS)

- Boot Process Procedure

User Interface

User Interface (UI) is a hardware and software which facilitate communication between the user and the computer. Which mean, it is a part of human-computer interaction field and is one of the most interesting parts of application development thus it is one of the subjects to intensive research to many computer scientists. In order for UI to work, the system needs to have input devices such as mouse, keyboard, touch screen, microphone, etc. Moreover, the system requires output devices such as monitor, printer, speaker, etc. Both user and computer send and receive data or instructions through these devices. There are many types of UI since the computer expends a years ago.

Command-Line Interface

Often called as CLI is a user interface to a computer's operating system or an application in which the user responds to a visual prompt by typing in a command on a specified line, receives a response back from the system, and then enters another command, and so forth. Using this kind of UI, the program use less memory because they no need any graphical items to be load. It also provide quickness to operate and very flexible, but the constraints are the user needs to learn and remember all the commands and type them correctly The MS-DOS Prompt application in a Windows operating system is an example of the provision of a command line interface.

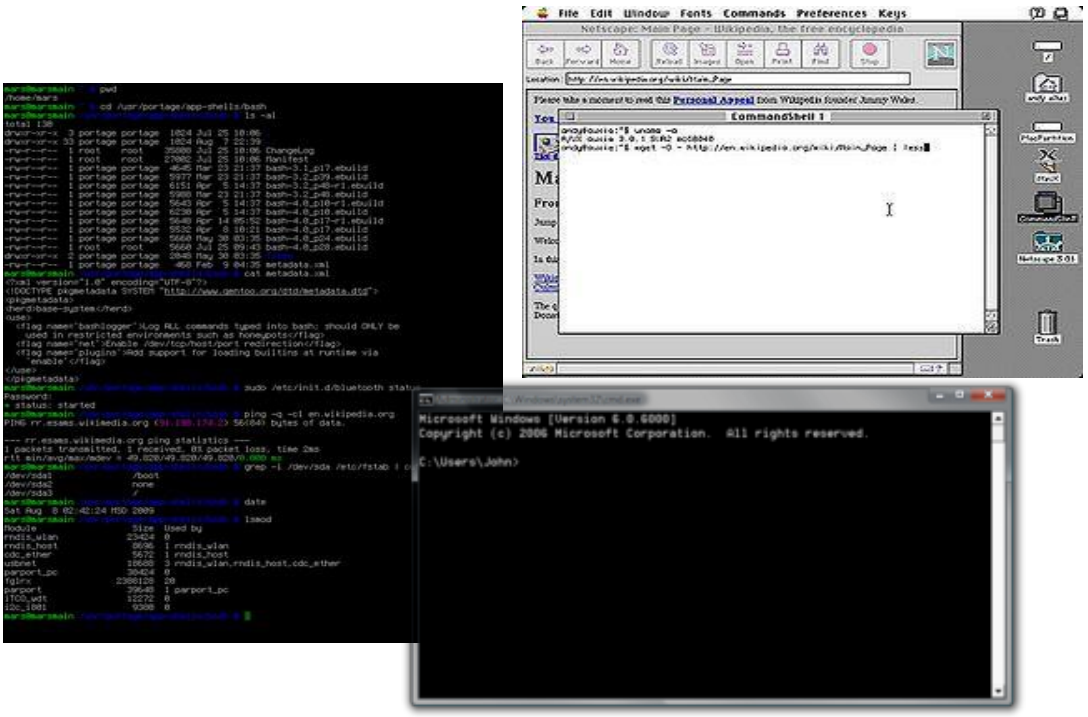
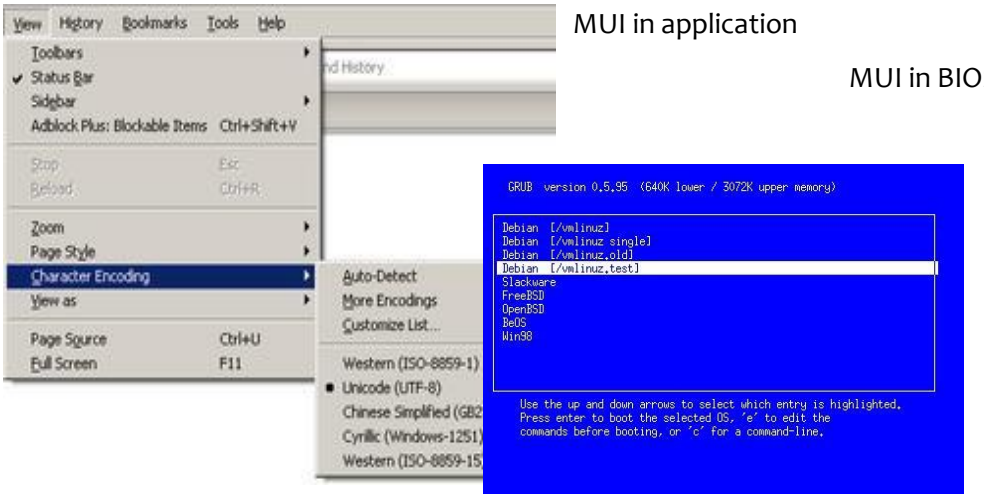


Figure 2.0: A various CLI in various of OS

Menu User Interface

MUI or Menu User Interface is a user interface that uses menus to communicate with the computer. Rather than having a single line where a command must be typed in, the user has a list of items to choose from, and can make selections by highlighting one. This kind of interface is easier to use than a command-line interface, but does not have all the visual elements of a graphical user interface. The user selects an option with either a key or a click of the mouse. Nowadays, you can see many type of menu interface in many form of applications that running in operating system including booting option in your BIOS program



MUI in application

MUI in BIOS

Figure 2.1: Menu User Interface

Graphical User Interface

GUI is a computer program that enables a person to communicate with a computer through the use of symbols and pointing devices. In addition of using graphical-based elements to execute a task, GUI also known as a **WIMP** (Windows, Icons, Menus, Pointers) interface. The users realize that using this kind of interface is the easiest way to communicate through the system. Moreover, well-designed graphical user interfaces can free the user from learning complex command languages. GUIs make computer operation more intuitive, and thus easier to learn and use. Example of GUI is Windows

You can see many form of GUI in previous chapter under part of graphical-based interface.

Voice User Interface

VUI or often known as sound interface is UI which accept input and provide output by generating voice prompts. Also known as Voice-actuated user interface. In this UI, the user input is made by pressing keys or buttons, or responding verbally to the interface. A Voice User Interface (VUI) makes human interaction with computers possible through a voice/speech platform in order to initiate an automated service or process. VUI designed for the general public should emphasize ease of use and provide a lot of help and guidance for first-time callers.

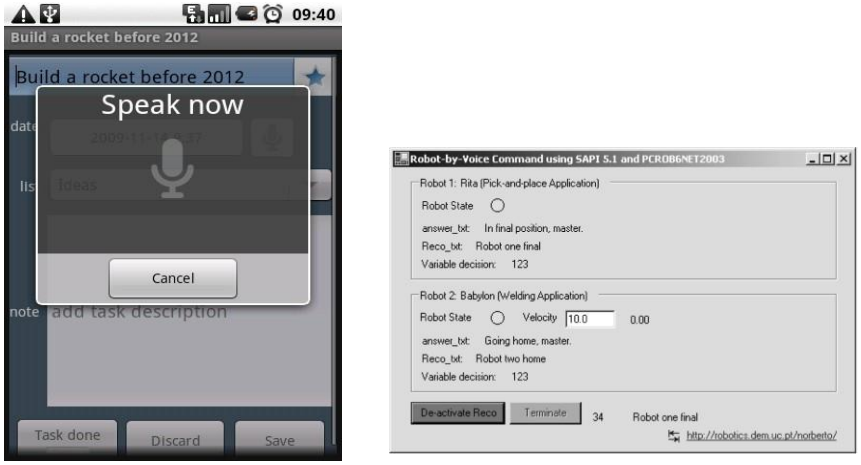


Figure 2.2: Voice User Interface forms

Web User Interface

This WUI or also called as Web-form user interface is always found within the net application-based. This UI appear in web-based environment which the user can feel it through a browser. This Web-based user interfaces or web user interfaces (WUI) accept input and provide output by generating web pages which are transmitted via the Internet.

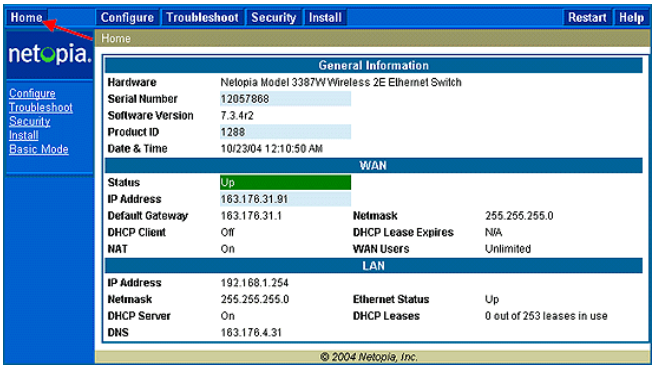


Figure 2.3: an example of Web User Interface

Basic Function of Operating System components

User Interface function

UI as a communication medium between user and computer or application is the important component to ensure the system performing the task properly as defined by the user. Hence, the functionality of UI are to :

- provide user and computer sends and receives data or instructions
- Manage task or process that operated by the user
- facilitate communication between the user and the computer
- Performed user satisfaction

File System function

"*a file system (sometimes written file system) is the way in which files are named and where they are placed logically for storage and retrieval. "*

File systems are responsible in management of :

File mgt: providing mechanism for files to be stored, referenced, shared and secure

Auxiliary storage mgt: allocating space for files on secondary or tertiary storage device

File integrity mechanism: ensuring that the information stored in a file is uncorrupted. When file integrity is assured, files contain only the information that they are intended to have.

Access methods: how the stored data can be accessed.

Otherwise, the functionality of file system includes as:

- Enable users to create, modify and delete files and to structure files in manner most appropriate for each application and initiate data transfer between files
- Enable users to share each other's files in a carefully controlled manner to build upon each other work.
- Enable users to refer to their files by symbolic names
- Provide backup capabilities that facilitate the creation of redundant copies of data
- Provide recovery capabilities that enable users to restore any lost or damage data
- Provide encryption and decryption capabilities – makes information useful only to its intended audience

Logical I/O and Physical I/O

This is an interface driven by user software interrupt system calls and, in turn, issues commands, based upon those calls, to the rest of the system. Use is made of the directory structure to implement the Logical I/O module and physical I/O module. This section is normally responsible for the security and protection of the system data. Examples of system calls made to the Logical file system are Open, Close, Read, and Write.

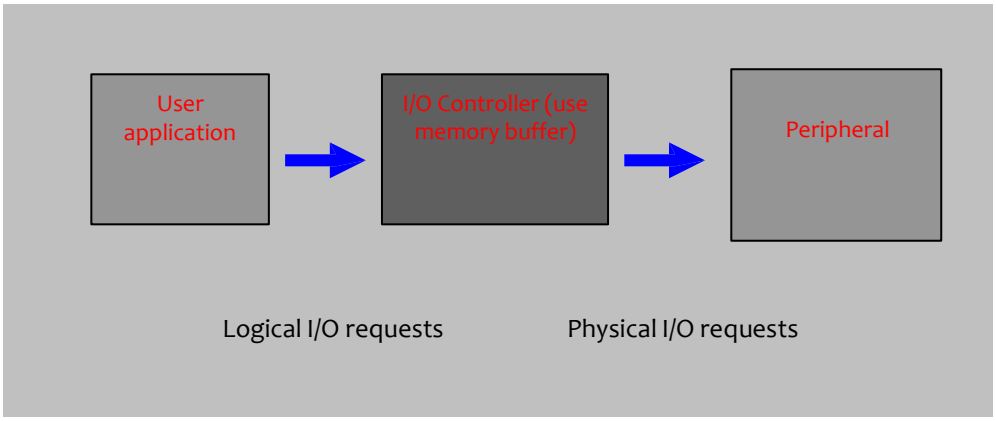


Figure 2.4: A Logical i/o and Physical i/o concepts

In logical, the operation (read+write) request is defined in different item composed to the IO Controller(in virtual) for translation process with sequence order and signal to peripherals, here, the requisition reference hold into memory buffer.

Later then, IO Controller will configure a detail transition by translated logical request into physical type of request for performing track, sector and block information and send them to peripherals to complete an IO operation

Directory Management and Disk Space Management

Directory Management

The selection of directory management algorithm has large effect on the efficiency, performance and reliability of the file systems. Therefore, need to know about tradeoffs involved in these algorithms such as linear list and hash table.

Linear List

Is the simplest method to implement a directory by using a way of a file names with pointers to the data blocks. A linear list of directory entries requires a linear search to find

a particular entry.

: To create new files - ensure that has no file name with the same name to the new one existed in the same directory. Then we add a new entry at the end of directory.

: To delete files - search the directory for the named file, then release the space allocated to it.

: to reuse the directory entry – mark entry as unused(by assigning at it special name), or attach it to a list of free directory entries

A linked list can also be used to decrease the time to delete a file

Hash Table

In this method, a linear list stores the directory entries but a hash data structures 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. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions-situation where two file names hash to the same location. The major difficulties with the hash table are its generally fixed size and the dependence of the hash function on that size.

Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an ***n*** byte file: n consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory management systems between pure segmentation and paging. Storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises of how big the block should be. Given the **way** disks are organized, the sector, the track and the cylinder are obvious candidates for the unit of allocation. In a paging system, the page size is also a major contender. On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a

rotational delay, so reading a file consisting of many small blocks will be slow.

Keep track of Free Block

Once a block size has been chosen, the *next* issue is how to keep track of free blocks. Two methods *are* widely used, as shown in Figure 2.4. The first one con*sists of* using a linked list of disk blocks, with each block holding as many free disk b~ock numbers as will fit. With a 1K block and a 32-bit disk block number, each black *on* the free list holds the numbers of 255 free blocks. (One slot is needed *for the* painter to the next block). A 200M disk n d s a free list of maximum 804 blocks *to* hold all 200K disk block numbers. Often free blocks are used *to* hold the free list.

The other free space management technique is the bit map. A disk with *n* blocks requires a bit *mag with n* bits. Pree blocks are represented by *Is* in the *map,* allocated Macks by *Os* (or vice *versa). A 26MM* disk requires 200K bits for the map, which requires only 25 blocks. It is not surprising *that the* bit map requires less space, since it uses I bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full will the linked list scheme require fewer blocks than *the* bit *map* as shown in figure below

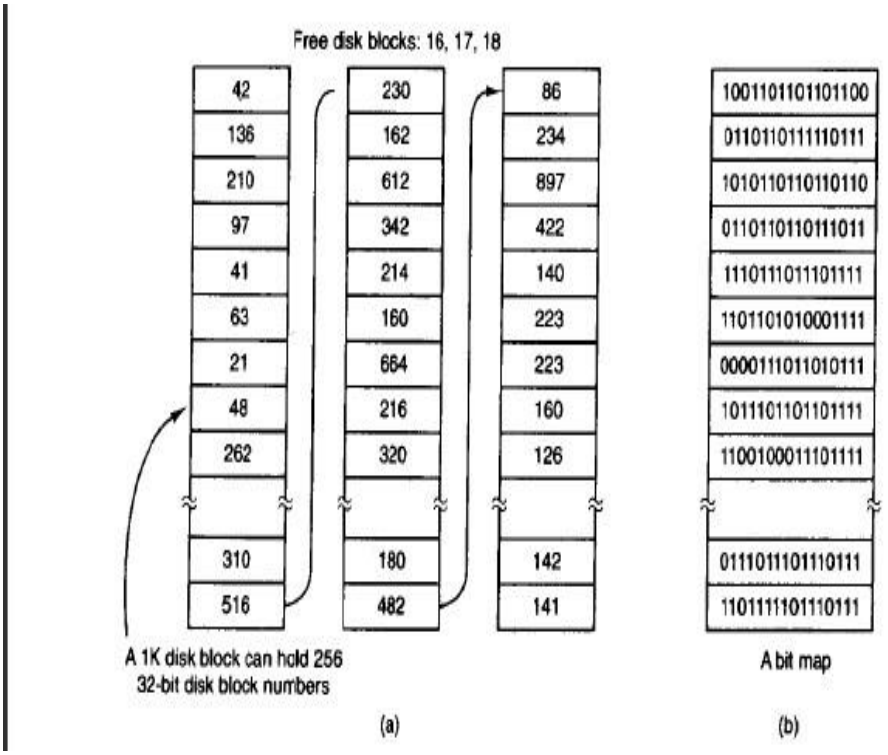


Figure 2.4: (a) storing the free list in linked-list. (b) A bit map

IO Control System(IOCS)

"A set of flexible routines that supervise the input and output operations of a computer at the detailed machine-language level. Abbreviated IOCS."

-answers.com

In many early operating system designs the software known as the input/output control system (IOCS) played a central conceptual and functional role. In the pre-multiprogramming, batch operating systems, many supervisory functions had to do with input/output control -- e.g., control over queued jobs, control for management and operation of secondary storage, control for operation of display devices and other peripheral equipment, etc. A system programmer (or subsystem designer) for such operating systems could hardly prove his professional competence without acquiring a reasonable familiarity with the intricacies of the IOCS for his "installation".

This is possible partly because two operations sometimes associated with the IOCS have been separated into separate functional units which are made use of by other parts of the system as well as the IOCS. First, the file system makes known and dynamically links files that are stored within the system to processes that legitimately request this service. It does not matter on what storage device these files reside at the time of the request. The users (or for that matter other supervisory modules) are unaware of any explicit data movement in accessing these segments even though physical transfer from actual secondary devices to central memory may occur. Secondly, the traffic controller handles all multiplexing of processors including the relinquishing of a processor by a process and the awakening of processes which have been waiting for I/O transactions to be completed. What remains for the IOCS is strategic control of I/O devices and the binding of these devices with symbolic names used to represent them.

Boot Process Procedure

In order for a computer to successfully boot, its BIOS, operating system and hardware components must all be working properly; failure of any one of these three elements will likely result in a failed boot sequence.

When the computer's power is first turned on, the CPU initializes itself, which is triggered by a series of clock ticks generated by the system clock. Part of the CPU's initialization is to look to the system's ROM BIOS for its first instruction in the startup program.

The ROM BIOS stores the first instruction, which is the instruction to run the power-on self test (POST), in a predetermined memory address.

POST begins by checking the BIOS chip and then tests CMOS RAM. If the POST does not detect a battery failure, it then continues to initialize the CPU, checking the inventoried hardware devices (such as the video card), secondary storage devices, such as hard drives and floppy drives, ports and other hardv e, to ensure they are functioning properly.

Figure 2.5 : Common display on startup

Once the POST has determined that all components are functioning properly and the CPU has successfully initialized, the BIOS looks for an OS to load. The BIOS typically looks to the CMOS chip to tell it where to find the OS, and in most PCs, the OS loads from the C drive on the hard drive even though the BIOS has the capability to load the OS from a floppy disk, CD or ZIP drive. The order of drives that the CMOS looks to in order to locate the OS is called the *boot sequence*, which can be changed by altering the CMOS setup.

Figure 2.6: boot sequence interface

Looking to the appropriate boot drive, the BIOS will first encounter the boot record, which tells it where to find the beginning of the OS and the subsequent program file that will initialize the OS.

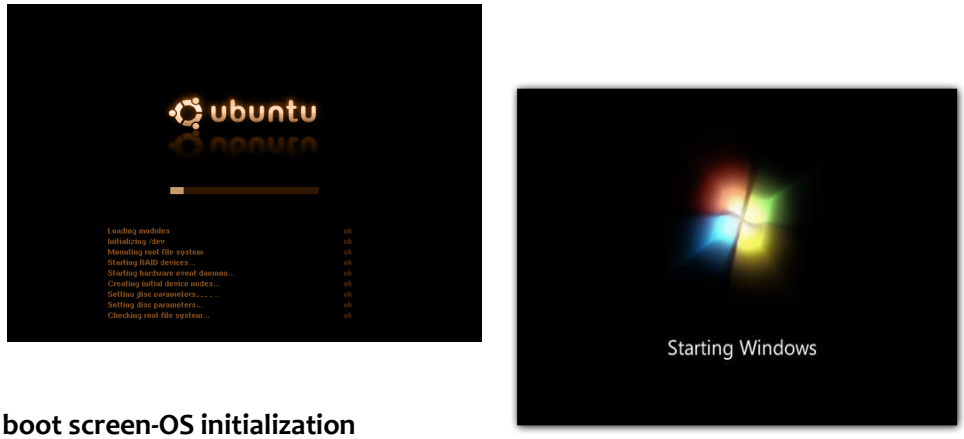


Figure 2.7: boot screen-OS initialization

Once the OS initializes, the BIOS copies its files into memory and the OS basically takes over control of the boot process. Now in control, the OS performs another inventory of the system's memory and memory availability (which the BIOS already checked) and loads the device drivers that it needs to control the peripheral devices, such as a printer, scanner, optical drive, mouse and keyboard. This is the final stage in the boot process, after which the user can access the system's applications to perform tasks.

Figure 2.8: Logon screen-entrance of OS

CHAPTER THREE

RESOURCE MANAGEMENT

Memory Management Concept and terminologies

Resident and transient

Fixed partitioning

Dynamic

Segmentation

Paging

Dynamic Address Translation

Virtual Memory concepts

Processor Management

Memory Management

Memory management is concerned with managing the computer's available pool of memory, allocating space to application routines, and making sure that they do not interfere with each other.

Resident and Transient

The operating system is a collection of software routines. Some routines, such as the ones that control physical I/O, directly support application programs as they run and thus must be **resident.** Others, such as the routine that formats disks, are used only occasionally. These **transient** routines are stored on disk and read into memory only when needed.

Generally, the operating system occupies low memory beginning with address 0. Key control information comes first (Figure 3.0), followed by the various resident operating system routines. The remaining memory, called the **transient area,** is where application programs and transient operating system routines are loaded.

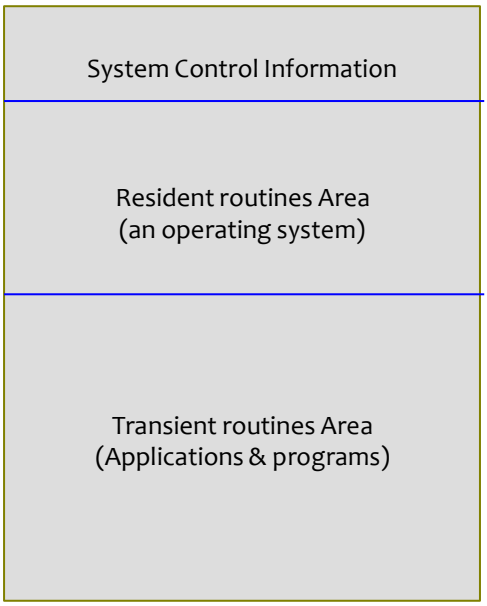


Figure 3.0: Resident and Transient Area

Memory Allocation Concepts

Contiguous Allocation

- Fixed-length partitioning

The simplest approach to managing memory for multiple, concurrent programs, **fixed-partition memory management** (Figure 3.1), divides the available space into fixed-length **partitions,** each of which holds one program. Partition sizes are generally set when the system is initially started, so the memory allocation decision is made before the actual amount of space needed by a given program is known.

Because the size of a partition must be big enough to hold the largest program that is likely to be loaded, fixed-partition memory management wastes space. Its major advantage is simplicity.

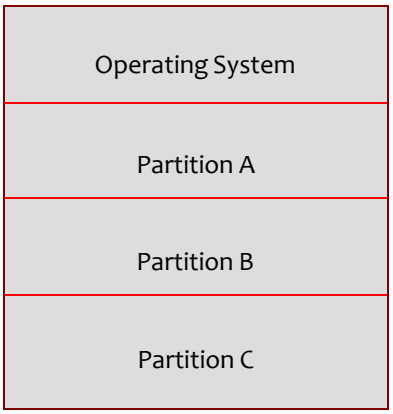


Figure 3.1: Fixed-partitioning allocation

- Dynamic region

Under **dynamic memory management,** the transient area is treated as a pool of unstructured free space. When the system decides to load a particular program, a **region** of memory just sufficient to hold the program is allocated from the pool. Because a program gets only the space it needs, relatively little is wasted.

Dynamic memory management does not completely solve the wasted space problem, however. Assume, for example, that a 640K program has just finished executing (Figure 3.2). If there are no 640K programs available, the system might load a 250K program and a 300K program, but note that 90K remains unallocated. If there are no 90K or smaller programs available, the space will simply not be used. Over time, little chunks of unused space will be spread throughout memory, creating a **fragmentation** problem.

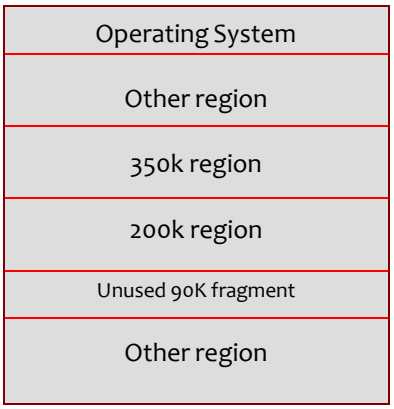


Figure 3.2: Dynamic region

Uncontiguous Allocation

- Segmentation

One reason for the fragmentation problem is that both fixed-partition and dynamic memory management assume that a given program must be loaded into contiguous memory. With **segmentation,** programs are divided into independently addressed segments and stored in noncontiguous memory (Figure 3.3). Segmentation requires adding a step to the address translation process. When a program is loaded into memory, the operating system builds a **segment table** listing the (absolute) entry point address of each of the program's segments (Figure 3.4). (Note that there is one segment table for each active program.) Later, when the operating system starts a given program, it loads the address of that program's segment table into a special register. As the program runs, addresses must be translated from relative to absolute form because programmers still write the same code and compilers still generate base-plus-displacement addresses. After fetching an instruction, the instruction control unit expands each operand address by adding the base register and the displacement. Traditionally, the expanded address was an absolute address. On a segmented system, however, the expanded address consists of two parts: a segment number and a displacement (Figure 3.4).

To convert the segment/displacement address to an absolute address, hardware:

1. checks the special register to find the program's segment table,
2. extracts the segment number from the expanded address,
3. uses the segment number to search the program's segment table,
4. finds the segment's absolute entry point address, and
5. adds the displacement to the entry point address to get an absolute address

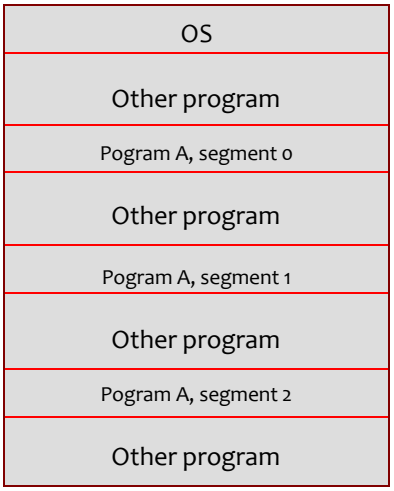


Figure 3.3: Segmentation

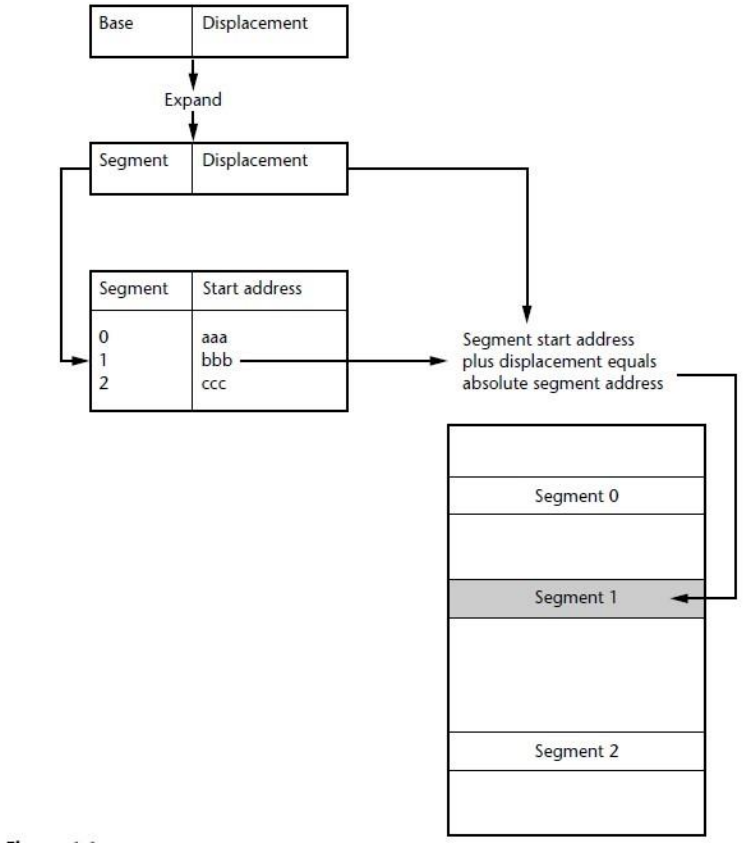


Figure 3.4: Segmentation Allocation with Dynamic Address Translation

- Paging Allocation

A program's segments can vary in length. Under **paging,** a program is broken into fixed-length pages. Page size is generally small (perhaps 2K to 4K), and chosen with hardware efficiency in mind.

Like segments, a program's pages are loaded into noncontiguous memory. Addresses consist of two parts (Figure 6.7), a page number in the high-order positions and a displacement in the low-order bits. Addresses are dynamically translated as the program runs. When an instruction is fetched, its base-plus displacement addresses are expanded to absolute addresses by hardware. Then the page's base address is looked up in a program **page table** (like the segment table, maintained by the operating system) and added to the displacement.

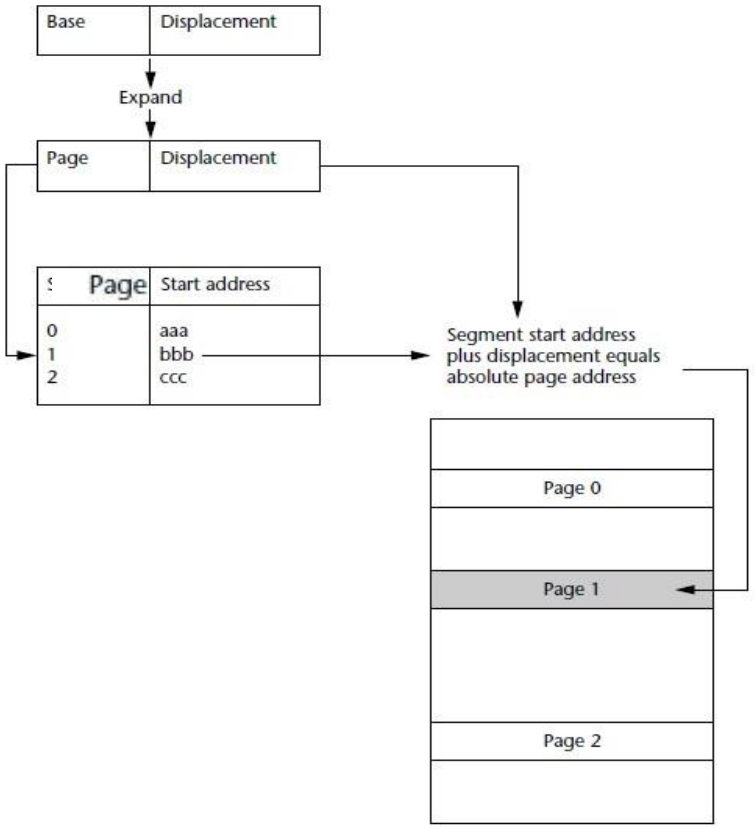


Figure 3.5: Paging Allocation mechanism

Dynamic Address Translation

Dynamic address translation, or DAT, is the process of translating a virtual address during a storage reference into the corresponding real address. If the virtual address is already in central storage, the DAT process may be accelerated through the use of a translation look aside buffer. If the virtual address is not in central storage, a page fault interrupt occurs, OS is notified and brings the page in from auxiliary storage.

Looking at this process more closely reveals that the machine can present any one of a number of different types of faults. A type, region, segment, or page fault will be presented depending on at which point in the DAT structure invalid entries are found. The faults repeat down the DAT structure until ultimately a page fault is presented and the virtual page is brought into central storage either for the first time (there is no copy on auxiliary storage) or by bringing the page in from auxiliary storage.

DAT is implemented by both hardware and software through the use of page tables, segment tables, region tables and translation look aside buffers. DAT allows different address spaces to share the same program or other data that is for read only. This is because virtual addresses in different address spaces can be made to translate to the same frame of central storage. Otherwise, there would have to be many copies of the program or data, one for each address space.

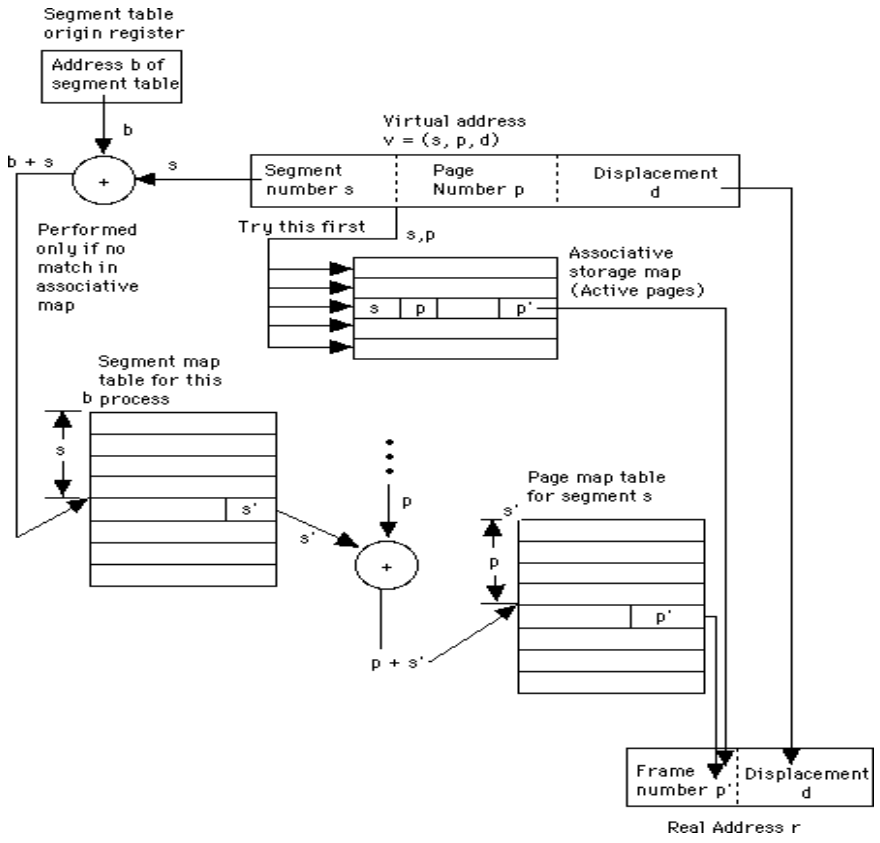


Figure 3.6 : A DAT in segmentation and paging allocation (combined associative/direct mapping)

Virtual Memory Mechanism

If a processor can execute only one instruction at a time, why is it necessary for every instruction in a program to be in memory before that program can begin executing? It isn't. You have already considered overlay structures in which only portions of a program are in memory at any given time. Those early overlay structures were precursors of modern **virtual memory** systems.

The word virtual means "not in actual fact." To the programmer or user, virtual memory acts just like real memory, but it isn't real memory.

Figure 3.7 illustrates a common approach to implementing virtual memory. It shows three levels of storage—virtual memory, the external paging device, and real memory. **Real memory** is good, old-fashioned main memory, directly addressable by the processor. The **external paging device** is usually disk. Virtual memory is a model that simplifies address translation. It "contains" the operating system and all the application programs, but it does not physically exist anywhere. Its contents are physically stored in real memory and on the external paging device.

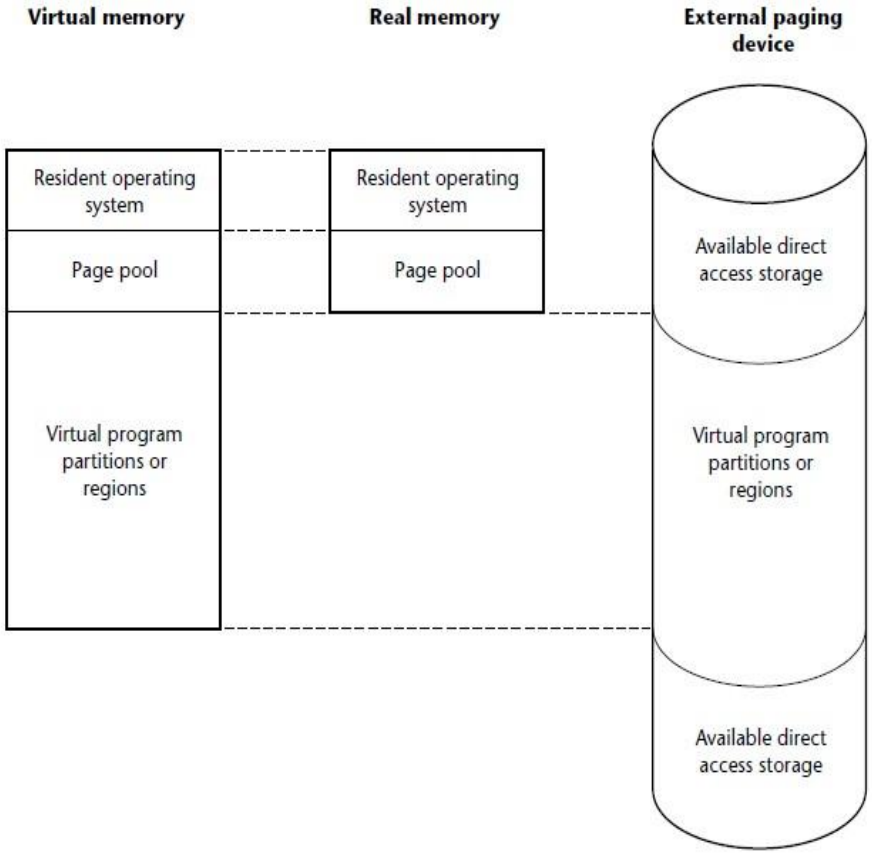


Figure 3.7 : Virtual Memory model

Virtual memory is divided into two components. The first is exactly equal to the amount of real memory on the system and is physically stored in real memory. It contains the resident operating system and the transient program area (called the **page pool**).

The second component of virtual memory consists of space over and above real memory capacity. It is physically stored on the external paging device and contains the application programs. The operating system is loaded into real memory. Application programs are loaded onto the external paging device. Selected pages are then swapped between the real memory page pool and the external paging device (Figure 3.8).

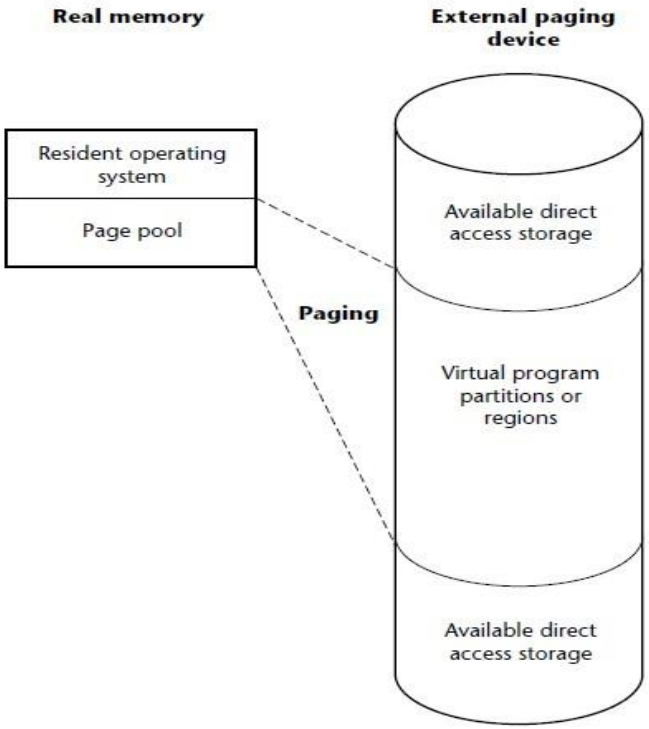


Figure 3.8: Swapping process

Traditionally, the operating system's memory management routine was concerned with allocating real memory space. On a virtual memory system, an equivalent module allocates space on the external paging device. This space can be divided into fixed-length partitions, variable-length regions, segments, pages, or any other convenient unit. Swapping pages between the external paging device and real memory is a system function and thus is transparent to the user.

Addressing in Virtual Memory

The instructions that run on a virtual memory system are identical to the instructions that run on a regular system. The operands hold relative (base-plus-displacement) addresses. Thus, immediately after an instruction is fetched, the instruction control unit expands the address by adding the displacement to the contents of a base register. On a regular system, the base register holds the program's load point in real memory. On a virtual system, however, the base register holds the program's load point in virtual memory, so the computed address reflects the page's virtual memory location.

The dynamic address translation process (which resembles segmentation and paging addressing, see Figure 6.8) starts when the program is loaded into virtual memory. The operating system allocates space on the external paging device and notes the virtual addresses of the program's segments and pages in the program's segment and page tables. Later, when a given page is swapped into real memory, the page's real address is noted in the page table.

Note that a page must be in real memory for the processor to execute its instructions. When an instruction executes, the instruction control unit (in its usual way) adds the base register and the displacement to get an address in virtual memory.

To convert a virtual address to a real address, hardware:

1. accesses the program's segment table using the high-order bits of the virtual address as a key,
2. locates the program's page table using the pointer in the segment table,
3. accesses the page table to find the page's real base address using the middle bits in the virtual address as a key, and
4. adds the displacement found in the low-order bits of the virtual address to the page's real memory base address.

On most systems, the process is streamlined through the use of special registers and other hardware.

Page Faults

When a virtual address points to a page that is not in real memory, a **page fault** is recognized and a page-in (or swap-in) operation begins. If no real memory is available for the new page, some other page must be swapped out. Often the "least currently accessed" page (the page

that has gone the longest time without being referenced) is selected.

Bringing pages into memory only when they are referenced is called **demand paging.** An option called **pre-paging** involves predicting the demand for a new page and swapping it into memory before it is actually needed. Many pre-paging algorithms assume that segments hold logically related code, so if the instructions on page 1 are currently executing, the chances are that the instructions on page 2 will be executed next. While far from perfect, such techniques can significantly speed up program execution.

Thrashing

When real memory is full, a demand for a new page means that another page must be swapped out. If this happens frequently, the system can find itself spending so much time swapping pages into and out from memory that little time is left for useful work.

This problem is called **thrashing,** and it can seriously degrade system performance. The short-term solution is removing a program or two from real memory until the system settles down. The long-term solution is to improve the real-to-virtual ratio, usually by adding more real memory.

Processor Management

The Dispatcher

Imagine two programs concurrently occupying memory. Some time ago, program A requested data from disk (Figure 6.12). Because it was unable to continue until the input operation was completed, it dropped into a **wait state** and the processor turned to program B. Assume the input operation has just been completed. Both programs are now in a **ready state;** in other words, both are ready to resume processing.

Which one goes first? Computers are so fast that a human operator cannot effectively make such real-time choices. Instead, the decision is made by an operating system routine called the **dispatcher.**

Consider a system with two partitions: foreground and background. The dispatcher typically checks the program in the foreground partition first. If the program is ready, the dispatcher restarts it. Only if the foreground program is still in a wait state does the dispatcher check the background partition. The foreground has high priority; the background has low priority.

This idea can be extended to larger systems with multiple concurrent programs in memory. The dispatcher checks partitions in a fixed order until a ready state program is found. The first partition checked has highest priority; the last has lowest priority. The only way the low-priority program can execute is if all the higher priority partitions are in a wait state.

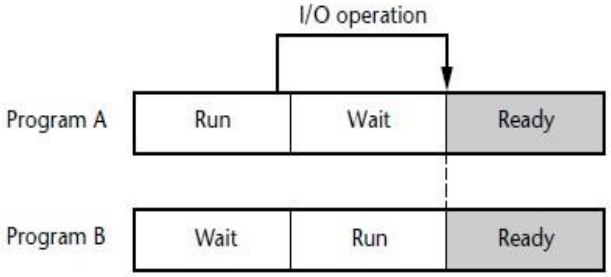


Figure 3.9 : Both in ready state-the dispatcher will decide

Control Block

There are several control fields that must be maintained in support of each active program. Often, a **control block** is created to hold a partition's key control flags, constants, and variables (Figure 3.10). The control blocks (one per partition) are linked to form a linked list. The dispatcher typically determines which program is to start by following the chain of pointers from control block to control block. A given control block's relative position in the linked list might be determined by its priority or computed dynamically, perhaps taking into account such factors as program size, time in memory, peripheral device requirements, and other measures of the program's impact on system resources.

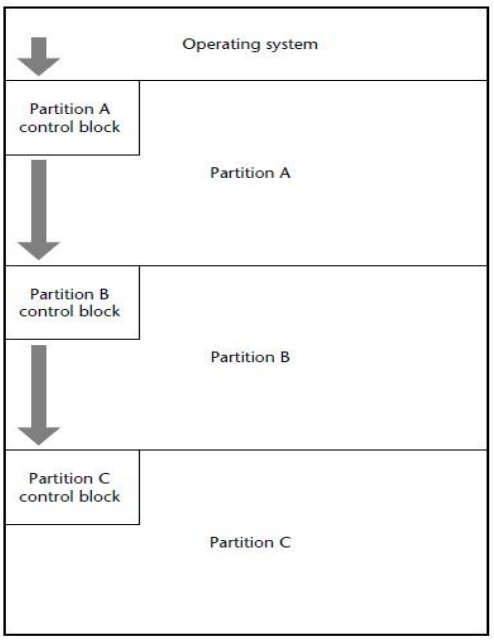


Figure 3.10 : Control Blocks mechanism

Interrupts

A program normally surrenders control of the processor when it requests an I/O operation and is eligible to continue when that I/O operation is completed. Consequently, the key to multiprogramming is recognizing when input or output operations begin or end. The operating

system knows when these events occur because they are marked by interrupts.

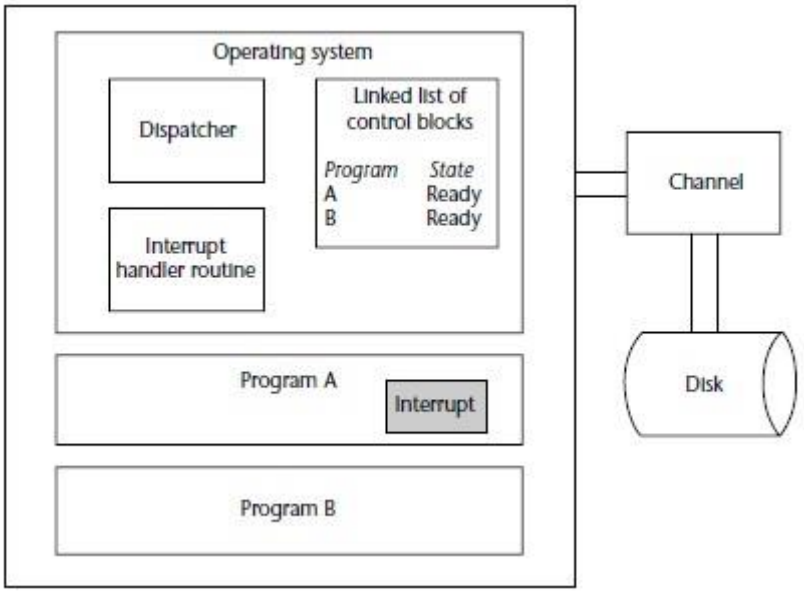
An interrupt is an electronic signal. Hardware senses the signal, saves key control information for the currently executing program, and starts the operating system's **interrupt handler** routine. At that instant, the interrupt ends. The operating system then handles the interrupt. Subsequently, after the interrupt is processed, the dispatcher starts an application program. Eventually, the program that was executing at the time of the interrupt resumes processing.

For example, follow the steps in Figure 3.11. When an application program needs data, it issues an interrupt (Figure 3.11a). In response, hardware starts the interrupt handler routine, which saves key control information, drops the application program into a wait state (Figure 3.11b), and calls the input/output control system to start the I/O operation. Finally, control flows to the dispatcher, which starts a different program (Figure 3.11c).

Later, when the I/O operation is finished, the channel issues an interrupt (Figure 3.11d). Once again the interrupt handler routine begins executing (Figure 3.11e). After verifying that the I/O operation was successfully completed, it resets the program that initially requested the data (program A) to a ready state. Then it calls the dispatcher, which starts the highest priority "ready" application program (Figure 3.11f). In this example, note that program A goes next even though program B was running at the time the interrupt occurred.

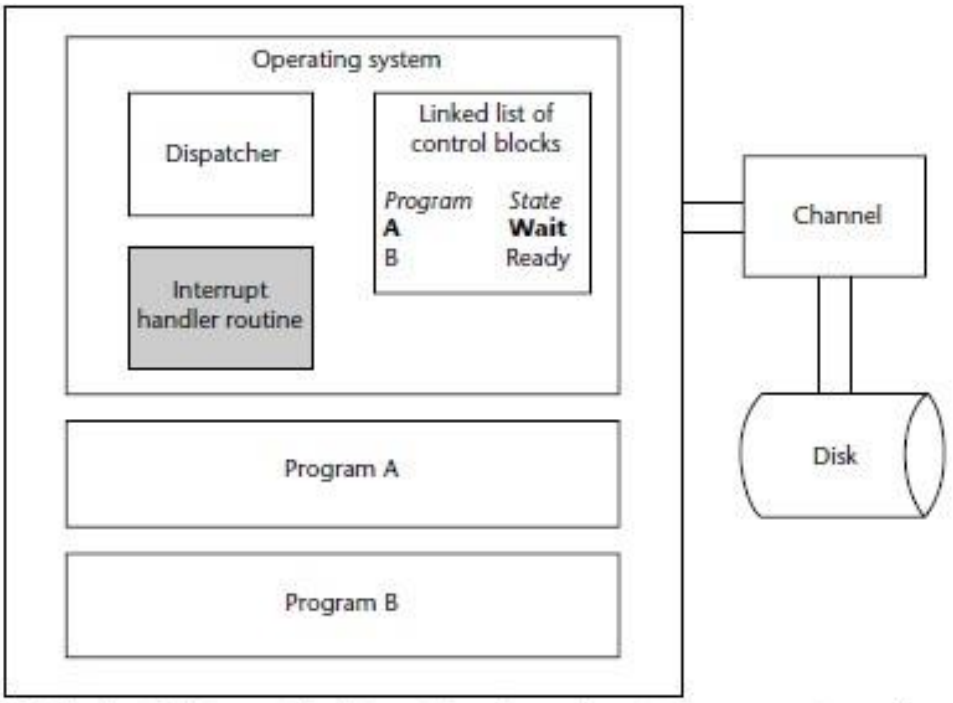
Interrupts can originate with either hardware or software. A program issues an interrupt to request the operating system's support (for example, to start an I/O operation). Hardware issues an interrupt to notify the processor that an asynchronous event (such as the completion of an I/O operation or a hardware failure) has occurred. Other types of interrupts might signal an illegal operation (a zero divide) or the expiration of a preset time interval.

Interrupts mark events. In response, hardware starts the interrupt handler routine, which performs the appropriate logical functions and, if necessary, resets the affected program's state (wait, ready). Following each interrupt, the dispatcher starts the highest priority ready program. That, in a nutshell, is the essence of multiprogramming.

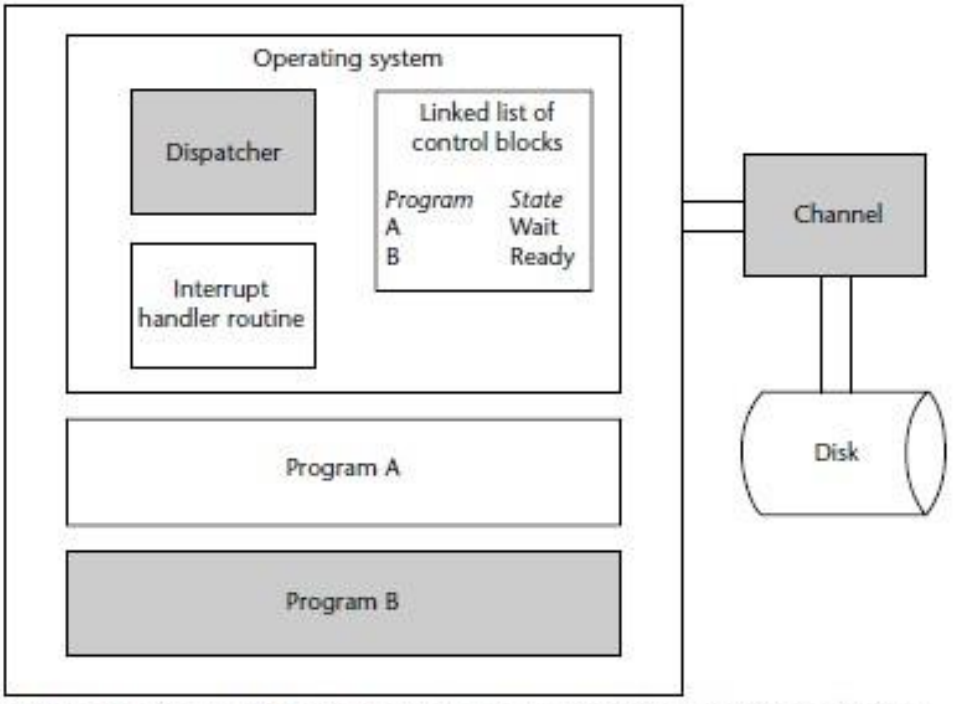


a. program requests the OS support by issuing an interrupts

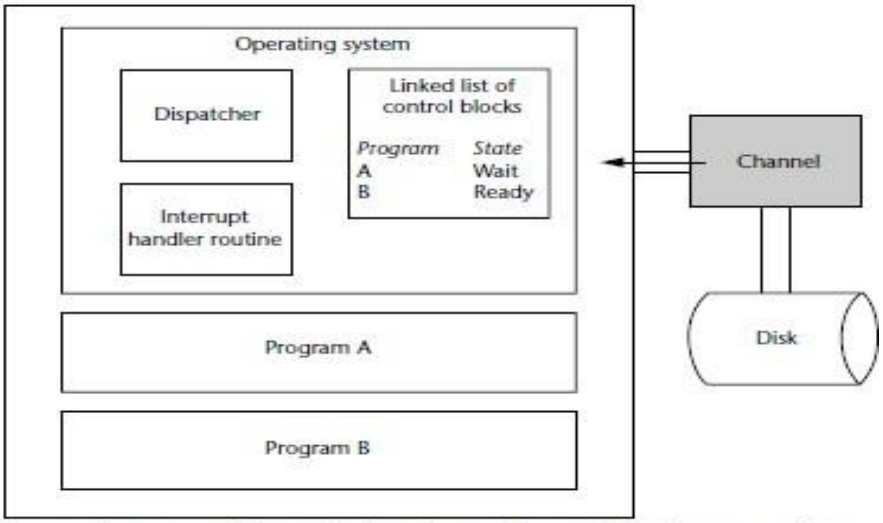
Figure 3.11-(a,b,c,d,e,f) – explanation of interrupts mechanism in processor management



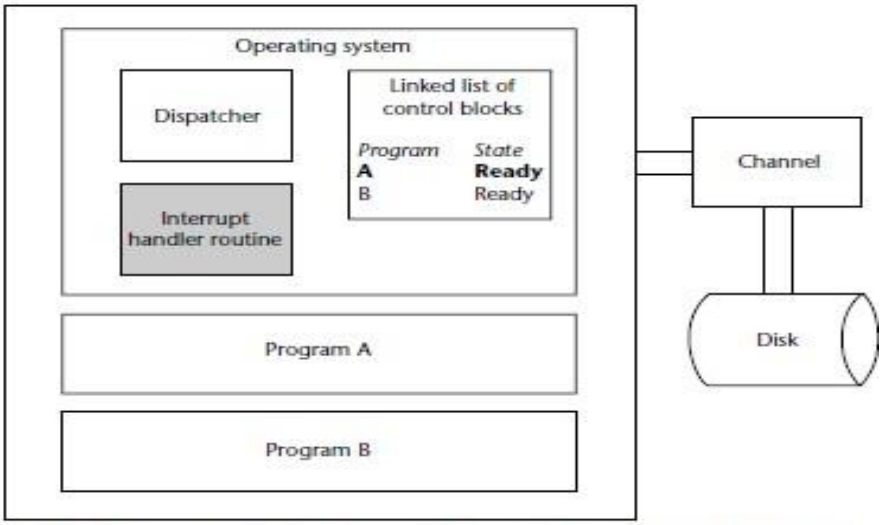
b. Following the interrupt, the interrupt handler routine sets the program to a wait state.



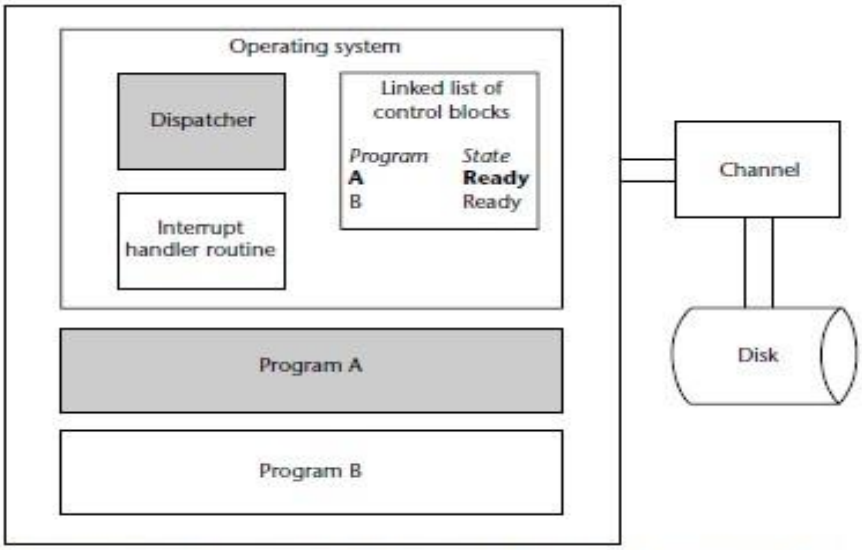
c. After the interrupt handler routine starts the requested input or output operation, the dispatcher starts another application program.



d. Several microseconds later, the channel signals the end of the I/O operation by sending the processor an interrupt.



e. Following the interrupt, the interrupt handler routine resets program A to a ready state.



f. After the interrupt is processed, the dispatcher selects an application program and starts it.

Processor/CPU Scheduling

Schedulling Types

In general, (job) scheduling is performed in three stages:

- Short-term scheduling
- Medium scheduling
- long-term scheduling

These types of scheduler can be applied by processor to complete the task.

***Short-term* (process or CPU) scheduling** occurs most frequently and decides which process to execute next. Short-term scheduler, also known as the process or CPU scheduler, controls the CPU sharing among the "ready" processes. The selection of a process to execute next is done by the short-term scheduler.

Usually, a new process is selected under the following circumstances:

- When a process must wait for an event.
- When an event occurs (e.g., I/O completed, quantum expired).
- When a process terminates.

The goal of short-term scheduling is to optimize the system performance, and yet provide responsive service. In order to achieve this goal, the following set of criteria is used:

- CPU utilization
- I/O device throughput
- Total service time
- Responsiveness
- Fairness
- Deadlines

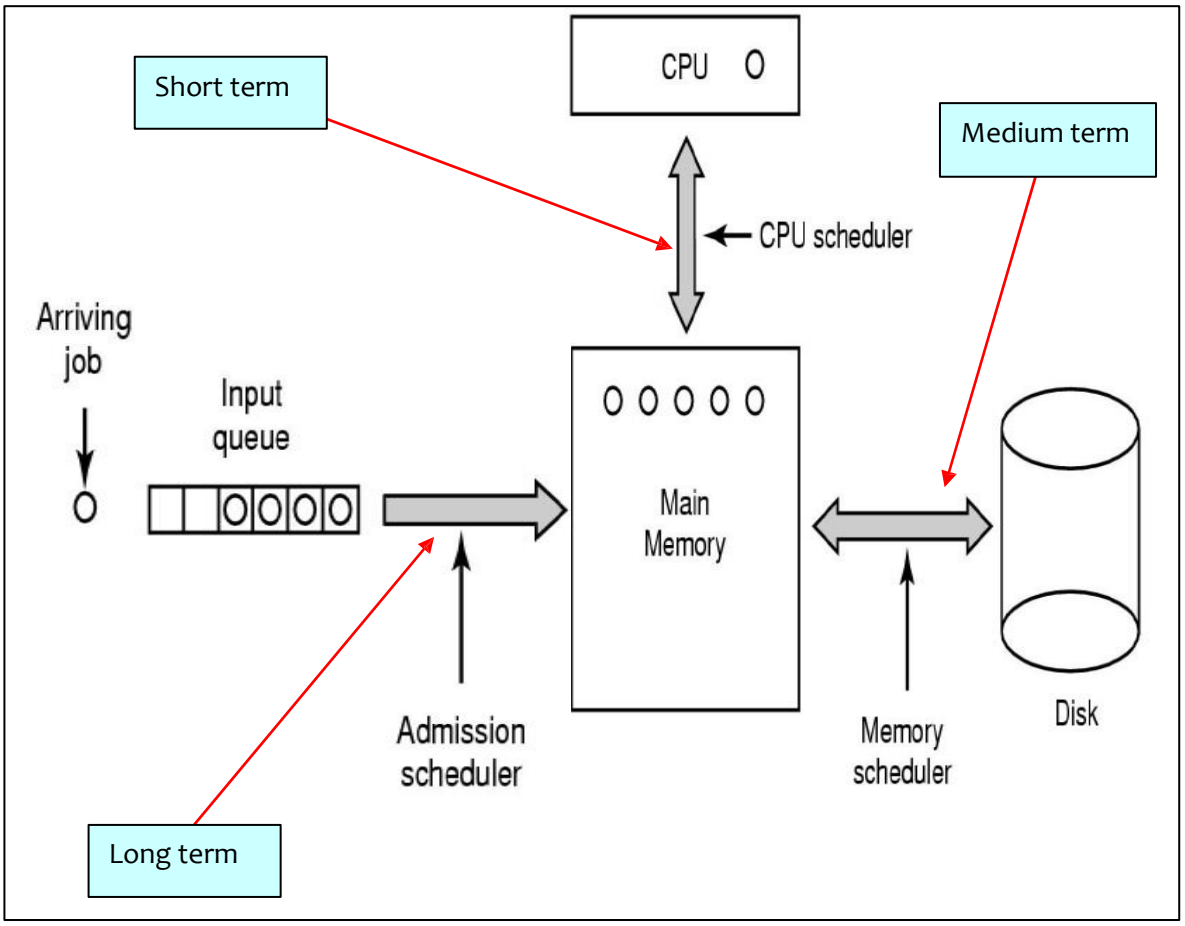
Medium-term scheduling involves suspending or resuming processes by swapping (rolling) them out of *or* into memory.

Long-term (job) scheduling is done when a new process is created. It initiates processes and so controls the *degree of multi-programming* (number of processes in memory).

Acting as the primary resource allocator, the longterm scheduler admits more jobs when the resource utilization is low, and blocks the incoming jobs from entering the ready queue when

utilization is too high. When the main memory becomes over-committed, the medium-term scheduler releases the memory of a suspended (blocked or stopped) process by swapping (rolling) it out.

In summary, both schedulers control the level of multiprogramming and avoid (as much as possible) overloading the system by many processes and cause "thrashing"



Figuren3.12: Types of scheduling – short-term, medium-term and long-term

Scheduling Algorithm

In general, scheduling policies may be *preemptive* or *non-preemptive*.

In a non-preemptive pure multiprogramming system, the short-term scheduler lets the current process run until it blocks, waiting for an event or a resource, or it terminates. Preemptive policies, on the other hand, force the currently active process to release the CPU on certain events, such as a clock interrupt, some I/O interrupts, or a system call.

The following are some common scheduling algorithms:

Non-preemptive

- First-Come-First-Served (FCFS) or often called as First-In,First-Out(FIFO)
- Shortest Job first (SJF)
- Shortest Remaining Time

Good for ' 'background' ' batch jobs.

Preemptive

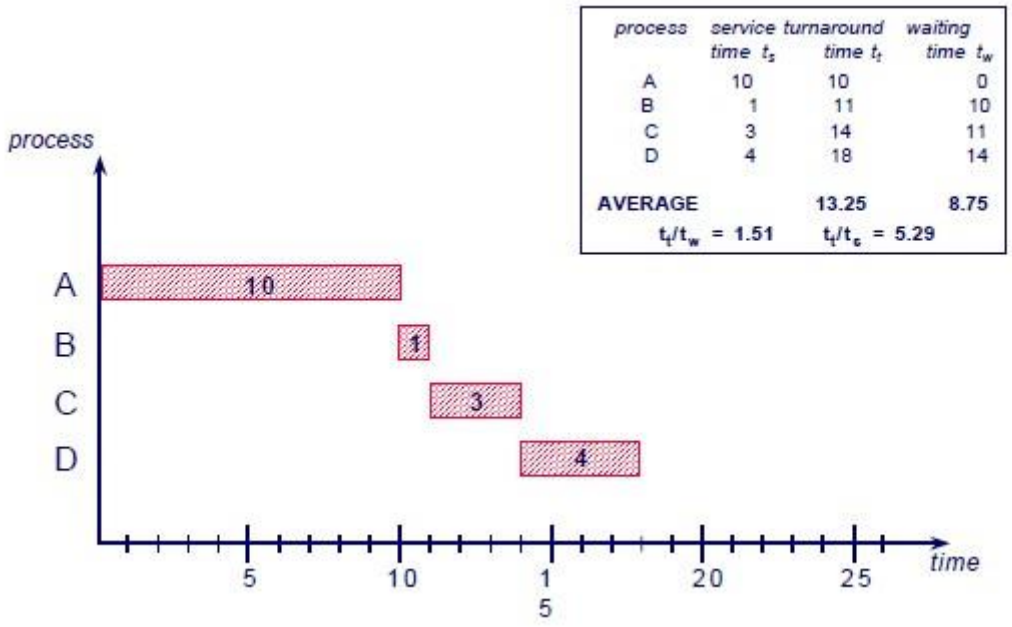
- Round-Robin (RR)
- Priority

Good for ' 'foreground' ' interactive jobs.

First-Come-First-Serve/ First-In-First-Out Scheduling

FCFS, also known as First-In-First-Out (FIFO), is the simplest scheduling policy. Arriving jobs are inserted into the tail (rear) of the ready queue and the process to be executed next is removed from the head (front) of the queue.

FCFS performs better for long jobs. Relative importance of jobs measured only by arrival time (poor choice). A long CPU-bound job may hog the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods. This in turn may lead to a lengthy queue of ready jobs, and thence to the "convoy effect."

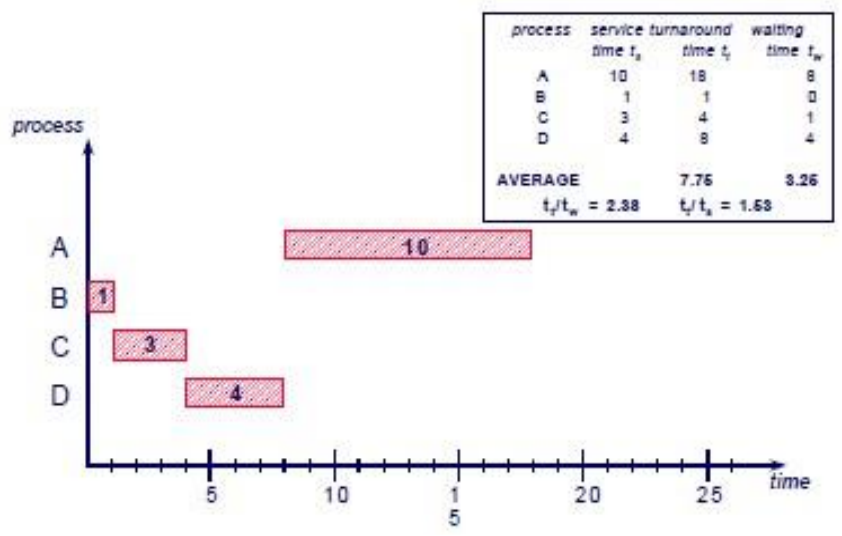


process	service time t_s	turnaround time t_t	waiting time t_w
A	10	10	0
B	1	11	10
C	3	14	11
D	4	18	14
AVERAGE		13.25	8.75
	t_t/t_w = 1.51	t_t/t_s = 5.29	

Figure 3.13: A FIFO/FCFS process

Shortest Job First(SJF) scheduling

SJF policy selects the job with the shortest (expected) processing time first. Shorter jobs are always executed before long jobs. One major difficulty with SJF is the need to know or estimate the processing time of each job (can only predict the future!) Also, long running jobs may starve, because the CPU has a steady supply of short jobs. In a case that has a lot of equal shortest job queuing, the FCFS algorithm will be take part.



process	service time t_s	turnaround time t_r	waiting time t_w
A	10	18	8
B	1	1	0
C	3	4	1
D	4	8	4
AVERAGE		7.75	3.25
	t_r/t_w = 2.38	t_r/t_s = 1.63	

Figure 3.14: a Shortest-Job First process

Shortest Job Remaining Time scheduling

Shortest remaining time is a method of CPU scheduling that is a preemptive [defensive] version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added, though this threat can be minimal when process times follow a heavy-tailed distribution.

Like shortest job next scheduling, shortest remaining time scheduling is rarely used outside of

specialized environments because it requires accurate estimations of the runtime of all processes that are waiting to execute. It doesn't use RR if two processes are of the same length because of overhead space

Round-Robin Scheduling

Round-robin scheduling is an algorithm used to assist in creating process or job schedules to ensure that each process required to complete a job gets an ample amount of run time. CPUs in computers can use time slicing to provide a set amount of time for each process to use per cycle. Using round-robin scheduling allots a slice of time to each process that is running. In a computer for example, the user starts three applications, Email, a web browser, and a word processor. These applications are loaded into system memory as processes and each is allowed to run without the user considering which applications are running in the background.

Round-robin scheduling handles the sharing of resources between the three application processes (and the countless others running in the background completely invisible to the user). This scheduling works well because each application gets a certain amount of time per processor cycle. A processor cycle is the amount of time it takes the CPU to manage each process running, one time.

The running applications in the earlier example provide a short cycle for the processor and more time would be allotted to each of these three processes, making them appear to perform better to the end user. Without round robin scheduling, the application loaded first into memory would likely monopolize the processor until it was finished performing any of the tasks it had been assigned. When that application closed, the next application could start and process without interruption. This would get in the way of the multi-window environments on which computer users have come to depend. The use of round-robin scheduling helps the computer keep up with the end user and effectively manage all three application processes.

Round-robin scheduling keeps all of the running jobs or processes progressing forward a little bit at a time, during each processor cycle, to help them all run together and improve the usability experience for the person working with the system. The CPU will then poll each task that is running during a cycle to help determine if the process has finished.

Suppose the user decides that they have completed their work in the word processor application and close it. This leaves only the e-mail and web browser applications running. The CPU would have no way of keeping track of this fact without round-robin scheduling to poll the applications and discover that the word processor has closed and is no longer needing any processor time.

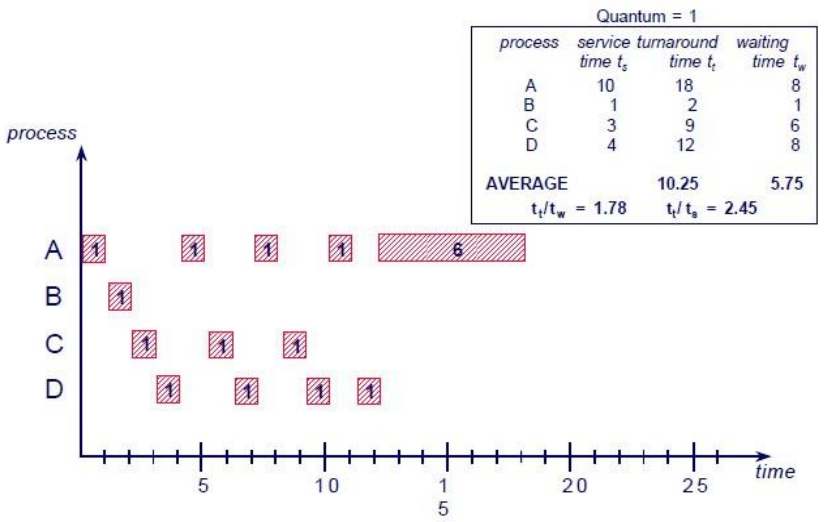


Figure 3.15: A Round Robin process

Priority Scheduling

Priorities may be static or dynamic. Static priorities are assigned at the time of creation, while dynamic priorities are based on the processes' behavior while in the system. For example, the scheduler may favor I/O-intensive tasks so that expensive requests can be issued as early as possible.

A danger of priority scheduling is starvation, in which processes with lower priorities are not given the opportunity to run. In order to avoid starvation, in preemptive scheduling, the priority of a process is gradually reduced while it is running. Eventually, the priority of the running process will no longer be the highest, and the next process will start running. This method is called aging.

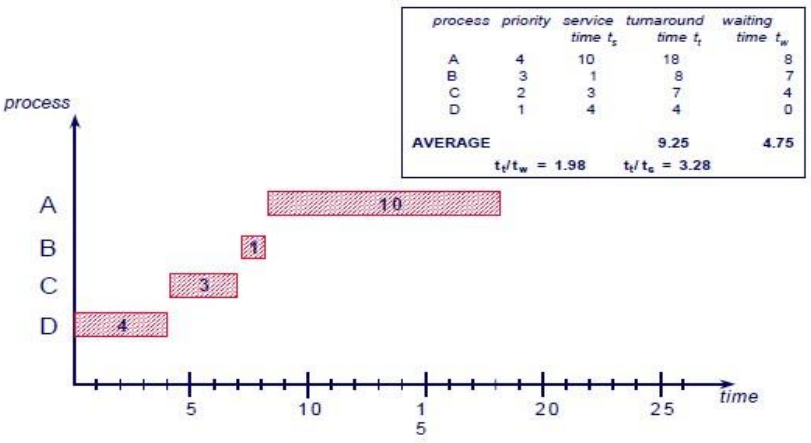


Figure 3.16: Priority Scheduling

Scheduling Algorithm Comparison

Unfortunately, the performance of scheduling policies vary substantially depending on the characteristics of the jobs entering the system (job mix), thus it is not practical to make definitive comparisons.

For example, algorithm of First Comes First Serve(FCFS)/First In Forst Out(FIFO) performs better for "long" processes and tends to favor CPU-bound jobs.

Whereas Shortest Job First(SJF) is risky, since long processes may suffer from CPU starvation. Furthermore, FCFS/FIFO is not suitable for ' 'interactive'' jobs, and similarly, Round Robin(RR) is not suitable for long "batch" jobs. The (processing) overhead of FCFS/FIFO is negligible, but it is moderate in RR and can be high for SJF.

As mentioned earlier, the previous policies cannot efficiently handle a mixed collection of jobs (e.g., batch, interactive, and CPU-bound). So, there are developed some else algorithms to handle this problem. One of them is known as Multi-level queue scheduling.

Multilevel Queue Algorithm

Multi-Level Queue (MLQ) scheme solves the mix job problem by maintaining separate "ready" queues for each type of job class and apply different scheduling algorithms to each. Figure 3.17 show the state of MLQ scheduling algorithm. A MLQ 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 choose the process from the occupied queue that has the highest priority, and run that process either Preemptive or Non-preemptively

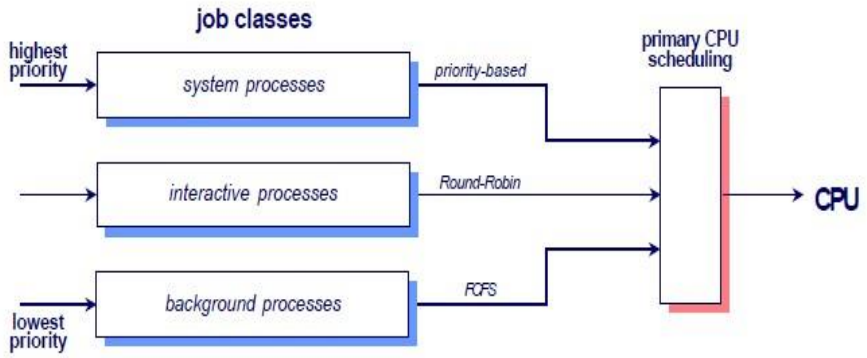


Figure 3.17 : Multilevel Queue Algorithm

Queuing Routines and Scheduler to load application

Processor management is concerned with the *internal* priorities of programs already in memory. A program's *external* priority is a different issue. As one program finishes processing and space becomes available, which program is loaded into memory next? This decision typically involves two separate modules, a queuing routine and a scheduler.

As programs enter the system, they are placed on queue by the queuing routine as shown in Figure 3.18. When space becomes available, the scheduler selects a program from the queue and loads it into memory. Clearly distinguish between a program's internal and external priorities. Once a program is in memory, the dispatcher uses its *internal* priority to determine its right to access the processor. In contrast, the program's *external* priority has to do with loading it into memory in the first place. Until the program is in memory, it has no internal priority. Once in memory, its external priority is no longer relevant.

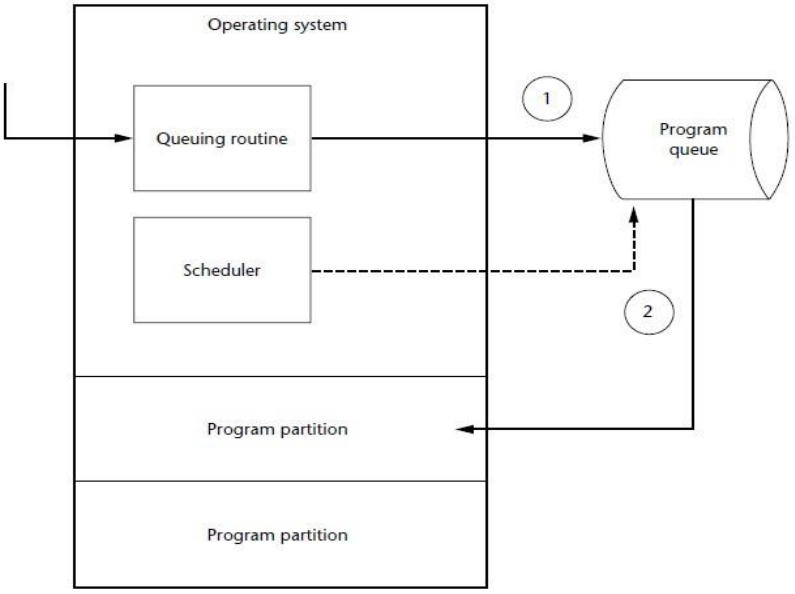


Figure 3.18 : Queuing routines and scheduler concept

Multiprogramming vs Time Sharing

As mentioned earlier, *Multiprogramming* is a common approach to processor management when two or more programs occupy memory and execute concurrently. Originally developed to support batch-processing applications, multiprogramming takes advantage of the extreme speed disparity between a computer and its peripheral devices. Traditionally, the key measures of effectiveness are throughput (run time divided by elapsed time) and turnaround (the time between job submission and job completion).

Time-sharing is a different approach to managing multiple concurrent users designed with interactive processing in mind. The most important measure of effectiveness is response time, the elapsed time between entering a transaction and seeing the system's response appear on the screen. Note, however, that timesharing and multiprogramming are not mutually exclusive. In fact, it is not uncommon for an interactive, time-sharing system to run in the high priority partition on a large, multiprogramming mainframe.

DeadLock

Deadlock is one possible consequence of poor resource management. Imagine, for example, that two programs need data from the same disk. Program A issues a seek command and drops into a wait state. Subsequently, program B begins executing and issues its own seek command. Eventually, the first seek operation is completed and the dispatcher starts program A, which issues a read command. Unfortunately, the second seek command has moved the access mechanism, so A must reissue its seek command and once again drop into a wait state. Soon B issues its read command, discovers that the access mechanism is in the wrong place, and reissues its seek command.

Consider the outcome of this nightmare. Program A positions the access mechanism. Program B moves it. Program A repositions it; program B does the same thing. Picture the access mechanism moving rapidly back and forth across the disk's surface. No data are read or written. Neither program can proceed. The result is deadlock.

Deadlock is not limited to peripheral devices. It happens when two (or more) programs each control *any* resource needed by the other. Neither program can continue until the other "gives in," nor if neither is willing to give in, does the system, almost literally, "spin its wheels." At best, that leads to inefficiency. At worst, it can bring the entire system to a halt.

One solution is prevention; some operating systems will not load a program unless all its resource needs can be guaranteed. Other operating systems allow some deadlocks to occur, sense them, and take corrective action.

Summarily, deadlock occurs when two programs each control a resource needed by the other but neither is willing to give up its resource. Some operating systems are designed to prevent deadlock. Others sense deadlock and take corrective action

CHAPTER FOUR

FILE MANAGEMENT

File Management

File System

Files

File Naming

File Structure

File Operations

File System Implementation

Implementing Files

Contiguous Allocation

Linked List Allocation

Linked List Allocation Using Index

i-Nodes

Caching enhances performance of FS

"The term OS file management refers to the manipulation of documents and data in files on a computer"-(wikipedia)

File Systems

Introduction

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications, (e.g., for data bases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process. A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an on-line telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process. Thus we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called files. Processes can then read them and write new ones if need be. Information stored in files must be persistent, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it. Files are managed by the operating system. How they are structured, named, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the file system and is the subject of this chapter.

From the users' standpoint, the most important aspect of a file system is how it appears to them, that is, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bit maps are used to keep track of free *storage* and how many sectors there are in a logical block are of less interest, although they are of great importance to the designers of the file system, For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed discussion of how the file system is implemented. After that we will look at security and protection mechanisms in file systems.

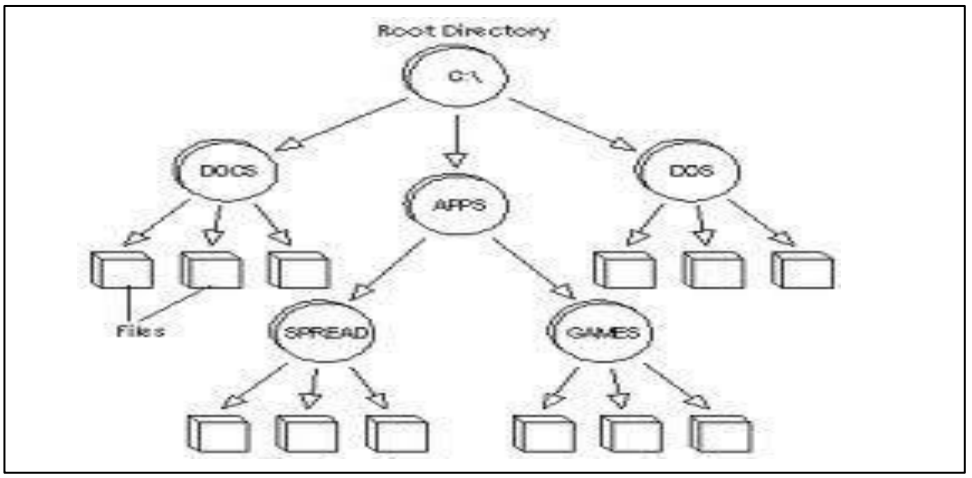


Figure 4-0 : Structure of file system concept

Files

In this section we will look at files from the user's point of view, that is, how they are used and what properties they have.

File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work. Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of file naming. When a process creates a, file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all operating systems allow strings of one to eight letters as legal file names. Many file systems support names as long as 255 characters. Some of the more common file extensions and their meanings are shown in Figure 4- 1.

Extension	Meaning
file-bak	Backup file
file.c	C source program
file.gif	Graphical Interchange Format image
file.help	Help file
file.html	World wide web Hyper Text
file.mpg	Movie encoded with MPEG standard
file.o	Object file
file.ps	PostScript file
file.txt	General text file
file.zip	Compressed archive

Figure 4.1 : Some typical file extensions.

File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 4-2. The file in Fig. 4-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs.

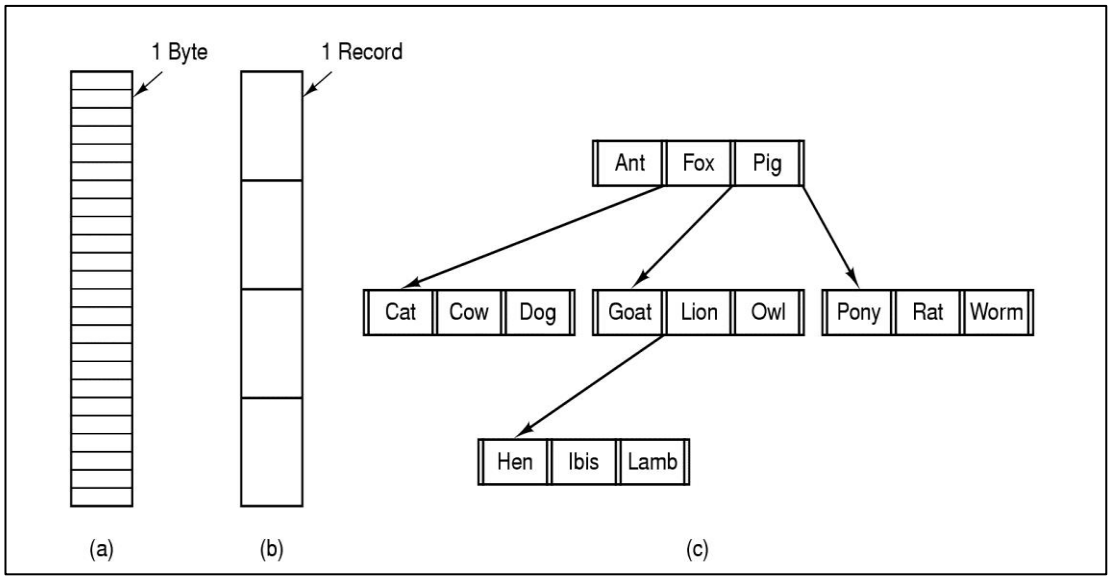


Figure 4-2:-Three kinds of files. (a) Byte sequence. (b) Record sequence. (c)*Tree*.

Having the operating system regard files as nothing more than byte sequences provides the maximum flexibility. User programs can put anything they want in files and name them any way that is convenient. The operating system does not help. but it also does not get in the way. For users who want to do unusual things, the latter can be very important. The first step up in structure is shown in Fig. 4-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operations overwrites or appends one record. In years gone by when the 80-column punched card was king, many operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of
80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course.

The third kind of file structure is shown in Fig. 4-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files as shown in Table 4.0

Operations	Description
1. CREATE	The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. DELETE	When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. OPEN	Before using a file, a process must open it. The purpose of the OPEN call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. CLOSE	When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal

	table space. Many systems encourage this by imposing a maximum number of open files on processes.
5. READ	Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.
6. WRITE	Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. APPEND	This call is a restricted form of WRITE. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have APPEND, but many systems provide multiple ways of doing the same thing, and these systems sometimes have APPEND.
8. SEEK	For random access files, a method is needed to specify from where to take the data. One common approach is a system call, SEEK, that repositions the pointer to the current position to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
9. GET ATTRIBUTES	Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.
10. SET ATTRIBUTES.	Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.
11. RENAME.	It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

Table 4.0 : File Operation Attributes

File System Implementation

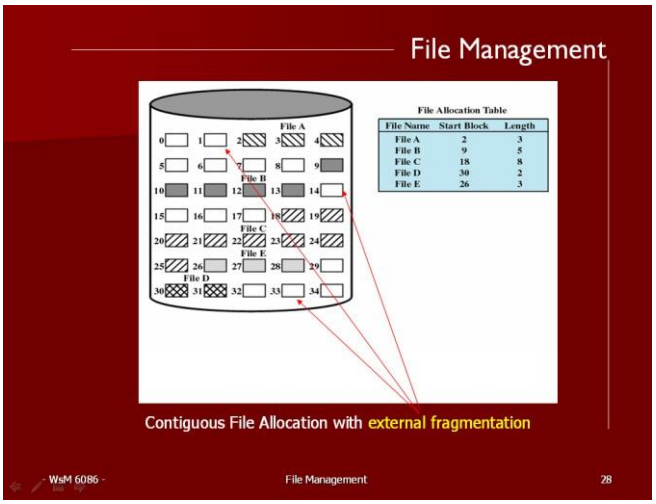
Now it is time to turn from the user's view of the file system to the implementer's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementers are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

Files Implementation

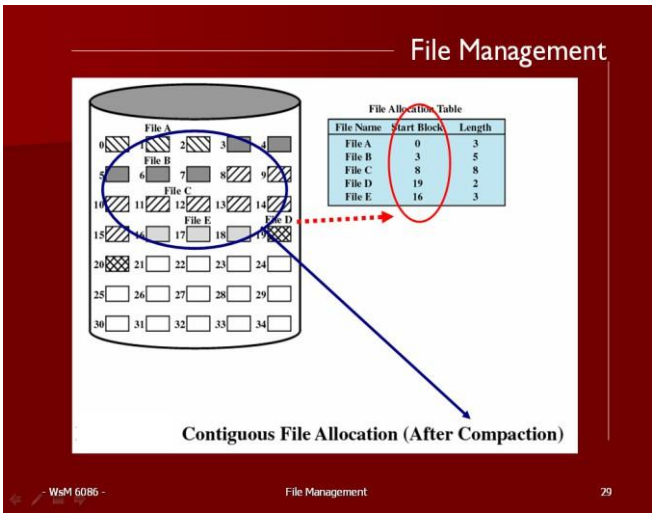
Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous block of data on the disk. Thus on a disk with 1K blocks, a 50K file would be allocated 50 consecutive blocks. This scheme has two *significant* advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering one number, the disk address of the first block. Second, the performance is excellent because the entire file can be read from the disk in a single operation. Unfortunately, contiguous allocation also has two equally significant drawbacks. First, it is not feasible unless the maximum file size is known at the time the file is created. Without this information, the operating system does not know how much disk space to reserve. In systems where files must be written in a single blow, it can be used to great advantage, however. The second disadvantage is the fragmentation of the disk that results from this allocation policy. Space is wasted that might otherwise have been used. Compaction of the disk is usually prohibitively expensive, although it can conceivably be done late at night when *the* system is otherwise idle.



(a)



(b)

Figure 4.3: Contiguous Allocation ((a) before and (b) after compaction)

Advantages

- Simple to implement (start block & length is enough to define a file)
- Fast access as blocks follows each other

Disadvantages

- Does not allow simple expansion of files
- Risk of external fragmentation
- Solving the fragmentation problem requires compaction, a time consuming process
- The kernel must allocate and reserve contiguous space when the file is first created

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 4-4. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

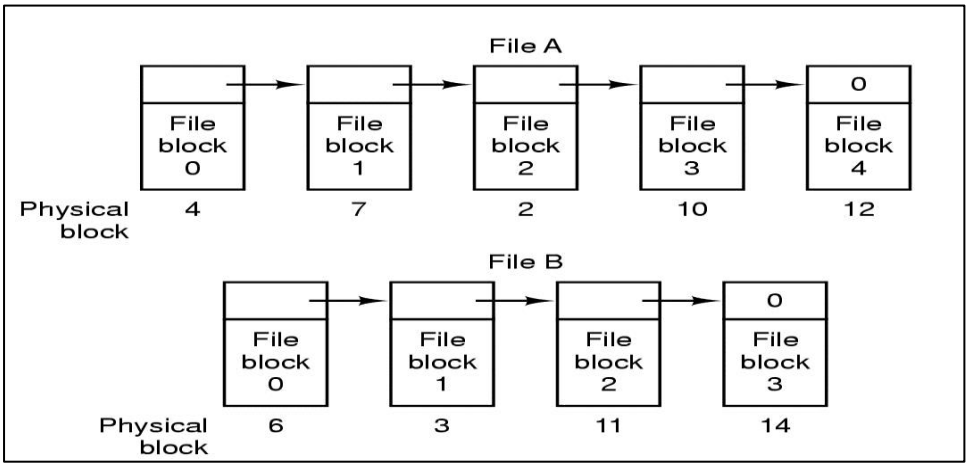


Figure 4.4: Linked List Allocation

Advantages

- No external fragmentation. Any free block can be used to satisfy the request
- A file can expand. No need to declare the file size when the file is first created
- No need to compact disk space

Disadvantages

- Direct-access is very inefficient
- The pointers take up some space and,
- Scattering the pointers all over the disk poses a reliability problem

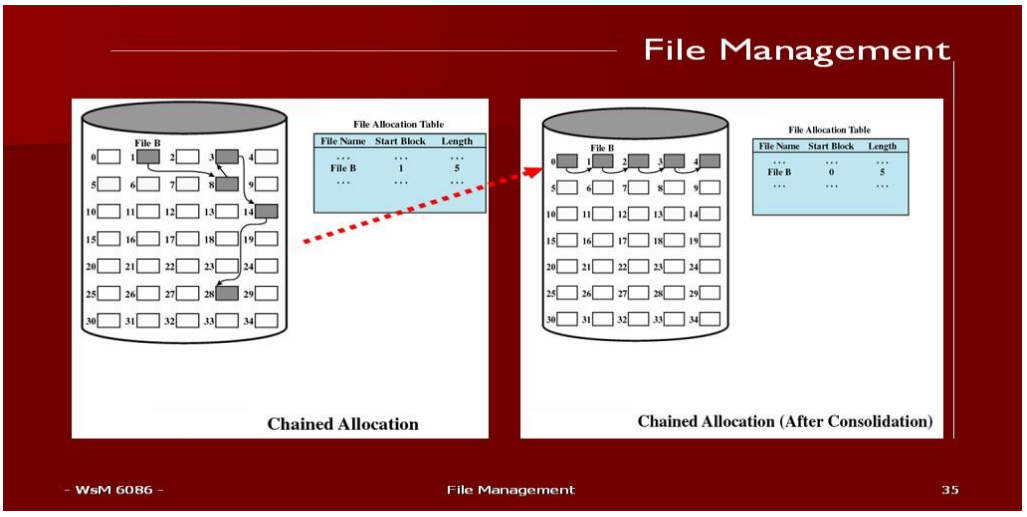


Figure 4.5 : Linked list Allocation Concept

Linked List Allocation Using an Index

Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table or index in memory. Figure 4.5 shows what the table looks like for the example of Fig. 4.4. In both figures, we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6,3, 1 I, and 14, in that order. Using the table of Fig. 4.5, we can start with block 4 and follow the chain dl the way to the end. The same can be done starting with block 6.

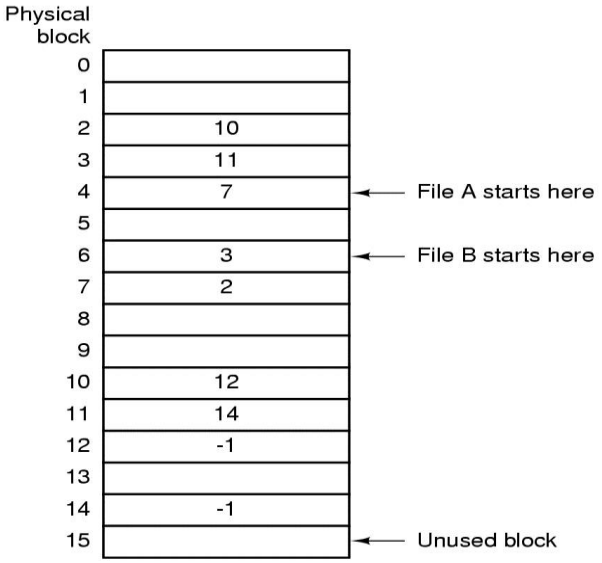


Figure 4.5: Linked list allocation using a table in main memory.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is. MS-DOS uses this method for disk allocation.

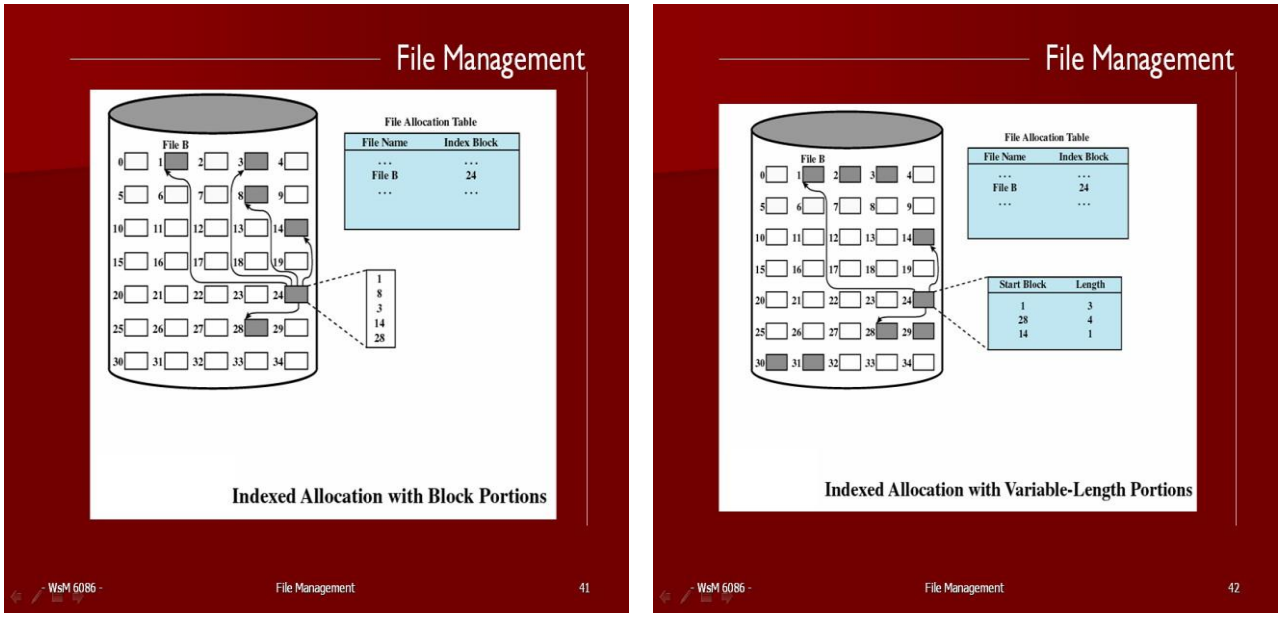


Figure 4.5.1: Linked-list Allocation using index mechanism

The primary disadvantage of this method is that the entire table must *be* in memory all the time to make it work. With a large disk, say, 500,000 1K blocks (500M), the table will have *500,000* entries, each of which will have to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 1.5 or 2 megabytes aI1 the time depending on whether the system is optimized for space or time. Although MS-DOS uses this mechanism, it avoids huge table5 by using large blocks (up to 32K) on large disks.

Advantages

- Grouping all the pointers into one location solves the problem of reliability.
- Direct-access is efficient. The 'i'th entry in the index block points to the 'i'th block of the file.

Disadvantages

- The pointers may waste a lot of space since an entire disk block must be allocated to hold them even if few pointers are actually used.

I-Nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a little table called an i-node (index-node), which lists the attributes and disk addresses of the file's blocks, as shown in **Figure.** 4.6.

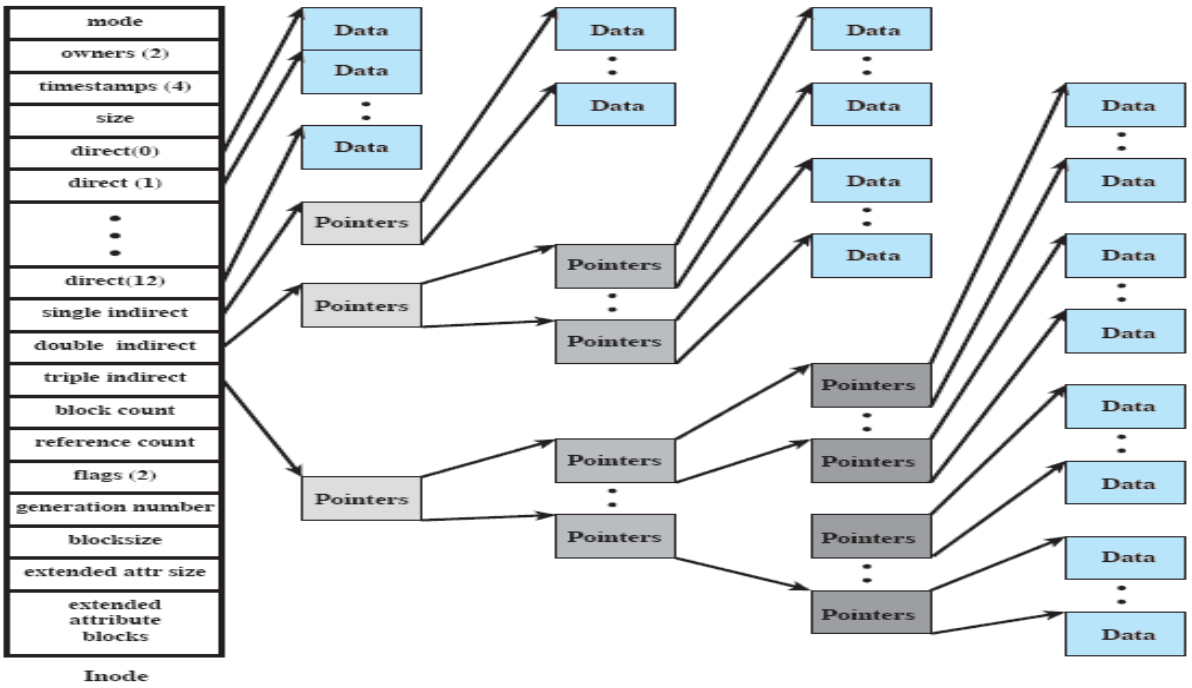


Figure 4.6: Structure of I-Nodes

The first few disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a single indirect block. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a double indirect block, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a triple indirect block can also be used.

Explanation how caching enhances file system performance

A file system (often also written as *filesystem*) is a method of storing and organizing computer files and the data. Using this approach, the performance of the system will enhances based on the factors as listed below :

- make it easy to find and access them
- may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location
- they might provide access to data on a file server by acting as clients for a network protocol
- a file system is a special-purpose database for the storage, organization and manipulation of data.

References

1. Andrew S. Tananbaum and Albert S Woodhull, (2006). Operating Systems Design and Implementation, Third Edition. Prentice Hall (ISBN: 0-131-42938-8)

2. Silberschatz, Galvin and Gagne, (2006). Operating System Concepts, Sixth Edition. John Wiley & Sons (ISBN: 0-471-69446-5)

3. Todd Lammle (2007). CCNA Cisco Certified Network Associate Study Guide (6th Edition), Wiley Publishing. (ISBN:978-0-470-110089)

4. Wendell Odam, Thomas Knott (2006). Networking Basics CCNA 1 Companion Guide (Cisco Networking Academy). Cisco Press. (ISBN: 978-1-58713-164-6)

5. Frederic Haziza (daz@it.uu.se) – Spring 2007 ; Paging and Segmentation

6. Memory and processor management *; www.aw-bc.com/info/davis/davis_rajkumar/sample.pdf*

7. Principles of Operating System CS 446/646 ; *doursat.free.fr/cs446.html*

8. Operating System Model; *www.buyya.com/microkernel/chap2.pdf*

9. Professor Mark Handley, (2007). Scheduling in Operating System

10. Vivek Pai, (2001) Lecture 4 teaching slides, COS31

11. Nurul Fadhlun Awang, (2010). Chapter 3:Resource Management

12. Hairi Alias, (2010). Chapter 4 : File Management

13. Website : *www.wabopedia.com, http://en.wikipedia.org*