Chapter: 1

Introduction to Operating Systems

1.1 INTRODUCTION

Operating System Definition

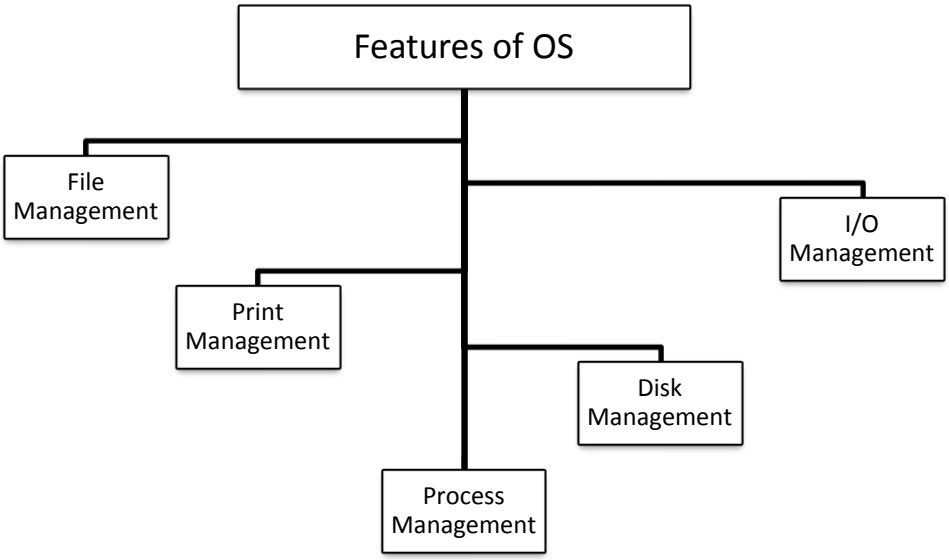
An operating system is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

An Operating system is a part of system software that is loaded in to computer on boot up that is responsible for running other applications and provides interface to interact with other programs that uses system hardware.

This interface is either command line user interface or Graphical User Interface. Command Line User interface is used in operating systems like MSDOS, UNIX, LINUX etc. and GUI is used with most of MS Windows operating systems like Windows XP, Windows Vista Windows 7 etc.

The hardware—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users.

1.2 FEATURES


Features of OS
File Management
I/O Management
Print Management
Disk Management
Process Management

1.3 Logical structure of an operating system:

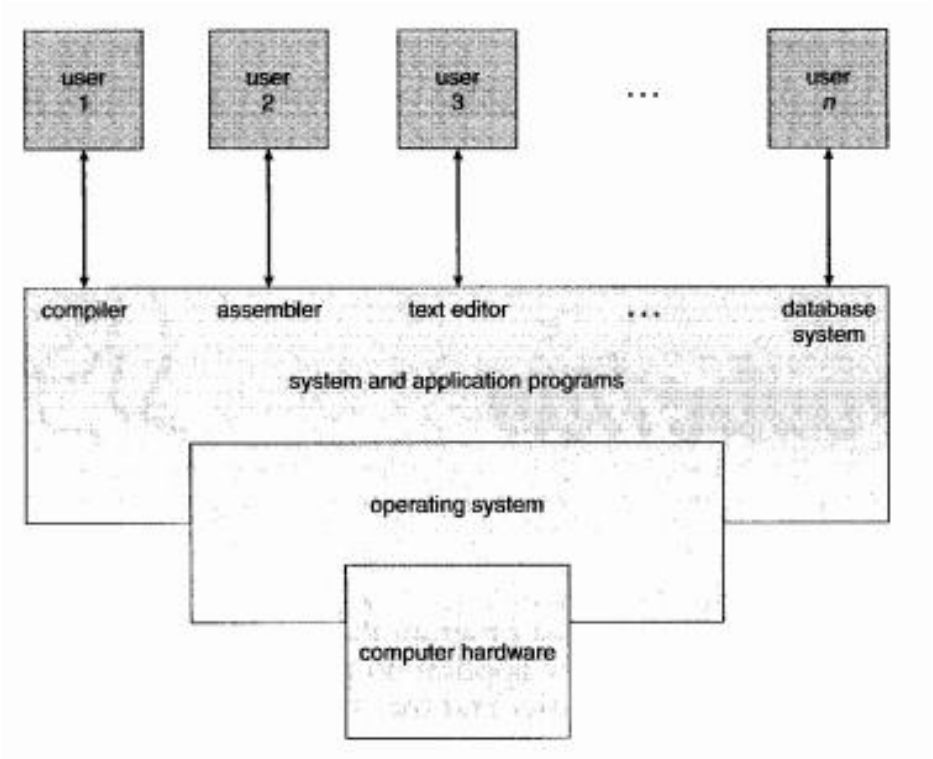


Figure: logical structure of an operating system

The hardware-the central processing unit (CPU), the memory, and the input/output (I/O) devices-provides the basic computing resources for the system. The application programs-such as word processors, spreadsheets, compilers, and web browsers-define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a *government.* Like a government, it performs no useful function by itself. It simply

provides an *environment* within which other programs can do useful work. Operating system can be divided into two groups:

1] Single process

2] Multi process.

Single process operating systems are capable of working on one task at a time while multi process operating systems can work on several processes at once by breaking the tasks into threads. Smallest part of programs that can be scheduled for execution is called as a thread. There are several terms related to multiprocessing which are as follows

Multitasking - It is the capability of an operating system to handle more than one task at a time. There are two types of multitasking

1] Co-operative Multitasking

2] Preemptive multitasking.

Co-operative Multitasking - Applications can control the system resource untilthey are finished. If a task caused faults or other problems, it would cause the system to become unstable and force a reboot. This type of multitasking is used in Windows 3.x.

Pre-emptive Multitasking - Applications are allowed to run for a specifiedperiod of time depending on how important the application is to the operation of the system (priority basis). If a particular task is causing problems or faults, that application can be stopped without the system becoming unstable. Used in Windows 9.x, Windows XP, Vista and Windows 7 and all network operating systems.

Multi user - This is similar to multitasking and is the ability for multiple users to access resources at the same time. The OS switches back and forth between users. For example all network operating systems like Windows server 2003, Windows server 2008, Linux, UNIX etc.

Multiprocessor - Having multiple processors installed in a system such thattasks are divided between them. Now all latest operating systems uses symmetric multiprocessing.

Multiprogramming - It is the capability of an operating system to run multipleprograms at once. Multiprogramming is possible because of multi threading.

Multithreading - It is the capability of an operating system to handle (execute) multiple threads of multiple programs at a time. One program may have at least one thread. Thread is a smallest part of a program that can be scheduled for execution.

There are two broad categories of operating systems

1] Desktop operating system

2] Network operating system.

1.3 DESKTOP OPERATING SYSTEM

Features of Desktop operating system

1] It a single user operating system.

2] It can support Multitasking, Multiprogramming, Multiprocessing and Multithreading.

3] User interface can be command line or GUI.

4] Desktop PCs, workstations and laptops are used to install desktop operating Systems.

5] Cost of the operating system is low as compare to Network operating system.

6] Desktop operating system can be configured as a client in network Environment.

7] Provides user level and share level security.

8] Desktop operating systems are as follows:

MSDOS, Windows 95/98, Windows 2000 Professional, Windows XP, Windows Vista and Windows 7.

1.4 NETWORK OPERATING SYSTEM

Features of NOS

1] It is a multi user operating system.

2] It is also called as server operating system.

3] It can support Multitasking, Multiprogramming, Multiprocessing and Multithreading and Multi User.

4] User interface can be Command Line or GUI.

5] Generally server grade computers are used for installation of network operating system.

6] Cost of network operating system is very high as compared to desktop operating system.

7] It provides high security that includes user level, share level as well as file level.

8] It provides backup tools to take the backup of important data. Backup can be scheduled in non working hours also.

9] It can be configured as a server to provide various types of network services like file, print, database, mail, proxy, web etc.

10] Network operating systems are as follows: Windows Server 2000, Windows Server 2003, Windows Server 2008.Red Hat Enterprise Linux Server 4.0, and 5.0, UNIX, Novell's Netware 4.x, 5.x and 6.x servers etc.

Functions of Operating Systems.

Apart from being as a User interface between system hardware and User applications operating system have other functions which are as follows

1] Memory management

2] Process Management

3] Disk Management

4] I/O management

5] Device Management

6] Backup Management

7] Security Management.

1.1 &1.2 Check Your Progress.

Q1. State Whether the following statements are true or false

1. Operating system is a part of Application software.
2. Smallest part of programs that can be scheduled for execution is called as a thread
3. All operating systems support Multi User feature. Multiprocessing supports multiple processors

1.3 & 1.4 Check Your Progress.

Fill in the Blanks

1] Desktop operating System is a ---------- ----- user operating system.

2] Desktop operating system can be configured as a ----- ------ in network environment.

3] Network operating system is a ---------user operating system.

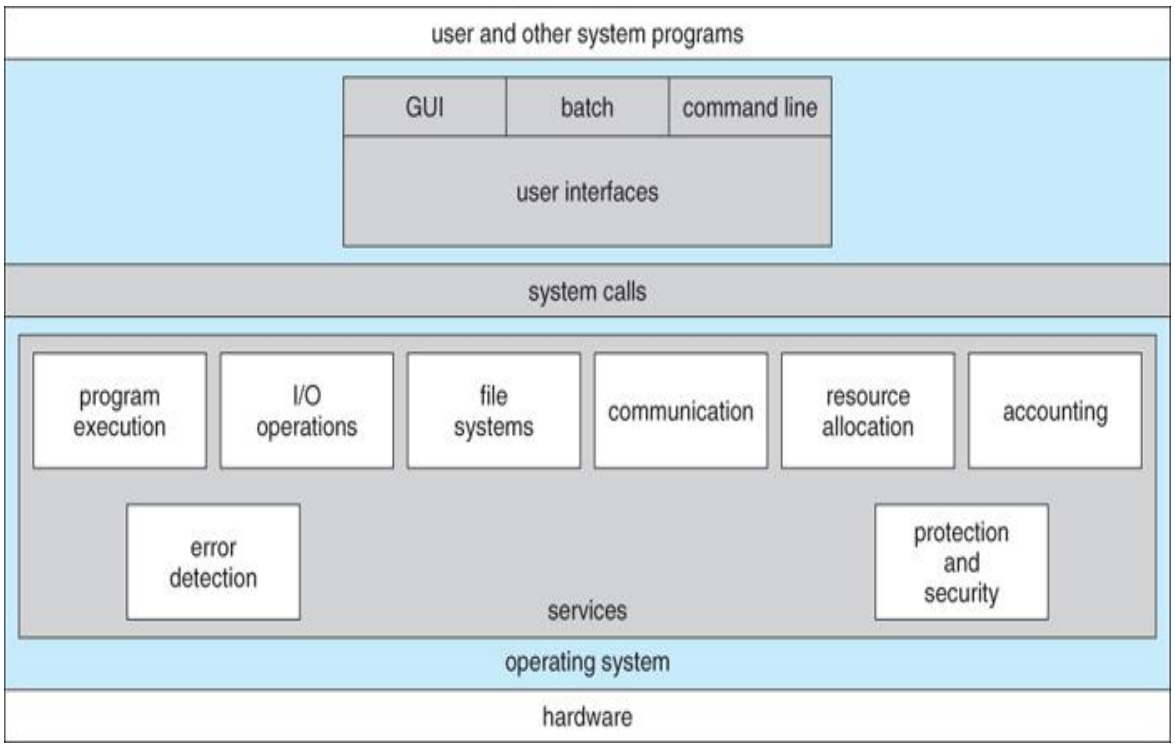
4] Cost of network operating systems are --- ----- as compared to Desktop operating system.

.1.5 OPERATING SYSTEM SERVICES :

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.

The specific services provided, of course, differ from one operating system to another but we can identify common classes. These services are provided for the convenience of the programmer to make the programming task easier.

The common services provided by the operating system is listed in following figure



Following are the basic services provided by operating systems for convenience of the users.

1. User Interface :

   User interface is a medium through which user actually interacts with the

computer system via operating system. Almost all operating systems have a user interface.(UI)

User Interfaces are categorised into three types as follows :

a) CLI: Command line interface provides an environment where user can write their commands in text format on command prompts.
b) Batch Interface: It allows a command environment which gets execute one by one or sequentially. User creates a batch file which contains multiple executable commands in sequence. This batch file is executed to execute the set of commands included in it.
c) GUI: Graphical User interface. Most of the operating systems provide graphical user interfaces which have tools like pointing devices which directs the input/output operations with the help of graphical control like menu etc.

2. Program Execution :

   The purpose of computer system is to allow the users to execute programs in an efficient manner. The operating system provides an environment where the user can conveniently run these programs. OS takes care of assigning work to the CPU (CPU job scheduling) for the purpose of program execution.

   Every program that runs on computer system needs memory. This memory allocation is done by operating systems. Also multitasking, multiprogramming etc features are provided by OS. After executing the program, de-allocation of memory is done by OS.The OS must be able to load the program into memory and to execute it. The program must be able to terminate its execution either normally or abnormally.

3. I/O Operations :

   Each program requires an input and after processing the input submitted by user it produces output. This involves the use of I/O devices. The input may be either from the file on the disk or from some other input device. The output may be written to some file on the disk or sent to some output devices such

as printer, monitor. Since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.

4. File-system manipulation :

   While working on the compute, generally a user is required to manipulate various types of files like as opening a file, saving a file and deleting a file from the storage disk. Program needs to read a file or write a file. The OS gives the permission to the program for operation on file. Maintain details of files or directories with their respective details. This is an important task that is also performed by the operating system.

5. Communications :

   Operating system performs the communication among various types of processes in the form of shared memory. In multitasking environment, the processes need to communicate with each other and to exchange their information. These processes are created under a hierarchical structure where the main process is known as parent process and the sub processes are known as child processes.

6. Error Detection:

   Error can occur anytime and anywhere. Error may occur in CPU, in I/O devices or in the memory hardware. Operating system deals with hardware problems. To avoid hardware problems the operating system constantly monitors the system for detecting the errors and fixing these errors (if found). The main function of Operating system is to detect the errors like bad sector on hard disk, errors related to I/O devices. After detecting the errors, OS takes appropriate action for consistent computing.

7. Resource Allocation :

   In the multitasking environment, when multiple jobs are running at a time, it is the responsibility of an OS to allocate the required resources (like CPU, main memory, tape drive or secondary storage etc) to each process for its better

utilisation. For this purpose various types of algorithms are implemented such as process scheduling, CPU scheduling, disk scheduling etc.

8. Accounting :
   OS keeps an account of all the resources accessed by each process or user. In multitasking, accounting enhances the system performance with the allocation of resources to each process ensuring the satisfaction to each process.

9. Protection :
   The owners of information stored in a multi user or networked computer system may want to control use of that information i.e. access to data and other related privileges like reading, writing, executing etc.
   Even the concurrent processes should not interfere with each other which may cause damage to the information. Protection: This involves ensuring that all access to system resources is controlled. Security: Unwanted users requires user authentication. The defending of external I/O devices from invalid access is important. It is important to protect the system and keep it secured.

1.7 UNIX OPERATING SYSTEM

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

The UNIX operating system is a set of programs that act as a link between the computer and the user.

The computer program that allocates the system resources and coordinates all the details of the computer's internals is called the **operating system** or the **kernel**.

Users communicate with the kernel through a program known as the **shell**. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

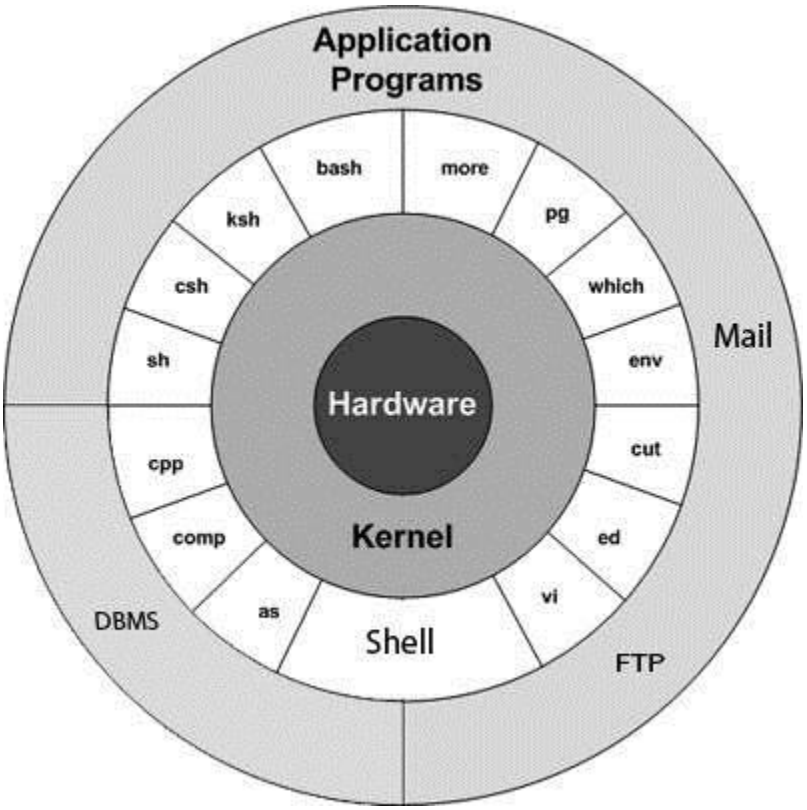
- UNIX was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.
- There are various UNIX variants available in the market. Solaris UNIX, AIX, HP UNIX and BSD are a few examples. Linux is also a flavor of UNIX which is freely available.
- Several people can use a UNIX computer at the same time; hence UNIX is called a multiuser system.
- A user can also run multiple programs at the same time; hence Unix is a multitasking environment

Types:

- There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.
- Here in the School, we use Solaris on our servers and workstations, and Fedora Linux on the servers and desktop PCs.

1.8 Unix Architecture

Basic block diagram of a Unix system −



The main concept that unites all the versions of Unix is the following four basics −

- **Kernel** − The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management.
- **Shell** − The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.
- **Commands and Utilities** − There are various commands and utilities which you can make use of in your day to day activities. **cp**, **mv**, **cat**and **grep**, etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various options.
- **Files and Directories** − All the data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the **file system**.

System Boot up

If you have a computer which has the Unix operating system installed in it, then you simply need to turn on the system to make it live.

As soon as you turn on the system, it starts booting up and finally it prompts you to log into the system, which is an activity to log into the system and use it for your day-to-day activities.

Login Unix

When you first connect to a Unix system, you usually see a prompt such as the following −

```
login:
```

To log in

- Have your user id (user identification) and password ready. Contact your system administrator if you don't have these yet.
- Type your user id at the login prompt, and then press **ENTER**. Your user id is **case-sensitive**, so be sure you type it exactly as your system administrator has instructed.
- Type your password at the password prompt, and then press **ENTER**. Your password is also case-sensitive.
- If you provide the correct user id and password, then you will be allowed to enter into the system. Read the information and messages that comes up on the screen, which is as follows.

```
login :amrood

amrood's password:

Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73

$
```

You will be provided with a command prompt (sometime called the **$** prompt) where you type all your commands. For example, to check calendar, you need to type the **cal** command as follows −

```
$ cal

    June 2009

Su Mo Tu We Th Fr Sa

1  2  3  4  5  6

7  8  9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29 30
```

```
$
```

Change Password

All Unix systems require passwords to help ensure that your files and data remain your own and that the system itself is secure from hackers and crackers. Following are the steps to change your password −

Step 1 − To start, type password at the command prompt as shown below.

Step 2 − Enter your old password, the one you're currently using.

Step 3 − Type in your new password. Always keep your password complex enough so that nobody can guess it. But make sure, you remember it.

Step 4 − You must verify the password by typing it again.

```
$ passwd

Changing password for amrood

(current) Unix password:******

New UNIX password:*******

Retype new UNIX password:*******

passwd: all authentication tokens updated  successfully


$
```

Note − We have added asterisk (*) here just to show the location where you need to enter the current and new passwords otherwise at your system. It does not show you any character when you type.

Listing Directories and Files

All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

You can use the **ls** command to list out all the files or directories available in a directory. Following is the example of using **ls** command with **-l** option.

```
$ ls -l
total 19621
drwxrwxr-x  2amroodamrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amroodamrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2amroodamrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root       4096 Dec  9  2007 urlspedia
-rw-r--r--  1 root   root     276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root       4096 Nov 25  2007 usr
-rwxr-xr-x  1 root   root       3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 amroodamrood     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amroodamrood      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 amroodamrood    166255 Aug  9  2007 yourfile.swf

$
```

Here entries starting with **d.....** represent directories. For example, uml, univ and urlspedia are directories and rest of the entries are files.

Who Are You?

While you're logged into the system, you might be willing to know : **Who am I**?

The easiest way to find out "who you are" is to enter the **whoami** command −

```
$ whoami
amrood
```

```
$
```

Try it on your system. This command lists the account name associated with the current login. You can try **who am i** command as well to get information about yourself.

Who is Logged in?

Sometime you might be interested to know who is logged in to the computer at the same time.

There are three commands available to get you this information, based on how much you wish to know about the other users: **users**, **who**, and **w**.

```
$ users
amroodbabluqadir

$ who
amrood ttyp0 Oct 8 14:10 (limbo)
bablu  ttyp2 Oct 4 09:08 (calliope)
qadir  ttyp4 Oct 8 12:09 (dent)

$
```

Try the **w** command on your system to check the output. This lists down information associated with the users logged in the system.

Logging Out

When you finish your session, you need to log out of the system. This is to ensure that nobody else accesses your files.

To log out

- Just type the **logout** command at the command prompt, and the system will clean up everything and break the connection.

System Shutdown

The most consistent way to shut down a Unix system properly via the command line is to use one of the following commands −

Sr. No	Command & Description
1	**halt** Brings the system down immediately
2	**init 0** Powers off the system using predefined scripts to synchronize and clean up the system prior to shutting down
3	**init 6** Reboots the system by shutting it down completely and then restarting it
4	**poweroff** Shuts down the system by powering off
5	**reboot** Reboots the system
6	**shutdown** Shuts down the system

You typically need to be the super user or root (the most privileged account on a Unix system) to shut down the system. However, on some standalone or personally-owned Unix boxes, an administrative user and sometimes regular users can do so.

1.5 & 1.6 Check your progress.

State Whether the following statements are true or false

1] Disk Operating System (DOS) is a single user single-process operating system

2] DOS support Graphical User Interface.

3] In DOS First 640k is Conventional Memory

4] Windows 9X support GUI as well as Command Line Interface (CLI)

5] SYSTEM.DAT - Contains information about hardware and system settings.

6] Windows 9x supports long file names up to 155 characters

6] Windows XP support two processors.

7] Windows XP has two editions as 32bit and 64 bits.

8] Windows Vista is a successor of Windows 98.

9] Windows Vista supports 4 way SMP i.e. 4 processors.

10] Windows XP Home Edition cannot take part in Domain Networks.

Fill in the Blanks

1] DOS stands for _____

2] Windows 9x supports long file names up to ----------

characters.

3] Windows XP supports maximum ---------- CPUs.

4] Windows Vista does not use ---------- ----- file as used in windows 2000 and windows XP.

5] Windows XP supports ---------- ----- , ----------and ---------- file systems.

Match the followings

1. Windows XP	1. Single user single task operating system.
2. DOS	2. Two editions of 32 and 64 bits.
3. Windows Vista	3. Desktop Operating System.
4. Windows 2000	4. Boot.ini file is not used. Professional

Summary

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Operating System can be divided into 2 groups i.e. single and multi process. Multitasking is the capability of operating system to handle more than one task at a time.

Desktop as is single use operating system Networks Operating System is multiuser Operating System.

The main benefit of using XP include its reliability, performance, security, ease of use, support for remote users, management and development capabilities. Windows Vista is also language neutral

Check your progress – answers

1.1 & 1.2

1] False 2] True 3] False 4] True

1.3 - 1.4

1] Single 2] Client 3] Multi 4] High

1.5 - 1.6

1]	True	2]	False	3]	True
4]	True	5]	True	6]	False
6]	True	7]	True	8]	False
9]	False	10]	True		

Fill in the Blanks

1] Disk Operating System. 2] 255 3] Two 4] Boot.ini

5] FAT16, FAT 32 and NTFS

Match the followings

1-2 2-1 3-4 4-3

Questions for self study

1. Define Operating system.
2. Explain logical structure of operating system.
3. List and explain all services of operating system.
4. Explain UNIX operating system architecture.

Suggested readings

Operating systems By Stuart E. Madnick, John J. Donovan

References used:

1. Operating systems by William Stallings
2. Operating systems fundamental concepts
3. www.tutorialspoint.com

Notes

Chapter: 2

Processes & Processor Management

2.0 Objectives

2.1 Introduction

2.2 Process concept and process states

2.3 CPU and I/O bound

2.4 Operating system services

2.4.1 Process management

2.4.2 Thread management

2.5 CPU scheduler

2.5.1 Short

2.5.2 Medium

2.5.3 Long-term, dispatcher

2.6 Scheduling:

2.6.1 Preemptive

2.6.2 Non-preemptive

2.7 Scheduling algorithms

2.7.1 FCFS

2.7.2 SJFS

2.7.3 Shortest remaining time

2.7.4 RR

2.7.5 Priority scheduling

2.0 OBJECTIVES

- Describe/Analyze processor management with the observance of Job Scheduler and process scheduler.
- Discuss the process synchronization and multi-processor systems.

2.1 INTRODUCTION

In most systems it is necessary to synchronize processes and jobs, a task that is also -performed by modules of processor management. The job scheduler can be viewed as a macro scheduler, choosing which jobs will run. The process scheduler can be viewed as a micro scheduler, assigning processors to the processes associated with scheduled jobs.

The user views his job as a collection of tasks he wants the computer system to perform for him. The user may subdivide his job into job steps, which are sequentially processed sub tasks (e.g., compile, load, execute). The system creates processes to do the computation of the job steps.

Job scheduling is concerned with the management of jobs, and processor scheduling is concerned with the management of processes. In either case, the processor is the key resource.

In a non-multiprogramming system no distinction is made between process scheduling and job scheduling, as there is a one-to-one correspondence between a job and its process and only one job is allowed in the system at a time. In such a simple system the job scheduler chooses one job to run. Once it is chosen, a process is created and assigned to the processor.

For more general multiprogramming systems, the job scheduler chooses a small subset of the jobs submitted and lets them "into" the system. That is, the job scheduler creates processes for these jobs and assigns the processes some resources. It must decide, for example, which two or three jobs of the 100 submitted will have any

resources assigned to them. The process scheduler decides which of the processes within this subset will be assigned a processor, at what time, and for how long.

2.2 PROCESS CONCEPT AND PROCESS STATES

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

Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

2.3 CPU and I/O bound - Processes

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution.

A process is the unit of work in a modern time-sharing system. The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive. In this chapter, you will read about what processes are and how they work.

2.4.1 The Process Management

A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A

process may also include a heap, which is memory that is dynamically allocated during process run time. The structure of a process in memory.

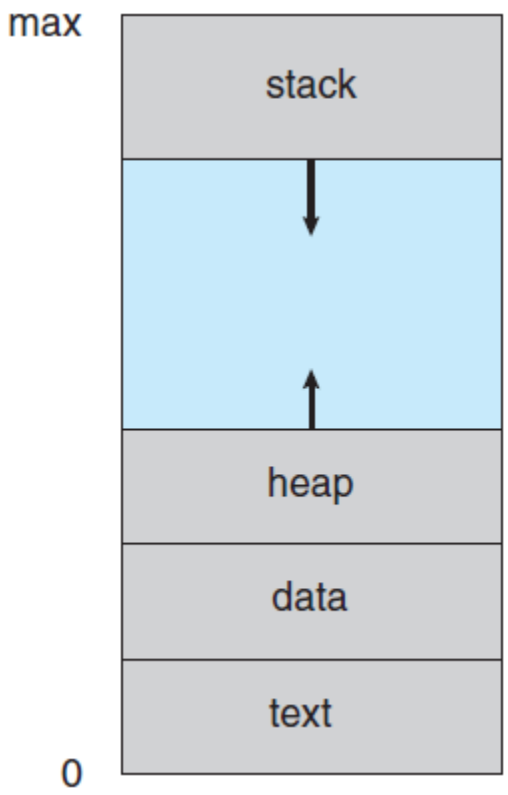


Figure: Process in memory

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files

2.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

• **New**. The process is being created.

• **Running**. Instructions are being executed.

• **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• **Ready**. The process is waiting to be assigned to a processor.

• **Terminated**. The process has finished execution.

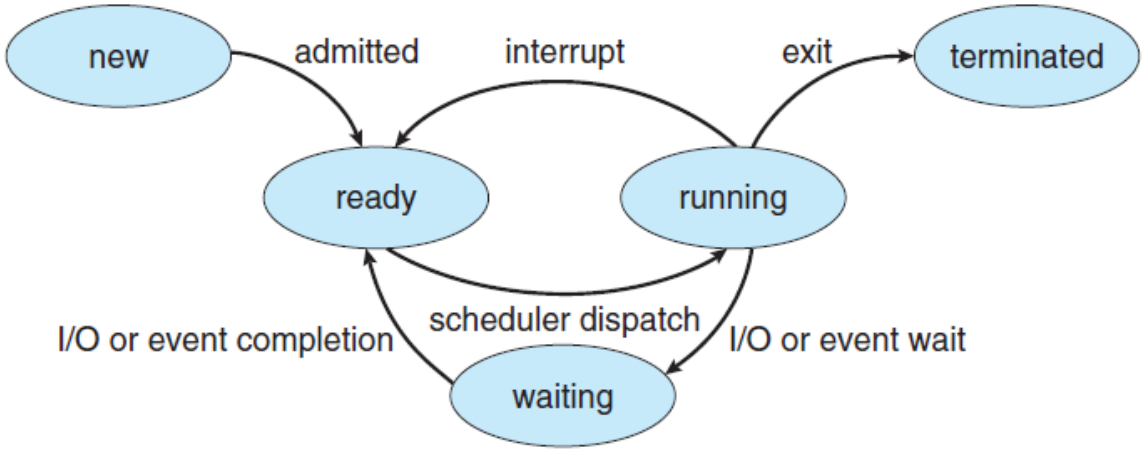


Figure: Diagram of process state.

Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure. It contains many pieces of information associated with a specific process, including these:

• **Process state.** The state may be new, ready, running, waiting, halted, and so on.

• **Program counter.** The counter indicates the address of the next instruction to be executed for this process.

• **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

• **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 6 describes process scheduling.)

• **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system

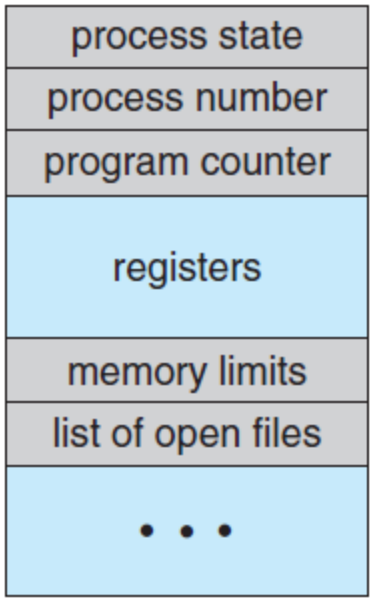


Figure: Process Control Block

Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

• **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on. In brief, the PCB simply serves as the repository for any information that may vary from process to process.

2.4.2 Thread Management

A thread is a path of execution within a process. Also, a process can contain multiple threads. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multi core systems, where multiple threads can run in parallel. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

Process vs. Thread?

The typical difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

2.2 Check Your Progress.

Fill in the Blanks.

1. When the operating system received the signal that my I/O process was finished, the traffic controller module changed my CPU _____

2. Allocate the necessary resources for the scheduled job by use of memory, device, and _____

.

2.4 Check Your Progress.

Fill in the Blanks.

1. In _____ each process in turn is run to a time quantum limit, such as 100 ms.

2. In _____ if the process used its entire quantum last time, it goes to the end of the list. If it used only half (due to I/O wait), it goes to the middle. Besides being "fair," this works nicely for I/O-bound jobs.

Advantages of Thread over Process

1. Responsiveness*:* If the process is divided into multiple threads, if one thread completed its execution, then its output can be immediately responded.

2. Faster context switch*:* Context switch time between threads is less compared to process context switch. Process context switch is more overhead for CPU.

3. Effective Utilization of Multiprocessor system*:* If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.

***4. Resource sharing**:* Resources like code, data and file can be shared among all threads within a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

***5. Communication**:* Communication between multiple thread is easier as thread shares common address space. While in process we have to follow some specific communication technique for communication between two processes

***6. Enhanced Throughput of the system**:* If process is divided into multiple threads and each thread function is considered as one job, then the number of jobs completed per unit time is increased. Thus, increasing the throughput of the system.

Types of Thread

There are two types of thread.

User Level Thread

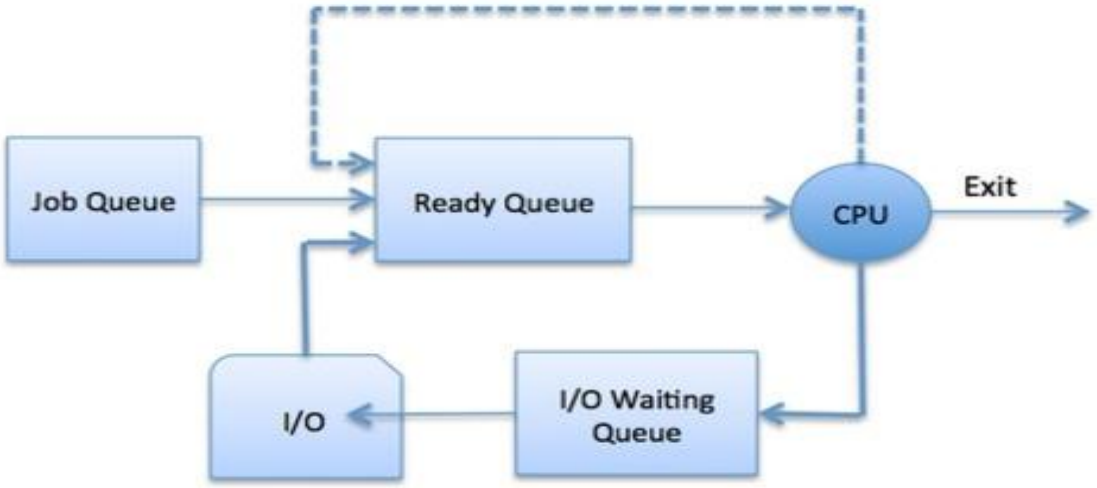
Kernel Level Thread.

2.5 CPU Scheduler

Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

2.5 CPU Schedulers

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

2.5.1 Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

2.5.2 Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

2.5.3 Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended process cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

2.6 Process Scheduling in an OS

In an operating system (OS), a process scheduler performs the important activity of scheduling a process between the ready queue and waiting queue and allocating them to the CPU. The OS assigns priority to each process and maintains these queues. The scheduler selects the process from the queue and loads it into memory for execution. There are two types of process scheduling: preemptive scheduling and non-preemptive scheduling.

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

Non-preemptive Scheduling

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

Following are some characteristics of non-preemptive scheduling

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

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.

The strategy of allowing processes that are logically run able to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

2.7 Process scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

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

- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state; it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

2.7.1 First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4

P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

2.7.2 Shortest Job First (SJF)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3
P1	0 - 0 = 0
P2	16 - 2 = 14

P3	8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

2.7.3 Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

2.7.4 Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

2.7.5 Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Atomic Transactions

A sequence of operations that perform a single logical function is called as **Atomic Transactions**

A transaction that happens completely or not at all It contains no partial results.

Ex: Cash machine hands you cash and deducts amount from your account

Airline confirms your reservation and – Reduces number of free seats

ACID Properties

A transaction is a very small unit of a program and it may contain several low level tasks. A transaction in a database system must maintain Atomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a

database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

2.5 Check Your Progress.

Fill in the Blanks.

1. This configuration flexibility is useful if there are some jobs that require the full complement of memory and/or I/O _____

2. A variation on the separate system multiprocessor technique is the _____ approach (this is also called loosely coupled multiprocessing).

2.6 Check Your Progress.

Fill in the Blanks.

1. Associated with processor allocation and interprocess communication are two synchronization problems, _____.

2. A _____ occurs when the scheduling of two processes is so critical that the various orders of scheduling them result in different computations.

In many instances, the job scheduling and process scheduling algorithms must interact closely.

If we attempt multiprogramming too many processes at the same time, especially in a demand paging system, all the processes will suffer from poor performance due to an overloading of available system resources (this phenomenon, called thrashing). On the other hand, in a conversational timesharing system, every user wants a "crack at the machine" as soon as possible-I-hour waits will not be tolerated.

As a compromise, we can adopt the philosophy that it is far better to give a few jobs at a time a chance than to starve them all! For example, we may allow two to five users to be ready at a time.

The basic concepts here are: (l) multiprogramming "in the small" for two to five jobs, process-scheduling every few ms; and (2) timeshare "in the large," where every few 100 ms or seconds one ready job is placed in hold status and one of the hold status jobs is made ready. (See Figure 2.13) Thus, over a period of a minute or so, every job has had its turn, and furthermore, the system runs fairly efficiently.

A SIGNAL (X) checks the blocked list associated with lock byte X; if there are any processes blocked waiting for X, one is selected and its PCB is set to the ready state. Eventually, the process scheduler will select this newly "awakened" process for execution.

The WAIT and SIGNAL mechanisms can be used for other purposes, such as waiting for I/O completion. After an I/O request (i.e., SIO instruction) has been issued, the process can be blocked by a WAlT (X) where X is a status byte associated with the I/O device (e.g., the lock byte can be stored in the device's Unit Control Block as shown in the sample operating system). When the I/O completion interrupt occurs, it is converted into a SIGNAL (X).

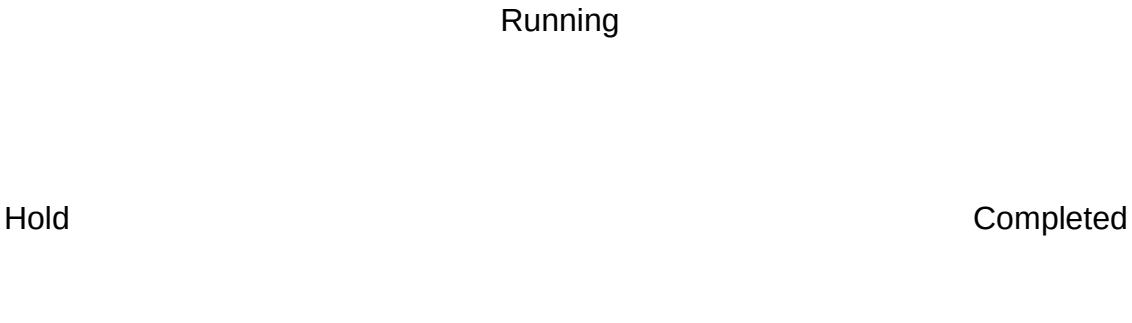


Figure 2.13 State Diagram

2.8 SUMMARY

Processor management has major functions: (1) job scheduling, (2) process scheduling, and (3) traffic control (such as inter process communication and synchronization). Job schedulers range from very simple, even manually done, algorithms to complex automated schedulers that consider the mix of jobs waiting to be run (i.e., in the hold state) as well as the available resources of the system. A unipro-gramming operating system or simple priority-based process scheduler does not provide the same degree of optimization available in an adaptive process scheduling algorithm. In any type of system the interaction between job scheduling and process scheduling must always be carefully considered. If the job scheduler keeps dumping more and more jobs on an already overloaded process scheduler, system performance will suffer greatly.

Considerable research and study are still needed to resolve many of the problems connected with interprocess communication and synchronization. These features are very important to modern modular operating systems, as illustrated by their extensive usage in the sample operating system.

2.9 CHECK YOUR PROGRESS – ANSWERS

2.2 Fill in the blanks

1. Traffic controller module
2. Memory, device, and processor management.

2.4 Fill in the blanks

1. Round robin

2. Inverse of remainder of quantum.

2.5 Fill in the blanks

1. Device resources
2. Coordinated job scheduling

2.6 Fill in the blanks

1. Race condition and deadly embrace.
2. Race condition

Questions for self - study

1. Describe state model.
2. What is process scheduling?
3. Describe process synchronization in detail.
4. What's the difference between combined job and process scheduling?
5. Explain process control block.

Suggested readings

1. **Operating System Concepts b**y Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
2. **Operating systems** By Stuart E. Madnick, John J. Donovan

References used:

1. Operating systems by William Stallings
2. Operating systems fundamental concepts
3. www.tutorialspoint.com
4. Abraham-Silberschatz-Operating-System-Concepts---9th edition
5. Operating systems by Achyut Godbole

Notes

Chapter 3:

Inter Process Communication and Synchronization

3.0 Objectives

- To learn message passing concepts.
- To describe communication in client-server systems.
- To understand race condition.
- To understand critical section problem.

3.1 Introduction

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves

synchronizing their actions and managing shared data. This tutorial covers a foundational understanding of IPC.

3.2 Introduction to message passing

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

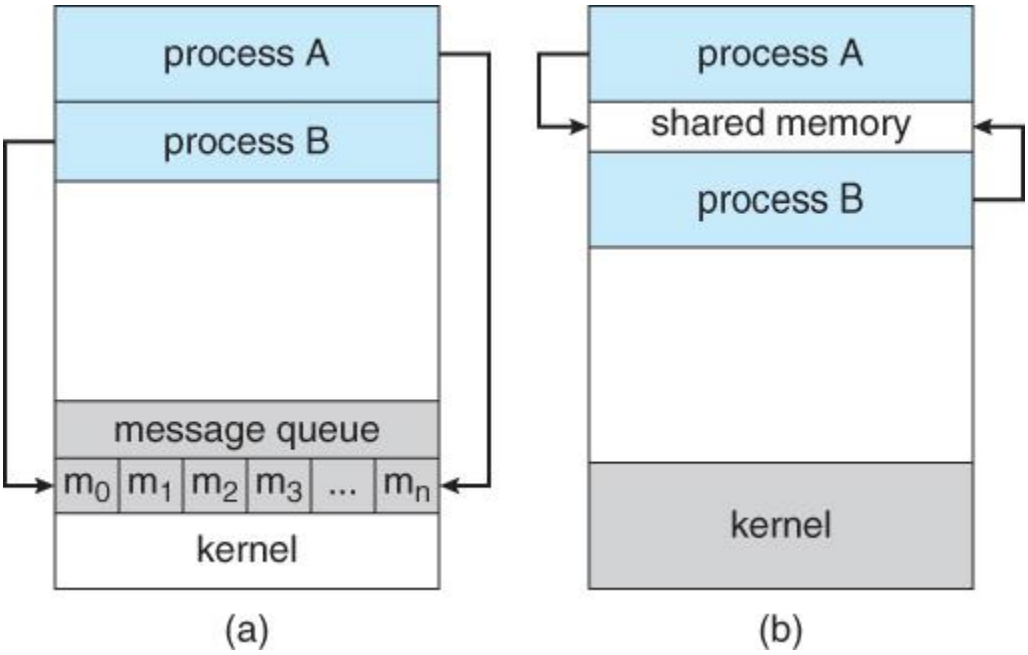
1. Shared Memory
2. Message passing

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

A process can be of two types:

- Independent process.
- Co-operating process.

The Figure below shows a basic structure of communication between processes via shared memory method and via message passing.



Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

Communication can be of two types −

- Between related processes initiating from only one process, such as parent and child processes.
- Between unrelated processes, or two or more different processes.

Following are some important terms that we need to know before proceeding further on this topic.

Pipes − Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

FIFO − Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

Message Queues − Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

Shared Memory − Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

Semaphores − Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.

Signals − Signal is a mechanism to communication between multiple processes by way of signalling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

3.3 Race Condition:

Where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

Its solution is that these processes must be synchronized in some way. We must guarantee that only one process at a time can be manipulating this resource.

3.2 & 3.3 Check your progress

Q1. Fill in the blanks.

1. An IPC facility provides at least two operations _____ & _____ message.
2. _____ system allows processes to communicate with one another without resorting to shared data

Q2. True or false.

1. Message passing system allows processes to communicate with one another without resorting to shared data
2. Messages sent by a process can be fixed or variable sized.

3.4 Critical Section:

Consider a system consisting of n processes {P0, P1,…, Pn-1}. Each process has a segment of code, called a Critical section, in which the process may be changing common variables, updating a table, writing a file and so on.

The important feature of the system is that when one process is executing in its critical section no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process p is shown as below :

do

[

Entry section

Critical section

Exit section

Remainder section

} while(True);

The critical section can be defined by following three sections:

1. Entry Section : Each process must request permission to enter its critical section; processes may go to critical section through entry section.
2. Exit Section : Each critical section is terminated by exit section.
3. Remainder section : The code after exit section is remainder section.

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

   1. Mutual Exclusion : If process Pi is executing in its critical section, the no other processes can be executing in their critical sections. So mutual exclusion can be defined as : " A way of making sure that if one process is using a shared modifiable data , the other processes will be excluded from doing the same thing ". We need four conditions to have a good solution for the mutual exclusion is :
      i) No two processes may at the same moment inside their critical section.
      ii) No assumptions are made about relative speeds of processes or number of CPU's
      iii) No process should outside its critical section block other processes.
      iv) No process should wait arbitrary long to enter its critical section.
   2. Progress : If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next and this selection cannot be postponed indefinitely.

The progress criteria prevent an infinite loop executing by one process in non-critical section code from stopping a different process from entering a critical region. But it does not prevent an infinite loop inside a critical region from making the system hang.

3. Bounded waiting : There exists a bound, or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical Section Solutions :

1. Disabling Interrupts :

   This is the simplest solution. When process enters in critical region, interrupts are disabled by it and once it leaves critical section , it again enables interrupts. Because of either clock or interrupt, CPU makes switching from one process to other. When interrupts are disabled, CPU switching will nbot happen.

Here process can complete its shared memory operations without any fear and bothering about other process to enter.

Important thing is that we have to give power to a process to disable interrupt. That will be unwise because if the processes disables the interrupts and do not make them enable then many problems can occur. Only kernels can easily enable and disable the interrupts. So it is not a good approach for mutual exclusion.

2. Lock Variables :

This could be one of the solutions. Here we can declare two variables one single and shared(lock) initialized to 0. If the process finds that lock is 0 then it enters into critical region and makes lock variable 1 and just before leaving it is again set to 0.

If another process enters then it can see that lock variable is 1 then it waits till it becomes 0. There could another case where process keep waiting and if the lock variable becomes 0 then one more process may enters and tries to make that lock variable to 1 before waiting process makes it. So this approach is also not that effective because this solution will bring race condition.

3.5 TSL Instructions

The only problem with the lock variable solution is that the action of testing the variable and the action of setting the variable are executed as separate instructions. If these operations could be combined into one indivisible step, this could be a workable solution. These steps can be combined, with a little help from hardware, into what is known as a **TSL** or **TEST and SET LOCK** instruction. A call to the **TSL** instruction copies the value of the lock variable and sets it to a nonzero (locked) value, all in one step. While the value of the lock variable is being tested, no other process can enter its critical section, because the lock is set.

If a process tests a variable which is locked, it will continue to test the variable again and again until it can enter its critical section. In other words, as long as a process is denied access to its critical section, it will stay in a tight loop and wait until it can proceed, all the while wasting processor time. This continuous testing of the lock variable is called **busy waiting**. Mutual exclusion policies that require busy waiting waste valuable processor time, and in some cases can lead to situations where a process will test the lock variable forever, a very undesirable occurrence.

3. Strict Alteration : In strict alteration, variable named turn is used to run two processes alternately. All processes share the common integer variable ‘turn’ initialise as 0 or 1. If ‘turn =I’, then process Pi is allowed to execute in its critical section. This algorithm ensures that only one process at a time can be in critical section but does not satisfy the progress condition.

Using Systems calls 'sleep' and 'wakeup'

Basically, what above mentioned solution do is this: when a process wants to enter in its critical section , it checks to see if then entry is allowed. If it is not, the

process goes into tight loop and waits (i.e., start busy waiting) until it is allowed to enter. This approach waste CPU-time.

Now look at some inter process communication primitives is the pair of steep-wakeup

.Sleep :

It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.

Wakeup

It is a system call that wakes up the process.

Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups' .

4. **Peterson's Solution**

Peterson's Solution is a classical software based solution to the critical section problem.

In Peterson's solution, we have two shared variables:

i) Boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section

ii) int turn : The process whose turn is to enter the critical section.

Peterson's Solution preserves all three conditions

 Mutual Exclusion is assured as only one process can access the critical section at any time.

- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

3.6 Semaphores and monitors:

Semaphores :

Semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. It is a synchronisation tool. Semaphore S is integer variable. Semaphore can be accesses with the help of only two atomic operations Signal() and wait().

Wait () operation was originally referred as P and signal operation was called as V.

The definition of wait() is as below:

```
Wait(s)
{
While s<=0;
s--;
}
```

The definition of signal() is :

```
Signal(s)
{
S++;
}
```

Only one operation i.e. either wait() or signal() can access the value of semaphore and modify it.

No two operations can access the value of semaphore at a time.

Semaphores could be binary semaphores and counting semaphores. Operating system can distinguish them. Binary semaphores can be sometime called as mutex locks for they provide locks for mutual exclusions.

Binary semaphores can be used in critical section when multiple processes are involved.

Counting semaphores can be used for controlling the access to the resources whose instances are finite number. Every process that needs resource performs the wait() operation on semaphore and decrements it.

When process releases the resource it uses signal() operation and thus semaphore is incremented. When semaphores value is zero, it indicated that all resources are used. At this stage if any process needs any resources, that processes will be blocked till the semaphores value becomes greater than one.

Let us assume, multiple processes are using the same region of code and if all want to access parallelly then the outcome is overlapped. Say, for example, multiple users are using one printer only (common/critical section), say 3 users, given 3 jobs at same time, if all the jobs start parallelly, then one user output is overlapped with another. So, we need to protect that using semaphores i.e., locking the critical section when one process is running and unlocking when it is done. This would be repeated for each user/process so that one job is not overlapped with another job.

Basically semaphores are classified into two types −

Binary Semaphores − Only two states 0 & 1, i.e., locked/unlocked or available/unavailable, Mutex implementation.

Counting Semaphores − Semaphores which allow arbitrary resource count are called counting semaphores.

Assume that we have 5 printers (to understand assume that 1 printer only accepts 1 job) and we got 3 jobs to print. Now 3 jobs would be given for 3 printers (1 each). Again

4 jobs came while this is in progress. Now, out of 2 printers available, 2 jobs have been scheduled and we are left with 2 more jobs, which would be completed only after one of the resource/printer is available. This kind of scheduling as per resource availability can be viewed as counting semaphores.

To perform synchronization using semaphores, following are the steps −

Step 1 − Create a semaphore or connect to an already existing semaphore (semget())

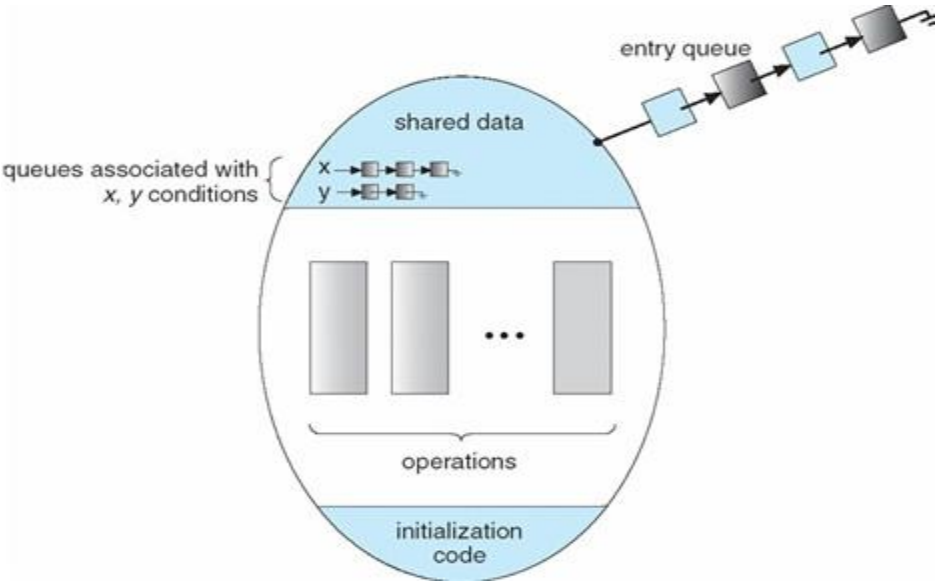
Step 2 − Perform operations on the semaphore i.e., allocate or release or wait for the resources (semop())

Step 3 − Perform control operations on the message queue (semctl())

Monitors :

Monitor is one of the ways to achieve Process synchronization. Monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can’t access the internal variable of monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.



Block diagram of Monitors

Syntax of Monitor

Monitor Demo //name of monitor-monitor is the keyword, Demo is the name of monitor

{

Variables;

Condition variables;

Procedure P1{…}

Procedure P2 {…}

}

Condition Variables

Two different operations are performed on the condition variables of the monitor.

Wait.
signal.

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Signal operation

x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

CLASSICAL EPIC PROBLEMS :

Below are some of the classical problem depicting flaws of process synchronization in systems where cooperating processes are present.

We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

Bounded Buffer Problem

- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.

  Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.

Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

3.4 & 3.5 Check your progress

Q1. Fill in the blanks.

1. _____ of the following is a synchronization tool.
2. Mutual exclusion can be provided by the_____.
3. Process synchronization can be done on both _____ & _____.

Q2. True or false.

1. If a process is executing in its critical section, then no other processes can be executing in their critical section. This condition is called mutual exclusion
2. Parent process can be affected by other processes executing in the system

3.7 Summary

The role of the operating system in matters of IPC is to offer a sufficiently rich set of IPC-supporting primitives. These should allow the tasks to engage in IPC without having to bother with the details of their implementation and with hardware dependence. This is not a minor achievement of the operating system developers, because making these IPC primitives safe and easy to use requires a lot of care and insight. In any case, the current state-of-the-art in operating systems' IPC support is such that they still don't offer much more than just primitives. Hence, programmers have to know how to apply these primitives appropriately when building software systems consisting of multiple concurrent tasks; this often remains a difficult because error-prone design and implementation job. Not in the least because no one-size-fits-all solution can exist for all application needs.

Synchronization and data exchange are complementary concepts, because the usefulness of exchanged data often depends on the correct synchronization of all tasks involved in the exchange. Both concepts are collectively referred to as Inter process communication ("IPC").

Check your progress – Answers

3.2 & 3.3

Q1. Fill in the blanks.

1. Receive and send
2. Message Passing

Q2. True or false.

1. True
2. True.

3.3 & 3.5

Q1. Fill in the blanks

1. Semaphore
2. Mutex locks and binary semaphores
3. Software and hardware.

Q2. True or false.

1. True
2. False.

Questions for self study.

1. **What is Inter-process communication?**
2. **What are the models of IPC?**
3. **Explain classical epic problems.**
4. **Explain message passing.**

Suggested readings

1. **Operating System Concepts b**y Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
2. **Operating systems** By Stuart E. Madnick, John J. Donovan

References used:

6. Abraham-Silberschatz-Operating-System-Concepts---9th edition
7. Operating systems by William Stallings
8. Operating systems fundamental concepts
9. www.tutorialspoint.com
10. Operating systems by Achyut Godbole

Notes

Chapter 4:

Memory Management

4.0 OBJECTIVES

- To provide a detailed description of various ways of organizing memory hardware.
- To explore various techniques of allocating memory to processes.
- To discuss in detail how paging works in contemporary computer systems.
- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.
- To discuss the principles of the working-set model.
- To examine the relationship between shared memory and memory-mapped

files.

- To explore how kernel memory is managed.

4.1 Introduction

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution. To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the *hardware* design of the system. Most algorithms require hardware support

As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep several processes in memory—that is, we must share memory.

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

4.2 Basic Hardware Issues

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them. Registers that are built into the CPU

are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control. Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation we must protect the operating system from access by user processes.

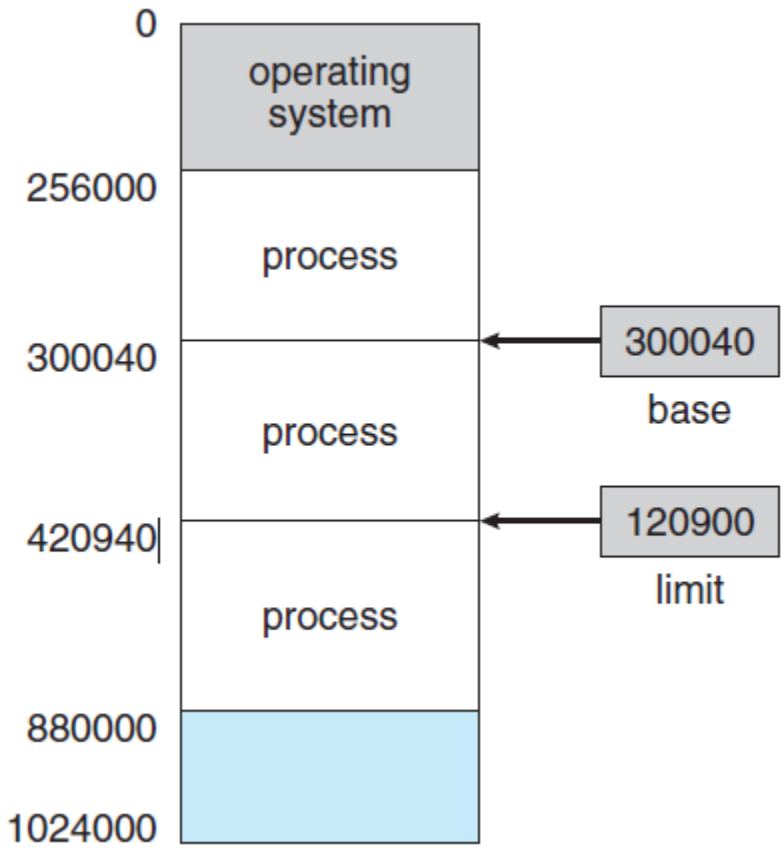


Figure: A base and a limit register define a logical address space.

On multiuser systems, we must additionally protect user processes from one another. This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this production in several different ways, as we show throughout the chapter. Here, we outline one possible implementation. We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.

For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive). Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 8.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers

This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents. The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access

and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next processes
context from main memory into the registers.

4.3 Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. The normal single-tasking procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. You will see later how a user program actually places a process in physical memory. In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 8.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

• **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that

location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.

• **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed Most general-purpose operating systems use this method

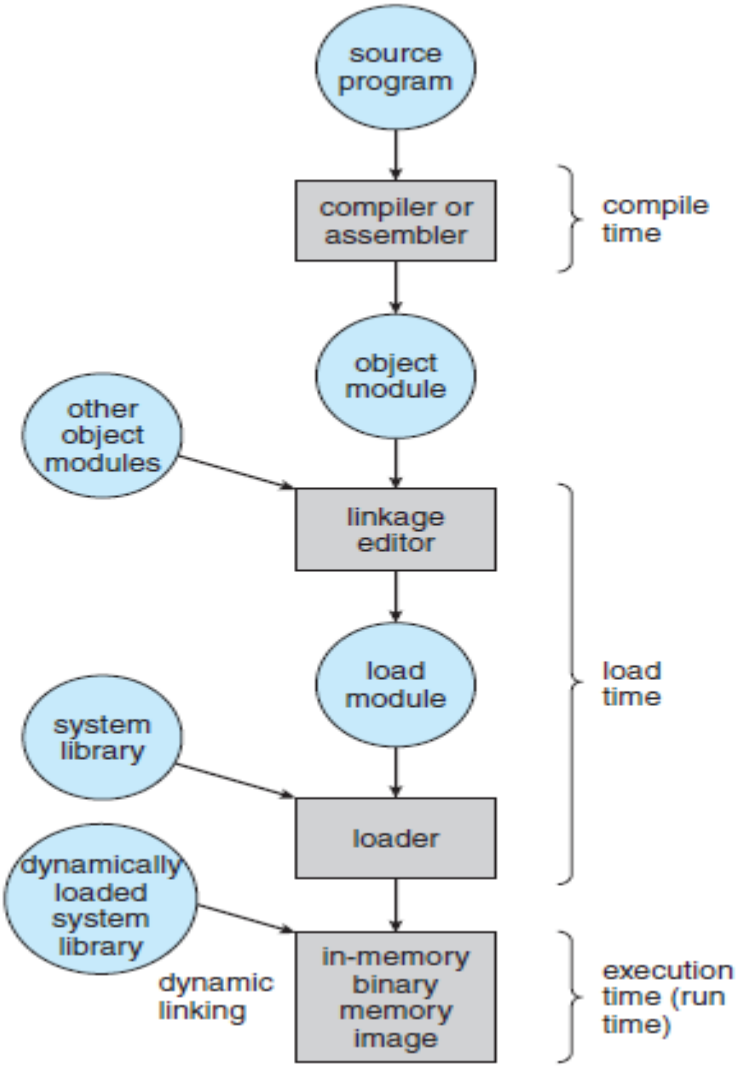


Figure: Multistep processing of a user program

4.4 Logical Vs Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**. The compile-time and load-time address-binding methods generate identical logical and physical addresses.

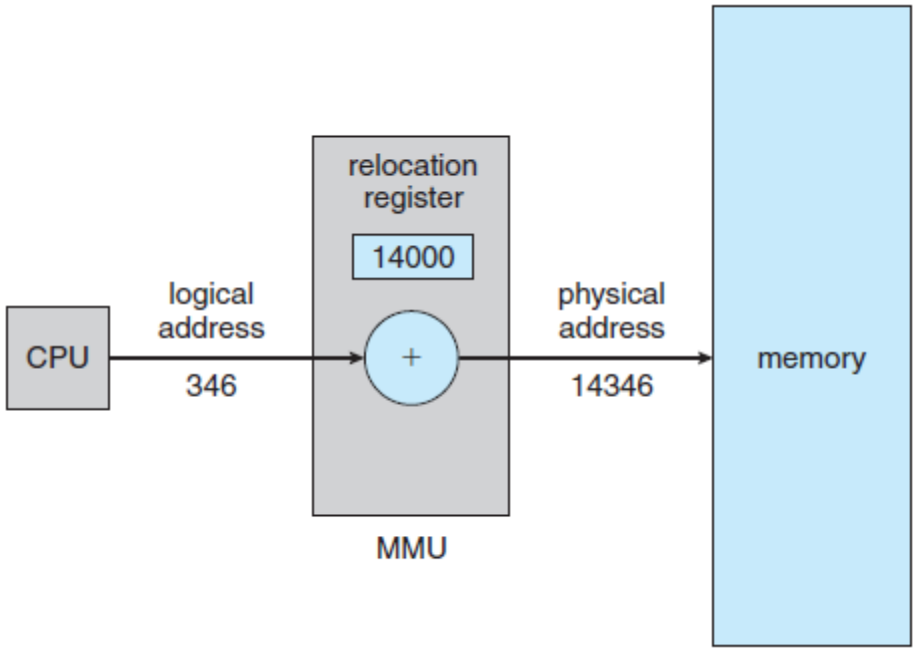


Figure: Dynamic relocation using a relocation register

However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.We can choose from many different methods to accomplish such mapping,. For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme The base register is now called a **relocation register.** The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346. The user program never sees the real physical addresses. The program can create a pointer to

location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base

register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. The final location of a referenced memory address is not determined until the reference is made. We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R*). The user program generates only logical addresses and thinks that the process runs in locations 0 to *max.* However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

4.4.1 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar. We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process. Memory Allocation Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of

partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT)but is no longer in use.

The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes. As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue. At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met. In general, as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are

processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

4.5 Paging

Segmentation permits the physical address space of a process to be noncontiguous.

Paging is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems, from those for mainframes through those for smart phones. Paging is implemented through cooperation between the operating system and the computer hardware. A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory. Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

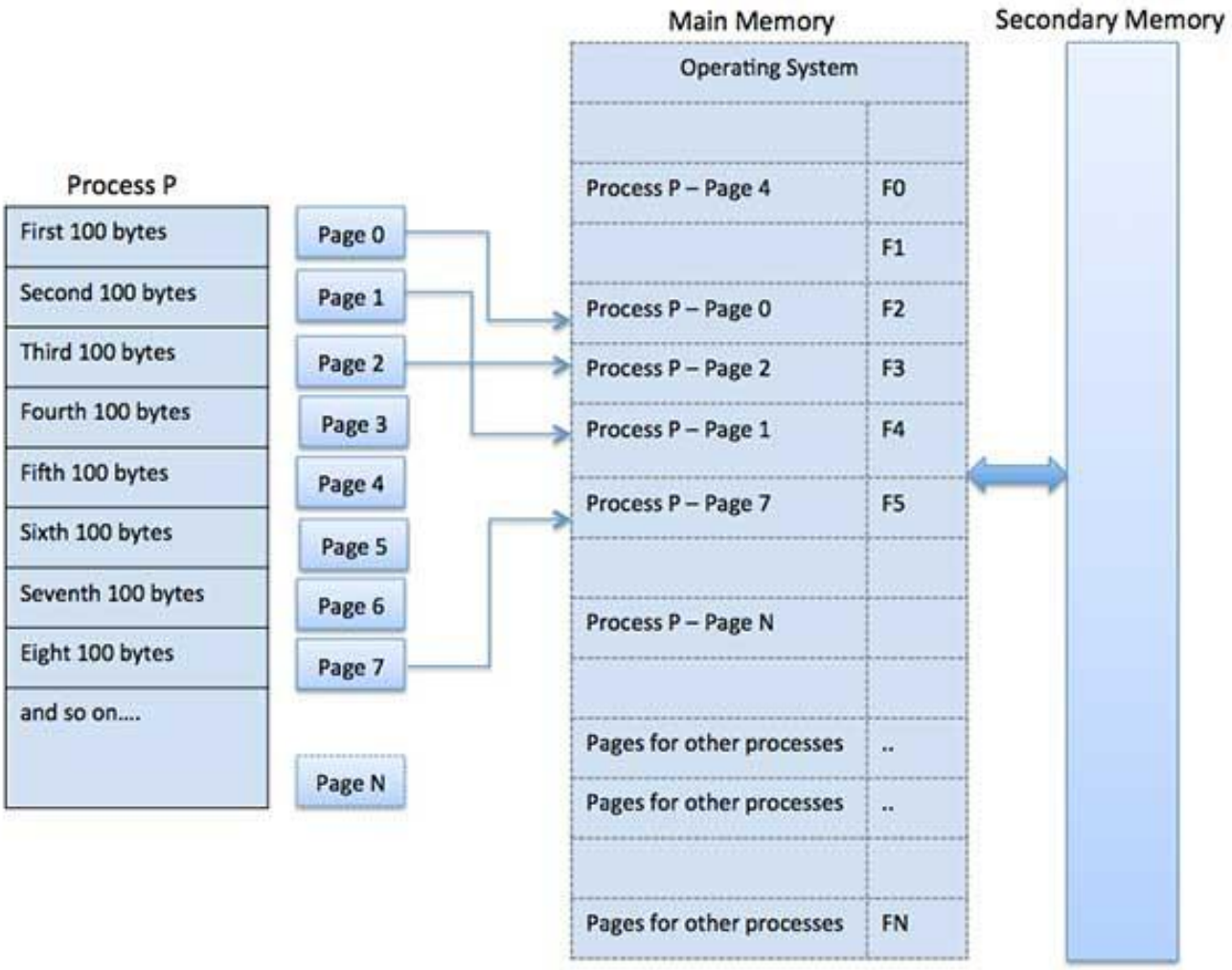


Figure: Paging with main memory

Hardware Support

Each operating system has its own methods for storing page tables. Some allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table. Other operating systems provide one or at most a few page tables, which decreases the overhead involved when processes are context-switched. The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make

the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers. The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time. The problem with this approach is the time required to access a user memory location. If we want to access location *i,* we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, ***two*** memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024

entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches. The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

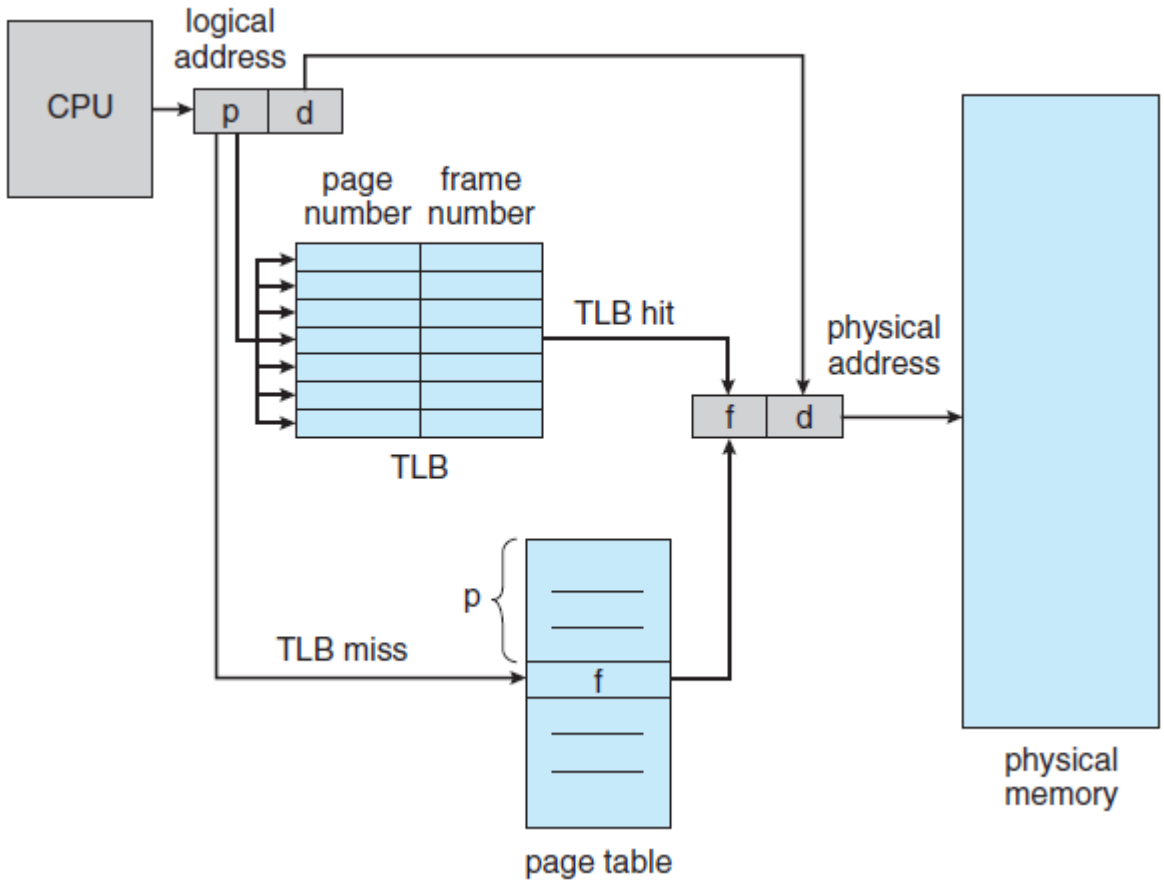


Figure: Paging hardware with TLB

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in

hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down. Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process. The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

Effective access time = 0.80 × 100 + 0.20 × 200

= 120 nanoseconds

In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds).

For a 99-percent hit ratio, which is much more realistic, we have

$$\text{Effective access time} = 0.99 \times 100 + 0.01 \times 200$$
$$= 101 \text{ nanoseconds}$$

This increased hit rate produces only a 1 percent slowdown in access time. As we noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work. A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs).We will further explore the impact of the hit ratio on the TLB. TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between generations of Intel CPUs)may necessitate a change in the paging implementation of the operating systems that use it.

Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.

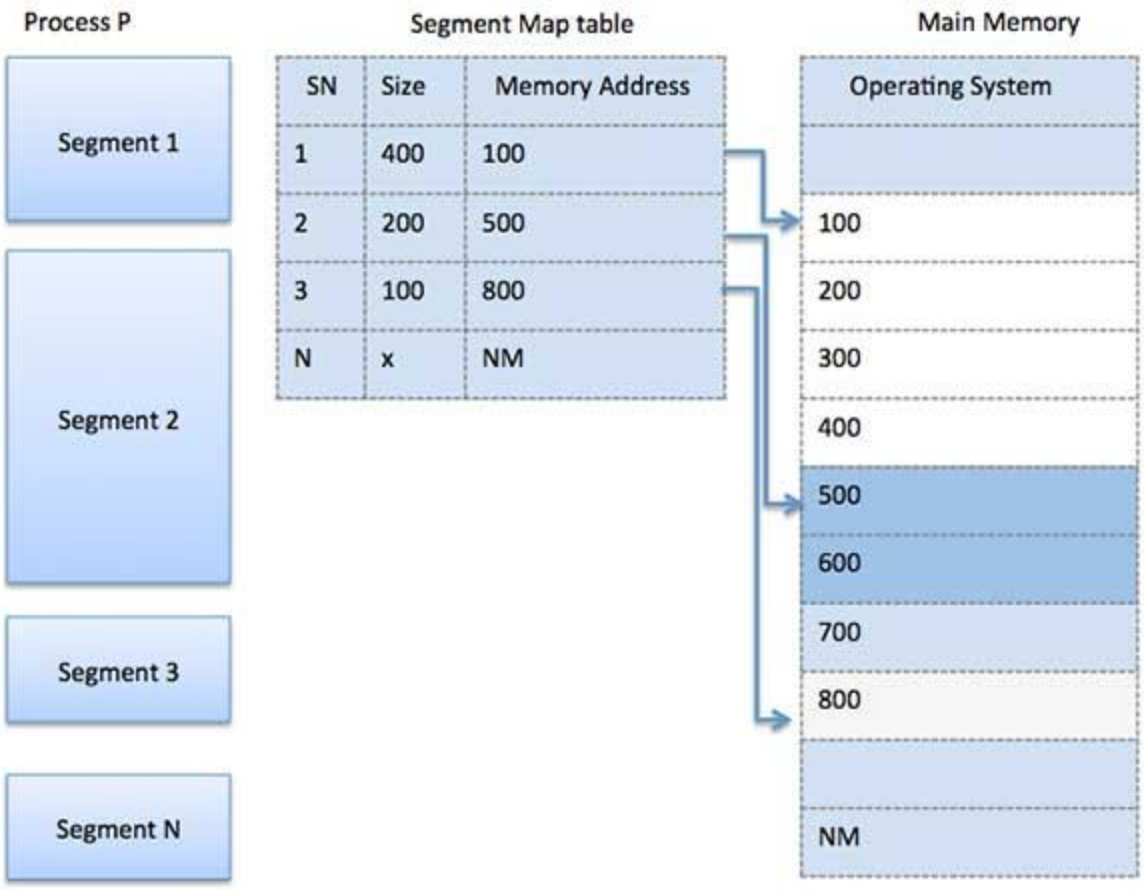


Figure: Segment map table

Check your progress

Q1. Fill in the blanks

1. Physical memory is broken into fixed-sized blocks called _____
2. Paging increases the _____ time.
3. The _____ is used as an index into the page table.
4. The _____ table contains the base address of each page in physical memory.

Q2. True or false.

1. For every process there is a page table.
2. Each entry in a Translation lookaside buffer (TLB) consists of a bit value.
3. The percentage of times a page number is found in the TLB is known as hit ratio.

4.6 Virtual memory

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

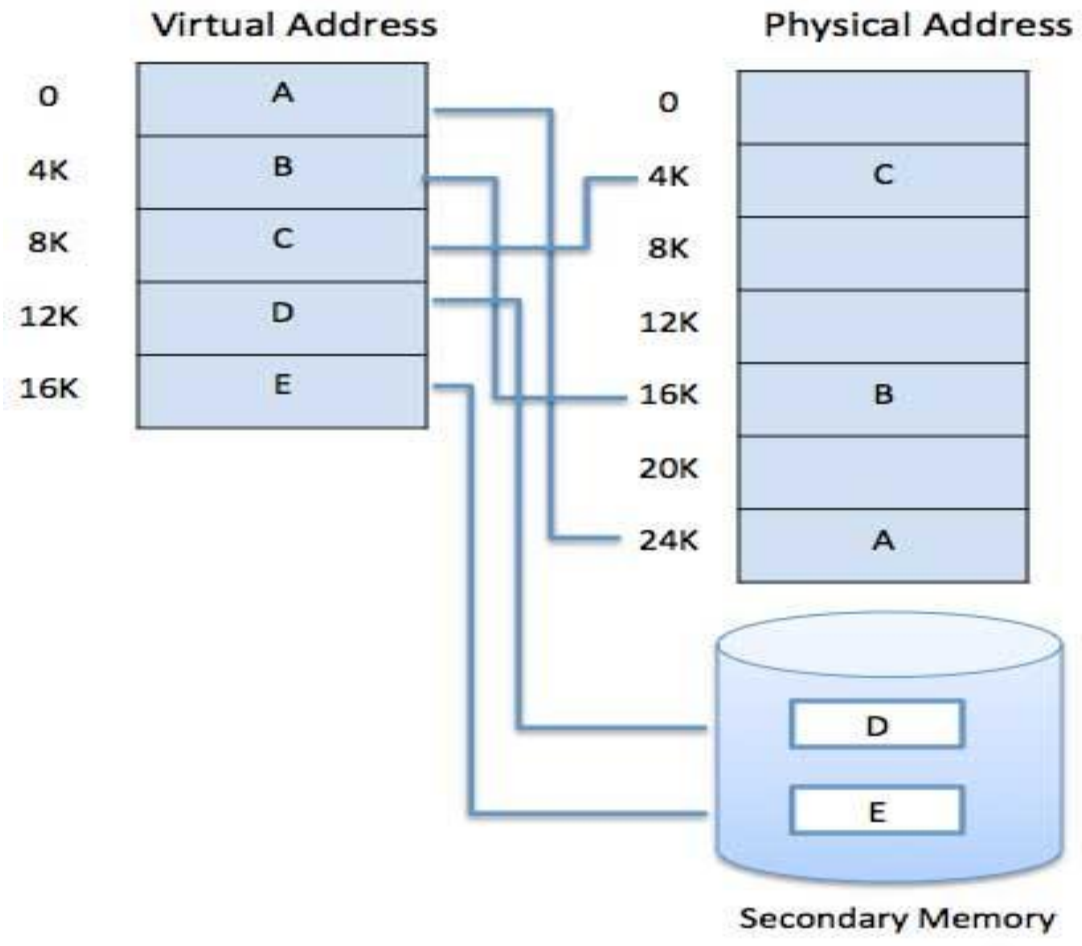


Figure : virtual address and physical address using secondary memory

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory. The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

• Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed. • Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.

• Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years. Even in those cases where the entire program is needed, it may not all be needed at the same time. The ability to execute a program that is only partially in memory would confer many benefits:

• A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large ***virtual*** address space, simplifying the programming task. • Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.

• Less I/O would be needed to load or swap user programs into memory, so each user program would run faster. Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

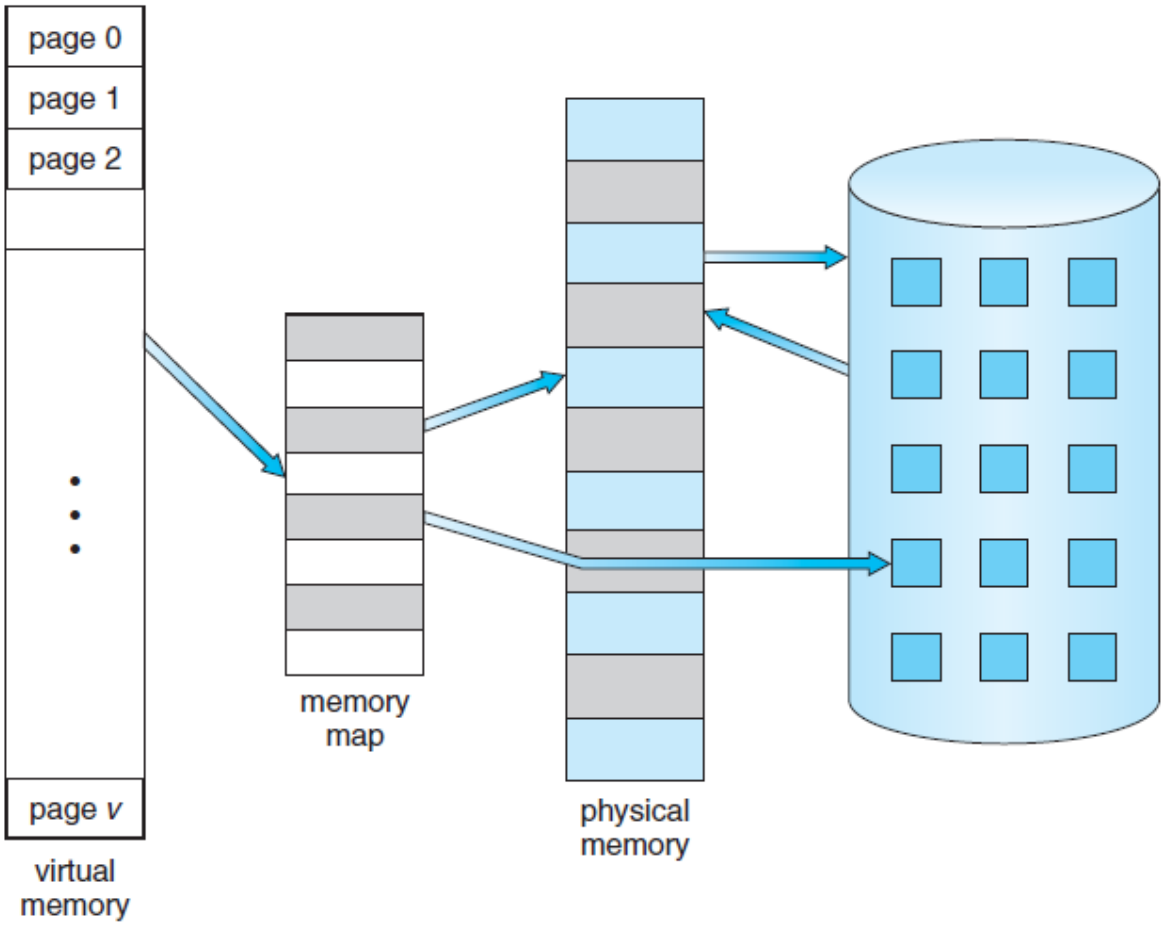


Figure: Diagram showing virtual memory that is larger than physical memory

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure 9.2. Recall from Chapter 8, though, that in fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.

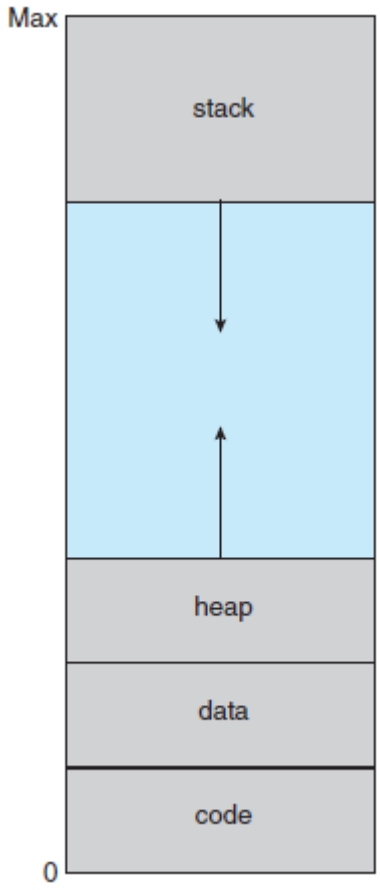


Figure: Virtual address space

4.7 Page Replacement

If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used). If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages,

resulting in a need for sixty frames when only forty are available. Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory. Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are ***no*** free frames on the free-frame list; all memory is in use. The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is not the best choice. The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances, and we consider it further in. Here, we discuss the most common solution: **page replacement**.

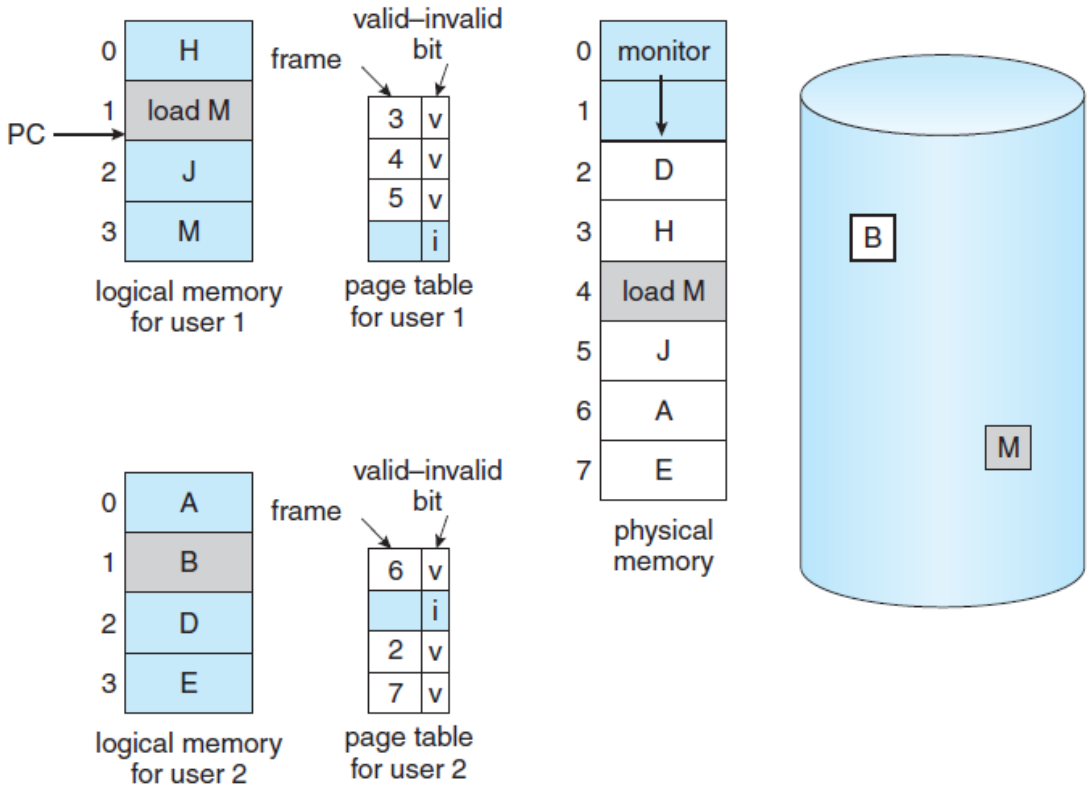


Figure: Need for page replacement

PAGE REPLACEMENT ALGORITHMS

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

While it would be possible to pick a random page to replace at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back

in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

4.5.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shortest job first scheduling algorithm how can the system tell which job is shortest?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the second run by using the page reference information collected during the first run.

In this way it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say, only 1 percent worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1 percent improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured. The page replacement algorithm derived from it is thus specific to that one program. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that are useful on real systems.

In order to allow the operating system to collect useful statistics about which pages are being used and which ones are not, most computers with virtual memory have two status bits associated with each page. R is set whenever the page is referenced (read or written). M is set when the page is written to (i.e., modified). The bits are contained in each page table entry. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it to 0 in software.

If the hardware does not have these bits, they can be simulated as follows. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the R bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently written on, another page fault will occur, allowing the operating system to set the M bit and change the page's mode to READ/WRITE.

The R and M bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the R bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their R and M bits. Class 0 : not referenced, not modified. Class 1 : not referenced, modified. Class 2 referenced, not modified. Class 3: referenced, modified. Although class I pages seem, at first glance, impossible, they occur when a class 3 page has its R bit cleared by a

clock interrupt. Clock interrupts do not clear the M bit because this information is needed to know whether the page has to be rewritten to disk or not.

The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, efficient to implement, and gives a performance that, while certainly not optimal, is often adequate.

The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the FIFO (First-In, First-Out) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly k different products. One day, some company introduces a new convenience food-instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises. For this reason, FIFO in its pure form is rarely used.

4.5.4 The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

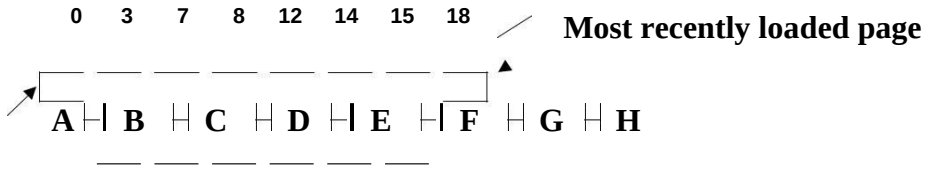


Figure 4-12.Operation of second chance, (a) Pages sorted in FIFO order, (b) Page list if a page fault occurs at line 20 and A has its A bit set

Summary

Memory-management algorithms for multiprogrammed operating systems range from the simple single-user system approach to segmentation and paging. The most important determinant of the method used in a particular system is the hardware provided. Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address. The checking cannot be implemented (efficiently) in software. Hence, we are constrained by the hardware available. The various memory-management algorithms (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In comparing different memory-management strategies, we use the following considerations:

• **Hardware support**. A simple base register or a base–limit register pair is sufficient for the single- and multiple-partition schemes, whereas paging and segmentation need mapping tables to define the address map.

• **Performance**. As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only compare or add to the

logical address—operations that are fast. Paging and segmentation can be as fast if the mapping table is implemented in fast registers. If the table is in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.

• **Fragmentation**. A multiprogrammed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by

packing more processes into memory. To accomplish this task, we must reduce memory waste, or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

• **Relocation**. One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory in such a way that the program does not notice the change. This consideration

requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

• **Swapping**. Swapping can be added to any algorithm. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time. In general, PC operating systems support paging, and operating systems for mobile devices do not.

• **Sharing**. Another means of increasing the multiprogramming level is to share code and data among different processes. Sharing generally requires that either paging or

segmentation be used to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.

• **Protection**. If paging or segmentation is provided, different sections of a user program can be declared execute-only, read-only, or read–write. This restriction is necessary with shared code or data and is generally useful in any case to provide simple run-time checks for common programming errors.

Virtual memory is a technique that enables us to map a large logical address space onto a smaller physical memory. Virtual memory allows us to run extremely large processes and to raise the degree of multiprogramming, increasing CPU utilization. Further, it frees application programmers from worrying about memory availability. In addition, with virtual memory, several processes can share system libraries and memory. With virtual memory, we can also use an efficient type of process creation known as copy-on-write, wherein parent and child processes share actual pages of memory. Virtual memory is commonly implemented by demand paging. Pure demand paging never brings in a page until that page is referenced. The first reference causes a page fault to the operating system. The operating-system kernel consults an internal table to determine where the page is located on the backing store. It then finds a free frame and reads the page in from the backing store. The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even though its entire memory image is not in main memory at once. As long as the page-fault rate is reasonably low, performance is acceptable.

Check your progress

Fill in the blanks

1. CPU fetches the instruction from memory according to the value of _____.
2. A memory buffer used to accommodate a speed differential is called_____.
3. Run time mapping from virtual to physical address is done by _____.
4. The address of a page table in memory is pointed by _____.

Check your progress – Answers

Q1. Fill in the blanks

1. Frames
2. Context-switch
3. Page number
4. Page

Q2. True or false

1. True
2. False
3. True

Q1. Fill in the blanks.

1. Program counter
2. Cache
3. Memory management unit
4. Page table base register.

QUESTIONS FOR SELF - STUDY

1. Explain logical and physical address space.
2. Define TLB.
3. Describe virtual memory.
4. Explain various algorithms used for page replacement.
5. Describe paging in virtual memory.
6. Explain segmentation with paging.
7. Distinguish between contiguous and non contiguous address binding.

SUGGESTED READINGS

3. **Operating System Concepts b**y Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
4. **Operating systems** By Stuart E. Madnick, John J. Donovan

References used:

11. Abraham-Silberschatz-Operating-System-Concepts---9th edition

12. Operating systems by William Stallings

13. Operating systems fundamental concepts

14. www.tutorialspoint.com

15. Operating systems by Achyut Godbole

Notes

Chapter 5:

File Systems

5.0 Objectives

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

5.1 Introduction

A *file system* is the part of the operating system that is responsible for managing files and the resources on which these reside. Without a file system, efficient computing would essentially be impossible. This chapter discusses the

organization of file systems and the tasks performed by the different components. The first part is concerned with general user and implementation aspects of file management emphasizing centralized systems; the last sections consider extensions and methods for distributed systems.

Given that main memory is volatile, i.e., does not retain information when power is turned off, and is also limited in size, any computer system must be equipped with secondary memory on which the user and the system may keep information for indefinite periods of time. By far the most popular secondary memory devices are disks for random access purposes and magnetic tapes for sequential, archival storage. Since these devices are very complex to interact with, and, in multiuser systems are shared among different users, operating systems (OS) provide extensive services for managing data on secondary memory. These data are organized into *files*, which are collections of data elements grouped together for the purposes of access control, retrieval, and modification.

5.1.1 BASIC FUNCTIONS OF FILE MANAGEMENT

The file system, in collaboration with the I/O system, has the following three basic functions:

1. Present a *logical* or *abstract view* of files and directories to the users by hiding the physical details of secondary storage devices and the I/O operations for communicating with the devices.

2. Facilitate *efficient use* of the underlying storage devices.

3. Support the *sharing of files* among different users and applications. This includes providing *protection* mechanisms to ensure that information is exchanged in a controlled and secure manner.

The first function exists because physical device interfaces are very complex. They also change frequently as new devices replace outdated ones. The I/O system, provides the first level of abstraction on top of the hardware devices. It presents an interface where devices may be viewed as *collections or streams of logical blocks*,

which can be accessed sequentially or directly, depending on the type of device. This level of abstraction is still too low for most applications, which must manipulate data in terms of named collections of data records—**files**—and organize these into various hierarchical structures using **directories** (sometimes called **folders**). Thus, the role of the files system is to extend the logical I/O device abstraction to a file-level abstraction. A file is defined as a set of related data items, called **logical records**. It is largely independent of the medium on which it is stored. The logical record is the smallest addressable unit within a file. The purpose of the file concept is to give users one simple uniform linear space for their code and data. Files and records within files then may be manipulated via a set of high-level operations defined by the **file system interface**, whose implementation is hidden from the user by the file system. The second goal and function of a file system is the efficient use of storage devices on which files that may belong to many different users are kept. Unlike main memory, which has no moving parts and where access to any location takes the same amount of time, most secondary memory devices are electromechanical, are much slower than main memory, and have data access times that depend greatly on the data location on the device.

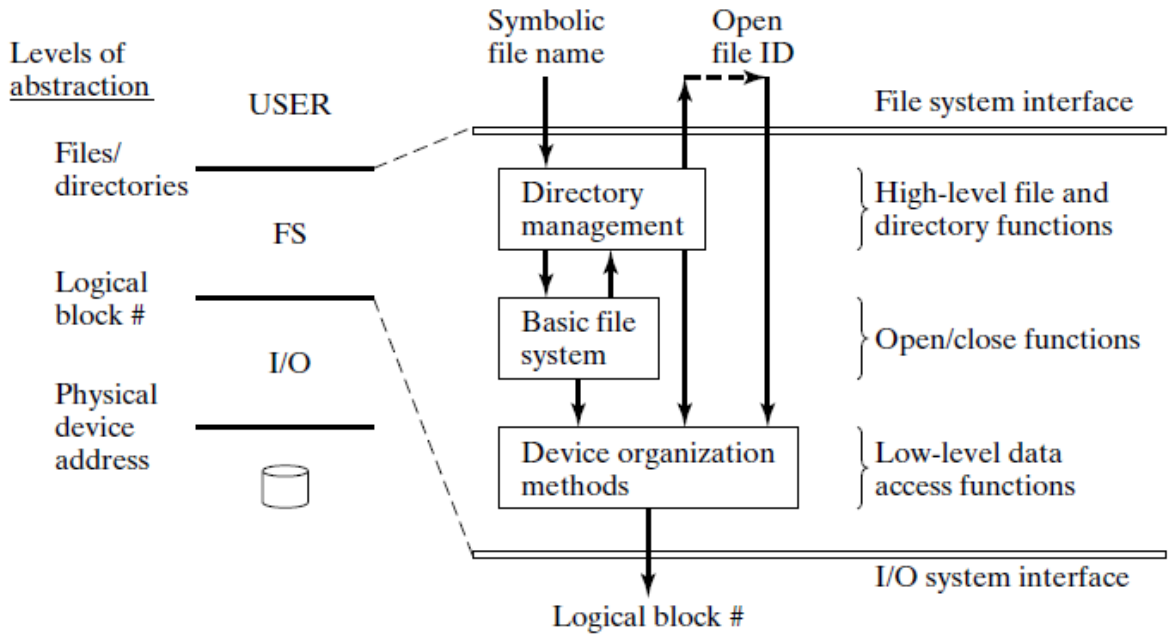
For example, a disk access may or may not require a seek operation, i.e., the physical movement of the read/write head, depending on the head's current position. Since each seek is very time-consuming, the file system must strive for placing data items in such a way as to minimize the read/write head movements. the I/O system then may provide additional optimizations, for example by dynamically reordering the disk accesses to minimize the distance traveled by the read/write head. The situation is even more critical with tapes and other one-dimensional storage devices, where data can be accessed effectively only in the order in which they appear on the tape. The file system must ensure that a data placement appropriate for the underlying storage device is chosen for any given file. The third reason for providing a file system is to facilitate file sharing. This is similar to the sharing of code and data in main memory, where two or more processes have simultaneous access to the same portion of main memory. The main difference is that files, unlike main memory information, generally persist in the secondary memory even when its owner or creator is not currently active, e.g., the owner may be logged out. Thus, sharing of files may not be concurrent.

A process may wish to access a file written earlier by another process that no longer exists. The key issue is protection, i.e. controlling the type of access to a file permitted by a given process or user. In this chapter, we are concerned primarily with the first two functions of a file system, which address the need for making details of the physical storage devices transparent to users and their programs. The problems of protection and security are treated separately

5.1.2 HIERARCHICAL MODEL OF A FILE SYSTEM

The file system, like any other complex software system, can be decomposed into several distinct parts according to primary function. We will adopt a hierarchical organization, where each level represents a successively more abstract machine in which a module at a given level may call upon the service of only modules at the same or lower levels. Although not all file systems are designed according to a strict hierarchy, this conceptual view is useful because it permits us to understand the complex functionalities of the file system and its interfaces to both the user and the I/O system by studying each level in isolation. The left-hand side illustrates the file system's position as a layer between the user and the I/O system.

For the user, it creates the abstraction of logical files, which can be manipulated and organized using various file and directory commands. It manages these on the underlying disks and other secondary memory devices by calling upon the services of the I/O system and exchanging data with it using logical block numbers. The responsibility of the I/O system is to translate the logical block numbers into actual disk or tape addresses. The remainder of the file system subdivided into three main components in a hierarchy. Below, we briefly outline the tasks within each level, proceeding from the abstract user interface down to the file system and I/O interface. Succeeding sections describe the functions of each level in more detail.



5.1.3 The File System Interface

The file system presents the abstraction of a **logical file** to the user, and defines a set of operations to use and manipulate these. It also provides a way to organize individual files into groups and to keep track of them using **directories**. Typically, a file can be identified using two forms of names, depending on the operation to be performed on that file. A **symbolic name**, chosen by the user at file creation, is used to find a file in a directory. It also is used when manipulating the file as a whole, e.g., to rename or delete it. Symbolic names, however, are not convenient to use within programs, especially when used repeatedly.

Thus, operations that access the file data (notably read and write operations) use a numeric identifier for the file. This is generated by lower levels of the file system. When a desired file is first opened by setting up data structures necessary to facilitate its access, an identifier, the **open file ID**, is returned and used for subsequent read, write, and other data manipulation operations. Note that the symbolic name remains valid even when the file is not in use, whereas the open file ID is only temporary. It is valid only when the process that opened the file remains active or until the file is explicitly closed.

Directory Management

The primary function of this level is to use the symbolic file name to retrieve the **descriptive information** necessary to manipulate the file. This information is then passed to the basic file system module, which opens the file for access and returns the corresponding open file ID to the directory management module. Depending on the user-level command being processed, the open file ID may be returned to the user for subsequent read/write access to the file, or it may be used by the directory management module itself. For example, a search through the hierarchy structure requires that the directory management repeatedly opens and reads subdirectories to find a desired file. At each step of the search, the directory management routines must call upon the lower-level basic file system to open these subdirectories.

Basic File System This part activates and deactivates files by invoking **opening** and **closing** routines. It is also responsible for verifying the **access rights** of the caller on each file request. The basic file system maintains information about all open files in main memory data structures called **open file tables** (OFT). The open file ID that is returned to the caller when a file is opened points directly to the corresponding entry in an open file table. Using this ID instead of the symbolic file name allows subsequent read and write operations to bypass the directory management and basic file system modules when accessing the file data.

Device Organization Methods

This module performs the mapping of the logical file to the underlying blocks on the secondary memory device. It receives read and write requests from the higher-level routines or the user interface. The device organization methods translate each request into the corresponding block numbers and pass these on to the underlying I/O system, this then carries out the actual data transfers between the device and the caller's main memory buffers. The allocation and deallocation of storage blocks and main memory buffers is also handled at this level.

THE USER'S VIEW OF FILES

From the user's point of view, a file is a named collection of data elements that have been grouped together for the purposes of access control, retrieval, and modification. At the most abstract, logical level, any file is characterized by its **name**, its **type**, its logical **organization**, and several additional **attributes**.

5.1.4 File Names and Types

What constitutes a legal file name varies between different file systems. In older systems, the name length was limited to a small number of characters. For example, MS-DOS allows up to eight characters, whereas older versions of UNIX support 14 characters.

Certain special characters are frequently disallowed as part of the file name because they are reserved to serve a special role within the OS. This includes the blank, since this is usually a delimiter for separating commands, parameters, and other data items.

More recent systems have relaxed these restrictions, allowing file names to be arbitrary strings of characters found on most keyboards, and supporting name lengths Those, for most practical purposes, are unlimited. For example, MS Windows allows up to
255 characters, including the blank and certain punctuation characters.

Many systems, including MS-DOS or Windows 95, do not differentiate between lower- and uppercase characters. Thus *MYFILE, Myfile, MyFile, myfile*, or any other combination of the upper- and lowercase letters all refer to the same file. In contrast, UNIX and Linux differentiate between lower- and uppercase; thus, the above names would each refer to a different file. An important part of a file name is its **extension**, which is a short string of additional characters appended to the file name. In most systems, it is separated from the name by a period. File extensions are used to indicate the file **type**.

For example, *myfile.txt* would indicate that the file is a text file, and *myprog.bin* would indicate that this file is an executable binary program (a load module). A typical file extension is between one and four characters in length. Older systems, e.g., MS-

DOS, are more rigid, requiring an extension of one to three characters, while UNIX, Linux, and many other more recent systems allow more flexibility. The file type implies the *internal format* and the *semantic meaning* of the file contents. In the simplest case, the file is a sequence of ASCII characters with no other format imposed on it. Such files are generally called *text* files and carry the extension *.txt*. A file containing a source program generally carries the extension indicating the programming language. For example, *main.c* might be a C program, whereas *main.f77*would be a Fortran 77 program. When compiled, the resulting files are object modules, denoted by the extension *.o* or *.obj*. A linker then takes an object module and produces the corresponding load module, a binary file, either annotated with the extension *.bin, .com*, or, as in the case of UNIX, left without an extension.

The file extension generally denotes the type of the file; the type, in turn, determines the kinds of operations that may be meaningfully performed on it by a program. Note, however, that it is the file type and *not* the extension that determines the file's semantic meaning. In most cases, the extension is only an annotation for the *convenience* of the user and the various applications using the files. That means, most systems allow the extension to be changed at will, but that does not change the type of the file, which is inherent to its internal format. A compiler could still process a source code file, even if its extension was changed. However, the compiler would fail when presented with an object module (even if this had the proper extension), since only a linker is able to correctly interpret the contents of an object module.

The file type is usually stored in the form of a file **header**.

For example, any executable UNIX file must begin with a specific "magic number"—a code indicating that this is indeed an executable file. This is followed by information on how to find the code, the data, the stack, the symbol table areas, the starting point within the code, and other information needed to load and execute this file.

The number of file types supported and the strictness of enforcing the correspondence between the file type and the file name extension depends on the OS. Every system must support a minimal set of file types, including text files, object files, and load module files. In addition, the file system must be able to distinguish between ordinary files and file directories. Other utilities and applications establish and follow

their own conventions regarding file types, their internal formats, and extensions. For example, a *.doc* file would normally be a formatted text document understandable by *MS Word*, *.ps* would be a file understandable by a *Postscript* printer, and *.html* would be a file containing information in the *Hypertext Markup Language* understandable by a *Web* browser.

5.1.5 Logical File Organization

At the highest level, the file system is concerned with only two kinds of files, directories and ordinary (non-directory) files. For directories, it maintains its own internal organization to facilitate efficient search and management of the directory contents. For ordinary files, the file system is concerned either with the delivery of the file contents to the calling programs, which can then interpret these according to their type, or with the writing of information to files. Traditionally, a file system views any file as a sequence of **logical records** that can be accessed one at a time using a set of specific operations. These operations also are referred to as **access methods**, since they determine how the logical records of a given file may be accessed. A logical record in a file is the smallest unit of data that can read, written, or otherwise manipulated by one of the access method operations.

Logical Records

Logical records may be of **fixed** or **variable length**. In the first case, a logical record can be a single byte, a word, or a structure of arbitrarily nested fixed-length components. Regardless of the complexity of each record, all records within a file must have the same length. In the second case, records may have different lengths; the size or length of a record is typically recorded as part of each record so that access method operations can conveniently locate the end of any given record. Fixed-length record files are easier to implement and are the most common form supported by modern OSs.

Record Addressing

To access a record within a file, the operation must address it in some way. This can be done either **implicitly** or **explicitly**. For implicit addressing, all records are accessed in

the sequence in which they appear in the file. The system maintains an internal pointer to the current record that is incremented whenever a record is accessed. This **sequential** access is applicable to virtually all file types. Its main limitation is that a given record i can only be accessed by reading or scanning over all preceding records 0 through $i - 1$. Explicit addressing of records allows **direct** or random access to a file. It can be done either by specifying the record **position** within the file or by using a specific field within the record, called the **key**. In the first case, we assume that the file is a sequence of records such that each record is identified uniquely by an integer from 1 to n (or 0 to $n - 1$), where n is the number of logical records comprising the file. An integer within that range then uniquely identifies a record. In the second case, the user must designate one of the fields within a record as the key. This can be a variable of any type, but all key values within a file must be unique. A typical example is a social security number. A record is then addressed by specifying its key value.

Each OS implements its own view of what specific attributes should be maintained. The following types of information are generally considered:

- **Ownership.** This records the name of the file owner. In most systems, the file creator is its owner. Access privileges are usually controlled by the owner, who may then grant them to others.
- **File Size.** The file system needs current file size information to effectively manage its blocks on disk. File size is also important for the user, and hence is readily available at the user level, typically as one of the results of a directory scan operation.
- **File Use.** For a variety of reasons, including security, recovery from failure, and performance monitoring, it is useful to maintain information about when and how the file has been used. This may include the time of its creation, the time of its last use, the time of its last modification, the number of times the file has been opened, and other statistical information.
- **File Disposition.** A file could be *temporary*, to be destroyed when it is closed, when a certain condition is satisfied, or at the termination of the process for which it was created; or it may be stored indefinitely as a *permanent* file.
- **Protection.** This information includes *who* can access a file and *how* a file can

be accessed, i.e., the *type of operation*, such as read, write, or execute, that can be performed. Enforcing protection to files is one of the most important services provided by the file system. are dedicated exclusively to issues of protection and security.

• **Location.** To access the data in a file, the file system must know which device blocks the file is stored in. This information is of no interest to most users and is generally not available at the abstract user interface. The different schemes for mapping a file onto disk blocks and keeping track of them affect performance crucially

5.2 OPERATIONS ON FILES

The abstract user interface defines a set of operations that the user may invoke—either at the command level or from a program—to manipulate files and their contents. The specific set of operations depends on the OS. It also depends on the type of files that must be supported. Below we present an overview of the commonly provided classes of operations.

5.2.1 Create/Delete

Before a file can be used in any way, it must be created. The *create* command, which generally accepts a symbolic file name as one of its parameters, creates the file identity so that it can be referred to in other commands by its symbolic name. The *delete* or *destroy* command reverses the effect of the creation by eliminating the file and all its contents from the file system. Since deleting a file by mistake is a common problem with potentially serious consequences, most file systems will ask for confirmation before actually performing the operation. Another common safety measure is to delete any file only tentatively so that it can later be recovered if necessary. For example, the Windows OS will place a file to be deleted into a special directory called the *Recycle Bin*, where it remains indefinitely. Only when the *Recycle Bin* is explicitly emptied, e.g., to free up disk space, is the file deleted irrevocably.

5.2.2 Open/Close

A file must be opened before it can be used for reading or writing. The *open* command sets up data structures to facilitate the read and write access. An important role of the *open* command is to set up internal buffers for transferring the file data between the disk and main memory. The *close* command reverses the effect of *open*; it disconnects the file from the current process, releases its buffers, and updates any information maintained about the file.

5.2.3 Read/Write

A *read* operation transfers file data from disk to main memory, whereas a *write* operation transfers data in the opposite direction. The *read/write* operations come in two basic forms, *sequential* and *direct*, corresponding to the implicit and explicit forms of record addressing A direct *read* or *write* will access a record that must be designated explicitly by its number (i.e., position within the file), or by its key. A sequential *read* or *write* assumes the existence of a logical pointer that is maintained by the system and always points to the record to be accessed next. Each time a sequential *read* operation is executed, it transfers the current record and advances the pointer to the next record. Thus, if the last record read was record *i*, issuing the same *read* command repeatedly will access a sequence of consecutive records *i*+1, *i*+2, and so on. Similarly, a sequential *write* command will place a new record at the position of the current pointer. If this position already contains a record, the operation overwrites it; if the pointer is at the end of the file, the write expands the file.

5.2.4 Seek/Rewind

Purely sequential access is too restrictive for many applications. To avoid having to read a file each time from the beginning, a *seek* command is frequently provided. It moves the current record pointer to an arbitrary position within the file. Thus a *seek* followed by a *read* or *write* essentially emulates the effect of a direct access. How the *seek* operation is implemented depends on how the file is organized internally, but it can usually be done much more efficiently than reading all records from the beginning. (Note that the *seek* operation discussed here is a high-level file operation, and should not be confused with a disk seek operation, which moves the disk's read/write head to a

specified track.) A *rewind* command resets the current record pointer to the beginning of the file. This is equivalent to a *seek* to the first record of the file.

5.1 & 5.2 Check your progress

Q1. Fill in the blanks.

1. _____ is a unique tag, usually a number, identifies the file within the file system.
2. Mapping of file is managed by _____.
3. _____ is a sequence of bytes organized into blocks understandable by the system's linker

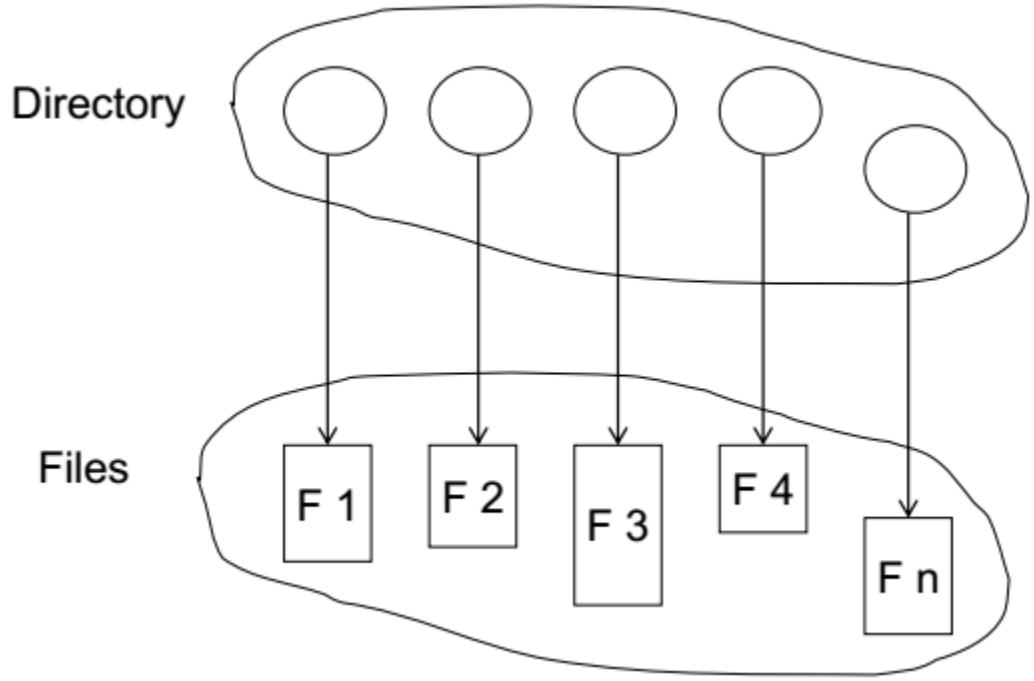
Q2. True or False.

1. If the block of free-space list is free then bit will 1.
2. A file is a/an abstract data type.
3. The file name is generally split into two parts i.e. extension and file name.

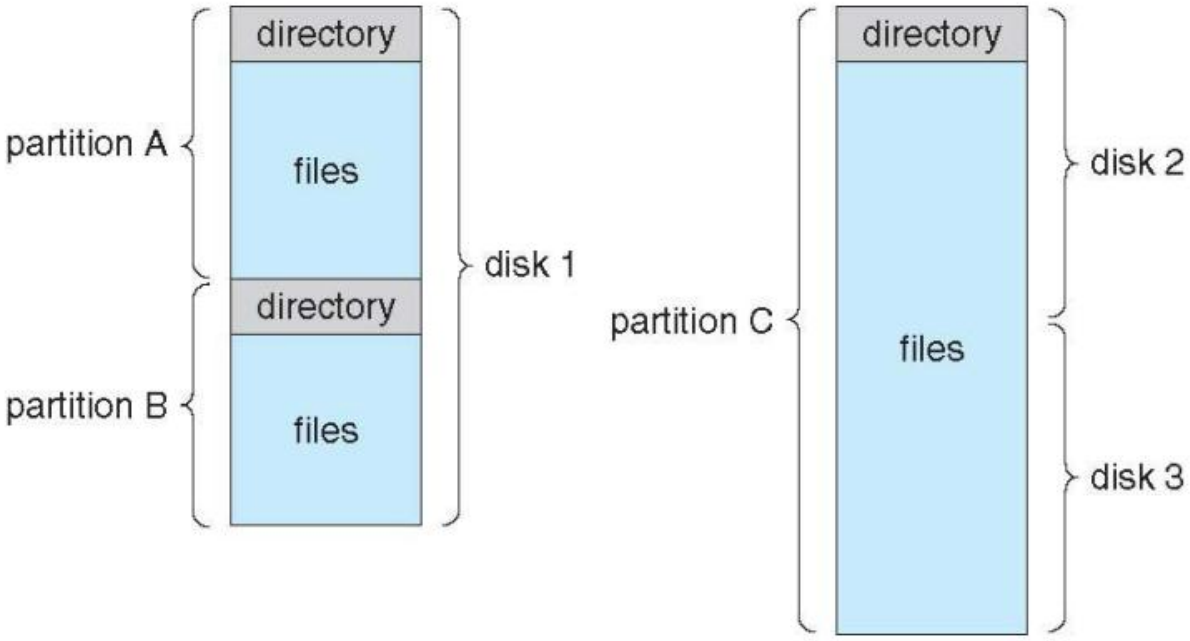
5.3 FILE DIRECTORIES

An OS generally contains many different files. Similarly, every user often has many files. To help organize these in some systematic way, the file system provides **file directories**. Each directory is itself a file, but its sole purpose is to record information about other files, including possibly other directories. The information in a directory associates symbolic names given to files by users with the data necessary to locate and use the files. This data consists of the various attributes, particularly the information necessary to locate the blocks comprising the file on the disk or other storage media. How this location information is organized and where is it kept is the subject we are concerned with the organization of file directories from the user's point of view, and with the operations defined at the user interface to locate and manipulate files. The simplest possible form of a directory is a flat **list of files**, where all files are at the same level, with no further subdivisions. Such a directory is clearly inadequate for most applications. Thus, virtually all file systems support a multilevel **hierarchy**, where a directory may point to lower-level subdirectories, which, in turn, may point to yet lower-level subdirectories, and so on. Depending on the rules enforced when creating new files and

file directories, or when changing the connections between existing ones, we can identify several specific organization within a general hierarchical structure



A Typical File-system Organization



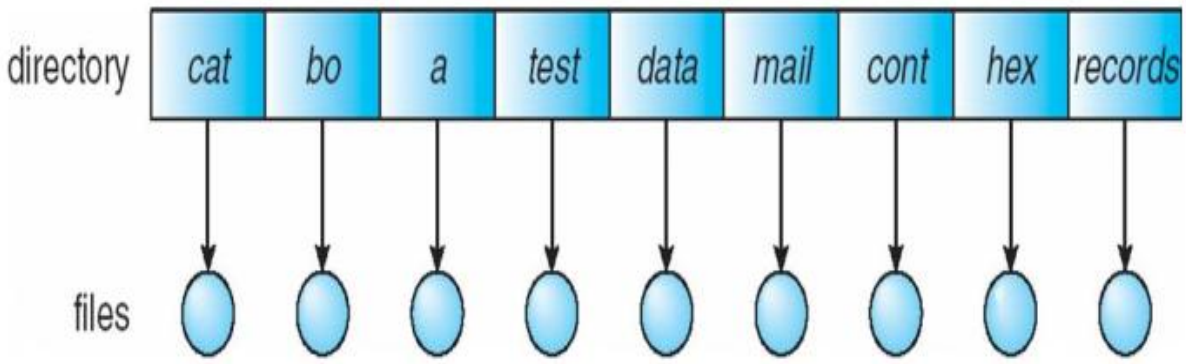
Types of File Systems

We mostly talk of general-purpose file systems. But systems frequently have may file systems, some general- and some special- purpose

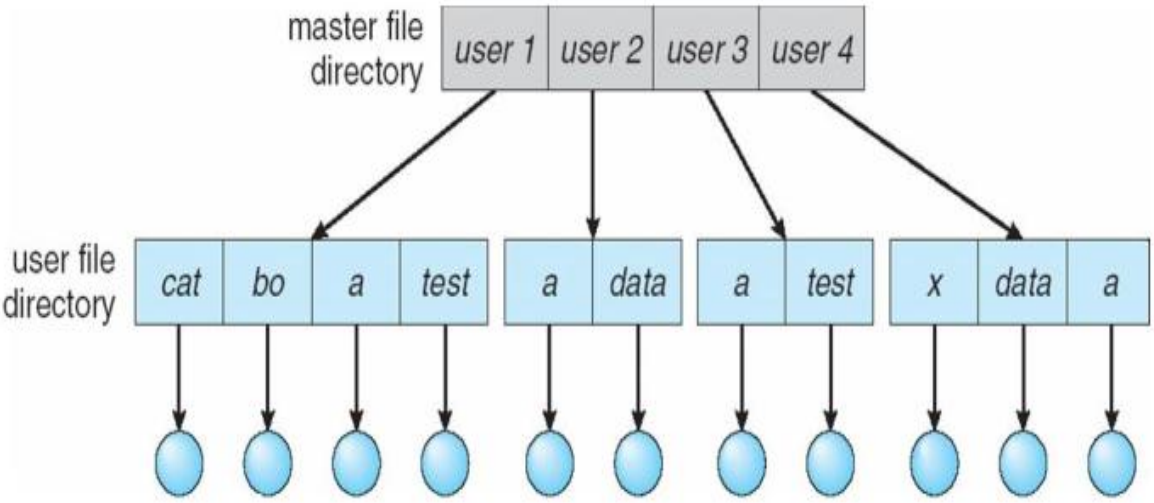
Consider Solaris has :

- tmpfs – memory-based volatile FS for fast, temporary I/O
- objfs – interface into kernel memory to get kernel symbols for debugging
- ctfs – contract file system for managing daemons
- lofs – loopback file system allows one FS to be accessed in place of another
- procfs – kernel interface to process structures
- ufs, zfs – general purpose file systems

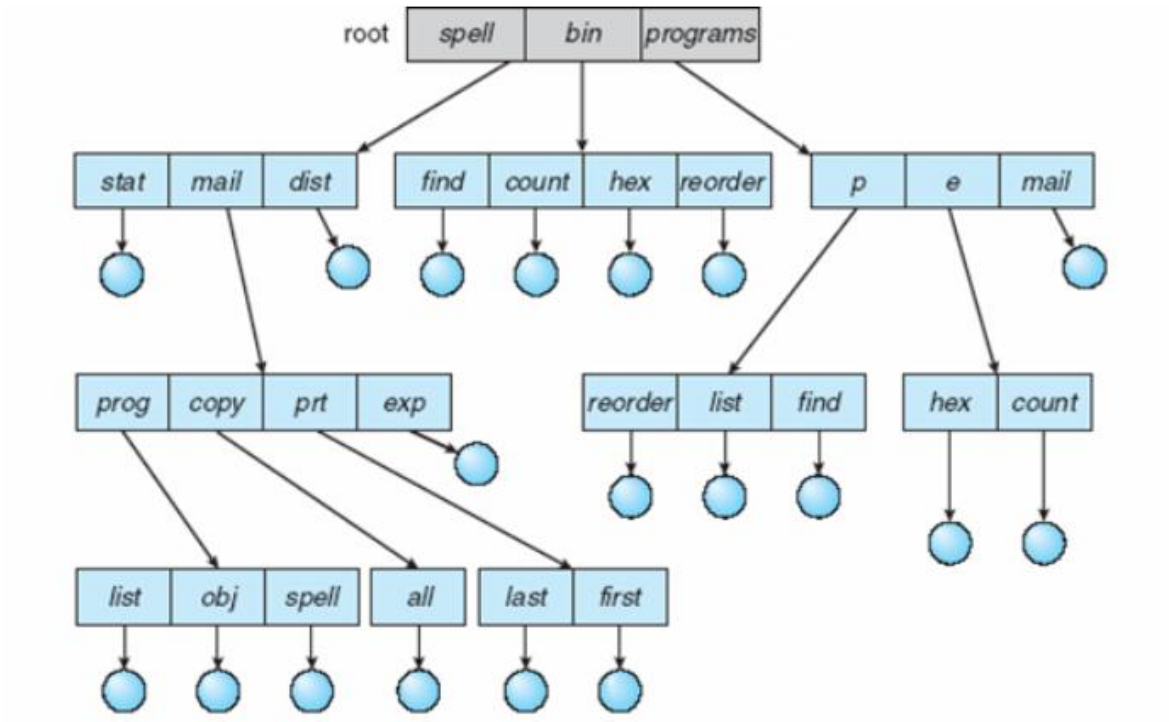
5.3.1 Single-Level Directory



5.3.2 Two-Level Directory



5.3.3 Tree-Structured Directories



BASIC FILE SYSTEM

The main functions of the basic file system are to **open** and **close** files. Opening a file means to set it up for efficient access by subsequent read, write, or other data manipulation commands. Closing a file reverses the effect of the open command and occurs when a file is no longer needed. These and other operations require basic file data structures or descriptors.

File Descriptors

Section 10.3 introduced types of descriptive information associated with a file, including its name, type, internal organization, size, ownership and protection information, and its location on secondary memory. Where this information is kept is an important design decision. One extreme is to maintain the descriptive information dispersed throughout different data structures based on its use by different subsystems or applications. For example, some information can be kept in the parent directory, some on a dedicated portion of the disk, and some with the file itself. The other extreme—a cleaner solution keeps all descriptive information about a file segregated in a separate data structure, pointed to from the parent directory. This data structure is generally referred to as the **file descriptor**.

Opening and closing of files involve two components of the file descriptor:
1) protection information to verify the legality of the requested access;
2) location information necessary to find the file data blocks on the secondary storage device.

Opening and Closing Files

Regardless of how the descriptive information about a file is organized, most of it is maintained on disk. When the file is to be used, relevant portions of this information is brought into main memory for efficient continued access to the file data. For that purpose, the file system maintains an **open file table** (OFT) to keep track of currently open files. Each entry in the OFT corresponds to one open file, i.e., a file being used by one or more processes. The OFT is managed by the **open** and **close** functions. The

open function is invoked whenever a process first wishes to access a file. The function finds and allocates a free entry in the OFT, fills it with relevant information about the file, and associates with it any resources, e.g., read/write buffers, necessary to access the file efficiently. Some systems do not require an explicit *open* command, in which case it is generated implicitly by the system at the time the file is accessed for the first time. When a file is no longer needed, it is closed either by calling the *close* command, or implicitly as the result of a process termination. The *close* function frees all resources used for accessing the file, saves all modified information to the disk, and releases the OFT entry, thereby rendering the file inactive for the particular process.

The implementation of the *open* and *close* functions vary widely with different file systems, but the following list gives the typical tasks performed in some form as part of these functions.

- **Open**

• Using the protection information in the file descriptor, verify that the process (user) has the right to access the file and perform the specified operations.
• Find and allocate a free entry in the OFT.
• Allocate read/write buffers in main memory and other resources as necessary for the given type of file access.
• Complete the components of the OFT entry. This includes initialization information, such as the current position (zero) of a sequentially accessed file. It also includes relevant information copied from the file descriptor, such as the file length and its location on the disk. Additional runtime information, such as the pointers to the allocated buffers or other resources, also is placed into the OFT entry.
• If all of the above operations are successful, return the index or the pointer to the allocated OFT entry to the calling process for subsequent access to the file.

- **Close**

• Flush any modified main memory buffers by writing their contents to the corresponding disk blocks.

• Release all buffers and other allocated resources.

• Update the file descriptor using the current data in the OFT entry. This could include any changes to the file length, allocation of disk blocks, or use information (e.g., date of last access/modification).

• Free the OFT entry.

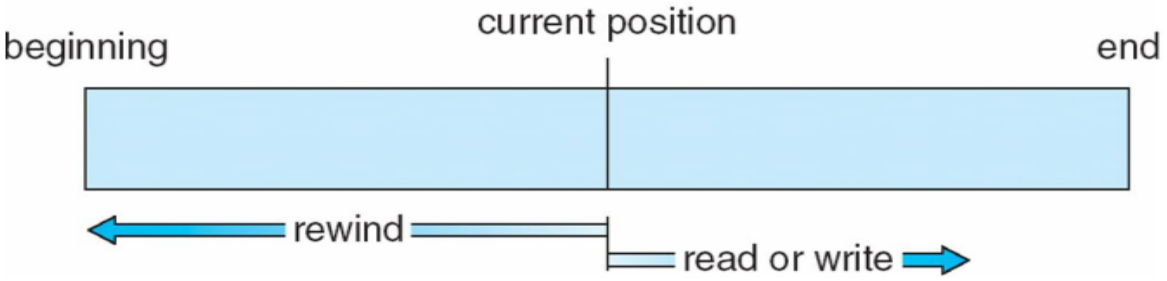
5.4 FILE TYPES – NAME, EXTENSION

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

The minimal information that must be recorded within each entry is the file *symbolic name* and a pointer or *index* to additional descriptive information. At the other extreme, all attributes pertaining to a given file could appear in the directory entry. The disadvantage of the first method is that we need an additional disk operation to access the descriptive information. The disadvantage of the second is that directories could become very large and more difficult to manage.

Furthermore, the entries could be variable in length. Thus, it would be necessary to support a file structure consisting of complex variable-size records. Even with the first minimal approach, we face the problem of having to manage symbolic file names that can vary in length. If the length is limited to a small number of characters, each entry can reserve space for the maximum name length. This is the technique used by most older systems, where file names were limited to eight or another small number of characters. Reserving the maximum space for long name is wasteful. One possible solution is to reserve a relatively small fixed space for each name, followed by a pointer to an overflow heap for long names.

5.4.1 Sequential-access File



5.5 Access Methods

Sequential Access

- read next
- write next
- reset
- no read after last write
- (rewrite)

Direct Access – file is fixed length logical records

- read *n*
- write *n*
- position to *n*
- read next
- write next
- rewrite *n*
- *n*= relative block number

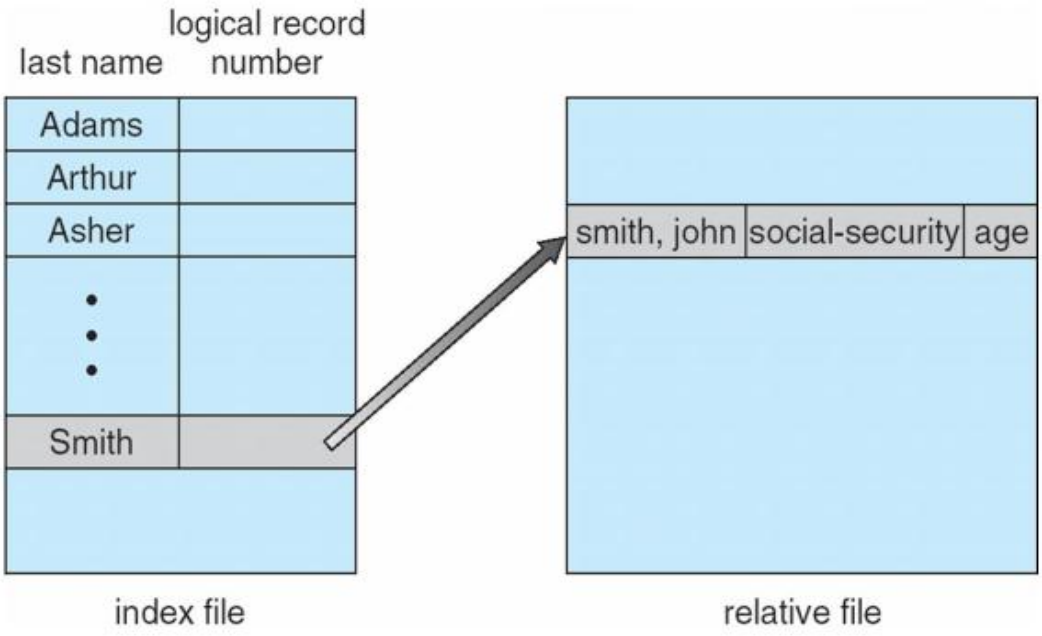


Figure: Example of Index and Relative Files

5.6 Acyclic-Graph Directories

When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the directed arcs from parent to child.)

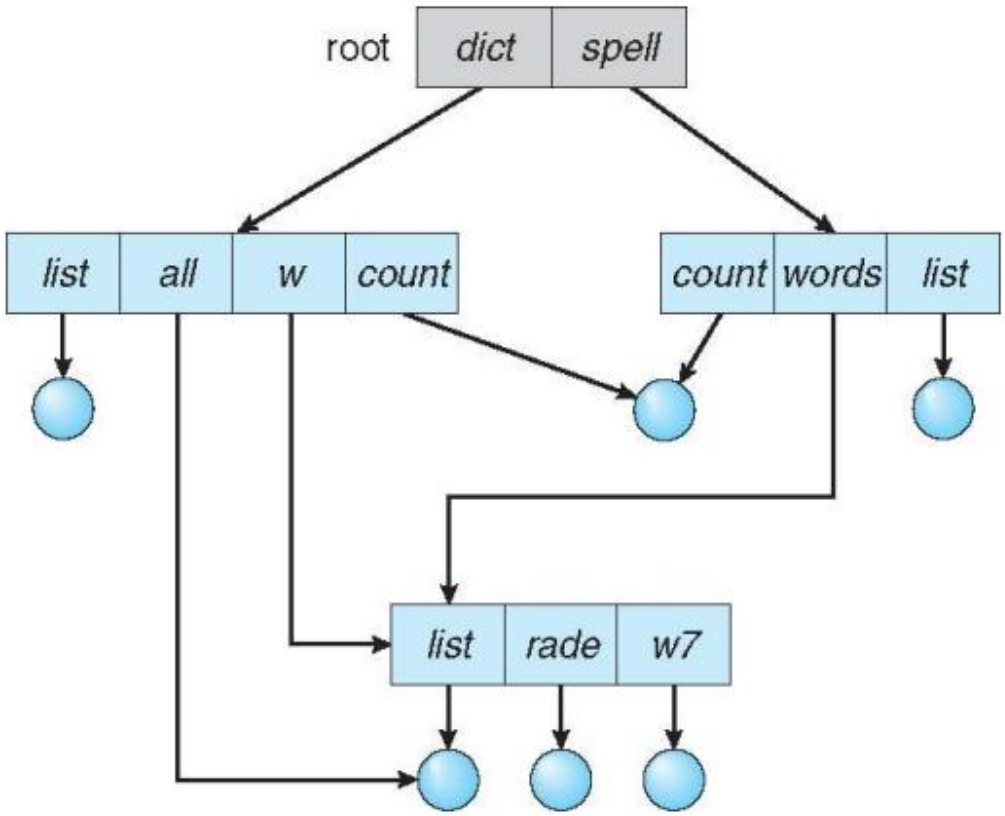
A hard link (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.

A symbolic link that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system. Windows only supports symbolic links, termed shortcuts.

Hard links require a reference count, or link count for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted: One option is to find all the symbolic links and adjust them also. Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.

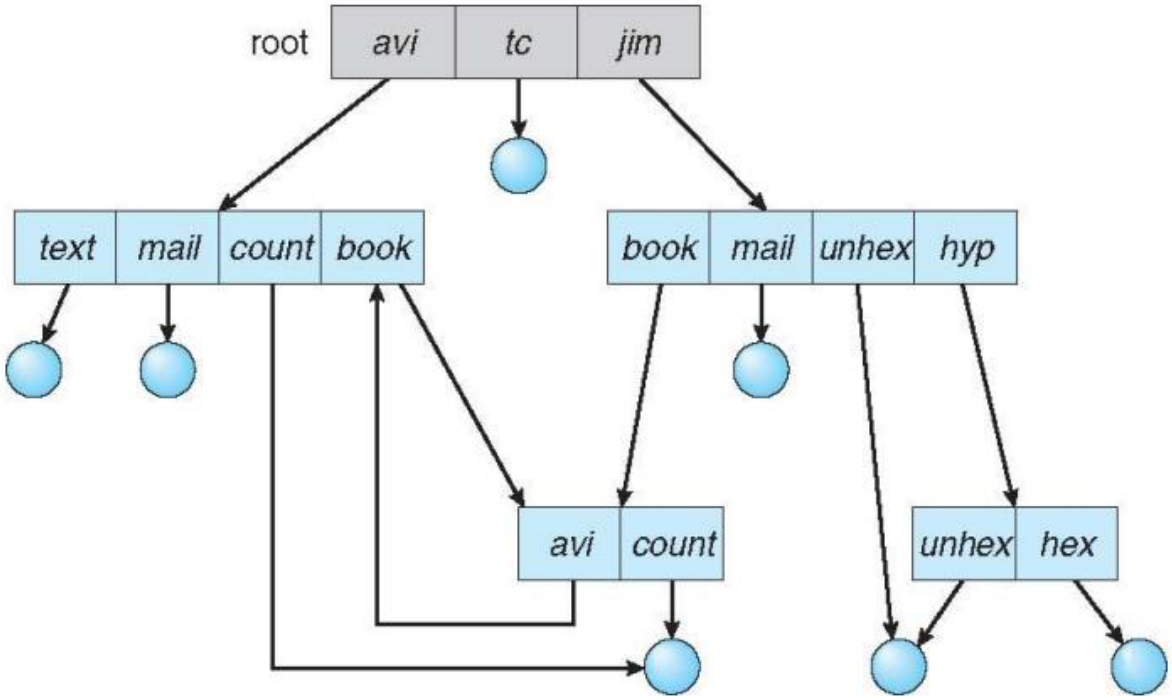
What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?



5.6.1 General Graph Directory

If cycles are allowed in the graphs, then several problems can arise: Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)

Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted.)



5.7 File System Mounting

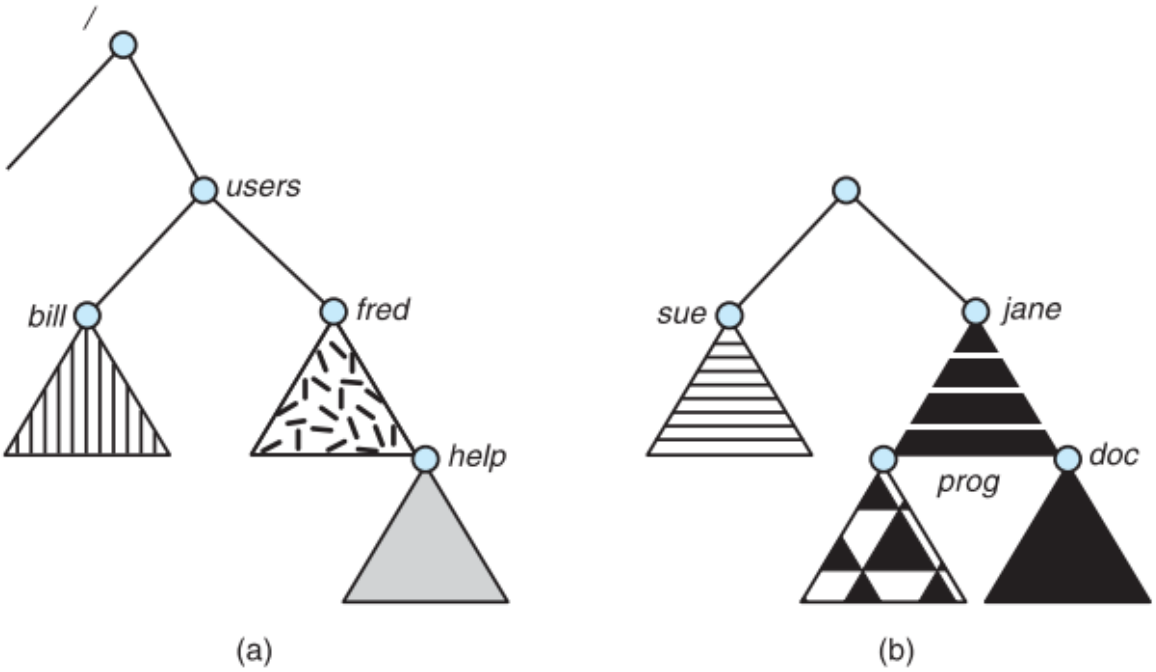
The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure. The mount command is given a file system to mount and a mount point (directory) on which to attach it.

Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

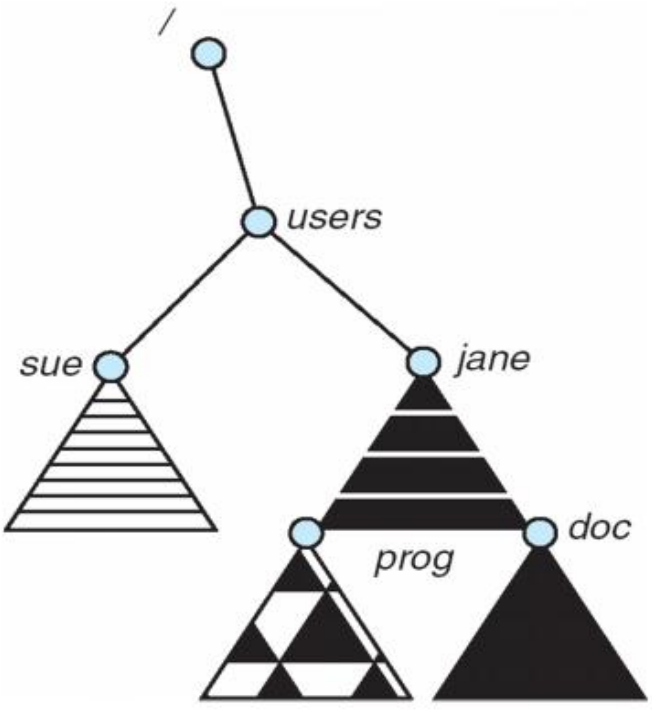
Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.

File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted.

File systems may be mounted read-only, or have other restrictions imposed.


/
users
bill
fred
help
sue
jane
prog
doc
(a)
(b)

Mount Point :


/
users
sue
jane
prog
doc

5.7.1 File Sharing

Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner (user) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
  - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or *anonymous*, not requiring any user name or password.
  - Various forms of *distributed file systems* allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
  - The WWW has made it easy once again to access files on remote systems without mounting their file systems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

- When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a ***server***, and the system which mounts them is the ***client.***
- User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)
- There are a number of security concerns involved in this model:
    - Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
    - Servers may restrict remote access to read-only.
    - Servers restrict which file systems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.
- The NFS (Network File System) is a classic example of such a system.

Distributed Information Systems

- The ***Domain Name System, DNS,*** provides for a unique naming system across all of the Internet.
- Domain names are maintained by the ***Network Information System, NIS***, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.

- Microsoft's ***Common Internet File System, CIFS***, establishes a ***network login*** for each user on a networked system with shared file access. Older Windows systems used ***domains***, and newer systems (XP, 2000), use ***active directories.*** User names must match across the network for this system to be valid.
- A newer approach is the ***Lightweight Directory-Access Protocol, LDAP,*** which provides a ***secure single sign-on*** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

Failure Modes

- When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

File Replication

Keeping multiple copies of a file on different servers has several important benefits:

• **Availability**. When a server holding the only copy of a file crashes, the file becomes inaccessible. With additional replicas, users may continue accessing the file unimpeded.

• **Reliability.** When a file server crashes, some files may be left in an inconsistent state. Having multiple independent copies of a file is an important tool for the recovery process.

• **Performance**. Maintaining multiple copies of files at different locations increases the chances for a client to access a file locally, or at least on a server that is in its close proximity. This decreases network latency and improves the time to access the file. It also reduces the overall network traffic.

• **Scalability**. A single server can easily be overwhelmed by a number of concurrent accesses to a file. Maintaining multiple copies of the file defuses the bottleneck and permits the system to easily scale up to a larger numbers of users.

File replication is similar to file-level caching in the client. In both cases, multiple copies of a file are maintained on different machines. The main difference is that caching only maintains a temporary copy in the client when the file is being used. With file replication, the copy is permanent. Furthermore, a cached copy is often kept on the client's machine, and a replicated file is kept on a separate server that may be a different machine from the client's.

Again, the problem with file replication is consistency. Similar to caching, the system must guarantee that all update operations are applied to all replicas at the same time or the file semantics must be relaxed. There are two main protocols the file system can follow when reading and writing file replicas to keep them consistent:

5.5 to 5.7 Check your progress

Q1. Fill in the blanks.

1. The machine containing the files is the _____ and the machine wanting to access the files is the _____
2. To recover from failures in the network operations, _____ information may be maintained.
3. In UNIX, exactly which operations can be executed by group members and other users is definable by_____.

Q2. True or false.

1. The series of accesses between the open and close operations is a file session.
2. The main problem with access control lists is their maintenance.
3. All users in a group get similar access to a file.

Summary

A file management system is a type of software that manages data files in a computer system. It has limited capabilities and is designed to manage individual or group files, such as special office documents and records. It may display report details, like owner, creation date, state of completion and similar features useful in an office environment. A file management system is also known as a file manager.

A file management system should not be confused with a file system, which manages all types of data and files in an operating system (OS), or a database management system (DBMS), which has relational database capabilities and includes a programming language for further data manipulation.

A file management system's tracking component is key to the creation and management of this system, where documents containing various stages of processing are shared and interchanged on an ongoing basis.

The system may contain features like:

- Assigning queued document numbers for processing
- Owner and process mapping to track various stages of processing
- Report generation
- Notes
- Status
- Create, modify, copy, delete and other file operations

5.1 & 5.2 Check your progress ---- Answers

Q1. Fill in the blanks.

1. File identifier
2. File metadata
3. Object file

Q2. True or false.

1. True
2. True
3. True

5.5 to 5.7 Check your progress ---- Answers

Q1. Fill in the blanks.

4. server, client
5. state
6. the file's owner

Q2. True or false.

4. True
5. False
6. True

Questions for self study

1. Describe basic functions of file systems.
2. Explain hierarchical model of file systems using diagram.
3. Describe operations of file systems.
4. Explain types of directory using diagram.
5. Explain file mounting mechanisms.

Suggested readings

5. **Operating System Concepts b**y Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
6. **Operating systems** By Stuart E. Madnick, John J. Donovan

References used:

16. Abraham-Silberschatz-Operating-System-Concepts---9th edition
17. Operating systems by William Stallings
18. Operating systems fundamental concepts
19. www.tutorialspoint.com
20. Operating systems by Achyut Godbole

Notes

Chapter 6:

I/O Systems

6.0 Objectives

- To describe various I/O devices
- To understand disk structure and disk attachment mechanisms
- To learn different disk scheduling algorithms.

6.1 Introduction

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

- **Block devices** − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices** − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

6.2 I/O Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

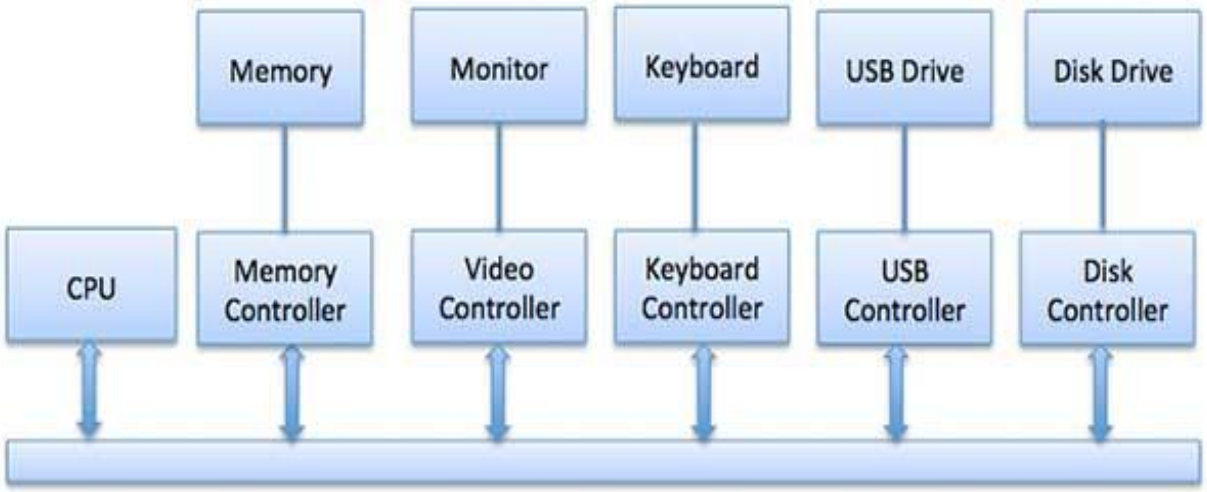


Figure: typical Device Controller

I/O Hardware:

Incredible variety of I/O devices

Storage (disk, tapes)

Transmission (network connections, Bluetooth)

Human-interface (screen, keyboard, mouse)

Common concepts – signals from I/O devices interface with computer

Port – connection point for device (e.g., serial port)

Bus – a set of wires and a rigidly defined protocol

- **daisy chain** or shared direct access
- **PCI** bus common in PCs and servers, PCI Express (**PCIe**) (>16GB/s)
- **expansion bus** connects relatively slow devices

Controller (**host adapter**) – electronics that operate port, bus, device

- Sometimes integrated
- Sometimes separate circuit board (host adapter)
- Contains processor, microcode, private memory, bus controller, etc.

- I/O instructions control devices

  Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution Data-in register, data-out register, status register, control register

Typically 1-4 bytes, or FIFO buffer

Devices have addresses, used by

Direct I/O instruction

- **Memory-mapped I/O**

  ▸ Device data and command registers mapped to processor address space

  ▸ Especially for large address spaces (graphics)

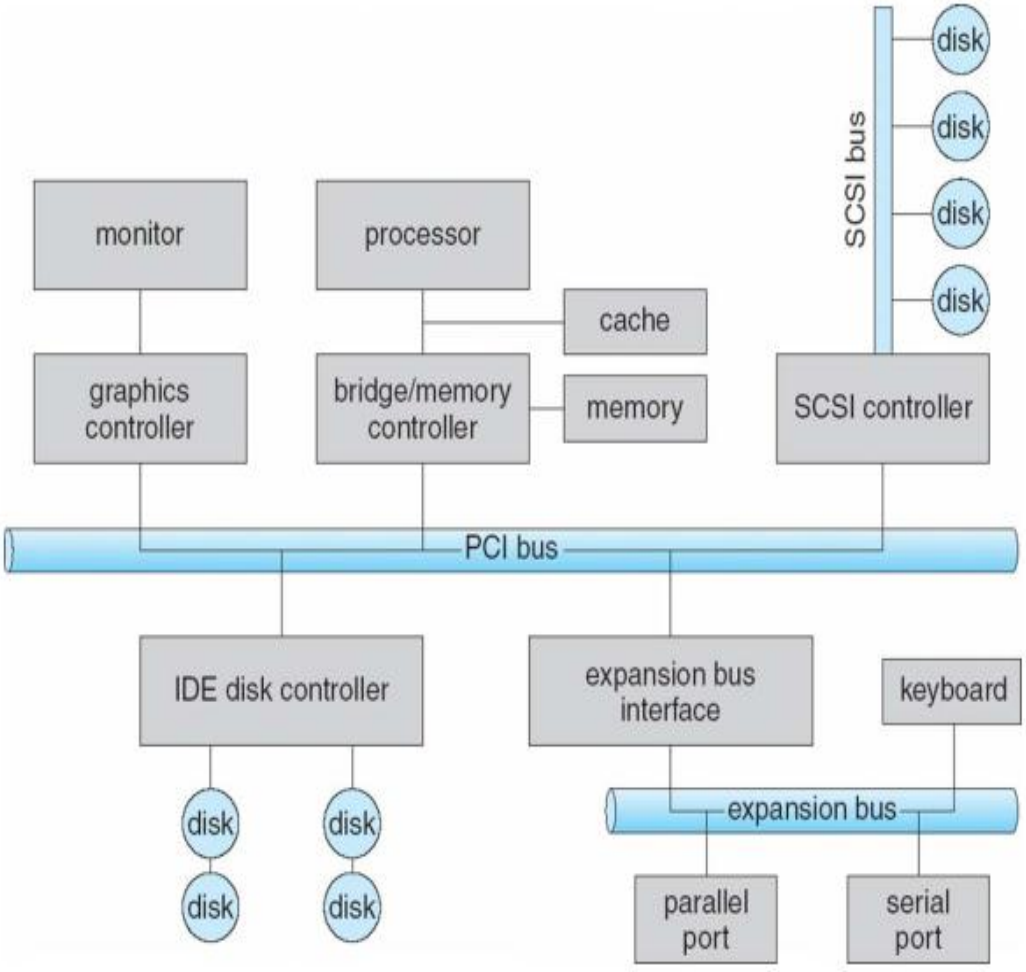


Figure: Typical PC Bus Structure

6.3 Disk structure

- One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.
- Each platter has two working ***surfaces.*** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
- Each working surface is divided into a number of concentric rings called ***tracks.*** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.
- Each track is further divided into ***sectors,*** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
- The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)
- The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A

particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

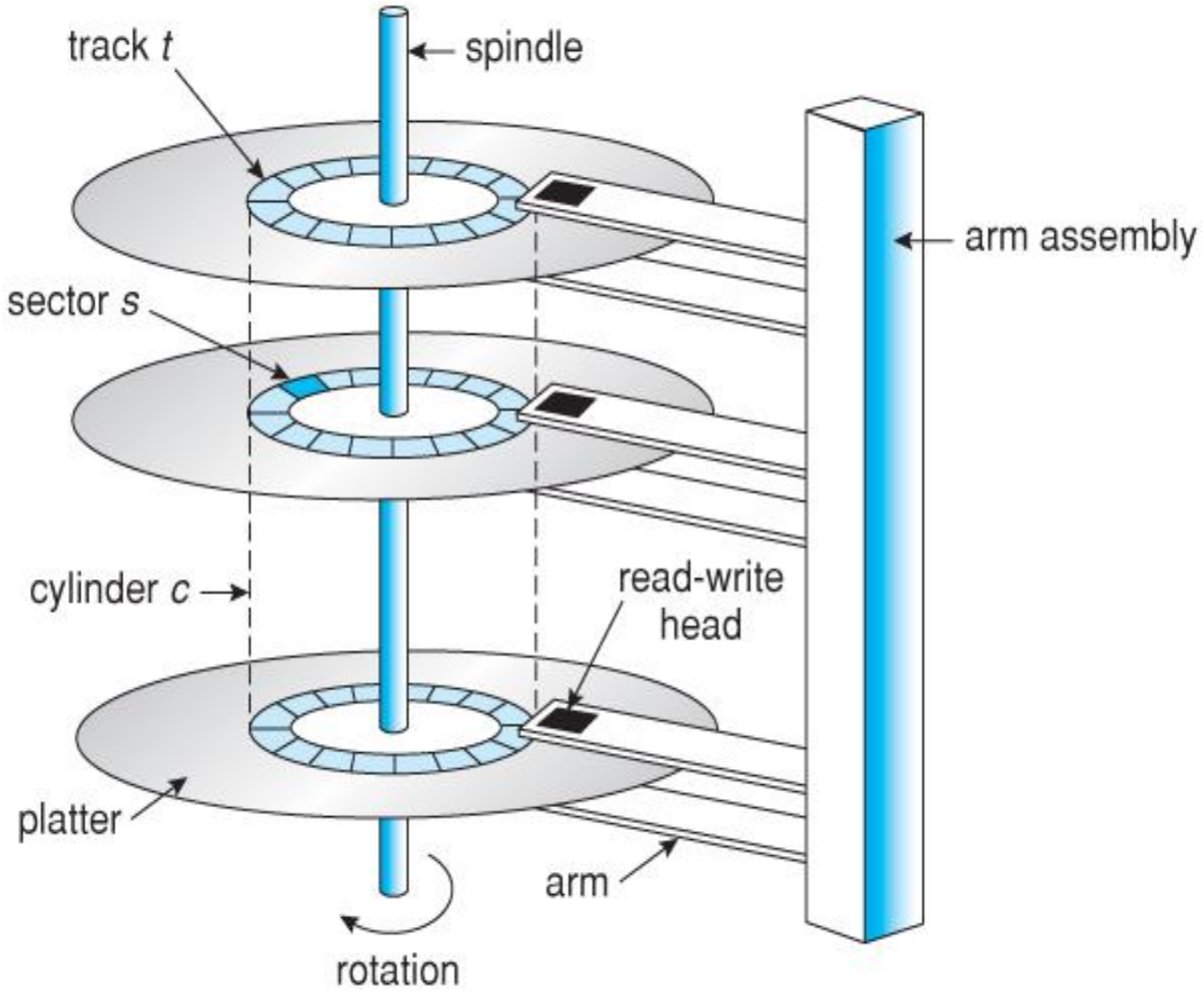
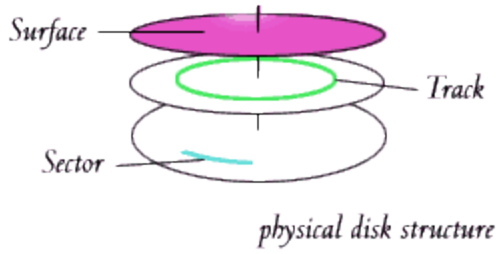


Figure - Moving-head disk mechanism.

- In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:
    - The ***positioning time***, a.k.a. the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.

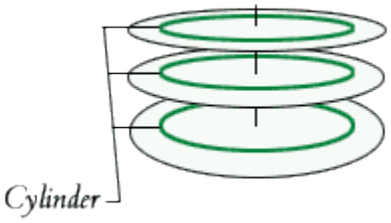
    - The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head. This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.
    - The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)
- Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which May or may not permanently damage the disk or even destroy it completely. For this reason it is normal to ***park*** the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.
- Floppy disks are normally ***removable***. Hard drives can also be removable, and some are even ***hot-swappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.
- Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.
- The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.



physical disk structure

The actual physical details of a modern hard disk may be quite complicated. Simply, there are one or more surfaces, each of which contains several tracks, each of which is divided into sectors. There is one read/write head for every surface of the disk.

Also, the same track on all surfaces is knows as a 'cylinder'. When talking about movement of the read/write head, the cylinder is a useful concept, because all the heads (one for each surface), move in and out of the disk together.



We say that the "read/write head is at cylinder #2", when we mean that the top read-write head is at track #2 of the top surface, the next head is at track #2 of the next surface, the third head is at track #2 of the third surface, etc.

The unit of information transfer is the sector (though often whole tracks may be read and written, depending on the hardware).

As far as most file systems are concerned, though, the sectors are what matter. In fact we usually talk about a 'block device'. A block often corresponds to a sector, though it need not do: several sectors may be aggregated to form a single logical block.

Disk Structure

- The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through

the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:

1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
    - With ***Constant Linear Velocity, CLV,*** the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
    - With ***Constant Angular Velocity, CAV,*** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (Thesedisks would have a constant number of sectors per track on all cylinders.)

6.2 & 6.3 Check your progress.

Fill in the blanks.

1. The Device Controller works like an interface between a device and a _____.
2. The _____ is at the computer end of the I/O bus, and the _____ is built into the disk itself.

6.4 Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

10.3.1 Host-Attached Storage

- Local disks are accessed through I/O Ports as described earlier.
- The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.
- SATA is similar with simpler cabling.
- High end workstations or other systems in need of larger number of disks typically use SCSI disks:
    - The SCSI standard supports up to 16 ***targets*** on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.
    - A SCSI target is usually a single drive, but the standard also supports up to 8 ***units*** within each target. These would generally be used for accessing individual disks within a RAID array.
    - The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.

    - Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.
    - SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.
- FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:
    - A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the ***storage-area networks, SANs,*** to be discussed in a future section.
    - The ***arbitrated loop, FC-AL,*** that can address up to 126 devices (drives and controllers.)

10.3.2 Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS file system mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.
- NAS can be implemented using SCSI cabling, or ***ISCSI*** uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.
- NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

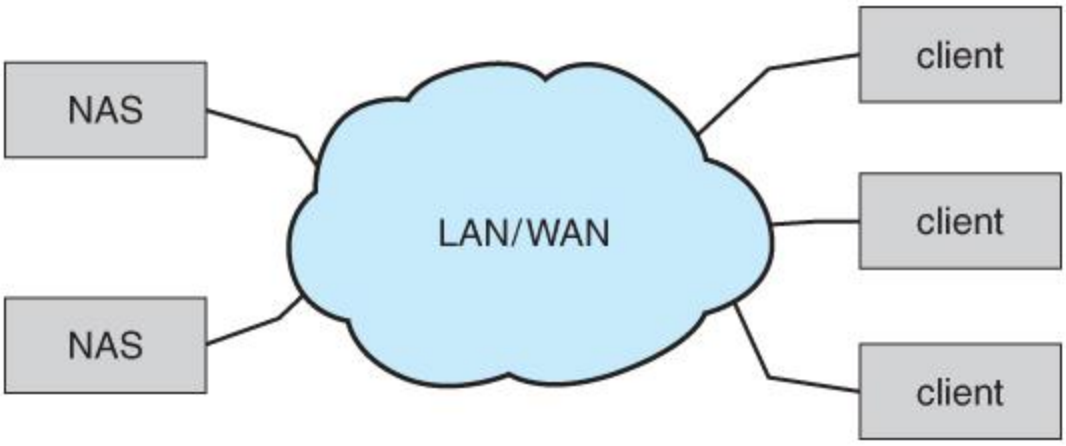


Figure - Network-attached storage.

10.3.3 Storage-Area Network

- A ***Storage-Area Network, SAN,*** connects computers and storage devices in a network, using storage protocols instead of network protocols.
- One advantage of this is that storage access does not tie up regular networking bandwidth.
- SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.
- SAN is also controllable, allowing restricted access to certain hosts and devices.

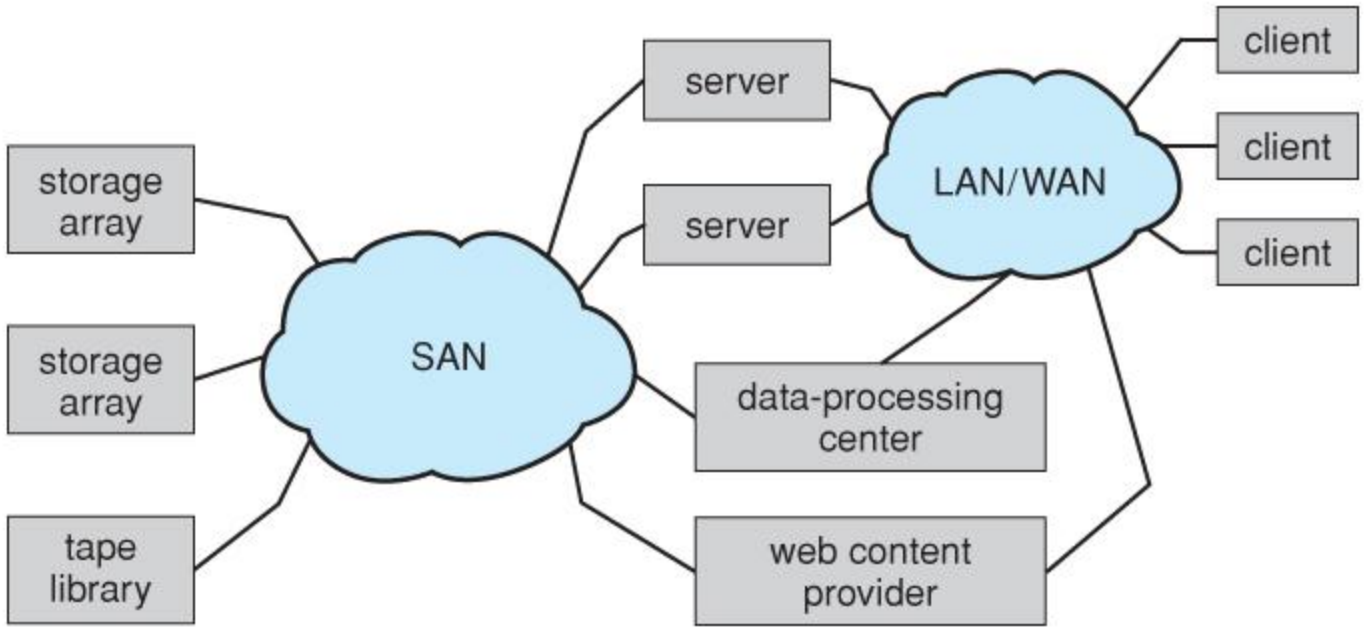


Figure - Storage-area network.

6.5 Disk Storage Capacity:

Storage capacity is a measurement of how much data you can store on a memory device. In computing, one unit of information is measured as one bit. Larger units can be converted as follows:

- 8 bits = 1 byte
- 1024 bytes = 1 kilobyte (KB)
- 1024 kilobytes = 1 megabyte (MB)
- 1024 megabytes = 1 gigabyte (GB)
- 1024 gigabytes = 1 terabyte (TB)

6.6 Disk Scheduling Algorithms

Disk scheduling is is done by operating systems to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling.

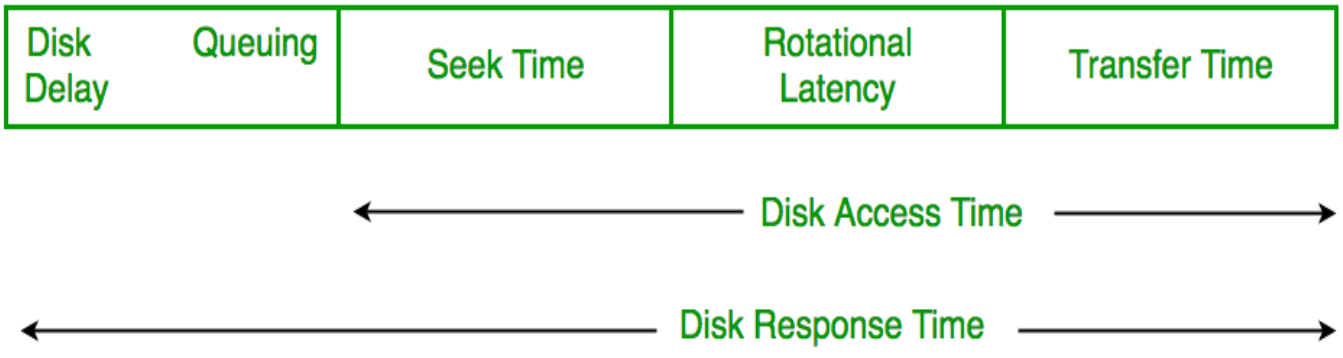
Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by disk controller. Thus other I/O requests need to wait in waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:** Disk Access Time is:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time



- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithm

1. **FCFS: (First come first serve)**

   FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

- Does not try to optimize seek time
- May not provide the best possible service

2. **SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

   Advantages:

   - Average Response Time decreases

- Throughput increases

Disadvantages:

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favors only some requests

3. **SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

   Advantages:

   - High throughput
   - Low variance of response time
   - Average response time

   Disadvantages:

   - Long waiting time for requests for locations just visited by disk arm

4. **CSCAN**: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

   These situations are avoided in *CSAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Advantages:

- Provides more uniform wait time compared to SCAN

6.4 & 6.5 Check your progress

Q1. True or false.

1. SAN stands for server area network.
2. FCFS is the simplest of all the Disk Scheduling Algorithms.

6.7 Summary

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Check your progress - Answers

6.2& 6.3 Q1. Fill in the blanks.

1. Device Driver
2. host controller, disk controller

6.4 & 6.5 Q1. True or false

1. False
2. True

6.8 Questions for self study.

1. Explain I/O devices.
2. Describe disk structure.
3. Explain SAN and NAS.
4. Describe all disk scheduling algorithms.

Suggested readings

7. **Operating System Concepts b**y Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
8. **Operating systems** By Stuart E. Madnick, John J. Donovan

References used:

21. Abraham-Silberschatz-Operating-System-Concepts---9th edition

22. Operating systems by William Stallings

23. Operating systems fundamental concepts

24. www.tutorialspoint.com

25. Operating systems by Achyut Godbole

Notes

Question bank for Operating systems

1. Define Operating Systems and discuss its role from different perspectives.
2. Explain fundamental difference between i) N/w OS and DOS
3. What do you mean by cooperating process? Describe its four advantages.
4. List out different services of Operating Systems and Explain.
5. Explain the concept of virtual machines. Bring out its advantages.
6. Distinguish among following terminologies
    a. Multiprogramming systems
    b. Multitasking Systems
    c. Multiprocessor systems.
7. What are system calls? Explain different categories of system calls with example?
8. Why is the Operating System viewed as a resource allocator & control program?
9. What is the Kernel?
10. What is an Interactive Computer System?
11. . What are the various OS Components?
12. What do you mean by PCB? Where is it used? What are its contents? Explain.
13. Explain direct and indirect communications of message passing systems.
14. Explain the difference between long term and short term and medium term schedulers.
15. What is a process? Draw and explain process state diagram.
16. Define IPC. What are different methods used for logical implementations of message passing
17. Discuss common ways of establishing relationship between user and kernel thread.
18. What is a Process?
19. What is a Process State and mention the various States of a Process?
20. What is Process Control Block (PCB)?
21. What are System Calls?
22. Explain multithreading models.
23. What are the design goals of an Operating System?

24. What are the five major categories of System Calls?
25. What are semaphores? Explain two primitive semaphore operations. What are its advantages?
26. Explain any one synchronization problem for testing newly proposed sync scheme.
27. Explain three requirements that a solution to critical–section problem must satisfy.
28. Define Race Condition.
29. What is Waiting Time in CPU scheduling?
30. What is Response Time in CPU scheduling?
31. Differentiate Long Term Scheduler and Short Term Scheduler
32. State dining philosopher's problem and give a solution using semaphores. Write structure of philosopher.
33. Describe term monitor. Explain solution to dining philosopher's problem using monitor.
34. What is synchronization? Explain its hardware.
35. What are semaphores? Explain solution to producer-consumer problem using semaphores
36. What is paging and swapping?
37. What are the drawbacks of Monitors?
38. What are the advantages of Contiguous Allocation?
39. With a diagram discuss the steps involved in handling a page fault.
40. What is address binding? Explain the concept of dynamic relocation of addresses.
41. What is paging? Explain the paging hardware?
42. Explain the following i) file types ii) file operation iii) file attributes.
43. Explain the method used for implementing directories.
44. Describe various file access methods.
45. Explain file system mounting operation.
46. Mention the different file attributes and file types
47. What is a File?

48. List the various File Attributes..
49. What are the various File Operations?
50. How free space is managed? Explain.
51. What are the three methods for allocating disk space? Explain.
52. Explain various disk scheduling algorithms?
53. Explain the File System Structure in detail
54. Discuss the File System Organization and File System Mounting.
55. Discuss briefly about Memory Management in UNIX.