KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT : OPERATING SYSTEMS
SEMESTER : III
SUBJECT CODE : 16CSU302 **CLASS : II B.Sc.CS**

COURSE OBJECTIVE:

This course recognize the concepts and principles of operating systems, provide students with the basic knowledge and skills of memory, device and Process management and techniques and provide experience on MS Windows and LINUX environment.

COURSE OUTCOME:

A student who successfully completes this course should, at a minimum, be able to:

- Explain basic Idea about the operating system.
- Concept and techniques involved in memory, device and Process management.
- Work in MS Windows and LINUX environment.

UNIT-I

Introduction to Operating System: Basic OS Functions-Resource Abstraction-Types of Operating Systems–Multiprogramming Systems-Batch Systems-Time Sharing Systems-Operating Systems for Personal Computers & Workstations-Process Control & Real Time Systems.

UNIT-II

Operating System Organization: Processor and user modes-Kernels-System Calls and System Programs. **Process Management:** System view of the process and resources-Process abstraction-Process hierarchy-Threads-Threading issues-Thread libraries-Process Scheduling-Non pre-emptive and Preemptive scheduling algorithms-Concurrent and processes-Critical Section-Semaphores-Methods for inter-process communication-Deadlocks.

UNIT-III
Memory Management: Physical and Virtual address space-Memory Allocation strategies –Fixed and Variable partitions-Paging-Segmentation-Virtual memory.

UNIT-IV
File and I/O Management: Directory structure-File operations-File Allocation methods-Device management.

UNIT-V
Protection and Security: Policy mechanism-Authentication-Internal access Authorization.

TEXTBOOKS :

1. Silberschatz, A ., Galvin, P.B. , & Gagne, G. (2008). Operating Systems Concepts, 8th ed.). New Delhi: John Wiley Publications.
2. Tanenbaum, A.S. (2007).Modern Operating Systems (3rd ed.). New Delhi: Pearson Education.
3. Stallings, W. (2008). Operating Systems, Internals & Design Principles (5th ed.). New Delhi: Prentice Hall of India.

WEB SITES

1. www.cs.columbia.edu/~nieh/teaching/e6118_s00/
2. www.clarkson.edu/~jnm/cs644
3. pages.cs.wisc.edu/~remzi/Classes/736/Fall2002/

ESE MARKS ALLOCATION

1.	**Section A** 20 x 1 = 20	20
2.	**Section B** 5 x 2 = 10	10
3.	**Section C** 5 x 6 = 30 Either ‘A’ or ‘B’ choice	30
	Total	60

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore - 641021.

(For the candidates admitted from 2015 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

LECTURE PLAN

STAFF NAME: D.MANJULA, N. MANONMANI
SUBJECT NAME: OPERATING SYSTEMS **SUB.CODE: 16CSU302**
SEMESTER: III **CLASS: II B.Sc (CS)**

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT – I			
1.	1	Basic OS Functions	W1
2.	1	Resource Abstraction	T1: 4-6, W1
3.	1	Types of Operating Systems	T2: 32-35, W1
4.	1	Multiprogramming Systems	T1: 19.T2:32, W1
5.	1	Batch Systems	T1: 19, W1
6.	1	Time Sharing Systems	T1: 20, W1
7.	1	Operating Systems for Personal Computers & Workstations	W1, W2
8.	1	Process Control	T2: 34, W1
9.	1	Real Time Systems	W1
10.	1	Recapitulation and discussion of important questions	
	Total no. of Hours planned for Unit – I		10 hrs

T1: A .Silberschatz, , P.B Galvin, G.Gagne (2008). Operating Systems Concepts, 8th ed.). John Wiley Publications.
T2: A.S. Tanenbaum, (2007). Modern Operating Systems (3rd ed.). New Delhi: Pearson Education.
W1. https://www.tutorialspoint.com/operating_system/os_overview.htm
W2: https://en.wikipedia.org/wiki/Workstation

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT II			
1.	1	**Operating System Organization:** Processor and user modes	T1: 20-23;55-58,
2.	1	Kernels	T1: 66-68
3.	1	System Calls	T2: 47-50

4.	1	System Programs	T2: 50-59
5.	1	**Process Management:** System view of the process and resources	T1: 101-110;
6.	1	Process abstraction , Process hierarchy	T2: 81-93
7.	1	Threads	T1: 104 – 105
8.	1	Threading issues- Thread libraries	T2: 93-115;
9.	1	Process Scheduling-Non pre-emptive and Preemptive scheduling algorithms	T1: 105 – 110
10.	1	Concurrent and processes	T2: 143 - 161
11.	1	Critical Section	T1: 227 – 229 T2: 117,866
12.	1	Semaphores	T1: 234 – 239 T2: 126 - 128
13.	1	Methods for inter-process communication	T1: 897 – 898 T2: 115 - 126
14.	1	Deadlocks	T1: 283 – 305
15.		Deadlock Avoidance and Prevention	T2: 435 – 459
16.	1	Recapitulation and discussion of important questions	
	Total no. of Hours planned for Unit-II		16 hrs
T1: A .Silberschatz, , P.B Galvin, G.Gagne (2008). Operating Systems Concepts, 8th ed.). John Wiley Publications. **T2:** A.S. Tanenbaum, (2007). Modern Operating Systems (3rd ed.). New Delhi: Pearson Education.			
UNIT - III			
1.	1	**Memory Management:** Introduction	T1: 315 - 320
2.	1	Physical address space	T2: 173 - 185
3.	1	Virtual address space	T1: 359 – 360 T2: 173 - 185
4.	1	Memory Allocation Strategies (First Fit, Best Fit)	W3
5.	1	Memory Allocation Strategies (Worst Fit)	W3
6.	1	Fixed partitions	T1: 325; W3
7.	1	Variable partitions	T1: 326 ; W3
8.	1	Paging	T2: 186 – 195, 214,225 T1:328 - 342
9.	1	Segmentation	T1: 342 - 345
10.	1	Virtual memory	T1: 357 – 393 T2: 186 - 196

11.	1	Recapitulation and discussion of important questions	
	Total no. of Hours planned for Unit – III		11 hrs

T1: A .Silberschatz, , P.B Galvin, G.Gagne (2008). Operating Systems Concepts, 8th ed.). John Wiley Publications.
T2: A.S. Tanenbaum, (2007). Modern Operating Systems (3rd ed.). New Delhi: Pearson Education.
W3: u.cs.biu.ac.il/~ariel/download/os381/ppts/os7-2_rea.ppt

UNIT - IV

1.	1	File and I/O Management	T1: 64-67, 421; W4, W5
2.	1	Directory structure	T2: 266 - 270
3.	1	File operations	T1: 423-425
4.	1	File operations - Example	T2: 262 - 263
5.	1	File Allocation methods – Contiguous	T1: 471 – 476
6.	1	File Allocation methods –Linked	T1: 471 – 476
7.	1	File allocation methods - Indexed	T1: 476 - 479
8.	1	Device management – Device type, storage, access	T1: 64 W6
9.	1	Device management - Device drivers, detection, IPC, Driver interface	W7
10.	1	Recapitulation and discussion of important questions	
	Total no. of Hours planned for Unit – IV		10 hrs

T1: A .Silberschatz, , P.B Galvin, G.Gagne (2008). Operating Systems Concepts, 8th ed.). John Wiley Publications.
T2: A.S. Tanenbaum, (2007). Modern Operating Systems (3rd ed.). New Delhi: Pearson Education.
W4: https://www.slideshare.net/DamianGordon1/operating-systems-file-management
W5: http://nptel.ac.in/courses/106108101/pdf/Lecture_Notes/Mod%205_LN.pdf
W6: https://www.slideshare.net/honeyturqueza/ch-7-device-management
W7: http://wiki.osdev.org/Device_Management

UNIT - V

1.	1	Protection and Security-Introduction	T1:629
2.	1	Policy Mechanism	T2:620-623
3.	1	Authentication	T2:639
4.	1	(i)Password authentication	T2:640
5.	1	(ii)biometric authentication	T2:651-654
6.	1	(iii)Encrypted and one time passwords	T1:661-662
7.	1	Program threats	T1:663-664
8.	1	Internal Access authorization	W1
9.	1	Access Control	W1

10.	1	Recapitulation and discussion of important questions	
11.	1	Previous ESE Question Paper Discussion	
12.	1	Previous ESE Question Paper Discussion	
13.	1	Previous ESE Question Paper Discussion	
	Total no. of Hours planned for Unit – V		13 hrs

T1: A .Silberschatz, , P.B Galvin, G.Gagne (2008). Operating Systems Concepts, 8th ed.). John Wiley Publications.
T2: A.S. Tanenbaum, (2007). Modern Operating Systems (3rd ed.). New Delhi: Pearson Education.
W1: https://www.tutorialspoint.com/operating_system/os_overview.htm

UNIT I

Syllabus

Introduction to Operating System: Basic OS Functions-Resource Abstraction-Types of Operating Systems–Multiprogramming Systems-Batch Systems-Time Sharing Systems-Operating Systems for Personal Computers & Workstations-Process Control & Real Time Systems.

Introduction to Operating System

An operating system is a program that manages the computer hardware. An Operating System (OS) is an interface between computer user and computer hardware. An operating system is 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.

Some popular Operating Systems include Linux, Windows, Macintosh, etc.

Definition

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

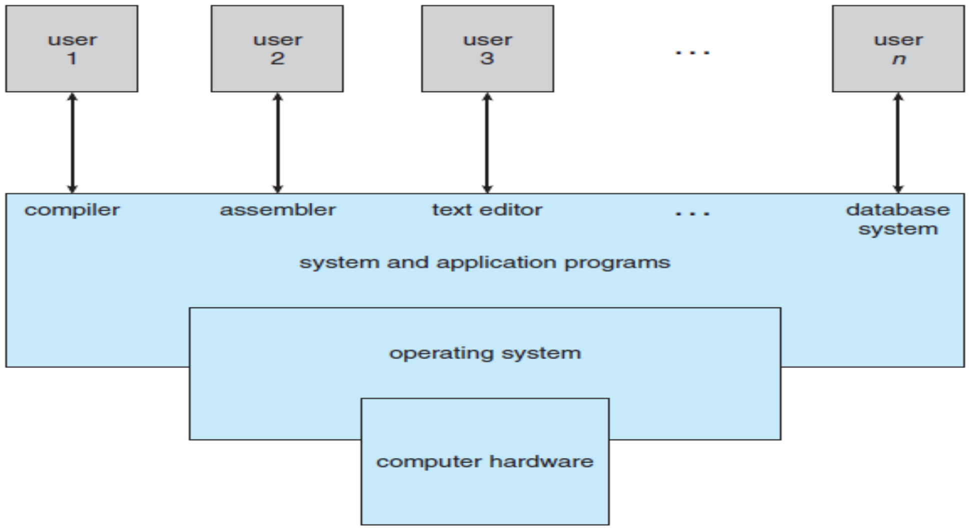


Figure 1.1 Abstract views of the components of a computer system

A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and the ***users*** (Figure 1.1).

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 the hardware and coordinates its use 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.

Basic OS Functions

Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Following are some of the important activities that an Operating System performs −

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.

- **Coordination between other software's and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Resource Abstraction

The concept of an operating system as primarily providing abstractions to application programs is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the alternative view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and input/output devices among the various programs competing for them.

When a computer (or network) has multiple users, the need for managing and protecting the memory, input/output devices, and other resources is even greater, since the users might otherwise interface with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of which programs are using which resources, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes **multiplexing** (sharing) resources in two different ways:

1. Time Multiplexing
2. Space Multiplexing

1. Time Multiplexing

When the resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on.

For example:

With only one CPU and multiple programs that want to run on it, operating system first allocates the CPU to one long enough, another one gets to use the CPU, then another and ten eventually the first one again.

Determining how the resource is time multiplexed – who goes next and for how long – is the task of the operating system.

2. Space Multiplexing

In space multiplexing, instead of the customers taking turns, each one gets part of the resource.

For example:

Main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it, especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them.

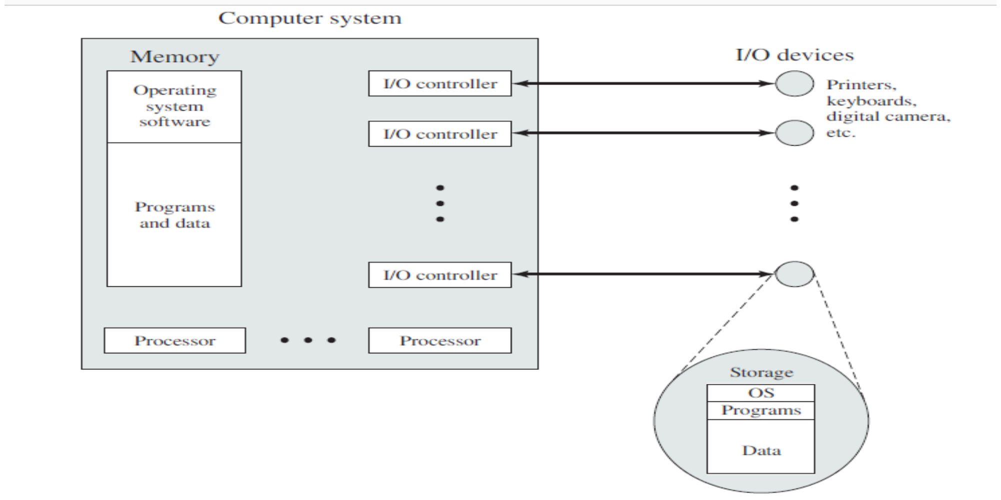


Figure 1.2 The Operating System as Resource Manager

Figure 1.2 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel**, or **nucleus**, which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The memory management hardware in the processor and the OS jointly control the allocation of main memory, as we shall see. The OS decides when an I/O device can be used by a program in execution and controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

Types of Operating Systems

Mainframe Operating Systems

At the high end are the operating systems for mainframes, those room-sized computers still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and millions of gigabytes of data is not unusual; a personal computer with these specifications would be the envy of its friends.

Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions. The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O.

They typically offer three kinds of services: batch, transaction processing, and timesharing.

Batch systems

A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode.

Transaction-processing systems

Transaction-processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second.

Timesharing systems

Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related; mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360. However, mainframe operating systems are gradually being replaced by UNIX variants such as Linux.

Server Operating Systems

One level down are the server operating systems. They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Websites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are Solaris, FreeBSD, Linux and Windows Server 201x.

Multiprocessor Operating Systems

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication, connectivity, and consistency.

With the recent advent of multicore chips for personal computers, even conventional desktop and notebook operating systems are starting to deal with at least small-scale multiprocessors and the number of cores is likely to grow over time. Luckily, quite a bit is known about multiprocessor operating systems from years of previous research, so using this knowledge in multicore systems should not be hard. The hard part will be having applications make use of all this computing power. Many popular operating systems, including Windows and Linux, run on multiprocessors.

Personal Computer Operating Systems

The next category is the personal computer operating system. Modern ones all support multiprogramming, often with dozens of programs started up at boot time. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets, games, and Internet access. Common examples are Linux, FreeBSD, Windows 7, Windows 8, and Apple’s OS X. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

Handheld Computer Operating Systems

Continuing on down to smaller and smaller systems, we come to tablets, smartphones and other handheld computers. A handheld computer, originally known as a **PDA** (**Personal Digital Assistant**), is a small computer that can be held in your hand during operation. Smartphones and tablets are the best-known examples. As we have already seen, this market is currently dominated by Google’s Android and Apple’s iOS, but they hav e many competitors. Most of these devices boast multicore CPUs, GPS, cameras and other sensors, copious amounts of memory, and sophisticated operating systems. Moreover, all of them have more third-party applications (**‘‘apps’’**) than you can shake a (USB) stick at.

Embedded Operating Systems

Embedded systems run on the computers that control devices that are not generally thought of as computers and which do not accept user-installed software. Typical examples are microwave ovens, TV sets, cars, DVD recorders, traditional phones, and MP3 players. The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it. You cannot download new applications to your microwave oven—all the software is in ROM. This means that there is no need for protection between applications, leading to design simplification. Systems such as Embedded Linux, QNX and VxWorks are popular in this domain.

Sensor-Node Operating Systems

Networks of tiny sensor nodes are being deployed for numerous purposes. These nodes are tiny computers that communicate with each other and with a base station using wireless communication. Sensor networks are used to protect the perimeters of buildings, guard national borders, detect fires in forests, measure temperature and precipitation for weather forecasting, glean information about enemy movements on battlefields, and much more.

The sensors are small battery-powered computers with built-in radios. They have limited power and must work for long periods of time unattended outdoors, frequently in environmentally harsh conditions. The network must be robust enough to tolerate failures of individual nodes, which happen with ever-increasing frequency as the batteries begin to run down.

Each sensor node is a real computer, with a CPU, RAM, ROM, and one or more environmental sensors. It runs a small, but real operating system, usually one that is event driven, responding to external events or making measurements periodically based on an internal clock. The operating system has to be small and simple because the nodes have little RAM and battery lifetime is a major issue. Also, as with embedded systems, all the programs are loaded in advance; users do not suddenly start programs they downloaded from the Internet, which makes the design much simpler. TinyOS is a well-known operating system for a sensor node.

Real-Time Operating Systems

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process- control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time. If, for example, a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system**. Many of these are found in industrial process control, avionics, military, and similar application areas. These systems must provide absolute guarantees that a certain action will occur by a certain time.

A **soft real-time system**, is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Digital audio or multimedia systems fall in this category. Smartphones are also soft realtime systems.

Since meeting deadlines is crucial in (hard) real-time systems, sometimes the operating system is simply a library linked in with the application programs, with ev erything tightly coupled and no protection between parts of the system.

An example of this type of real-time system is eCos. The categories of handhelds, embedded systems, and real-time systems overlap considerably. Nearly all of them have at least some soft real-time aspects. The embedded and real-time systems run only software put in by the system designers; users cannot add their own software, which makes protection easier. The handhelds and embedded systems are intended for consumers, whereas real-time systems are more for industrial usage. Nevertheless, they hav e a certain amount in common.

Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU chip. They hav e very severe processing power and memory constraints. Some are powered by contacts in the reader into which they are inserted, but contactless smart cards are inductively powered, which greatly limits what they can do. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions. Often these are proprietary systems.

Some smart cards are Java oriented. This means that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

Multiprogramming Systems

To overcome the problem of underutilization of CPU and main memory, the multiprogramming was introduced. The multiprogramming is interleaved execution of multiple jobs by the same computer.

In multiprogramming system, when one program is waiting for I/O transfer; there is another program ready to utilize the CPU. So it is possible for several jobs to share the time of the CPU. But it is important to note that multiprogramming is not defined to be the execution of jobs at the same instance of time. Rather it does mean that there are a number of jobs available to the CPU (placed in main memory) and a portion of one is executed then a segment of another and so on.

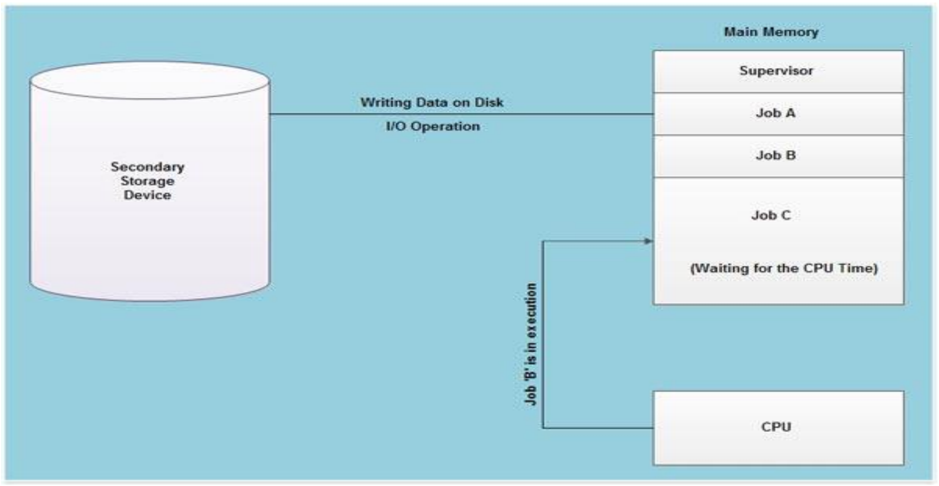


Figure 1.3. A simple process of multiprogramming.

as shown in fig, at the particular situation, job' A' is not utilizing the CPU time because it is busy in I/ 0 operations. Hence the CPU becomes busy to execute the job 'B'. Another job C is waiting for the CPU for getting its execution time. So in this state the CPU will never be idle and utilizes maximum of its time.

A program in execution is called a "Process", "Job" or a "Task". The concurrent execution of programs improves the utilization of system resources and enhances the system throughput as compared to batch and serial processing. In this system, when a process requests some I/O to allocate; meanwhile the CPU time is assigned to another ready process. So, here when a process is switched to an I/O operation, the CPU is not set idle.

Multiprogramming is a common approach to resource management. The essential components of a single-user operating system include a command processor, an input/ output control system, a file system, and a transient area. A multiprogramming operating system builds on this base, subdividing the transient area to hold several independent programs and adding resource management routines to the operating system's basic functions.

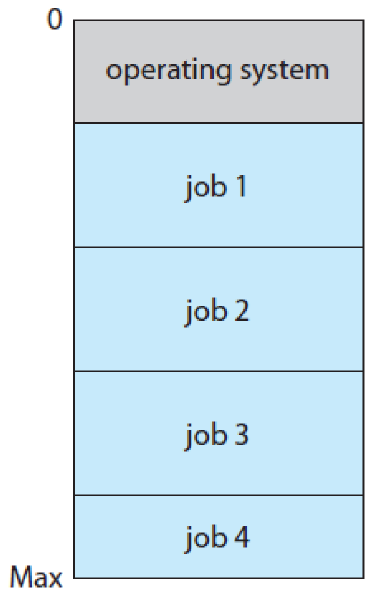


Figure 1.4 Memory layout for a multiprogramming system.

Multiprogramming increases CPUutilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job.

When ***that*** job needs to wait, the CPU switches to ***another*** job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle. This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

Time sharing (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device.

Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user. Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**.

Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text. In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk. A more common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory.

This arrangement frees programmers from concern over memory-storage limitations.

A time-sharing system must also provide a file system. The file system resides on a collection of disks; hence, disk management must be provided. In addition, a time-sharing system provides a mechanism for protecting resources from inappropriate use.

To ensure orderly execution, the system must provide mechanisms for job synchronization and communication, and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another.

Batch Processing Systems

To avoid the problems of early systems the batch processing systems were introduced. The problem of early systems was more setup time. So the problem of more set up time was reduced by processing the jobs in batches, known as *batch processing system.*In this approach similar jobs were submitted to the CPU for processing and were run together.

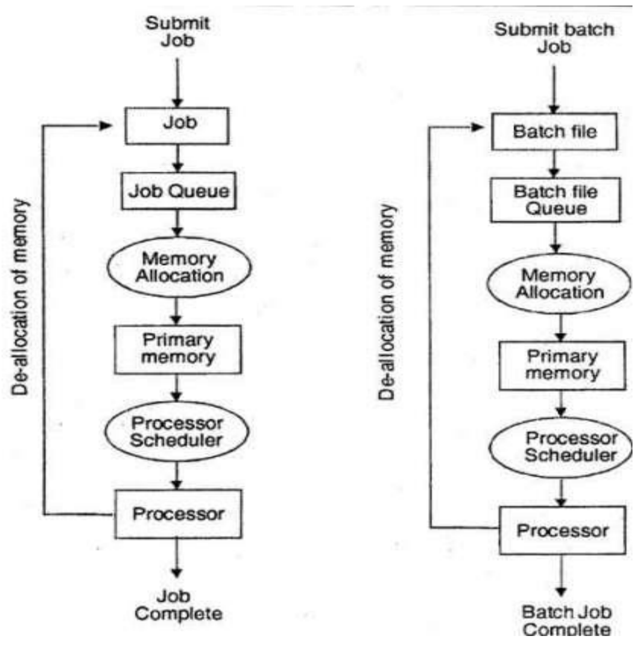
The main function of a batch processing system is to automatically keep executing the jobs in a batch. This is the important task of a batch processing system i.e. performed by the 'Batch Monitor' resided in the low end of main memory.

This technique was possible due to the invention of hard-disk drives and card readers. Now the jobs could be stored on the disk to create the pool of jobs for its execution as a batch. First the pooled jobs are read and executed by the batch monitor, and then these jobs are grouped; placing the identical jobs (jobs with the similar needs) in the same batch, So, in the batch processing system, the batched jobs were executed automatically one after another saving its time by performing the activities (like loading of compiler) only for once. It resulted in improved system utilization due to reduced turn around time.

In the early job processing systems, the jobs were placed in a job queue and the memory allocate or managed the primary memory space, when space was available in the main memory, a job was selected from the job queue and was loaded into memory.

Once the job loaded into primary memory, it competes for the processor. When the processor became available, the processor scheduler selects job that was loaded in the memory and execute it.

In batch strategy is implemented to provide a batch file processing. So in this approach files of the similar batch are processed to speed up the task.



Traditional Job Processing **Batch File Processing**

In batch processing the user were supposed to prepare a program as a deck of punched cards. The header cards in the deck were the "job control" cards which would indicate that which compiler was to be used (like FORTRAN, COBOL compilers etc). The deck of cards would be handed in to an operator who would collect such jobs from various users. Then the submitted jobs were 'grouped as FORTRAN jobs, COBOL jobs etc.

In addition, these jobs were classified as 'long jobs' that required more processing time or short jobs which required a short processing time. Each set of jobs was considered as a batch and the processing would be done for a batch. For instance, there maybe a batch of short FORTRAN jobs. The output for each job would be separated and turned over to users in a collection area. So in this approach, files of the similar batch were processed to speed up the task.

In this environment there was no interactivity and the users had no direct control. In this system, only one job could engage the processor at a time and if there was any input/ output operation the processor had to sit idle till the completion of I/O job. So it resulted to the underutilization of CPU time.

In batch processing system, earlier; the jobs were scheduled in the order of their arrival i.e. First Come First Served (FCFS).Even though this scheduling method was easy and simple to implement but unfair for the situations where long jobs are queued ahead of the short jobs. To overcome this problem, another scheduling method named as 'Shortest Job First' was used. As memory management is concerned, the main memory was partitioned into two fixed partitions. The lower end of this partition was assigned to the resident portion of the OS i.e. named as Batch Monitor. Whereas, the other partition (higher end) was assigned to the user programs.

Though, it was an improved technique in reducing the system setup time but still there were some limitations with this technique like as under-utilization of CPU time, non-interactivity of user with the running jobs etc. In batch processing system, the jobs of a batch were executed one after another. But while these jobs were performing I/O operations; meantime the CPU was sitting idle resulting to low degree of resource utilization.

Time Sharing System

A **time sharing system** allows many users to share the computer resources simultaneously. In other words, time sharing refers to the allocation of computer resources in time slots to several programs simultaneously. For example a mainframe computer that has many users logged on to it. Each user uses the resources of the mainframe -i.e. memory, CPU etc. The users feel that they are exclusive user of the CPU, even though this is not possible with one CPU i.e. shared among different users.

The time sharing systems were developed to provide an interactive use of the computer system. A time shared system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. It allows many users to share the computer resources simultaneously. As the system switches rapidly from one user to the other, a short time slot is given to each user for their executions.

The time sharing system provides the direct access to a large number of users where CPU time is divided among all the users on scheduled basis. The OS allocates a set of time to each user. When this time is expired, it passes control to the next user on the system. The time allowed is extremely small and the users are given the impression that they each have their own CPU and they are the sole owner of the CPU. This short period of time during that a user gets attention of the CPU; is known as a *time slice or a quantum.* The concept of time sharing system is shown in figure.

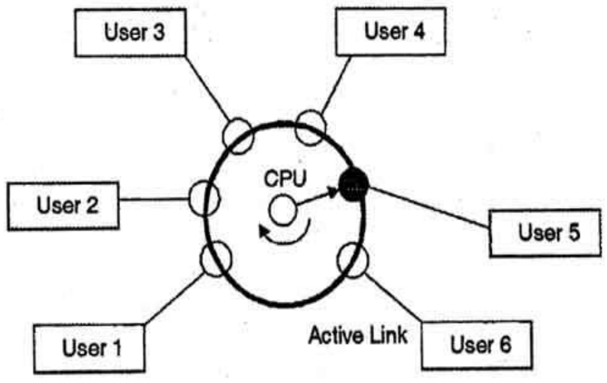


Figure 1.5 Time Sharing System Active State of User

In above figure the user 5 is active but user 1, user 2, user 3, and user 4 are in waiting state whereas user 6 is in ready status.

As soon as the time slice of user 5 is completed, the control moves on to the next ready user i.e. user 6. In this state user 2, user 3, user 4, and user 5 are in waiting state and user 1 is in ready state. The process continues in the same way and so on.

The time-shared systems are more complex than the multi-programming systems. In time-shared systems multiple processes are managed simultaneously which requires an adequate management of main memory so that the processes can be swapped in or swapped out within a short time.

Note: The term 'Time Sharing' is no longer commonly used, it has been replaced by 'Multitasking System'.

Time-sharing operating systems

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if **n** users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows −

- Provides the advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows −

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

Operating Systems for Personal Computers

The next category is the personal computer operating system. Modern ones all support multiprogramming, often with dozens of programs started up at boot time. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Linux, FreeBSD, Windows Vista, and the Macintosh operating system. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

A local computer manufacturer, Seattle Computer Products, developed an operating system, **DOS (Disk Operating System).** The revised system was renamed **MS-DOS (MicroSoft Disk Operating System)** and quickly came to dominate the IBM PC market.

Although the initial version of MS-DOS was fairly primitive, subsequent versions included more advanced features, including many taken from UNIX. (Microsoft was well aware of UNIX, even selling a microcomputer version of it called XENIX during the company's early years.) CP/M, MS-DOS, and other operating systems for early microcomputers were all based on users typing in commands from the keyboard.

Engelbart invented the **GUI Graphical User Interface,** complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

In the creative world of graphic design, professional digital photography, and professional digital video production, Macintoshes are very widely used and their users are very enthusiastic about them. When Microsoft decided to build a successor to MS-DOS, it was strongly influenced by the success of the Macintosh. It produced a GUI-based system called Windows, which originally ran on top of MS-DOS (i.e., it was more like a shell than a true operating system).

For about 10 years, from 1985 to 1995, Windows was just a graphical environment on top of MS-DOS. However, starting in 1995 a freestanding version of Windows, Windows 95, was released that incorporated many operating system features into it, using the underlying MS-DOS system only for booting and running old MS-DOS programs. In 1998, a slightly modified version of this system, called Windows 98 was released.

Nevertheless, both Windows 95 and Windows 98 still contained a large amount of 16-bit Intel assembly language. Another Microsoft operating system is **Windows NT** (NT stands for New Technology), which is compatible with Windows 95 at a certain level, but a complete rewrite from scratch internally. It is a full 32-bit system. The lead designer for Windows NT was David Cutler, who was also one of the designers of the VAX VMS operating system, so some ideas from VMS are present in NT.

In fact, so many ideas from VMS were present in it that the owner of VMS, DEC, sued Microsoft. The case was settled out of court for an amount of money requiring many digits to express. Microsoft expected that the first version of NT would kill off MS-DOS and all other versions of Windows since it was a vastly superior system, but it fizzled. Only with Windows NT 4.0 did it finally catch on in a big way, especially on corporate networks. Version 5 of Windows NT was renamed Windows 2000 in early 1999. It was intended to be the successor to both Windows 98 and Windows NT 4.0. That did not quite work out either, so Microsoft came out with yet another version of Windows 98 called **Windows Me (Millennium edition).**

In 2001, a slightly upgraded version of Windows 2000, called Windows XP was released. That version had a much longer run (6 years), basically replacing all previous versions of Windows. Then in January 2007, Microsoft finally released the successor to Windows XP, called Vista. It came with a new graphical interface, Aero, and many new or upgraded user programs. Microsoft hopes it will replace Windows XP completely, but this process could take the better part of a decade.

The other major contender in the personal computer world is UNIX (and its various derivatives). UNIX is strongest on network and enterprise servers, but is also increasingly present on desktop computers, especially in rapidly developing countries such as India and China. On Pentium-based computers, Linux is becoming a popular alternative to Windows for students and increasingly many corporate users. As an aside, throughout this book we will use the term "Pentium" to mean the Pentium I, II, III, and 4 as well as its successors such as Core 2 Duo.

The term **x86** is also sometimes used to indicate the entire range of Intel CPUs going back to the 8086, whereas "Pentium" will be used to mean all CPUs from the Pentium I onwards.

Admittedly, this term is not perfect, but no better one is available. One has to wonder which marketing genius at Intel threw out a brand name (Pentium) that half the world knew well and respected and replaced it with terms like "Core 2 duo" which very few people understand—quick, what does the "2" mean and what does the "duo" mean? Maybe "Pentium 5" (or "Pentium 5 dual core," etc.) was just too hard to remember.

FreeBSD is also a popular UNIX derivative, originating from the BSD project at Berkeley. AH modern Macintosh computers run a modified version of FreeBSD. UNIX is also standard on workstations powered by high-performance RISC chips, such as those sold by Hewlett- Packard and Sun Microsystems. Many UNIX users, especially experienced programmers, prefer a commandbased interface to a GUI, so nearly all UNIX systems support a windowing system called the X Window System (also known as Xll) produced at M.I.T.

This system handles the basic window management, allowing users to create, delete, move, and resize windows using a mouse. Often a complete GUI, such as Gnome or KDE is available to run on top of Xll giving UNIX a look and feel something like the Macintosh or Microsoft Windows, for those UNIX users who want such a thing. An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running network operating systems and distributed operating systems (Tanenbaum and Van Steen, 2007). In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users).

Personal computer (PC) operating systems support complex games, business applications, and everything in between. The PC was, of course, envisioned as a ***personal computer***—an inherently single-user machine. Modern Windows, however, supports the sharing of a PC among multiple users. Each user that is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes created to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, Windows only supports a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session that was previously created. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions ***fast user switching***.

Workstation Operating System

Workstation operating system are for example, Windows XP, Windows Vista, Windows 7, Windows 8 and similar. Workstation operating system is primarily designed to run applications. Those applications can be text processor, a spreadsheet application, presentation software, video or audio editors, games, etc. Workstation operating systems can run services, but are not really designed for it. By services we mean on services that other users can use on the network. For example, services like DHCP, DNS, FTP, Mail, Web servers, etc. Well, some of that services actually are available on workstation operating systems, but they are not optimized for them. As we know, almost all workstation operating systems support multiple user accounts on the same workstation, but the thing is they are not designed to be concurrent multi-user. Workstation OS are not designed to support multiple users at the same time, meaning they don't do it very well. Most Windows operating systems have a limit of 10 concurrent users at the time. This limit is applied when we share something on our workstation computer, for example printer or some folder. Only 10 users maximum will be able to utilize our shared resources on the workstation OS. Also, workstation operating systems are designed to run on lower end hardware. That's why the workstation operating system can run on many different and cheap computers. Examples of workstation operating systems include Windows 95, Windows 98, Windows ME, Windows 2000, Windows XP, Windows Vista, Windows 7, Windows 8, and various Macintosh operating systems as well

Process control & Real Time System

Process control is an engineering discipline that deals with architectures, mechanisms and algorithms for maintaining the output of a specific process within a desired range. For instance, the temperature of a chemical reactor may be controlled to maintain a consistent product output.

Process control is extensively used in industry and enables mass production of consistent products from continuously operated processes such as oil refining, paper manufacturing, chemicals, power plants and many others. Process control enables automation, by which a small staff of operating personnel can operate a complex process from a central control room.

Embedded systems almost always run real-time operating systems. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are realtime systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building.

A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.

Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

Soft real-time systems

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc.

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period – the correctness depends not only on the logical result but also the time it was delivered – failure to respond is as bad as the wrong response!

A system is called a real-time system, when we need quantitative expression of time (i.e. real-time) to describe the behavior of the system.

Applications of Real-Time Systems Real-time systems have of late, found applications in wide ranging areas. In the following, we list some of the prominent areas of application of real-time systems and in each identified case, we discuss a few example applications in some detail. As we can imagine, the list would become very vast if we try to exhaustively list all areas of applications of realtime systems.

UNIT I

POSSIBLE QUESTIONS

(2 MARKS)

1. What is an Operating System?
2. List any four basic OS Functions?
3. What is Batch Systems
4. What is Timesharing systems
5. What is Real Time Systems?

(6 MARKS)

1. Explain Basic OS Functions.

2. Describe about Operating Systems for Personal Computers & Workstations.

3. Explain the Types of Operating Systems.

4. Discuss about (i)Batch Systems (ii)Time Sharing Systems

5. Discuss about (i)Real Time Systems (ii) Multiprogramming Systems

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

KARPAGAM ACADEMY OF HIGHER EDU

DEPARTMENT OF COMPUTER SCIENCE,

II B.Sc CS (Batch 2016-2019)

OPERATING SYSTEMS

PART - A OBJECTIVE TYPE/MULTIPLE CHOIC

ONLINE EXAMINATIONS ONE MARK

UNIT - 1

Sno	Questions	opt1	opt2	opt3
1	The primary job of an OS is to ________	command resource	manage resource	provide utilities
2	The term " Operating System " means ________	A set of programs which controls computer working	The way a computer operator works	Conversion of high-level language in to machine level language
3	With .... more than one process can be running simultaneously each on a different processer.	Multiprogramming	Uniprocessing	Multiprocessing
4	The two central themes of modern operating system are	Multiprogramming and Distributed processing	Multiprogramming and Central Processing	Single Programming and Distributed processing
5	.... is a special type of programming language used to provide instructions to the monitor simple batch processing schema	Job control language (JCL)	Processing control language (PCL)	Batch control language (BCL)
6	……………….. is a example of an operating system that support single user process and single thread	UNIX	MS-DOS	OS/2
7	File management function of the operating system includes	File creation and deletion	Disk scheduling	Process scheduling
8	The operating system of a computer serves as a software interface between the user and the ________.	Hardware	Peripheral	Memory

9	What is a shell	It is a hardware component	It is a command interpreter	It is a part in compiler
10	The main function of the command interpreter is:	to get and execute the next user-specified command	to provide the interface between the API and application program	to handle the files in operating system
11	The systems which allows only one process execution at a time, are called	uniprogramming systems	uniprocessing systems	unitasking systems
12	As OS that has strict time constraints	Sensor Node OS	Real Time OS	Mainframe OS
13	The OS that groups similar jobs is called as	Network OS	Distributed OS	Mainframe OS
14	Which one of the following error will be handle by the operating system?	power failure	lack of paper in printer	connection failure in the network
15	By operating system, the resource management can be done via:	time division multiplexing	space division multiplexing	both (a) and (b)
16	Which one of the following is not a real time operating system?	VxWorks	Windows CE	RTLinux
17	Operating System is _________	the intermediate between hardware and user	the set of programs make the hardware usable	software that controls the hardware
18	Micro code is __________	programs written in secondary storage	programs written in main memory	programs written in Read only memory
19	To access the services of operating system, the interface is provided by the _____________	system calls	API	library
20	In 1940s, the electronic digital computers has __________	OS/2 as operating system	Dos as operating system	No operating system

21	After 1960s which features the operating systems have ________	Multiprogramming	Time sharing	Real time system
22	_____ systems are required to complete a critical task within a guaranteed amount of time.	hard real time	Priority inversion	load sharing
23	Clients are _________	Network components that perform services	Network users that need various services to be performed	Network components that provide communication facility
24	A system program that combines the separately compiled modules of a program into a form suitable for execution	assembler	linking loader	cross compiler
25	A ______________manages the execution of user programs to prevent errors and improper use of the computer.	Control program	Managing Program	allocating program
26	________ is a program associated with the operating system but are not part of the kernel,	System Program	User program	System calls
27	________ is a program that includes all programs not associated with the operation of the system	Application Programs	Kernel	Thread Program
28	General-purpose computers run most of their programs from rewriteable memory, called as _________________	Floppy disk	ROM	Random access Memory
29	On systems with multiple command interpreters to choose from, the interpreters are known as _________	GUI	shells	Signal
30	The term PDA is ______________	Personal Digital Assistant	Personal Data Assistant	Personal Data Accountant
31	_______________ handle large numbers of small requests	Batch systems	Time sharing	Transaction-processing systems

32	The occurrence of an event is usually signaled by an ___________from either the hardware or the software.	interrupt	signal	service
33	Operating systems have a ______________for each device controller	Process	device driver	controller
34	CPU design that includes multiple computing cores on a single chip. Such multiprocessor systems are termed __________	multicore	uniprocessor	singlecore
35	A ___________ is a software-generated interrupt caused either by an error or by a specific request from a user program.	trap	driver	error
36	A _________is added to the hardware of the computer to indicate the current mode kernel or user	byte	character	mode bit
37	logical storage unit is called as ___________	folder	file	RAM
38	___________ is any mechanism for controlling the access of processes or users to the resources defined by a computer system.	Protection	authorization	policy
39	A _____________is an operating system that provides features such as file sharing across the network.	network operating system	Distributed OS	Parallel OS
40	______________operating systems are even more complex than multi programmed operating systems.	Time-sharing	desktop systems	Multiprogrammed systems
41	___________can save more money than multiple single-processor systems	Multiprocessor systems	desktop systems	Time sharing systems
42	_______________ is also known as parallel systems or tightly coupled systems	Multiprocessor systems	desktop systems	Time sharing systems

43	Another form of a special-purpose operating system is the	real-time system	distributed operating system	Process states
44	The message-passing facility in Windows 2000 is called	MUTUAL EXCLUSION	Buffering	local procedure call facility
45	Which process is known for initializing a microcomputer with its OS	cold booting	boot recording	booting
46	A series of statements explaining how the data is to be processed is called	instruction	compiler	program
47	Distributed systems should	high security	have better resource sharing	better system utilization
48	Which of the following is always there in a computer	Batch system	Operating system	Time sharing system
49	When did IBM released the first version of its disk operating system DOS version 1.0	1981	1982	1983
50	Main function of shared memory is________________	to use primary memory efficently	to do intra process communicatio n	to do inter process communication
51	The kernel is a____________________	memory manager	resource manager	file manager
52	________________contains the address of an instruction to be fetched from memory	Program counter (PC)	Instruction register (IR)	Control registers
53	____________________contains the instruction most recently fetched.	Program counter (PC)	Instruction register (IR)	Control registers
54	If a process fails, most operating system write the error information to a	log file	another running process	new file
55	The OS X has _________	monolithic kernel	hybrid kernel	microkernel
56	Which Operating system does not support long file names	OS/2	Windows 95	MS-DOS
57	Which Operating system does not support networking between computers	Windows 3.1	Windows 95	Windows 2000

58	Which Operating system is better for implementing client server network	MS DOS	Windows 95	Windows 98
59	____________is the commercial UNIX-based operating system of Sun Microsystems.	Solaris	UNIX	Linux
60	___________ is an example of an open-source operating system	GNU/Linux	Windows 3.1	Windows NT

CATION

CA & IT

E QUESTIONS

S QUESTIONS

opt4	opt5	opt6	answer
Be user friendly			**manage resource**
The way a floppy disk drive operates			**A set of programs which controls computer working**
Uniprogramming			**Multiprogramming**
None of above			**Multiprogramming and Distributed processing**
Monitor control language (MCL)			**Job control language (JCL)**
Windows 2000			**MS-DOS**
Multiprogramming			**File creation and deletion**
Screen			**Hardware**

It is a tool in CPU scheduling			**It is a command interpreter**
none of the mentioned			**to get and execute the next user-specified command**
none of the mentioned			**uniprogramming systems**
Timesharing OS			**Real Time OS**
Batch OS			**Batch OS**
all of the mentioned			**all of the mentioned**
none of the mentioned			**both (a) and (b)**
Palm OS			**Palm OS**
all the above			**all the above**
none of the above			**programs written in Read only memory**
assembly instructions			**system calls**
UNIX as operating system			**No operating system**

All the above			**All the above**
Priority inheritance			**hard real time**
none			**Network users that need various services to be performed**
load and go			**linking loader**
User program			**Control program**
Functions			**System Program**
Process			**Application Programs**
Hard disk			**Random access Memory**
Command			**shells**
Private Digital Assistant			**Personal Digital Assistant**
Distributed systems			**Transaction-processing systems**

routine			**interrupt**
allocator			**device driver**
multichips			**multicore**
program			**trap**
integer			**mode bit**
ROM			**file**
privacy			**Protection**
Sensor OS			**network operating system**
Multiprocessor systems			**Time-sharing**
Multiprogrammed systems			**Multiprocessor systems**
Multiprogrammed systems			**Multiprocessor systems**

multiframe computer system			**real-time system**
CRITICAL SECTIONS			**local procedure call facility**
warm booting			**booting**
interpretor			**program**
low system overhead			**have better resource sharing**
Controlling system			**Operating system**
1984			**1981**
to do other process communication			**to do inter process communicatio n**
directory manager			**resource manager**
Status registers			**Instruction register (IR)**
Status registers			**Program counter (PC)**
none of the mentioned			**log file**
monolithic kernel with modules			**hybrid kernel**
Windows NT			**MS-DOS**
Windows NT			**Windows 3.1**

Windows 2000			**Windows 2000**
Macintosh			**Solaris**
Macintosh			**GNU/Linux**

UNIT II

Syllabus

Operating System Organization: Processor and user modes-Kernels-System Calls and System Programs. **Process Management:** System view of the process and resources- Process abstraction-Process hierarchy-Threads-Threading issues-Thread libraries-Process Scheduling-Non pre-emptive and Preemptive scheduling algorithms-Concurrent and processes-Critical Section-Semaphores-Methods for inter-process communication- Deadlocks.

Operating System Organization

Processor and user modes

The unrestricted mode is often called *kernel mode*, but many other designations exist (*master mode*, *supervisor mode*, *privileged mode*, etc.).

Restricted modes are usually referred to as *user modes*, but are also known by many other names (*slave mode, problem state*, etc.).

1. Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.
- It can execute any CPU instruction and reference any memory address.
- Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

2. User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.
- Code running in user mode must allot to system APIs to access hardware or memory.
- Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

At the very least, we need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**).

A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).

With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 2.1. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

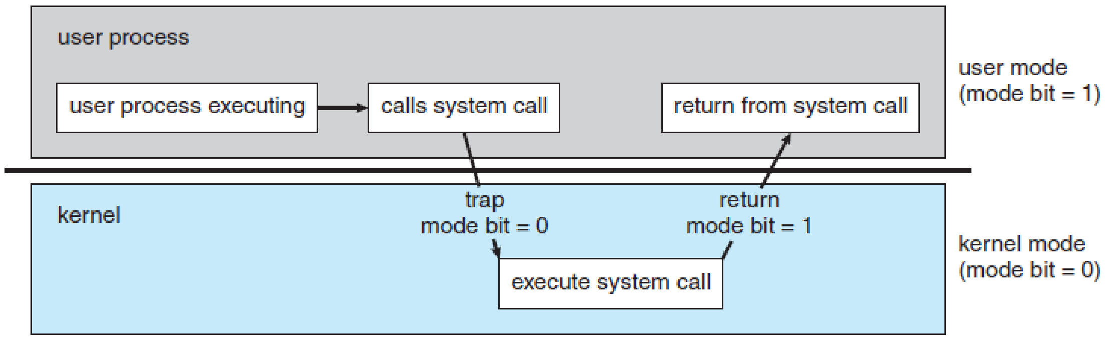


Fig 2.1 Transition from user to kernel mode.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.

The life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

Kernels

A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware. The sole aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory etc). The main tasks of the kernel are :

- Process management
- Device management
- Memory management
- Interrupt handling
- I/O communication
- File system...etc..

Types Of Kernels

Kernels may be classified mainly in two categories

1. Monolithic

2. Micro Kernel

1 Monolithic Kernels

Earlier in this type of kernel architecture, all the basic system services like process and memory management, interrupt handling etc were packaged into a single module in kernel space.

This type of architecture led to some serious drawbacks like

1) Size of kernel, which was huge.

2) Poor maintainability, which means bug fixing or addition of new features resulted in

recompilation of the whole kernel which could consume hours.

A common approach is to partition the task into small components, or modules, rather than have one **monolithic** system.

In a modern day approach to monolithic architecture, the kernel consists of different modules which can be dynamically loaded and un-loaded. This modular approach allows easy extension of OS's capabilities. With this approach, maintainability of kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module. So, there is no need to bring down and recompile the whole kernel for a smallest bit of change. Also, stripping of kernel for various platforms (say for embedded devices etc) became very easy as we can easily unload the module that we do not want.

Linux follows the monolithic modular approach

2 Microkernels

This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.

This architecture caters to the problem of ever growing size of kernel code which we could not control in the monolithic approach. This architecture allows some basic services like device driver management, protocol stack, file system etc to run in user space. This reduces the kernel code size and also increases the security and stability of OS as we have the bare minimum code running in kernel. So, if suppose a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through **message passing,**

In this architecture, all the basic OS services which are made part of user space are made to run as servers which are used by other programs in the system through inter process communication (IPC). eg: we have servers for device drivers, network protocol stacks, file systems, graphics, etc. Microkernel servers are essentially daemon programs like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs. This allows some servers, particularly device drivers, to interact directly with hardware. These servers are started at the system start-up.

So, what the bare minimum that micro Kernel architecture recommends in kernel space?

- Managing memory protection
- Process scheduling
- Inter Process communication (IPC)

Apart from the above, all other basic services can be made part of user space and can be run in the form of servers.

System Calls

The system call provides an interface to the operating system services.

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

- Portability: as long a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier then using the actual system call.

System Call Parameters

Three general methods exist for passing parameters to the OS:

1. Parameters can be passed in registers.
2. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register.
3. Parameters can also be pushed on or popped off the stack by the operating system.

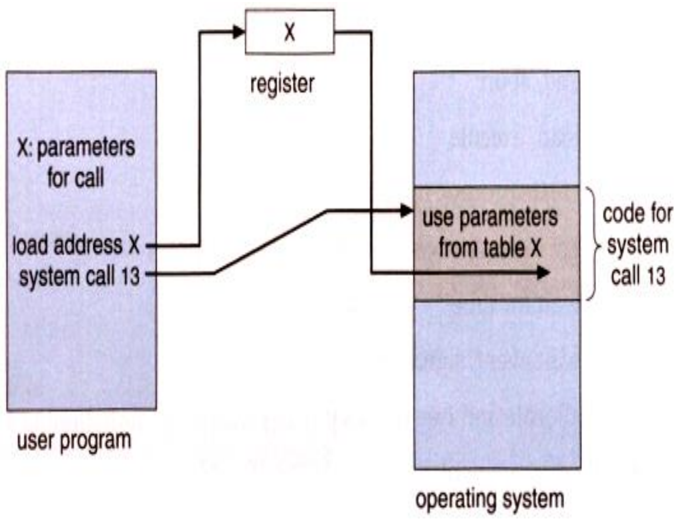


Fig 2.2 System call parameters

Types of System Calls

There are 5 different categories of system calls:

process control, file manipulation, device manipulation, information maintenance and communication.

Process Control

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

File Management

Some common system calls are *create*, *delete*, *read*, *write*, *reposition*, or *close*. Also, there is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls.

Device Management

Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

User programs *request* the device, and when finished they *release* the device. Similar to files, we can *read*, *write*, and *reposition* the device.

Information Management

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*.

The OS also keeps information about all its processes and provides system calls to report this information.

Communication

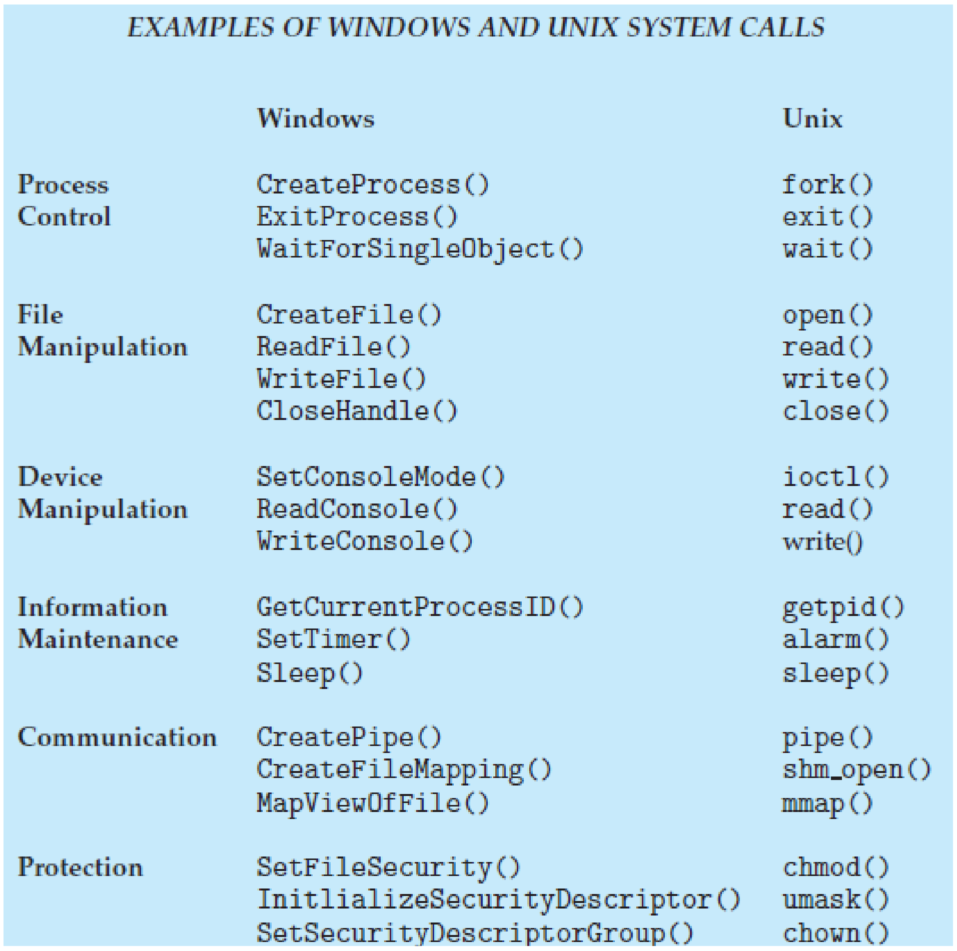
There are two models of interprocess communication, the message-passing model and the shared memory model.

- Message-passing uses a common mailbox to pass messages between processes.
- Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.

System Call

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

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

System Programs

System programs, also known as **system utilities**, provide a convenient environment for program development and execution.

Some of them are simply user interfaces to system calls.

These programs are not usually part of the OS kernel, but are part of the overall operating system.

They can be divided into these categories:

• **File management**. These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

• **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.

• **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

• **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.

• **Program loading and execution**. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

• **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

• **Background services**. All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or daemons.

One example is the network daemon, a system needed a service to listen for network connections in order to connect those requests to the correct processes.

Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons.

In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities. Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations.

Such **application programs** include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the Mac OS X operating system, the user might see the GUI, featuring a mouse-and-windows interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways.

Process Management:

System view of the process and resources

The Process

A process generally consists of:

• The program's instructions (aka. the "program text")

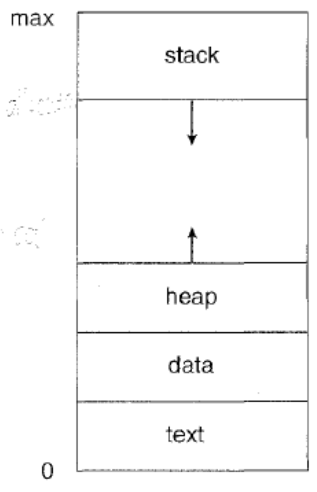
• CPU state for the process (program counter, registers, flags, …)

• Memory state for the process

• Other resources being used by the process

Informally, as mentioned earlier, a process is a program in execution. 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 is shown in Figure 2.1.

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), whereas 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 are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)



Process in memory

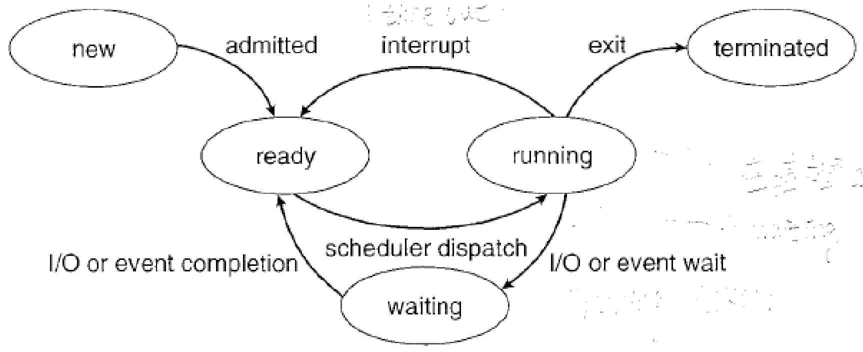


Fig 2.3 Diagram of process state

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

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.

Each 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/0 completion or reception of a signal).

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

Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*, however.

The state diagram corresponding to these states is presented in Figure 2.2.

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 2.3. It contains many pieces of information associated with a specific process, including these:

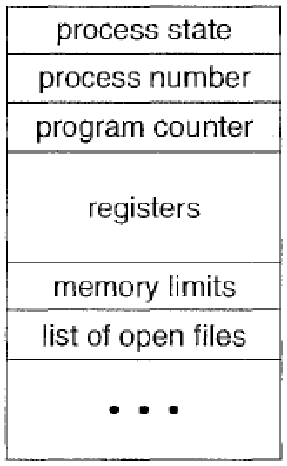


Fig 2.4. Process Control Block

• **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.

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.

Process Abstraction

Process abstraction Main Point: What are processes? How are process, programs, threads, and address spaces related?

The Abstraction: A Process

- The abstraction provided by the OS of a running program is something we will call a process. As we said above, a process is simply a running program; at any instant in time,
- One obvious component of machine state that comprises a process is its memory.
- Instructions lie in memory; the data that the running program reads and writes sits in memory as well.
- Thus the memory that the process can address (called its address space) is part of the process.
- Also part of the process's machine state are registers; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process.
- Note that there are some particularly special registers that form part of this machine state.
- Finally, programs often access persistent storage devices too. Such I/O information might include a list of the files the process currently has open.

Process Hierarchies

Modern general purpose operating systems permit a user to create and destroy processes.

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

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

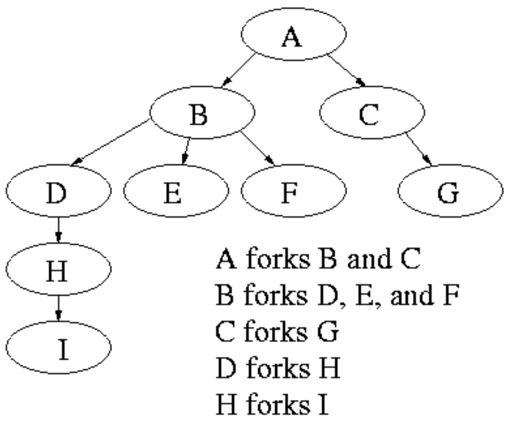


Fig 2.5 Process Hierarchy

Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy.

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually ail active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a role, let us look at how UNIX initializes itself when it is started. A special process, called *init*, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off one new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root.

In contrast, Windows has no concept of a process hierarchy. All processes are equal. The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a **handle)** that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

Threads

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

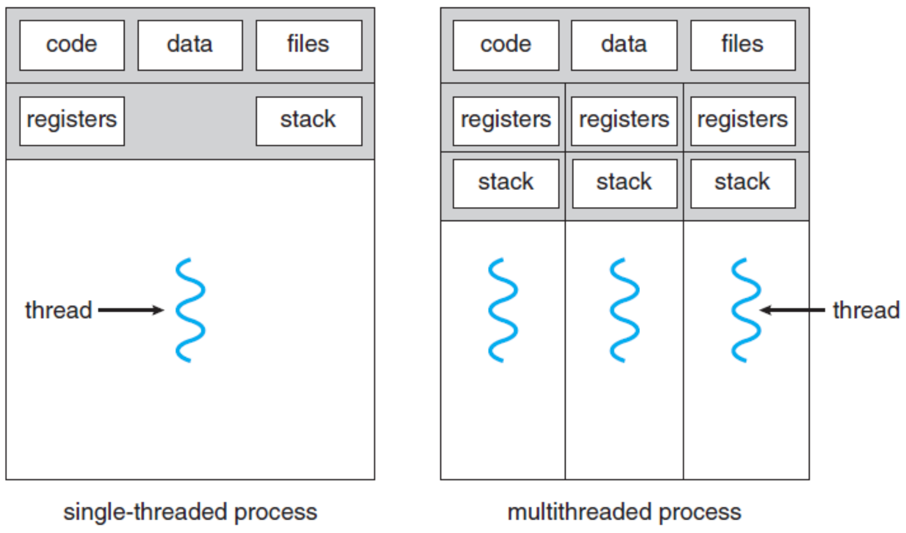


Fig 2.6. Single-threaded and multithreaded processes

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.

- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

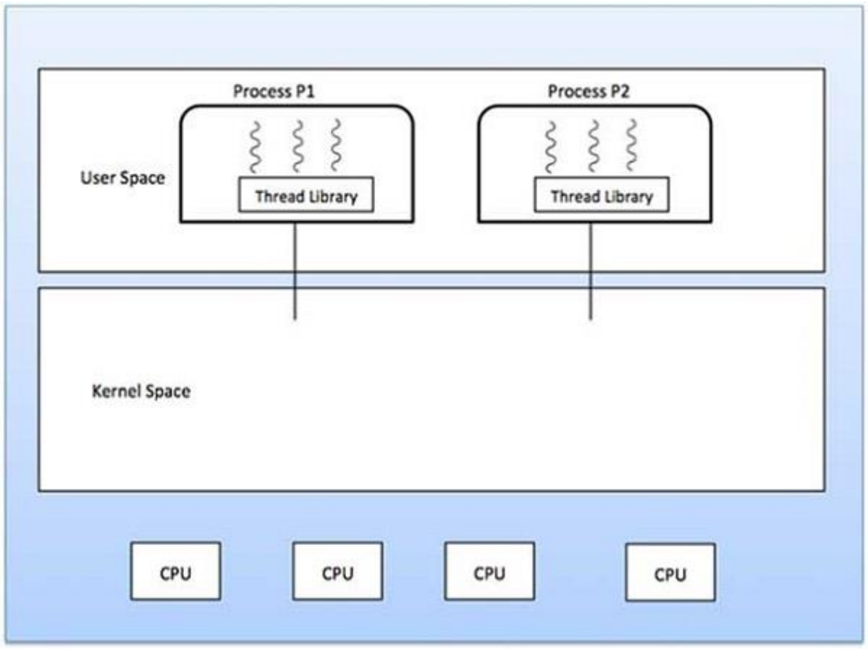


Fig 2.7. User level threads

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.

- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

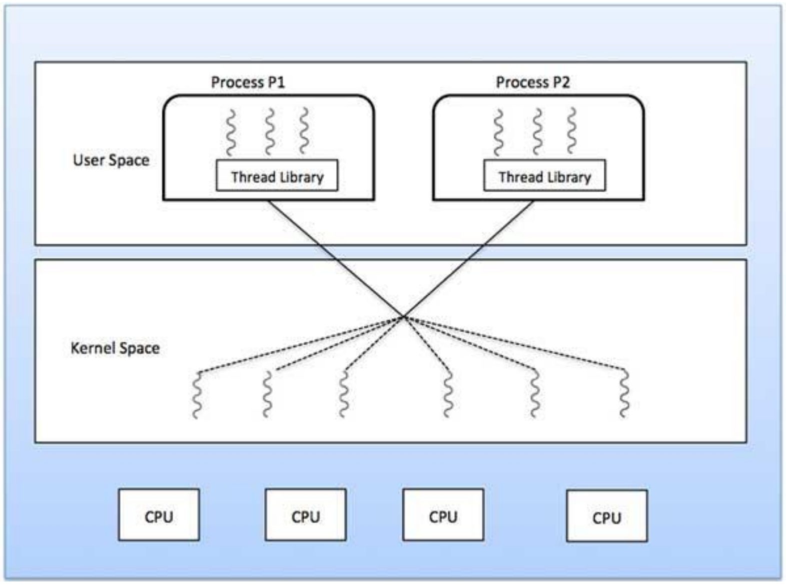


Fig 2.8. Many to Many Model

Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

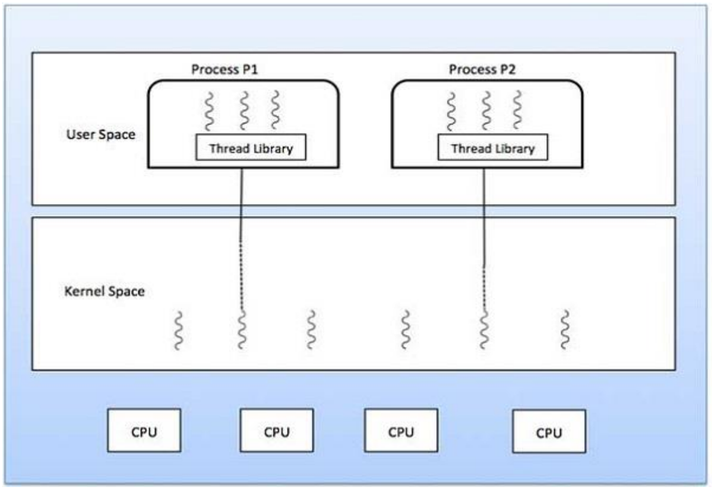


Fig 2.9. Many to One Model

One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

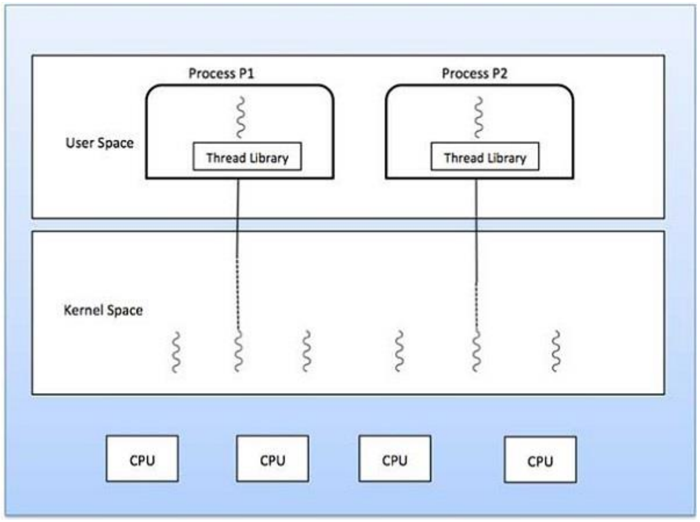


Fig 2.10.One to One Model

Threading issues

There are a variety of issues to consider with multithreaded programming

- Semantics of fork() and exec() system calls
- Signal handling
- Thread cancellation
- Thread-Local Storage
- Scheduler Activations

Semantics of fork() and exec()

Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

• The exec() system call continues to behave as expected. if a thread invokes the exec() system call, the program specified in the parameter to exec () will replace the entire process-including all threads. If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process.

- Replaces the entire process that called it, including all threads

• If planning to call exec() after fork(), then there is no need to duplicate all of the threads in the calling process

- All threads in the child process will be terminated when exec() is called.

Signal Handling

• Signals are used in UNIX systems to notify a process that a particular event has occurred - CTRL-C is an example of an asynchronous signal that might be sent to a process

An asynchronous signal

• An asynchronous signal is one that is generated from outside the process that receives it - Divide by 0 is an example of asynchronous signal that might be sent to a process

Synchronous signal

• A synchronous signal is delivered to the same process that caused the signal to occur

• All signals follow the same basic pattern:

1. A signal is generated by particular event
2. The signal is delivered to a process
3. The signal is handled by a signal handler (all signals are handled exactly once)

All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.

2. The signal is delivered to a process.

3. Once delivered, the signal must be handled.

A signal may be ***handled*** by one of two possible handlers:

1. A default signal handler

2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general the following options exist:

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

(1) Deliver the signal to the thread to which the signal applies

Most likely option when handling synchronous signals (e.g. only the thread that attempts to divide by zero needs to know of the error)

(2) Deliver the signal to every thread in the process

- Likely to be used in the event that the process is being terminated (e.g. a CTRLC is sent to terminate the process, all threads need to receive this signal and terminate)

Thread Cancellation

• Thread cancellation is the act of terminating a thread before it has completed

- Example

- clicking the stop button on your web browser will stop the thread that is rendering the web page

• The thread to be cancelled is called the target thread

• Threads can be cancelled in a couple of ways

- Asynchronous cancellation

- Deferred cancellation

- Asynchronous cancellation terminates the target thread immediately

- Deferred cancellation allows the target thread to periodically check if it should be cancelled

• Allows thread to terminate itself in an orderly fashion

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**.

Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation. One thread immediately terminates the target thread.

2. Deferred cancellation. The target thread periodically checks whether it

should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Thread-Local Storage

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**.) For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-local storage.

Scheduler Activations

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlersmust run on a virtual processor. One event that triggers an upcall occurs when an application thread is about to block.

Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel. Three main thread libraries are in use today:

- POSIX Pthreads,
- Windows, and
- Java.

Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***. Operating-system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux, Mac OS X, and Solaris. Although Windows doesn't support Pthreads natively, some third party implementations for Windows are available.

The Windows thread library is a kernel-level library available on Windows systems. The technique for creating threads using theWindows thread library is similar to the Pthreads technique in several ways.

The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using theWindows API; UNIX and Linux systems often use Pthreads.

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads.

All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and Mac OS X. The Java thread API is available for Android applications as well.

Process scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a 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. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

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.

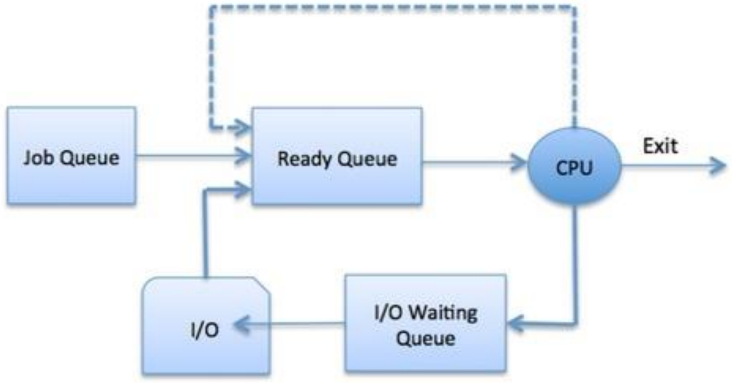


Fig 2.11. Process scheduling queues

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.

Schedulers

Schedulers are special system software which handle 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

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.

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.

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 processes 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.

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Non pre-emptive and Preemptive scheduling algorithms

Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called **preemptive scheduling.**

Eg: Round robin

In **non-preemptive scheduling**, a running task is executed till completion. It cannot be interrupted.
Eg First In First Out

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

1. When a process switches from the running state to the waiting state (for example, as the result of an I/0 request or an invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/0)
4. When a process terminates.

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

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
- Multiple-Level Queues 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.

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

Shortest Job Next (SJN)

- 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

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

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.

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

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.
-

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Concurrent process

- Modern operating systems can handle more than one process at a time
- System scheduler manages processes and their competition for the CPU
- Memory manager role is to manage sharing of main memory between active processes

Look at how processes coexist and communicate with each other on modern computer

- Concurrency is good for users
  - One of the reasons for multiprogramming
    - Working on the same problem, simultaneous execution of programs, background execution
- Concurrency is a "pain in the neck" for the system
  - Access to shared data structures
  - Deadlock due to resource contention

Enabling process interaction

A ***parallel program*** is a concurrent program with more than one thread executing simultaneously.

- Fully indenendant – separate applications running on the one system
- Indenpendant but related – example users running their own copy of a data entry program but accessing and updating the one database

Concurrent processes – set of cooperating processes, example C program

- It is necessary for some resources to remain allocated to a process for as long as process requires. Serial reusable resources require mutual exclusion. Example printer is being used by a process then it must remain allocated to the process until printout is complete.

Mutual exclusion gives rise to another problem which OS must handle – deadlock.

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads, discussed in Chapter 4. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Critical section

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.do {

```
        entry section

                critical section

        exit section

                remainder section

} while (true);
```

General structure of a typical process *Pi* .

The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so called critical-section problem. Consider a system consisting of *n* processes {*P*0, *P*1, ..., *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 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 entry section and exit section are enclosed in boxes to highlight these important segments of code.

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, then no other processes can be executing in their critical sections.

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.

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.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

All these solutions are based on the premise of **locking** —that is, protecting critical regions through the use of locks.

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.

The classical definition of wait and signal are :

- Wait : decrement the value of its argument S as soon as it would become non-negative.
- Signal : increment the value of its argument, S as an individual operation.

Properties of Semaphores

1. Simple
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores
5. Can permit multiple processes into the critical section at once, if desirable. Types of Semaphores

Semaphores are mainly of two types:

1. **Binary Semaphore**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called *Mutex*. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.

2. **Counting Semaphores**

   These are used to implement bounded concurrency. Limitations of Semaphores

1. Priority Inversion is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called Deadlock. Binary semaphore can take the value 0 & 1 only. Counting semaphore can take nonnegative integer values.

Two standard operations, wait and signal are defined on the semaphore. Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care by signal operation. The wait, signal operations are also called P and V operations. The manipulation of semaphore (S) takes place as following:

1. The wait command P(S) decrements the semaphore value by 1. If the resulting value becomes negative then P command is delayed until the condition is satisfied.
2. The V(S) i.e. signals operation increments the semaphore value by 1.

Mutual exclusion on the semaphore is enforced within P(S) and V(S). If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting.These operations are defined as under −

```
P(S) or wait(S):

If S > 0 then

  Set S to S-1

Else

  Block the calling process (i.e. Wait on S)



V(S) or signal(S):

If any processes are waiting on S

  Start one of these processes

Else

  Set S to S+1
```

The semaphore operation are implemented as operating system services and so wait and signal are atomic in nature i.e. once started, execution of these operations cannot be interrupted.

Thus semaphore is a simple yet powerful mechanism to ensure mutual exclusion among concurrent processes.

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can for processes to synchronize their activities.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch ***proberen,*** “to test”); signal() was originally called V (from ***verhogen,*** “to increment”). The definition of wait() is as follows:

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

; // busy wait

S--;

}
```

The definition of signal() is as follows:

```
signal(S) { S++;

}
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption. First, let’s see how semaphores can be used.

Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphores are used to solve various synchronization problems. For example, consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2. Suppose we require that *S*2 be executed only after *S*1 has completed. We can implement this scheme readily by letting *P*1 and *P*2 share a common semaphore synch, initialized to 0. In process *P*1, we insert the statements

*S*1;

signal(synch);

In process *P*2, we insert the statements

wait(synch);

*S*2;

Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked

signal(synch), which is after statement *S*1 has been executed.

Methods for inter-process communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.12.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared

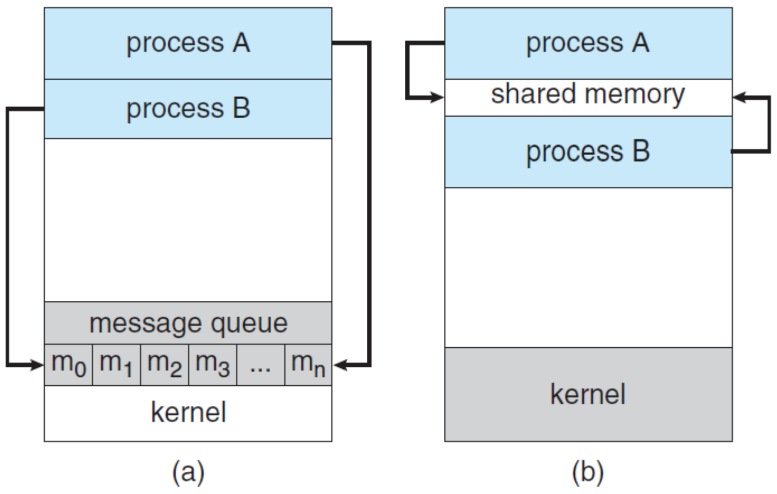


Fig 2.12. Communications models. (a) Message passing. (b) Shared memory.

memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. Recent research on systems with several processing cores indicates that message passing provides better performance than shared memory on such systems. Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches. As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC. In the remainder of this section, we explore shared-memory and message passing systems in more detail.

1 Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. The producer–consumer problem also provides a useful metaphor for the client–server paradigm.We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource. One solution to the producer–consumer problem uses shared memory.

To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems

cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region ofmemory and that the code for accessing andmanipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passingfacility.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

send(message) receive(message)

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design. If processes *P* and *Q* want to communicate, theymust send messages to and receive messages from each other: a ***communication link*** must exist between them. This link can be implemented in a variety of ways.We are concerned here not with the link's physical implementation) but rather with its logical implementation. Here are several methods for logically implementing a link

and the send()/receive() operations:

• Direct or indirect communication

• Synchronous or asynchronous communication

• Automatic or explicit buffering

Issues related to each of these features next.

1 Naming

Processes that want to communicate must have a way to refer to each other.

They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate

must explicitly name the recipient or sender of the communication. In this

scheme, the send() and receive() primitives are defined as:

• send(P, message)—Send a message to process P.

• receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that

want to communicate. The processes need to know only each other's

identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link. This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

• send(P, message)—Send a message to process P.

• receive(id, message)—Receive a message from any process. The

variable id is set to the name of the process with which communicationhas taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such ***hard-coding*** techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With ***indirect communication***, the messages are sent to and received from ***mailboxes***, or ***ports***.A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send() and receive()

primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes *P*1, *P*2, and *P*3 all share mailbox *A*. Process *P*1 sends a message to *A*, while both *P*2 and *P*3 execute a receive() from *A*. Which process will receive the message sent by *P*1? The answer depends on which of the following methods we choose:

• Allow a link to be associated with two processes at most.

• Allow at most one process at a time to execute a receive() operation.

• Allow the system to select arbitrarily which process will receive the message (that is, either *P*2 or *P*3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, ***round robin,*** where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows aprocess to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

2 Synchronization

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

• **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.

• **Nonblocking send**. The sending process sends the message and resumes operation.

• **Blocking receive**. The receiver blocks until a message is available.

• **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Different combinations of send() and receive() are possible.When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements. The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available. This is

illustrated in Figures 3.15 and 3.16.

3 Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

```
message next produced;

while (true) { /* produce an item in next produced */

send(next produced);

}
```

Figure The producer process using message passing.

• **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messageswaiting in it. In this case, the sender must block until the recipient receives the message.

• **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks. The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use. The process can operate on the resource (for example, if the resource

is a printer, the process can print on the printer).

3. **Release**. The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously

in a system:

1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

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

3. No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set {$P0$, $P1$, ..., Pn} of waiting processes must exist such that $P0$ is waiting for a resource held by $P1$, $P1$ is waiting for a resource held by $P2$, ..., $Pn-1$ is waiting for a resource held by Pn, and Pn is waiting for a resource held by $P0$.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P = {$P1$, $P2$, ..., Pn}, the set consisting of all the active processes in the system, and R = {$R1$, $R2$, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from process *Pi* to resource type *Rj* is denoted by *Pi* → *Rj* ; it signifies that process *Pi* has requested an instance of resource type *Rj* and is currently waiting for that resource. A directed edge from resource type *Rj* to process *Pi* is denoted by *Rj* → *Pi*; it signifies that an instance of resource type *Rj* has been allocated to process *Pi*. A directed edge *Pi* → *Rj* is called a **request edge**; a directed edge *Rj* → *Pi* is called an **assignment edge**.

Pictorially, we represent each process *Pi* as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *Rj* , whereas an assignment edge must also designate one of the dots in the rectangle.

When process *Pi* requests an instance of resource type *Rj*, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 7.1 depicts the following

situation.

• The sets *P*, *R*, and *E*:

◦ *P* = {*P*1, *P*2, *P*3}

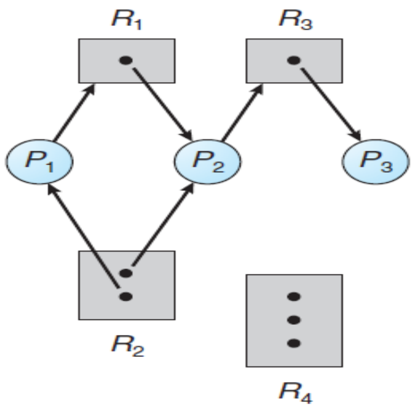


Fig 2.13. Resource-allocation graph.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

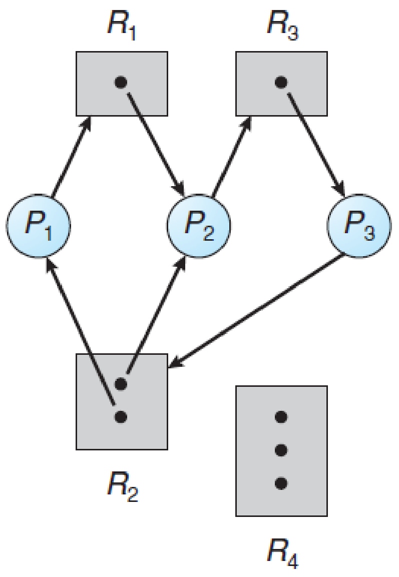


Fig 2.14. Resource-allocation graph with a deadlock.

type *R*2. Since no resource instance is currently available, we add a request edge *P*3→ *R*2 to the graph (Figure 7.2). At this point, two minimal cycles exist in the

system:

*P*1 → *R*1 → *P*2 → *R*3 → *P*3 → *R*2 → *P*1

*P*2 → *R*3 → *P*3 → *R*2 → *P*2

Processes *P*1, *P*2, and *P*3 are deadlocked. Process *P*2 is waiting for the resource

*R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or

process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process

*P*2 to release resource *R*1

Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three

ways:

• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.

• We can allow the system to enter a deadlocked state, detect it, and recover.

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Methods

■Deadlock Prevention

■ Deadlock Avoidance

■Deadlock Detection

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several processes.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file, and then request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

A set $\{P0, P1, ..., Pn\}$ of waiting processes must exist such that $P0$ is waiting for a resource held by $P1$, $P1$ is waiting for a resource held by $P2$, ..., $Pn-1$ is waiting for a resource held by Pn, and Pn is waiting for a resource held by $P0$.

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Although ensuring that resources are acquired in the proper order is the responsibility of application developers, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible. One lock-order verifier, which works on BSD versions of UNIX such as FreeBSD, is known as **witness**

Deadlock Avoidance

Requires that the system has some additional a priori information available.

■ Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

■ The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

■ Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

A state is *safe* if the system can allocate resources to each process (up to its

maximum) in some order and still avoid a deadlock. More formally, a system

is in a safe state only if there exists a **safe sequence**.

Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm.** The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Deadlock Detection

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

• An algorithm that examines the state of the system to determine whether a deadlock has occurred

• An algorithm to recover from the deadlock

Deadlock Detection

■ Allow system to enter deadlock state

■ Detection algorithm

■ Recovery scheme

Handling Deadlock

The above points focus on preventing deadlocks. But what to do once a deadlock has occured. Following three strategies can be used to remove deadlock after its occurrence.

1. **Preemption**

   We can take a resource from one process and give it to other. This will resolve the deadlock situation, but sometimes it does causes problems.

2. **Rollback**

   In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process and when deadlock occurs, roll everything back to the last checkpoint, and restart, but allocating resources differently so that deadlock does not occur.

3. **Kill one or more processes**

   This is the simplest way, but it works.

UNIT II

POSSIBLE QUESTIONS

(2 MARKS)

1. What is a Kernel?
2. What is Process?
3. What is a Deadlock?
4. What is System Call?
5. What is preemptive scheduling?

(6 MARKS)

1. Explain in detail about System Calls and System Programs.
2. Write a note on Non pre-emptive and Preemptive scheduling algorithms.
3. Discuss about (i)Kernels (ii)Processor and user modes
4. Explain in detail about Threads and Threading issues
5. Explain the concept of System view of the process and resources.
6. Explain about Deadlocks in detail.
7. Discuss about (i)Process (ii)Process hierarchy.
8. Describe about Critical Section and Semaphores in detail
9. Explain about Process Scheduling in detail.
10. Discuss about Methods for inter-process communication

UNIT III

Syllabus

Memory Management: Physical and Virtual address space-Memory Allocation strategies – Fixed and Variable partitions-Paging-Segmentation-Virtual memory

Memory Management

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.

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.

Memory consists of a large array of bytes, each with its own address.

The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).

Basic Hardware

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.

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, as illustrated in Figure 3.1.

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).

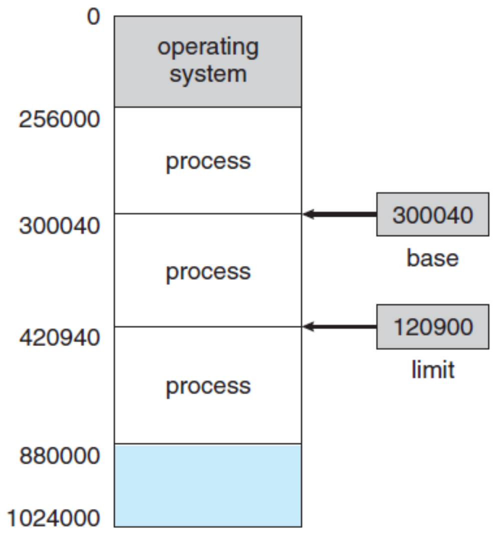


Fig 3.1. A base and a limit register define a logical address space.

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**.

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.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated –

1 Symbolic addresses

The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.

2 Relative addresses

At the time of compilation, a compiler converts symbolic addresses into relative addresses.

3 Physical addresses

The loader generates these addresses at the time when a program is loaded into main memory.

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. 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**.

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.

Logical versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

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. 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**.

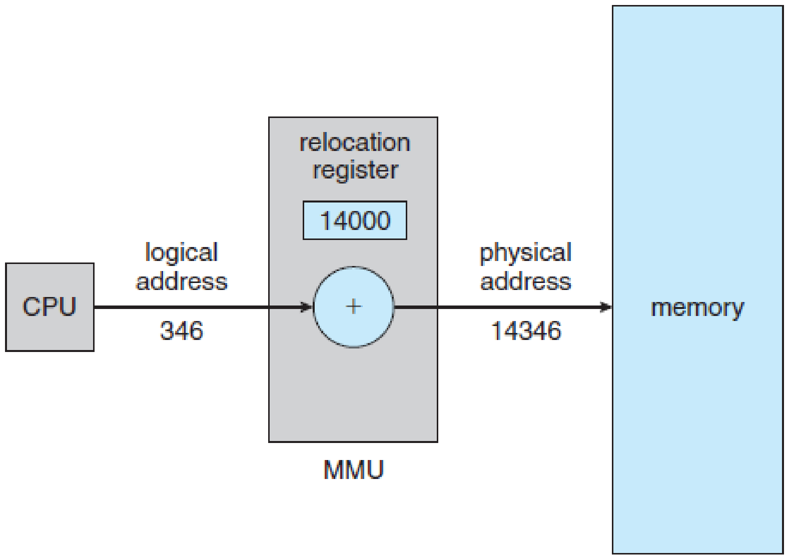


Fig 3.2. Physical address and Logical Address

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)**. 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 (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location14000; an access to location 346 is mapped to location 14346.

The user program never sees the real physical addresses.

Static vs Dynamic Loading

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory to a backing store , and then brought back into memory for continued execution.

Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users. It must be capable of providing direct access to these memory images.

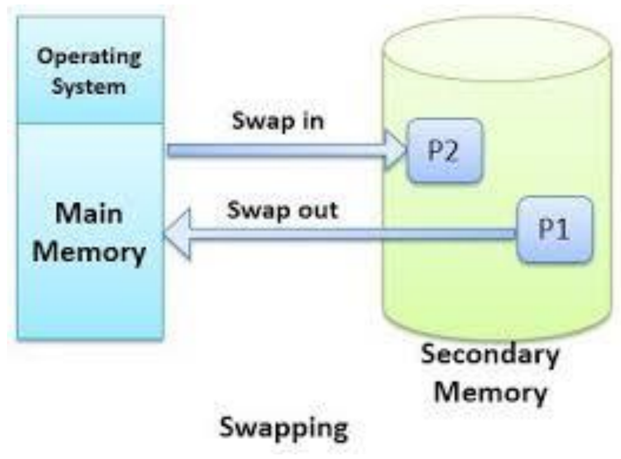


Fig 3.3. Process of Swapping

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 **multiplepartition 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. it is used primarily in batch environments.

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.

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.

Memory Allocation Strategies

1. First Fit
2. Best fit
3. Worst fit
4. Next fit

First Fit

• **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage

Fastest algorithm because it searches as little as possible.

Disadvantage

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

First Fit

The first of these is called first fit. The basic idea with first fit allocation is that we begin searching the list and take the first block whose size is greater than or equal to the request size. If we reach the end of the list without finding a suitable block, then the request fails.

To illustrate the behavior of first fit allocation, as well as the other allocation policies later, we trace their behavior on a set of allocation and deallocation requests. We denote this sequence as A20, A15, A10, A25, D20, D10, A8, A30, D15, A15, where An denotes an allocation request for n KB and Dn denotes a deallocation request for the allocated block of size n KB. (For simplicity of notation, we have only one block of a given size allocated at a time. None of the policies depend on this property; it is used here merely for clarity.) In these examples, the memory space from which we serve requests is 128 KB. Each row of Figure 9-9 shows the state of memory after the operation labeling it on the left.

Shaded blocks are allocated and unshaded blocks are free. The size of each block is shown in the corresponding box in the figure. In this, and other allocation figures in this chapter, time moves downward in the figure. In other words, each operation happens prior to the one below it.

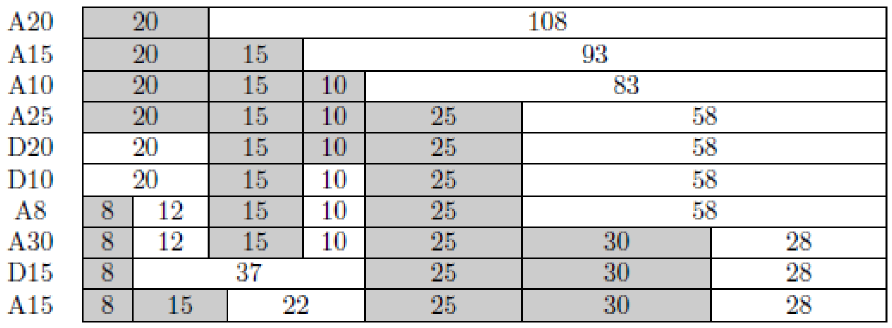


Fig 3.4. First Fit Allocation

Next Fit

If we want to spread the allocations out more evenly across the memory space, we often use a policy called next fit. **This scheme is very similar to the first fit approach, except for the place where the search starts. In next fit, we begin the search with the free block that was next on the list after the last allocation.** During the search, we treat the list as a circular one. If we come back to the place where we started without finding a suitable block, then the search fails.

For the next three allocation policies in this section, the results after the first six requests (up through the D10 request) are the same. In Figure 9-10, we show the the results after each of the other requests when following the next fit policy.

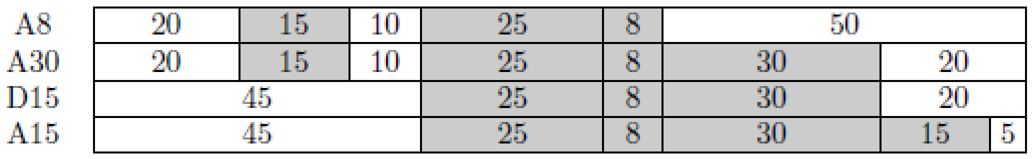


Fig 3.5. Next Fit Allocation

Best Fit

• **Best fit**. Allocate the smallest hole that is big enough.Wemust search theentire list, unless the list is ordered by size. This strategy produces thesmallest leftover hole.

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage

It is slower and may even tend to fill up memory with tiny useless holes.

In many ways, the most natural approach is to allocate the free block that is closest in size to the request. This technique is called best fit. In best fit, we search the list for the block that is smallest but greater than or equal to the request size.

As with the next fit example, we show only the final four steps of best fit allocation for our example.

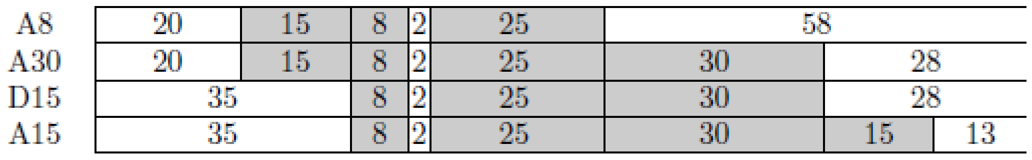


Fig 3.6. **Best Fit Allocation**

Worst fit

• **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage

Reduces the rate of production of small gaps.

Disadvantage

If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

Worst Fit

If best fit allocates the smallest block that satisfies the request, then worst fit allocates the largest block for every request. Although the name would suggest that we would never use the worst fit policy, it does have one advantage: If most of the requests are of similar size, a worst fit policy tends to minimize external fragmentation.

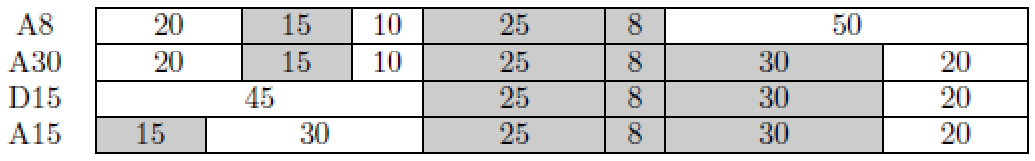


Fig 3.7. Worst fit allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

1 Single-partition allocation

In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.

2 Multiple-partition allocation

In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. 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.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

1 External fragmentation

Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

2 Internal fragmentation

Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

Fixed and Variable partitions

1. Fixed Partitioning: Main memory is divided into a no. of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.

Memory Manager will allocate a region to a process that best fits it

Unused memory within an allocated partition called internal fragmentation.

Advantages:

Simple to implement

Little OS overhead

Disadvantages:

Inefficient use of Memory due to internal fragmentation. Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as internal fragmentation.

Two possibilities:

a). Equal size partitioning

b). Unequal size Partition

Not suitable for systems in which process memory requirements not known ahead of time; i.e. timesharing systems

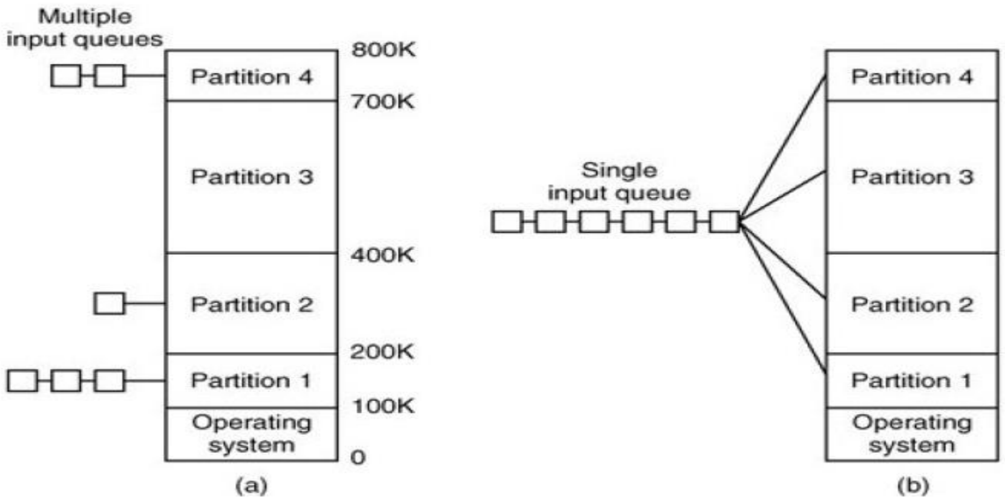


Fig 3.8. (a) Fixed memory partitions with separate input queues for each partition.

(b) Fixed memory partitions with a single input queue.

When the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3. Here small jobs have to wait to get into memory, even though plenty of memory is free.

An alternative organization is to maintain a single queue as in Fig. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

2. Dynamic/Variable Partitioning:

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic

partitioning was developed . The partitions are of variable length and number. When a process is

brought into main memory, it is allocated exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure Eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory

becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as **external fragmentation**, indicating that the memory that is external to all partitions becomes increasingly fragmented.

One technique for overcoming external fragmentation is compaction: From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure h, compaction will result in a block of free memory of length 16M.

This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time.

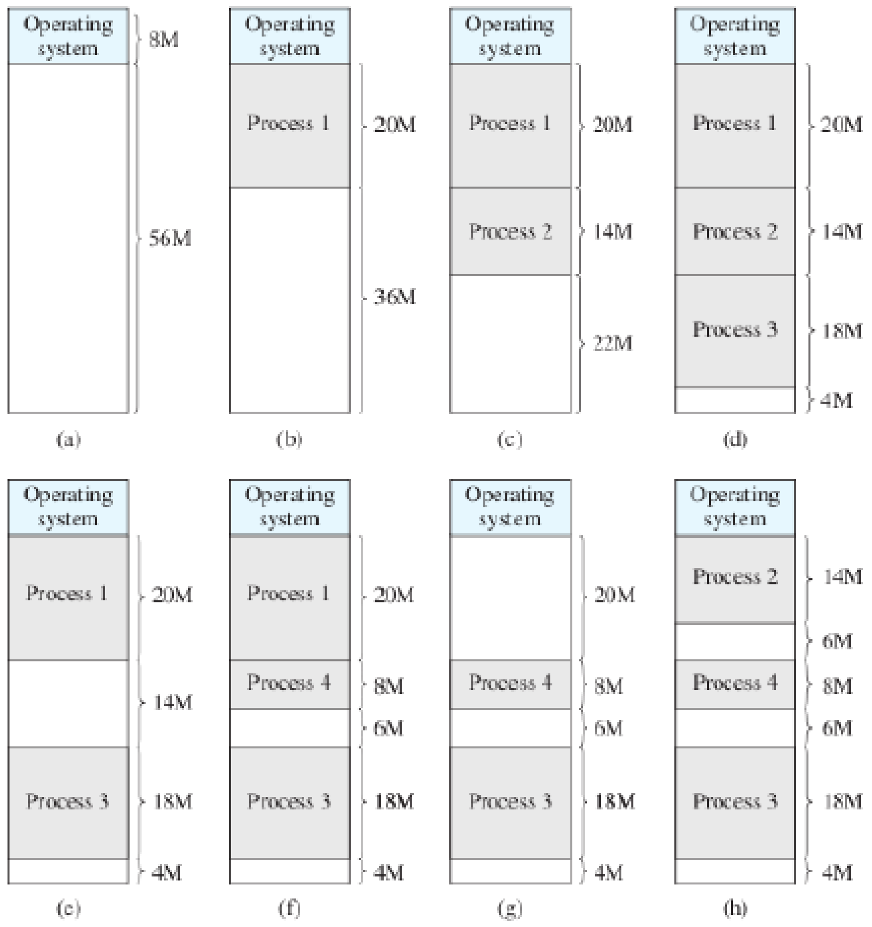


Fig.3.9. The Effect of dynamic partitioning

Paging

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.

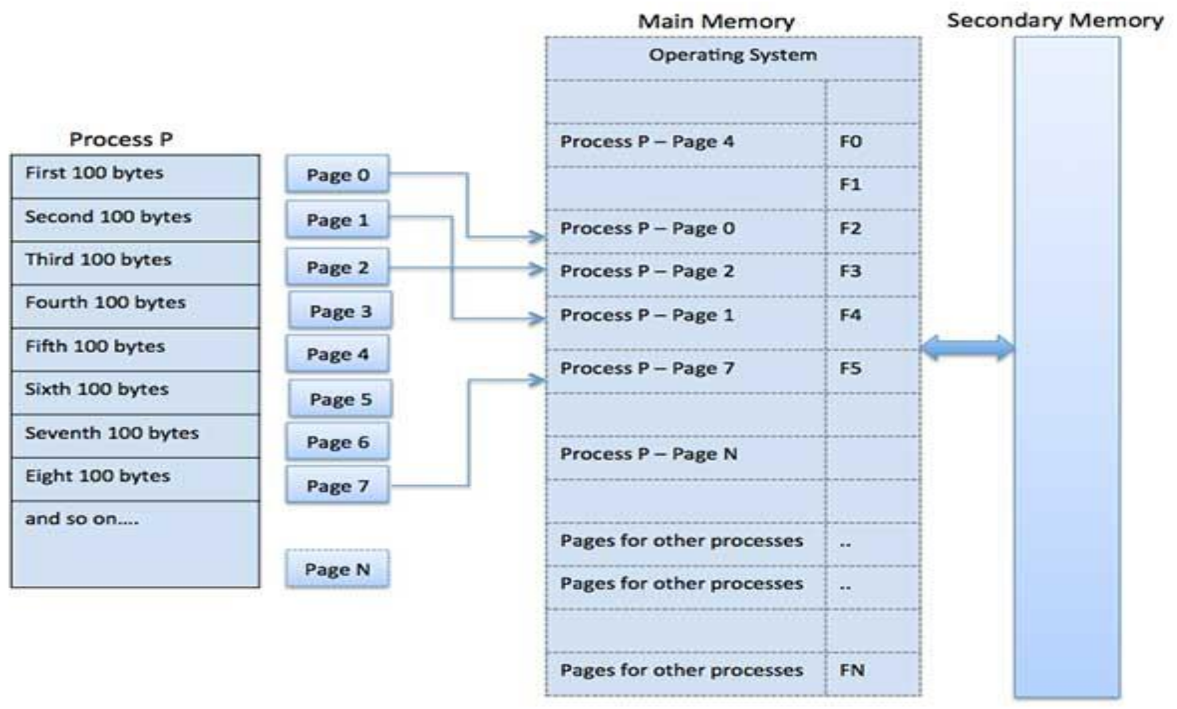


Fig 3.10. Paging Process

Address Translation

Page address is called **logical address** and represented by **page number**and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.

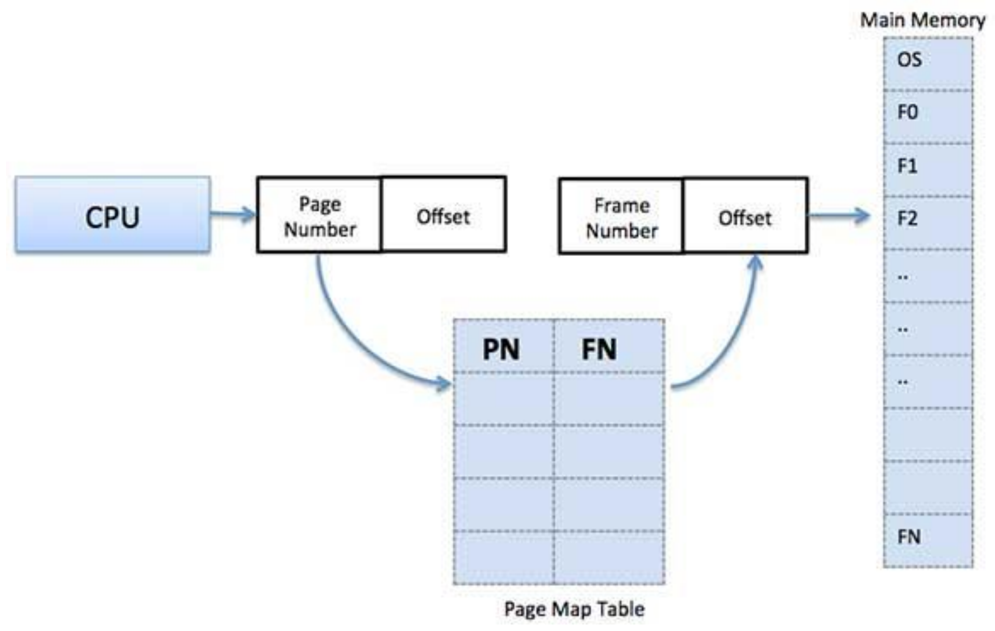


Fig 3.11. Paging with Page Table

When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.

- Page table requires extra memory space, so may not be good for a system having small RAM.

Paging

External fragmentation is avoided by using paging technique. Paging is a technique in which physical memory is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). When a process is to be executed, it's corresponding pages are loaded into any available memory frames.

Logical address space of a process can be non-contiguous and a process is allocated physical memory whenever the free memory frame is available. Operating system keeps track of all free frames. Operating system needs n free frames to run a program of size n pages.

Address generated by CPU is divided into

- **Page number (p)** -- page number is used as an index into a page table which contains base address of each page in physical memory.
- **Page offset (d)** -- page offset is combined with base address to define the physical memory address.

● Each virtual address space is divided into fixed-size chunks called pages.

● The physical address space is divided into fixed-size chunks called frames.

● Pages have same size as frames.

● The kernel maintains a page table (or page-frame table) for each process, specifying the frame within which each page is located.

● The CPU's memory management unit (MMU) translates virtual addresses to physical addresses on-the-fly for every memory access.

Properties:

● relatively simple to implement (in hardware);

● virtual address space need not be physically contiguous.

Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 264 bytes of physical memory.

The hardware support for paging is illustrated. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure.

Following figure show the paging table architecture.

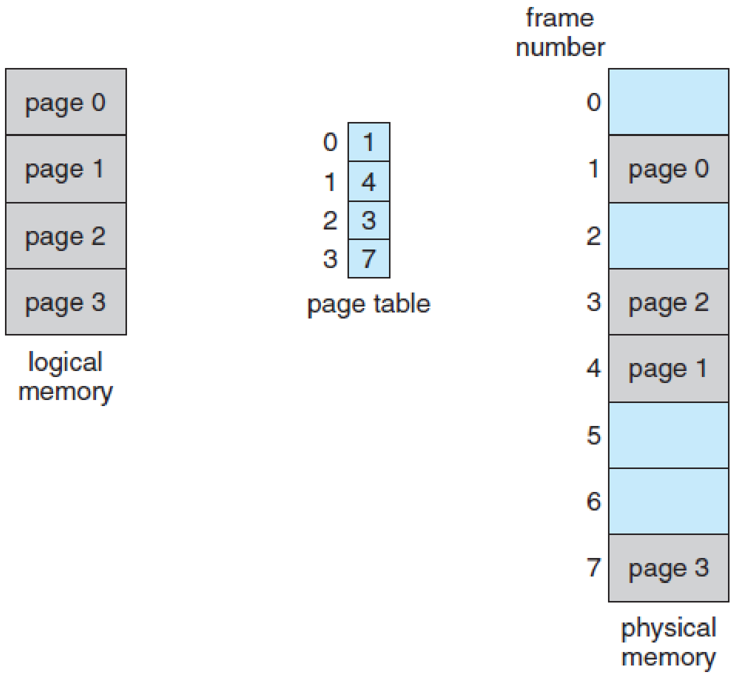
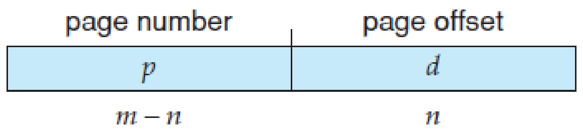


Fig 3.12. Page Table

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2*m*, and a page size is 2*n* bytes, then the high-order *m*− *n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. Thus, the logical address is as follows:



where *p* is an index into the page table and *d* is the displacement within the page.

Segmentation

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.

As we've already seen, the user's view of memory is not the same as the actual physical memory. This is equally true of the programmer's view of memory. Indeed, dealing with memory in terms of its physical properties is inconvenient to both the operating system and the programmer. What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory? The system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

Basic Method Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data? Most programmers would say "no." Rather, they prefer to view memory as a collection of variable-sized segments, with no necessary ordering among the segments. When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. Itmayalso include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy. She is not concerned with whether the stack is stored before or after the Sqrt() function. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt(), and so on.

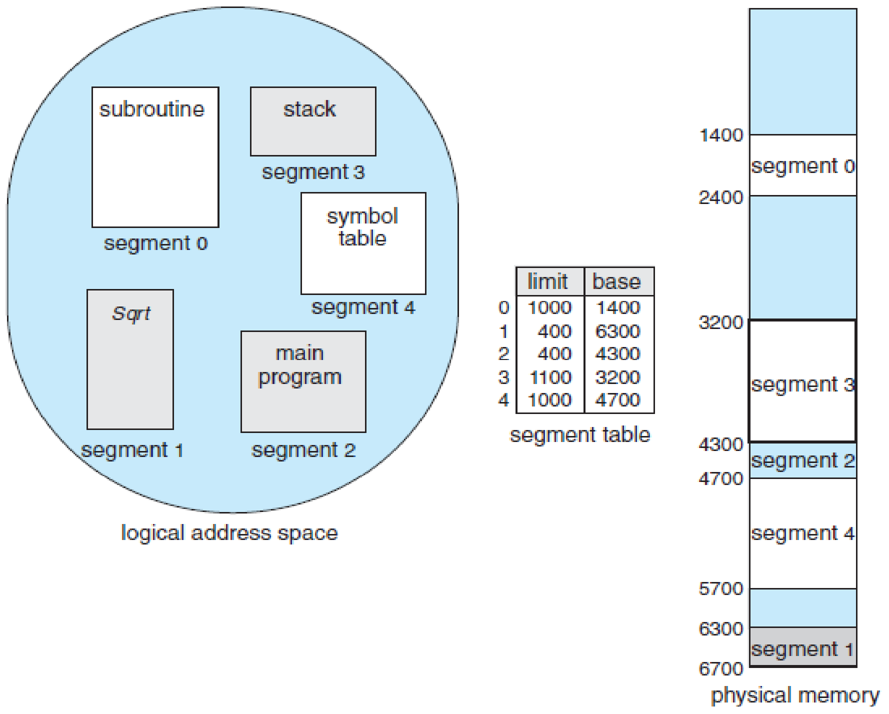


Fig 3.13. Segmentation

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:* <segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs

segments reflecting the input program.

A C compiler might create separate segments for the following:

1. The code

2. Global variables

3. The heap, from which memory is allocated

4. The stacks used by each thread

5. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

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.

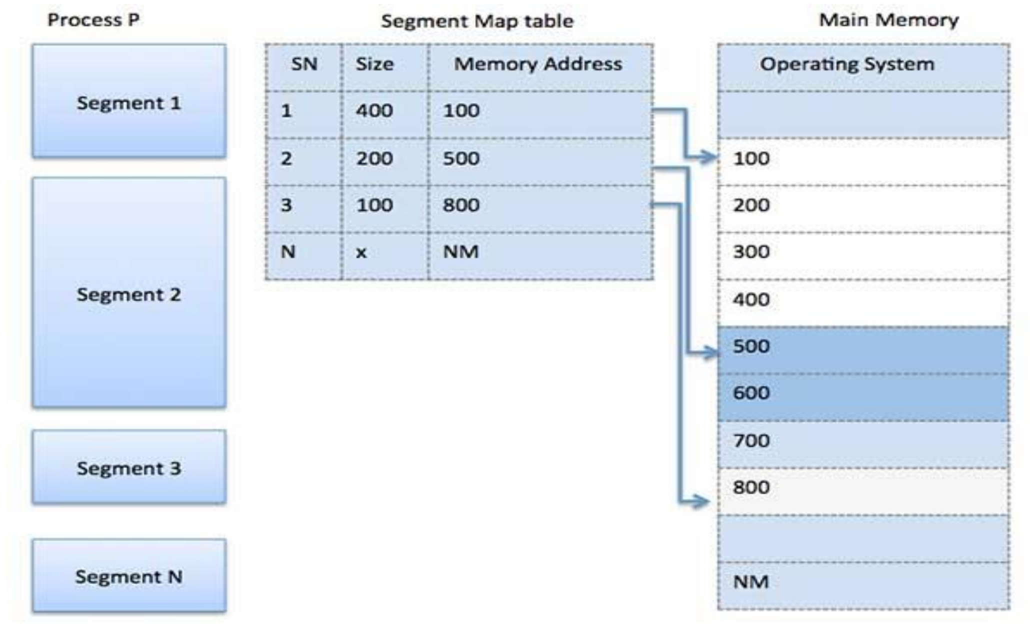


Fig 3.14. Segmentation Process

Virtual Memory

Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory. This technique is useful as large virtual memory is provided for user programs when a very small physical memory is there.

In real scenarios, most processes never need all their pages at once, for following reasons :

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
- Certain features of certain programs are rarely used.

Benefits of having Virtual Memory :

1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

Demand Paging

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them(On demand). This is termed as lazy swapper, although a pager is a more accurate term.

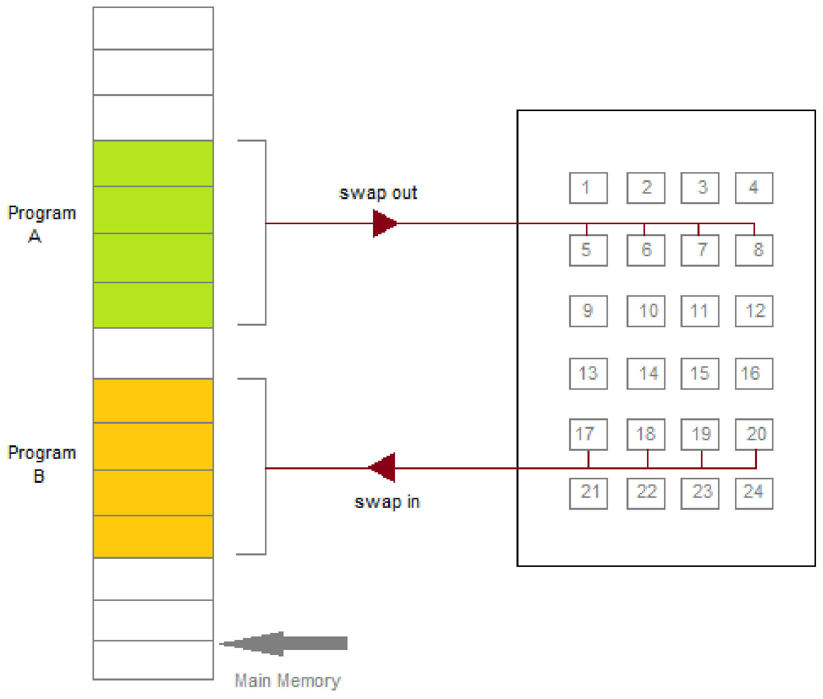


Fig 3.15. Demand Paging Process

Initially only those pages are loaded which will be required the process immediately.

The pages that are not moved into the memory, are marked as invalid in the page table. For an invalid entry the rest of the table is empty. In case of pages that are loaded in the memory, they are marked as valid along with the information about where to find the swapped out page.

When the process requires any of the page that is not loaded into the memory, a page fault trap is triggered and following steps are followed,

1. The memory address which is requested by the process is first checked, to verify the request made by the process.
2. If its found to be invalid, the process is terminated.
3. In case the request by the process is valid, a free frame is located, possibly from a free-frame list, where the required page will be moved.
4. A new operation is scheduled to move the necessary page from disk to the specified memory location. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to valid.
6. The instruction that caused the page fault must now be restarted from the beginning.

There are cases when no pages are loaded into the memory initially, pages are only loaded when demanded by the process by generating page faults. This is called **Pure Demand Paging**.

The only major issue with Demand Paging is, after a new page is loaded, the process starts execution from the beginning. Its is not a big issue for small programs, but for larger programs it affects performance drastically.

Page Replacement

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more number of processes into the memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem :

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called **Page replacement** and is most commonly used. We have some great algorithms to carry on page replacement efficiently.

Basic Page Replacement

- Find the location of the page requested by ongoing process on the disk.

- Find a free frame. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced, such frame is known as **victim frame**.
- Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
- Move the required page and store it in the frame. Adjust all related page and frame tables to indicate the change.
- Restart the process that was waiting for this page.

FIFO Page Replacement

- A very simple way of Page replacement is FIFO (First in First Out)
- As new pages are requested and are swapped in, they are added to tail of a queue and the page which is at the head becomes the victim.
- Its not an effective way of page replacement but can be used for small systems.

LRU Page Replacement

FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be ***used.*** If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**.

UNIT III

POSSIBLE QUESTIONS

(2 MARKS)

1. What is Paging?
2. What is a Physical address space?
3. What is Segmentation?
4. What is Virtual memory?
5. What is Fixed partition?

(6 MARKS)

1. Write about Memory Allocation strategies with neat sketch.
2. Explain the concept of Virtual memory in detail
3. Write about Physical and Virtual address space in detail.
4. Explain in detail about Segmentation with neat sketch.
5. Discuss about Paging in detail.
6. Write about Fixed and Variable partitions in detail.

KARPAGAM ACADEMY OF HIGHER EDUCA

DEPARTMENT OF COMPUTER SCIENCE, CA

II B.Sc CS (Batch 2016-2019)

OPERATING SYSTEMS

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE (

ONLINE EXAMINATIONS ONE MARKS Q

UNIT - 3

Sno	Questions	opt1	opt2	opt3
1	Memory is array of ___________	bytes	circuits	ics
2	CPU fetches instructions from ______	memory	pendrive	dvd
3	Program must be in ________________	memory	pendrive	dvd
4	Collection of process in disk forms_____	input queue	output queue	stack
5	Address space of computer starts at _________	0000	4444	3333
6	If process location is found during compile time then _________ code is generated	absolute	relative	approximate
7	Address generated by CPU is ______________ address	logical	physical	direct
8	Logical address can be also called as ______________ address	physical	virtual	direct
9	Run time mapping is done using _____________	MMU	CPU	CU
10	In address binding base register is also called as ______________	relocation register	memory register	hard disk
11	Better memory space is utilized using ________________	dynamic loading	dynamic linking	registers
12	_________________ routine is never loaded in dynamic loading	unused	used	regular
13	Some operating systems support only __________ linking	static	dynamic	temporary
14	______________ is a code that locates library routine	stub	dll	recursive routine
15	____________ can be used to manage large memory requirement for a process	overlays	swapping	roll in and out
16	______________ error is raised in memory	addressing	swapping	dynamic

17	Set of ____________ are scattered throughout the memory	holes	gaps	free space
18	______________ can be internal and external	fragmentation	merging	grouping
19	_____________ is used to divide a process into fixed size chunks	paging	segmentation	sp
20	In paging physical memory is divided into ________________	frames	pages	segments
21	In paging virtual memory is divided into ________________	frames	pages	segments
22	______________ is first of virtual address in paging	page number	segment number	frame number
23	______________ is second part of virtual address in paging	page number	segment number	frame number
24	Page mapping entries are found in ___________	page table	segment table	hash table
25	Page size is defined by _______________	hardware	software	os
26	______________ is first in mapping of virtual to physical address in paging	direct	associate	direct & associative
27	______________ is second in mapping of virtual to physical address in paging	direct	associate	direct & associative
28	______________ is third in mapping of virtual to physical address in paging	direct	associate	direct & associative
29	_____________ is used to divide a process into variable size chunks	paging	segmentation	sp
30	In segmentation virtual memory is divided into ________________	frames	pages	segments
31	____________ view is supported in segmentation	user	system	cpu
32	_____________ is format for segmentation virtual address	(s,d)	(p,d)	(v,d)
33	_____________ is the first element in segment table	limit	base	offset
34	_____________ is the second element in segment table	limit	base	offset
35	Addressing in segmentation is similar as ____________ addressing in paging	direct	associate	direct & associative
36	How many elements are there in segmentation address _________	1	2	3
37	_____________ is organization in physical memory in segmentation	frames	pages	segments

38	________________ memory is used to manage incompleteness of a process execution	virtual	physical	rom
39	virtual memory abstracts ________ memory	virtual	eerom	main
40	_______________ reasons are there for existence for virtual memory	1	2	3
41	_______________ benefits are there from virtual memory	1	2	3
42	Virtual memory is commonly implemented by ____________ paging	demand	bargain	quarrel
43	________________ fault occurs when desired page is not in memory	page	segment	pages
44	_______________ table is used in demand paging	page	segment	pages
45	_______________ methods are there for process creation	1	2	3
46	_______________ method implements partial sharing in process creation	copy on write	memory mapping	paging
47	_______________ is done for page fault	replacement	swapping	logging
48	_______________ is unrealizable page replacement algorithm	optimal	FIFO	LRU
49	_______________ is first page replacement algorithm	optimal	FIFO	LRU
50	_______________ is second page replacement algorithm	optimal	FIFO	LRU
51	_______________ is third page replacement algorithm	optimal	FIFO	LRU
52	_______________ is associated with each page in optimal algorithm	label	index	number
53	_______________ labelled page replaced in optimal algorithm	highest	lowest	moderate
54	_______________ end page is removed in fifo algorithm	rear	head	top
55	Modified version of fifo algorithm gives ____________ chance to a page	1	2	3
56	______________ is called as high paging activity	thrashing	smashing	mocking
57	______________ occurs frequently during thrashing	page fault	segment fault	memory fault
58	_______________ strategy is used to solve thrashing a little	working set	pff	lpr algorithm

59	_______________ algorithm is used to solve thrashing a little	working set	pff	lpr algorithm
60	_______________ is a basic solution for thrashing	working set	pff	lpr algorithm

TION

& IT

QUESTIONS

UESTIONS

opt4	opt5	opt6	answer
ram			**bytes**
cmos			**memory**
cmos			**memory**
circle			**input queue**
2222			**0000**
more or less			**absolute**
indirect			**logical**
indirect			**virtual**
IU			**MMU**
pendrive			**relocation register**
array of words			**dynamic loading**
recursive			**unused**
interruptive			**static**
exe file			**stub**
libraries			**overlays**
index			**addressing**

words			**holes**
fixing			**fragmentatio n**
swapping			**paging**
bytes			**frames**
bytes			**pages**
offset			**page number**
offset			**offset**
pointing table			**page table**
kernel			**hardware**
pointing			**direct**
pointing			**associate**
pointing			**direct & associative**
swapping			**segmentation**
bytes			**segments**
manager			**user**
(k,d)			**(s,d)**
page number			**limit**
page number			**base**
pointing			**direct**
4			**3**
bytes			**frames**

eprom			**virtual**
eprom			**main**
4			**3**
4			**3**
order			**demand**
segments			**page**
segments			**page**
4			**2**
segmentation			**copy on write**
locking			**replacement**
NRU			**optimal**
NRU			**FIFO**
NRU			**optimal**
NRU			**NRU**
identity			**label**
below average			**highest**
bottom			**head**
4			**2**
breaking			**thrashing**
address fault			**page fault**
gpl algorithm			**working set**

gpl algorithm			**lpr algorithm**
gpl algorithm			**pff**

UNIT-IV

Syllabus

File and I/O Management: Directory structure-File operations-File Allocation methods- Device management.

File and I/O Management

No general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer. Files are stored on random-access storage devices, including hard disks, optical disks, and solid-state (memory-based) disks. A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control. For example, a disk can be **partitioned** into quarters, and each quarter can hold a separate file system.

A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a **volume**. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory (more commonly known simply as the **directory**) records information—such as name, location, size, and type—for all files on that volume.

Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

• **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

• **Create a file**. New files need to be created and added to the directory.

• **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.

• **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

• **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

• **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

Directory structure

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

Single-Level Directory

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

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

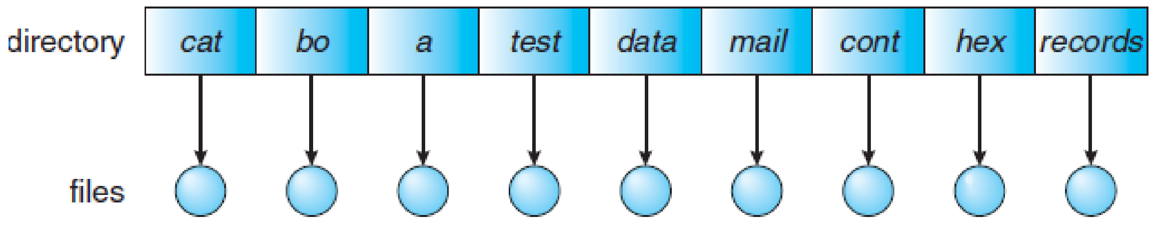


Fig 4.1. Single-level directory.

If two users call their data file test.txt, then the unique-name rule is violated.

For example, in one programming class, 23 students called the program for their second assignment prog2.c; another 11 called it assign2.c. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system.

Keeping track of so many files is a daunting task.

Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user file directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user.

When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user. When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

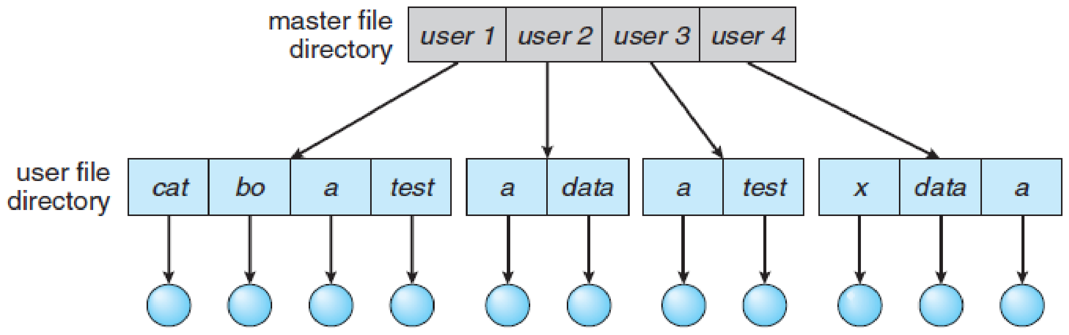


Fig 4.2. Two-level directory structure.

The user directories themselves must be created and deleted as necessary. **A special system program** is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. **Isolation** is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a **path name**. Every file in the system has a path name.

To name a file uniquely, a user must know the path name of the file desired. For example, if user A wishes to access her own test file named test.txt, she can simply refer to test.txt. To access the file named test.txt of user B (with directory-entry name userb), however, she might have to refer to /userb/test.txt. Every system has its own syntax for naming files in directories other than the user's own. Additional syntax is needed to specify the volume of a file. For instance, in Windows a volume is specified by a letter followed by a colon. Thus, a file specification might be C:\userb\test.

Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the **search path**.

The search path can be extended to contain an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and Windows. Systems can also be designed so that each user has his own search path.

Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly.

A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.

All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

Special system calls are used to create and delete directories.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.

To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants.

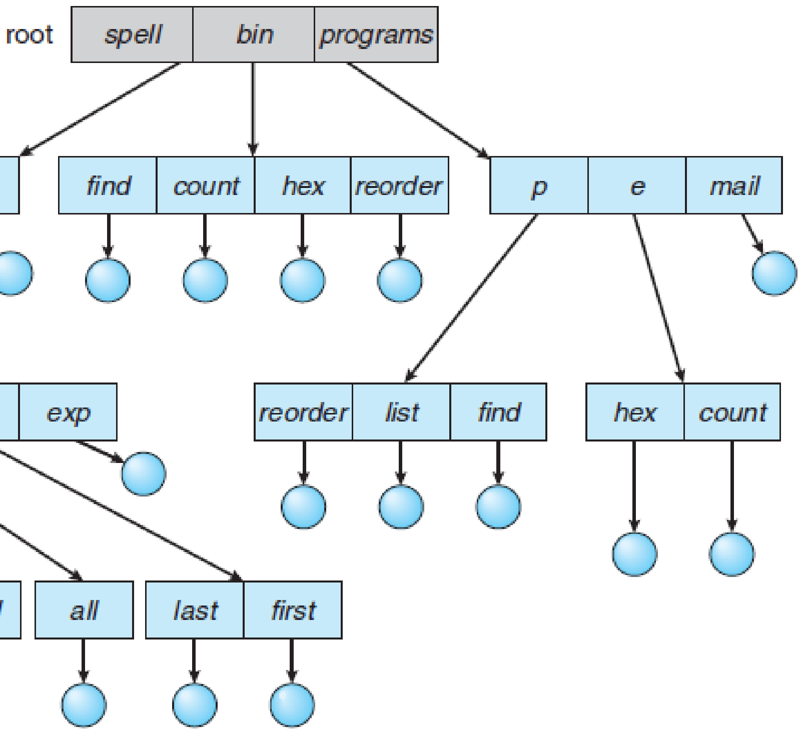


Fig 4.3. Tree-structured directory structure.

From **one change directory() system call to the next, all open() system calls search the current directory for the specified file.** Note that the search path may or may not contain a special entry that stands for "the current directory." The initial current directory of a user's login shell is designated when the user job starts or the user logs in.

The current directory of any subprocess is usually the current directory of the parent when it was generated. Path names can be of two types: absolute and relative.

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

For example, in the tree-structured file system, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first. Allowing a user to define her own subdirectories permits her to impose a structure on her files.

This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory programs may contain source programs; the directory bin may store all the binaries).

An interesting policy decision in a tree-structured directory concerns **how to handle the deletion of a directory**. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also.

This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX rm command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy.

Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be ***shared.*** A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories.

An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme. It is important to note that a shared file (or directory) is not the same as two copies of the file.

With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.

With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories. When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.

Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project. Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name.

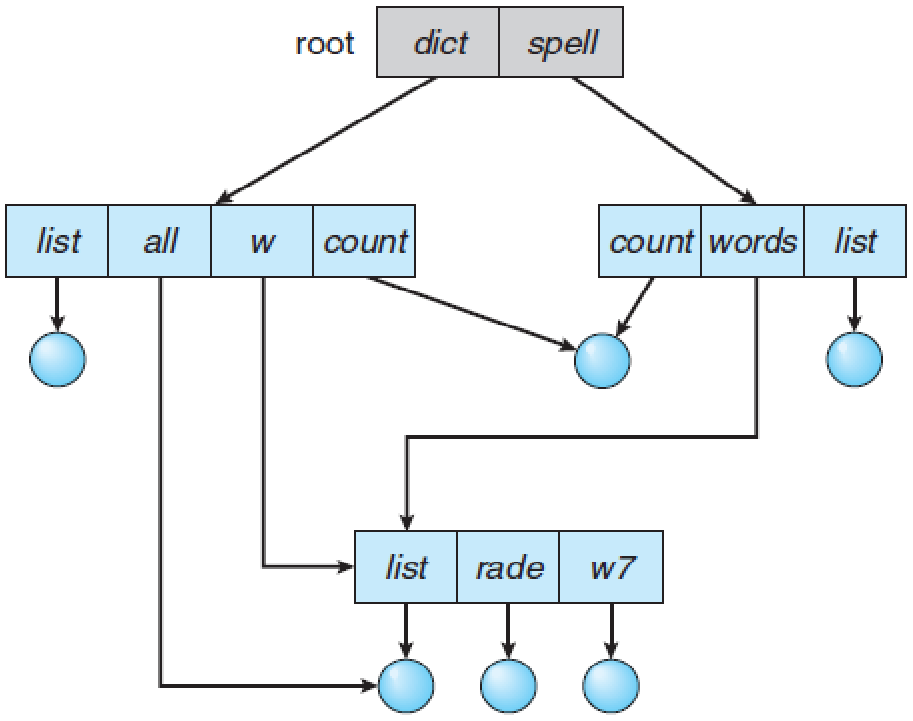


Fig 4.4. Acyclic-graph directory structure.

When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.

We **resolve** the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link. The link is clearly different from the original directory entry; thus, the two are not equal.

The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling.We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive.

General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure.

If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

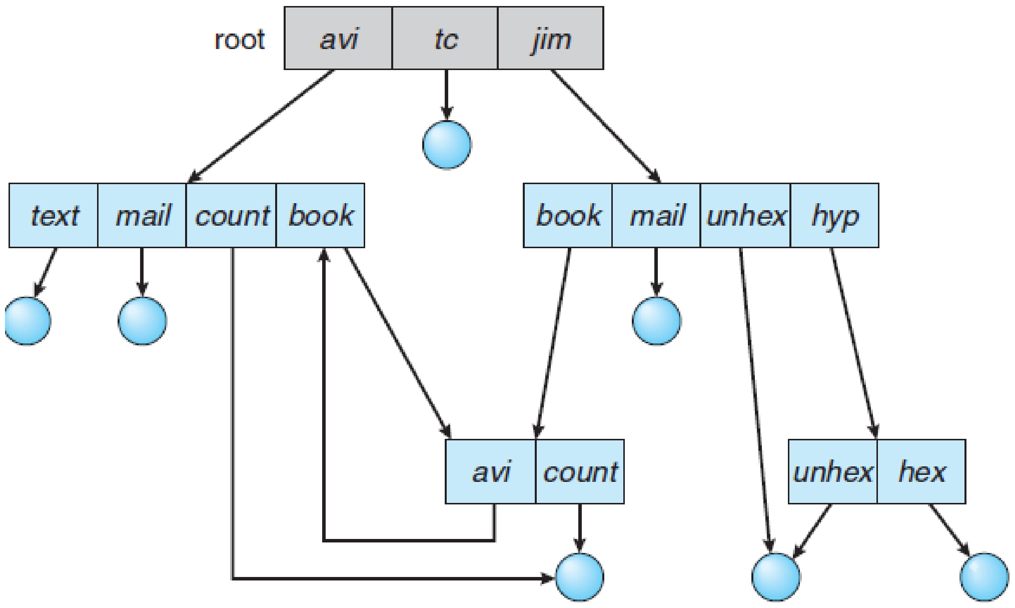


Fig 4.5. General graph directory.

A similar problem exists when we are trying to determine when a file can be deleted.

With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.

However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure.

In this case, we generally need to use a **garbage collection** scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed.

Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided, and no extra overhead is incurred.

File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

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

1. **Create.** The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open. Before using a file, a process must open it. The purpose of the **open** call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes.
A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet. 5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.
6. Write. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. Append. This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
8. Seek. For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.
10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.
11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

• **Creating a file**. Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
• **Writing a file**. To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
• **Reading a file**. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a **read pointer** to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **currentfile- position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
• **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
• **Deleting a file**. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
• **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

These six basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file.

These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file—or copy the file to another I/O device, such as a printer or a display—by creating a new file and then reading from the old and writing to the new. Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open() system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table. create() and delete() are system calls that work with closed rather than open files.

In summary, several pieces of information are associated with an open file.

• **File pointer**. On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read– write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
• **File-open count**. As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
• **Disk location of the file**.Most file operations require the systemtomodify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
• **Access rights**. Each process opens a file in an accessmode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests. Some operating systems provide facilities for locking an open file (or sections of a file). File locks provide functionality similar to reader–writer locks, covered in

A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not all operating systems provide both types of locks: some systems only provide exclusive file locking.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file.

File Allocation Methods

The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.

Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages.

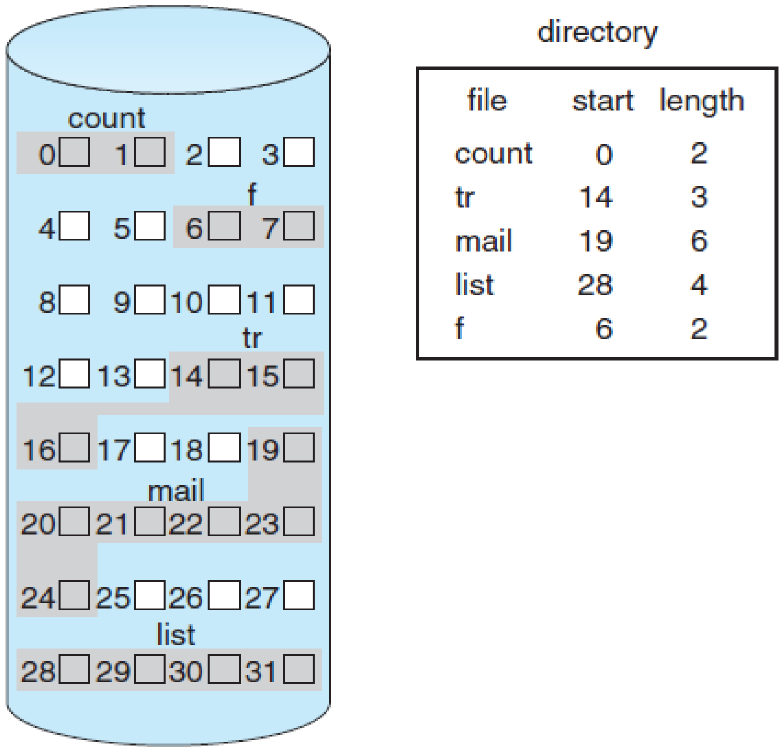
Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

Contiguous Allocation

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

When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed. Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is *n* blocks long and starts at location *b*, then it occupies blocks *b*, *b* + 1, *b* + 2, ..., *b* + *n* − 1.

The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 12.5). Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block *i* of a **554**



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Fig 4.6. Contiguous allocation of disk space

file that starts at block *b*, we can immediately access block *b* + *i*. Thus, both sequential and direct access can be supported by contiguous allocation. Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished;

Any management system can be used, but some are slower than others. The contiguous-allocation problem can be seen as a particular application of the general **dynamic storage-allocation** problem which involves how to satisfy a request of size *n* from a list of free holes.

First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster. All these algorithms suffer from the problem of **external fragmentation**.

As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.

Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem. One strategy for preventing loss of significant amounts of disk space to external fragmentation is to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space. We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem. The cost of this **compaction** is time, however, and the cost can be particularly high for large hard disks. Compacting these disks may take hours and may be necessary on a weekly basis.

Some systems require that this function be done **off-line**, with the file system unmounted. During this **down time**, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines. Most modern systems that need defragmentation can perform it **on-line** during normal system operations, but the performance penalty can be substantial.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example). In general, however, the size of an output file may be difficult to estimate. If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place.

Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming.

Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

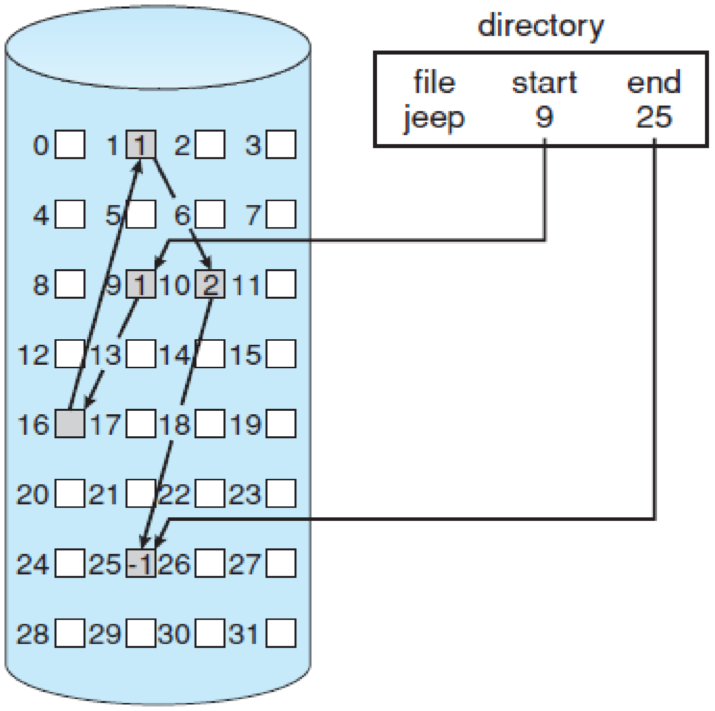


Fig 4.7. Linked allocation of disk space.

For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 . Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes. To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file.

This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.

To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space. Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files.

To find the *i*th block of a file, we must start at the beginning of that file and follow the pointers until we get to the *i*th block. Each access to a pointer requires a disk read, and some require a disk seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files. Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

Each file requires slightly more space than it would otherwise. The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple but improves disk throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management.

The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full. Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems. Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.

A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block. However, these schemes require even more overhead for each file. An important variation on linked allocation is the use of a **file-allocation table (FAT)**.

This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file.

This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 12.7 for a file consisting of disk blocks 217, 618, and 339. The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**. Each file has its own index block, which is an array of disk-block addresses. The *ith* entry in the index block points to the *ith* block of the file.

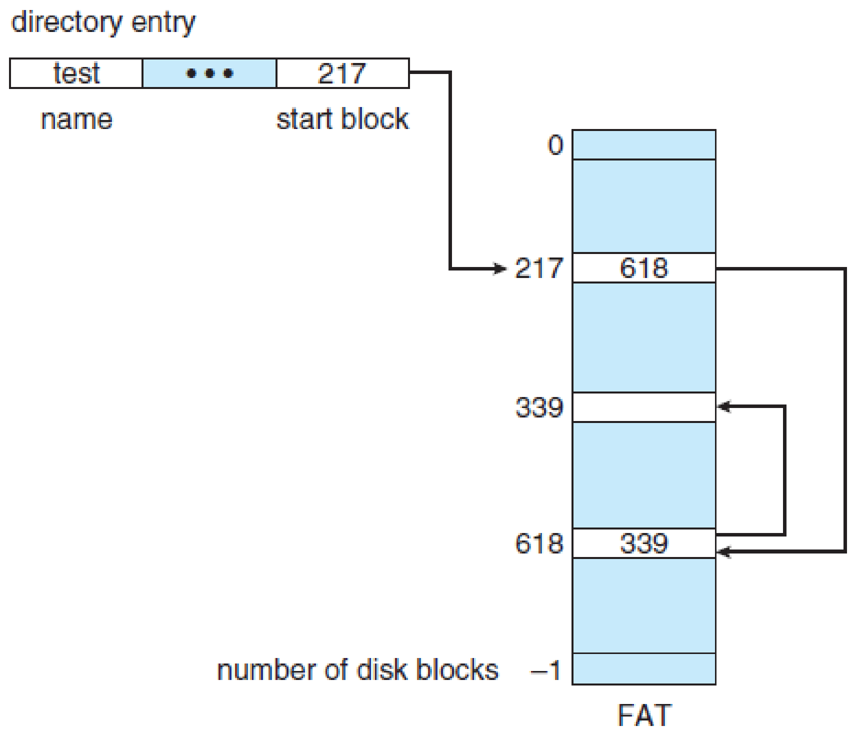


Fig 4.8. File-allocation table.

The *ith* entry in the index block points to the *ith* block of the file. The directory contains the address of the index block (Figure 12.8). To find and read the *ith* block, we use the pointer in the *ith* index-block entry. This scheme is similar to the paging scheme described in Section 8.5. When the file is created, all pointers in the index block are set to null. When the *ith* block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry. Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

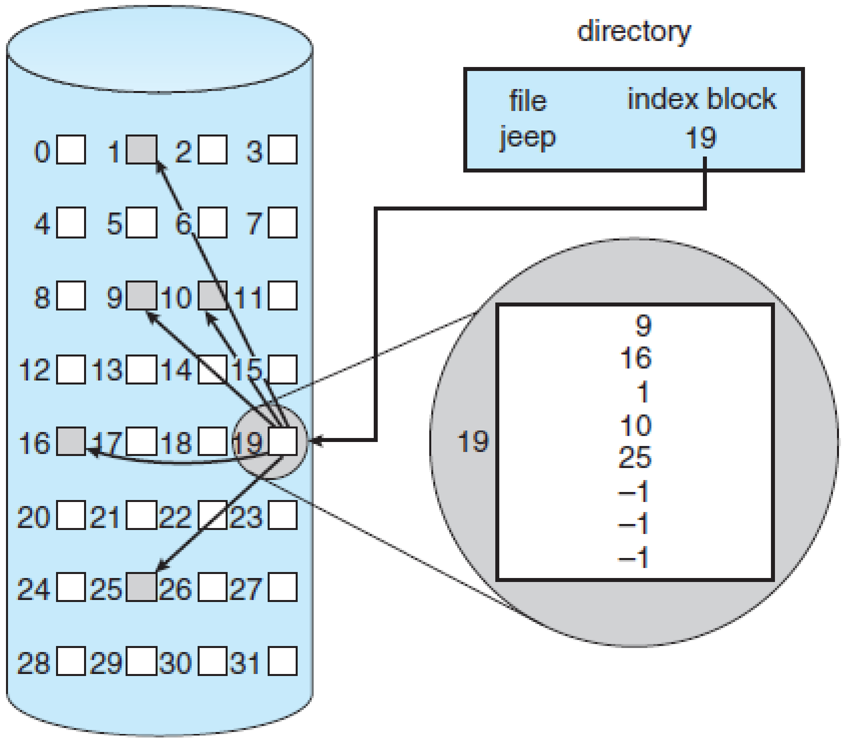


Fig 4.9. Indexed allocation of disk space.

Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue.

Space Allocation

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

Contiguous Allocation

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- External fragmentation is a major issue with this type of allocation technique.

Linked Allocation

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

Indexed Allocation

- Provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files.

To keep track of files, file systems normally have **directories** or **folders,** which in many systems are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).A system with multiple users may require us to first request() a device, to ensure exclusive use of it. After we are finished with the device, we release() it.

These functions are similar to the open() and close() system calls for files. Other operating systems allow unmanaged access to devices. The hazard then is the potential for device contention and perhaps deadlock.

Once the device has been requested (and allocated to us), we can read(), write(), and (possibly) reposition() the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on both files and devices.

Sometimes, I/O devices are identified by special file names, directory placement, or file attributes. The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

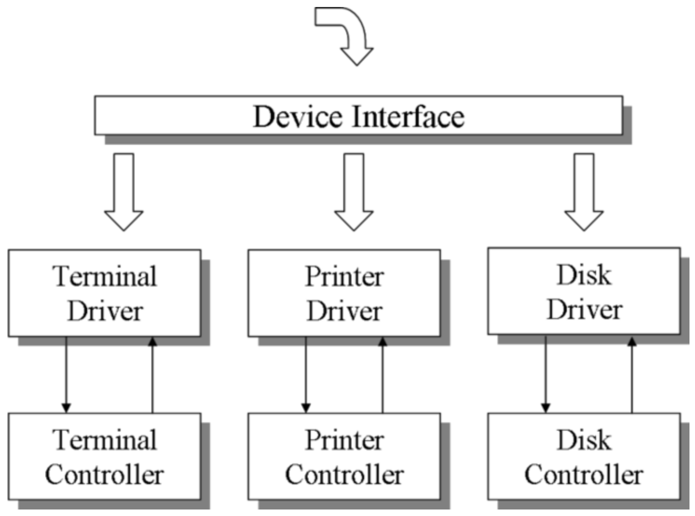
Hardware devices typically provide the ability to **input** data into the computer or **output** data from the computer. To simplify the ability to support a variety of hardware devices, standardized application programming interfaces (API) are used.

- Application programs use the System Call API to request one of a finite set of preset I/O requests from the Operating System.
- The Operating System uses algorithms for processing the request that are device independent.
- The Operating System uses another API to request data from the device driver.
- The device driver is third party software that knows how to interact with the specific device to perform the I/O.
- Sometimes we have a layering of device drivers where one device driver will call on another device driver to facilitate the I/O. An example of this is when devices are

connected to a USB port. The driver for the device will make use of the USB device driver to facilitate passing data to and from the device.

Device Drivers

The Device Driver Interface



System Call Interface

- Functions available to application programs
- Abstract all devices (and files) to a few interfaces
- Make interfaces as similar as possible
    - Block vs character
    - Sequential vs direct access
- Device driver implements functions (one entry point per API function)

. Example - UNIX Driver

open	Prepare dev for operation
close	No longer using the device
ioctl	Character dev specific info
read	Character dev input op

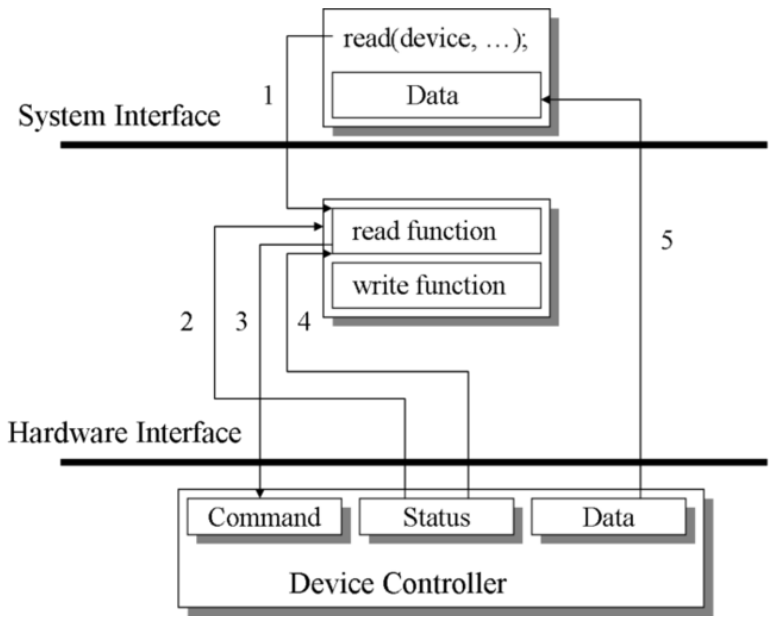
write	Character dev output op
strategy	Block dev input/output ops
select	Character dev check for data
stop	Discontinue a stream output op

. Waiting for I/O

Many types of input / output (I/O) do not occur immediately. So the process must wait in a waiting queue and the device driver needs a strategy of how to effectively wait for I/O data.

Polling

Polling is not very efficient because the system must continually check the device for data.



Interrupt Driven

When an interrupt occurs it calls the interrupt handlers.

It is more efficient to start an I/O peripheral doing a task and let it interrupt the system when the I/O operation is finished.

The interrupt driven approach causes drivers to consist of a top and bottom half.

Non maskable interrupt

These kind of interrupts are reserved for events like unrecoverable memory errors

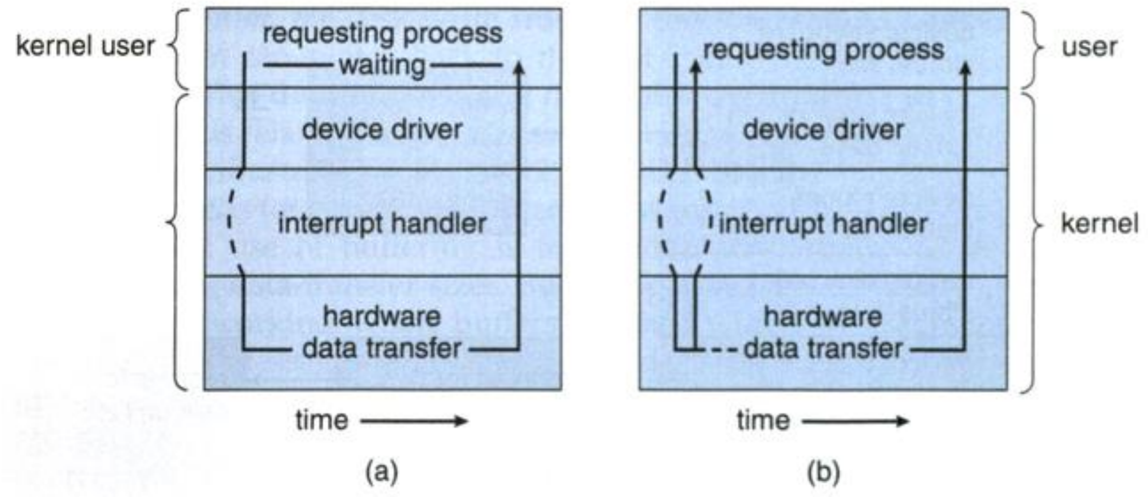
Maskable interrupt

Such interrupts can be switched off by the CPU before the execution of critical instructions that must not be interrupted.

Blocking and Nonblocking I/O

Some control over how the wait for I/O to complete is accommodated is available to the programmer of user applications. Most I/O requests are considered **blocking** requests, meaning that control does not return to the application until the I/O is complete. The delayed from systems calls such read() and write() can be quite long. Using systems call that block is sometimes call **synchronous** programming. In most cases, the wait is not really a problem because the program can not do anything else until the I/O is finished.

One solution for these situations is to use multiple threads so that one part of the program is not waiting for unrelated I/O to complete. Another alternative is to use **asynchronous** programming techniques with **nonblocking** system calls. An asynchronous call returns immediately, without waiting for the I/O to complete.



Blocking I/O system calls (a) do not return until the I/O is complete. Nonblocking I/O systems calls return immediately. The process is later notified when the I/O is complete.

A good example of nonblocking behavior is the select() system call for network sockets. Using select(), an application can monitor several resources at the same time and can also poll for network activity without blocking. The select() system call identifies if data is pending or not, then read() or write()may be used knowing that they will complete immediately.

UNIT IV
POSSIBLE QUESTIONS
(2 MARKS)

1. List any four File Operations?
2. What are the file allocation methods?
3. What is linked allocation?
4. What are the directory structures available?
5. What is device management?

(6 MARKS)

1. Explain File operations with example.
2. Briefly describe about Device management.
3. Describe about Directory structure with neat diagram.
4. Write about File Allocation methods.

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

KARPAGAM ACADEMY OF HIGHER EDUCAT

DEPARTMENT OF COMPUTER SCIENCE, CA

II B.Sc CS (Batch 2016-2019)

OPERATING SYSTEMS

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE Q

ONLINE EXAMINATIONS ONE MARKS QU

UNIT - 4

SNO	QUESTIONS	opt1	opt2	opt3
1	The file system consist of -------------- Distinct parts	2	3	4
2	A --------------- File is a sequence of character organized into lines	Source	Object	Text
3	A --------------- File is a sequence of subroutines and functions	Source	Object	Text
4	The operating system keeps a small table called the --------------- ,containing information about all open files	Show file table	Visible file table	Open file ta
5	A file is executed in --------------- extension	External structure	.bat	.mdb
6	The .bat file is a ----------------containing in ANSI format,command to the operating system	Binary file	Batch file	Text file
7	The file type is used to indicate the ---------------- of the file	.txt	Internal structure	Block structure
8	Information in the file is processed in the order called------------------	Direct access	Sequence access	Dynamic access
9	A file is made up of fixed length that allows the program to read and write record rapidly in no particular order is called -------------------	Direct access	Sequence access	Dyanamic access
10	Data cannot be written in secondary storage unless written with in a --------------------	File	Swap space	Directory
11	File attribute consist of ----------------	Name,Type,Content	Name,type,Size	Seperate directory system
12	The information about all files is kept in -----------------	swap space	operating system	Name,Size,Type,identifier

13	A file is a ------------- type	Abstract	Primitive	Public
14	In UNIX Open system call returns -----------------	pointer to the entry in the open file table	pointer to the entry in the system wide table	A file to the process calling it
15	The open file table has a ------------------ Associated with each file	File content	File permission	open count
16	The file name is generaly split into which of the two parts ----------------	Name and type	Name and identifier	Name and extension
17	In the sequential access method, information in the file is processed	One disk after the other	One record after the other	One text document after the other
18	Sequential access method --------------- ,on random access devices	Works well	Dosen’t works well	Works slow
19	The direct access method is based on a ------------- model of a file as--------------- allow random access to any file block	Magnetic tape,magnetic tapes	Tape,Tapes	Disk,Disks
20	A relative block number is an index relative to --------------------	The beginning of the file	The end of the file	The last written position in file
21	The index contains -----------------------	Name of all content of file	Pointer to each page	Pointers to the various blocks
22	The directory can be viewed as a --------------,that translate the file name into their directory entries	Symbol table	Partition	Swap space
23	In the single level directory: -----------------------	All files are contain in different directories	All files are contained in the same directory	Depend on the operating system
24	In the single level directory -------------	All directory must have a unique name	All files must have a unique name	All files must have a unique owner

25	In the two level directory structure -----------------	Each user has its own user file directory	The system has its own master file directory	)Both a and b
26	When a user refers to a particular file -------------------	System MFD is searched	His own UFD is searched	Both MFD and UFD are searched
27	The disadvantage of the two level directory structure is that	It does not solve the name collision problem	It solve the name collision problem	It does not isolate users from one another
28	In the tree structure directory ------------------	The tree has the same directory	The tree has the leaf directory	The tree has the root directory
29	The three major methods of allocating disk space that are in wide use are ---------------	Contiguous,Linked,Hashed	Contiguous,Linked,Indexed	Linked,Hashed,Indexed
30	In Contiguous allocation ----------------	each file must occupy a set of contiguous block on the disk	Each file is a linked list of disk blocks	All the pointers to scattered
31	In linked allocation -----------------	Each file must occupy a set of contiguous block on the disk	Each file is a linked list of disk blocks	All the pointers to scattered
32	In indexed allocation ------------	Each file must occupy a set of contiguous block on the disk	Each file is a linked list of disk blocks	All the pointers to scattered blocks are placed together in one location

33	One system where there are multiple operating system, the decision to load a particular one is done by ----------------	Boot loader	Boot strap	Process control block
34	The VFS refers to -----------	Virtual File System	Valid File System	Virtual Font System
35	The disadvantage of a linear list of directory entries is the --------------------	Size of the linear list in the memory	Linear search to find a file	It is not reliable
36	One difficulty of contiguous allocation is ---------------	Finding space for a new file	Ineffecient	Costly
37	To solve the problem of external fragmentation ---------------- needs to be done periodically	Compaction	Check	Formatting
38	If too little space is allocated to a file ----------------	The file will not work	There will not be any space	The file cannot be extended
39	A system program such as fsck ----------------- is a consistency checker	UNIX	Windows	Macintosh
40	Each set of operations for performing a specific task is a ---------------------	Program	Code	Transaction
41	Once the changes are written to the log, they are considered to be --------------	Committed	Aborted	Completed
42	When an entire command transaction is completed,-------------------	It is stored in the memory	It is removed from the log file	It is redone
43	A circular buffer is ----------------	Write to the end of its space	Overwrite older value as it goes	both A and B
44	In -------------- information is recorded magnetically on platters	Magnetic disk	Electrical disk	Assemblies
45	The head of the magnetic disk are attached to a -------------- that moves all the head as unit	Spindle	Disk arm	Track
46	The set of tracks that are at one arm position make up a -----------	Magnetic disk	Electrical disk	Assemblies
47	The time taken to move a disk arm to the desired cylinder is called as--------------	Positioning time	Random access ti	Seek time

48	When a head damages the magnetic surface, it is known as ---------------------	Disk crash	Head crash	Magnetic damage
49	A flopy disk is designed to rotate -------------- as compared to a hard disk drive	Faster	Slower	At the same speed
50	The host controller is -------------------	Controller built at the end of each disk	Controller at the computer end of the bus	Both a and b
51	The process of dividing a disk into sectors that the disk controller can read and write, before a disk can store data is known as-----------------	Partitioning	Swap space creation	Low-level formatting
52	the data structure for a sector typically contains -------------------	Header	Data area	Trailer
53	The header and trailer of a sector contains information used by the disk controller such as ______________.	Main section	Error corecting codes	Sector number
54	The two steps that the operating system takes to use a disk to hold its files are ---------------- and -------------	partitioning	Swap space creation	Catching
55	The ------------- program initializes all aspects of the system, from CPU registers to device controllers and the content of main memory, and then starts the operating system	Main	Boot loader	Boot strap
56	For most computers the boot strap is stored in--------------------	RAM	ROM	Cache
57	A disk that has a boot partition is called a --------------	Start disk	Destroyed blocks	Boot disk
58	Defective sectors on disks are often known as -----------------	Good blocks	System disk	Bad blocks
59	Bad blocks are called as ___________	Good Sectors	Defective Sectors	boot disks
60	ROM got _________ file	boot strap	Data area	head data

ION

& IT

UESTIONS

JESTIONS

opt4	opt5	opt6	ANSWER
5			**2**
Executable			**Text**
Executable			**Source**
Manage file table			**Open file table**
.in			**.bat**
Word file			**Batch file**
Outer structure			**Internal structure**
Random access			**Sequence access**
Random access			**Direct access**
Text format			**File**
Name,identifie r			**Name,Size ,Type,ide ntifier**
Hard disk			**Seperate directory system**

Private			**Abstract**
pointer to the entry in the close file			**pointer to the entry in the open file table**
Close count			**open count**
Extension and type			**Name and extension**
One name after the other			**One record after the other**
Works normal			**Works well**
Tape,Disk			**Disk,Disks**
Middle of the file			**The beginning of the file**
Pointer to same page			**Pointers to the various blocks**
Cache			**Symbol table**
Depend on the file name			**All files are contained in the same directory**
All files must have a different names			**All files must have a unique name**

Each user has its different file directory			**Both a and b**
Every directory is searched			**Both MFD and UFD are searched**
It isolates users from one another			**It isolates users from one another**
The tree has no directory			**The tree has the root directory**
Contiguous ,Linked			**Contiguous,Linked ,Indexed**
All the files are blocked			**each file must occupy a set of contiguous block on the disk**
All the files are blocked			**Each file is a linked list of disk blocks**
All the files are blocked			**All the pointers to scattered blocks are placed together in one location**

File control block			**Boot loader**
Virtual Function System			**Virtual File System**
It is not valid			**Linear search to find a file**
Time taking			**Finding space for a new file**
Replacing memory			**Compaction**
file cannot be opened			**The file cannot be extended**
Solaris			**UNIX**
Method			**Transaction**
Finished			**Committed**
It is deleted from the memory			**It is removed from the log file**
overwrite new values			**both A and B**
Cylinders			**Magnetic disk**
Pointer			**Disk arm**
Cylinders			**Cylinders**
Rotational latency			**Seek time**

All of these			**Head crash**
Normal speed			**Slower**
Controller at the system side			**Controller at the computer end of the bus**
Physical formatting			**Low-level formatting ,Physical formatting**
Main section			**Header ,Data area ,Trailer**
Disk identifier			**Sector number**
Logical formatting			**partitioning**
ROM			**Boot strap**
Tertiary storage			**ROM**
Format disk			**System disk,boot disk**
Semi blocks			**Bad blocks**
boot strap			**Defective Sectors**
random data			**boot strap**

UNIT V

Syllabus

Protection and Security: Policy mechanism-Authentication-Internal access Authorization.

Protection and Security

Policy Mechanism

The policies what is to be done while the mechanism specifies how it is to be done. For instance, the timer construct for ensuring CPU protection is mechanism. On the other hand, the decision of how long the timer is set for a particular user is a policy decision.

The separation of mechanism and policy is important to provide flexibility to a system. If the interface between mechanism and policy is well defined, the change of policy may affect only a few parameters. On the other hand, if interface between these two is vague or not well defined, it might involve much deeper change to the system.

Once the policy has been decided it gives the programmer the choice of using his/her own implementation. Also, the underlying implementation may be changed for a more efficient one without much trouble if the mechanism and policy are well defined. Specifically, separating these two provides flexibility in a variety of ways. First, the same mechanism can be used to implement a variety of policies, so changing the policy might not require the development of a new mechanism, but just a change in parameters for that mechanism, but just a change in parameters for that mechanism from a library of mechanisms. Second, the mechanism can be changed for example, to increase its efficiency or to move to a new platform, without changing the overall policy.

Policy vs mechanism OS examples

- Granting a resource to a process using first come first serve algorithm (policy). This policy can be implemented using a queue (mechanism).
- Thread scheduling or answering the question "which thread should be given the chance to run next?" is a policy. For example, is it priority based ? or just round robin ?. Implementing context switching is the corresponding mechanism.
- In virtual memory, keeping track of free and occupied pages in memory is a mechanism. Deciding what to do when a page fault occurs is a policy. You may check the following articles cpu scheduling, paging vs segmentation and page tables

Separation of mechanism and policy

Separation of policy and mechanism is a design principle to achieve flexibility. In other words, adopting a certain mechanism should not restrict existing policies. The idea behind this concept is to have the least amount of implementation changes if we decide to change the way a particular feature is used. We can also look at it from the other side. For example, if a certain implementation needs to be changed (ex. improve efficiency). This must not greatly influence the way it is used. In the login example mentioned earlier (logging to a website) switching from a user name password pair to Facebook account should not prevent a user from logging in to the website.

Summary

- Policy is the what and mechanism is the how.
- The separation between the two gives us the flexibility to add and modify existing policies and reuse existing mechanisms for implementing new policies.

Mechanism versus Policy

- Another principle that helps architectural coherence, along with keeping things small and well structured, is that of separating mechanism from policy. By putting the mechanism in the operating system and leaving the policy to user processes, the system itself can be left unmodified, even if there is a need to change policy. Even if the policy module has to be kept in the kernel, it should be isolated from the mechanism, if possible, so that changes in the policy module do not affect the mechanism module.

- To make the split between policy and mechanism clearer, let us consider two real-world examples. As a first example, consider a large company that has a payroll department, which is in charge of paying the employees' salaries. It has computers, software, blank checks, agreements with banks, and more mechanism for actually paying out the salaries. However, the policy—determining who gets paid how much—is completely separate and is decided by management. The payroll department just does what it is told to do.

- As the second example, consider a restaurant. It has the mechanism for serving diners, including tables, plates, waiters, a kitchen full of equipment, agreements with credit card companies, and so on. The policy is set by the chef, namely, what is on the menu. If the chef decides that tofu is out and big steaks are in, this new policy can be handled by the existing mechanism.

- Now let us consider some operating system examples. First, consider thread scheduling. The kernel could have a priority scheduler, with *k* priority levels. The mechanism is an array, indexed by priority level, as shown in Fig. 10-11 or Fig. 11-19. Each entry is the head of a list of ready threads at that priority level. The scheduler just searches the array from highest priority to lowest priority, selecting the first threads it hits. The policy is setting the priorities. The system may have different classes of users, each with a different priority, for example. It might also allow user processes to set the relative priority of its threads. Priorities might be increased after completing I/O or decreased after using up a quantum. There are numerous other policies that could be followed, but the idea here is the separation between setting policy and carrying it out.

- A second example is paging. The mechanism involves MMU management, keeping lists of occupied pages and free pages, and code for shuttling pages to and from disk. The policy is deciding what to do when a page fault occurs. It could be local or global, LRU-based or FIFO-based, or something else, but this algorithm can (and should) be completely separate from the mechanics of actually managing the pages.

- A third example is allowing modules to be loaded into the kernel. The mechanism concerns how they are inserted, how they are linked, what calls they can make, and what calls can be made on them. The policy is determining who is allowed to load a module into the kernel and which modules. Maybe only the superuser can load modules, but maybe any user can load a module that has been digitally signed by the appropriate authority.

Policy and Mechanism

Critical to our study of security is the distinction between policy and mechanism.

- **Definition 1–1**. A *security policy* is a statement of what is, and what is not, allowed.

- **Definition 1–2**. A *security mechanism* is a method, tool, or procedure for enforcing a security policy.

Mechanisms can be nontechnical, such as requiring proof of identity before changing a password; in fact, policies often require some procedural mechanisms that technology cannot enforce.

As an example, suppose a university's computer science laboratory has a policy that prohibits any student from copying another student's homework files. The computer system provides mechanisms for preventing others from reading a user's files. Anna fails to use these mechanisms to protect her homework files, and Bill copies them. A breach of security has occurred, because Bill has violated the security policy. Anna's failure to protect her files does not authorize Bill to copy them.

In this example, Anna could easily have protected her files. In other environments, such protection may not be easy. For example, the Internet provides only the most rudimentary security mechanisms, which are not adequate to protect information sent over that network. Nevertheless, acts such as the recording of passwords and other sensitive information violate an implicit security policy of most sites (specifically, that passwords are a user's confidential property and cannot be recorded by anyone).

Policies may be presented mathematically, as a list of allowed (secure) and disallowed (nonsecure) states. For our purposes, we will assume that any given policy provides an axiomatic description of secure states and nonsecure states. In practice, policies are rarely so precise; they normally describe in English what users and staff are allowed to do. The ambiguity inherent in such a description leads to states that are not classified as "allowed" or "disallowed." For example, consider the homework policy discussed above. If someone looks through another user's directory without copying homework files, is that a violation of security? The answer depends on site custom, rules, regulations, and laws, all of which are outside our focus and may change over time.

When two different sites communicate or cooperate, the entity they compose has a security policy based on the security policies of the two entities. If those policies are inconsistent, either or both sites must decide what the security policy for the combined site should be. The inconsistency often manifests itself as a security breach. For example, if proprietary documents were given to a university, the policy of confidentiality in the corporation would conflict with the more open policies of most universities. The university and the company must develop a mutual security policy that meets both their needs in order to produce a consistent policy. When the two sites communicate through an independent third party, such as an Internet service provider, the complexity of the situation grows rapidly.

Authentication

Every *secured* computer system must require all users to be authenticated at login time. After all, if the operating system cannot be sure who the user is, it cannot know which files and other resources he can access. While authentication may sound like a trivial topic, it is a bit more complicated than you might expect.

Early minicomputers (e.g., PDP-1 and PDP-8) did not have a login procedure, but with the spread of UNDC on the PDP-11 minicomputer, logging in was again needed. Early personal computers (e.g., Apple II and the original IBM PC) did not have a login procedure, but more sophisticated personal computer operating systems, such as Linux and Windows Vista, do (although foolish users can disable it). Machines on corporate LANs almost always have a login procedure configured so that users cannot bypass it. Finally, many people nowadays (indirectly) log into remote computers to do Internet banking, e-shopping, download music, and other commercial activities. All of these things require authenticated login, so user authentication is once again an important topic.

Having determined that authentication is often important, the next step is to
find a good way to achieve it. Most methods of authenticating users when they attempt
to log in are based on one of three general principles, namely identifying
1. Something the user knows.
2. Something the user has.
3. Something the user is.

Sometimes two of these are required for additional security. These principles lead to different authentication schemes with different complexities and security properties. In the following sections we will examine each of these in turn. People who want to cause trouble on a particular system have to first log in to that system, which means getting past whichever authentication procedure is used. In the popular press, these people are called **hackers.** In deference to true hackers, we will use the term in the original sense and will call people who try to break into computer systems where they do not belong **crackers.**

Authentication Using Passwords
The most widely used form of authentication is to require the user to type a login name and a password. Password protection is easy to understand and easy to implement.

The simplest implementation just keeps a central list of (login-name, password) pairs. The login name typed in is looked up in the list and the typed password is compared to the stored password. If they match, the login is allowed; if they do not match, the login is rejected.

It goes almost without saying that while a password is being typed in, the computer should not display the typed characters, to keep them from prying eyes near the monitor. With Windows, as each character is typed, an asterisk is displayed. With UNIX, nothing at all is displayed while the password is being typed. These schemes have different properties. The Windows scheme may make it easy for absent-minded users to see how many characters they have typed so far, but it also discloses the password length to "eavesdroppers" (for some reason, English has a word for auditory snoopers but not for visual snoopers, other than perhaps Peeping Tom, which does not seem right in this context). From a security perspective, silence is golden.

How Crackers Break In
Most crackers break in by connecting to the target computer (e.g., over the Internet) and trying many (login name, password) combinations until they find one that works. Many people use their name in one form or another as their login name. Of course, guessing the login name is not

enough. The password has to be guessed, too. How hard is that? Easier than you might think. The classic work on password security was done by Morris and Thompson (1979) on UNIX systems.

They compiled a list of likely passwords: first and last names, street names, city names, words from a moderate-sized dictionary (also words spelled backward), license plate numbers, and short strings of random characters. They then compared their list to the system password file to see if there were any matches. Over 86% of all passwords turned up in their list.

UNIX Password Security

Some (older) operating systems keep the password file on the disk in unencrypted form, but protected by the usual system protection mechanisms. Having all the passwords in a disk file in unencrypted form is just looking for trouble because all too often many people have access to it. These may include system administrators, machine operators, maintenance personnel, programmers, management, and maybe even some secretaries.

A better solution, used in UNIX, works like this. The login program asks the user to type his name and password. The password is immediately "encrypted" by using it as a key to encrypt a fixed block of data. Effectively, a one-way function is being run, with the password as input and a function of the password as output. This process is not really encryption, but it is easier to speak of it as encryption. The login program then reads the password file, which is just a series of ASCII lines, one per user, until it finds the line containing the user's login name.

If the (encrypted) password contained in this line matches the encrypted password just computed, the login is permitted, otherwise it is refused. The advantage of this scheme is that no one, not even the superuser, can look up any users' passwords because they are not stored in unencrypted form anywhere in the system.

One-Time Passwords

Most superusers exhort their mortal users to change their passwords once a month. It falls on deaf ears. Even more extreme is changing the password with every login, leading to one-time passwords. When one-time passwords are used, the user gets a book containing a list of passwords. Each login uses the next password in the list. If an intruder ever discovers a password, it will not do him any good, since next time a different password must be used. It is suggested that the user try to avoid losing the password book.

Authentication Using a Physical Object

The second method for authenticating users is to check for some physical object they have rather than something they know. Metal door keys have been used for centuries for this purpose. Nowadays, the physical object used is often a plastic card that is inserted into a reader associated with the computer. Normally, the user must not only insert the card, but must also type in a password, to prevent someone from using a lost or stolen card. Viewed this way, using a bank's ATM (Automated Teller Machine) starts out with the user logging in to the bank's computer via a remote terminal (the ATM machine) using a plastic card and a password (currently a 4-digit PIN code in most countries, but this is just to avoid the expense of putting a full keyboard on the ATM machine).

Information-bearing plastic cards come in two varieties: magnetic stripe cards and chip cards. Magnetic stripe cards hold about 140 bytes of information written on a piece of magnetic tape glued to the back of the card. This information can be read out by the terminal and sent to the central computer. Often the information contains the user's password (e.g., PIN code) so the terminal can do an identity check even if the link to the main computer is down. Typically the password is encrypted by a key known only to the bank.

Authentication Using Biometrics
The third authentication method measures physical characteristics of the user that are hard to forge. These are called biometrics (Pankanti et al., 2000). For example, a fingerprint or voiceprint reader hooked up to the computer could verify the user's identity.

A typical biometrics system has two parts: enrollment and identification. During enrollment, the user's characteristics are measured and the results digitized. Then significant features are extracted and stored in a record associated with the user. The record can be kept in a central database (e.g., for logging in to a remote computer), or stored on a smart card that the user carries around and inserts into a remote reader (e.g., at an ATM machine).

The other part is identification. The user shows up and provides a login name. Then the system makes the measurement again. If the new values match the ones sampled at enrollment time, the login is accepted; otherwise it is rejected. The login name is needed because the measurements are never exact, so it is difficult to index them and then search the index. Also, two people might have the same characteristics, so requiring the measured characteristics to match those of a specific user is stronger than just requiring it to match those of any user.

Access Control

Operating System Security

Security refers to providing a protection system to computer system resources such as CPU, memory, disk, software programs and most importantly data/information stored in the computer system. If a computer program is run by unauthorized user then he/she may cause severe damage to computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc. We're going to discuss following topics in this article.

- Authentication
- One Time passwords
- Program Threats
- System Threats
- Computer Security Classifications

Authentication

Authentication refers to identifying the each user of the system and associating the executing programs with those users. It is the responsibility of the Operating System to create a protection system which ensures that a user who is running a particular program is authentic. Operating Systems generally identifies/authenticates users using following three ways:

- **Username / Password** - User need to enter a registered username and password with Operating system to login into the system.
- **User card/key** - User need to punch card in card slot, or enter key generated by key generator in option provided by operating system to login into the system.
- **User attribute - fingerprint/ eye retina pattern/ signature** - User need to pass his/her attribute via designated input device used by operating system to login into the system.

One Time passwords

One time passwords provides additional security along with normal authentication. In One-Time Password system, a unique password is required every time user tries to login into the system. Once a one-time password is used then it cannot be used again. One time password are implemented in various ways.

Random numbers - Users are provided cards having numbers printed along with **corresponding alphabets. System asks for numbers corresponding to few alphabets randomly chosen.**

- **Secret key** - User are provided a hardware device which can create a secret id mapped with user id. System asks for such secret id which is to be generated every time prior to login.
- **Network password** - Some commercial applications send one time password to user on registered mobile/ email which is required to be entered prior to login.

Program Threats

Operating system's processes and kernel do the designated task as instructed. If a user program made these process do malicious tasks then it is known as Program Threats. One of the common example of program threat is a program installed in a computer which can store and send user credentials via network to some hacker. Following is the list of some well known program threats.

- **Trojan Horse** - Such program traps user login credentials and stores them to send to malicious user who can later on login to computer and can access system resources.

- **Trap Door** - If a program which is designed to work as required, have a security hole in its code and perform illegal action without knowledge of user then it is called to have a trap door.
- **Logic Bomb** - Logic bomb is a situation when a program misbehaves only when certain conditions met otherwise it works as a genuine program. It is harder to detect.
- **Virus** - Virus as name suggests can replicate themselves on computer system .They are highly dangerous and can modify/delete user files, crash systems. A virus is generally a small code embedded in a program. As user accesses the program, the virus starts getting embedded in other files/ programs and can make system unusable for user.

System Threats

System threats refers to misuse of system services and network connections to put user in trouble. System threats can be used to launch program threats on a complete network called as program attack. System threats creates such an environment that operating system resources/ user files are misused. Following is the list of some well known system threats.

- **Worm** -Worm is a process which can choked down a system performance by using system resources to extreme levels. A Worm process generates its multiple copies where each copy uses system resources, prevents all other processes to get required resources. Worms processes can even shut down an entire network.
- **Port Scanning** - Port scanning is a mechanism or means by which a hacker can detects system vulnerabilities to make an attack on the system.
- **Denial of Service** - Denial of service attacks normally prevents user to make legitimate use of the system. For example user may not be able to use internet if denial of service attacks browser's content settings.

Identification, Authentication, Authorization

For a user to be able to access a resource, he must first prove that he is who he claims to be, has the necessary credentials, and has been given the necessary rights or privileges to perform the actions he is requesting.

Identification describes a method of ensuring that a subject (user, program, or process) is the entity it claims to be. Identification can be provided with the use of a username or account number.

To be properly **authenticated**, the subject is usually required to provide a second piece to the credential set (a password, a cryptographic key, personal identification number (PIN), ….).

Identification, Authentication, Authorization (cont'd)

If identification and authentication credentials match the stored information, the subject is **authenticated**.

Once the subject is authenticated, the system it is trying to access needs to determine if this subject has been given the necessary rights and privileges to carry out the requested actions.

If the system determines that the subject may access the resource, it **authorizes** the subject. These mechanisms are enforced through AAA (Authentication, Authorization and Auditing) tools.

AAA tools

Access controls tools are used for identification, authentication, authorization, and auditability. They are software components that enforce access control measures for systems, programs, processes, and information.

They can be embedded within operating systems, applications, add-on security packages, or database and telecommunication management systems.

They can be offered as outsourced services by trusted third parties.

It can be challenging to synchronize all access controls and ensure that all vulnerabilities are covered without producing overlaps of functionality.

Identification and Authentication

Once a person has been identified, through the user ID, he must be authenticated; He must prove he is who he says he is.

There are three general factors that can be used for authentication:

- something a person knows (a password, PIN, …);
- something a person has (a key, an access card, a badge);
- something a person is (physical attributes).

Identification and Authentication (cont'd)

Authenticating a person by so*mething that he knows* is usually the least expensive to implement, but it is less secure, too. Another person may easily acquire this knowledge and gain unauthorized access to a system.
Something a person has is a very common mechanism but the token's life-cycle needs to be managed, they can be lost or stolen, which could result in unauthorized access.
Authenticating a person's identity based on a unique *physical attribute* is referred to as biometrics.
Strong authentication contains two out of these three methods: something a person knows, has, or is (*two-factors authentication*).

Identification Component Requirements

When issuing identification values to users, the following should be in place:

- Each value should be unique, for user accountability.
- A standard naming scheme should be followed.
- The value should be non-descriptive of the user's position or tasks.

- The value should not be shared between users.

Identity management

Identity management is a broad term that encompasses the use of different products to identify, authenticate, and authorize users through automated means.

The continual increase in complexity and diversity of networked environments only increases the complexity of keeping track of who can access what and when.

Users usually access several different types of systems throughout their daily tasks, which makes controlling access and providing the necessary level of protection on different data types difficult and full of obstacles.

This complexity usually results in unforeseen and unidentified holes in asset protection, overlapping and contradictory controls, and policy and regulation noncompliance.

It is the goal of identity management technologies to simplify the administration of these tasks and bring sanity to chaos.

Identity management (cont'd)

The following are many of the common problems that enterprises deal with today in controlling access to assets:

- Various types of users need different levels of access: internal users, contractors, outsiders, partners, etc.
- Resources have different classification levels: confidential, internal use only, private, public, etc.
- Diverse identity data must be kept on different types of users: credentials, personal data, contact information, work-related data, digital certificates, cognitive passwords, etc.
- The corporate environment is continually changing: business environment needs, resource access needs, employee roles, current employees, etc.

Identity management (cont'd)

The traditional identity management process has been manual, using directory services with permissions and profiles.

This approach has proven incapable of keeping up with complex demands and thus has been replaced with the use of newly arrived automated applications that are rich in functionality, including enterprise-wide products and single sign-on solutions.

Identity management (cont'd)

The following are some of the services that these types of products supply:

- User provisioning
- Password synchronization and resetting

- Self service for users on specific types of activities
- Delegation of administrative tasks
- Centralized auditing and reporting
- Integrated workflow and increase in business productivity
- Decrease in network access points
- Regulatory compliance

Authentication mechanisms

Biometrics:

- Fingerprint
- Palm Scan
- Hand Geometry
- Retina Scan
- Iris Scan
- Signature Dynamics
- Keyboard Dynamics
- Voice Print
- Facial Scan
- Hand Topography

Authentication mechanisms (cont'd)

Passwords….. weakness:

If an attacker is after a password, he can try different techniques:

- **Electronic monitoring** Listening to network traffic to capture information, especially when a user is sending her password to an authentication server. The password can be copied and reused by the attacker at another time, which is called a replay attack.
- **Access the password file** Usually done on the authentication server. The password file contains many users' passwords and, if compromised, can be the source of a lot of damage. This file should be protected with access control mechanisms and encryption.
- **Brute force attacks** Performed with tools that cycle through many possible character, number, and symbol combinations to uncover a password.
- **Dictionary attacks** Files of thousands of words are used to compare to the user's password until a match is found.
- **Social engineering** An attacker falsely convinces an individual that she has the necessary authorization to access specific resources.

Authentication mechanisms (cont'd)

- Password Checkers;
- Password Hashing and Encryption;

- Password Aging;
- Limit Logon Attempts;
- Cognitive Passwords;
- One-Time Passwords:
- Synchronous token device,
- Asynchronous token device;
- Cryptographic keys;
- Smart Cards.

Authorization mechanisms (cont'd)

After successful authentication, the system must establish whether the user is authorized to access the particular resource and what actions he is permitted to perform on that resource.

Authorization is a core component of every operating system, but applications, security add-on packages, and resources themselves can also provide this functionality.

The decision of whether or not to allow users to access some resource was based on access criteria.

Access criteria is the crux of authentication.

Authorization: Access Criteria

This subject can get very granular in its level of detail when it comes to dictating what a subject can or cannot do to an object or resource.

This is a good thing for network administrators and security professionals, because they want to have as much control as possible over the resources they have been put in charge of protecting, and a fine level of detail enables them to give individuals just the precise level of access that they need.

It would be frustrating if access control permissions were based only on full control or no access. These choices are very limiting, and an administrator would end up giving everyone full control, which would provide no protection.

There are different ways of limiting access to resources and, if they are understood and used properly, they can give just the right level of access desired.

Authorization: Access Criteria (cont'd)

Granting access rights to subjects should be based on the level of trust a company has in a subject and the subject's need to know. Just because a company completely trusts Alice with its files and resources does not mean she fulfills the need-to-know criteria to access the company's tax returns and profit margins.

These issues need to be identified and integrated into the access criteria.

The different access criteria can be broken up into different types:

- roles,
- groups,
- location,
- time,
- transaction types.

Access Criteria

- Using ***roles*** is an efficient way to assign rights to a type of user who performs a certain task. The role is based on a job assignment or function.
- Using ***groups*** is another effective way of assigning access control rights. If several users require the same type of access to information and resources, putting them into a group and then assigning rights and permissions to that group is easier to manage than assigning rights and permissions to each and every individual separately.

Access Criteria (cont'd)

- **Physical or logical location** can also be used to restrict access to resources. Some files may be available only to users who can log on interactively to a computer. Logical location restrictions are usually done through network address restrictions.
- **Time of day**, or temporal isolation, is another access control mechanism that can be used.
- **Transaction-type** restrictions can be used to control what data is accessed during certain types of functions and what commands can be carried out on the data. An online banking program may allow a customer to view his account balance, but may not allow the customer to transfer money until he has a certain security level or access right. (A database administrator may be able to build a database for the human resources department, but may not be able to read certain confidential files within that database).

Authorization Creep

As employees work at a company over time and move from one department to another, they often are assigned more and more access rights and permissions.

This is commonly referred to as authorization creep. It can be a large risk for a company, because too many users have too much privileged access to company assets.

Users' access needs and rights should be periodically reviewed to ensure that the principle of least privilege is being properly enforced.

Notes on Authorization

It is important to understand that it is management's job to determine the security requirements of individuals and how access is authorized.

The security administrator configures the security mechanisms to fulfill these requirements, but it is not her job to determine security requirements of users.

Access Control Models

An **access control model** is a framework that dictates how subjects access objects. It uses access control technologies and security mechanisms to enforce the rules and objectives of the model.

There are three main types of access control models:

- Discretionary (DAC),
- Mandatory (MAC),
- Nondiscretionary (also called role-based RBAC).

Each model type uses different methods to control how subjects access objects.

For every access attempt, before a subject can communicate with an object, the security monitor reviews the rules of the access control model to determine whether the request is allowed.

Discretionary Access Control

If a user creates a file, he is the owner of that file. An identifier for this user is placed in the file header.

A system that uses ***discretionary access control (DAC)*** enables the owner of the resource to specify which subjects can access specific resources.

This model is called discretionary because the control of access is based on the discretion of the owner.

The most common implementation of DAC is through ACLs, which are dictated and set by the owners and enforced by the operating system.

Discretionary Access Control (cont'd)

Most of the operating systems are based on DAC models, such as all Windows, Linux, and Macintosh systems and most flavors of Unix.

When you look at the properties of a file or directory and you see the choices that allow you to control which users can have access to this resource and to what degree, you are witnessing an instance of ACLs enforcing a DAC model.

DACs can be applied to both the directory tree structure and the files it contains.

The PC world has access permissions of No Access, Read (r), Write (w), Execute (x), Delete (d), Change (c), and Full Control.

Notes on DAC

Identity-Based Access Control

- DAC systems grant or deny access based on the identity of the subject. The identity can be a user identity or group membership. So, for example, a data owner can choose to allow Bob (user identity) and the Accounting group (group membership identity) to access his file.

Mandatory Access Control

In a ***mandatory access control (MAC)*** model, users and data owners cannot determine who can access files.

This model is much more structured and strict and is based on a security label system. Users are given a security clearance (secret, top secret, confidential, and so on), and data is classified in the same way. The clearance and classification data is stored in the security labels, which are bound to the specific subjects and objects.

When the system makes a decision about fulfilling a request to access an object, it is based on the clearance of the subject, the classification of the object, and the security policy of the system.

The rules for how subjects access objects are made by the security officer, configured by the administrator, enforced by the operating system, and supported by security technologies.

DAC and MAC limitations

Each organization has unique security requirements, many of which are difficult to meet using traditional DAC and MAC controls.

DAC is an access control mechanisms that permits system users to allow or disallow other users access to objects under their control without the intercession of a system administrator.

In many organizations, the end users do not “own” the information for which they are allowed access; the actual “owner” is the corporation -> control has to be based on employee functions rather than data ownership.

Role-based Access Control

A user has access to an object based on the assigned role.

Roles are defined based on job functions.

Permissions are defined based on job authority and responsibilities within a job function.

Operations on an object are invocated based on the permissions.

The object is concerned with the user’s role and not the user.

UNIT V
POSSIBLE QUESTIONS
(2 MARKS)

1. What is Authentication?
2. What is Security Policy?
3. What is Security Mechanism?
4. What is Access Control Lists?
5. What is internal access authorization?

(6 MARKS)

1. Discuss about Policy mechanism with example.
2. Explain in detail about Authentication.
3. Describe about Internal access Authorization.

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

II B.Sc CS (Batch 2016-2019)

OPERATING SYSTEMS

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE QUESTI

ONLINE EXAMINATIONS ONE MARKS QUESTI

UNIT - 5

SNO	QUESTIONS	opt1	opt2	opt3	opt4
1	In computer security, ……………………. means that computer system assets can be modified only by authorized parities.	Confidentiality	Integrity	Availability	Authenticity
2	In computer security, …………………….. means that the information in a computer system only be accessible for reading by authorized parities.	Confidentiality	Integrity	Availability	Authenticity
3	Which of the following is independent malicious program that need not any host program?	Trap doors	Trojan horse	Virus	Worm
4	The ……….. is code that recognizes some special sequence of input or is triggered by being run from a certain user ID of by unlikely sequence of events.	Trap doors	Trojan horse	Logic Bomb	Virus
5	The …………….. is code embedded in some legitimate program that is set to “explode” when certain conditions are met.	Trap doors	Trojan horse	Logic Bomb	Virus
6	Which of the following malicious program do not replicate automatically?	Trojan Horse	Virus	Worm	Zombie
7	…………… programs can be used to accomplish functions indirectly that an unauthorized user could not accomplish directly.	Zombie	Worm	Trojan Horses	Logic Bomb

8	A ………….. is a program that can infect other programs by modifying them, the modification includes a copy of the virus program, which can go on to infect other programs.	Worm	Virus	Zombie	Trap doors
9	Which principle states that programs, users and even the systems be given just enough privileges to perform their task?	principle of operating system	principle of least privilege	principle of process scheduling	none of the mentioned
10	_______ is an approach to restricting system access to authorized users.	Role-based access control	Process-based access control	Job-based access control	none of the mentioned
11	For system protection, a process should access	all the resources	only those resources for which it has authorization	few resources but authorization is not required	all of the mentioned
12	The protection domain of a process contains	object name	rights-set	object name and rights-set	none of the mentioned
13	If the set of resources available to the process is fixed throughout the process’s lifetime then its domain is	static	dynamic	neither static nor dynamic	none of the mentioned
14	Access matrix model for user authentication contains	a list of objects	a list of domains	a function which returns an object’s type	all the options
15	Global table implementation of matrix table contains	domain	object	right-set	all the options
16	For a domain _______ is a list of objects together with the operation allowed on these objects.	capability list	access list	authorization	none of the mentioned
17	Which one of the following is capability based protection system?	hydra	cambridge CAP system	hydra & cambridge CAP system	none of the mentioned

18	In UNIX, domain switch is accomplished via	file system	user	superuser	none of the mentioned
19	___________ is an important property of an operating system that hopes to keep up with advancements in computing technology.	Portability	Reliability	Extensibility	compatibility
20	___________ is the ability to handle error conditions, including the ability of the operating system to protect itself and its users from defective or malicious software.	Portability	Reliability	Extensibility	compatibility
21	__________ is the ability to move from one hardware architecture to another with relatively few changes.	Portability	Reliability	Extensibility	compatibility
22	Windows NT is designed to afford good ___________.	Portability	Reliability	Extensibility	performance
23	A __________ is created by the NT disk administrator utility, and is based on a logical disk partition.	Volume	File	Directory	subdirectory
24	A __________ of a directory contains the top level of the B+ tree.	index root	file reference	attributes	metadata
25	The _________ in NT may occupy a portion of a disk, may occupy an entire disk or may span across several disks.	Volume	File	Directory	subdirectory
26	The ___________ of a directory contains the top level of the B+ tree.	Volume	File	index root	subdirectory
27	The _____________ attribute contains the access token of the owner of the file, and an access control list that states the access privileges that are granted to each user that has access to the file.	Portability	Recovery	Reliability	Security

28	To deal with disk sectors that go bad, ____________ uses a hardware technique called sector spanning.	Ps	Valloc	Kmalloc	FtDisk
29	In the security literature, people who are nosing around places where they have no business being are called __________	**intruders**	crackers	hackers	worms
30	Outsiders can sometimes take command of people's home computers (using viruses and other means) and turn them into ______________	virus	worms	malware	**zombies**
31	Most operating systems allow individual users to determine who may read and write their files and other objects, This policy is called __________________	mandatory access	access matrix	**discretionary access control.**	access control lists
32	Every secured computer system must require all users to be ___________________at login time	**authenticated**	authorized	transferred	scheduled
33	The most widely used form of authentication is to require the user to type a __________and a _______________`	mailid, PIN numb	**login name, password.**	PIN number, Account number	Username, mailid
34	The authentication method that measures the physical characteristics of the user that are hard to forge is called as ______________	**Biometrics**	password	stegnography	access control
35	____________is the name given to hackers who break into computers for criminal gain	hackers	spoofing	phising	**Crackers**
36	A typical biometrics system has two parts: ________________	**enrollment and identification**	identification & authentication	authentication & confidentiality	authorization and authentication
37	Any malware hidden in software or a Web page that people voluntarily download is called _________________	worm	**Trojan Horse**	Virus	Backdoor

38	The idea of creating a virus that could overwrite the master boot record or the boot sector, with devastating results, such viruses called as ________________	device driver virus	source code virus	companion virus	**boot sector viruses**
39	The trick to infect a device driver leads to a ________________	source code virus	**device driver virus**	companion virus	boot sector viruses
40	When an attempt is to make a machine or network resource unavailable to its intended users, the attack is called	**denial-of-service attack**	slow read attack	spoofed attack	starvation attack
41	The code segment that misuses its environment is called a	internal thief	**trojan horse**	code stacker	none of the mentioned
42	The internal code of any software that will set of a malicious function when specified conditions are met, is called	**logic bomb**	trap door	code stacker	none of the mentioned
43	The pattern that can be used to identify a virus is known as	stealth	**virus signature**	armoured	multipartite
44	Which one of the following is a process that uses the spawn mechanism to revage the system performance?	**worm**	trojen	threat	virus
45	What is a trap door in a program?	**a security hole, inserted at programming time in the system for later use**	a type of antivirus	security hole in a network	none of the mentioned
46	Which one of the following is not an attack, but a search for vulnerabilities to attack?	denial of service	**port scanning**	memory access violation	dumpster diving
47	File virus attaches itself to the	source file	object file	**executable file**	all of the mentioned
48	Multipartite viruses attack on	files	boot sector	memory	**all of the mentioned**

49	In asymmetric encryption	same key is used for encryption and decryption	**different keys are used for encryption and decryption**	no key is required for encryption and decryption	none of the mentioned
50	Which of the following are forms of malicious attack ?	Theft of information	Modification of data	Wiping of information	All of the mentioned
51	What are common security threats ?	File Shredding	File sharing and permission	File corrupting	File integrity
52	From the following, which is not a common file permission ?	Write	Execute	Stop	Read
53	Which of the following is a good practice ?	Give full permission for remote transferring	Grant read only permission	Grant limited permission to specified account	Give both read and write permission but not execute.
54	What is not a good practice for user administration ?	Isolating a system after a compromise	Perform random auditing procedures	Granting privileges on a per host basis	Using telnet and FTP for remote access.
55	Which of the following is least secure method of authentication ?	Key card	fingerprint	retina pattern	Password
56	Which of the following is a strong password ?	19thAugust88	Delhi88	P@assw0rd	!augustdelhi
57	What does Light Directory Access Protocol (LDAP) doesn't store ?	Users	Address	Passwords	Security Keys
58	Which happens first authorization or authentication ?	Authorization	Authentication	Both are same	None of the mentioned
59	What is characteristics of Authorization ?	RADIUS and RSA	3 way handshaking with syn and fin.	Multilayered protection for securing resources	Deals with privileges and rights

60	What forces the user to change password at first logon ?	Default behavior of OS	Part of AES encryption practice	Devices being accessed forces the user	Account administrator

IONS

ONS

opt5	opt6	**ANSWER**
		Integrity
		Confidentiality
		Worm
		Trap doors
		Trap doors
		Trojan Horse
		Trojan Horses

		Virus
		principle of least privilege
		Role-based access control
		only those resources for which it has authorization
		object name and rights-set
		static
		all the options
		all the options
		capability list
		hydra & cambridge CAP system

		file system
		Extensibility
		Reliability
		Portability
		performance
		Volume
		index root
		Volume
		index root
		Security

		FtDisk
		intruders
		zombies
		discretionary access control
		authenticated
		login name, password.
		Biometrics
		Crackers
		enrollment and identification
		Trojan Horse

		boot sector viruses
		device driver virus
		denial-of-service attack
		trojan horse
		logic bomb
		virus signature
		worm
		a security hole, inserted at programming time in the system for later use
		port scanning
		executable file
		all of the mentioned

		different keys are used for encryption and decryption
		All of the mentioned
		File sharing and permission
		Stop
		Grant limited permission to specified account
		Using telnet and FTP for remote access.
		Password
		P@assw0rd
		Address
		Authorization
		Deals with privileges and rights

		Account administrator

Register Number____________

[16CSU302]

KARPAGAM ACADEMY FOR HIGHER EDUCATION
KARPAGAM UNIVERSITY
(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2016 onwards)

B.Sc COMPUTER SCIENCE
SECOND INTERNAL EXAMINATION – AUGUST 2017
OPERATING SYSTEMS

Date & Session : .08.2017 &
Maximum: 50 marks **Duration: 2 hours**

PART-A (20 X 1= 20 Marks)
ANSWER ALL THE QUESTIONS

1. Messages sent by a process :
   a) have to be of a fixed size b) have to be a variable size
   c) can be fixed or variable sized d) None of these
2. A direct method of deadlock prevention is to prevent the occurrences of ....
   a) Mutual exclusion b) Hold and wait **c) Circular waits** d) No preemption
3. A ___________ is a program segment where shared resources are accessed.
   a) critical section b) sub section c) cross section d) class section
4. Each process accessing the shared data excludes all the others from doing so simultaneously
   a) Mutual exclusion b) Deadlock prevention c) Preemption d) Circular Wait
5. Semaphores are used to solve the problem of ____________
   a) race condition b) process synchronization **c) mutual exclusion** d) belady problem

______________can assume only the value 0 or the value 1
   a) Binary semaphores b) Counting semaphores
   c) semaphore operations d) normal semaphores

6. Memory is array of ____________
   a) bytes b) circuits c) ics d) ram
7. Run time mapping is done using ______________
   a) MMU b) CPU c) CU d) IU
8. In paging physical memory is divided into ________________
   a) frames b) pages c) segments d) bytes
9. CPU fetches instructions from _______
   a) memory b) pendrive c) dvd d) cmos
10. _____________ is used to divide a process into fixed size chunks
    a) paging b) segmentation c) sp d) swapping

11. ________________ fault occurs when desired page is not in memory
 a) page b) segment c) pages d) segments

12. _______________ is first page replacement algorithm
 a) optimal **b) FIFO** c) LRU d) NRU

13. Program must be in _________________
 a) memory b) pendrive c) dvd d) cmos

14. Address generated by CPU is _______________ address
 a) logical b) physical c) direct d) indirect

15. ______________ is called as high paging activity
 a) thrashing b) smashing c) mocking d) breaking

16. Virtual memory abstracts _________ memory
 a) virtual b) eerom **c) main** d) eprom

17. Collection of process in disk forms_____
 a) input queue b) output queue c) stack d) circle

18. If process location is found during compile time then __________ code is generated
 a) absolute b) relative c) approximate d) more or less

19. ______________ can be internal and external
 a) fragmentation b) merging c) grouping d) fixing

20. ______________ is first in mapping of virtual to physical address in paging
 a) direct b) associate c) direct & associative d) pointing

PART-B (3 X 2 = 6 Marks)
(Answer ALL the Questions)

21. What is a Deadlock?

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

22. What is a Physical address space?

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a 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**

23. What is Paging?

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.

Page address is called **logical address** and represented by **page number**and the **offset**.

PART-C (3 X 8 = 24 Marks)
(Answer ALL the Questions)

24. a) Write a note on Non pre-emptive and Preemptive scheduling algorithms.

higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called **preemptive scheduling.**

Eg: Round robin

In **non-preemptive scheduling**, a running task is executed till completion. It cannot be interrupted.
Eg First In First Out

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

1. When a process switches from the running state to the waiting state (for example, as the result of an I/0 request or an invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/0)
4. When a process terminates.

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

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
- Multiple-Level Queues 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.

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

Shortest Job Next (SJN)

- 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

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

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.

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

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.
-

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

[OR]

24. b) Explain about Deadlocks in detail.

<u>Deadlocks</u>

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use. The process can operate on the resource (for example, if the resource

is a printer, the process can print on the printer).

3. **Release**. The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously

in a system:

1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

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

3. No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set $\{P0, P1, ..., Pn\}$ of waiting processes must exist such that $P0$ is waiting for a resource held by $P1$, $P1$ is waiting for a resource held by $P2$, ..., $Pn-1$ is waiting for a resource held by Pn, and Pn is waiting for a resource held by $P0$.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: $P = \{P1, P2, ..., Pn\}$, the set consisting of all the active processes in the system, and $R = \{R1, R2, ..., Rm\}$, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by $Pi \rightarrow Rj$; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by $Rj \rightarrow Pi$; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge $Pi \rightarrow Rj$ is called a **request edge**; a directed edge $Rj \rightarrow Pi$ is called an **assignment edge**.

Pictorially, we represent each process Pi as a circle and each resource type Rj as a rectangle. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle Rj , whereas an assignment edge must also designate one of the dots in the rectangle.

When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure depicts the following situation.

• The sets P, R, and E:

◦ $P = \{P1, P2, P3\}$

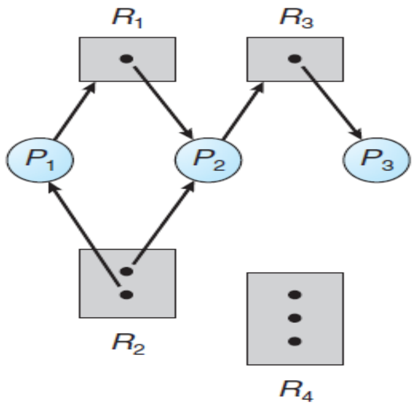


Fig 2.13. Resource-allocation graph.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.

• We can allow the system to enter a deadlocked state, detect it, and recover.

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Methods

■Deadlock Prevention

■ Deadlock Avoidance

■Deadlock Detection

25. a) Write about Memory Allocation strategies in detail.

Memory Allocation Strategies

1. First Fit
2. Best fit
3. Worst fit
4. Next fit

First Fit

• **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage

Fastest algorithm because it searches as little as possible.

Disadvantage

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

First Fit

The first of these is called first fit. The basic idea with first fit allocation is that we begin searching the list and take the first block whose size is greater than or equal to the request size. If we reach the end of the list without finding a suitable block, then the request fails.

To illustrate the behavior of first fit allocation, as well as the other allocation policies later, we trace their behavior on a set of allocation and de-allocation requests. We denote this sequence as A20, A15, A10, A25, D20, D10, A8, A30, D15, A15, where An denotes an allocation request for n KB and Dn denotes a de-allocation request for the allocated block of size n KB. (For simplicity of notation, we have only one block of a given size allocated at a time. None of the policies depend on this property; it is used here merely for clarity.) In these examples, the memory space from which we serve requests is 128 KB. Each row of Figure 9-9 shows the state of memory after the operation labeling it on the left.

Shaded blocks are allocated and unshaded blocks are free. The size of each block is shown in the corresponding box in the figure. In this, and other allocation figures in this chapter, time moves downward in the figure. In other words, each operation happens prior to the one below it.

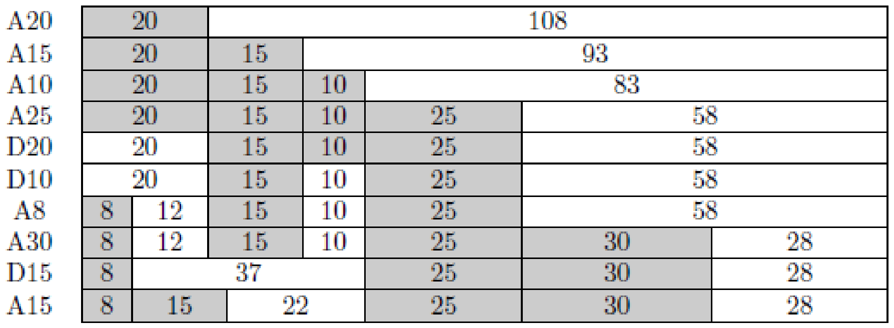


Fig 3.4. First Fit Allocation

Next Fit

If we want to spread the allocations out more evenly across the memory space, we often use a policy called next fit. This scheme is very similar to the first fit approach, except for the place where the search starts. In next fit, we begin the search with the free block that was next on the list after the last allocation**.** During the search, we treat the list as a circular one. If we come back to the place where we started without finding a suitable block, then the search fails.
For the next three allocation policies in this section, the results after the first six requests (up through the D10 request) are the same. In Figure, it is shown the results after each of the other requests when following the next fit policy.

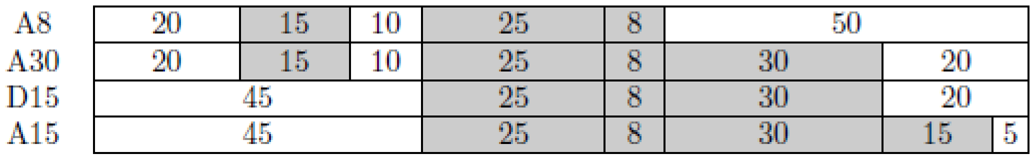


Fig 3.5. Next Fit Allocation

Best Fit

• **Best fit**. Allocate the smallest hole that is big enough.Wemust search theentire list, unless the list is ordered by size. This strategy produces thesmallest leftover hole.

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage

It is slower and may even tend to fill up memory with tiny useless holes.

In many ways, the most natural approach is to allocate the free block that is closest in size to the request. This technique is called best fit. In best fit, we search the list for the block that is smallest but greater than or equal to the request size.

As with the next fit example, we show only the final four steps of best fit allocation for our example.

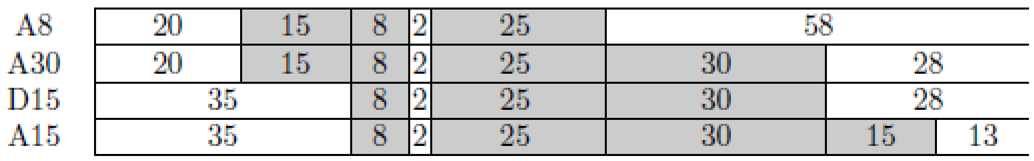


Fig 3.6. **Best Fit Allocation**

Worst fit

• **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage

Reduces the rate of production of small gaps.

Disadvantage

If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

Worst Fit

If best fit allocates the smallest block that satisfies the request, then worst fit allocates the largest block for every request. Although the name would suggest that we would never use the worst fit policy, it does have one advantage: If most of the requests are of similar size, a worst fit policy tends to minimize external fragmentation.

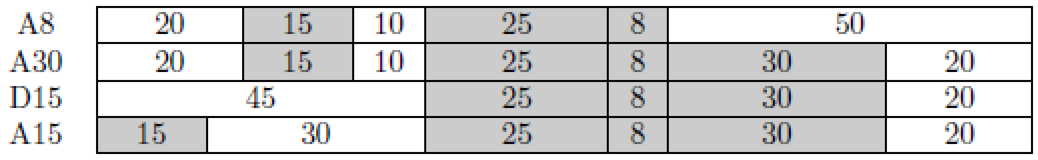


Fig 3.7. Worst fit allocation

[OR]

25. b) Discuss about Paging in detail.

<u>**Paging**</u>

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.

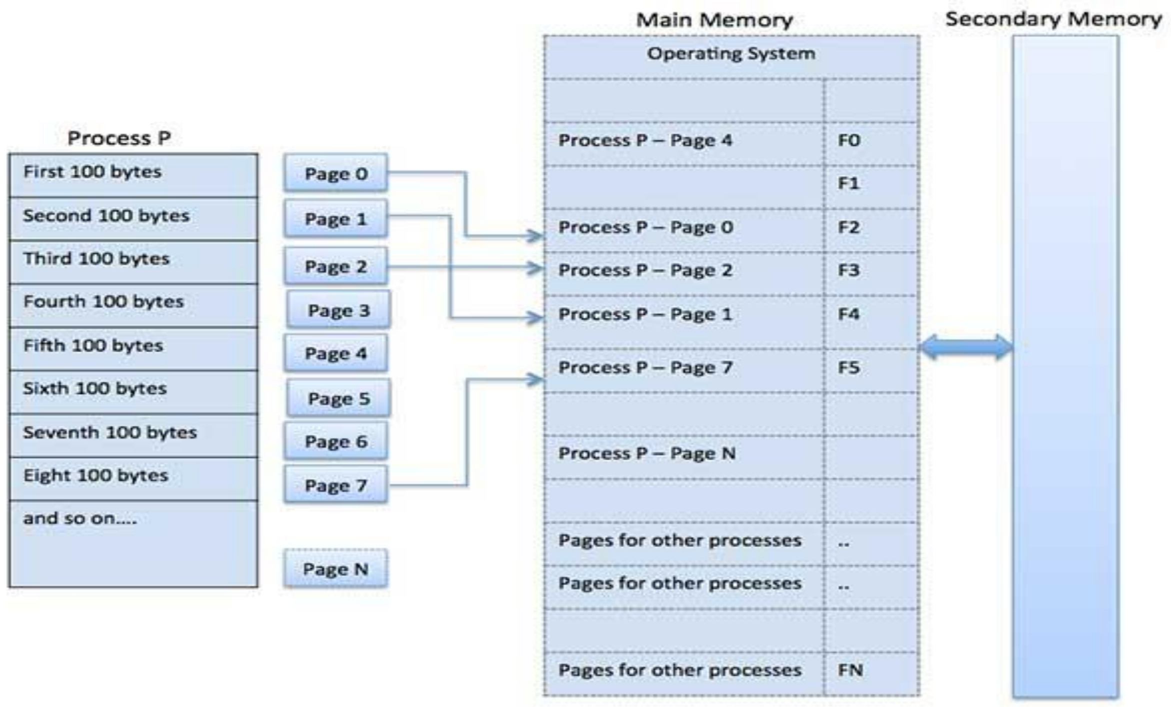


Fig 3.10. Paging Process

Address Translation

Page address is called **logical address** and represented by **page number**and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.

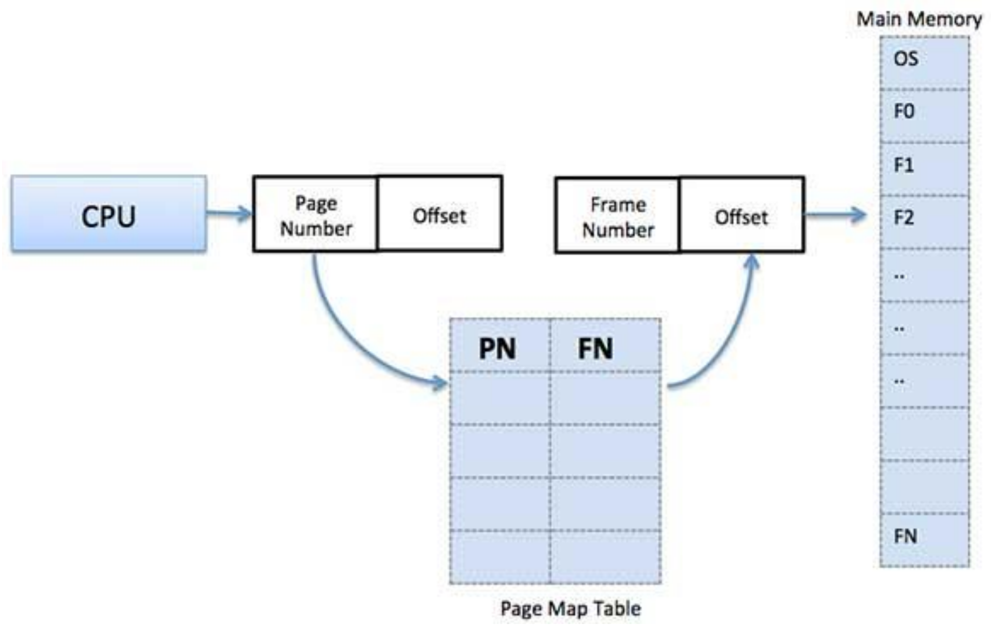


Fig 3.11. Paging with Page Table

When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

Paging

External fragmentation is avoided by using paging technique. Paging is a technique in which physical memory is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). When a process is to be executed, it's corresponding pages are loaded into any available memory frames.

Logical address space of a process can be non-contiguous and a process is allocated physical memory whenever the free memory frame is available. Operating system keeps track of all free frames. Operating system needs n free frames to run a program of size n pages.

Address generated by CPU is divided into

- **Page number (p)** -- page number is used as an index into a page table which contains base address of each page in physical memory.
- **Page offset (d)** -- page offset is combined with base address to define the physical memory address.

● Each virtual address space is divided into fixed-size chunks called pages.

● The physical address space is divided into fixed-size chunks called frames.

● Pages have same size as frames.

● The kernel maintains a page table (or page-frame table) for each process, specifying the frame within which each page is located.

● The CPU's memory management unit (MMU) translates virtual addresses to physical addresses on-the-fly for every memory access.

Properties:

● relatively simple to implement (in hardware);

● virtual address space need not be physically contiguous.

26. a) Write about Physical and Virtual address space in detail.

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

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. 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**.

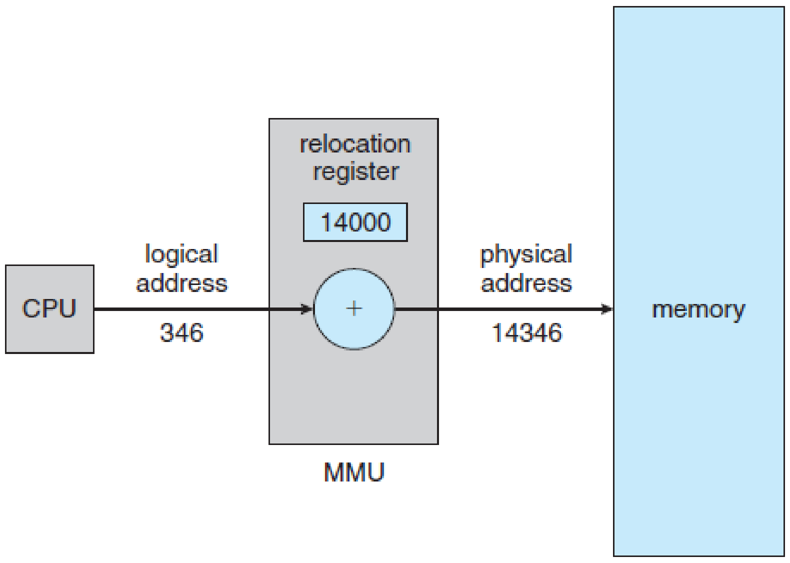


Fig 3.2. Physical address and Logical Address

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)**. 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 (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location14000; an access to location 346 is mapped to location 14346.

The user program never sees the real physical addresses.

Static vs Dynamic Loading

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

[OR]

26. b) Describe about Virtual memory.

Virtual Memory

Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory. This technique is useful as large virtual memory is provided for user programs when a very small physical memory is there.

In real scenarios, most processes never need all their pages at once, for following reasons :

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
- Certain features of certain programs are rarely used.

Benefits of having Virtual Memory :

1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.

3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

Demand Paging

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them(On demand). This is termed as lazy swapper, although a pager is a more accurate term.

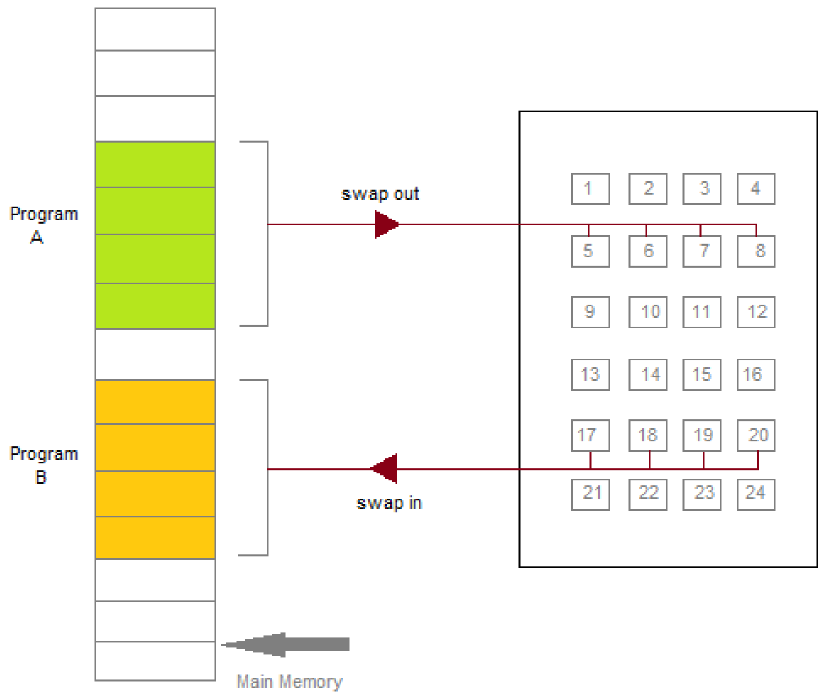


Fig 3.15. Demand Paging Process

Initially only those pages are loaded which will be required the process immediately.

The pages that are not moved into the memory, are marked as invalid in the page table. For an invalid entry the rest of the table is empty. In case of pages that are loaded in the memory, they are marked as valid along with the information about where to find the swapped out page.

When the process requires any of the page that is not loaded into the memory, a page fault trap is triggered and following steps are followed,

1. The memory address which is requested by the process is first checked, to verify the request made by the process.
2. If its found to be invalid, the process is terminated.

3. In case the request by the process is valid, a free frame is located, possibly from a free-frame list, where the required page will be moved.
4. A new operation is scheduled to move the necessary page from disk to the specified memory location. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to valid.
6. The instruction that caused the page fault must now be restarted from the beginning.

There are cases when no pages are loaded into the memory initially, pages are only loaded when demanded by the process by generating page faults. This is called **Pure Demand Paging**.

The only major issue with Demand Paging is, after a new page is loaded, the process starts execution from the beginning. Its is not a big issue for small programs, but for larger programs it affects performance drastically.

Page Replacement

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more number of processes into the memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem :

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called **Page replacement** and is most commonly used. We have some great algorithms to carry on page replacement efficiently.

Basic Page Replacement

- Find the location of the page requested by ongoing process on the disk.
- Find a free frame. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced, such frame is known as **victim frame**.
- Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
- Move the required page and store it in the frame. Adjust all related page and frame tables to indicate the change.
- Restart the process that was waiting for this page.

FIFO Page Replacement

- A very simple way of Page replacement is FIFO (First in First Out)
- As new pages are requested and are swapped in, they are added to tail of a queue and the page which is at the head becomes the victim.
- Its not an effective way of page replacement but can be used for small systems.

LRU Page Replacement

FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be ***used.*** If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**.