OPERATING SYSTEM

V.C.S.

OPERATING SYSTEM : IN A NUTSHELL

WHAT IS OPERATING SYSTEM

Operating systems perform two basically unrelated functions, extending the machine and managing resources and depending on who is doing the talking, you hear mostly about one function or the other.

The Operating System as an Extended Machine

The **architecture** (instruction set, memory organization, I/O and bus structure) of most computers at the machine language level is primitive to program, especially for input/output.

The Operating System as a Resource Manager

The concept of the operating system is primarily providing its users with a convenient interface 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.

The operating system is to provide for an orderly and controlled allocation of the processors, memories and I/O devices among the various programs competing for them.

When a computer (or network) has multiple users, the need for managing and protecting the memory, I/O devices and other resources is even greater, since the users might otherwise interfere with one another.

TYPES OF OPERATING SYSTEM

- Mainframe operating system
- Server operating system
- Multiprocessor operating system
- Real-time operating system
- Embedded operating system
- Smart card operating system

OPERATING SYSTEM CONCEPTS

All operating systems have certain basic concepts such as processes, memory and files that are central to understanding them.

CPU SCHEDULING

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

First Come First Served (FCFS) scheduling

First Come First Served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes. Shortest Job-First (SJF) scheduling is probably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult because predicting the length of the next CPU burst is difficult. The SJF algorithm is a special case of the general priority scheduling algorithm, which simply allocates the CPU to the highest priority process. Both priority and SJF scheduling may suffer from starvation. Aging is a technique to prevent starvation.

Round Robin (RR) Scheduling

Round Robin (RR) Scheduling is more appropriate for a time shared (interactive) system. RR scheduling allocates the CPU to the first process in the reach queue for q time units where q is the time quantum. After q time units, if the process has not relinquished the CPU, it is preempted and the process is put a tail of the ready queue. The major problem is the selection of the time quantum. If the quantum is too large, RR scheduling degenerate to FCFS scheduling and if the quantum is too small, scheduling overhead in the form of context switch time becomes excessive.

The FCFS algorithm is non preemptive, the RR algorithm is preemptive. The SJF and priority algorithms may be either preemptive or non preemptive.

Multilevel queue algorithms allow different algorithms to be used for various classes of processes. The most common is a foreground interactive queue, which use RR scheduling and a background batch queue, which uses FCFS scheduling. Multilevel feedback queues allow processes to move from one queue to another.

INPUT/OUTPUT

All computers have physical devices for acquiring input and producing output. After all, what good would a computer be if the users could not tell it what to do and could not get the results after it did the work requested. Many kinds of input and output devices exist, including

keyboards, monitors, printers and so on. It is up to the operating system to manage these devices.

One of the main functions of an operating system is to control all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices. The I/O code represent a significant fraction of the total operating system. How the operating system manages I/O.

First we will look at some of the principles of I/O hardware and then we will look at I/O software in general. I/O software can be structured in layers with each layer having a well-defined task to perform.

I/O DEVICES

I/O devices can be roughly divided into two categories: **block devices** and **character devices.** A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones.

Table : Some typical device, network and bus data rates

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-1 2 network	78 MB/sec
SCSI Ultra 2/tiisk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

PROCESS AND THREAD

In a multiprogramming system, the CPU also switches from program to program running each for tens for hundreds of milliseconds. While, strictly speaking at any instant of time, the CPU is running only one program in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism. Sometimes people speak of **pseudo parallelism** in this context, in contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with.

THREADS

A process has an address space containing program text and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information and more. By putting them together in the form of a process, they can be managed more easily.

The other concept is a process has a thread of execution, usually shortened to just **thread.** The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together Threads are the entities scheduled for execution on the CPU.

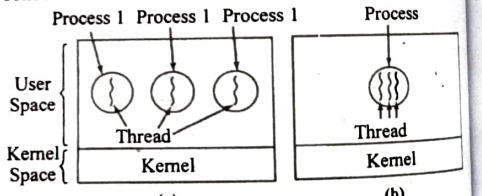


Fig. 1 : (a) Three processes each with one thread (b) One process with three threads

Thread Usage

1. Instead of thinking about interrupts, timers and context switches, we can think about parallel processes. Only now with threads we add a new element: the ability for the parallel entities to share an address space and all of its data among themselves. This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work.

2. A second argument for having threads is that since they do not have any resources attached to them, they are easier to create and destroy than processes.
3. Performance argument, threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O having threads allows these activities to overlap.
4. Finally, threads are useful on systems with multiple CPUs, where real parallelism is possible.

Implementing Threads in User Space

Each process needs its own private user space in **read table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties such the each thread's program counter, stack pointer, registers, state, etc. The thread table is managed by the run-time system.

Implementing Threads in the Kernel

No run-time system is needed in each, as shown in fig.2. Also there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system.

The kernel's thread table holds each thread's registers, state and other information.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready) or a thread from a different process.

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads and if so, run one of them while waiting for the required page to be brought in from the disk.

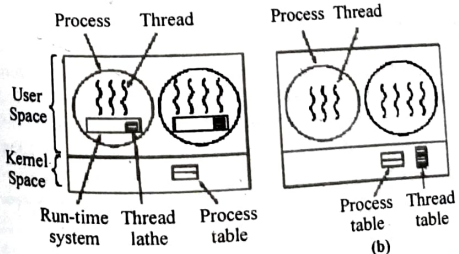


Fig. 2 : (a) A user-level threads package (b) A threads package managed by the kernel

INTER-PROCESS COMMUNICATION

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process and so on down the line. Thus, there is a need for communication between processes.

There are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 1 MB of memory). The third concerns proper sequencing when dependencies are present. If process A produces data and process B prints them, B has to wait until A has produced some data before starting to print.

- Race Condition
- Critical Region
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Semaphores
- Mutexes
- Monitors
- Message Passing

CLASSICAL IPC PROBLEM

The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem, he called the **dining philosophers problem.** Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.

The Readers and Writers Problem

Example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.

Memory hierarchy, with a small amount of a very fast expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, volatile main memory (RAM) and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

DEADLOCKS

When two or more processes are interacting, they can sometimes get themselves into a stalemate situation, they cannot get out of such a situation is called a deadlock.

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.
2. **Hold and wait :** A process must be holding at least one resource and waiting to acquire additional resource that are currently being held by other processes.
3. **No preemption :** Resources cannot be preempted.
4. **Circular wait :** A set $\{ P_0, P_1 \ldots P_r \}$ of waiting process must exist such that P_0 is waiting for a resource that to held by P_1. P_i is waiting for a resource that is held by P_2.

(i) A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting process.

(ii) To prevent deadlock we ensure that at least one of the necessary conditions never holds.

(iii) Principally there are three methods for dealing with deadlock.

(iv) Use some protocol to prevent or avoid deadlock ensuring that the system will never enter a deadlock state.

(v) Allow the system to enter deadlock state, detect it and then recover.

(vi) Ignore the problem all together and pretend that deadlock never occur in the system. This solution is the one used by most operating system including UNIX.

(vii) Another method for avoiding deadlock that is less stringent than the prevention algorithms is to have a prior information on how each process will be utilizing the resources. The banker's algorithm for example needs to know the maximum number of each resource class that may be requested by each process using this information, we can define a deadlock avoidance algorithm.

(viii) If a system does not employ a protocol to ensure that deadlock will never occur then a detection and recovery scheme must be employed. A deadlock detection algorithm must be invoked to determine whether a deadlock has occurred. If a deadlock is detected, the system must recover either by terminating some of the deadlocked process or by preempting resources from some of the deadlocked processes. In a system that select victims for primarily on the basis of cost factors, starvation may occur. As a result, the selected process never completes its designated task.

MEMORY MANAGEMENT

Every computer has some main memory that it uses to hold executing programs. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one placed in memory.

More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with the operating system), some kind of protection mechanism is needed. While this mechanism has to be in the hardware, it is controlled by the operating system.

Basic Memory Management

Memory management systems can be divided into two classes: those that move processes back and forward between main memory and disk during execution (swapping and paging) and those that do not.

Monoprogramming without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system.

Multiprogramming with Fixed Partitions

Except simple embedded systems, monoprogramming is hardly used any more. Most modern systems allow multiple processes to run at the same time. Having multiple processes running at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization. Network servers always have the ability to run multiple processes (for different clients) at the same time, but most client (i.e., desktop) machines also have this ability nowadays.

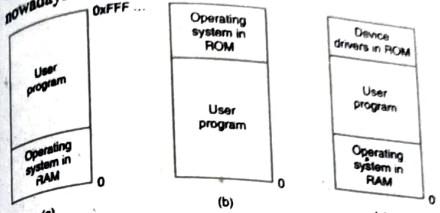


Fig. 3 : Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. If the average process computes only 20 percent of the time, it is sitting in memory with five processes in memory at once, the CPU should be busy all the time. This model is unrealistically optimistic, however since it assumes that all five processes will never be waiting for I/O at the same time.

Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

Fig.4 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming.**

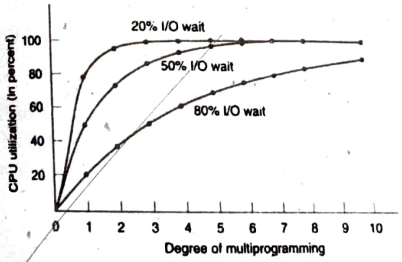


Fig.4 : CPU utilization as a function of the number of processes in memory

VIRTUAL MEMORY

The solution usually adopted was to split the program into pieces, called overlays. Overlay would start running first. When it was done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once.

Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces was time consuming and boring. .

The method that was devised has come to be known as virtual memory. The basic idea behind virtual memory is that the combined size of the program, data and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory and the rest on the disk. For example, a 16-MB program can run on a 4-MB machine by carefully choosing which 4 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run. So the CPU can be given to another process, the same way as in any other multiprogramming system. The topic covered under this are :

1. Paging
2. Page tables
   (i) Multi-levels page tables
   (ii) Structure of page table entry
3. TLB's-Translation Look aside Buffers
4. Inverted page tables

PAGE REPLACEMENT ALGORITHM

The Least Recently Used (LRU)

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for pages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU (Least Recently Used)** paging.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated every memory reference.

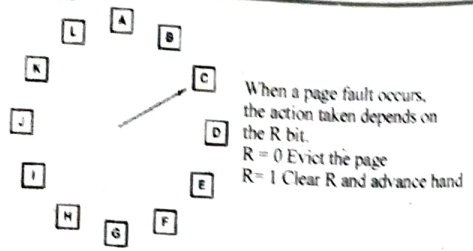


Fig. 5 : The clock page replacement algorithm

For a machine with n page frames the LRU hardware can maintain a matrix or $n \times n$ bits, initially all zero. Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used and so forth. The workings of this algorithm are given in fig.6 for four page frames and page references in the order 0 1 2 3 2 1 0 3 2 3

After page 0 is referenced, we have the situation. After page 1 is referenced, we have the situation and so forth.

DISK SCHEDULING

1. The basic hardware elements involved in I/O are buses, device controllers and the devices themselves. The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is off loaded to a DMA controller. The kernel module that controls a device is a device driver. The system call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character devices, memory mapped files, network sockets and programmed interval timers. The system calls usually block the process that issue them, but non blocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.
2. Disk drives are the major secondary storage I/O device on most computers, requests for disk I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the disk to be referenced, in the form of a logical block number.
3. Disk scheduling algorithms can Improve the effective bandwidth, the average response time and the variance in response time. Algorithms such as SSTF, SCAN, C-SCAN, LOOK and C-LOOK are designed to make such improvements by strategies for disk queue ordering.
4. Performance can be harmed by external fragmentation, some systems have utilities that scan the file system to identify fragmented files. They then move blocks around to decrease the fragmentation. Defragmenting a badly fragmented file system can significantly improve the performance, but the system may have reduced performance while the defragmentation is in progress. The operating system manages the disk blocks. First, a disk must be low level formatted to create the sectors, the raw hardware new disk usually come preformatted. Then the disk is partitioned and file systems created and boot blocks are allocated to store the system bootstrap program. Finally, when a block is corrupted, the system must have a way to lock out that block or to replace it logically with a spare.

   Because an efficient swap space is a key to good performance. Systems usually bypass the full system and use raw disk access for paging I/O. Some systems dedicate a raw disk partition to swap space and others, use a file writing the file system instead, other system allow the user or system administer to makes the decision by providing both options.
5. The write ahead log scheme requires the availability of stable storage. To implement such storage, we need to replicate the needed information on multiple non-volatile storage devices (usually disks) with independent failure modes. We also need to update the information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.
6. Because of the amount of storage required on large systems disk are frequently mode redundant via RAID algorithm. These algorithms allow more than one disk to used for a given operation and allow continued operation and even automatic recovery in the face of a disk failure. RAID algorithms are organized into different levels where each level provides some combination of reliability and high transfer rates.
7. Disks may be attached to a computer system by one of two ways : (i) using the local I/O parts on the host computer (ii) using a network connection such as storage area networks.

8. Tertiary storage is built from disk and tape drives that use removable media. Many different technologies are available, including magnetic tape, removable magnetic and magneto-optic disks and optical disks for removable disks. The operating system generally provides the full service of a file system interface, including space management and request queue scheduling. For many operating systems the name of a file on a removable cartridge is a combination of a drive name and a file name within that drive. This conversation is simpler but potentially more confusing than using a name, that identifies a specific cartridge.
9. For tapes, the operating system generally just provides a raw interface. Many operating system have no built in support for jukeboxes. Jukebox support can be provided by a device driver or by a privileged application designed for backups or for HSM.
10. Three important-aspects of performance are bandwidth, latency and reliability. A wide variety of bandwidth is applicable for both disks and tapes but the random- access latency for a tape is generally much slower than that for a disk. Switching cartridges in a jukebox is also relatively slow, because a jukebox has a low ratio of device to cartridges. Reading a large fraction of the data in a jukebox can take a long time. Optical media, which protect the sensitive layer by a transparent coating, are generally more robust than magnetic media, which expose the magnetic material to a greater possibility of physical damage.

FILE SYSTEMS

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

The second problem with keeping information within a process address space is that when the process terminates, the information is lost.

The third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time.

Essential requirements for long-term information storage:

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

The usual solution to all these problems is to store information on disks and other external media in units called **files.**

1. A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (fined or variable length) or a more complex data item. The operating system may specifically support various record types or may leave that support to the application programs.
2. Each device in a file system keeps a volume table of contents or device directory listing the location of the files on the device. In addition, it is useful to create directories to allow files to be organized. A single level directory in a multi-user system causes naming problems since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user. Each user has her own directory, containing her own files. The directory lists the files by name and includes such information as the file's location on the disk, length type, owner, time of creation, time of last use and so on.
3. The natural generalization of a two level directory is a tree structured directory. A tree structured directory allows a user to create subdirectories to reorganize his files. A cyclic-graph directory structures allow subdirectories and files to be shared, but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories, but sometimes requires garbage collection to recover unused disk space.
4. Disks are segmented into one or more partitions, each containing a file system or left 'row'. File systems may be mounted into the system's naming structures to make them available. The naming scheme varies by operating system. Once mounted, the files within the partition are available for user. File systems may be unmounted to disable access or for maintenance.
5. File sharing depends on the semantics provided by the system. Files may have multiple readers, multiple writers or limits on the sharing. Distributed file systems allow client host to mount partitions or directories from servers as long as they can access each other across a network remote file systems have challenger in reliability, performance and security

Distributed information systems maintain user host and access information such that clients and servers share state information to manage user and access.

6. Since files are the main information-storage mechanism in most computer systems, file protection is naked. Access to files can be controlled separately for each type of access-read write, execute, append, delete, list directory and so on. File protection can be provided by passwords, by access lists or by special adhoc techniques.
7. The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk- physical disks may be segmented into partitions to control media user and to allow multiple possible varying file system per spindle. These file systems are mounted onto a logical file system architecture to make them available for use. File systems are often implemented in a layered or module structure. The lower levels deal with the physical properties of storage devices, upper levels deal with symbolic file concepts into physical device properties.
8. Every file system type can have different structure and algorithms. A VFS layer allows the upper layers to deal with each file system type uniformly. Even remote file system can be integrated into the systems directory structure and acted on by standard system calls via the VFS interface. A various files can be allocated space on the disk in three ways: through contiguous, linked or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block.
9. These algorithms can be optimized in many ways. Contiguous space may be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.
10. Free-space allocation methods also influence the efficiency of use of disk space, the performance of the file system and the reliability of secondary storage. The methods used include bit vectors and the FAT, which places the linked list in one contiguous area.
11. The directory management routines must consider efficiency performance and reliability. A hash table is the most frequently used method which is fast and efficient.
12. Network file systems, such as NFS, use client server methodology to allow users to access files and directories from remote machines as if they were on local file systems. System calls on the client are translated into network protocols and retranslated into file system operations on the server. Networking and multiple client access create challenges in the areas of data consistency and performance.

File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in fig.7.

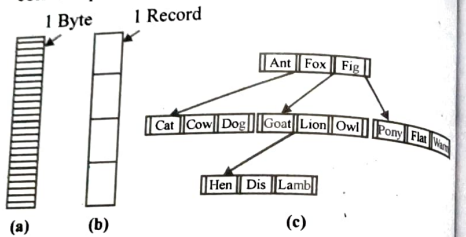


Fig. 6 : Three kinds of files, (a) Byte sequence, (b) Record sequence (c) Tree

The file in fig.7(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

The file in fig.7(b) is a structured sequence of files known as records and in fig.7(c), the file is arranged in form of tree.

File Type

Many operating systems support several types of files. UNIX and Windows, for example, have regular files and directories. UNIX also has charaeecter and block special files.Fourteen **Regular files** are the ones that contain user information. All the files of fig.8 are regular files. **Directories** are system files for maintaining the structure of the file system. **Character special files** are related to input/output and used to model serial I/O devices such as terminals, printers and networks. **Block special files** are used to model disks.

Regular files are generally either ASCII files or binary files.

1 INTRODUCTION AND HISTORY OF OPERATING SYSTEMS

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***Consider the following set of processes, with the arrival times and the CPU burst times given in milliseconds.***

Process	*Arrival Time*	*Burst Time*
P1	*0*	*5*
P2	*1*	*3*
P3	*2*	*3*
P4	*4*	*1*

What is the average turn around time for these processes with the preemptive shortest remaining process time first algorithm? *[R.T.U. 2016]*

Ans. For this, we need turnaround time, which is as follows:

Sequence of the process execution:

P1 P2 P4 P3 P1

0 1 4 5 8

Process	Arrival Time	Burst Time	Turnaround Time
P1	0	5	12
P2	1	3	3
P3	2	3	6
P4	4	1	1

Therefore, Average waiting time is

$= (12 + 3 + 6 + 1)/4$

$= 5.5$

Q.2 ***Write difference between multiprogramming and multiprocessing.***

Ans. Diffrence between multiprogramming and multi-processing

S.No.	Multiprogramming	Multitasking	Multiprocessing
1.	Single CPU is decides its time between more than one job.	Any system that run more than one application program onetime.	Multiple CPU perform more than one job at a time.
2.	Time sharing system application.	Resource management.	Main frame and super mini computers.

Q.3 ***Define long term scheduling.***

Ans. Long Term Scheduling : Which determines which programs are admitted to the system for execution and when, and which ones should be exited.

Q.4 ***Write key features of monolithic kernel.***

Ans. The key features of the monolithic kernels are :

- Monolithic kernel interacts directly with the hardwa
- Monolithic kernel can be optimized for particu hardware architectural. Monolithic kernel is not v portable.

Q.5 ***What do you mean by process.***

Ans. Process : Process is an operation which takes given instructions and performs the manipulation a code. It is program in execution.

PART-B

Q.6 *What is the need of BIOS ? Explain Boot strap loader also.* *[R.T.U. 2018, 2015]*

Ans. Need of BIOS : Computers are run by Operating Systems (OS). They are hosted in RAM memory, which is volatile, i.e., it loses its content when the power is turned off.

The BIOS helps the pc to turned on and start. It is a very small program, hosted on Read-Only-Memory (ROM) which is non-volatile, i.e., it does not vanish when the power is turned off. It is automatically loaded onto the Computer from the ROM by special circuitry, so that the Computer can start its boot-up process.

Since the amount of ROM memory is small, it is a small program, which can do a limited number of things, primarily the following three:

1. It performs a self-test.
2. It checks that hardware peripherals (disk, video, keyboard, etc.) work correctly, and initializes them.
3. Determines a list of places where the Bootstrap loader, a more advanced stage for the initialization, might reside (hard drive, a cd-rom disk, a USB peripheral device, the network, etc.), and tries to pass control to this new stage. If it succeeds, the start-up process continues, otherwise it halts with error messages.

A bootstrap loader is a computer program that loads an operating system or some other system software for the computer after completion of the power-on self-tests; i.e., it is the loader for the operating system itself. A boot loader is loaded into main memory from persistent memory, such as a hard disk drive. The bootstrap loader then loads and executes the processes that finalize the boot.

Boot Strap Loader : The operating system has to be loaded in memory when a computer's power is switched on. It typically involves loading of several programs in memory. Since, the computer's memory does not contain any programs or data at this time, not even an absolute loader, the task of loading the operating system is performed by a special-purpose loader called the *bootstrap loader.*

The bootstrap loader is a tiny program that can fit into a single record on a floppy or hard disk. Recall that an absolute loader loads a program and passes control to it for execution. The bootstrap loader exploits this scheme in its operation. The computer is configured such that when its power is switched on, its hardware loads a special record from a floppy or hard disk that contains the bootstrap loader and transfers control to it for execution. When the bootstrap loader obtains control, it loads a more capable loader in memory and passes control to it. This loader loads the initial set of components of the operating system which load more components, and so on until the complete operating system has been loaded in memory. This scheme is called *bootstrap loading* because of the legend of the man who raised himself to the heaven by using his own bootstraps.

Q.7 *What do you mean by processor scheduling? Explain the various levels of scheduling.* *[R.T.U. 2014]*

OR

Describe the difference between short term, medium term and long term scheduling? *[R.T.U. 2014]*

Ans. Process Scheduling: This refers to the process by which processor determines which job (task) should be run on the computer at which time. Without scheduling the processor would give attention to jobs based on when they arrived in the queue, which is usually not optimal. As part of the scheduling, the processor gives a priority level to different processes running on the machine. When two processes are requesting service at the same time, the processor performs the jobs for the one with the higher priority.

Levels of Scheduling : In operating systems the processor scheduling subsystem operates on three levels, differentiated by the time scale at which they perform their operations. In this sense we have :

- **Long Term Scheduling :** Which determines which programs are admitted to the system for execution and when, and which ones should be exited.
- **Medium Term Scheduling :** Which determines when processes are to be suspended and resumed.
- **Short Term Scheduling (or Dispatching):** Which determines which of the ready processes can have CPU resources, and for how long.

Taking into account the states of a process, and the time scale at which state transition occur, we can immediately recognize that

- Dispatching affects processes
  - running
  - ready
  - blocked
- The medium term scheduling affects processes
  - ready-suspended
  - block-suspended
- The long term scheduling affects processes
  - new
  - exited

Long Term Scheduling : Long term scheduling controls the degree of multiprogramming in multitasking systems, following certain policies to decide whether the system can honor a new job submission or, if more than one job is submitted, which of them should be selected. The need for some form of compromise between degree of multiprogramming and throughput seems evident, especially when one considers interactive systems. The higher the number of processes, the smaller the time each of them may control CPU for, if a fair share of responsiveness is to be given to all processes. A very high number of processes causes waste of CPU time for system housekeeping chores. However, the number of active processes should be high enough to keep the CPU busy servicing the payload *(i.e.* the user processes) as much as possible, by ensuring that-on average-there always be a sufficient number of processes not waiting for I/O.

Medium Term Scheduling : Medium term scheduling is essentially concerned with memory management, hence it's very often designed as a part of the memory management subsystem of an OS. Its efficient interaction with the short term scheduler is essential for system performances, especially in virtual memory systems. This is the reason why in paged system the pager process is usually run at a very high (dispatching) priority level.

Short Term Scheduling: Short term scheduling concerns with the allocation of CPU time to processes in order to meet some predefined system performance objectives. The definition of these objectives (scheduling policy) is an overall system design issue, and determines the "character" of the operating system from the user's point of view, giving rise to the traditional distinctions among "multi-purpose, time shared", "batch production", "real-time" systems, and so on.

Q.8 *What do you understand by semaphores? Can it be useful to solve reader-writer problem? Explain.* *[R.T.U. 2018, 2015]*

Ans. Semaphore : A semaphore, in its most basic form, is a protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphores represent multiple resources, while binary semaphores, as the name implies, represents two possible states (generally 0 or 1; locked or unlocked).

A semaphore can only be accessed using the following operations: wait() and signal(). wait() is called when a process wants access to a resource. If the semaphore is greater than 0, then the process can take that resource. If the semaphore is 0, that is the resource isn't available, that process must wait until it becomes available. signal() is called when a process is done using a resource.

Yes, semaphores can be used to solve the reader-writer problem with the following implementation.

Conditions:

1. No reader will be kept waiting unless a writer has the object.
2. Writing is performed ASAP - i.e. writers have precedence over readers.

The reader processes share the semaphores mutex and wrt and the integer readcount. The semaphore wrt is also shared with the writer processes.

Mutex and wrt are each initialized to 1, and readcount is initialized to 0.

Writer Process

```
wait(wrt);
    /*writing is performed*/
signal(wrt);
```

Reader Process

```
wait(mutex);
    readcount := readcount + 1;
    if readcount = 1 then wait(wrt);
signal(mutex);
    /*reading is performed*/
wait(mutex);
    readcount := readcount - 1;
    if readcount = 0 then signal(wrt);
signal(mutex);
```

Q.9 *What are different algorithmic solutions of critical section problem? Explain.* *[R.T.U. 2018, 2015]*

Ans. Solutions to the Critical Section Problem

Solution to the Critical Section Problem must meet three conditions :

1. Mutual exclusion : If process Pi is executing in its critical section, no other process is executing in its critical section

2. Progress : If no process is executing in its critical section and there exists some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision, of which will enter its critical section next, and this decision cannot be postponed indefinitely

(a) If no process is in critical section, can decide quickly who enters

(b) Only one process can enter the critical section so in practice, others are put on the queue

3. Bounded waiting : There must exist a bound 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

(a) The wait is the time from when a process makes a request to enter its critical section until that request is granted

(b) In practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue)

Q.10 ***What you mean by scheduling? Why scheduling is required? Differentiate the Preemption and Non-Preemption Scheduling?*** *[R.T.U. 2017]*

Ans. Process Scheduling : *Refer to Q.7.*

Importance of CPU Scheduling : The CPU scheduler is an important component of the operating system. Processes must be properly scheduled, or else the system will make inefficient use of its resources. Different operating systems have different scheduling requirements, for example a supercomputer aims to finish as many jobs as it can in the minimum amount of time, but an interactive multi-user system such as a Windows terminal server aims to rapidly switch the CPU between each user in order to give users the "illusion" that they each have their own dedicated CPU.

A scheduler may aim at one or more of many goals, for example: maximizing throughput (the total amount of work completed per time unit); minimizing wait time (time from work becoming enabled until the first point it begins execution on resources); minimizing latency or response time (time from work becoming enabled until it is finished in case of batch activity, or until the system responds and hands the first output to the user in case of interactive activity); or maximizing fairness (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process).

Difference between Preemptive and Non-preemptive Scheduling :

S.No.	Preemptive scheduling	Non preemptive scheduling
1.	It allows a process to be interrupted in the middle of its execution, taking the CPU away and allocating it to another process.	It ensures that a process relinquishes control of the CPU only when it finished with its current CPU burst.
2.	Complex to implement	Simple to implement
3.	Costly	Cost is less.
4.	High overhead	Low overhead.
5.	Process switches form running state to ready and waiting state to ready state.	Process switches from running state to waiting state and process terminates.

Q.11 ***What you mean by process and lifecycle of process. Explain context switching between two processes.*** *[R.T.U. 2017]*

Ans. Process : A key concept in all operating systems is the process. A process is basically a program in execution. Associated with each process is its address space, a list of memory locations from some minimum (usually 0) to some maximum which the process can read and write. The address space contains the executable program, the program's data and its stack. Also associated with each process is some set of registers, including the program counter, stack pointer and other hardware registers and all the other information needed to run the program.

Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time :

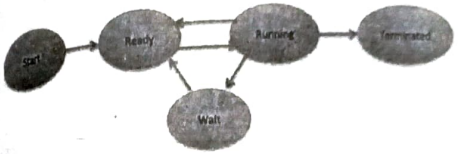
(i) **Start:** This is the initial state when a process is first started/created.

(ii) **Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.

(iii) **Running:** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

(iv) **Waiting:** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

(v) **Terminated or Exit:** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



Context Switching in Processes

Process switching is context switching from one process to a different process. It involves switching out all of the process abstractions and resources in favor of those belonging to a new process. Most notably and expensively, this means switching the memory address space. This includes memory addresses, mappings, page tables, and kernel resources, a relatively expensive operation.

Context Switching in Threads

Thread switching is context switching from one thread to another in the same process. Thread switching is much cheaper as it involves switching out only the abstraction unique to threads, the processor state. Switching processor state (such as the program counter and register contents) is generally very efficient. For the most part, the cost of thread-to-thread switching is about the same as the cost of entering and exiting the kernel.

Consequently, thread switching is significantly faster than process switching.

Q.12 ***What you mean by thread? Explain kernel and user level thread.*** *[R.T.U. 2017]*

Ans. Thread : *A thread is a single sequential flow of control within a program.*

All programmers are familiar with writing sequential programs. You have probably written a program that displays or sorts a list of names, or computes a list of prime numbers. These are sequential programs: each has a beginning, an end, a sequence, and at any given time during the runtime of the program there is a single point of execution.

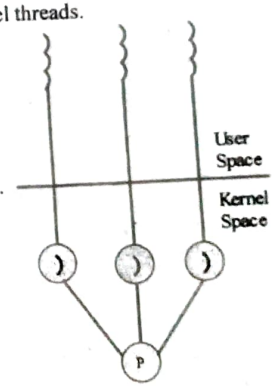
A thread is similar to a sequential program a single thread also has a beginning, an end, a sequence, and at any given time during the runtime of the program, there is a single point of execution. However, a thread itself is not a program, it cannot run on its own, but runs within a program.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than 1 task at a time.

Threads are visible only from within the process, where they share all process resources like address space, open files, and so on. The following state is unique to each thread.

- Thread ID
- Register state (including PC and stack pointer)
- Stack
- Signal mask
- Priority
- Thread-private storage

Kernel Level Threads : Kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems - including Windows XP, Linux, Mac OS X, Solaris, and Tru64 UNIX support kernel threads.



) Kernel Level Thread

Fig. 1 : Kernel-level threads

Advantages

- Since, kernel has full knowledge of all threads scheduler may decide to give more time to

process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages
- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

User level threads : In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating System kernel provides system call to create and manage threads. User threads are supported above the kernel and managed directly by the operating system. User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

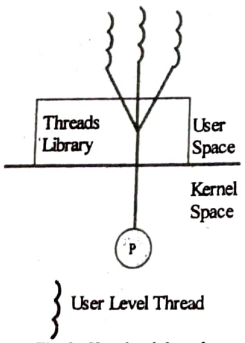


Fig. 2 : User level threads

Advantages
- The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
- User-level threads do not require modification to operating systems.
- **Simple Representation :** Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- **Simple Management :** This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- **Fast and Efficient :** Thread switching is not much more expensive than a procedure call.

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

Q.13 ***What are the five major activities of an operating system with regard to file management?*** *[R.T.U. 2016]*

Ans. Five major activities of an operating system with regard to file management are:

1. **Creating and Deleting Files:** File creation and deletion are fundamental to computer operations. In the former, data cannot be stored in an efficient manner unless arranged in some form of file structure. In the latter, permanent storage would quickly fill up if files were not deleted and the space occupied by them reallocated to new files.
2. **Creating and Deleting Directories:** As a corollary to the need to store data in files, files themselves need to be arranged in directories or folders in order to allow their efficient storage and retrieval. Much like file deletion, unnecessary directories or folders need to be removed in order to keep the system uncluttered.
3. **File Manipulation Instructions:** Since operating systems allow application software to perform file manipulation using symbolic instructions, the operating system itself needs to have a machine-level instruction set in order to interface with the hardware directly. The application's symbolic instructions need to be translated into the machine-level instructions either by an interpreter or by compiling the application code. The operating system contains provisions to manage this machine-level file manipulation.

4. **Mapping to Permanent Storage:** Operating systems need to be able to map files and folders to their physical location on permanent storage in order to be able to store and retrieve them. This will be recorded in some form of disk directory, which varies according to the file system, or systems that the operating system uses. The operating system will include a mechanism to locate the separate file segments where it has divided a file.
5. **Backing up Files:** Files represent a considerable investment in time, intellectual effort and often money as well, thus their loss can have a severe impact. Computer's permanent storage devices generally contain a number of mechanical devices, which can fail, and the storage media itself may degrade. A function of operating systems is to obviate the risk of data loss by backing files up on additional secure and stable media in a redundant system.

Q.14 ***What are the two models of interprocess communication? What are the strengths and weakness of the two approaches?*** *[R.T.U. 2016]*

Ans. Two Models of Interprocess Communication : There are two interprocess communication models given below:

1. Message-passing Model : In this, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for inter computer communication. But the main disadvantage is it can handle only small amounts of data.

2. Shared-Memory Model : In this, processes use shared memory creates and shared memory attaches system calls to create and gain access to regions of memory owned by other processes. Two or more processes can exchange information by reading and writing data in the shared areas. Shared memory allows maximum speed and convenience of communication, since; it can be done at memory speeds when it takes place within a computer problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

Strengths and weakness of these two approaches are given below:

(a) Message Passing Model :

Strengths:
(i) Easier to implement
(ii) Best suited for smaller amount of data

Weakness:
(i) Only suitable for the exchange of small amount of data.
(ii) Communication using message passing is slower than shared memory because of the time involved in connection setup.

(b) Shared Memory Model:

Strengths :
Shared memory communication is faster the message passing model when the processes are on the same machine.

Weakness :
(i) Different processes need to ensure that they are not writing to the same location simultaneously.
(ii) Processes that communicate using shared memory need to address problems of memory protection and synchronization.

Q.15 ***What are the difference between user level threads and kernel level threads under what circumstances is one type better than the other?*** *[R.T.U. 2016]*

Ans. Difference between user level threads and kernel level threads :

Yes, Kernel level and user level threads are different.

S. No.	User Level Thread	Kernel Level Thread
1.	User thread are implemented by users.	Kernel threads are implemented by OS.
2.	OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
3.	Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
4.	Context switch time is less.	Context switch time is more.
5.	Context switch requires no hardware support.	Hardware support is needed.
6.	If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
7.	Example : Java thread, POSIX threads.	Example : Window Solaris.

Circumstances is one type better than the other

A user thread is more appropriate for low-le tasks, whereas a kernel thread is better for high-prio tasks that should get high priority to system resource

Q.16 ***Compose FCFS, SJF and Round-Robin scheduling algorithms by computing average waiting time. There are 5 processes with CPU burst time as 10, 5, 17, 25, 6 and arrival times are 0, 1, 0, 2, 7 units. Assume time quantum for Round Robin scheduling as 5 units.*** *[R.T.U. 2015]*

Ans. Let the jobs be J10, J5, J17, J25, J6 and the wait time be denoted by w

FCFS Scheduling

00: J10	w = 0
10: J17	w = 10
27: J5	w = 26
32: J25	w = 30
57: J6	w = 50
Avg	w = 116/5=23.2

SJF Scheduling

00: J10	w = 0
10: J5	w = 10
15: J6	w = 15
21: J17	w = 21
38: J25	w = 36
Avg	w = 82/5=16.4

Round-robin scheduling

00: J10	w = 0
05: J17	w = 5
10: J5	w = 9
15: J25	w = 13
20: J6	w = 13
25: J10	w = 20
30: J17	w = 20
35: J25	w = 15
40: J6	w = 15
41: J17	w = 6
46: J25	w = 6
51: J17	w = 5
53: J25	w = 2
Avg	w = 129/5=25.8

Q.17 ***What are the benefits of threads? Explain context switching of processes and threads.*** *[R.T.U. 2015]*

Ans. Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others. This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.

The benefits of executing tasks in modular threads instead of independent processes are as follows:

1. Takes less time to create a new thread than a process.
2. Less time to terminate a thread than a process.
3. Less time to switch between two threads within the same process.
4. Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.
5. Responsiveness - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
6. Resource Sharing - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
7. Scalability, i.e, Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

Context Switching in Processes : *Refer to Q.11.*
Context Switching in Threads : *Refer to Q.11*

PART-C

Q.18 ***How an operating system works as a resource manager and virtual machine?***
[R.T.U. 2018, 2014]
[Note : Vertical actually should be taken as virtual.]

Ans. Operating System : An operating system (OS) is a software that manages the computer hardware and provide an environment in which a user can execute programs in a convenient and efficient manner.

An operating system acts as an intermediary between the user of a computer and the computer hardware as shown in below fig. 1 :

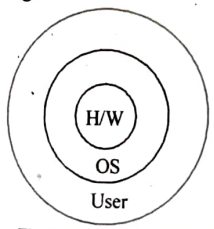


Fig. 1 : OS as an intermediary

Operating System as a Resource Manager

The concept of the operating system as primarily providing its users with a convenient interface 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 computer consists 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 processor, memories and I/O devices among the various programs competing for them.

When a computer (or network) has multiple users, the need for managing and protecting the memory, I/O 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, database, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of who is using which resource, to grant resource requests, so account for usage and to mediate conflicting requests from different programs and users.

Resource management includes *multiplexing (Sharing)* resources in two ways: in time and in space. When a 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, the operating system first allocates the CPU to one program, then after it has run long enough, another one gets to use the CPU then another and then 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. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing. Instead of the customer taking turns, each 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. Another resource that is space multiplexed is the (hard) disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system resource management disk.

Operating System as Virtual Machines

Virtual-memory techniques as, an operating system can create the illusion that a process has its own processor with its own (virtual) memory. Of course, normally, the process has additional features, such as system calls and a file system, that are not provided by the bare hardware. The virtual-machine approach, on the other hand, does not provide any additional functionality, but rather provides an interface that is identical to the underlaying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (fig. 2).

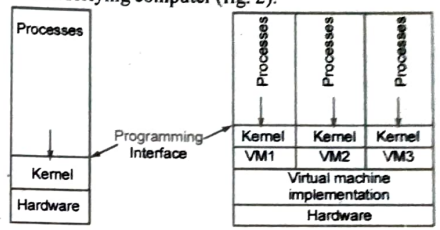


Fig. 2 : System models (a) Non-virtual machine (b) Virtual machine

The physical computer shares resources to create the virtual machines. CPU scheduling can share out the CPU to create the appearance that users have their own processors. Spooling and a file system can provide virtual card readers and virtual line printers. A normal user time-sharing terminal provides the function of the virtual-machine operator's console.

A major difficulty with the virtual-machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines. It cannot allocate a disk drive to each virtual machine. The virtual-machine software itself will need substantial disk space to provide virtual memory. The solution is to provide virtual disks, which are identical in all respects except size - termed minidisks in IBM's VM operating system. The system implements each minidisk by allocating as many tracks on the physical disks as the minidisk needs. Obviously, the sum of the sizes of all minidisk must be smaller than the size of the physical disk space available.

Users thus are given their own virtual machines. They can then run any of the operating systems or software packages that are available on the underlying machine. For the IBM VM system, a user normally runs CMS-a single-user interactive operating system. The virtual machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but it

does not need to consider any user-support software. This arrangement may provide a useful partitioning into two smaller pieces of the problem of designing a multiuser interactive system.

Q.19 ***Explain the architecture of an operating system.*** ***[R.T.U. 2018, 2012]***

OR

Explain the architecture of operating system with neat and clean diagram. ***[R.T.U. 2017]***

Ans. Architecture of Operating System

There are different architectures of the operating system are used in computer world.

Monolithic Architecture for Operating System

Fig.1 shows the monolithic architecture of an operating system. It is the oldest architecture used for developing an operating system.

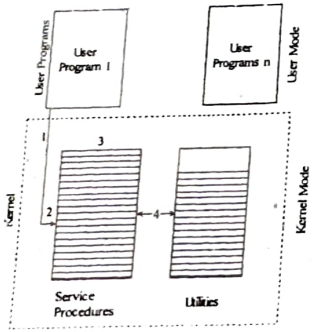


Fig. 1 : Monolithic architecture

The key features of the monolithic kernels are :

- Monolithic kernel interacts directly with the hardware.
- Monolithic kernel can be optimized for particular hardware architectural. Monolithic kernel is not very portable.
- Monolithic kernel gets loaded completely into memory at boot time.
- Most system calls are made by trapping to the kernel, having the work performed there, and having the kernel return the desired result to the user process.
- System call 1. (User → Kernel Mode) 2. Check parameters 3. Call service routine 4. Service routine call utilities Reschedule/Return to user.

Layered Architecture of Operating System

Dijkstra introduced the layered architecture for operating systems.

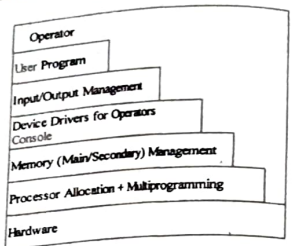


Fig. 2 : Layered architecture

The main benefit of this approach is modularity. The layers are selected such that each uses services and functions of its lower layers. All the benefits of modular programming can be achieved with this architecture. The layered architecture of the operating system has been shown in fig.2. At the lowest level, it has hardware, layer 1 has processor allocation and support for multiprogramming, layer 2 implements memory management, layer 3 contains the device drivers for operator's console, layer 4 contains input-output buffering support.

Virtual Machine Architecture of Operating System

Virtual machine architecture is sometimes, also known as enterprise system architecture. Virtual machine operating system for IBM systems is the best example of Virtual machine concept because IBM pioneered the work in this area. The VM operating system is built on the virtual storage concept to subdivide a single computer system into multiple, virtual computer systems, each with its own processor, disk storage, tape storage, and other input-output devices. That is, VM uses software techniques to make a single computer appear to be multiple smaller computer systems.

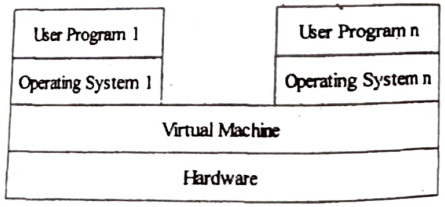


Fig. 3 : Virtual machine arhitecture

Virtual machine has two main components :

(i) The control program (CP) controls the real machine.

(ii) The conversational monitor system (CMS) controls virtual machine.

Micro-Kernel Architecture of Operating System

A very modern architecture is the micro-kernel architecture. This architecture strives to take out of the kernel as much functionality as possible, so as to limit the code executed in privileged mode and to allow easy modifications and extensions. It does so by moving many operating system services from the kernel into the "user space". Thus, making kernel as small as possible and therefore, it is called Micro Kernel.

This is one advantge as it always stays in the main memory and thus consumes less memory of the system.

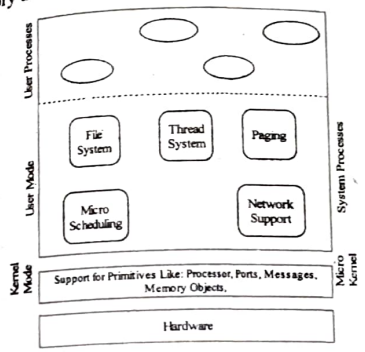


Fig. 4 : Micro kernal architecture

Exokernel Architecture of Operating System

Exokernel is a further extension of the microkernel approach where the "kernel" is almost devoid of functionality; it merely passes requests for resources to "user space" libraries.

This would mean that (for instance) requests for file access by one process would be passed by the kernel to the library that is directly responsible for managing file systems. Initial reports are that, this in particular result in significant performance improvements, as it does not force data to even pass through kernel data structures.

Table below shows a table that compar is one of these architectures in brief.

Table: Comparison of kernel implementation of various operating system architectures.

	Monolithic	Micro Kernel	Exo Kernel
Implementation of operating system abstractions	All are implemented in kernel space	Only lower level operating system facilities are implemented in kernel	Nothing is implemented in kernel space

Q.20 ***Describe the solution of Dining-Philosophers problem.*** ***[R.T.U. 2018]***

OR

What is dining-philosophers problem? Explain the solution of this problem by using a suitable example. ***[R.T.U. 2014, 2013]***

Ans. The Dining Philosophers Problem

The Dining Philosopher's Problem is summarized as five philosophers sitting around a circular table doing one of two things: eating spaghetti or thinking. While eating, they are not thinking, and while thinking, they are not eating.

Each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his immediate left or right as shown in the fig. 1.

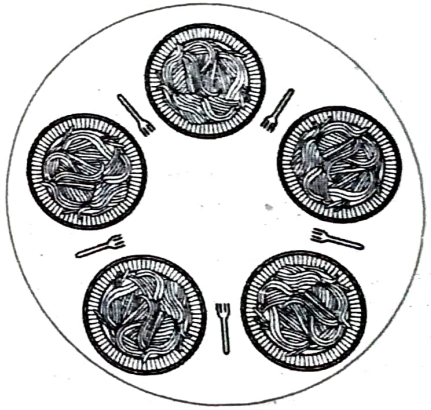


Fig. 1 : Five Philosophers sitting around a circular table

Whenever a philosopher is hungry he starts eating spaghetti having forks in both hands. He eats without releasing his forks.

The philosophers never speak to each other, which create a possibility of **deadlock** since, every philosopher holds a left fork and waits continuously for a right fork (or vice versa). To avoid the problem of deadlock we can implement the rule that if a philosopher is waiting for the right fork (to be free) for more than five minutes, he put down the left fork. He then waits for five more minutes to make the new attempt. This scheme eliminates the possibility of deadlock but can cause another problem. Consider a scene if all five philosophers appear in the

dining room exactly at the same time and each picks up their left fork at the same time. Then all philosophers will wait for five minutes (for the right fork to be free) then they all put their forks down and wait for another five minutes before they all pick them up again. This result in continuous waiting of all philosophers and cause a problem called **'Starvation'**.

Lack of available forks refers to the lacking of shared resources.

Solution using Semaphores :

The simplest solution to this problem is to represent each of the fork with a semaphore and implement take_fork () and put_fork() functions.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. This must be noted that each process will execute philosopher () as its main function but take_fork(), Eat() and put_fork() are separate procedure for each process.

```
#define N 5

Philosopher() {
  while (TRUE) {
    Think();
    take_fork(i)
    take_fork((i+1)%N);
    Eat();
    put_fork(i);
    put_fork((i+1)%N);
  }
```

put_forks(i)

take_forks(i)

Fig. 2 : Semaphore implementation of a philosopher

Solution using Mutexes : Another way of solving Dining Philosophers problem is by using mutexes. Before starting to acquire forks, a philosopher would do a down on mutex. After replacing the forks, he would do an up to the mutex. This solution allows two philosophers to eat at a same time.

(i) Picking up forks

```
int state [N]
semaphore mutex = 1
semaphore sem [i]
```

```
take_forks (int i) {
down (mutex);
state [i] = HUNGRY;
test (i);
up (mutex);
down (sem[i]);
}
```

```
test (int i) {
if (state [1] == HUNGRY &&
 state [LEFT]! = EATING &&
 state [RIGHT] ! = EATING) {
  state [i] = EATING;
  up (sem [i]);
 }
}
```

(ii) Putting down forks

```
int state [N]
semaphore mutex = 1
semaphore sem [i]
```

```
put_forks (int i) {
down (mutex);
state [i] = THINKING;
test (LEFT);
test (RIGHT;
up (mutex);
}
```

```
test (int i) {
if (state [1] == HUNGRY &&
 state [LEFT]! = EATING &&
 state [RIGHT] ! = EATING) {
  state [i] = EATING;
  up (sem [i]);
 }
}
```

Deadlock, is such a problem, can be removed by imposing several constraints :

1. Maximum of four philosophers are allowed to sit simultaneously on the dining table.
2. Allow a philosopher to pick the forks if both of them on either side are available.
3. Let the odd philosopher picks up their left fork before their right fork, whereas even philosophers pick their right fork before their left fork.

Q.21 ***(a) Consider the following set of processes with arrival time and CPU burst time given in ms.***

Process	*Arrival time*	*Burst time*
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

What is the average waiting time for these processes with preemptive SJF scheduling? *[R.T.U. 2018, 2014, 2013]*

(b) Consider the following four processes, with the length of the CPU burst time given in milliseconds.

Process	*Burst time (ms)*	*Arrival time (ms)*
P0	*15*	*0.0*
P1	*20*	*1.0*
P2	*3*	*2.0*
P3	*7*	*2.0*

Consider the Shortest Remaining Time First (SRTF), Round Robin (RR) (Quantum = 5ms) scheduling algorithms. Illustrate the scheduling using Gantt chart. Which algorithm will give the minimum average waiting time? *[R.T.U. 2017]*

Ans.(a)

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Gantt's chart for the system using preemptive SJF scheduling.

P_1	P_2	P_4	P_1	P_3
0	1	5	10	17	26

Process P_1 is started at time 0, it is only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milli seconds) is larger than the time required by process P_2 (4 milli seconds), so process P_1 is preempted, and process P_2 is scheduled.

The average waiting time

Process	Arrival time	Burst time	Completing time	Waiting time
P_1	0	8	17	9
P_2	1	4	5	0
P_3	2	9	26	15
P_4	3	5	10	2

Waiting time of the process.

$(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)$

or $9 + 0 + 15 + 2 = 26$

The average waiting time for the system is

$26/4 = 6.5$

Ans.(b)

A. Using Shortest Remaining Time First (SRTF): steps of Gantt chart preparation:

(i) At time = 0.0
Aavailable process = P0
∴ Scheduled process = P0

(ii) At time = 1.0
Available process = P0, P1
Check remaining burst time P0 = 14, P1 = 20
P0 less time
∴ Scheduled process = P0

(iii) At time = 2.0
Available process = P0, P1, P2, P3
Check remaining burst time P0 = 13, P1 = 20, P2 = 3, P3 = 7
∴ Scheduled process = P2

(iv) At time = 5.0, P2 Completes
Available process = P0, P1, P3
Remaining B.I. P0 = 13, P1 = 20, P3 = 7
∴ Scheduled process = P3

(v) At time = 12.0, P3 Completes
Available process = P0, P1
Remaining B.I. P0 = 13, P1 = 20
∴ Scheduled process = P0

(vi) At time = 25.0, P0 Completes
Available process = P1
∴ Scheduled process = P1

(vii) At time = 45.0, P1 Completes
Process stops

Gantt Chart

P0	P0	P2	P3	P0	P1
0	1.0	2.0	5.0	12.0	25.0	45.0

SRTF Scheduling

Average Waiting Time = $(WT_{P0} + WT_{P1} + WT_{P2} + WT_{P3})/4$
$= (10 + 24 + 0 + 3)/4$
$= 37/4$
$= 9.25$ ms

B. Using Round Robin Scheduling :
Quentum = 5ms
Each process will get max 5 ms per round.

(i) At time = 0.0
Aavailable process = P0
It is scheduled for 5 ms

(ii) At time = 5.0
Available process = P0, P1, P2, P3
Since, P0 was scheduled earlier
Process scheduled = P1 for 5 ms

Similarly all process are scheduled in round robin manner till time = 45 ms

Gantt Chart

P0	P1	P2	P3	P0	P1	P3	P0	P1	P1
0	5	10	13	18	23	28	30	35	40	45

$WT_{P0} = (18 - 5) + (30 - 23) = 13 + 7 = 20$ ms
$WT_{P1} = (5 - 1) + (23 - 10) + (35 - 28) = 4 + 13 + 7$
$= 24$ ms
$WT_{P2} = (10 - 2) = 8$ ms
$WT_{P3} = (13 - 2) + (28 - 18) = 11 + 10 = 21$ ms
Average waiting time = $(20 + 24 + 8 + 21)/4$
$= 73/4$
$= 18.25$ ms
Average waiting time of SRTF is minimum at 9.25 m

Q.22 ***Write short notes on the following:***

(i) Fair share scheduling

(ii) Race condition

(iii) Critical section

(iv) Semaphore and mutex *[R.T.U. 2017]*

Ans.(i) Fair Share Scheduling : Fair-share scheduling is a scheduling strategy for computer operating systems in which the CPU usage is equally distributed among system users or groups, as opposed to equal distribution among processes.

For example, if four users (A,B,C,D) are concurrently executing one process each, the scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole (100% / 4 = 25%). If user B starts a second process, each user will still receive 25% of the total cycles, but each of user B's processes will now use 12.5%. On the other hand, if a new user starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole (100% / 5 = 20%).

Another layer of abstraction allows us to partition users into groups, and apply the fair share algorithm to the groups as well. In this case, the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user. For example, if there are three groups (1,2,3) containing three, two, and four users respectively, the available CPU cycles will be distributed as follows:

100% / 3 groups = 33.3% per group

Group 1: (33.3% / 3 users) = 11.1% per user

Group 2: (33.3% / 2 users) = 16.7% per user

Group 3: (33.3% / 4 users) = 8.3% per user

(ii) Race condition : In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file; the location of the shared memory does not change the nature of the communication or the problems that arise. To see how interprocess communication works in practice, let us consider a simple but common example; a print spooler. When a process wants to print a file, it enters the file name in a special spooler directory. Another process, the printer daemon, periodically checks to see if there are any files to be printed and if there are, it prints them and then removes their names from the directory.

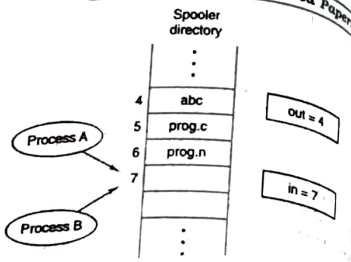


Fig. : Two processes want to access shared memory at the same time

Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, out which points to the next file to be printed and in which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes A and B decide they want to queue a file for printing. This situation is shown in fig.

Situations where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a rare while something weird and unexplaied happens.

(iii) Critical section :

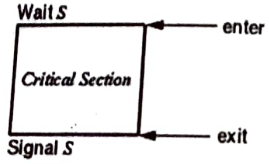


Fig.

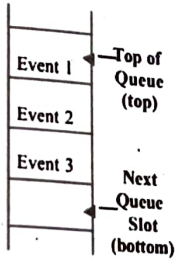
The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the *Critical Section*. To avoid race conditions and flawed results,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

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

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

Critical Section Example

```
Enqueue( Data )
{
   EnterCriticalSection(ENQ)
      Q[bottom] = Data
      bottom = bottom + 1
   LeaveCriticalSection(ENQ)
}
Dequeue( )
{
   EnterCriticalSection(DEQ)
      Data = Q[top]
      top = top + 1
   LeaveCriticalSection(DEQ)
   return Data
}
```



Does the above code avoid race conditons?

If a process tries to enter a named critical section, it will :

- Blocks : Critical section in use
- Enter : Critical section not in use

(iv) Semaphore : *Refer to Q.8.*

Mutex : Mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

There is an ambiguity between binary semaphore and mutex. But the purpose of mutex and semaphore are different.

Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is signaling mechanism ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an "Interrupt Service Routine (ISR)" signals the call processing task to wakeup.

Q.23 ***What is operating system? Explain its types and services provided by operating system in detail.*** *[R.T.U. 2017]*

OR

What are the different services provided by the operating system? Explain all of them in detail? *[R.T.U. 2016]*

Ans. Operating System : *Refer to Q.18.*

Types of Operating Systems : Operating system can be classified into following categories:

(i) Single-user Single-tasking Operating system
(ii) Batch Operating system
(iii) Multi-user Operating system
(iv) Multi-tasking Operating system
(v) Real-time Operating system
(vi) Network Operating system
(vii) Distributed Operating system

(i) Single-user Single-tasking Operating System: An operating system that allows a single user to work on a computer at a time and can execute a single job at a time is known as singe-user single-tasking operating system. For example, MS-DOS is a single-user single-tasking operating system because you can open and run only one application at a time in MS-DOS.

(ii) Batch Operating System: A single user operating system that can execute various types of jobs in batches but one after another. In a batch-operating environment, users submit their programs and data to the operator and the operator groups the similar jobs, and then loads them (all groups of programs along with their relevant data) simultaneously. When the execution of one program for performing a similar kind of jobs is over, a new program is loaded for the execution by the operating system. Batch operating system is suitable for such applications that require long computation time without user intervention. Some examples of such applications are payroll processing, forecasting, statistical analysis, etc.

(iii) Multi-user Operating System: It permits simultaneous access to a computer system through two or more terminals for users. UNIX is an example of multi-

user operating system. It allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.

(iv) Multi-tasking Operating System : It is also called Multiprocessing Operating system. A multitasking operating system is able to handle more than one processor as the jobs have to be executed on more than one processor (CPUs). The running state of program is called a process or task. A multitasking operating system supports two or more processes to execute simultaneously.

A multiuser operating system allows simultaneous access to a computer system through one or more terminals. Although frequency associated with multiprogramming, multiuser operating system does not imply multiprogramming or multitasking. A dedicated transaction processing system such as railway reservation system that hundreds of terminals under control of a single program is an example of multiuser operating system. Window 98/2000/XP/Vista, OS/2, UNIX, LINUX etc. are examples of multi-tasking operating system.

Time Sharing System : Time sharing is a processor (CPU) management technique. In a time sharing operating system, the CPU is allocated to each user, in sequence, for a small amount of time called time-slice (from 10-100 milliseconds). A time slice is allocated to each user using round-robin scheduling algorithm. As soon as the time slice is over, the CPU is allocated to the next user.

Time sharing system is a form of multiprogrammed operating system which operates is an interactive mode with a quick response time. The user types a request to the computer through a keyboard. The computer processes it and a response (if any) is displayed on the user's terminal. A time sharing system allows many users to simultaneously share the computer resources. Since each action or command in a time-sharing system takes a very small fraction of time, only a little CPU time is needed for each user. As the CPU switches rapidly from one user to another user, each user is given impression that he has his own computer while it is actually one computer shared among many users. Most time sharing systems use time-slice (round robin) scheduling of CPU. In this approach, programs are executed with rotating priority that increases during waiting and drops after the service is granted. In order to prevent a program from monopolizing the processor, a program executing longer than the system defined time-slice is interrupted by the operating system and placed at the end of the queue of waiting program.

Memory management in time sharing system provides for the protection and separation of user programs. Input/Output management feature of time-sharing system must be able to handle multiple users (terminals). However the processing of terminals interrupts is not time critical due to the relative slow speed of terminals and users. As required by most multiuser environment allocation and deallocation of device must be perfomed in a manner that preserves integrity and provides for good performance.

(v) Real-time Operating System: A real-time operating system (RTOS) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft industrial control and scientific research equipment.

A RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time; this requires correct development of the software. The primary objective of real-time system is to provide quick response time. Resource utilisation and user convenience are of lesser concern to real-time system. In order to provide quick response time, most of the time processing remain in primary memory. If a job is not completed within the fixed deadline, this situation is called deadline overrun. A real time operating system must minimise the possible deadline overruns.

An early example of a large-scale real-time operating system was Transaction Processing Facility developed by American Airlines and IBM for the Sabre Airline Reservations System.

(vi) Network Operating System (NOS): A network operating system (NOS) is an operating system which makes it possible for computers to be on a network, and manages the different aspects of the network. Network operating system (NOSs) are designed to support interconnection and communication among computers. Network Operating system provides support for communication, network management functions and administration, multi-user operations and security. Novel's Net ware, Microsoft's Windows NT, UNIX and Linux are examples for network operating sytem. Some examples are Windows for Workgroups, Windows NT, AppleShare, DEDnet, LAN tastc, etc.

(vii) Distributed Operating System: A distributed operating system is one that looks to its users like an ordinary centralised operating system but runs on multiple independent CPUs. The use of multiple processors is invisible to the user. In a true distributed system, users are not aware of where their programs are being run or where their files are residing; they should all be handled automatically and efficiently by the operating system.

Distributed operating systems have many aspects in common with centralised ones but they also differ in certain ways. Distributed operating system, for example often allows programs to run on several processors at the same time, thus requiring more complex processor scheduling (scheduling refers to a set of policies and mechanisms built into the operating systems that controls the order in which the work to be done is completed, algorithms in order to achieve maximum utilisation of CPU's time).

Fault-tolerance is another area in which distributed operating systems are different. Distributed systems are considered to be more reliable than uniprocessor based system. They perform even if certain part of the hardware is malfunctioning. This additional feature, supported by distributed operating system has enormous implications for the operating system.

Operating System Services

Operating system provides certain services to programs and to users of those programs. These services are useful for the users for successful execution of programs and efficient utilization of resources. Series of operating system may differ from one operating system to another. Some common types of services are as follows:

1. User Interface : User interface (UI) may be command line interface (CLI), which uses text commands or graphical user interface (GUI), which uses menus or icons.

2. Operating System as a Resource Manager: Operating system provides certain services to the users and their program. Almost all operating system provides common types of services. These services are useful for the users for successful execution of programs and efficient utilization of resources.

Common Types of Services

1. Execution of Program : This is the first and foremost service that is provided by the operating system to the users. The computing system must execute and terminate all user programs successfully by loading them into main memory. A proper error message must be displayed for the incomplete or abnormally terminated programs.

2. Input/Output Operations : During normal execution of a program user may be required to input data or may need some processed data as output. It is the job of the operating system to obtain data from the input device or from some data files and send the output to a data file or output device.

3. File Management : The user of the computing system may definitely need to create the files on secondary storage to record their work permanently for future use or delete the file that is already in existence. The operating system must provide a mechanism that carefully handles all these file manipulation operations on the respective devices.

4. Communication : In the computing system there can be more number of programs that need to share information among them. The program may be running on a single machine or may be run on several machines. This phenomenon is known as 'Inter Process Communication'. To maintain data security and integrity, an operating system must handle inter process communication.

5. Error Detection : If any resource or device in a computing system comes across any error it should be promptly handled by the operating system. Invalid or improper use of devices by the programs must be reported. The most common types of errors are: memory allocation error, divide by zero error, printer error, power off error etc.

6. Protection : The operating system should provide the services to all users by ensuring the security of access by authenticated users to some important files and data.

7. Accounting : In case of multiple users of the computing system the information of duration and number of resources that are used by each user may be tracked by the operating system for accounting purpose.

8. Resource Sharing and Allocation : When there are more number of users in the computing system or there are many programs competing for limited resources, then the optimal allocation and de-allocation of resource is the responsibility of operating system to ensure increased throughput.

Q.24 ***In-connection with interprocess communication explain the following :***

(i) Race Condition

(ii) Critical Condition

(iii) Sleep and Wake up

(iv) Sleeping Barber's Problem ***[R.T.U. 201***

Ans.(i) Race Condition : *Refer to Q.22(ii).*

(ii) Critical Condition Or Critical Section: *Ref to Q.22(iii).*

(iii) Sleep and Wakeup :

The simplest solution is to create two system calls **sleep( )** and **wakeup( ).**

- by calling **sleep ( )** the calling process is suspended till being woken by other process calling wakeup ( ),
- **wakeup** function called with process number as a single argument.

Example of sleep ( )/wakeup ( ) Usage

Producer-consumer problem-problem of a buffer with limited capacity.

Let two processes share a buffer with limited capacity. Process called producer will put pieces of information in the buffer. Process called consumer will take pieces of information from that buffer.

Let us assume :

Pr = Producer

Co = Consumer

- if Pr is trying to put a message into full buffer, Pr hat to be suspended,
- if Co is trying to take a message from the empty buffer, Co has to be suspended.

Let is *count* variables number of taken positions in the buffer is hold. Let the size of the buffer will be *N*.

Producer : if (count == N) {go to sleep} else {add *msg* and count ++}

Consumer : if (count == 0) {go to sleep} else {take *msg* and count –}

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a wakeup waiting bit to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting but is on, it will be turned off, but the process will stay awake.

(iv) Sleeping Barber Problems : The barber shop has one barber, one barber chair and n chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and fails asleep, as illustrated in figure 1. When a customer arrives, he has to wake up the sleeping barber.

Fig. 1 : The sleeping barber

If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.

Our solution uses three semaphores; (i) customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), (ii) barbers, the number of barbers (0 or 1) who are idle, waiting for customers and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers. It is essentially a copy of customers. The reason for having waiting is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

Our solution is shown in figure 2. When the barber shows up for work in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. The barber then goes to sleep, as shown in figure. He stays asleep until the first customer shows up.

When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released mutex. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases mutex and leaves without a haircut.

If there is an available chair, the customer increments the integer variable, waiting. Then he does an up on the semaphore customers, thus waking up the barber. At this point, the customer and barber are both awake. When the customer releases mutex, the barber grabs it, does some housekeeping and begins the haircut.

When the haircut is over, the customer exists the procedure and leaves the shop. Unlike earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, another haircut is given. If not, the barber goes to sleep.

```
#define CHAIRS 5          /*#chairs for waiting
                            customers*/
typedef int semaphore;    /* use your imagination */
semaphore customers=0;/* # of customers waiting
                            for service*/
semaphore barbers = 0;    /* # of barbers waiting for
                            customers */
semaphore mutex = 1;      /* for mutual exclusion */
int waiting = 0;          /* customers are waiting
                            (not being cut) */
void barber (void)
{
while (TRUE) {
down(&customers);         /* go to sleep if # of
                            customers is 0 *
down (&mutex);            /* acquire access to
                            'waiting'*/
waiting = waiting - 1;    /* decrement count of
                            waiting customers */
up(&barbers);             /* one barber is now ready
                            to cut hair */
up(&mutex);               /* release 'waiting'*/
cut_hair();/* cut hair (outside critical region)*/
}
void customer(void)
{
down (&mutex);            /* enter critical region */
if (waiting <CHAIRS) {    /* if there are no free chairs,
                            leave */
{
waiting = waiting + 1;    /* increment count of
                            waiting customers */
up(&customers);           /* wake up barber if
                            necessary */
up(&mutex);               /* release access to
                            'waiting'*/
down(&barbers);           /* go to sleep if = of free
                            barbers is 0 */
get_haircut ();           /* be seated and be
                            serviced */
}
else{
up(&mutex);               /*shop is full; do not wait */
} }
```

Fig.2 : A solution to the sleeping barber problem

Q.25 (a) ***What is critical section problem? Explain the role of lock variable and TSL instruction in busy waiting.*** *[R.T.U. 2014, 2011]*

(b) ***What are the main functions of an operating system? Explain the types of operating systems in brief.*** *[R.T.U. 2014]*

Ans.(a) The Critical-Section Problem : A critical section is a part of a program which has exclusive access to shared data.

Each process has a segment of code, called a critical region, in which the process may change common variables, update tables, write a file etc. The most important feature of this system is that when one process is operating in a critical region, no other process is allowed to execute in its critical region.

In the past it was possible to implement this technique by using interrupts. By switching off interrupts a process can guarantee itself uninterrupted access to shared data. But this method has some drawbacks.

1. Masking interrupts can be dangerous, since there is always a possibility that important interrupts may be missed.
2. Generally it is not implementable in a multiprocessor environment since interrupts will continue to service for other processors too.

Each process must seek for the permission while entering in the **'critical section'** The section of code implementing this request is **entry section**. The critical section may be followed by **exit section.** The remaining code is **remainder section.**

```
Do {
  Entry section
Critical section
  Exit section
Remainder section
} while (TRUE);
```

Lock Variable : As a second attempt, for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

This idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will als

set the lock to 1, and two processes will be in their critical regions at the same time.

TSL RX, LOCK : (Test and Set Lock) that works as follows:

It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock. The operations of reading the word and storing into it are guaranteed to be indivisible no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

Ans. (b) Operating System Services : *Refer to Q.18*
Types of Operating Systems : *Refer to Q.21*

Q.26 (a) ***What is a process? What is the difference between a program and a process? Explain PCB using a suitable example.***

(b) ***Explain the following :***
(i) Kernel level thread
(ii) Boot strap loader
(iii) Multithreading OS *[R.T.U. May*

Ans. (a) Process : *Refer to Q.11*

Difference between Program and Process

S.No.	Process	Program
1.	Process is a program in execution.	Program is a series of instructions to perform a particular task.
2.	Process is a part of a program.	Program is given as a set of process.
3.	Process is the part where logic of that particular program exists.	In some cases we may divide a problem into numbers of parts. At these times we write a separate logic for each part known as process.
4.	Process is a program in memory.	Program is only a set of instructions.
5.	Process can be program.	Program is very rarely a process.
6.	The process is an operation which takes the given instructions and performs the manipulations as per the code, called execution of instructions. A process is entirely dependent of a program.	A program is a set of instruction that are to perform a designated task.
7.	A process is module that executes modules concurrently. They are separate loadable modules.	The program perform task directly relating to an operation of a user like word processing executing presentation software etc.

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

- **Process State :** The state may be new, ready, running, waiting, halted, and so on.
- **Program Counter :** The counter indicates the address of the next instruction to be executed for this process.
- **CPU Registers :** The vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, 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.

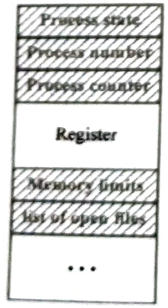


Fig. : Process control block (PCB)

- **CPU Scheduling Information :** This information includes a process priority pointers to scheduling queues, and any other scheduling parameters.
- **Memory Management Information :** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting Information :** This information includes the account 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 input 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.

Ans. (b) (i) Kernel Level Threads : *Refer to Q.12*

(ii) Boot Strap Loader : *Refer to Q.6.*

(iii) Multithreading OS : A multithreading operating system is one that is capable of handling processes and threads at the same time and in which every process is allowed to generate more than one thread. In such an operating system, there must be facilities for thread creation, deletion, switching, etc. Such an operating system allows users to generate more than one request to a process at a time. For example, a browser can be made to search simultaneously for more than one topic, even though there is only one copy of the "browser program" in main memory.

The multiprogramming methodology and technique are essential in the implementation of multithreading. In this new environment, a thread becomes the smallest functional and active object to which CPU (or a PU) is assigned.

Another example of multithreaded OS, an application might be divided into four threads : a user interface thread, a data acquisition thread, network communication, and a logging thread. You can prioritize each of these so that they operate independently. Thus, in multithreaded applications, multiple tasks can progress in parallel with other applications that are running on the system.

Q.27 (a) ***Explain the following :***
(i) Process
(ii) Thread
(iii) Kernel

(b) ***Define Operating System. Explain how operating system acts as a resource manager? Differentiate between Multiprogramming and Multi-processing?*** *[R.T.U. Dec. 2013]*

Ans. (a) (i) Process : *Refer to Q.11*

Process Control Block : *Refer to Q.26(a).*

Process States : When program executes, it changes its state. During whole span it can be divided into several stages known as states. Each state process has certain characteristics that describes the process. It means that process start executing, it goes through one state to another state.

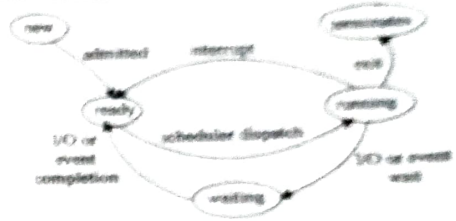


Fig. : State of process

Each process may be in one of following state :

1. **New :** The process is being created.
2. **Running :** Instructions are being executed. When a process gets a control from CPU & other resources, it starts executing.
3. **Waiting :** The process is waiting for some event to occur (such as an input I/O completion or reception of a signal). It is due to process lacks some resources other than the CPU.
4. **Ready :** The ready state requires. (a) The process is waiting to be assigned to a processor. (b) All ready queue process keeps waiting for CPU time to be allocated by operating system in order to run. (c) A program called scheduler which is part of operating system pick-up one ready process for execution by passing control to it.
5. **Terminated :** The process has finished execution.

These state names are vary across operating systems. They states, that they represent, are found on all systems. Certain operating systems more finely delineate process states. Only one process can be running on any processor at given moment of time, although many processes may be ready and waiting.

(ii) Thread : *Refer to Q.12.*

(iii) Kernel : The kernel is the central module of an operating system (OS). It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other of the operating system

and applications. The kernel code is usually loaded into a protected area of memory to prevent it from being overwritten by programs or other parts of the operating system

Typically, the kernel is *responsible* for memory management, process and task management, and disk management. *The kernel connects the system hardware to the application software.*

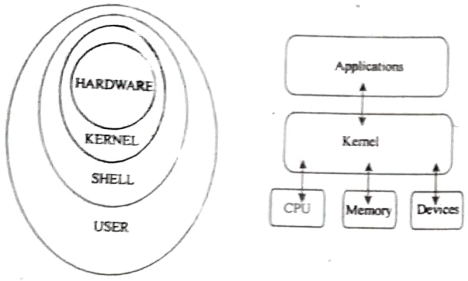


Fig.

Every operating system has a kernel. For example the Linux kernel is used numerous operating systems including Linux. Free BSD, Android and others.

Ans.(b) Operating System : *Refer to Q.18.*

Diffrence between multiprogramming and multi-processing

S.No.	Multiprogramming	Multitasking	Multiprocessing
1.	Single CPU is decides its time between more than one job.	Any system that run more than one application program onetime.	Multiple CPU perform more than one job at a time.
2.	Time sharing system application.	Resource management.	Main frame and super mini computers.

Q.28 ***What is critical section problem? How are semaphores used for solving critical section problem.*** ***[R.T.U. Dec. 2013]***

OR

Explain critical sections problem. How are semaphores used for solving critical section problem? ***[R.T.U. 2011]***

OR

How does a semaphore solve the critical section problem ? Discuss whether semaphores satisfy the three requirements for a solution to the critical section problem.

Ans. The Critical-Section Problem : *Refer to Q.25(a).*

A solution to critical section problem must satisfy following requirements :

1. **Mutual Exclusion :** Principle of mutual exclusion states that "If a process is executing in its critical section, then no other processes can be executing in their critical sections".
2. **Progress :** If there is no process in critical region and there are some processes that wants to execute in critical region, then the process that are not executing in the remainder section has a right to race for critical region.
3. **Bounded Waiting :** There is always a bound or a limit for a process to enter in the critical section.

We assume that each process is executing at a nonzero speed. However we can make no assumption covering the relative speed of the n processes.

At a given point in time, many kernels-made processes may be active in the operating system. As a result, the code implementing an operating system (Kernel code) is subject to several possible rare convolutions. Consider as an example a kernel data Structure that maintains a list of all open files in the system.

Solution of C.S.P. using Semaphores : We can use binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, meter, initialized to 1 each process Pi is organized as shown in figure.

```
do{
   Wait (mutex);
   // critical section
   Signal (mutex);
   // remainder section
   } While (True);
```

Fig. : Mutual exlusion implementation with Semaphores

The main disadvantage of the semaphore definition given here is that it require busy waiting while a process is in its critical section any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system. Where a single CPU is shared among may processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "Spins" while waiting for the lock.

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () Semaphore operations when a process execution the wait () operation and finds that the semaphore value is not positive, it must

wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphores, Should be restarted when some other process execute a signal () operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready state.

To implement semaphore under this section (definition) we define a semaphore as a "C" Struct:

```
typed of struct {
   int value;
struct Process * list;
} Semaphore;
```

Two general approaches are used to handle critical section in operating System:

(1) Preemptive kernels and

(2) Non preemptive kernels.

A preemptive kernel allows a process to be preempted while it is running in kernel mode.

A non preemptive kernel does not allow a process running in kernels made to be preempted; a kernel - mode process will run until it exists kernels made, blocks or voluntarily yield control of the CPU. Obviously, a non preemptive kernel is essentially free from rare conditions on kernel dates structure as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernels data are free from rare conditions. Preemptive kernel are especially difficult to design for SMP architecture, in these environments it is possible for two kernel mode processes to run simultaneously on different processors.

Q.29 (a) ***What is scheduling? Explain short term and long term scheduling. Describe the performance criteria of a scheduler.*** ***[R.T.U. Dec. 2013, 2012]***

(b) ***Consider the following set of process with the arrival time and CPU burst time given in milliseconds:***

Process	*Arrival time*	*CPU burst time*
P_1	*0*	*24*
P_2	*3*	*7*
P_3	*5*	*6*
P_4	*10*	*10*

Determine Average Waiting time and Average turn around time with the preemptive SJF scheduling. ***[R.T.U. Dec. 2013]***

Ans. (a) Process Scheduling: *Refer to Q.7.*

Levels of Scheduling : *Refer to Q.7.*

From a user's point of view, the performance criteria can be stated as follows :

- **Response Time :** The interval of time from the moment a service is requested until the response begins to be received. In time-shared, interactive systems this is a better measure of responsiveness from a user's point of view than turnaround time, since processes may begin to produce output early in their execution.

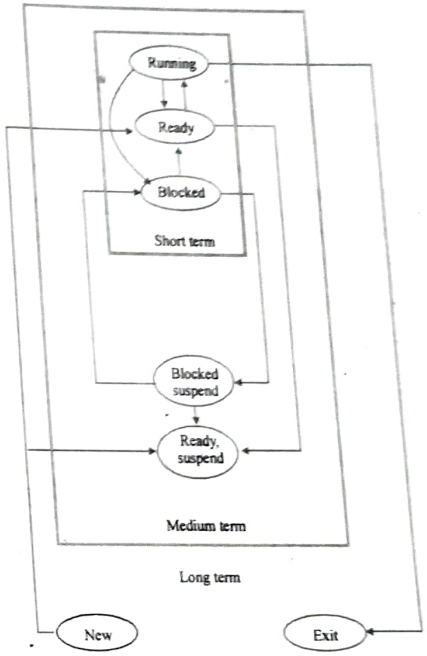


Fig. : Levels of scheduling

- **Turnaround Time :** The interval between the submission of a process and the completion of its execution, including the actual running time, plus the time spent sleeping before being dispatched or while waiting to access various resources. This is the appropriate responsiveness measure for batch production, as well as for time-shared systems that maintain multiple batch queues, sharing CPU time among them.

- **Meeting Deadlines :** The ability of the OS to meet predefined deadlines for job completion. It makes sense only when the minimal execution time of an application can be accurately predicated.
- **Predictability :** The ability of the system to ensure that a given task is executed within a certain time interval, and to ensure that a certain constant response time is granted within a strict tolerance, no matter what the machine load is.

The design of the short term scheduler is one of critical areas in the overall system design, because of the immediate effects on system performance from the user's point of view. It's usually one of the trickiest as well. Since most process switch mechanism is generally machine-dependent.

Short Term Scheduler : Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-strage device (typically a disk), where they are kept for later execution. The long term scheduler or job scheduler selects processes from this pool and loads them into memory for execution. The short term scheduler or CPU schedular selects from among the processes that are already to execute and allocates the CPU to one of them.

The primary distinction between these two scheduler lies in frequency of execution. The short terms-scheduler must select a new process for the CPU frequently. A Process may execute for only a few milliseconds before waiting for an I/O request often, the short term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short- terms scheduler must be fast. If it take 10 milliseconds to decide to execute a process for 100 milliseconds then 10/ (100+10) = 9 percent of the CPU is being used (Wasted) simply for scheduling the work.

Long Term Scheduler : The long-terms scheduler executes much less frequently minutes may separate the creation of one new process and next. The long - term scheduler controls the degree of multiprogramming. If the degree of multiprogramming is stable and then the average rate of processes leaving the system. Then, the long term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions the long - terms scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long- terms scheduler make a careful section. In general most processes can be described as either I/O bound or CPU bound. An I/O bound process is one that spends more of its time doing I/O than it spends doing computations A CPU bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-terms scheduler select a good process mix of I/O bound and CPU-bound processes. If all process are I/O bound the ready queue will almost always be empty, and the short term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU - bound are I/O - bound processes.

On some systems, the long-term scheduler may be absent or minimum for example, time - sharing systems such as UNIX and Micro-soft window systems often have no-long term scheduler but simply put every new process in memory for the short-term scheduler. The stability of these in memory depends either on a physical lamination or on the self - adjusting nature of human users. If performance decline to unsuitable levels on a multi-user system, some users will simply quit.

Ans. (b) As per given information the resulting preemptive SJF scheduling is depicted in the following Gantt chart :

P_1	P_2	P_3	P_4	P_1

0 3 10 16 26 47

Fig. : Gantt chart of given problem using preemptive SJF scheduling

As shown in above fig. process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 3. The remaining time for process P_1 (24.3 = 21 miliseconds) is larger than the time required by process P_2 (7 miliseconds), so process P_1 is preempted, and process P_2 is scheduled.

Similarly, process P_3 arrives at time 5. But the remaining time for process P_2 (7 – 2 = 5 miliseconds) is smaller than the time required by process P_3(6 miliseconds), so P_2 will continue its execution. After P_2, the process P_3 is scheduled and then P_4 and then P_1 will continue its execution.

Average waiting time

$$= \frac{(26-3)+(3-3)+(10-5)+(16-10)}{4}$$

$$= \frac{23+0+5+6}{4} = \frac{34}{4}$$

AWT = 8.5 miliseconds

Average turn around time = Average Waiting Time + Average Execution Time

= 8.5 + (47/4) = 20.25 miliseconds

□□□

MEMORY MANAGEMENT 2

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***Explain the difference between Paging and Segmentation.*** *[R.T.U. 2016]*

Ans. Difference between Segmentation and Paging

S. No.	Segmentation	Paging
1.	Programmer is aware of segmentation	Paging is hidden.
2.	Segmentation maintains multiple address spaces per process	Paging maintains one address space.

Q.2 ***Consider the following segment table.***

Segment	*Base*	*Length*
0	*219*	*600*
1	*2300*	*14*
2	*90*	*100*
3	*1327*	*580*
4	*1952*	*96*

Calculate the physical address for the following logical addresses? *[R.T.U. 2016]*

Ans.(i) 0,430 : 219 + 430 = 649
(ii) 1,10 : 2300 + 10 = 2310
(iii) 2,500 : Illegal Reference
(iv) 3,400 : 1327 + 400 = 1727
(v) 4,112 : Illegal Reference

Q.3 ***Compute page fault ratio. The pages referenced are 7, 5, 2, 1, 7, 5, 4, 5, 1, 2, 5 and 7 (12 pages). The job is allowed 3 blocks. Compare LRU and FIFO page replacement schemes.*** *[R.T.U. 2015]*

Ans. Let f denote fault and h denote hit.

FIFO Scheme

7-f
5-f
2-f
1-f
7-f
5-f
4-f
5-h
1-f
2-f
5-f
7-f

Page Fault Ratio = 11/12

LRU Scheme

7-f
5-f
2-f
1-f
7-f
5-f
4-f
5-h
1-f
2-f
5-h
7-f

Page Fault Ratio = 10/12

Q.4 ***Define Bankers's algorithm.***

Ans. Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

PART-B

Q.5 ***What is memory allocation schemes? Explain with example.*** ***[R.T.U. 2017]***

OR

Consider the following snapshot of system. The given jobs are of memory size 13 kB, 5 kB only.

Address	*Size of Free space*
005	*2*
070	*28*
105	*12*
279	*82*
395	*15*

Compare best fit, worst fit and first fit memory allocation schemes. Show the allocated addresses and free spaces after every job for all 3 schemes. ***[R.T.U. 2018]***

Compare best fit, worst fit and first fit memory allocation schemes. The given jobs are of memory size 13 KB, 5 KB only.

Address	*Size of free space*
005	*2*
070	*28*
105	*12*
279	*82*
395	*15*

Show the allocated addresses and free space table after every job for all 3 schemes. ***[R.T.U. 2015]***

Ans. Lets call the Job with size 13 as J13 and that with size 5 as J5.

Best Fit Scheme

J13 will be allocated 395

The free space table will then become:

Address	Size
005	02
070	28
105	12
279	82
408	02

J5 will be allocated 105.

The free space table will then become:

Address	Size
005	02
070	28
110	07
279	82
408	02

Worst Fit Scheme

J13 will be allocated 279.

The free space table will then become:

Address	Size
005	02
070	28
105	12
292	69
395	15

J5 will be allocated 292.

First Fit Scheme

The free space table will then become:

Address	Size
005	02
070	28
105	12
297	64
395	15

Q.6 ***Explain the difference between internal and external fragmentation.*** ***[R.T.U. 2018, Dec. 2013]***

OR

What is fragmentation? Differentiate between external and internal fragmentation. ***[R.T.U. 2017]***

Ans. Fragmentation : As processes is located and removed from memory, the free memory space is broken into little pieces. **External fragmentation** exists when there is enough total memory space to satisfy a request but the available space are not contiguous; storage is fragmented into a large number of small holds. This fragmentation problem can be severe. In the worst case, we could have a block of free memory between every two processes. It all these small pieces of memory were in one **big free** block instead, we might be able to run several **more process.**

Another factor in which end of a free block is allocated. Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another O.5N blocks will be lost to fragmentation.

Memory fragmentation : Memory fragmentation can be **internal** as well **as external.** Consider a multiple-partition allocation Scheme with a hole of 18, 464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed sized blocks and allocates memory in units based on block size with this approach the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation - unused memory that is internal to a partition.

One solution to be problem of external fragmentation is **compaction.** The goal is to shuffle the memory contents so as to place all free memory together in one large block compaction is not always possible, however. It relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible only if relocation is dynamic and is done at execution time. It addresses an relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to made all processes towards one end of memory: all holes move in the other direction, producing one large hole of available memory. This schemes can be expensive.

Q.7 ***Explain the FIFO, Optimal, LRU page replacement algorithm for the reference string.***

7 0 1 2 0 3 0 4 2 3 10 3. ***[R.T.U. 2017]***

Ans. (a) FIFO Page Replacement

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 10, 3

Considering no. of frames = 3 (A, B, C)

Time	0	1	2	3	4	5	6	7	8	9	10	11
A	(7)	7	7	(2)	2	2	2	(4)	4	4	(10)	10
B		(0)	0	0	0	(3)	3	3	(2)	2	2	2
C			(1)	1	1	1	(0)	0	0	(3)	3	3

At time = 3, the oldest string is replaced i.e. 7

At time = 4, 0 was already then, so no page fault.

At time = 5, 0 replaced by 3

At time = 6, 1 replace by 0

and so on.

Total page faults = 10

(b) Optimal Page Replacement

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 10, 3

Considering no. of frames = 3

Time	0	1	2	3	4	5	6	7	8	9	10	11
A	(7)	7	7	(2)	2	2	2	2	2	2	(10)	10
B		(0)	0	0	0	0	0	(4)	4	4	4	4
C			(1)	1	1	(3)	3	3	3	3	3	3

At time = 3, 2 is replaced by 7 as it is not seen anywhere in the remaining string.

At time = 5, 1 is replaced by 3 as 0 and 2 are seen earlier in the remaining string.

At time = 7, 0 is replaced by 4 as 2 and 3 are seen earlier in the remaining string.

No. of page faults = 7

(c) LRU Page Replacement

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 10, 3

Considering no. of frames = 3

Time	0	1	2	3	4	5	6	7	8	9	10	11
A	(7)	7	7	(2)	2	2	2	(4)	4	4	(10)	10
B		(0)	0	0	0	0	0	0	0	(3)	3	3
C			(1)	1	1	(3)	3	3	(2)	2	2	2

At time = 3, 7 is replaced by 2 as 0 and 1 were least recently used elements.

At time = 5, 1 is replaced by 3 as 0 and 2 were least recently used.

Total page faults = 9

Hence, for this reference string, page faults in optimal page replacement are found to be minimum:

Q.8 ***With the help of neat diagram Explain Memory hierarchy in detail.*** ***[R.T.U. 2016]***

Ans.

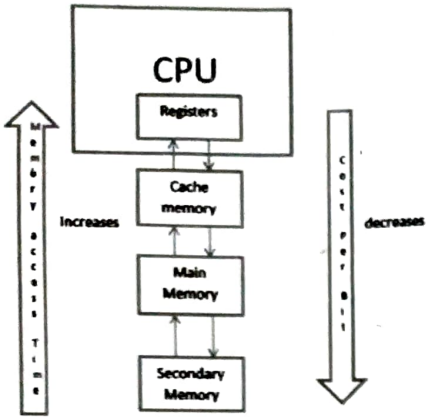


Fig.

1. Registers: CPU registers are at the top most level of this hierarchy, they hold the most frequently used data. They are very limited in number and are the fastest. They are often used by the CPU and the ALU for performing arithmetic and logical operations, for temporary storage of data.

2. Cache: The very next level consists of small, fast cache memories near the CPU. They act as staging areas for a subset of the data and instructions stored in the relatively slow main memory.

There are often two or more levels of cache as well. The cache at the top most level after the registers is the primary cache. Others are secondary caches. Many a times there is cache present on board with the CPU along with other levels that are outside the chip.

3. Main Memory: The next level is the main memory, it stages data stored on large, slow disks often called hard disks. These hard disks are also called secondary memory, which are the last level in the hierarchy. The main memory is also called primary memory.

The secondary memory often serves as staging areas for data stored on the disks or tapes of other machines connected by networks.

Q.9 ***There are 2 jobs of sizes 25 and 12 to be allocated memory. The free space table is :***

Address	*Size*
005	*02*
009	*17*
210	*89*
383	*13*
490	*11*

Apply best fit, first fit and worst fit schemes and show allocated addresses and free space table after allocation. *[R.T.U. 2015]*

Ans. Lets call the Job with size 25 as J25 and that with size 12 as J12.

Best Fit Scheme : J25 will be allocated 210 and J12 will be allocated 383. The free space table will then become:

Address	Size
005	02
009	17
235	64
395	02
490	11

First Fit Scheme : J25 will be allocated 210 and J12 will be allocated 009. The free space table will then become:

Address	Size
005	02
021	05
235	64
383	13
490	11

Worst Fit Scheme : J25 will be allocated 210 and J12 will be allocated 235. The free space table will then become:

Address	Size
005	02
009	17
247	52
383	13
490	11

Q.10 ***Explain free space management using bit map, linked list/free list.*** *[R.T.U. 2014, 2013]*

Ans. Memory Management with Bitmaps : With a bitmap, memory is divided into different allocation units of variable size. Corresponding to each allocation unit there is a bit in the bitmap, which is

- 0, if the unit is free
- 1, if it is occupied

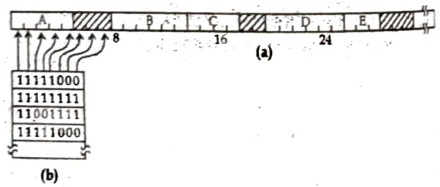


Fig. 1 : Memory management via bitmap

Figure Represents

(i) A part of memory with 5 processes and their holes.
(ii) The tick marks show the memory allocation units.
(iii) The corresponding bitmap.

The size of the allocation unit is an important design issue, smaller the allocation unit, larger the bitmap will be. If the allocation is chosen large, the bitmap will be smaller but appreciable memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.

The main problem with the bitmap is that when it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation.

Memory Management with Linked Lists

The way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.

Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

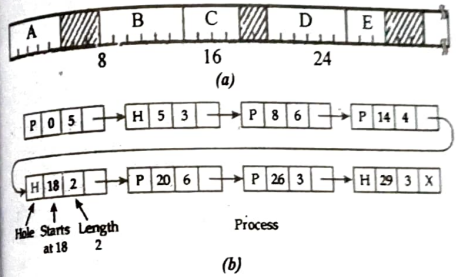


Fig. 2 : Memory management via linked list

The figure 2 shows :

(i) A part of memory
(ii) The corresponding Linked List.

Q.11 ***Compute number of page faults for LRU, FIFO and optimal page replacement algorithms. The given page trace is 7, 5, 1, 2, 7, 4, 5, 4, 5, 4, 5, 7 (12 pages). The job is allowed 3 blocks in primary memory.*** *[R.T.U. 2015]*

Ans. Let f denote fault and h denote hit.

FIFO Scheme

7-f
5-f
1-f
2-f
7-f
4-f
5-f
4-h
5-h
4-h
5-h
7-h

Page Fault Ratio = 7/12

LRU Scheme

7-f
5-f
1-f
2-f
7-f
4-f
5-f
4-h
5-h
4-h
5-h
7-h

Page Fault Ratio = 7/12

Optimal Scheme

7-f
5-f
1-f
2-f
7-h
4-f
5-h
4-h
5-h

Page Fault Ratio = 5/12.

Q.12 ***Explain the difference between logical and physical address space. Explain fragmentation. What are the various solutions for external fragmentation?*** *[R.T.U. 2014]*

Ans. Logical and 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 resistor 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

result in differing logical and physical address. In this case, we usually refer to the logical address as a **virtual address.** We use logical address and virtual address interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space;** the set of all physical address 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.

Fragmentation : Fragmentation, in the context of a hard disk, is a condition in which the contents of a single file are stored in different locations on the disk rather than in a contiguous space. This results in inefficient use of storage space as well as occasional performance degradation. Users frequently create, modify, delete and save files. Back-end operating systems (OS) continuously store these files on hard drives, which inevitably creates scattered files. When fragmentation occurs, the OS needs to consolidate stored files to enhance processing efficiency.

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 can not be used.

Internal Fragmentation

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

External Fragmentation is Solved by Any these methods :

1. **Compaction :** The solution to this kind of external fragmentation is compaction. Compaction is a method by which all free memory that are scattered are placed together in one large memory block. It is to be noted that compaction cannot be done if relocation is done at compile time or assembly time. It is possible only if dynamic relocation is done, that is relocation at execution time.
2. **Garbage Collection :** It collects all the memory which is inaccessible and return them as a free memory.
3. Another possible solution to the external fragmentation is to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available .Two complementary techniques to achieve this solution are paging and segmentation. These techniques can be combined also.

(i) **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.

(ii) **Segmentation :** Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information. For example ,data segments or code segment for each process, data segment for operating system and so on. Segmentation can be implemented using or without using paging. Unlike paging, segment are having varying sizes and thus eliminates internal fragmentation. External fragmentation still exists but to lesser extent.

Q.13 *Consider the following snapshot of the system:*

	Allocation	*Max*	*Available*
Process	*A B C*	*A B C*	*A B C*
P_0	*0 1 0*	*7 5 3*	*3 3 2*
P_1	*2 0 0*	*3 2 2*	
P_2	*3 0 2*	*9 0 2*	
P_3	*2 1 1*	*2 2 2*	
P_4	*0 0 2*	*4 3 3*	

If a request from process P_1 arrives for (0, 1, 2) can the request be granted immediately? What is the content of need matrix? *[R.T.U. 2014, 2013]*

Ans. The content of the matrix need is defined to be Max-Allocation and is

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

We claim that the system is currently in a safe state. Indeed, the sequence $< P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria. Now, according to problem (given), process P_1 requests one additional instance of resource type B and two instances of resource type C, so request, = (0, 1, 2).

To decide whether this request can be immediately granted, we first check that request ≤ Available (*i.e.*, (0, 1, 2)≤(3,3,2)), which is true. We then pretend that this request has been fulfilled and we arrive at the following new state.

Allocation			Need			Available		
A	B	C	A	B	C	A	B	C
0	1	0	7	4	3	3	2	0
2	1	2	1	1	0			
3	0	2	6	0	0			
2	1	1	0	1	1			
0	0	2	4	3	1			

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $< P_1, P_3, P_4, P_0, P_2>$ satisfies our safety requirement, Hence, we can immediately grant the request of process P_1.

PART-C

Q.14 (a) ***What do you understand by Belady's Anomaly? Explain.*** *[R.T.U. 2018, 2015]*

OR

What is Belady's Anomaly? In which algorithm does it occur? *[R.T.U. 2016]*

OR

What is Belady's Anomaly? *[R.T.U. 2011]*

OR

Describe Belady's Anomaly with the help of suitable example? *[R.T.U. 2009]*

(b) ***Consider 3 page frames and the following reference string using FIFO page replacement algorithm to calculate the number of page faults in each reference string :***

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

[R.T.U. 2018, 2014, 2013]

OR

Consider 3 page frames and the following references string. Use LRU page replacement algorithm to calculate the number of page faults in each. Preference string is :

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

[R.T.U. 2011]

OR

Consider the following page reference string.

(7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1)

How many page faults would occur for the replacement algorithms.

(i) FIFO (ii) LRU (iii) Optimal? Assume four frames available. *[R.T.U. 2009]*

Ans.(a) Belady's Anomaly : As the name suggests, in this scheme, the page that is removed from the memory is the one that entered first. Assuming the some page reference string as shown in fig.1, the states of various page frames and page faults after each page reference. As it is clear from the figure, 15 page faults result.

This algorithm is easy to understand and program. The first three columns are self-explanatory. In the fourth reference of page 3, a page fault results and the FIFO algorithm throws out page 8 because it was the first one to be brought in. (It is a coincidence that the OPT also would decide on the same.) The fifth page reference does not cause a page fault in both OPT and FIFO algorithms as page 1 is already in the memory. The sixth page reference is for page 4. Here, FIFO will throw out page 1, because it came in earlier than the remaining two pages at that time,viz. pages 3 and 2. Page 1 has been there the longest. Notice that OPT had chosen page 2 for throwing out.

Page References	8	1	2	3	1	4	1	5	3	4	1	4	3	2	3	1	2	8	1	2
Page Frame 0	8	8	8	3	3	3	3	5	5	5	1	1	1	1	1	1	1	8	8	8
Page Frame 1		1	1	1	1	4	4	4	3	3	3	3	3	2	2	2	2	2	1	1
Page Frame 2			2	2	2	2	1	1	1	4	4	4	4	4	3	3	3	3	3	2
Page Fault (* = yes)	*	*	*	*		*	*	*	*	*			*	*			*	*	*	

Fig.1 : FIFO algorithm

The reason is that it "knew" in advance that page 1 is going to be required sooner. The FIFO policy does not "know" this. Hence, in just the next, i.e. the seventh page reference, page 1 is required causing yet another page fault. Following this logic,the table can be completed.

The FIFO algorithm has one anomaly known as **Belady's anomaly**, named after the one who discovered it first.

Page reference	2	3	4	5	2	3	6	2	3	4	5	6
Page frame 0	2	2	2	5	5	5	6	6	6	6	6	6
Page frame 1		3	3	3	2	2	2	2	2	4	4	4
Page frame 2			4	4	4	3	3	3	3	3	5	5
Page fault (* = Yes)	*	*	*	*	*	*	*			*	*	

Fig.2 : 9 Page faults with 3 page frames

Normally, if the physical memory size and therefore, the number of page frames available increases, the number of page faults should decrease. This should enhance the performance. But this is not necessarily the case if we use FIFO as the page replacement policy. This, in essence, is Belady's anomaly. Let us, for example, take a reference string 2, 3, 4, 5, 2, 3, 6, 2, 3, 4, 5, 6.

Figure 2 shows that with 3 page frames, FIFO gives 9 page faults. However, fig.3 shows us that if we increase the number of page frames from 3 to 4, the number of page faults increases and comes 10.

Page reference	2	3	4	5	2	3	6	2	3	4	5	6
Page frame 0	2	2	2	2	2	2	6	6	6	6	5	5
Page frame 1		3	3	3	3	3	3	2	2	2	2	6
Page frame 2			4	4	4	4	4	4	3	3	3	3
Page frame 3				5	5	5	5	5	5	4	4	4
Page fault (* = Yes)	*	*	*	*			*	*	*	*	*	*

Fig. 3: 10 Page faults with 4 page frames

This clearly is anomalous. From these figures, we notice that the page faults increase because in the case represented by fig.3, the page is referenced as soon as it is evicted in this specific reference string. Fortunately, this anomalous behavior is rare and can be found only for specific types of page reference strings. It does not always happen for all reference strings and hence, FIFO is still a fairly good algorithm.

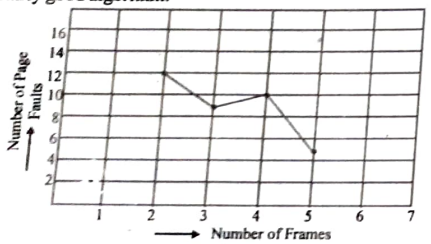
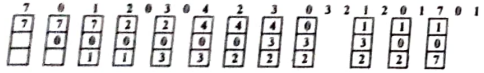


Fig.4

Ans.(b) (i) First In first Out (FIFO)

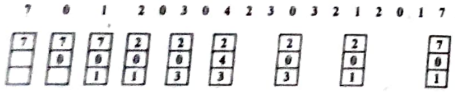
Total Page Fault = 15

(ii) Least Recently Used (LRU)



Total Page Fault = 12

(iii) Optimal Page Replacement :



Total Page Replacement = 9

For example, on our sample reference string, the optimal page - replacement algorithm would yield page faults as shown above. The First three reference cause faults that fill the three empty frames. The reference to page 4 replace page 3, because page 3 will not be used for the longest period of time.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

Q.15 ***Explain the following :***

(i) Demand Paging *[R.T.U. 2018]*

(ii) Segmentation with Paging scheme

[R.T.U. 2018, Dec. 2013, R.T.U. 2009]

Ans.(i) Demand Paging: Consider a program that starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether as option is ultimately selected by the user or not.

An alternative strategy is to initially load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand- paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

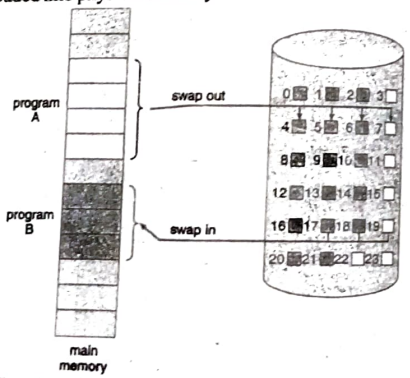


Fig. : Transfer of a paged memory to contiguous disk space

A demand-paging system is similar to a paging system with swapping (Figure) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we

use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term *swapper* is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

Ans. (ii) Segmentation with Paging Scheme : Users prefer to view memory as a collection of variable-sized segments with no necessary ordering among segments. Consider how we think of a program when a set of methods, procedures functions. It may also include various data structures: objects, arrays, stack, variables and so on. Each of these modules or data elements is referred to by name. You talk about "the stack", "the math library", "the main program", without caring what addresses in memory these elements occupy. We are not concerned with whether the stack is stored before or after the Sqrt( ) function. Each of these segments is of variable length, the length is intrinsically defined by the purpose of the segment 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.

Segmentation is a memory - management scheme that supports this user view of memory. A logical address is a collection of segment. Each segment has a segment name and length. The addresses specifies both the segment name and the offset within the segment. The user therefore specifies each addresses by two quantities : a segment name and an offset.

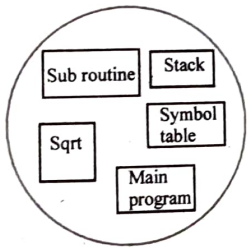


Fig. 1 : Logical address user's view of a program

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 two tuples:-

<Segment_number, offset >

Normally, the user program is compiled and the compiler automatically constructs segments reflecting the input program.

Example of segmentation : Consider the situation shown in fig.

We have fine segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segments table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of the segments. For example segment 2 is 400 bytes long and begins at location 4300. Thus, a referer byte 53 of segment 2 is mapped on location 4300 + 53 = 4553. A references to segment 3, byte, is mapped physical memory to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

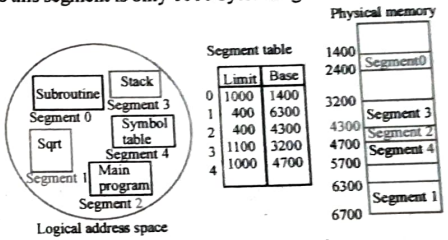


Fig. 2 : Example of segmentation

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

As jobs enter the system, they are put into a job queue. The job scheduler takes into account the memory requirements of each job and available regions in determining which jobs are allocated memory. When a job is allocated space, it is loaded into a region (relocating it if necessary). It can then compete for the CPU. When a job terminates, it releases its memory region, which the job scheduler may then fill with another job from the job queue.

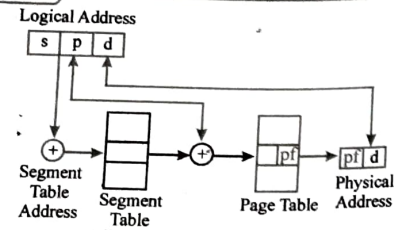


Fig. 3 : Segmentation with paging

Variable Partitions (MVT) : The main problem with MFT is determining the best region sizes to minimize internal and external fragmentation. Unfortunately, with a dynamic set of jobs to run, there is probably no one right partition of memory. For example, suppose that 120K of memory is available for user programs and that all user jobs are 20K, except one big job of 80K which runs once a day. We must allocate an 80K region to allow this program to run, but since all other jobs are 20K we are faced with 60K (half of the user memory space) of internal fragmentation, except when that one big job is run once a day.

The solution to this problem is to allow the region sizes to vary dynamically. This approach is called multiple contiguous variable partition allocation.

MVT Memory Management is Fairly Simple : The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user programs and is considered as one large block of available memory, a hole. When a job arrives and needs memory, we search for a hole large enough for this job. If we find one, we allocate only as much as is needed, keeping the rest available to satisfy future requests.

Difference between Segmentation and Paging

S. No.	Segmentation	Paging
1.	Programmer is aware of segmentation	Paging is hidden.
2.	Segmentation maintains multiple address spaces per process	Paging maintains one address space.
3.	Segmentation allows procedures and data to be separately protected.	This is hard with paging.
4.	Segmentation easily permits tables whose size varies.	Pages are of fixed size.
5.	Segmentation facilities sharing of procedures between processes.	This is hard with paging.

Paging :

Like in paging systems, the address in this case consists of a segment number (S) and an offset (D) within that segment. Hence, this is also a two-dimensional address. Within one segment, the address of any location is computed with respect to the beginning of the segment and hence, is a virtual address. When we put all these segments conceptually (or virtually) one after the other, the following picture emerges (Fig.1).

Virtual Address Limits	0 – 999	1000 – 1699	1700 – 2499	2500 – 3399	3400 – 3999
Virtual Segment Number	0	1	2	3	4

Fig.1: Virtual addresses in segmentation

Now, a virtual address 1100 in this program will mean segment 1, displacement = 100, i.e. S = 1, D = 100. (Address 1000 means S = 1, D = 0, address 1001 means S = 1, D = 1 and so on.) Similarly, virtual address 3002 will means S = 3, D = 502. These are the two dimensions or components of the address. If we specify segment number (S) and displacement (D) within it, we have specified a complete address. In segmentation, the compiler itself has to generate a two-dimensional address. This is different from the address on paging. In paging, a single-dimensional virtual address and a two-dimensional address would be exactly the same in the binary form as the page size is an exact power of 2. In segmentation, this is not so. The segment size is unpredictable. That is why we need to express an address explicitly in a two-dimensional form. Hence, if the operationg system uses segmentation as a philosophy for memory management, an executable image of a program has to have a two-dimensional address (S, D) in any program instruction. This means that in segmentation, we need a different address format and a different architecture to decode that address. For instance, in PDP-11/45, an address consists of 16 bits, of which 13 (bits 0 - 12) are reserved for displacement (D) and 3 bits (bits 13 - 15) are reserved for segment number (S). This is shown in Fig.2.

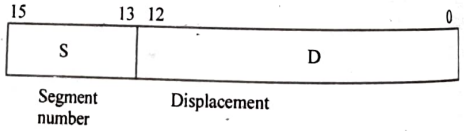


Fig.2: Address in PDP-11/45

Therefore, a PDP-11/45 can have 8 segments corresponding to 3 bits for S, where each segment can be of maximum 8 Kb, corresponding to 13 bits for D.

Address Translation and Relocation

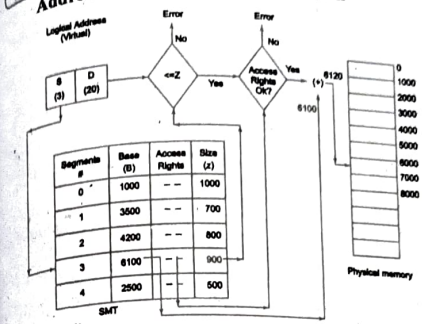


Fig.3: Address translation in segmentation

The SMT (Segment Map Table) is used for address translation as shown in Fig.3. The fields in the SMT are shown differently sequenced in the figure, to avoid cluttering. We will assume that the virtual address to be translated is 2520 consisting of two components: S and D. We assume the egments through we can easily find out that S = 3 and D = 20 for this virtual address 2520. The address as it exists in the instruction register (IR) itself consists of these two parts.

The steps followed for address translation are as given below :

(a) The high order bits representing S are taken out from the IR to form the input to the SMT as an index. For instance, the entry for S = 3 in the SMT can be directly accessed in Fig.3.

(b) The operating system now stores the data from that entry (in this case the one with S = 3). Hence, it will store the segment size Z = 900 and the starting address B = 6100 for future use.

(c) We know that displacment (D) is a virtual address within that segment. Hence, it has to be less than the segment size. Therefore, the displacement (D) is compared with Z to ensure that D <= Z. If not, the hardware itself generates an error for illegal address. This is evidently a protection requirement. In our example, D which is 20, is less than Z which is 900, which is acceptable.

(d) If the displacement (i.e. D) is legal, then the operating system checks the access rights and ensures that these are not violated.

(e) Now, the effective address is calculated as (B + D), as shown in the figure. This will be computed as 6100 + 20 = 6120 in our example.

(f) This address is used as the actual address and is pushed into the address bus to access the physical memory.

Thus we can calculate the physical address of segment by translating virtual address.

Virtual Paging : There are several major problems with paging. The first is the overhead associated with each address transformation. Another problem is the extra core or extra registers needed for the page tables. Since a job may not be a multiple of 1000 bytes long, a portion of the last page will be wasted. This is called page breakage. On the average, there is half a page wasted for each job. This can be serious if there are a large number of small jobs. There are three further disadvantages. First, there will still be wasted core if there is sufficient space for some, but not all of an additional job's pages. Second, in this type of paging scheme, all the pages of a particular job must be in core before that program can be executed. And third, a considerable amount of hardware support is needed.

Demand Paging : All these disadvantages are overcomed by demand paging. In demand paging a program can be executed without all pages being in core, *i.e.* pages are fetched into core as they are needed. By choosing sufficiently small intervals of time, demand paging can be very valuable in "time sharing" systems which attempt to run dozens of programs simultaneously. However, the problem associated with demand paging is thrashing.

Paged Segmentation : An appealing idea would be to combine the best features of the previously presented techniques. Segmentation can be used to facilitate sharing and protection paging can be used to resolve the fragmentation and recompacting problems and demand paging eliminates the restriction on address space size. Some computer systems combine the two approaches in order to enjoy the benefits of both. One popular approach is to use segmentation from the users point of view but to divide each segment into pages of fixed size for purposes of allocation. In this way, the combined system retains most of the advantages of segmentation. At the same time, the problems of complex segment placement and management of secondary memory are eliminated by using paging.

Q.16 ***What is thrashing? What do you understand by degree of multiprogramming.*** ***[R.T.U. 2017]***

OR

Explain Thrashing. ***[R.T.U. 2018, Dec. 2013]***

Ans. Thrashing : If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of

intermediate CPU scheduling.

In fact, look at any process that does not have "enough" frames. Although it is technically possible to reduce the number of allocated frames to the minimum, there is some (larger) number of pages in active use. If the process does not have this number of frames, it will quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again and again. The process continues to fault, replacing pages for which it then faults and brings back in right away.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing : Thrashing results in server performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.

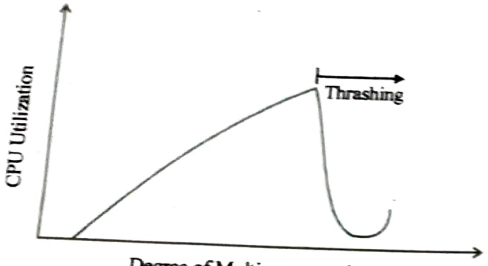


Fig.

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

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

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

Thrashing Avoidance : We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm**). With local replacement, if one process starts thrashing, it cannot steal frames from another process and it the latter to thrash also. Pages are replaced with regard to the process of which they are a part. However, if processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase, due to the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

In a multiprogramming-capable system, jobs to be executed are loaded into a pool. Some of these jobs are loaded into main memory, and one is selected from the pool for execution by the CPU. If at some point, the program in progress terminates or requires the services of a peripheral device, the control of the CPU is given to the next job in the pool. As programs terminate, more jobs are loaded into memory for execution, and CPU control is switched to another job in memory. In this way the CPU is always executing some program or some portion thereof, instead of waiting for a printer, tape drive, or console input.

An important concept in multiprogramming is the degree of multiprogramming. The degree of multiprogramming describes the maximum number of processes that a single-processor system can accommodate efficiently. The primary factor affecting the degree of multiprogramming is the amount of memory available to be allocated to executing processes. If the amount of memory is too limited, the degree of multiprogramming will be limited because fewer processes will fit in memory. A factor inherent in the operating system itself is the means by which resources are allocated to processes. If the operating system cannot allocate resources to executing processes in a fair and orderly

fashion, the system will waste time in reallocation, or process execution could enter into a deadlock state as programs wait for allocated resources to be freed by other blocked processes. Other factors affecting the degree of multiprogramming are program I/O needs, program CPU needs, and memory and disk access speed.

Q.17 *What you mean by paging? Explain the concept of demand paging with proper diagram.*

[R.T.U. 2017]

OR

What is demand paging?

[R.T.U. 2014]

Ans. Paging : Paging is a memory management technique in which the memory is divided into fixed size pages. Paging is used for faster access to data. When a program needs a page, it is available in the main memory as the OS copies a certain number of pages from your storage device to main memory. Paging allows the physical address space of a process to be noncontiguous.

Demand Paging : *Refer to Q.15(i).*

Basic Concepts : When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

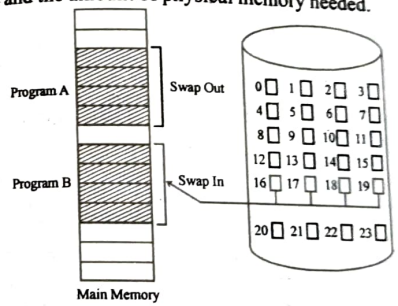


Fig. 1 : Transfer of a paged memory to contiguous disk space

With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid-invalid bit scheme can be used for this purpose. This time, however, when this bit is set to "valid", this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid" this value indicates that the page either is not valid (that is, not the logical address space of the process) or is valid but is currently on the disk. The page table entry for a page that is brought into memory is

set as usual, but the page table entry for a page that is not currently in memory is simply marked invalid or contains the address of the page on disk.

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

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page-fault trap. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk-transfer overhead and memory requirements), rather than an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript). We must therefore correct this oversight. The procedure for handling this page fault is straight forward.

1. We check an internal table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.

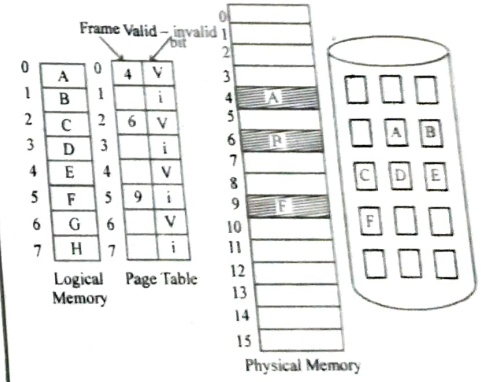


Fig. 2 : Page table when some pages are not in main memor

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

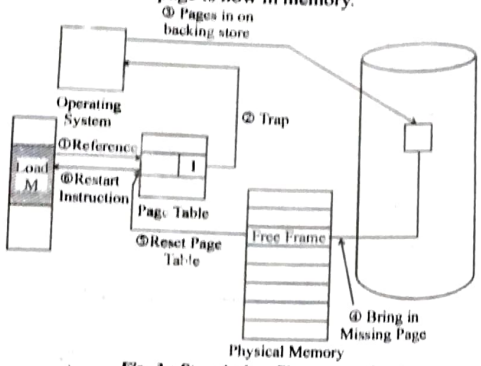


Fig. 3 : Steps in handling a page fault

6. We restart the instruction that was interrupted by the illegal address trap. This process can now access the page as though it had always been in memory.

It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process. When the page fault occurs, we can restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps to the operating system (page fault). The operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

In the extreme case, we could start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging.** Never bring a page into memory until it is required.

Theoretically, some programs may access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behaviour is exceedingly unlikely. Programs tend to have locality of reference, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping :

Page Table : This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.

Secondary Memory : This memory holds those pages that are not present in main memory. The secondary memory is usually a high speed disk. It is known as the swap device and the section of disk used for this purpose is known as **Swap Space.**

Performance of Demand Paging : Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the **effective access time** for a demand-paged memory. For most computer systems, the memory access time, denoted on a, now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero; that is, there will be only a few page faults. The effective access time is then

Effective Access Time = $(1 - p) \times ma + p \times$ page fault time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequences to occur :

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame :

(a) Wait in a queue for this device until the read request is serviced.

(b) Wait for the device seek and/or latency time.

(c) Begin the transfer of a page to a free frame.

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Interrupt from the disk (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.

12. Restore the user registers, process state and new page table, resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization, but requires additional time to resume that page-fault service routine when the I/O transfer is complete.

In any case, we are faced with three major components of the page-fault service-time :

(1) Service the page-fault interrupt.
(2) Read in the page.
(3) Restart the process.

The first and third tasks may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 millisecond, a seek of 15 milliseconds and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember also that we are looking at only the device service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device-queueing time as we wait for the paging device to be free to service our request, increasing even more time to swap.

If we take an average page-fault service time of 25 milliseconds and a memory-access time of 100 nanoseconds, then the effective access time in nanoseconds is

effective access time = $(1 - p) \times 100 + p$ (25 milliseconds)

$= (1 - p) \times 100 + p \times 25{,}000{,}000$

$= 100 + 24{,}999{,}900 \times p$

We see then that the effective access time is directly proportional to the **page fault-rate.** If one access out of 1,000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging. If we want less than 10 percent degradation, we need

$110 > 100 + 25{,}000{,}000 \times p$

$10 > 25{,}000{,}000 \times p$

$p < 0.0000004$

That is, to keep the slow down due to paging to a reasonable level. We can allow only less than one memory access out of 2,500,000 to page fault.

It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

Q.18 ***Explain the various page replacement policies using a suitable example.***

[R.T.U. Dec. 2013, 2012]

OR

Write short note on Page Replacement Algorithms in Detail. ***[R.T.U. 2016]***

Ans. (1) FIFO Page Replacement : The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our examples reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Fig. 1. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults alltogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows program execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

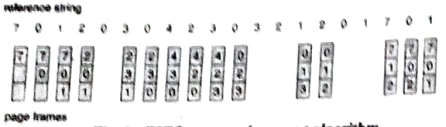


Fig. 1 : FIFO page-replacement algorithm

(2) Optimal Page Replacement : One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply "Replace the page that will not be used for the longest period of time."

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in fig. 2 The first three references causes faults that fill the three empty frames. The reference to page

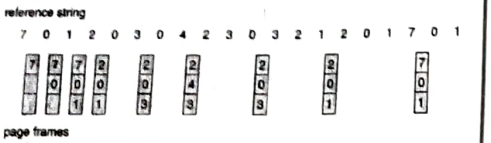


Fig. 2 : Optimal page-replacement algorithm

2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

(3) LRU Page Replacement : If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; 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 will replace the page that has not been used for the longest period of time (fig. 3). This approach is the *least-recently-used (LRU)* algorithm.

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

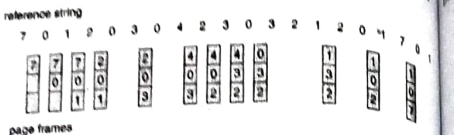


Fig. 3 : LRU page-replacement algorithm

The result of applying LRU replacement to our example reference string shown in fig. 3. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it them faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory P{0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm is considered to be good.

(4) Far Page Replacement : When programs execute, they tend to reference functions and data in predictable patterns. The *far page-replacement* strategy uses graphs to make replacement decisions based on these predictable patterns. The far strategy has been shown mathematically to perform at near-optimal levels, but it is complex to implement and incurs significant execution-time overhead.

The far strategy creates an access graph (fig. 4) that characterizes a process's reference patterns. Each vertex in the access graph represents one of the process's pages. An edge from vertex v to vertex w means that the process can reference page w after it has reference page v. For example, if an instruction on page v references data on page w, there will be a directed edge from vertex v to vertex w. Similarly, if a function call to page x returns to page y, there will be an edge from vertex x to vertex y. The graph, which can become quite complex, describes how a process can references pages as it executes. Access graphs can be created by analyzing a compiled program to determine which pages can be accessed by

each instruction on each page, which can require significant execution time. The access graph in fig. 4 indicates that, after the process references page B, it will next reference either page A, C, D or E, but it will not reference page G before it has referenced page E.

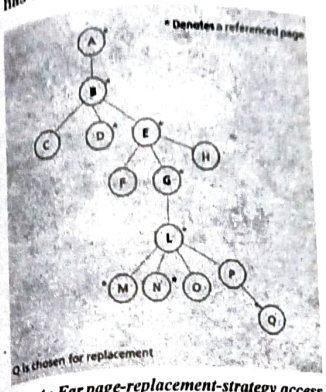


Fig. 4 : Far page-replacement-strategy access graph

The replacement algorithm operates in phases much like the clock algorithm. For initially marks all vertices in the access graph as unreferenced. When the process accesses a page, the algorithm marks as referenced the vertex that corresponds to the page. When the algorithm must select a page for replacement, it chooses the unreferenced page that is farthest away (hence the name "far") from any referenced page in the access graph (in fig 4, this corresponds to page Q). The intuitive appeal of this strategy is that the unreferenced page that is farthest from any referenced page is likely to be referenced farthest in the future. If the graph does not contain an unreferenced vertex, the current phase is complete, and the strategy marks all the vertices as unreferenced to begin a new phase. At this point, the algorithm replaces the page farthest in the graph from the most-recently referenced page.

Q.19 *(a) Is there any difference between pure paging and demand paging? Explain.*

(b) Compute page fault ratio. The pages referenced are 7, 5, 2, 1, 7, 5, 4, 5, 1, 2, 5 and 7 (12 pages). The job is allowed 3 blocks. Compare LRU and FIFO page replacement schemes. *[R.T.U. 2015]*

Ans.(a) In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (i.e., if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of process's working set of pages is located in physical memory. This is an example of a lazy loading technique.

Contrast this to pure paging, all memory for a process is swapped from secondary storage to main memory during the process startup.

Demand paging:

1. Only loads pages are demanded by the executing process.
2. As there is more space in main memory, more processes can be loaded reducing context switching time which utilizes large amounts of resources.
3. Less loading latency occurs at program startup, as less information is accessed from secondary storage and less information is brought into main memory.
4. As main memory is expensive compared to secondary memory, this technique helps significantly to reduce the bill of material (BOM) cost in smart phones for example.
5. Individual programs face extra latency when they access a page for the first time.
6. Memory management with page replacement algorithms becomes slightly more complex.

Pure paging:

1. All pages are pre-loaded.
2. Less processes can be loaded increasing context switching time.
3. More loading latency at program startup.
4. Bill of Material cost is increased because higher main memory is needed.
5. Individual programs face no latency when they access page at any time.
6. No need to memory management with page replacement algorithms.

Ans.(b) Let f denote fault and h denote hit.

FIFO Scheme

7-f
5-f
2-f
1-f
7-f
5-f
4-f
5-h
1-f
2-f
5-f
7-f

Page Fault Ratio = 11/12