

- 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  $P_i$  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)



- (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.13** What you mean by thread? Explain kernel and user level thread. [R.F.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.

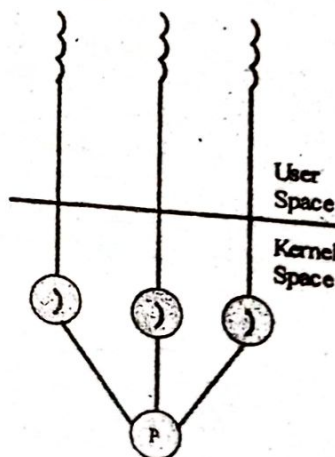
A thread is similar to a sequential program a single thread also has a beginning, an end, a sequence, and at any given time during the runtime of the program, there is a single point of execution. However, a thread itself is not a program, it cannot run on its own, but runs within a program.

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 a



process having large number of threads than process having small number of threads.

- Kernel-level threads are especially good for applications that frequently block.

#### Disadvantages

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes. It requires 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.

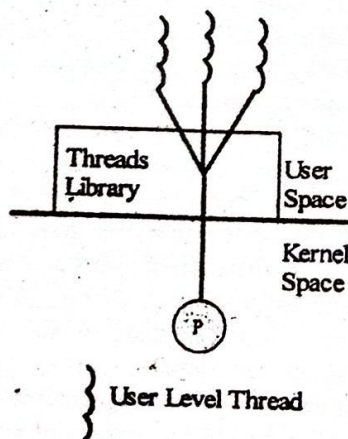


Fig. 2 : User level threads

#### 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 be 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.



**Operating System**

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-level tasks, whereas a kernel thread is better for high-priority tasks that should get high priority to system resources.



Robin  
Page  
CPU  
times  
n for  
2015/

wait

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 :

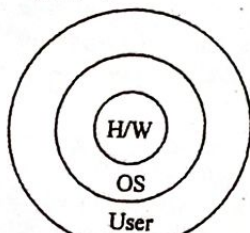


Fig. 1 : OS as an Intermediary

### 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 task.

### 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 underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (fig. 2).

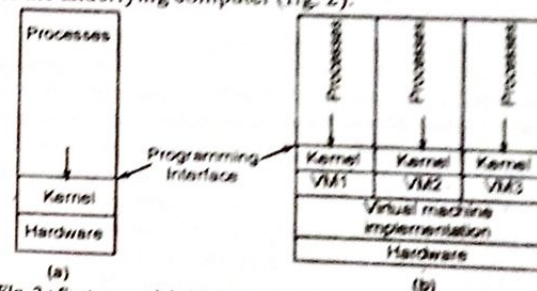


Fig. 2 : System models (a) Non-virtual machine (b) Virtual machine

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

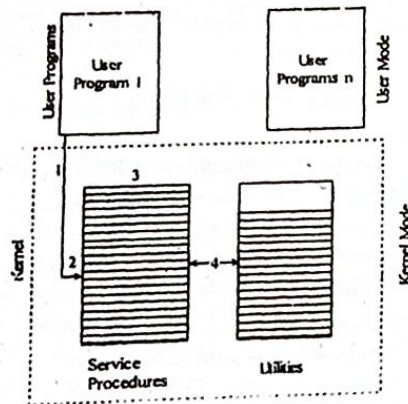
**Explain the architecture of operating system with neat and clean diagram.** [R.T.U. 2017]

**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.



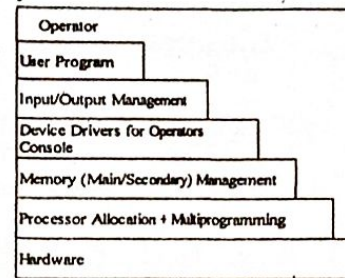
**Fig. 1 : Monolithic architecture**

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.

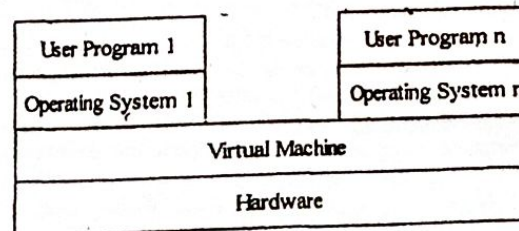


**Fig. 2 : Layered architecture**

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 architecture**

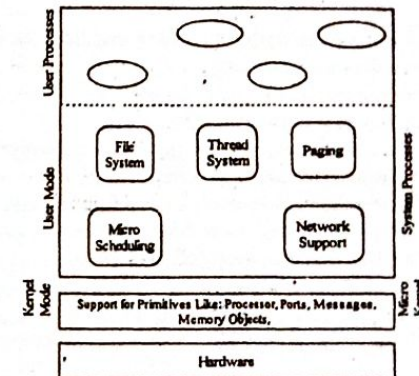
Virtual machine has two main components :

- The control program (CP) controls the real machine.
- 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 advantage as it always stays in the main memory and thus consumes less memory of the system.



**Fig. 4 : Micro kernel 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 compares 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.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:

$$100\% / 3 \text{ groups} = 33.3\% \text{ per group}$$

$$\text{Group 1: } (33.3\% / 3 \text{ users}) = 11.1\% \text{ per user}$$

$$\text{Group 2: } (33.3\% / 2 \text{ users}) = 16.7\% \text{ per user}$$

$$\text{Group 3: } (33.3\% / 4 \text{ users}) = 8.3\% \text{ per user}$$

**(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.

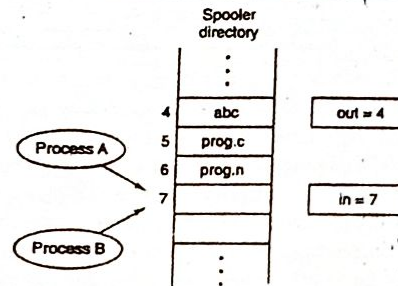


Fig. : Two processes want to access shared memory at the same time

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 unexplained happens.

(iii) Critical section :

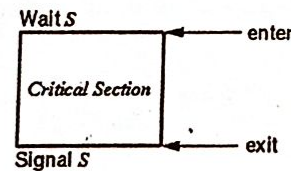


Fig.

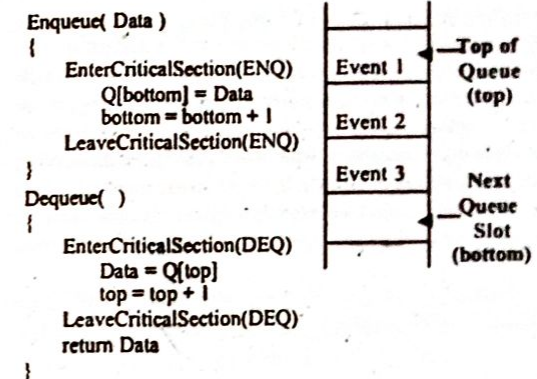
The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the *Critical Section*. To avoid race conditions and flawed results, 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**



Does the above code avoid race conditions?

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.

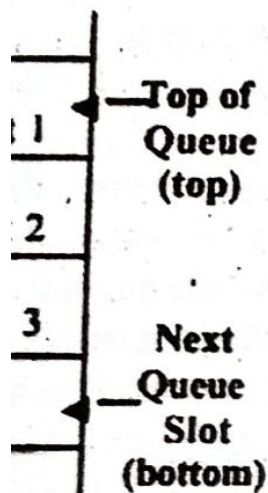
**Mutex :** Mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads 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



ts critical section,  
ecute in its critical  
l sections by the



ditions?  
critical section,

allows multiple  
ce, such as file  
rogram is started  
after this stage,  
lock the mutex  
rce. The mutex  
needed or the

semaphore and  
emaphore are

echanism used  
one task (can  
straction) can  
hip associated

**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 single-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 in 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 performed 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.

ARTOS 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 system. Some examples are Windows for Workgroups, Windows NT, AppleShare, DEDnet, LAN tastic, 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.



They are separate loadable modules.

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 :** They 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.

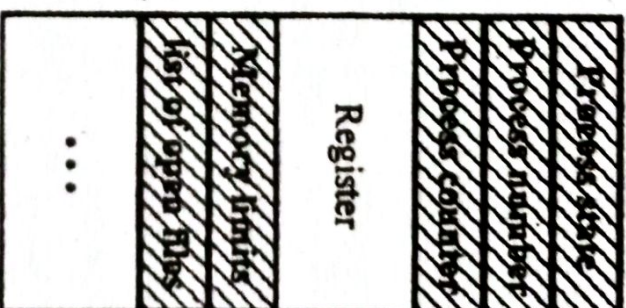


Fig. : Process control block (PCB)



- **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 :

- Process
- Thread
- 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.

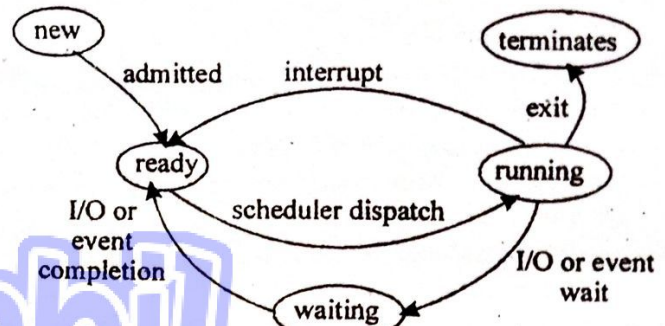


Fig. : State of process

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.

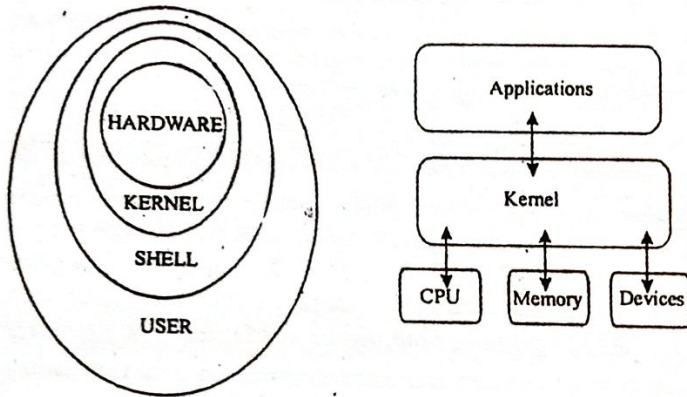


Fig.

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.

**Difference between multiprogramming and multiprocessing**

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 executed 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  $P_i$  is organized as shown in figure.

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

Fig. : Mutual exclusion 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.



## 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 Banker'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. If 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.S.N 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		①	0	0	0	③	3	3	②	2	2	2
C			①	1	1	1	①	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		①	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		①	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:



**Operating System**

To decide whether this request can be immediately granted, we first check that request  $\leq$  Available (i.e.,  $(0, 1, 2) \leq (3, 3, 2)$ ), 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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  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	4	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.



**OS.40**

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	2	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.

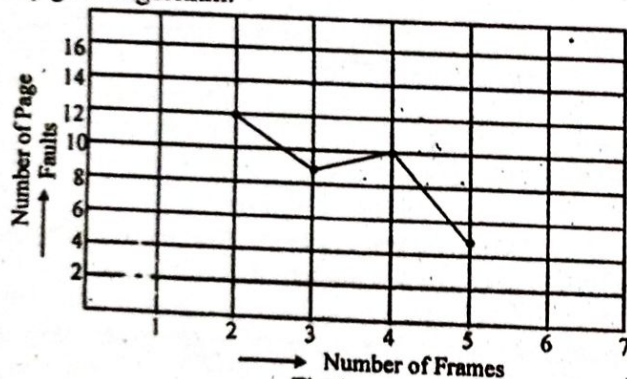


Fig.4

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

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

Total Page Fault = 15

(ii) Least Recently Used (LRU)

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

Total Page Fault = 12

(iii) Optimal Page Replacement :

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

Total Page Replacement = 9

**B.Tech. (V Sem.) C.S. Solved Papers**

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 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.

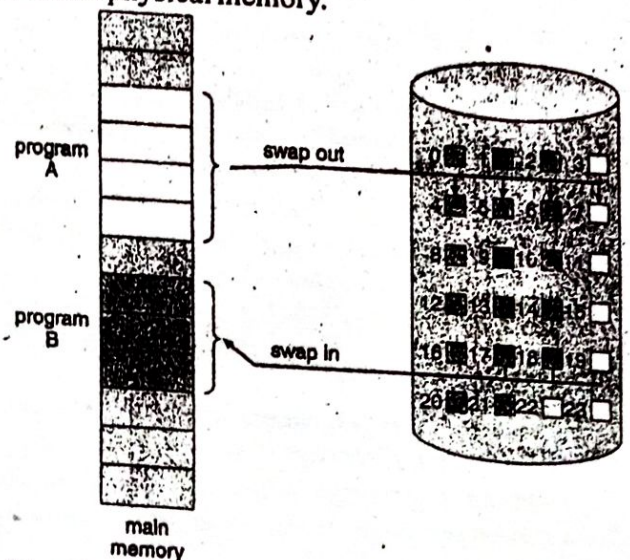


Fig. : Transfer of a paged memory to contiguous disk space

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  $\text{Sqrt}()$ -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  $\text{Sqrt}()$  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.

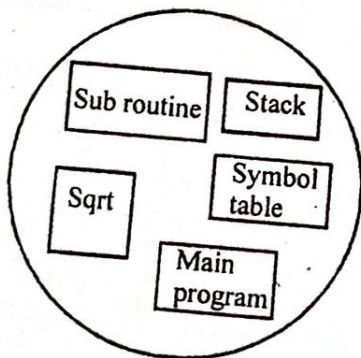


Fig. 1 : Logical address user's view of a program

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:-

$\langle \text{Segment\_number, offset} \rangle$

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 five 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 reference byte 53 of segment 2 is mapped on location  $4300 + 53 = 4353$ . A reference 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.

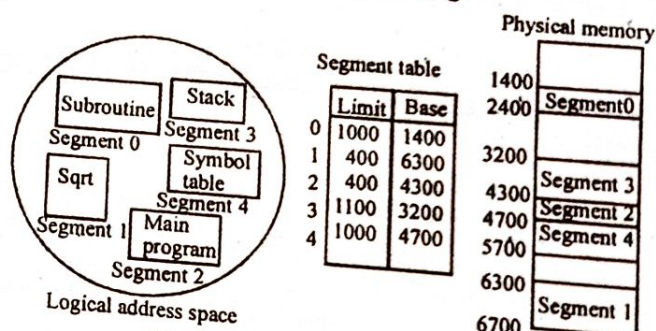


Fig. 2 : Example of segmentation

**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.



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.

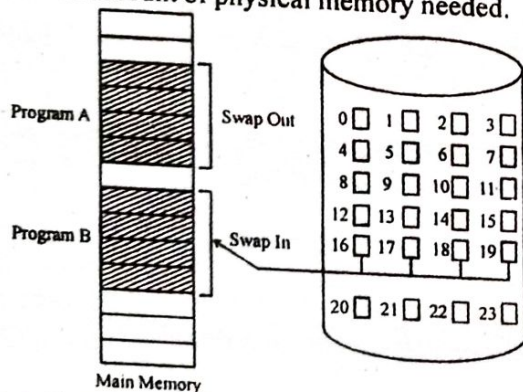


Fig. 1 : Transfer of a paged memory to contiguous disk space

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.

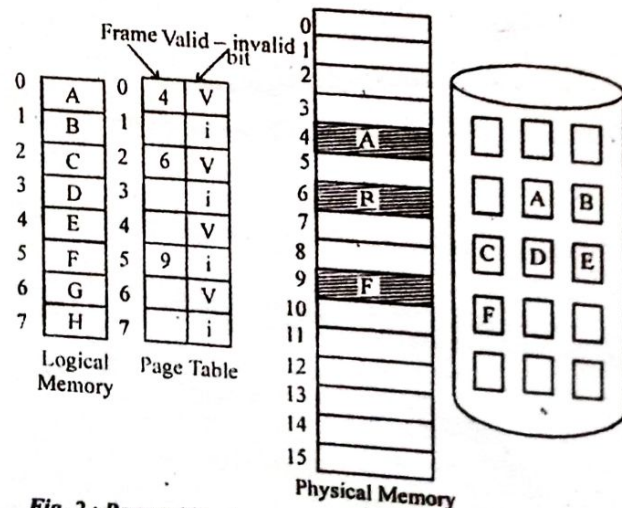


Fig. 2 : Page table when some pages are not in main memory



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 altogether.

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 process 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



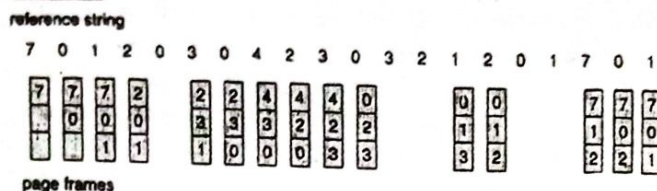


Fig. 1 : FIFO page-replacement algorithm

(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

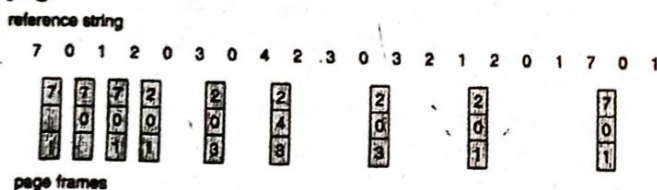


Fig. 2 : Optimal page-replacement algorithm

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.

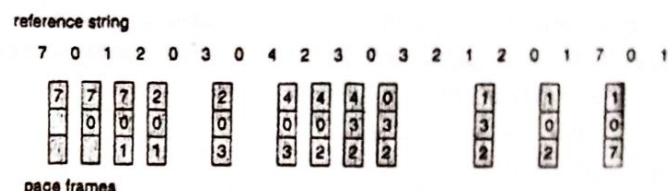


Fig. 3 : LRU page-replacement algorithm

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 then 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 reference pages as it executes. Access graphs can be created by analyzing a compiled program to determine which pages can be accessed by



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

**Q.5 What do you mean by FCFS scheduling.**

**Ans. First Come First Serve (FCFS) :** It is the simplest form of scheduling operations performed in order requested. No recording of request queue. No starvation i.e. every process is serviced.

### PART-B

**Q.6 Suppose that a disk drive has 200 cylinders, numbered 0 to 199. The drive is initially at cylinder 53. The queue with request from I/O to blocks in cylinders 98 183 37 122 14 124 65 67 Count the total head movement of cylinders in SCAN and C-SCAN scheduling.**

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

**Ans. (i) SCAN Scheduling :** The drive is initially at cylinder 53.

queue = 98, 183, 37, 122, 14, 124, 65, 67

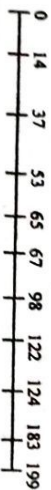


Fig.1

In SCAN scheduling the direction of head movement in addition to the head's current position 53. If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of disk. (Fig.(1))

Total head movement of cylinders in SCAN scheduling

$$= 16+23+14+65+2+31+24+2+59+16 = 232 \text{ tracks}$$

**(ii) C-SCAN Scheduling :** The drive is initially at cylinder 53

queue = 98, 183, 37, 122, 14, 124, 65, 67

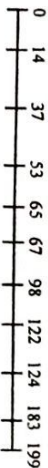


Fig.2

In C-SCAN scheduling movement the head from one end of the disk to the other, servicing request along the way. When the head reaches the other end, it immediately returns to the disk, without servicing any request on the return trip. (Fig.(2))

Total head movement of cylinders in C-SCAN scheduling  
 $= 12+2+31+24+2+59+16+199+14+23 = 382 \text{ tracks}$

**Q.7 Explain the concept of spooling with all its types and its advantage and disadvantages.**

[R.T.U. 2018, 2016]

**Ans. Spooling :** Spooling is an acronym for simultaneous peripheral operation On line. When the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called spooling.

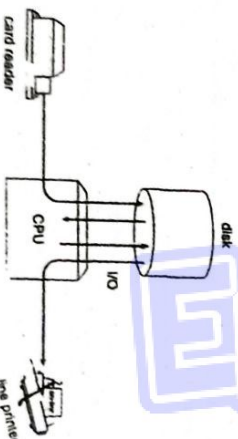


Fig.

Spooling, in essence, uses the disk as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

Spooling is also used for processing data at remote sites. The CPU sends the data via communication paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or other jobs) may be executed, reading its "cards" from disk and "printing" its output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job can overlap with the I/O of other jobs. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates. Spooling leads naturally to multiprogramming which is the foundation of all modern operating systems

### Types of Spoolers

#### 1. Print Spooler

A software program is responsible for managing all the current printing jobs which are sent to the computer printer or print server. The print spooler program may allow a user to delete a print job being processed or otherwise manage the print jobs currently waiting to be printed.

#### 2. The System V-style Spooler

The system V-style spooler of printing subsystem uses system V-Release 4 commands, queues, and files and is administered the same way. Typical user commands available to the system V-style spooler are:

- **lp :** the user command to print a document
- **lpstat :** shows the current print queue
- **cancel :** deletes a job from the print queue
- **lpadmin :** a system administration command that configures the print system
- **lpmove :** a system administration command that moves jobs between print queues

#### 3. The Berkeley-style Spooler

The Berkeley-style spoolers are one of several standard architectures for printing on the Unix platform. It originated in 2.10BSD, and is used in BSD derivatives such as FreeBSD, NetBSD, OpenBSD, and DragonFly BSD. A system running this print architecture could traditionally be identified by the use of the user command

**lpr** as the primary interface to the print system, as opposed to the system V-style **lp** command. Typical user commands available to the Berkeley-style spooler are:

- **lpr :** the user command to print
- **lpq :** shows the current print queue
- **lprm :** deletes a job from the print queue

#### 4. CUPS-based Spooler

CUPS (Common UNIX Printing System) is a new type of spooler. It was designed to work across most UNIX and Linux-based system. It is also standards based. It enables printing through RFC1179 (lpr), IPP, CIFS/SMB, Raw socket (JetDirect), and through local printing. CUPS uses network printer browsing and Postscript Printer Description (PPD) files to ease the common tasks of printing.

#### Advantages of Spooling

The advantages of spooling are as follows:

- The spooling operation uses a disk as a very large buffer.
- Spooling is capable of overlapping I/O operation for one job with processor operations for another job.
- Processes are not suspended for a long time.
- It can produce multiple copies of the output, without running the process again.

#### Disadvantages of Spooling

The disadvantages of spooling are as follows:

- Need large amounts of disk space.
- Increase disk traffic.
- Not practical for real-time environment, because results are produced at a later time.

**Q.8 Explain global versus local allocation.**

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

**Ans. Global Versus Local Allocation :** An important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process, that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.



For example, consider an allocation scheme where we allow high - priority processes to select frames from low priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high priority process to increase its frame allocation at the expense of a low priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose its frames for replacement).

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (for example, taking 0.5 seconds for one execution and 10.3 seconds for the next execution) taking 0.5 seconds for one execution and 10.3 seconds for the next execution because of totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory. Thus global replacement generally results in greater system throughput and is therefore the more common method.

Q.9 Explain various disk scheduling algorithms in brief. [R.T.U. 2017, Dec. 2013]

OR

Discuss the following disk scheduling algorithms:

- Shortest Seek Time First
- First Come First Served
- SCAN
- C-LOOK

[R.T.U. 2012]

**Ans. Hard Disk :** A hard disk drive is a collection of plates called platters. The surface of hard disk is made of concentric circles called tracks. Further more, each track is divided into smaller pieces called sectors.

For each I/O output request first head is selected. It is then moved over the destination track. The disk is then rotated to position the desired sector under the head and finally the read/write operation is performed.

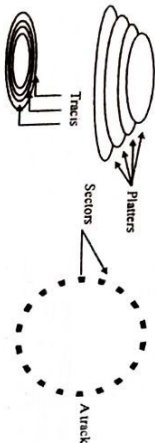


Fig.

The main responsibility of the operating system is to use hardware efficiently. For the disk gives the main responsibility are :

- Maximize the Throughput:** The average number of requests satisfied per unit time.
  - Minimize the Response Time:** The average time that a request must wait before it is satisfied.
- The access time consist of two major components:
- Seek Time:** Seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
  - Rotational Latency:** Rotational latency is the additional time for the disk to rotate the desired sector to the disk head.

**Disk Bandwidth** is the total number of bytes transferred divided by the total time between first request for service and the completion of the last transfer. There are several algorithms exist to schedule the servicing of disk I/O requests illustrate them with a request queue (0 - 199) suppose disk head initially at 100<sup>th</sup> position.

- FCFS Scheduling
  - SSTF Scheduling
  - SCAN Scheduling
  - C-SCAN Scheduling
  - LOOK Scheduling and C-LOOK Scheduling
- (i) **First Come First Serve (FCFS) :** It is the simplest form of scheduling operations performed in order requested. No recording of request queue. No starvation i.e. every process is serviced.

Poor Performance total time estimated by total arm motion  $|100 - 23| + |23 - 80| + |80 - 132| + |132 - 44| + |44 - 188| = 77 + 57 + 52 + 88 + 144 = 418$  Cylinders.

Disk head initially at cylinder 100. It will move from 100 to 23, then to 80, 132, 44 and finally to 188.

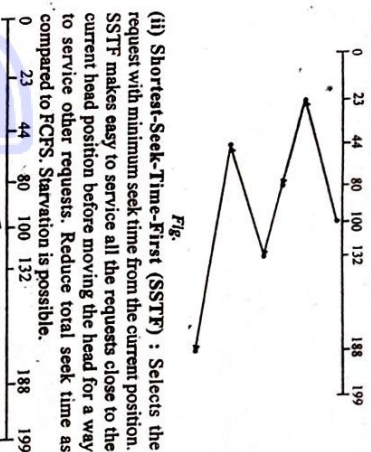


Fig.

(ii) **Shortest-Seek-Time-First (SSTF) :** Selects the request with minimum seek time from the current position. SSTF makes easy to service all the requests close to the current head position before moving the head for a way to service other requests. Reduce total seek time as compared to FCFS. Starvation is possible.

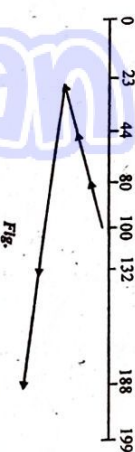


Fig.

The closest request to the initial head position (100) is at cylinder 80. Once we are at cylinder 80, the next closest requests is at 44 from there, the request at cylinder 23 is closer than the one at 132 so then 132 and it finally service 188. Total head movement  $= |100 - 80| + |80 - 44| + |44 - 23| + |23 - 132| + |132 - 188| = 142$  Cylinders.

(iii) **SCAN Scheduling :** The disk arm starts at one end of the disk and moves towards the other end, servicing a requests until it gets to the other end of the disk. When it reaches at the end, the direction of head is reserved and servicing continues. This is also known as Elevator algorithm because the head continuously scans back and forth across the disk. Reduces variance compared to SSTF.

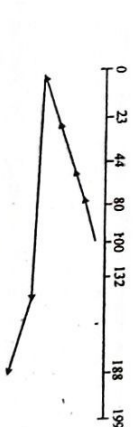


Fig.

If the disk arm is moving towards 0, the head will service 80 and then 44, 23. A cylinder 0, the arm will move towards the other ends of the disk servicing the requests at 132, 188. Total head movement  $= |100 - 80| + |80 - 44| + |44 - 23| + |23 - 0| + |0 - 32| + |32 - 188| = 88$  Cylinders.

(iv) **C-SCAN Scheduling :** Circular Scan Scheduling is a variant of SCAN, to provide information time. Like SCAN it servicing request along the way until the head reaches the end 0 or 199. It immediately moves in reverse direction, without servicing any request.

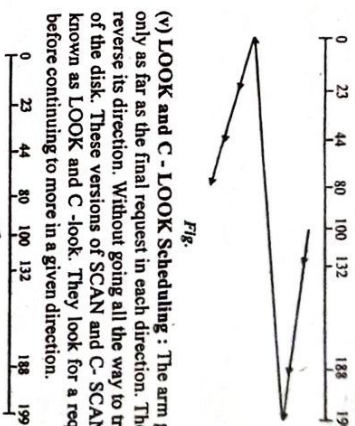
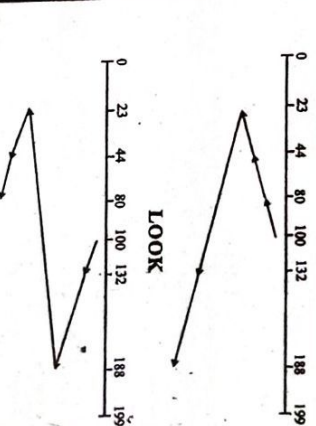


Fig.

(v) **LOOK and C-LOOK Scheduling :** The arm goes only as far as the final request in each direction. Then it reverse its direction. Without going all the way to travel of the disk. These versions of SCAN and C-SCAN all known as LOOK and C-look. They look for a request before continuing to move in a given direction.



LOOK

C-LOOK

Q.10 Apply deadlock detection algorithm to the following data and show the results :

Available = (2, 1, 0, 0)

$$\text{Request} = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

$$\text{Allocation} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

[R.T.U. 2016]

Ans.

- First process 3 is satisfied so it can be completed and resources freed.

Available :  
(2 2 2 0)



OS.57

## PART-C

Q.13 What is deadlock? What are necessary conditions for deadlock to occur?

[R.T.U. 2018, 2013]

OR

What is deadlock? Explain the conditions and prevention of deadlock? [R.T.U. 2017]

OR

What are the different deadlock prevention schemes? Explain. [R.T.U. 2015]

OR

What is deadlock? What are the necessary conditions to occur the deadlock? What are the various methods to recover from the deadlock?

[R.T.U. 2014]

**Ans. Introduction to Deadlocks :** A set of processes is deadlocked if each process in the set is waiting for an event that can cause only by another process in the set.

Because all the processes are waiting, none of them will, even cause any of the events that could wake up any of the other members of the set and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm and then causing events that release other processes in the set.

#### Conditions for Deadlock

**Four essential conditions for deadlock to occur :** Coffman et al. (1971) showed that four conditions must hold for these to be a deadlock :

**1. Mutual exclusion condition :** Each resource is either currently assigned to exactly one process or is available.

**2. Hold and wait condition :** Processes currently holding resource granted earlier can request new resources.

**3. No preemption condition :** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

**4. Circular wait condition :** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of chain.

OS.58

All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

**Deadlock Prevention :** Deadlock avoidance is essentially impossible because it requires information about future requests. Different deadlock prevention schemes are as follows :

(i) **Attacking the mutual exclusion condition :** If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos.

By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

(ii) **Attacking the hold and wait condition :** The second conditions stated by Coffman et al looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If every thing is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they started running. In fact, if they knew, the Banker's algorithm could be used. Another problem is that the resources will not be used optimally with this approach. Nevertheless, some mainframe batch system requires the user to list all the resources on the first line of each job. The system then acquires all resources immediately and keeps them until the job finishes. While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks.

(iii) **Attacking the no preemption condition :** Attacking the third condition (no preemption) is even less promising than attacking the second one. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst.

(iv) **Attacking the circular wait condition :** The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one for a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.



- Available[j] = k means there are 'k' instances of resource type  $R_j$

Max :

- It is a 2-d array of size ' $n \times m$ ' that defines the maximum demand of each process in a system.
- $\text{Max}[i, j] = k$  means process  $P_i$  may request at most 'k' instances of resource type  $R_j$ .

Allocation :

- It is a 2-d array of size ' $n \times m$ ' that defines the number of resources of each type currently allocated to each process.

- Allocation[i, j] = k means process  $P_i$  is currently allocated 'k' instances of resource type  $R_j$ .

Need :

- It is a 2-d array of size ' $n \times m$ ' that indicates the remaining resource need of each process.

- Need [i, j] = k means process  $P_i$  currently allocated 'k' instances of resource type  $R_j$ .
- Need [i, j] =  $\text{Max}[i, j] - \text{Allocation}[i, j]$

Allocation, specifies the resources currently allocated to process  $P_i$  and Need, specifies the additional resources that process  $P_i$  may still request to complete its task.

Banker's algorithm consist of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

- Let Work and Finish be vectors of length 'm' and 'n' respectively.  
Initialize: Work = Available  
Finish[i] = false; for  $i=1, 2, 3, \dots, n$
- Find an i such that both  
a. Finish[i] = false  
b.  $\text{Need}_i \leq \text{Work}$  if no such i exists goto step (4)
- Work = Work + Allocation  
Finish[i] = true  
goto step (2)
- if finish[i] = true for all i  
then the system is in a safe state

#### Resource-Request Algorithm

Let Request<sub>i</sub> be the request array for process  $P_i$ . Request<sub>i</sub>[j] = k means process  $P_i$  wants k instances of resource type  $R_j$ . When a request for resources is made by process  $P_i$ , the following actions are taken:

- If Request<sub>i</sub>  $\leq$  Need<sub>i</sub>  
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

#### Operating System

- If Request  $\leq$  Available Goto step (3); otherwise,  $P_i$  must wait, since the resources are not available.
- Have the system pretend to have allocated the requested resources to process  $P_i$  by modifying the state as follows:  
Available = Available - Request<sub>i</sub>  
Allocation<sub>i</sub> = Allocation<sub>i</sub> + Request<sub>i</sub>  
Need<sub>i</sub> = Need<sub>i</sub> - Request<sub>i</sub>

Allocation				Maximum				Available				
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

#### Step 1 - Calculate Need Matrix:-

Need = Max - Allocation

	A	B	C	D
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

#### Step 2

Predicting Safe Sequence-

- Work = Available  
= 1, 5, 2, 0

Finish = 

F	F	F	F	F
P0	P1	P2	P3	P4

- For i = 0  
Need P0 = 0, 0, 0, 0  
No allocation needed P0 is kept in safe sequence.  
Work = Work + Allocation P0  
= 1, 5, 2, 0 + 0, 0, 1, 2  
= 1, 5, 3, 2

Finish [P0] = T  
For i = 1  
Need P1 = 0, 7, 5, 0  
Need P1 > Work  
Need P1 = > 1, 5, 3, 2  
 $\therefore P1$  = must wait

- For i = 2  
Need P2 = 1, 0, 0, 2  
Need P2 < work  
1, 0, 0, 2  $\leq$  1, 5, 3, 2  
P2 is kept in safe sequence  
work = work + Allocation  
= 1, 5, 3, 2 + 1, 3, 5, 4  
= 2, 8, 8, 6  
Finish [P2] = T

- For i = 3  
Need P3 < Work  
P3 is kept in safe sequence  
work = 2, 8, 8, 6 + 0, 6, 3, 2  
= 2, 14, 11, 8  
Finish [P3] = T

- for i = 4  
Need P4 < work  
P4 is kept in safe sequence  
work = 2, 14, 11, 8 + 0, 0, 1, 4  
= 2, 14, 12, 12  
Finish [P4] = T

- For i = 5  
Need P1 < work  
P1 is kept in safe sequence  
work = 2, 14, 12, 12 + 1, 0, 0, 0  
= 3, 14, 12, 12  
Finish [P1] = T

Finish = 

T	T	T	T	T
P0	P1	P2	P3	P4

Hence, the system is in safe state and safe sequence = P0, P2, P3, P4, P1,

- P1 request resource as  
A B C D  
1, 0, 2, 1

Using Resource Request Algo-

Request P1 < Need P1

1, 0, 2, 1 < 0, 7, 5, 0

This statement is false

Hence resource request will not be full filled.

Q.16 What do you mean by disk scheduling? Suppose the head of moving head disk is currently servicing a request at track 60. If the queue of request is kept in FIFO order. What is the total head movement to satisfy these requests for the following disk scheduling algorithm:

- FCFS
- SCAN
- C-SCAN

Request Sequence	Track Number
1	55
2	175
3	30
4	125
5	10
6	140



Random access files are essential for many applications, for example, database systems.

In a banking application, a customer may want to look up his current balance. This can be easily done by locating this customer's record using his account number as a key, rather than sequentially reading the records for thousands of other customers before this customer's record is located and read.

It is worth mentioning here that not all operating systems support both sequential and random access for files. Some systems allow only sequential file access, others allow only random access. Some systems require that a file should be defined as sequential or random when it is created, so that it can be accessed in the way it has been declared.

**Q.7 Define file system? Explain file operations in detail?**  
(R.T.U. 2016)

**Ans. File System :** A file system is a method of organizing files on physical media, such as hard disks, CD's, and flash drives. In the Microsoft Windows family of operating systems, users are presented with several different choices of file systems when formatting such media. These choices depend on the type of media involved and the situations in which the media is being formatted. Common file systems in Windows are as follows :

- NTFS
- FAT
- exFAT
- HFS +
- EXT

#### File Operations :

A file is an abstract data type. Operation on file, operating system provides system calls for creating, deleting, read etc. Basic operations on files are :

1. Create a file
2. Writing a file
3. Reading a file
4. Delete a file
5. Truncating a file
6. Repositioning within a file

**1. Create a file :** For creating a file, address space in the file system is required. After creating a file, entry of the file is made in the directory. The directory entry records the name of the file and the location in the file system.

**2. Writing a file :** System call is used for writing into file. It is required to specify the name of the file and information to be written to the file. According to the file name, system

will search the name in the directory to find the location of the file.

**3. Reading a file :** To read a file, system call is used. It requires the name of file and memory address. Again the directory is searched for the associated directory entry and the system needs to keep a read pointer to the location in the file where the next read is to take place.

**4. Delete a file :** System will search the directory, which file to be deleted. If directory entry is found, it releases all file space. That free space can be reused by another (user) files.

**5. Truncating a file :** Refer to Q.4.

**6. Repositioning within a file :** The directory is searched for the appropriate entry, and the current file position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as file seek.

**Q.8 Explain various features of file system of linux.**  
(R.T.U. 2015)

**Ans. In Linux,** everything is configured as a file. This includes not only text files, images and compiled programs (also referred to as executables), but also directories, partitions and hardware device drivers.

Each filesystem contains a control block, which holds information about that filesystem. The other blocks in the filesystem are inodes, which contain information about individual files and data blocks, which contain the information stored in the individual files.

There is a substantial difference between the way the user sees the Linux filesystem and the way the kernel (the core of a Linux system) actually stores the files. To the user, the filesystem appears as a hierarchical arrangement of directories that contain files and other directories (i.e., subdirectories). Directories and files are identified by their names. This hierarchy starts from a single directory called root, which is represented by a "/" (forward slash).

The Filesystem Hierarchy Standard (FHS) defines the main directories and their contents in Linux and other Unix-like operating systems. All files and directories appear under the root directory, even if they are stored on different physical devices (e.g., on different disks or on different computers). A few of the directories defined by the FHS are /bin (command binaries for all users), /boot (boot loader files such as the kernel), /home (users home directories), /mnt (for mounting a CD-ROM or floppy disk),



**Q.11 Write short note on File organization.**

**[R.T.U. 2014]**

**Ans. File Organization :** A file consists, generally speaking, of a collection of records, a key element in file management is the way in which the records themselves are organized inside the file, since this heavily affects system performances and far as record finding and access. Note carefully that by "organization" we refer here to the logical arrangement of the records in the file (their ordering or, more generally, the presence of "closeness" relations between them based on their content), and not instead to the physical layout of the file as stored on a storage media, To prevent confusion, the latter is referred to by the expression "record blocking", and will be treated later on.



Choosing a file organization is a design decision, hence it must be done having in mind the achievement of good performance with respect to the most likely usage of the file. The criteria usually considered important are:

1. Fast access to single record or collection of related records.
2. Easy record adding/update/removal, without disrupting.
3. Storage efficiency.
4. Redundance as a warranty against data corruption.

Logical data organization is indeed the subject of whole shelves of books, in the "Database" section of your library. Here we'll briefly address some of the simpler used techniques, mainly because of their relevance to data management from the lower-level (with respect to a database's) point of view of an OS. Five organization models will be considered:

- Pile.
- Sequential.
- Indexed-sequential.
- Indexed.
- Hashed.



# PREVIOUS YEARS QUESTIONS

## PART-A

**Q.1** Describe the function of process scheduler in Linux OS.

**Ans.** Process Scheduler (SCHED) is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time.

**Q.2** Why NFS protocol is used?

**Ans.** The NFS protocol provides a set of RPCs for remote file operations. The protocol support the following operations :

- Searching for a file within a directory
- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

**Q.3** Write any two difference between Unix and Linux.

**Ans.** Difference between Unix and Linux:

	Unix	Linux
1.	UNIX is propriety system.	Linux is an Open Source system.
2.	Development is targeted toward specific audience and platform.	Linux development is diverse.

**Q.4** Define C-shell.

**Ans. C Shell :** The C shell, as its name might imply, was designed to allow users to write shell script programs using a syntax very similar to that of the C programming language. It is known as *csh*.

**Q.5** What do you mean by palm OS.

**Ans. Palm OS :** Palm OS is designed for ease of use with a touchscreen-based graphical user interface.

**Q.6** Describe the usage and functionality of the command "*rm -r \**" in UNIX?

**Ans.** The command "*rm -r \**" is a single line command to erase all files in a directory with its subdirectories.

- "*rm*" - Is for deleting files.
- "*-r*" - Is to delete directories and subdirectories with files within.
- "*\**" - Is indicate all entries.

**Q.7** Describe *fork ( )* system call.

**Ans.** The command use to create a new process from an existing process is called *fork ( )*. The main process is called parent process and new process id called child process. The parent gets the child process id returned and the child gets 0. The returned values are used to check which process which code executed. The returned values are used to check which process which code executed.

**Q.8** What is UNIX?

**Ans.** It is a portable operating system that is designed for both efficient multi-tasking and multi-user functions. Its portability allows it to run on different hardware platforms. It was written in C and lets users do processing and control under a shell.



**Processes and Threads : Operating System Concept**  
Processes and Threads Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent. Thread creation is done through clone() system call. Clone() allows a child task to share the address space of the parent task(process). The flags provided to clone() help specify the behaviour of the new process and detail what resources the parent and child will share.

Table 1 : Clone() Flags

Flags	Description
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share file-system information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PTRACE	Continue tracing child.
CLONE_SIGHAND	Parent and child share signal handlers.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTSL	Create a new TLS (thread-local storage) for the child.
CLONE_VM	Parent and child share address space.

**System Calls used for Process Management :** The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions. The system call is the means by which a process requests a specific kernel service. Some of the system calls for process management in Linux operating system are given below in Table 2.

Table 2 : System Calls for Process Management

System call	Description
Fork()	Used to create a new process.
Exec()	Execute a new program.
Wait()	Wait until the process finishes execution.
Exit()	Exit from the process.
Getpid()	Get the unique process ID of the process.
Getppid()	Get the parent process unique ID.
Nice()	To bias the existing property of process.

Process Scheduling - Refer to Q. 9

Ans. Unix Vs Linux

Unix vs Windows

Both of the operating systems are differ at the conceptual level, as process management, memory management,

### Difference between Unix and Linux

	Unix	Linux
1.	UNIX is propriety system.	Linux is an Open Source system.
2.	Development is targeted toward specific audience and platform.	Linux development is diverse.
3.	UNIX maintains consistency between different versions.	Linux have inconsistencies between versions.
4.	In UNIX developers are bounded by standard.	Linux developers are free and have no restriction.
5.	In UNIX commands, tool and utilities etc. are rarely changed over versions.	In Linux commands, tools and utilities may change over time.
6.	UNIX was coded for small handful hardware platform/architecture.	Linux was designed to be as compatible as possible. Runs on dozens of architecture and support numerous I/O devices & other external devices. Supported devices are limitless.
7.	UNIX kernel is not freely available.	Linux kernel is freely available.
8.	UNIX patches available are highly tested.	Linux patches are not highly tested as UNIX patches.
9.	Commercial UNIX is usually custom written for each system, making the original cost quite high.	Linux has base packages.
10.	User has to wait for a while, to get the proper bug fixing patch.	Threat detection and solution is very fast.
11.	Different flavors of Unix have different cost structures.	Linux can be freely distributed.
12.	A rough estimate of Unix viruses is between 85-120 viruses reported till date.	Linux has had about 60-100 viruses listed till date.
13.	Market share of Unix is less than 0.5 percent of the pc market.	The Market share of Linux is about 0.8%.

Q.21 (a) Explain network file system and its implementation.

(b) Explain the following :

(i) Inter-Process communication

(ii) Booting & login process

[R.T.U. 2015]



**Ans.(a)** A Network File System (NFS), the network file system, is probably the most prominent network services using RPC. It allows to access files on remote hosts in exactly the same way as a user would access any local files. This is made possible by a mixture of kernel functionality on the client side (that uses the remote file system) and an NFS server on the server side (that provides the file data). This file access is completely transparent to the client, and works across a variety of server and host architectures.

**NFS offers a number of advantages:**

- (i) Data accessed by all users can be kept on a central host, with clients mounting this directory at boot time. For example, you can keep all user accounts on one host, and have all hosts on your network mount /home from that host. If installed alongside with NIS, users can then log into any system, and still work on one set of files.
- (ii) Data consuming large amounts of disk space may be kept on a single host. For example, all files and programs relating to LaTeX and METAFONT could be kept and maintained in one place.
- (iii) Administrative data may be kept on a single host. No need to use rcp anymore to install the same stupid file on 20 different machines.

When someone accesses a file over NFS, the kernel places an RPC call to `nfdsd` (the NFS daemon) on the server machine. This call takes the file handle, the name of the file to be accessed, and the user's user and group id as parameters. These are used in determining access rights to the specified file. In order to prevent unauthorized users from reading or modifying files, user and group ids must be the same on both hosts.

On most implementations, the NFS functionality of both client and server are implemented as kernel-level daemons that are started from user space at system boot. These are the NFS daemon (`nfdsd`) on the server host, and the Block I/O Daemon (`biod`) running on the client host. To improve throughput, `biod` performs asynchronous I/O using read-ahead and write-behind; also, several `nfdsd` daemons are usually run concurrently.

The NFS implementation of is a little different in that the client code is tightly integrated in the virtual file system (VFS) layer of the kernel and doesn't require additional control through `biod`. On the other hand, the server code runs entirely in user space, so that running several copies of the server at the same time is almost impossible because of the synchronization issues this would involve. NFS currently also lacks read-ahead and write-behind, but Rick Sladkey plans to add this someday.

The biggest problem with the NFS code is that the kernel as of version 1.0 is not able to allocate memory in chunks bigger than 4K; as a consequence, the networking code cannot

handle data grams bigger than roughly 3500 bytes after subtracting header sizes etc. This means that transfers to and from NFS daemons running on systems that use large UDP data grams by default (e.g. 8K on SunOS) need to be downsized artificially.

**Ans.(b) (i) Linux Inter-Process Communication :** Inter-Process Communication, which in short is known as IPC, deals mainly with the techniques and mechanisms that facilitate communication between processes. Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix release in which they first appeared.

The types of inter process communication are :

1. **Signals :** Sent by other processes or the kernel to a specific process to indicate various conditions.
2. **Pipes :** Unnamed pipes set up by the shell normally with the "|" character to route output from one program to the input of another.
3. **FIFOs :** Named pipes operating on the basis of first data in, first data out.
4. **Message queues :** Message queues are a mechanism set up to allow one or more processes to write messages that can be read by one or more other processes.
5. **Semaphores :** Counters that are used to control access to shared resources. These counters are used as a locking mechanism to prevent more than one process from using the resource at a time.
6. **Shared memory :** The mapping of a memory area to be shared by multiple processes.

Message queues, semaphores and shared memory can be accessed by the processes if they have access permission to the resource as set up by the object's creator. The process must pass an identifier to the kernel to be able to get the access.

**(ii) Booting and Login Process :** When a PC is booted, it starts running a BIOS program which is a memory resident program on an EEPROM integrated circuit. The BIOS program will eventually try to read the first sector on a booting media such as a hard or floppy drive. The boot sector contains a small program that the BIOS will load and attempt to pass run control to. This program will attempt to read the operating system from the disk and run it. LILO is the program that Linux systems typically use to give users a choice of operating systems to run. It is usually installed in the boot sector which is also called the master boot record.

### Booting

In Linux, the flow of control during a boot is from BIOS, to boot loader, to kernel. The kernel then starts the scheduler (to allow multi-tasking) and runs the first userland (i.e. outside



blocks. There are no system control blocks accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The information in these control blocks is used by the kernel for process control and CPU scheduling.

### Process Control Blocks

The most basic data structure associated with processes is the process structure. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (such as the priority of the process) and pointers to other control blocks. There is an array of process structures whose length is defined at system linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue) and there are pointers from each process structure to the process parent, to its youngest living child and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The virtual address space of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but may grow separately and usually in opposite directions: most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack and is usually read only. The debugger puts a text segment in read-write mode to be able to allow insertion of breakpoints.

Every process with sharable text (almost all, under FreeBSD) has a pointer from its process structure to a text structure. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory: an array of such structures is allocated at system link time. The text, data and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The page tables record information on the mapping from the process virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process page table.

Information about the process that is needed only when the process is resident (that is, not swapped out) is kept in the user structure (or u structure), rather than in the process structure.

Every process has both a user and a system mode. Most ordinary work is done in user mode, but when a system call is made, it is performed in system mode. The system and

user phases of a process never execute simultaneously. When a process is executing in system mode, a kernel stack for that process is used, rather than the user stack belonging to that process. The Kernel stack for the process immediately follows the user structures. The kernel stack and the user structure together compose the system data segment for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure illustrates how the process structure is used to find the various parts of a process.

The fork system call allocates a new process structure (with a new process identifier) for the child process and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text; the appropriate counters and lists are merely updated. A new page table is constructed and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling and most similar properties of a process.

The vfork system call does not copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call is by a shell to execute a command and to wait for its completion. The parent process uses vfork to produce the child process. Because the child process wishes to use an execve immediately to change its virtual address space completely, there is no need for a complete copy of the parent process. Such data structure as are necessary for manipulating pipes may be kept in registers between the vfork and the execve.

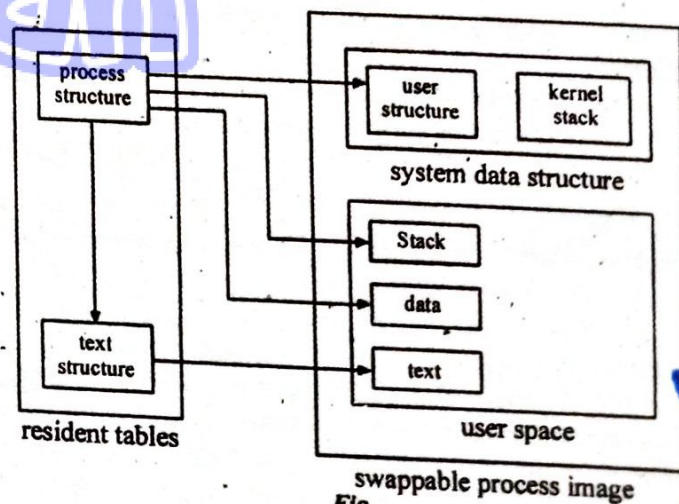


Fig. swappable process image

When the parent process is large, vfork can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory changes occurs in both processes until the execve occurs. An alternative is to share all pages by duplicating the page table, but to mark the entries of both page tables as copy-on-write.