Computer systems:

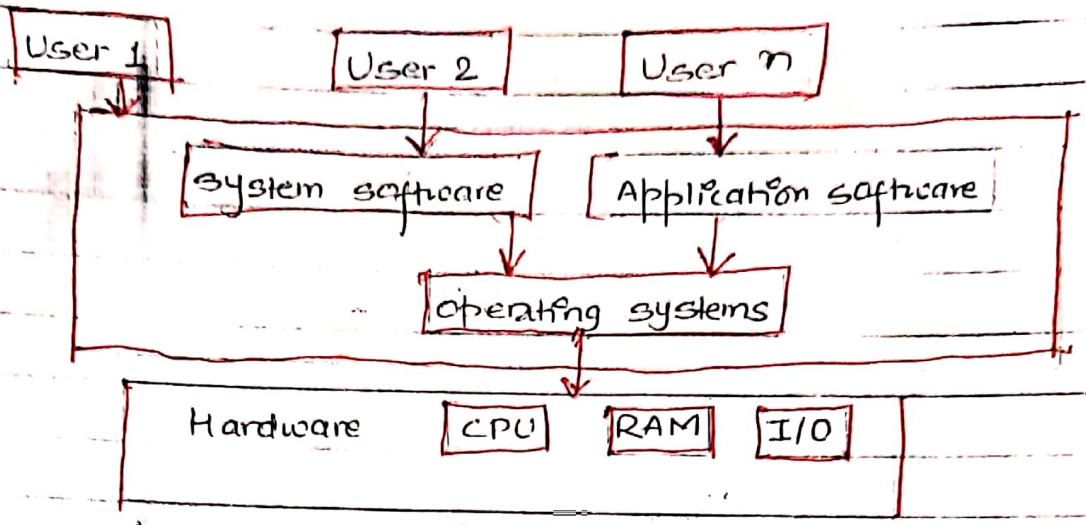


fig: Abstract view of a computer system

1) Hardware

→ It provides basic computing resources (CPU, memory, I/O devices)

2) Operating system: It controls and co-ordinates the use of the hardware among the various application programs for the various users.

3) Application Programs: Defines the ways in which the system resources are used to solve the computing problems of the users (video games, business programs, etc).

4) Users: People, Machine (other computers)

INTRODUCTION TO AN OPERATING SYSTEMS.

→ An operating system is an intermediary between user and computer hardware.

– It provides users an environment in which a user can

execute programs conveniently and efficiently.
- It is a software which manages the hardware.
- An operating system is a program that controls the allocath
- cation of resources and services such as memory, processors, de
devices and information.

OPERATING SYSTEM GOALS:
- Execute user programs and make solving user problems easi
easier
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.
- Provides an environment within which other programs can
can do useful work.

Operating systems can be defined as:
IMP (a) An extended machine (virtual machine)
(b) A resource manager.

(a) OS as an extended machine:
- It provides stable, portable, reliable, safe, well-behaved
- ed environment.
- It makes computer appear to be more than it really is.
- OS shields the programmer from the disk hardware and presents a simple file oriented interface.
- OS function is to present the user with the equivalent of
of an extended machine or virtual machine that is easier to program than the underlying hardware:
- It creates higher level of abstraction for programmer.
E.g. floppy disk I/O operation,
- Disks contain a collection of named files.

- Each file must be opened for read/write.
- After read/write ~~operation~~ completion, close that file.
- Programmer doesn't need deal in any detail.

(b) OS as a Resource Manager.

- The job of the operating system is to provide for an orderly and controlled allocation of the processes, memories and I/O devices among the various program competing for them.
- Resource management includes sharing resources in two ways: Time and space.
- When a resource is time multiplexed, different programs or users take turns using it.

For example: With only one CPU and multiple programs that want to run on it, the OS first allocated the CPU to one program, then ~~to run on it, the OS~~ after it eventually the first one again.

- The other kind of multiplexing is space multiplexing.
- Instead of the customers taking turns each one gets part of the resources.

For example: Main memory is normally divided up among the several running programs, so each one can be resident at the same time.

Functions of an operating system.

1. Memory management
2. CPU management.
3. Device management.
4. File Management.
5. Inter Process communication (IPC)
6. Security

(7) Error Detection.
(8) Shell / command Interface.
(9) I/O management

ASSIGNMENT NO. 1

Functions of an Operating System.

(1) Memory management.

The memory manager in an OS co-ordinates the memories by tracking which one is available, which is to be allocated or deallocated and how to swap between the main memory and secondary memories. The operating system tracks all memory used by each process so that when a process terminates, all memory used by the process will be available for other processes.

(2) CPU management.

It deals with running multiple processes. Most operating system allow a process to be assigned a priority which affects its allocation of CPU time. Interactive operating systems also employ some level of feedback in which the task with which the user is working receives higher priority. In many systems there is a background process which runs when no other process is waiting for CPU.

~~(3) Device management~~

~~Operating systems have a variety of native file~~

(3) Device management / Disk management.

Disk is the permanent storage unit and stores loads of data permanently. Operating system manages the efficient sectors in the disks for storing the files and data. Operating systems have a variety of native file systems that controls the creation, deletion, and access of files of data and programs.

(4) File management

The operating system also handles the organisation and tracking of files and directories (folders) saved or retrieved from a computer disk. The file management system allows the user to perform such tasks as creating files and directories, renaming files, copying and moving files and deleting files. The operating system keeps track of where files are located on the hardware through the type of file system.

(5) Inter Process Communication (IPC)

Many programs resides in computer. One program can contain many processess. Operating system manages the synchronization between two or more processes (multiple processes). If the processes are not synchronized properly, then there might be the condition of deadlock. So, operating system is used.

(6) Security.

Most operating systems include some level of security. OS provides sufficient support for multi level security and evidence of correctness to meet a particular set of governme requirements. E.g. Firewall helps in the filtration. User management is also the service provided by OS.

⑦ Error Detection.

Operating system also controls the errors those have occurred into the program and also provides recovery of the system when the system gets damaged. When due to some Hardware failure, if the system doesn't work properly then this recover the system and also corrects the system and also provides the back up facility.

⑧ Shell / Command Interface.

Shell is a command-line interface, which means it is solely text-based. The user can type commands to perform functions such as run programs, open and browse directories and view processes that are currently running. Shell / Command interface is an important job of OS. Shell provides the interface between user and computer to interact.

E.g. In DOS, command interpreter is shell.
In windows, windows explorer is shell.

⑨ Input / Output Management.

It means co-ordination and assignment of the different input and output devices while one or more programs are being executed.

OS Evolution:

1st Generation (1945-55) → Use Vaccum Tubes → User Drives
2nd Generation (1955-65) → Transistor - Batch Processing.
3rd Generation (1965-80) - IC → Multiprogramming.
4th Generation (1980-present) - PC → Client server / Distributed systems.

Imp

Types of Operating systems.

Operating systems are there from the very first computer generation. Operating systems keep evolving over the period of time. Following are few of the important types of operating system which are most commonly used.

(a) Batch Operating system.

The users of batch operating system do not interact with the computer directly. Each user prepares his job on an offline device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers leave their programs with the operator. The operator then sorts programs into batches with similar requirements.

1/ Windows

(b) Time sharing operating system

Time sharing is a technique which enables many people located at various terminals to use a particular computer system at the same time. Time sharing or Multitasking is a logical extension of multiprogramming.

no. of times it be executed ⇒ Burst time
complete exceeded time is stored, then that particular deadline is
time is quantum time.

Processors time which is shared among multiple users simultaneously is termed as Time sharing. The main difference between multiprogrammed Batch systems and time sharing systems is that in case of multiprogrammed batch systems, objective is to maximize the processor use whereas in Time sharing systems objective is to minimize response time. Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For e.g.: In a transaction processing, processor execute each user program in a short burst or quantum of computations.

(c) Distributed Operating systems:

Distributed systems use multiple central processors to serve multiple real time application and multiple users. Data Processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently. The processors communicate with one another through various communication lines. These are referred as loosely coupled systems or distributed systems. Processors in a distributed systems may vary in size and function.

(d) Network operating systems:

Network operating system runs in a server and provide server the capability to manage data, users, groups, security, applications and other networking functions. The primary purpose of the network operating system

is to allow shared file and printer access among multiple computers in network. Typically, a local area network (LAN), a private network or other networks. Examples of network operating systems are Microsoft windows server 2003, Microsoft windows server 2008, Unix, Linux, etc.

(e) Real Time operating systems:

Real Time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. Real Time processing is always on line whereas online system need not be real time. The time taken by the system to respond to an input and display of required updated information is termed as response time. So, in this method, response time is very less as compared to the online processing. Real time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real time systems can be used as a control device in a dedicated application. Real time operating system has user defined fixed time constraints otherwise system fails. For example: scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots and home appliance controllers, Air Traffic control systems, etc.

Operating System Structure

(a) Monolithic Systems.

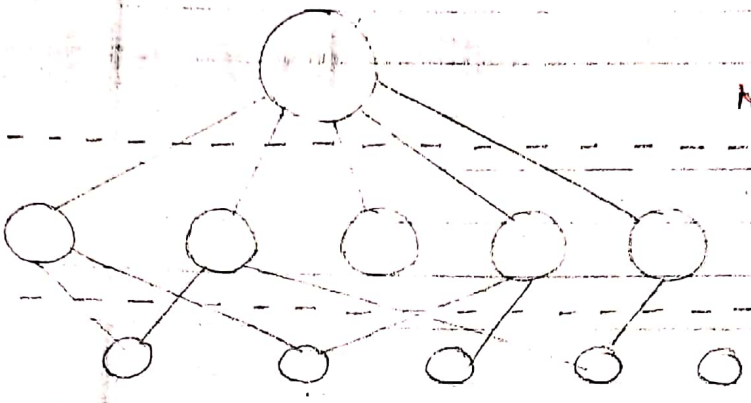


fig: A simple structuring model for a monolithic system

The structure is that there is no structure. The operating system is written as a collection of procedures each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well defined interface in terms of parameters and results. And each one is free to call any other one if the latter provides some useful computation that the former needs. To construct the actual object program of the OS when this approach is used, one first compiles all the individual procedures or files containing the procedures. And then binds them all together into a single object file using the system linker. Even in monolithic systems, however it is possible to have at least a little structure. The services (system calls) provided by the operating

system are requested by putting the parameters in a well defined place. And then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system. The organization suggests a basic structure for the OS.

1) A main program that invokes the requested service provider.
2) A set of services procedures that carry out the system calls.
3) A set of utility procedures that help the service procedures.

(b) Layered systems :-

Layer	Function
5	The operator
4	User programs – Application program – for interaction (interfacing program)
3	Input / output management
2	Operator - Process communication – Inter process comm.
1	Memory and Drum Management
0	Processor Allocation and multiprogramming – Hardware layer

Fig : Layered Approach.

The system has six layers. Layer 0 dealt with the allocation of the processor switching between processes when interrupts occured or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provides the basic multiprogramming of the CPU. Layer 1 did the memory

management and it allocated space for process in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum. The layer 1 software took care of making sure pages were brought in memory whenever they were needed. Layer 2 handled communication between each process and the operator console. Above this layer, each process effectively had its own operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could dealt with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process memory, console or I/O management. The system operator process was located in layer 5.

(c) Microkernel systems:

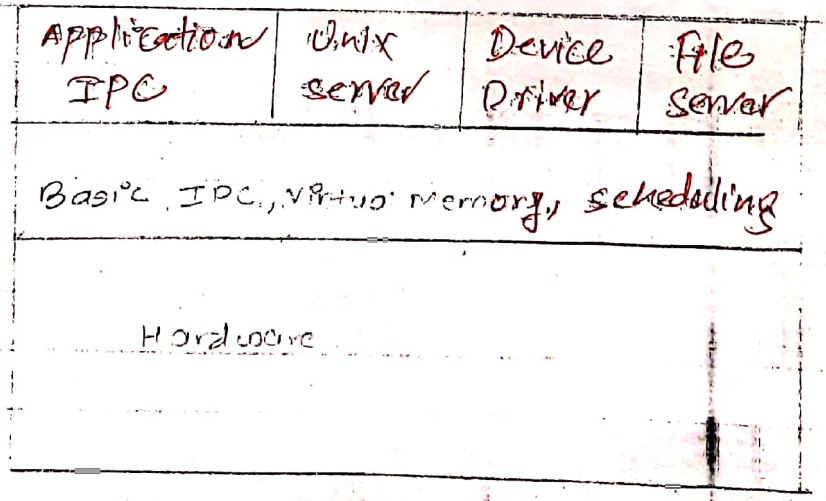


fig: Microkernel based operating System

The microkernel architecture is to define a very simple abstraction over the hardware with a set of primitives or system calls to implement minimal operating system services such as thread management, address spaces and interprocess communication. All other services those normally provided by the kernel such as networking are implemented in user space programs refered to as servers. Servers are programs like any others allowing the operating system to be modified simply by starting and stopping programs. Microkernel's generally underperform traditional design, sometimes dramatically. This is due to large part to the overhead of moving in and out of the kernel, a context switch, in order to move between the various applications and servers.

(d) Virtual Machine

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

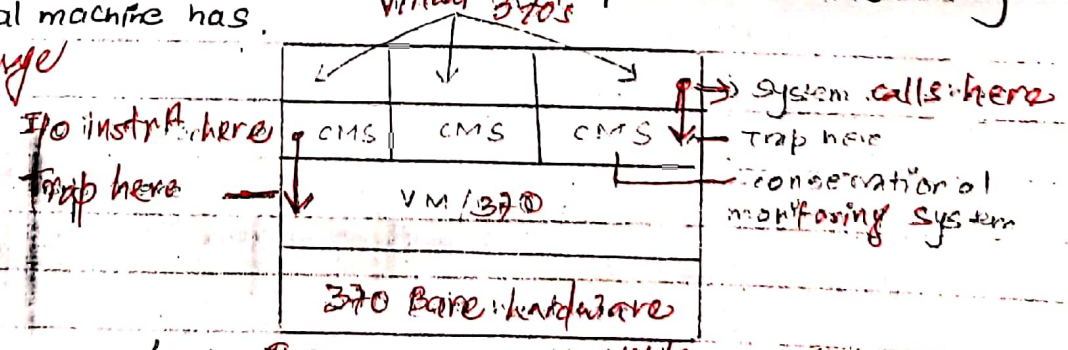


fig: The structure of VM/370 with CMS

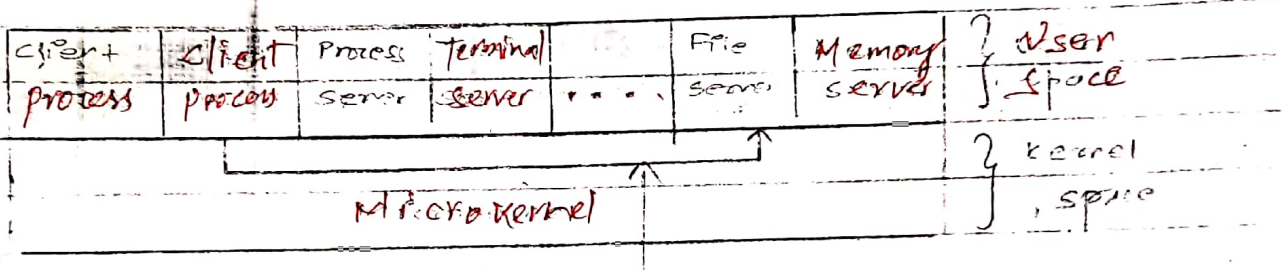
Because each virtual machine is identical to the true hardware, each one can run on any operating system that will run directly bare hardware. Different virtual machines can and frequently do, run different operating systems. Some run one of the descendents of OS/360 for batch or transaction processing, while other ones runs a single user, interactive systems called CMS (Conservational Monitoring system) for interactive time sharing users. When a CMS programs executes a system call, the call is trapped to the operating system in its own virtual machines, not to VM/370 just as it would if it were running on a real machine instead of a virtual one. CMS then issues the normal hardware I/O instructions for reading its virtual disk or whatever is needed to carry out the call. These I/O instructions are trapped by VM/370 which then performing the functions of multiprogramming and providing an extension machine, each of the pieces can be much simpler, more flexible and easier to maintain.

(e) Exokernels :-

At the bottom layer, running in kernel mode is a program called the exokernel. Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources. The advantages of the exokernel scheme is that it saves a layer of mapping. The exokernel need only keep track of which virtual machine has been assigned which resource.

④ Client Server model :

A trend in modern operating system is to take the idea of moving code up into higher layers even further and remove as much as possible from kernel mode. The usual approach is to implement most of the operating system in user processes. To request a service, such as reading a block of a file, a user process (known as client process) sends the request in the server process, which then does the work and sends back the answer.



client obtains service by sending messages to server processes

fig:- Two client-server model.

In this model, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, terminal service, or memory service each part becomes small and manageable. Furthermore, because all the servers runs as user mode processes and not in kernel mode, they do not have direct access to the hardware.

System calls

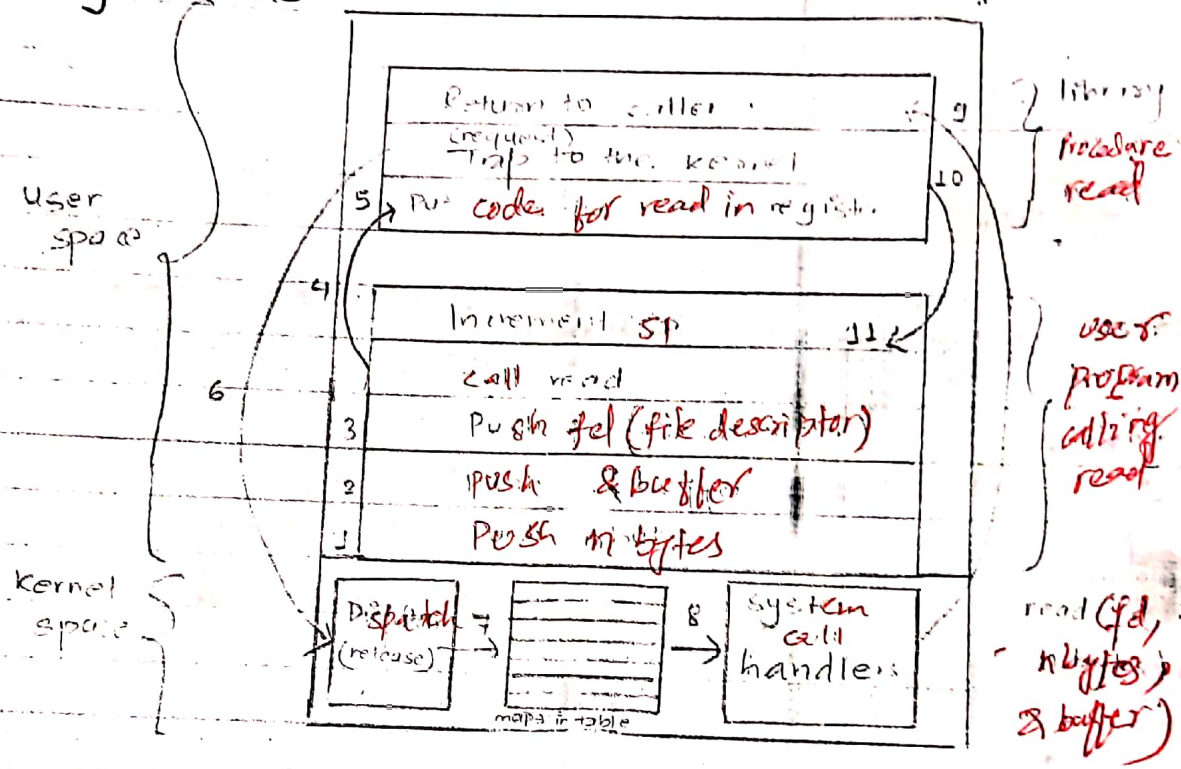


fig. The 11 steps in making a system call

The interface between the operating system and the user program is defined by the set of system calls that the operating system provides. To really understand what operating systems do, we must examine the interface closely. To make the system call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameter :-

- the first one specifying the file.
- the second one pointing to the buffer
- and the third one giving the number of bytes to read.

The system call (and the library procedure) return the number of bytes actually read in count. This value is

normally the same as n bytes, but may be smaller, if for example, end of file is encountered while reading. If the system calls are performed in a series of steps, In preparation for calling the read library procedure which actually makes the read system calls. The calling program first pushes the parameters onto the stack. The first and third parameters are call by value, but the second parameter is pass by reference. The library procedure possibly written in assembly language, typically puts the system calls number in a place, where the OS expects it, such as a register. Then, it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel. The kernel code that starts examine the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers. Once the system call handler has completed its work, control may be returned to the user space library procedures at the instruction following the TRAP instructions. This procedure then returns to the user program in the usual way procedure calls return. To finish the job, the user program has to clean up the stack, as it does after any procedure call. Assuming the stack grows downwards as it often does, the compile code increments the stack pointer exactly enough to remove the parameters pushed before to call the read.

Chapter - 2

Process Management

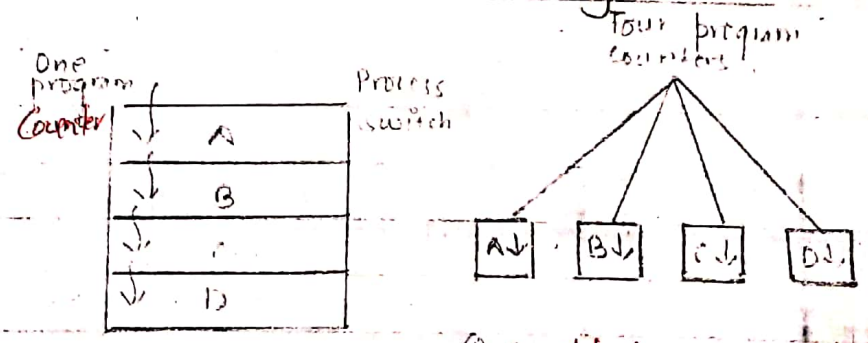
Process :

- A process is a program at execution.
- We are assuming a multiprogramming operating system that can be switched from one process to another.
- Sometimes this is called pseudoparallelism since one has the illusion of a parallel processor.
- The other possibility is a real parallelism in which two or more processes are actually running at once because the computer system is a parallel processor i.e has more than one processor.

concurrently running multiprograms at a time - multi programming
running program in multiple cpu - multi processing.

The process Model.

- Eventhough in actuality, there are many processes running at once, the operating system gives each process a illusion that it is running alone.



(a) Multiprogramming of 4 programs

(b) Conceptual model of four independent sequential processes

[No two process can run at same time]

(c) Only one program is active at once.

(a) In diagram, we see a computer & multiprogramming four programs in memory.

(b) We see four processes, each with its own flow of control (i.e its own logical program counter).

→ And each one running independently of other ones.

→ Of course, there is only one physical program counter.

→ So, when each process runs, its logical program counter is loaded into the real program counter.

→ When it is finished for the time being, the physical program counter is saved in the process logical program counter in memory.

(c) We see that viewed over a long enough time interval, all the processes have made progress but at any given instant only one process is actually running.

→ With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again.

Process Creation:

→ From the users or external viewpoint there are several mechanisms for creating a process.

(1) System initialization, including daemon (background) processes.

(2) Execution of a process creation system call by running process.

(3) A user request to create a new process.

(4) Initiation of a batch job.

fork command → create process

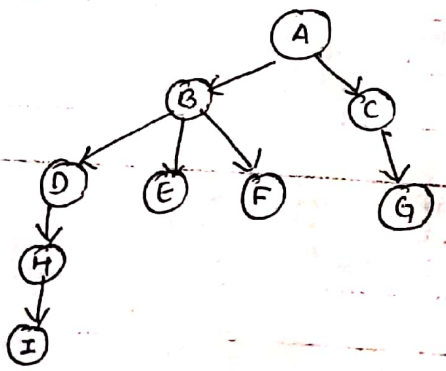
Process Termination:

- From the outside, there appear to be several termination mechanism.

1) Normal Exit (Voluntary)
2) Error Exit (Voluntary)
3) Fatal error (Involuntary)
4) Killed by another Process (Involuntary).

Process Hierarchy

- Modern general purpose operating systems permit a user to create and destroy processes.
- In Unix, this is done by the fork system call which creates a child process and the exit system call which terminates the current process.
- After a fork both parent and child keep running (Indeed they have the same program text) and each can fork off other processes.
- A process tree results: the root of the tree is a special process created by the OS during startup.
- A process can choose to wait for children to terminate. For e.g: If C issued a wait system call, it would block until G finished.



A forks B and C
B forks D, E and F.
C forks G
D forks H
H forks I

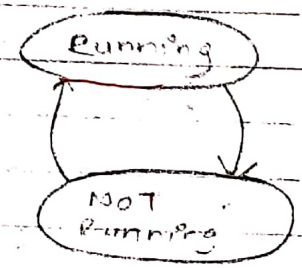
Program + State

Process states (Process - dynamic)

A process goes through a series of discrete process states.

(a) Two state process model.
(b) Three state process model
(c) Five state process model
(d) Seven state process model.

(a) Two state process model:
- Running : Process execution on CPU
- Not Running : Process not execution CPU.



(a) Two state process model

(b) Three state Process Model:

(i) Running state:
→ Process executing on CPU
→ Only one process at a time.

(ii) Ready state: (Ready queue)
→ Process that is not allowed to CPU but is ready to run
→ A list of processes ordered based on priority in a queue

(iii) (c) Blocked/waiting state:
→ Process that is waiting for some event to happen.
e.g: I/O operation.

→ A list of processes (no clear priority)

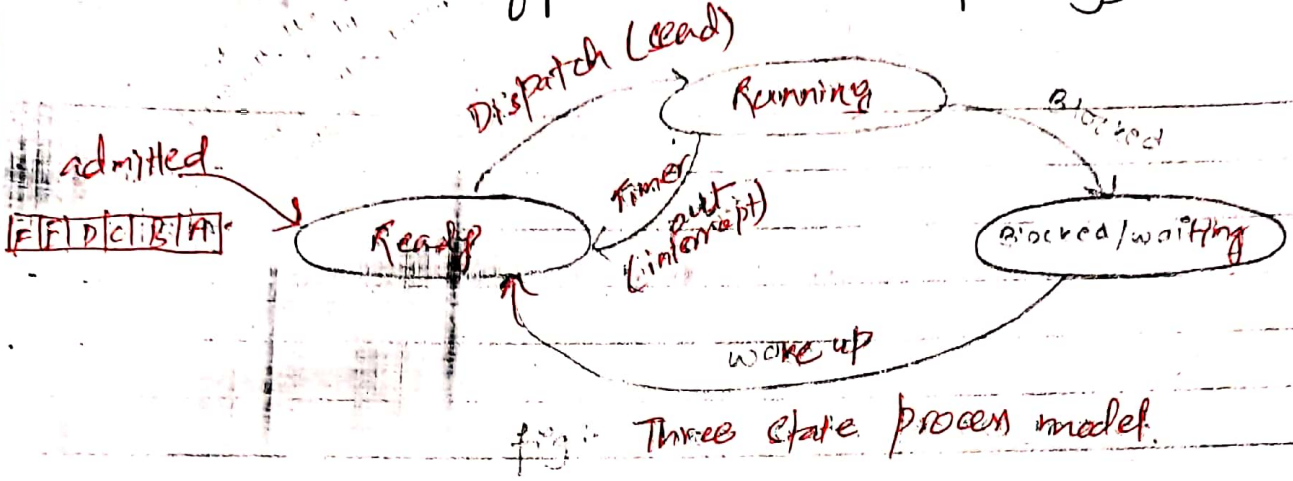


fig: Three state process model.

Three state Process Transition:

- When a job is admitted to the system, a corresponding process is created and normally inserted at the back of the ready list.
- When the CPU becomes available the process is said to make a state transition from Ready to Running. (memory is allocated)

Dispatch (processname): ready → running.

- To prevent any one process from monopolizing the CPU, os specify a time period (quantum) for the process.
- When the quantum expire makes state transition running to ready.

timerrunout (processname): running → ready.

- When the process requires an I/O operation before quantum expire the process voluntarily relinquishes the CPU and changed to the block state.

Blocked (process name): running → Blocked.

- When an I/O operation completes, the process make the

transition from block state to ready state.
wake up (process name) : blocked → ready.

Imp
Five State Process Model :

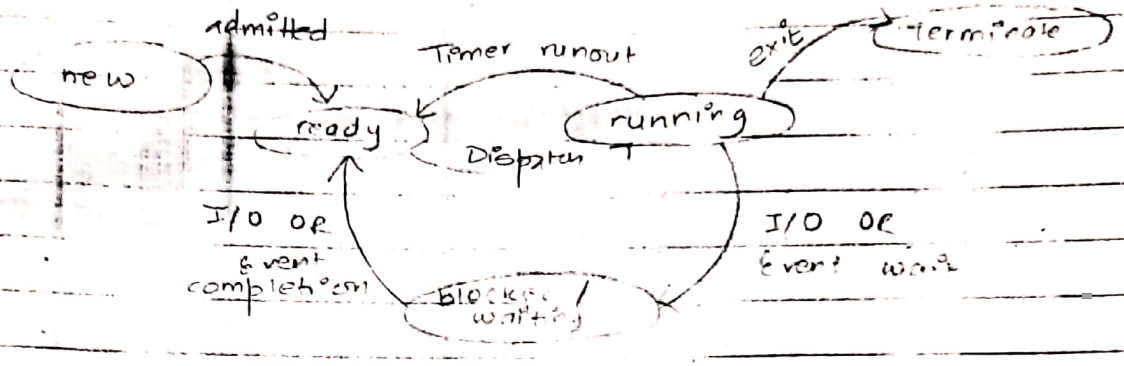


fig ⇒ five state process model

(a) new : The process is being created.
(b) Running : Instructions are being executed.
(c) Waiting / Blocked : The process is waiting for some event to occur.
(d) Ready : The process is waiting to be assigned to a processor
(e) Terminated : The process has finished execution.

Imp
Seven state Process Model :

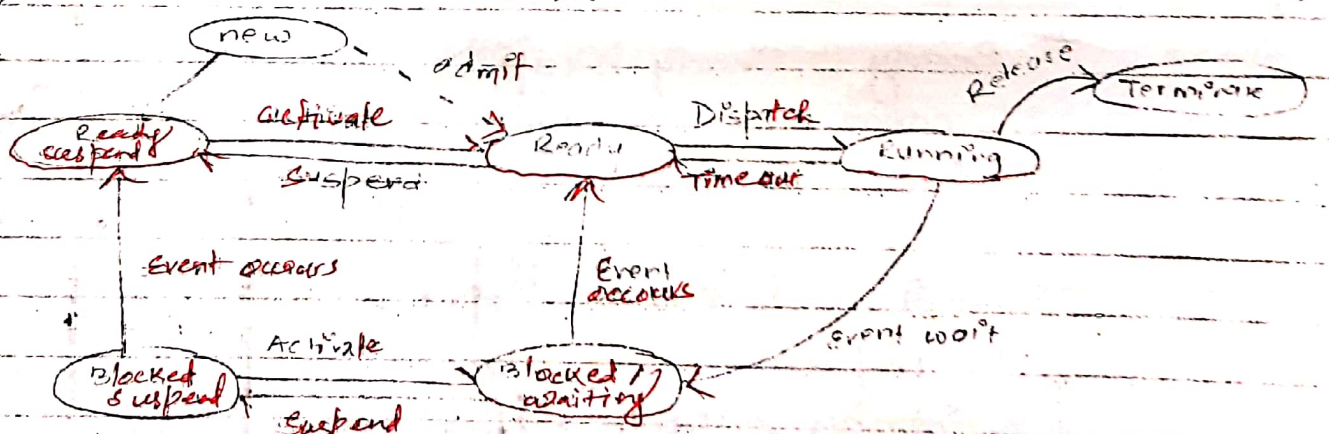


fig : 7 state process model.

(a) Blocked suspend
- blocked processes which have been swapped out to disk.

(b) Ready suspend
- Ready processes which have been swapped out of disk.

Block → Block suspend
When all processes are blocked, the OS will make room to bring a ready process in memory.

Block suspend → Ready suspend.
When the event for which it, as been waiting occurs

Ready → Ready suspend.
When there are no blocked processes and must free memory for adequate performance.

Implementation of Processes:
- To implement the process model, the operating system maintains a table called the Process table with one entry per process.
- These entries are called Process Control block (PCB).
- This entry contains information about the process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information.
- When the process is switched from running to

Thread is "light weight because it shares memory space of process. Thread can switch from one thread to another thread without reaching kernel.

ready or blocked state so that it can be restarted later as if it had never been stopped.

Thread.

- Thread like process are a mechanism to allow a program to do more than one thing at a time.
- Conceptually, a thread (also called light weight process) exists within a process.
- Threads are a finer grained unit of execution than processes.
- The term multithreading is used to describe the situation of allowing the multiple threads in same process.
- When multithreaded process is run on a single CPU system, the threads take turns running as in the multiple processes.
- All threads share the same address space, global variables, set of open files, child processes, alarms and signals etc.

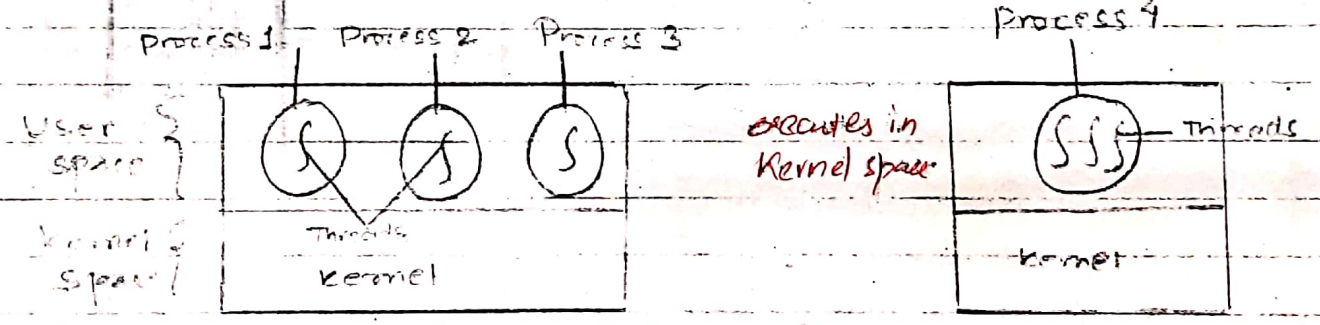
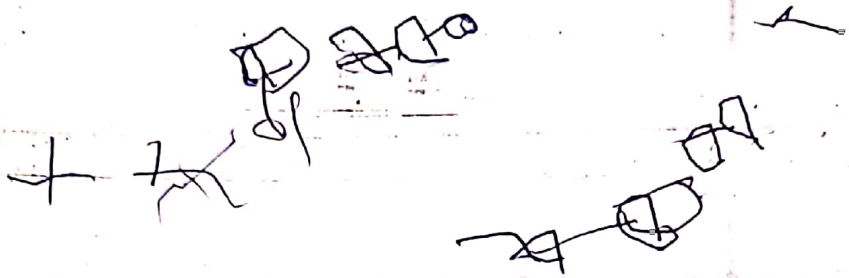


Fig (a) 3 process each with one thread (b) Single process with 2 threads

- Fig (a) organisation is used when three processes are unrelated.
- Fig (b) would be appropriate when the three threads are actually part of the same job and are actively co-operating with each other.



- Each thread maintain its own stack.

Advantages

(a) Responsiveness (b) Resource sharing (c) Economic.

Disadvantages.

(a) Designing complexity high.

Difference between Process and Thread.

Process	Thread.
(a) Process is heavy weight or resource intensive.	(a) Thread is light weight taking lesser resources than a process.
(b) Process switching needs interaction with operating system.	(b) Thread switching does not need interaction with operating system.
(c) In multiple processing environments, each process executes the same code but has its own memory and file resources.	(c) All threads can share same set of open files, child processes.
(d) If one process is blocked then no other process can execute until the first process is unblocked.	(d) While one thread is blocked or waiting, second thread in the same task can run.
(e) Multiple processes without using threads use more resources.	(e) Multiple threaded processes use fewer resources.
(f) In multiple processes, each process operates independently of the order.	(f) One thread can read, write or change another threads' data.

fast to create. (logical)
- written for concurrent programming

Kernel level thread
- slower execution as thread table is created at kernel space.
- no blockage (100% efficiency)
- never stops
- persistent
- related to system so os manage everything

User and kernel Threads :

(a) User thread :

- Thread management done by user level threads library.
- It is implemented as a library.
- Library provides support for thread creation, scheduling and management with no support for the kernel.
- It is fast to create.
- If kernel is single threaded, blocking system calls will cause the entire process to block. eg : Posix Thread.
- It shares address space, simple communication useful for application structuring.
- It has low overhead; everything is done at user level.
- If a thread blocks the whole process is blocked.
- All share the same processor so only runs at a time.
- The same thread library may be available on several systems.
- application specific thread management is possible.

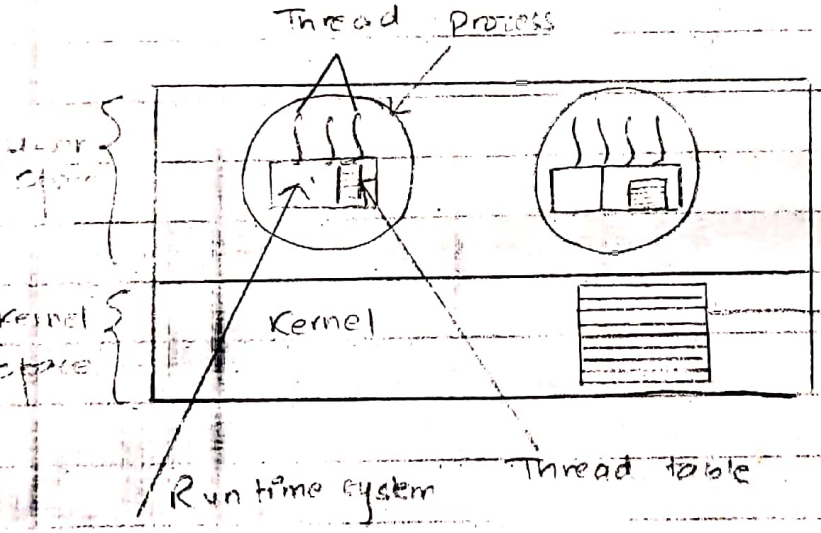


fig : A user level thread package

- Allow each process to have its own customized scheduling algorithm.

(b) Kernel level Thread

- It is supported by the kernel.
- kernel performs thread creation, scheduling and management in kernel space.
- slow to create and manage.
- There is no problem of blocking system calls.
- share address space, simple communication, useful for application structuring.
- medium overhead, operations require a kernel trap but little work.
- Independent, if one blocks, this does not affect others.
- can run in parallel in different processors in a multiprocessor

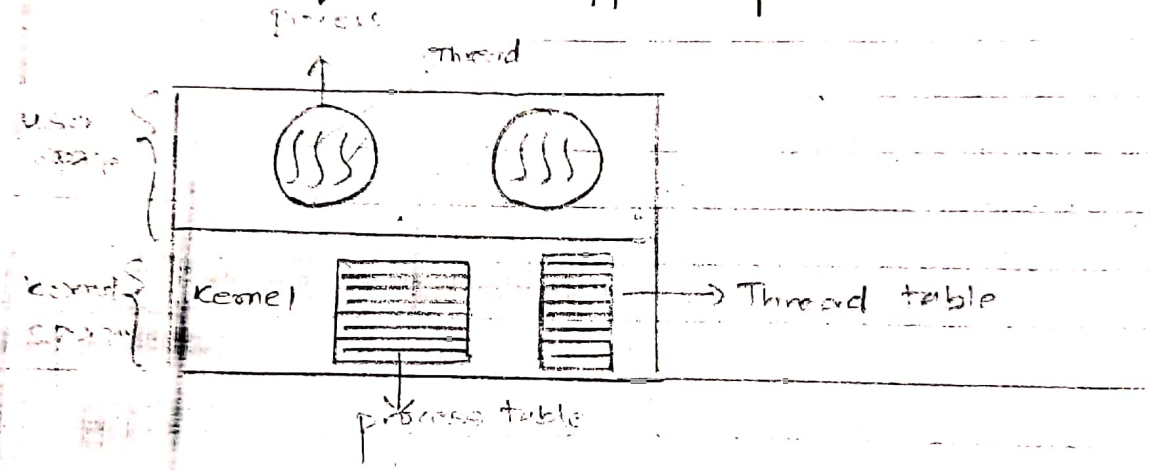


fig: A kernel level threads package

- No runtime system needed.
- The kernel's thread table hold each thread's registers, state and other information.
- The information is the same as user level threads.
- but it is now in the kernel space instead of user space.
- The kernel maintains single threaded processes and

traditional process table to keep track of processes.
- kernel threads do not require any new, non-blocking system calls.
- The main disadvantage is that cost of a system call is substantial.
- So, if thread operations are common, much more overhead will be incurred.

CPU scheduler:

(a) Long Term scheduler: new → ready → running. (time taking) — long
- Job scheduling is done when a new process is created.
- It initiates processes and so controls the degree of multiprogramming.

(b) Medium Term: ready → suspend → running
- It involves suspending or resuming processes by swapping them out of or into memory.

(c) Short term scheduler: memory ready state → CPU running state. (in queue)
- Short term process or CPU scheduling occurs most frequently and decides which process to execute next.

CPU Scheduling
- Scheduling refers to a set of policies and mechanisms to control the order of work to be performed by a computer system.
- Of all the resources in a computer system that are scheduled before use, the CPU is far the most important.

- Multiprogramming is the efficient scheduling of the CPU.
- The basic idea is to keep the CPU busy as much as possible by executing a (user) process until it must wait for an event and then switch to another process.

Scheduling Criteria :

- Different CPU scheduling algorithms have different properties.
- The choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use, the properties of the various algorithms should be considered.
- Criteria for comparing CPU scheduling algorithm may include the following :

(a) CPU utilization : Percent of time that the CPU is busy executing a process.

(b) Throughput : Number of processes that are completed per unit time.

(c) Response Time : ~~Am~~ Amount of time it takes from when request was submitted until the first response occurs (but not the time it takes to output the entire response.

(d) Waiting time : The amount of time before a process starts after first entering the ready queue (or the sum of the amount of time a process has spent waiting in the ready queue.

(e) Turn Around Time : Amount of time to execute a particular process from the time of admission

through the time of completion.

Completion time : The amount of time required to complete a process

Single processor scheduling algorithms :
(a) FCFS (First come first serve)
(b) SJF (shortest Job First) (preemptive and Non preemptive)
(c) Round Robin algorithm.
(d) Priority based algorithm.

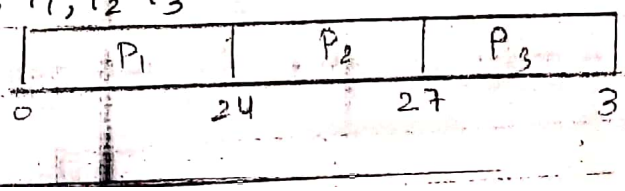
(a) First Come First Serve (FCFS) : (Non-preemptive)
- Processes are scheduled in the order they are received.
- Non-preemptive scheduling algorithm.
- With FCFS, the process that requests the CPU first allocated the CPU first.
- Large average waiting time.
- Not applicable for interactive systems.

(Q)

Process	Burst time
P_1	24
P_2	3
P_3	3

→ Suppose that the processes arrive in the order P_1, P_2, P_3. Draw the Gantt chart and find waiting time, average waiting time and Average Turn Around Time (ATAT).

Case I : P_1, P_2, P_3



① Waiting time for $P_1 = 0$

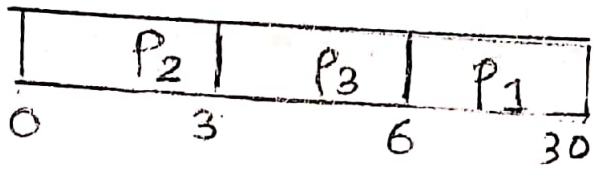
$P_2 = 24$

$P_3 = 27$

② AWT $= \frac{P_1 + P_2 + P_3}{3} = \frac{0 + 24 + 27}{3} = 17$

③ ATAT $= \frac{24 + 27 + 30}{3} = 27$

Case II : P_2, P_3, P_1



① waiting time for $P_2 = 0$

$P_3 = 3$

$P_1 = 6$

② AWT $= \frac{P_2 + P_3 + P_1}{3} = \frac{0 + 3 + 6}{3} = 3$

③ ATAT $= \frac{3 + 6 + 30}{3} = 13$

(b) Shortest Job First Scheduling : (SJF):

- The processing times are known in advance.
- SJF selects the process with shortest expected processing time.
- In case of the tie, FCFS scheduling is used.
- The decision policies are based on the CPU burst time.

Advantages :

- Reduces the average waiting time over FCFS.
- Favors short jobs at the cost of long jobs.

Problems :

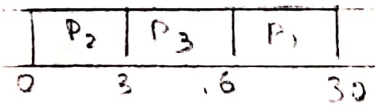
- Estimation of runtime to completion.

→ Not applicable in Time sharing.

Processes	Burst time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3 and are served in FCFS order.

Draw Gantt chart and find WT, TAT, AWT and ATAT?



Waiting time $(P_1) = 6$

$(P_3) = 3$

$(P_2) = 0$

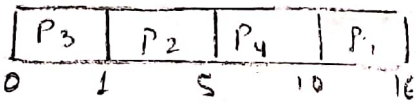
Average waiting time = 6+0+3/3 = 3.

Turn around time (P_1) = 30
(P_2) = 3
(P_3) = 6
Average TAT = 30+3+6 /3 = 13

Problem 2:

Processes	Arrival time	Burst time.
P_1	0	6
P_2	0	4
P_3	0	1
P_4	0	5

Draw, Gantt chart using SJF. Find WT, AWT, TAT, ATAT



~~Average~~ waiting time (P_1) = 10
(P_2) = 1
(P_3) = 0
(P_4) = 5

Average waiting time = 10+1+5/4 = 4

TAT = (P_1) = 16
(P_2) = 5
(P_3) = 1
(P_4) = 10

ATAT = 16+5+1+10/4 = 8

Problem 3

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

P_1	P_2	P_4	P_3
0	8 12	17	26

WT (P_1) = 0 AWT = ~~0+8+17+12/4~~ =

(P_2) = ~~8~~ 7 8-1 = 0+7+15+9/4 = 31/4 = 7.75

(P_3) = ~~17~~ 15 17-2

(P_4) = ~~12~~ 9 12-3

TAT (P_1) = 8 ATAT = (8+11+24+14)/4

(P_2) = 11 = 57/4 = 14.25

(P_3) = 24

(P_4) = 14

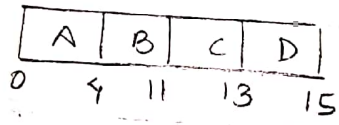
Problem 4 :

Draw a Gantt chart of (a) FCFS (b) SJF

Processes	AT	BT
A	0	4
B	2.01	7
C	3.01	2
D	3.02	2

find WT, AWT, TAT, ATAT

(a)



WT (A) = 0

(B) = 4 - 2.01 = 1.99

(C) = 11 - 3.01 = 7.99

(D) = 13 - 3.02 = 9.98

$AWT = \frac{0 + 1.99 + 7.99 + 9.98}{4}$

$= 4.99$

TAT (A) = 4

(B) = 8.99

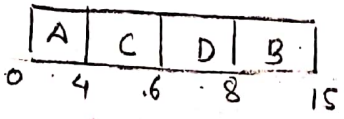
(C) = 9.99

(D) = 11.98

$ATAT = \frac{4 + 8.99 + 9.99 + 11.98}{4}$

$= 8.74.$

(b)



WT (A) = 0

(B) = 5.99

(C) = 0.99

(D) = 1.98

$AWT = 0 + 5.99 + 0.99 + 1.98 / 4$

$= 2.49$

TAT (A) = 4
(B) = 12.99
(C) = 2.99
(D) = 4.98

ATA = $\frac{4 + 12.99 + 3.99 + 4.98}{4}$

= 6.24

Shortest Remaining Time First (SRTF) :→ (Preemptive version of SJF) (works based on arrival time)

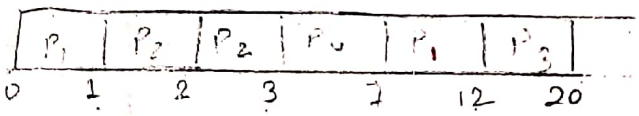
- Any time a new process enters the pool of processes to be scheduled.
- The scheduler compares the expected value for its remaining processing time with that of the process currently scheduled.
- If the new process's time is less, the currently scheduled process is preempted.

Merits:

- Low average waiting time than SJF.
- Useful in timesharing.

Processes	Arrival time	Burst time
P_1	0	6 ~~8~~ 0
P_2	1	2 ~~4~~ 0
P_3	2	~~8~~ 0
P_4	3	4 0

Draw Gantt chart and find WT, AWT, TAT, ATAT.



Completion time	TAT (CT − AT)	WT (TAT − BT)
12	12	6
3	2	0
20	18	10
7	4	0

ATAT = 36/4 = 9 AWT = 4

- P_1 starts at time 0 then P_2 arrives at time 1.
- As P_2 requires less time (2 milliseconds) to complete than P_1 (5 milliseconds), then P_1 is preempted and P_2 is scheduled.
- The next processes are then treated in the same manner.

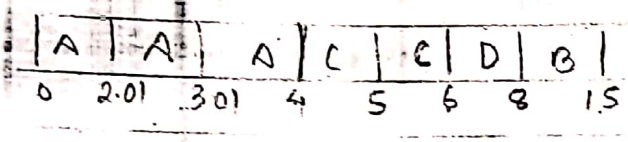
P.no.	AT	BT	CT	TAT	WT
P_1	0	6	19	19	12
P_2	1	5	13	12	7
P_3	2	3	6	4	1
P_4	3	1	4	1	0
P_5	4	2	9	5	3
P_6	5	1	7	2	1

ATAT = 43/6 = 7.16 AWT = 24/6 = 4

P_1	P_2	P_3	P_4	P_5	P_3	P_6	P_5	P_2	P_1
0	1	2	3	4	5	6	7	9	13	19

Processes	AT	BT
A	0	4 1.99 0.99 0
B	2.01	7
C	3.01	2 1 0
D	3.02	2 1 0

Draw Gantt chart.



(c) Round Robin Algorithm

→ Preemptive version of FCFS.

→ Poor average waiting time when process lengths are identical.

→ Imagine 10 processes each requiring 10msec burst time and 1msec quantum is assigned, performance depends on quantum size.

→ If the quantum is very large, each process is given as much time as it needs for completion.

→ Round Robin degenerates to FCFS policy.

→ If a quantum is very small, system busy at just switching from one process to another process. The overhead of context switching causes the system efficiently degrading.

Processes	AT	BT	CT	TAT	WT
P_1	0	8 4 2 0	19	19	13
P_2	0	8 8 x 0	20	20	15
P_3	0	20	6	6	4
P_4	0	3 x 0	15	15	12
P_5	0	7 5 5 0	23	23	16

ATAT = 16.6 AWT = 12

Draw Gantt and find WT, AWT, TAT, ATAT.
Quantum = 2

P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_4	P_5	P_1	P_2	P_5
0	2	4	6	8	10	12	14	15	17	19	20	23

Processes	AT	Burst time
P_1	0	70 58 46 34 22 10 0
P_2	0	82 70 58 46 34 22 10 0
P_3	0	14 2 0
P_4	0	48 36 24 12 0
P_5	0	66 54 42 30 18 6 0

Quantum = 12

P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_4	P_5
0	12	24	36	48	60	72	84	86	98	110	122	134	146	158

P_1	P_2	P_4	P_5	P_1	P_2	P_5	P_1	P_2	P_5	P_2
158	170	182	194	206	218	230	242	252	264	270	280

CT	TAT	WT
252	252	182
280	280	198
86	86	72
194	194	146
270	270	204

ATAT = 216.4 AWT = 160.4

(d) Priority scheduling

- The SJF is a special case of the general priority scheduling algorithm.
- A priority (an integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
- Equal priority processes are scheduled in FCFS order.
- Priority scheduling can be either preemptive or non-preemptive.
- A major problem with priority scheduling algorithm is indefinite blocking or starvation.
- Low priority processes could wait indefinitely for the CPU.
- A solution to the problem of starvation is aging.
- Aging is a technique of gradually increasing the priority of processes that wait in the system a long time.

Multilevel Queue scheduling.

- A multi-level queue scheduling algorithm partition ready queue into several separate queues.
- created for situations in which processes are easily classified into groups.

For eg: Foreground (interactive processes)
Background (Batch processes)

- These two types of processes have different response requirements and thus different scheduling needs.
- The processes are permanently assigned to one que based on some property of the process (e.g., memc size, priority or type).
- Each queue has its own scheduling algorithm.
- For eg: an RR algorithm might schedule the foregr queue, while the background a FCFS algorithm schedules queue.
- There must be scheduling between the queues, comi -only implemented as fixed priority preemptive sched -ling i.e foreground processes have absolute priority over the background processes.

Multilevel feedback Queue scheduling

- So far, we looked at CPU scheduling algorithms for single processor systems.
- If multiple CPUs exist, the scheduling problem is more complex.
- As with single processor systems, there is no one best solution.
- with identical processors, load sharing can occur.

- could provide a separate queue for each processor.
- This could lead to a situation where one processor could be idle with an empty queue, while another processor very busy.
- To prevent this situation, we could use a common ready queue.
- All processes enter one queue and are scheduled onto any available processors.

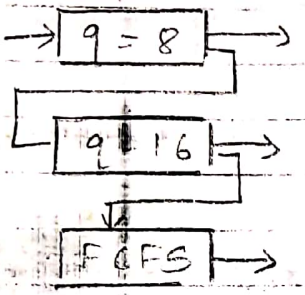


fig: Multilevel feedback queue

Interprocess Communication: ⇒ (IPC):
- Processes frequently need to communicate with other processes.
- For e.g: In a shell pipeline, the output of the first process must be passed to the second process and so on.
- Thus there is a need for communication between processes, preferably in a well structured way not using interrupts.
- In the following sections, we will look at some of the issues related to the Interprocess communication or IPC.

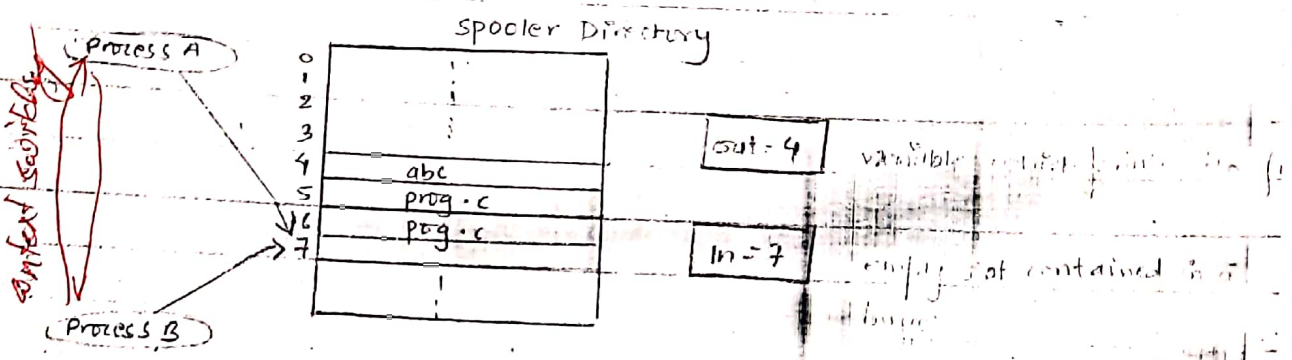
There are three major issues in IPC:
(a) How one process can pass information to another?
(b) The second has to do with making sure two or more processes do not get into each other's way when engaging

in critical activities.

(c) The third concerns proper sequencing when dependencies are present?

Process Synchronization (Race → Monitor) (long)

Race condition:



0 1 2 3 are the places reserved by spooler Directory. 4, 5 6 are empty but some process have been performed so reserved. The 7th place is empty. So, process A updates its buffer ie IN=7 (7th position of spooler directory is empty). But it doesn't keep the file in the spooler directory. Then process scheduler does context switching. After that, Process B also checks the spooler directory & finds the 7th place is empty then updates its buffer as IN=7 & keeps a file say, x.txt in the spooler directory. Then again the process scheduler does context switching. So, process A, without checking the spooler directly goes to its buffer & resumes its process. It overwrites the file x.txt created by process B. As soon as this process scheduler again does context switching. Process B keeps on searching the file x.txt which was previously created. This situation is known as 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.
- 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.
- 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 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.
- These two variables might At a certain instant, slots 0 to 3 are reserved and 4 to 6 slots are full.
- More or less simultaneously, process A and B decide they want to queue a file for printing.
- Process A reads in and stores the value, 7 in a local variable called next free - slot.
- Just then a clock interrupt occurs and the CPU decides that process A has long run enough, so it switches to process B. Process B reads in and also gets a 7.

Mutual exclusion: when one process A is executing

then another process B is blocked until the execution of process A and after the end of the execution, it unlocks the process B. Then Process A comes out of CS and Process B enters inside CS

- It too stores it in its local variable next-free-slot.
- Process B now continues to run. It stores the name of file in slot 7.
- Then it goes off and does other thing.
- Eventually a process A runs again, starting from the place it left off.
- It looks at next-free-slot, finds a 7 there and writes its file name in slot 7, erasing the name that process B just put there.
- Then it computes next-free-slot + 1 which is 8 and sets in to 8.
- The spooler directory is now internally consistent, so the printer daemon will not notice wrong.
- But process B will never receive any output.
- User B will hang around the printer room for years wishfully hoping for output that never comes.
- Situations like this, where two or more processes are reading or writing some shared data and the final results depend on who runs precisely are called Race Conditions.

VIVA

Critical Section : (CS/CR)

- How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files and shared everything else.
- To prohibit more than one process from reading and writing the shared data at the same time.
- Mutual Exclusion is some way of making sure that if

one process is using a shared variable or file, the other processes will be excluded from doing the same thing.
- The problem of avoiding race conditions can also be formulated in an abstract way.
- Part of the time, a process is busy doing internal computations and other things do not lead to race condition.
- However, sometimes a process have to access shared memory or files or doing other critical things that can lead to rules.
- That part of the program where the shared memory is accessed is called the critical region or critical section.
- If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

We need four conditions to hold to have a good solutions.
(1) No two processes may be simultaneously inside their critical sections.
(2) No assumptions may be made about speeds or the number of CPUs.
(3) No processes running outside its critical region may block other processes.
(4) No process should have to wait forever to enter its critical region.

What is mutual exclusion and race condition.

Mutual Exclusion : A entered critical region

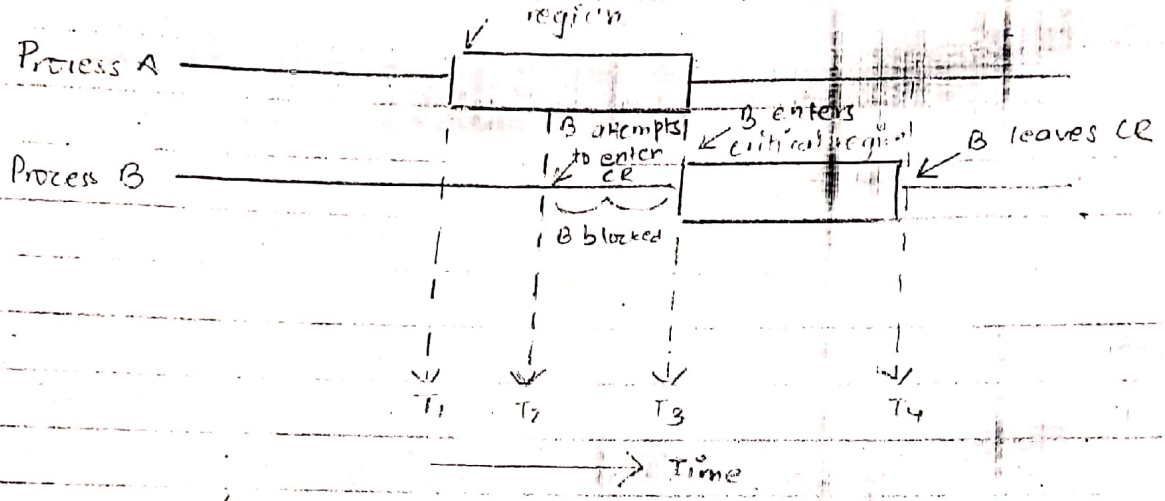


fig: Mutual Exclusion using critical region

→ Here process A enters the critical region at time T_1.
- A little later at time T_2, process B attempts to enter its critical region but fails because another process is already in its critical regions.
- And we allow only one at a time.
- Consequently B is temporarily suspended until time T_3 when A leaves its critical region allowing B to enter immediately.
- Eventually B leaves (at T_4) and we are back to the original situations with no processes in their critical region.

Mutual Exclusion with busy waiting :

① Hardware solution

② Disabling Interrupts

- The simplest solution is to have each process disable all interrupt just after entering its critical region and

re-enable them just before leaving it.
- With interrupts disabled, no clock interrupts can occur.
- The CPU is only switched from process to process as result of clock or other interrupts.
- After all and with interrupts turned off the CPU will not be switched to another process.
- Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.
- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.
- Suppose that one of them did it, and never turned them on again?
- That could be the end of the system.
- Furthermore, if the system is a multiprocessor, with two or more CPU's disabling interrupts affects only the CPU that executed the disable instruction.
- The other ones will continue running and can access the shared memory.
- On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists.

(2) Software solution

(a) lock variable :
- Consider having a single, shared (lock) variable initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process lets it to 1 and enters the

critical section.

- If the lock is already 1, the process just waits until it becomes 0.
- Thus a 0 means that no process is in its critical region and a 1 means that some process is in its critical region.
- Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory.
- Suppose that one process reads the lock and sees that it is 0.
- Before it can set the lock to 1, another process is scheduled, runs and sets the lock to 1.
- When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.
- Now we might think that we could get around this problem by first reading out the lock value then checking it again just before storing into it but that really does not help.
- The race now occurs if the second process modifies the lock just after the first process had finished its second check.

(b) Strict Alternation:

```
Process 0 while (TRUE){          turn=0        Process 1: while (TRUE){
          while (turn!=0);  → True                          while (turn!=1);
          critical_region();  then terminates;              critical_region();
          turn=1;             otherwise goes to CR          turn=0;
```

turn = 0 process can access CR & when it comes out it marks turn = 1 which means it has completed its work (can't access CR)
otherwise process 1 sees turn = 0, it sits in a tight loop (can't access CR)

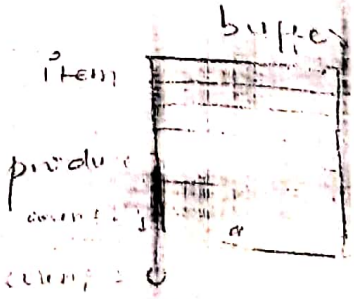
```
Process 0                          Process 1
while (TRUE) {                     while (TRUE) {
    while (turn != 0) /* wait */;      while (turn != 1) /* wait */;
    critical_region();                 critical_region();
    turn = 1;                          turn = 0;
    non-critical-region();             non-critical-region();
}                                  }
```

- The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process 0 inspects turn, find its to be 0 and enter its critical region.
- Process 1 also finds it to 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- Continuosly testing a variable until some value appears is called BUSY WAITING.
- It should usually be avoided, since it wastes CPU time.
- Only when there is a reasonable expectation that the wait will be short, is busy waiting used.
- A lock that uses busy waiting is called a spin lock.
- When process 0 leaves the critical region, it sets turn to 1. to allow process 1 to enter its critical region.
- Suppose that Process 1 finishes its critical region quickly so, both processes are in their non critical region and setting turn to 1.
- At this point turn is 1, and both processes are executing in their non critical regions.
- Suddenly, Process 0 finishes its non-critical region and goes back to the top of its loop.
- Unfortunately, it is not permitted to enter its critical region now, because turn is 1, and process 1 is busy with its non-critical region.

- It hangs in its while loop until process 1 sets turn to 0
- Put differently, taking turns is not a good idea when one of the processes is much slower than the other.
- This situation violates condition 3 in CS problem i.e. Process 0 is being blocked by a process 1 in its non-critical section.

Producer Consumer Problem (Bounded Buffer Problem):

```
#define N 100          /* No. of slots in the buffer */
int count = 0;         /* No. of items in the buffer */
void producer(void)
{
int item;
while (TRUE){
item = produce_item;   // generate next item.
if (count == N) sleep(); // If buffer full go to sleep.
insert_item(item);     // Put item in buffer.
count = count + 1;     // Increment count
if (count==1) wakeup(consumer); // was buffer empty?
}
}
void consumer(void)
{
int item;
while(TRUE){
if (count == 0) sleep(); /* If buffer empty go to sleep.
item = remove_item(); // Take item out of buffer.
count = count - 1;    // decrement count of items in buffer
```



```
If (count==N-1) wake up (Producer). // was buffer full?
    consume_item (item); // print them
}
}
```

Producer Consumer Problem : ⇒ (Bounded Buffer Problem)

- It is also known as Bounded buffer problem.
- Two processes share a common fixed size buffer.
- One of them, the producer puts information into the buffer and the other one, the consumer takes it once.
- It is also possible to generalize the problem to have m producers and n consumers but we will only consider the case of one producer and one consumer because the assumption simplifies the solutions.
- Trouble arises when the producer wants to put a new item in the buffer, but it was ready full.
- The solution is for the producer to go to sleep to be awakened when the consumer has removed one or more items.
- Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

- The approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier wit the spooler directory.
- To keep track of the number of items in the buffer, we will need a variable count.
- If the maximum number of items in the buffer can no. is N, the producers code will first test to see if count is
- If it is, the producer will go to sleep otherwise the producer will add an item and increment count.
- The consumers code is similar, first test count to see if it is 0. If it is, go to sleep.
- If it is non-zero, remove an item and decrement the counter.
- Each of the processes also tests to see if the other shoul be awakened and if so, wakes it up.
- To express system calls, such as, sleep and wakeup in C, we will show them as calls to library ~~routine~~ count
- Now, let us get back to the race condition.
- It can occur because access to count is unconstrained.
- The following situation could possibly occur.
- The buffer is empty and the consumer has just read count to see if it is 0.
- At that instant, the scheduler decides to stop running th consumer temporarily and start running the producer.
- The producer inserts an item in the buffer, increments count and notices that it is now 1.
- Reasoning that count was just 0, and thus the consume must be sleeping, the producer calls wake up to wake the customer up.

- Unfortunately, the consumer is not yet logically asleep, so the wake up signal is lost.
- When the consumer next runs, it will test the value of count it previously read, and finds it to be 0. and goto sleep.
- Sooner or later the producer will fill up the buffer and also go to sleep, both will sleep forever.
- The essence of the problem here is that a wake up sent to a process that is not sleeping is lost.
- If it were not lost, everything would work.
- A quick fix is to modify the rules to add a wakeup waiting bit.
- When a wakeup is sent to a process that is still awake the bit is set.
- Later when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off but the process will stay awake.

Semaphore

```
#define N 100 // no. of slots in buffer
typedef int semaphore; // semaphores are special kind of int
semaphore mutex = 1; // control access to critical section.
semaphore empty = N; // counts empty buffer slots.
semaphore full = 0; // counts full buffer slots.
void producer (void){          mutex => mutual exclusion
  int item;
  while (TRUE) {
item = produce_item(); // Generate something to put in buffer.
down(&empty); // Decrement empty count.
```

```
        down(&mutex); // Enter critical section.
        insert_item(item); // put new item in buffer.
        up(&mutex); // leave critical section.
        up(&full); // Increment count of full slots.
    }
}
void consumer(void)
{
    int item;
    while (TRUE) {
    down(&full); // Decrement full count.
    down(&mutex); // Enter critical section.
    item = remove_item(); // Take item from buffer
    up(&mutex); // leave critical section.
    up(&empty); // Increment count of empty slots.
    consume_item(item); // do something with item
    }
}
```

Semaphore:

- A new variable type called a semaphore was introduced
- A semaphore could have the value 0, indicating that no wakeups were saved or some positive value if one or more wakeups were pending.
- There are two operation on a semaphore checks to see if the value is greater than 0.
- If so, it decrements the value and just continues.
- If the value is 0, the process is put to sleep without

completing the down for the moment.
- Checking the value, changing it, and possibly going to sleep, is all done as a single indivisible atomic action.
- It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the a operation has completed or blocked.
- This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.
- The up operation increments the value of the semaphore addressed.
- If one or more processes were sleeping on that semaphore unable to complete an earlier down operation, one of them is chosen by the system.

Solving producer - consumer problem using semaphore :
- Semaphores solve the lost wakeup problem.
- It is essential that they be implemented in an indivisible way.
- The normal way is to implement up and down as system calls, with the OS briefly disabling all interrupts while it is testing the semaphore, updating it and putting the process to sleep.
- This solution ~~we~~ uses three semaphores, are called full for counting the number of ~~slots~~ slots that are full.
- One called empty for counting the number of slots that are empty and one called mutex to make sure the producer and consumer do not access the buffer at the same time.
- Full is initially 0, empty is initially equal to the number of slots in the buffer and mutex is initially 1.

- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical section at the same time are called binary semaphores.
- If each process does a down just before entering its critical section and an up just after leaving it, mutual exclusion is guaranteed.
- In a system using semaphores, the natural way to hide interrupts is to have a semaphore, initially set to 0, associated with each I/O devices.
- Just after starting an I/O device, the managing process does a down, on the associated semaphore thus blocking immediately.
- In example, we have actually used semaphores in two different ways.
- This difference is important enough to make explicit.
- The mutex semaphore is used for mutual exclusion.
- It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables.
- The other use of semaphores is for synchronization.
- The full and empty semaphores are needed to guarantee that certain event sequences do or do not occur.

Problem :

Suppose that two downs in producers code were reversed in order, so mutex were decremented before empty instead of after it.

- If the buffer were completely full, the producer would block with mutex set to 0.
- Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too.
- Both processes would stay blocked forever and no more work would be ever done.
- This situation is called a deadlock.

Monitors: → to solve deadlock condition.

- A monitor is a collection of producers, variables and data structures that are all grouped together in a special kind of module or package.
- Process may call the procedures in a monitor whenever they want to but they cannot access the monitor's internal data structures from procedures declared outside the monitor.
- Monitors have an important property that makes them useful for achieving mutual exclusion only one process can be active in a monitor at any instant.
- Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls.
- Typically when a process calls a monitor procedure, the first few instructions of the procedure will check to

see, if any other process is active within the monitor.
- If so, calling the process will be suspended until the other process has left the monitor.
- If no other process is using the monitor, the calling process may enter.
- It is up to the compiler to implement the mutual exclusion on monitor entries but a common way is to use a mutex or binary semaphore.
- Because the compiler, not the programmer is arranging for the mutual exclusion, it is much less likely that something will go wrong.
- In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion.
- It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.
- Although monitors provide an easy way to achieve mutual exclusion, that is not enough.
- We also need a way to block when they cannot proceed.
- In the producer consumer problem, it is easy enough to put all the tests for buffer full and buffer empty in monitor procedures but how should the procedure block when it finds the buffer full?
- The solution lies in the ~~produc~~ introduction of condition variables along with the operations on them wait and signal.

- When a monitor procedure discovers that it cannot continue (e.g. the producer finds the buffer full), it does a wait on some condition variable say full.
- This action causes the calling process to block.
- It also ~~knows~~ allows another process that had previously prohibited from entering the monitor to enter ~~so~~ now.

```
Monitor Producer Consumer
 Condition full, empty;
   integer count;
 procedure insert (item : integer)
 begin
    if count = N then wait (full);
     insert_item (item);
    count := count + 1;
  If count = 1 then signal (empty);
  end;
procedure remove : integer
 begin
  If count = 0 then wait (empty);
  remove = remove_item;
  count := count - 1;
  If count = N-1 then signal (full);
    end;
  count := 0;
End Monitor;

Procedure producer
  begin
```

```
while true do
begin
   item = produce_item;
Producer consumer. insert(item);
end;
end;
procedure consumer
begin
  while true do
  begin
     item = Producer Consumer. remove();
     consume_item (item);
  end;
  end;
```

Imp
short notes

Classical IPC Problems:

(a) The dining philosopher problem.
(b) Sleeping Barber Problem.
(c) The Readers writers Problem.

(a) The dining philosopher problem.

- Five philosopher's are seated around a circular table.
- Each philosopher has a plate of spaghetti.
- The spaghetti is so slippery that a philosopher needs two forks to eat it.
- Between each pair of plates is one fork.
- The life of the philosopher consists of alternate periods of eating and thinking.
- This is something of an abstraction, even for philosophers

but the other activities are irelevant here
- When a philosopher gets hungry, she tries to acquire her left and right fork, one at a time in either order.
- If successful in acquiring two forks, she eats for a while then puts down the forks and continue to think.
- The key question is: can you write a program for each philosopher that does what is supposed to do.
- The procedure take fork waits until the specified fork is available and then seizes it.
- Unfortunately, the obvious solution is wrong.
- Suppose that all five philosophers take their left fork simultaneously.
- None will be able to take their right forks and there will be a deadlock.
- We could modify the program so that after taking the left fork, the program checks to see if the right fork is available.
- If it is not, the philosopher puts down the left one waits for sometime and then repeats the whole process.
- This proposal too fails, although for a different reason.
- With the little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available.
- Putting down their left forks, waiting, picking up their left forks again simultaneously and so on forever.
- A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation (Indefinite postponement).

Solution:

- If the philosophers could just wait a random time instead of the same time after failing to acquire the right hand fork, the chance that everything would continue in lock step forever an hour is very small.
- This observation is true and in nearly all applications trying again later is not a problem.

for e.g: In the popular Ethernet local area Network, if two computers send a packet at the same time, each one waits a random time and tries again, In practice, this solution works fine.

The dining philosophers problem:

```
#define N 5 // no. of philosophers.
#define LEFT (i+N-1) % N // no. of left neighbours.
#define RIGHT (i+1) % N // no. of right neighbours.
#define THINKING 0 // philosopher is thinking.
#define HUNGRY 0 // philosopher is trying to get forks.
#define EATING 2 // philosopher is eating.
typedef int semaphore; // semaphores are a special kind int.
int state[N]; // array to keep track of everyone's state
semaphore mutex = 1; // Mutual exclusion for critical regions
semaphore s[N]; // one semaphore per philosoph
void philosopher(int i) // i: philosopher no. from 0 to N-1
{
while(TRUE) { /* Repeat */
think(); // philosopher is thinking.
take_forks(); // acquire 2 forks or block.
eat(); // eating yummy spaghetti.
```

```
put_forks(); // put both forks back on table.
}
}
void take_forks (int i) // i: philosopher no. from 0 to N-1
{
down (&mutex); // enter CS  mutex = 1
state[i] = Hungry ; // record fact that philosopher i is
                      Hungry.
test(i); // Try to acquire 2 forks.
up(&mutex); // exit CS
down (&s[i]); // block if forks were not acquired.
}
void put_forks () // i: philosopher no. from 0 to N-1.
{
down (&mutex); // enter CS
state[i] = THINKING ; // philosopher has finished eating.
test (LEFT); // see if left neighbour now can eat.
test (RIGHT); // see if right neighbour now can eat.
up (&mutex); // exit CS
}
void test (i) // i: philosopher no. from 0 to N-1.
{
if ( state[i] == HUNGRY && state [LEFT] != EATING &&
     state [RIGHT] != EATING) {
     state [i] = EATING;
     } up (&s[i]);
     }
}
```

The Sleeping Barber problem :

- Another IPC problem takes place in a barber shop.
- The barber shop has one barber, one barber chair and n chairs for waiting customers.
- If there are no customers present, the barber sits down in the barber chair and falls asleep.
- When a customer arrives, he has to wake up the sleeping barber.
- If additional customers arrive while the barber is cutting a customer's hair they either sit down (if there are empty chairs) or leave the shop (if all chairs are full)
- The problem is to program the barber and the customers without getting into race conditions.
- This problem is similar to various queueing situations, such as, a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.
- Our solution uses three semaphores : customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting.)
- Barbers, (the number of barbers 0 or 1) who are idle waiting for customers and
- a mutex which is used for mutual exclusion.
- We also need a variable, waiting which also counts the waiting customers.
- It is essentially, a copy of customers.
- The reason for having waiting is that there is no way to read the current value of a semaphore.
- In this solution, a customer entering the shop has to

count the number of waiting customers.
- If it is less than the number of chairs, he stays, otherwise he leaves.
- When the barber shows up for work in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0.
- The barber then goes to sleep.
- He stays asleep until the first customer shows up.
- When a customer arrives, he executes customer, standing by acquiring mutex to enter a critical region.
- If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released mutex.
- The customer then checks to see if the number of waiting customers is less than the number of chairs.
- If not, he releases mutex and leaves without a haircut.
- If there is an available chair, the customer increments the integer variable, waiting.
- Then he does an up on the semaphore customers, thus waking up the barber.
- At this point, the customer and the barber are both awake.
- When the customer releases mutex, the barber grabs and begins the hair cut.
- When the barber is over, the customer exits the procedure and leaves the shop.

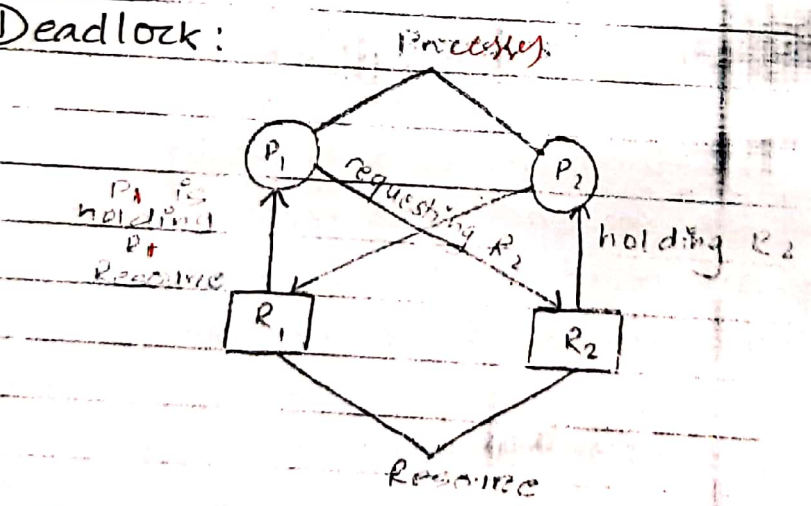
```
# # define CHAIR 5 // chairs for waiting customers.
typedef int semaphore;
semaphore customers = 0; // customers waiting.
semaphore barbers = 0;
semaphore mutex = 1; // for Mutual exclusion.
int waiting = 0; // customers are waiting.
void barber (void) {
  while (TRUE) {
  down (& customers); // Go to sleep if no. of customer
                         is 0.
  down (& mutex); // acquire access to waiting.
  waiting = waiting - 1; // Decrement count of waiting customer.
  up (& barbers); // one barber is now ready to cut hair.
  up (& mutex); // Release waiting.
  cut_hair (); // cut hair.
  }
}
void customer (void) {
  down (& mutex); // enter critical section.
  if (waiting < CHAIRS) { // if there are no free chairs, leave.
  waiting = waiting + 1;
  up (& customers); // wake up barber.
  up (& mutex); // Release Access to waiting.
  down (& barbers); // Go to sleep if no. of free barber is 0
  get_haircut (); // Be seated and get service.
  }
  else {
    up & (mutex); // seat is full, leave shop.
  }
```

(c) Readers writers Problem :

- It is acceptable to have multiple processes reading the database at the same time.
- But if one process is updating the database at the same time, no other processes may have access to the database not even readers.
- In this solution, the first reader to get access to the database does a down on the semaphore database.
- Subsequent readers merely increment a counter.
- As readers leave, they decrement the counter and the last one out does an up on the semaphore allowing a blocked writer, if there is one to get in.
- Suppose that while a reader is using the database, another reader comes along.
- Since having 2 readers at the same time is not a problem, the 2nd reader is admitted.
- A third and subsequent readers can also be admitted, if they come along.
- Now, suppose that a writer comes along, the writer cannot be admitted to the database, since writers must have exclusive access. So, the writer is suspended.
- Later, additional readers show up.
- As long as atleast one reader is still active subsequent readers are admitted.
- As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive.
- The writer will be kept suspended until no reader is

present.
- If a new reader arrives say every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.
- To prevent this situation, when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately.
- In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for reader that come along after it.

Deadlock:



Introduction
- A process in a multiprogramming system is said to be in deadlock if it is waiting for a particular event that will never occur.

e.g.: → All automobiles trying to access.
→ Traffic completely stopped.
→ Not possible without backing some.

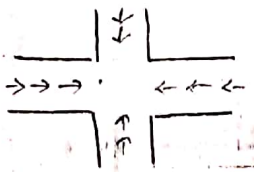


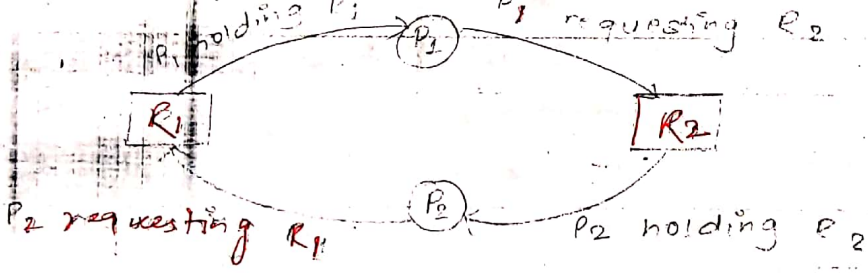
Fig: A traffic deadlock

Resource Deadlock:

A process request a resource before using it and release after using it.

(1) Request the Resource.
(2) Use the Resource
(3) Release the Resource.

- If the Resource is not available when it is requested, the requesting process is done to wait.
- Most deadlocks in OS developed because of the normal contention for dedicated resources.



- Process P_1 holds Resource R_1 and needs Resource R_2 to continue, process P_2 holds resource R_2 and needs resource R_1 to continue → occurs deadlock.

Conditions for deadlock.

(1) Mutual Exclusion: Process claims exclusive control of resources they require.
(2) Hold and wait: Processes hold resources already allocated to them while waiting for additional resources.

(3) No Preemption: Resoune previously granted cannot be forcibly taken away from the process.
(4) Cirular Wait: Each process holds one or more resources that are requested by next process in the chain.

- A, deadlock situation can arise if all four conditions had simultaneously in the system.

Handling Deadlock

Deadlock handling strategies:
(1) We can use a protocol to prevent or avoid dealocks ensuring that the system never enter a deadlock state.
(2) We can allow the system to enter a deadlock state detect it and recover.
(3) We can ignore the problem all together and prete that deadlock never occur in the system.

Deadlock Prevention.

Fact: If anyone of the four necessary conditions are denied, a deadlock cannot occur.

(a) Denying Mutual Exclusion.
- Shareable resources do not require mutually exclus access such as read only shared file.
- Some resources are strictly non-sharable mutuall exclusive control required.

(b) Denying Hold and wait.
- Resources grant on all or none basis
- If all resources needed for processing are available then granted and allocated to process.
- If complete set of resources are not available, the process must wait set available.
- While waiting, the process should not hold any resources.

(c) Denying No-preemption.
- When a process holding resources are denied a request for additional resources, that process must release its held resources and if necessary, request them again together with additional resources.

(d) Denying circular wait
- All resources are uniquely numbered and processes must request resources in linear ascending order.
- The only ascending order prevents the circular.

Deadlock Avoidance.
- Avoiding deadlock by careful resource allocation.
- Decide whether granting a resource is safe or not and only make the allocation when it is safe.
- Need extra information in advance, maximum number of resources of each type that a process may need.
- The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there never be a circular wait conditions.

Avoidance with single Resource.

	Has	Max
A	3	9
B	2	4
C	2	7

free : 3 (a)

	Has	Max
A	4	9
B	4	4
C	2	7

free : 1 (b)

	Has	Max
A	3	9
B	-	-
C	2	7

free : 5 (c)

	Has	Max
A	3	9
B	-	-
C	7	7

(d) free : 0

	Has	Max
A	3	9
B	-	-
C	-	-

(e) free : 7

fig: Safe state.

→ A state is said to be safe if it is not dead locked and there is same scheduling order in which every process can run completion.

Unsafe state :

	Has	Max
A	3	9
B	2	4
C	2	7

free: 3 (a)

	Has	Max
A	4	9
B	2	4
C	2	7

free : 2 (b)

	Has	Max
A	4	9
B	4	4
C	2	7

free : 0 (c)

	Has	Max
A	4	9
B	-	-
C	2	7

free : 4 (d)

Imp

Banker's algorithm with multiple Resource.

Need Matrix

Process	A	B	C
P_0	7	5	2
P_1	4	3	7
P_2	5	3	1
P_3	4	7	8
P_4	3	2	5

Allocated Matrix

Process	A	B	C
P_0	2	1	0
P_1	1	2	1
P_2	2	1	0
P_3	0	2	3
P_4	0	0	2

Available matrix

A	B	C
5	3	4

Is $P_4 \rightarrow P_2 \rightarrow P_1 \rightarrow P_0 \rightarrow P_3$ in safe state?

Soln:-

for P_4, required matrix. (need - allocated)

A	B	C
3	2	3

Remaining,			Distributed		
A	B	C	A	B	C
2	1	1	3	2	3

∴ P_4 can be executed successfully.

for P_2,

Available Matrix (Distributed + Remaining + Allocated)

A	B	C
5	3	6

Required matrix

A	B	C
3	2	1

Remaining			Distributed		
A	B	C	A	B	C
2	1	5	3	2	1

∴ P_2 can be executed safely.

for P_1,

Available matrix

A	B	C
7	4	6

Required matrix,

A	B	C
3	1	6

Remaining			Distributed		
A	B	C	A	B	C
4	3	0	3	1	6

for P_0

Available matrix

A	B	C
8	6	7

D = 4 D = 2

R = 3 R = 2 R = 5

Required matrix

A	B	C
5	4	2

P_0 can be executed safely.

for P_3

Available

A	B	C
10	7	7
D = 4	D = 5	D = 5
R = 6	R = 2	R = 2

Required

A	B	C
4	5	5

P_3 can be executed successfully.

∴ When we allocate all of these resources in this ~~resour~~ manner, then we will be successful. So, we ~~are~~ ~~exe~~ in safe state and thus, ~~we can~~ dead lock condition will never occur.

∴ free

A	B	C
10	9	10

[Remaining + Distributed + Alloca

Is $P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_0$ in safe state?

Soln:- for P_1,

Required matrix

A	B	C
3	1	6

	A	B	C
R:	2	2	-
D:	3	1	-

∴ There is high probability of ~~the~~ dead lock if we allocate the resources in ~~this~~ ~~resource~~ manner / order.

Hence, we can't execute it in safe state.

Need Matrix / Max

Process	A	B	C	D
P_0	0	0	1	2
P_1	1	7	5	0
P_2	2	3	5	6
P_3	0	6	5	2
P_4	0	6	5	6

Allocated

Process	A	B	C	D
P_0	0	0	1	2
P_1	1	0	0	0
P_2	1	3	5	4
P_3	0	6	3	2
P_4	0	0	1	4

Available

A	B	C	D
1	5	2	0

Is $P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3 \rightarrow P_4$ in safe state ?

for P_0

Required matrix

A	B	C	D
0	0	0	0

P_0 can be executed safely...

for P_2

Available

A	B	C	D
1	5	3	2
1+0=1	0	0	2
1+1=2	6	3	4

Required

A	B	C	D
1	0	0	2

P_2 can be executed safely and successfully.

for P_1

Available

A	B	C	D
2	8	8	6
2+0=2	7	5	0
2+2=4	1	3	6

Required

A	B	C	D
0	7	5	0

P_1 can be executed safely.

for P_3

Available				Required			
A	B	C	D	A	B	C	D
3	8	8	6	0	0	2	0

P_3 can be executed safely.

for P_4,

Available				Required			
A	B	C	D	A	B	C	D
3	14	11	8	0	6	4	2

P_4 can be executed safely.

∴ When we allocate all of these resources in this manner then we will be successful. So, we are in safe state and thus dead lock condition never occur.

Free :

A	B	C	D
3	14	12	12

Deadlock Recovery

- When a deadlock exists, then there are four possible methods of recovery from deadlock.

(1) Kill All :

- Kill all deadlocked processes. This is quite simple and easy to implement.

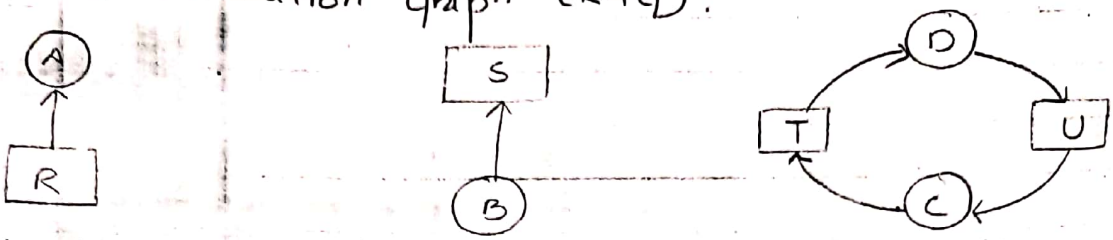
(2) For all the deadlocked processes back to their last known state and restart them, if check pointing is done.

(3) Kill processes one by one as at a time until deadl

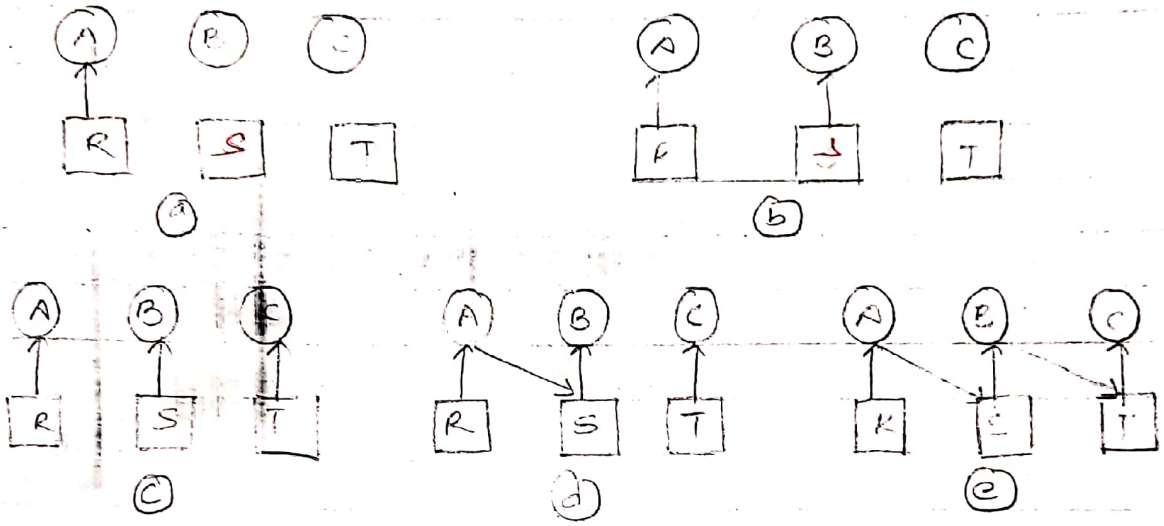
breaks.

(4) Preempt resources until the deadlock is broken.

Resource Allocation Graph (RAG): detects deadlock.



(a) Holding a resource (b) Requesting Resource (c) Deadlock.



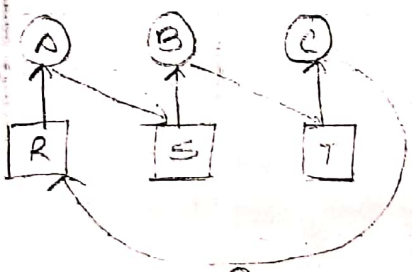


fig: Deadlock condition.

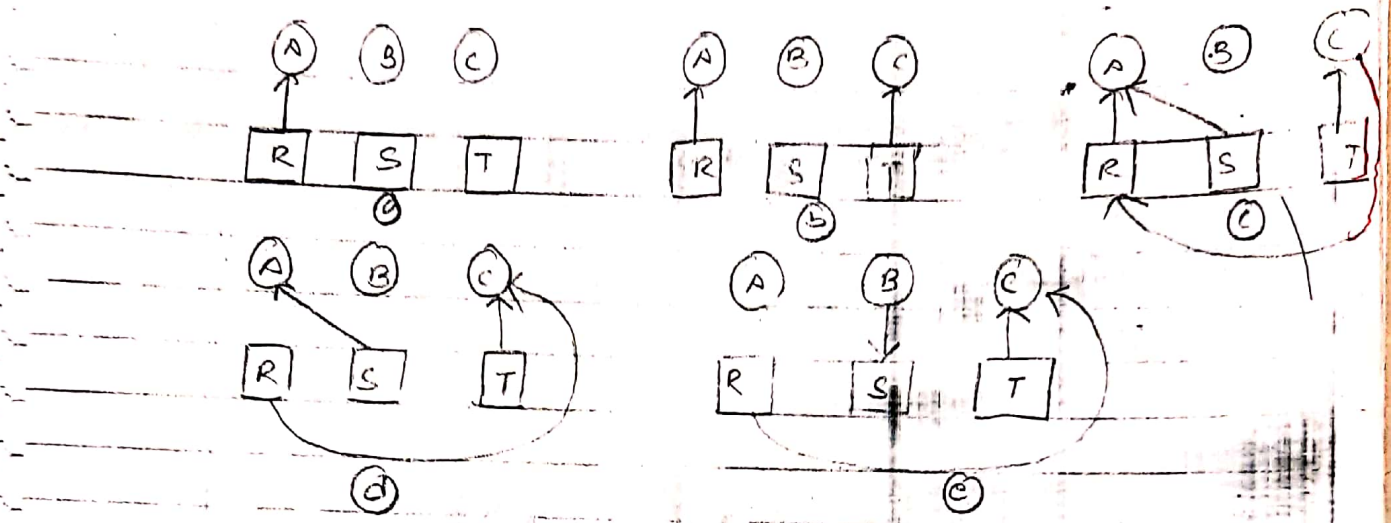
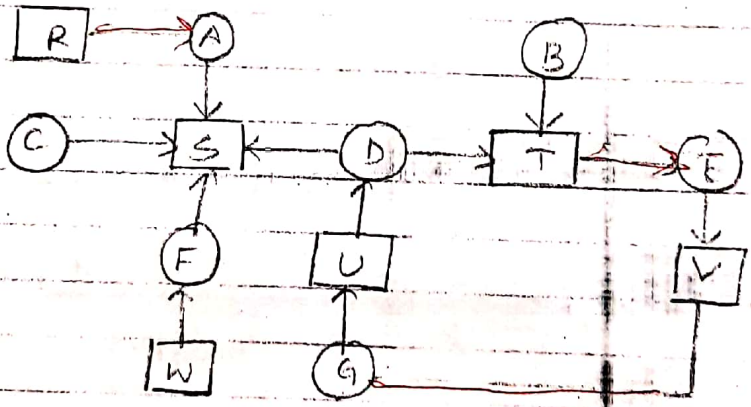
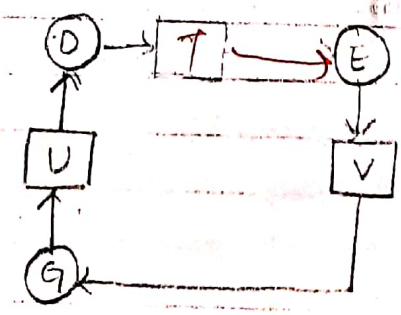


fig: No Deadlock

Deadlock Detection and Recovery:



(a) Resource Graph.



(b) Deadlock part

Chapter-3

Memory Management:

- The entire program and data of a process must be in main memory for the process to execute.
- How to keep track of processes currently being executed?
- Which processes to load when memory space is available?
- How to load the processes that are larger than main memory?
- OS component that is responsible for handling those issues is a memory manager.

Single process Monitor:

- It is also called as a monoprogramming or uniprogramming system.
- It is the simplest memory management approach.
- The memory is divided into two contiguous areas.
- First is the lower memory address area for an operating system program and the second is for the user program.
- These programs or processes are also known as transient processes which are loaded according to the users input.
- In this type of approach, OS keeps track of the first and the last location available for allocation of user programs.
- An OS is loaded either at the bottom or at the top, in order to provide a contiguous area of free storage for a user program.
- A transient program (user process) is only when the OS passes a control to it.
- After receiving a control, it starts running the program until its completion or termination due to I/O or some error.
- When this program is completed or terminated, the OS may

relative / fixed address ⇒ absolute address
changes ⇒ relocatable address

load another program for execution.

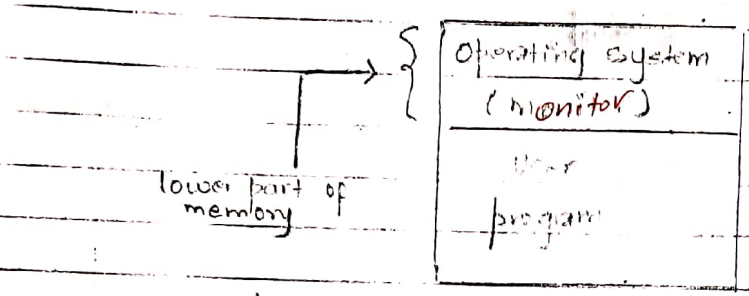


fig: main memory partition for a single process monitor

Address Binding: binding space to variable/data

- We know that our program is stored in RAM only.
- Each instruction has to go through different phases
- similarly, a program has also to go through 3 phases.

(a) Compilation phase.
(b) Loading phase
(c) Run phase.

(a) Compile Time Binding: It generates absolute address
- It requires that it must be known at the compile time its that is where will a process reside in the memory.
- It is quite approprate choice.
- small, simple systems or dedicated hardware.
- The problem in this approach is that if the starting add of a process in memory changes then the entire process must be recompiled to generate the absolute address again.

(b) Load Time Binding:
- The compiler generates relocatable addresses which are conve

to absolute address at the load time.
- Process cannot be moved around during execution.
- Here, in this method of binding, if the starting address of the process changes then it can be accomplished statically re-locating the code.

© Run Time Binding:
- we can move a process during runtime from one memory segment to another.

Logical vs physical address space :
Virtual vs Real address
- We use logical address and virtual address interchangeably.
- The set of all logical addresses generated by a program is a logical address space.
- The set of all physical address corresponding to these logical addresses is a physical address space.
- Thus, in the execution time, address binding scheme the logical and physical address spaces differ.
- The runtime mapping from virtual to physical addresses are done by a hardware device called the memory management Unit (MMU).
- This method requires hardware support slightly different from the hardware configured.
- The base register is now called a relocation register.
- The value in the relocation register is added to every address generated by a user process at one time it is sent to the memory.
- For eg: If the base is at 14000, then an attempt by

Roll in -> swap in hard disk to memory (processes)
Roll out -> swap out memory to hard disk

the user to address location 0 is dynamically relocated to location 14000, an access to location 346 is mapped to location 14346.

Memory Allocation Techniques

→ Memory Allocation is of two types:

(1) Contiguous memory allocation: (a) Fixed partition Allocation
(b) Variable/Dynamic partition Alloc

(2) Non-contiguous allocation: (a) Paging
(b) Segmentation.

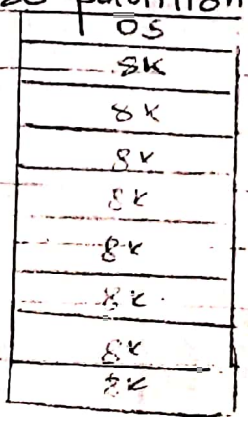
(1) Contiguous storage allocation:

- In contiguous memory allocation, a memory resident progra occupies a single contiguous block of physical memory.
- The memory is partitioned into blocks of different siz to accomodate the programs.

(a) Fixed partitioning:

- Internal fragmentation of memory.
- Inefficient memory utilisation
- Larger processes can be handled.
- There are two types of fixed partitioning.

(i) Equal fixed size partition:



- It is divided into equal fixed size partitions.
- Best suited for equal fixed size processes.
- Internal fragmentation occurs frequently.

(ii) Unequal fixed size partitions:

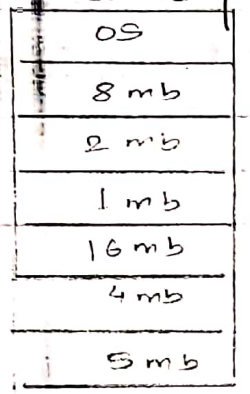


fig: Memory assignment for unequal fixed size partitioning.

- It is divided into unequal but fixed size partition.
- Have less internal fragmentation than equal size in case of unequal processes.
- But still have lot of smaller holes.

Advantages of fixed partitioning:
(a) Simple
(b) Minimal OS software
(c) Minimum processing overhead.

Disadvantages:
(a) Memory utilization inefficient.
(b) Small jobs do not utilize partitions efficiently.
(c) Job size cannot be greater than the size of partitions.
(d) Limits active processes by number of partitions.

(b) Dynamic / Variable partitioning :

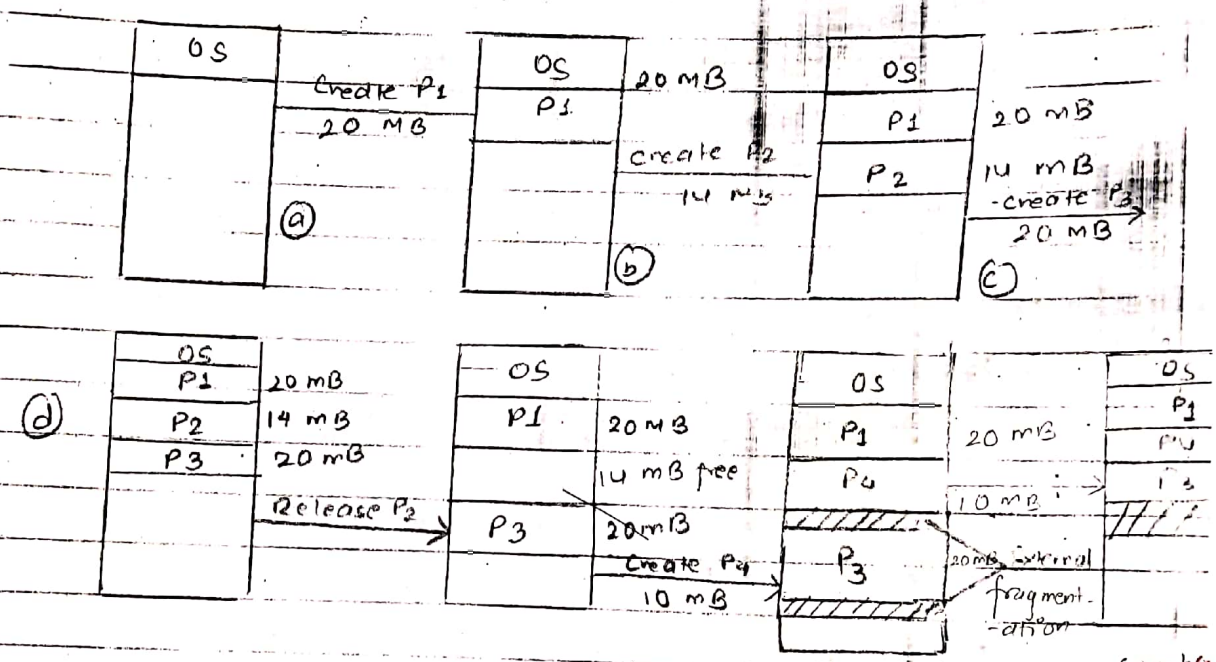


fig: External fragmentation in dynamic partition

→ partitions are of variable length and number determined by active processes.

→ Performance by the active processes.

→ It leads to a situation in which there are a lot of small holes in memory called external fragmentation.

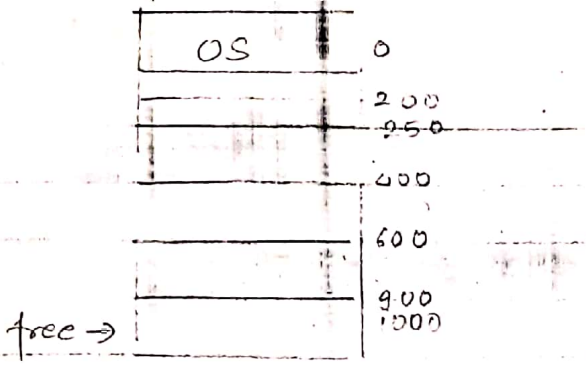
→ Compaction technique is used to overcome external fragmentation.

Placement Algorithms :

- The most common strategies to allocate free partition to the new processes are :-

1. First fit — allocates the process to the memory whenever it fits.
2. Best fit — scans all the memory and then allocate the process to exactly matched memory
3. Worst fit — allocates the process to big memory space.
4. Next fit — next fitted position from pointer.

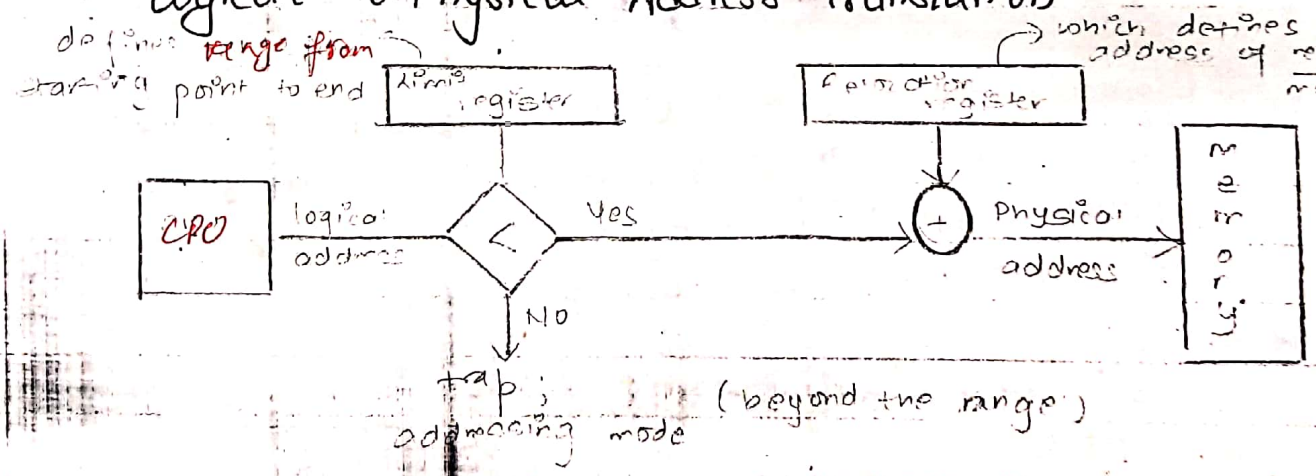
① First fit



Overlays

- In the past, when the programs were too big to fit in available main memory, the solution adopted was to split the program into pieces is called overlays.
- Overlays1 would start running first, when it was done, it would call next overlay2 and so on.
- The overlays are kept on disk and swapped in and swapped out by OS.

Logical to Physical Address Translation



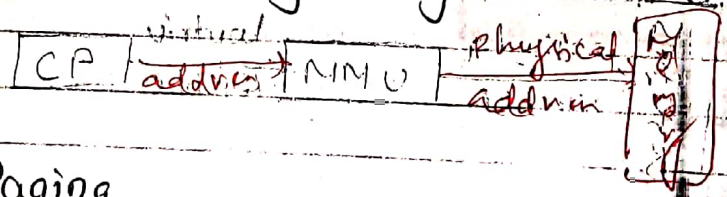
Imp.

Paging : memory mgmt technique where logical program is divided into no of equal parts (eg of 64 KB Pgm is divided into 16 equal parts of 4 KB) and that particular part is known as page.

Acc to size of page the main memory (real/physical) is also divided into equal no of blk, to map the each page of logical pgm to real/physical pgm frame.

Memory Management Unit (MMU)

- The runtime mapping from virtual address to physical address done by hardware device is called memory management unit (MMU).



Paging

The virtual address space is divided up into fixed size blocks called page and the corresponding same size block in main memory called frames.

- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The size of the pages is determined by the

hardware, normally from 512 bytes to 64KB (in power of 2).
- Paging permits the physical address space of process to be non-contiguous.
- Traditionally, support for paging has been handled by hardware, but the recent design have implemented by closely integrating the hardware and OS.
- Logical address space of a process can be non-contiguous and a process is allocated physical memory whenever the free memory frame is available.
- OS keep track of all free frames.
- OS needs n free frames to run a program of size n pages.
- Address generated by CPU is divided into
  - Page number (p): page number is used as an index into a page table which contains base address of each page in physical memory.

- Page offset (d): ⇒
- page offset is combined with base address to define the physical memory address.
- Example: consider the memory in figure, using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the users view of memory can be mapped into physical memory.
- Logical address 0 is page 0, offset 0, indexing into the page table, we find that page 0 is in frame 5.
- Thus logical address 0 maps to physical address 20 = ((5×4)+0)
- logical address 3 (page 0, offset 3) maps to physical address ~~23 ((5×4)+3)~~. 23 = ((5×4)+3).

∴ We can notice that paging itself is a form of dynamic relocation.

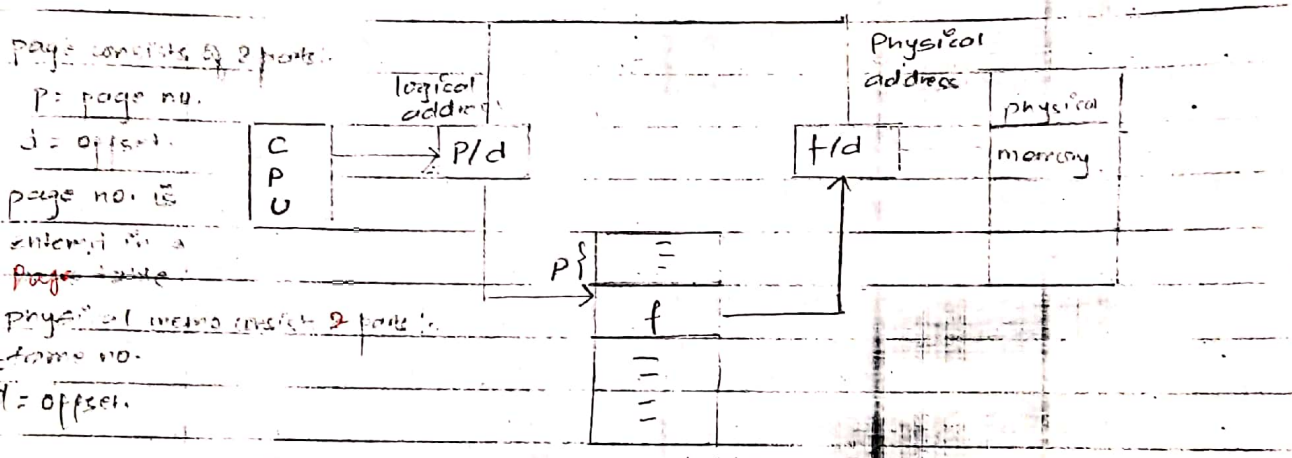
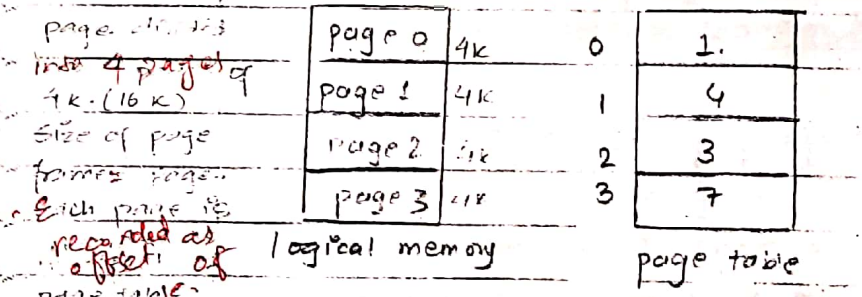


fig 1:- paging hardware.



page no. is index of page table.
Each page is recorded as offset of page table.
Logical page 0 is refer to page frame 1 of page table.
This is known as mapping (logical to physical)

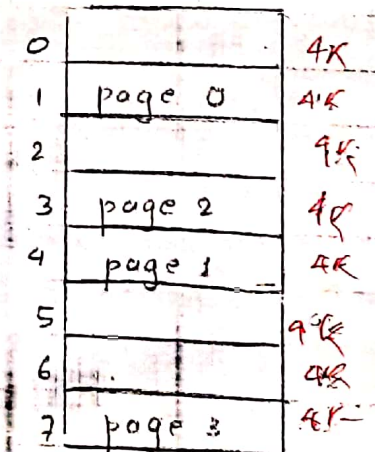


fig 2: paging model of logical and physical memory.

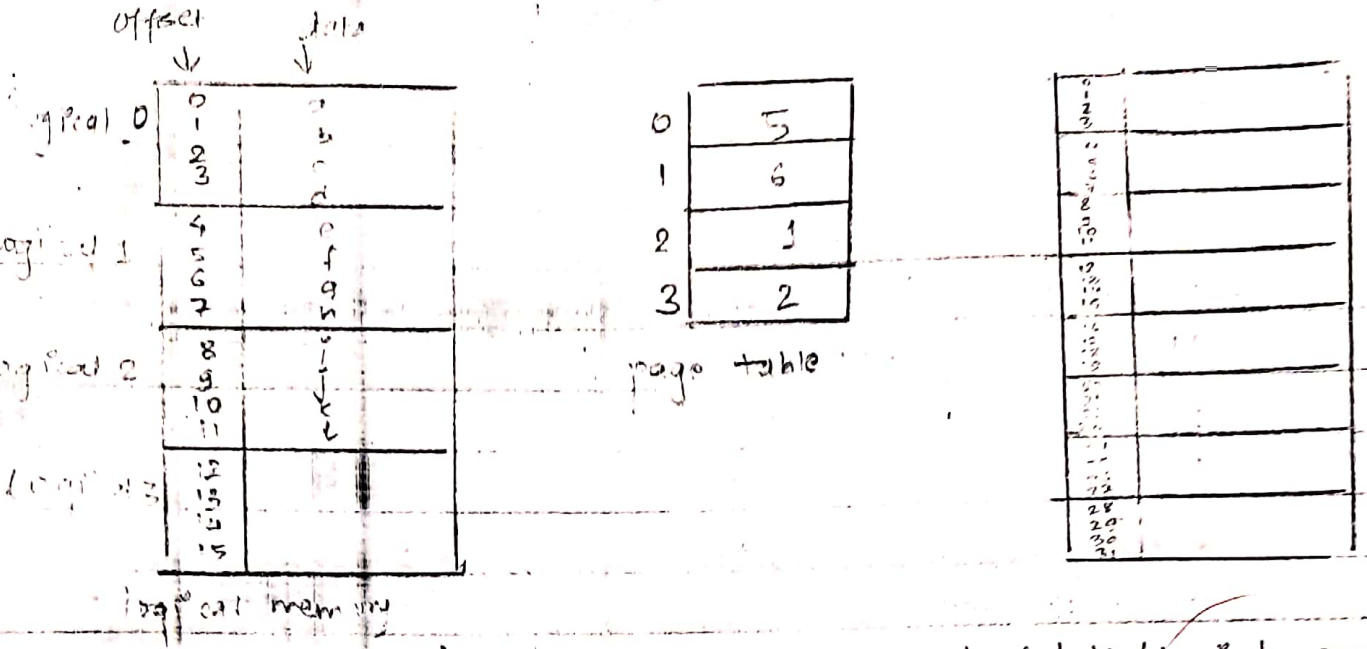


fig 3: mapping of page from logical to physical memory

- [logical 0, offset = 0]
  page size = 4K
  - Pg 0 → frame = ?

  (4 × 5 + 0) → offset
  = 20 → page size, frame

  [Log 1, offset 5]
  (4 × 6 + 5)
  = 29

Logical 0, offset 2
= (4 × 5 + 2)
= 22

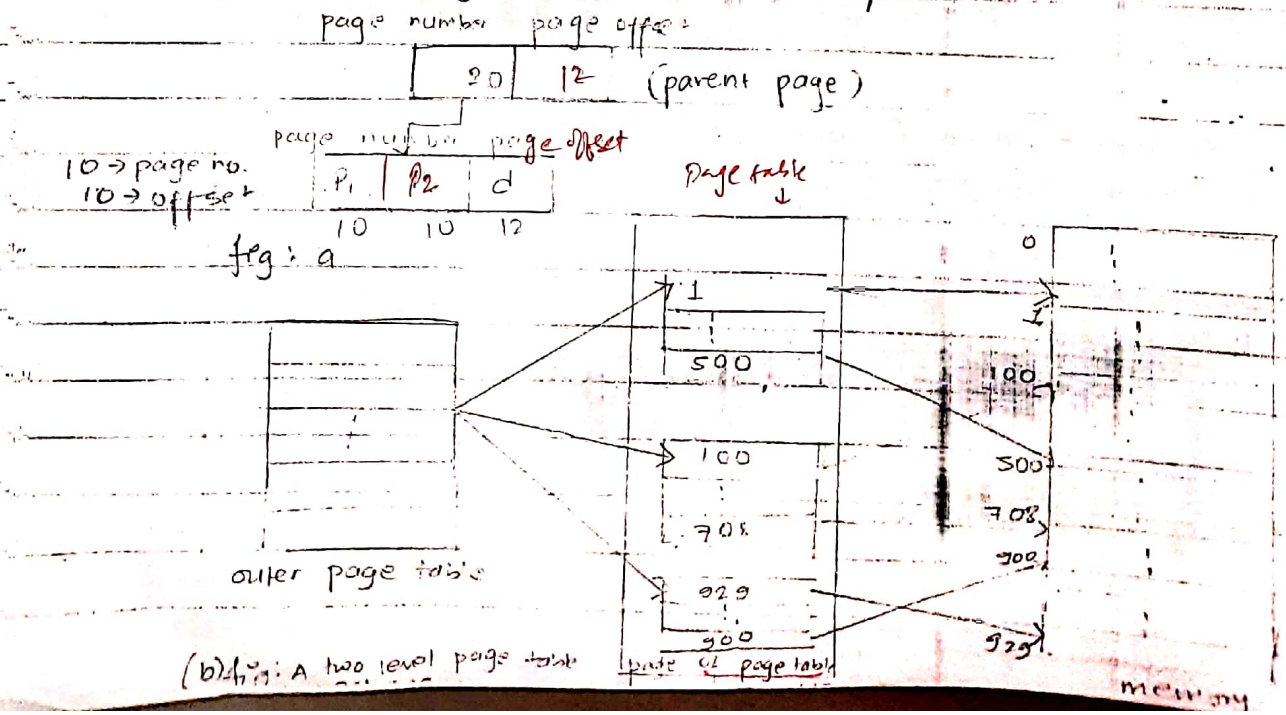
Imp

Hierarchical Paging : (Multilevel Paging)

- Most modern computer systems support a large logical address space (2^{32} to 2^{64}).
- In such an environment, the page table itself becomes excessively large.
- For eg: Consider a system with a 32 bit logical address space.
- If the page size in such a system is 4 kB then a page table may consists of upto 1 million entries ($2^{32}/2^{12}$).

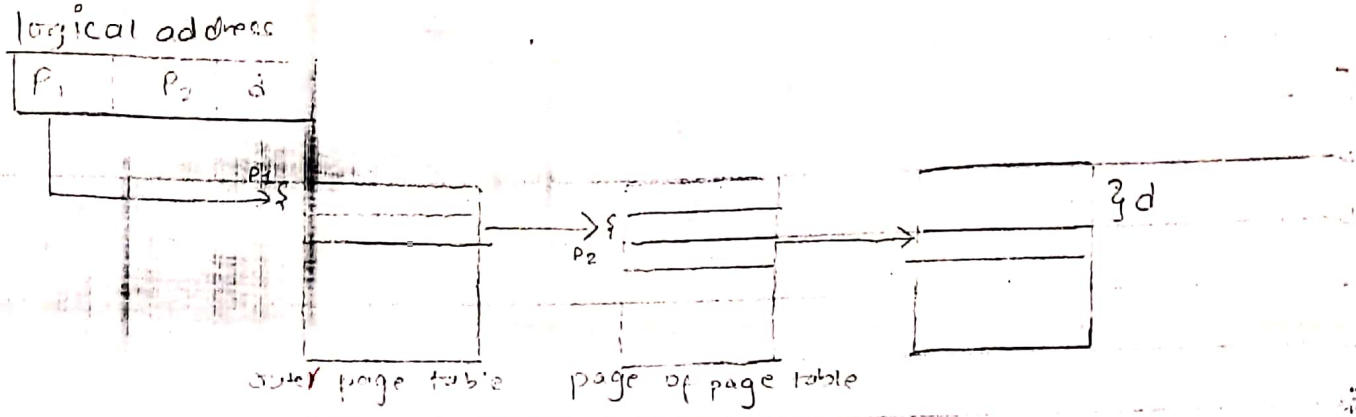
- Assuming that each entry consists of 4 bytes, each process may need upto 4 MB of physical address space for the page table alone.
- Clearly, we would not to allocate the page table contiguously in main memory.
- One simple solution to this problem is to divide the page table into smaller pieces.
- One way is to use a two level paging algorithm, in which the page table itself is also paged.
- Remember our example to our 32 bit machine with a page size of 4 kB.
- A logical address is divided into a page number consisting 20 bits and a page offset consisting of 12 bits.
- Because we page the page table, the page number is further divided into a 10 bit page number and a 10 bit page offset.

Thus, a logical address is as follows:



what is cache ratio? cache hit / cache miss

47.



(c) fig: Address Translation for a two level 32 bit paging architecture.

- where p1 is an index into the outer page table and p_2 is the displacement within the page of the outer page table.
- The address translation method for this architecture is shown in fig (c).
- Because Address Translation works from the outer page table inwards, the scheme is also known as forward mapped page table.

imp

Translation Lookaside Buffer (TLB): cache memory in paging hardware which enhance the speed of the translating virtual address to physical address. CACHE HIT - found in cache no need to translate address. CACHE MISS - address not found in cache and every address known as page miss.

- A translation lookaside Buffer (TLB) is a cache that memory management hardware uses to improve the virtual address transl-ation speed.
- All current desktop, notebook and server processors uses a

TLB to map virtual and physical address spaces, and it is nearly always present in any hardware which utilizes virtual memory.

- The TLB is typically implemented as content-addressable memory (CAM).
- The CAM search key is the virtual address and the search result is a physical address.
- If the requested address is present in the TLB, the CAM search yields a match quickly and the retrieved physical address can be used to access memory.
- This is called a TLB HIT OR (Cache HIT)
- If the requested address is not in the TLB, it is a miss, and called cache Miss OR TLB MISS then the translation proceeds by looking up the page table in a process called page walk.
- The page walk is an expensive process, as it involves reading the contents of multiple memory locations and using them to compute the physical address.
- After the physical address is determined by the page walk, the virtual address to physical address mapping is entered into the TLB.

Imp

Inverted Page table

- Usually each process has a page table associated with it.
- The page table has one entry for each page that the process using.
- This table representation is a natural one, since processes reference pages through the pages virtual addresses.
- The operating system must then translate this reference in a physical memory address.

- Since the table is sorted by virtual address, the OS is able to calculate where in the table the associated physical address entry is and to use that value directly.
- One of the drawback of this method is that each page table may consist of million of entries.
- These tables may consume large amounts of physical memory, which is required just to keep track of how the other physical memory is being used.
- To solve this problem, we can use inverted page table.
- An inverted page table has one entry for each real page (or frame) of memory.
- Each entry consists of the virtual address of the page stored in the real memory location, with information about the process that owns that page.
- Thus, only one page table in the system, and it has only one entry for each page of physical memory.
- Because only one page table is in the system, yet there are usually several different address spaces mapping physical memory, inverted page tables often require an address space identifier stored in each entry of the page table.
- Storing the address space identifier ensures the mapping of a logical page for a particular process to the corresponding physical page frame.

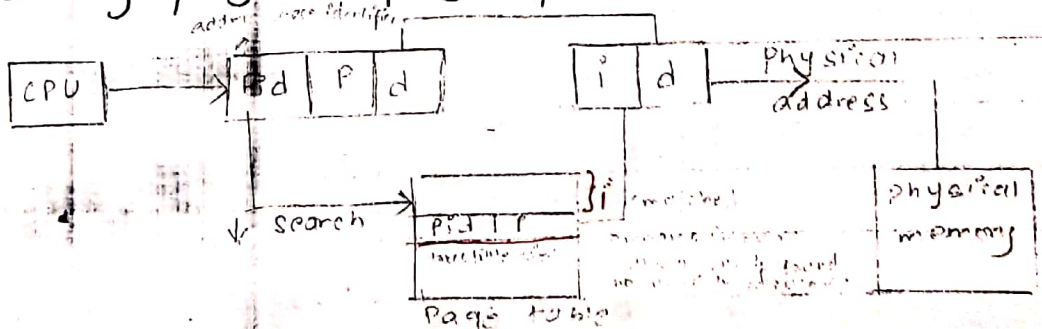


Fig: Inverted page table

Segmentation :

Even though the variables change their size during run time, segments cope with that problem.

What happens if program increase their size in their execution
- How to manage expanding and controlling table?
- How to protect only data from the program?
- How to share data to other program or functions?

The general solution of these issues is to provide the machine with many completely independent address spaces called segments.

- Memory management that supports variable partitioning and mechanisms with freedom of contiguous memory requirement restriction.
- The independent block of the program is a segment such as main program, prozedures, functions, methods, objects, local variables, global variables, common blocks, stacks, symbol table, arrays.
- The responsibility for dividing the program into segments lies with user (or compiler).

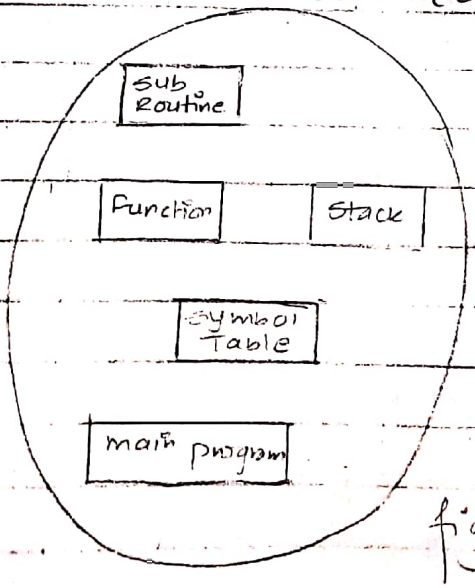


fig: Logical Address space

dividing particular segment into equal no. of pages / segmentation with paging.

- Different segments have its own name and size.
- The different segment can grow or shrink independently, without effecting the other, so the size of segment changed during execution.
- For the simplicity of implementation, segments are numbered and are referred to by a segment numbers rather than by segment name.
- Thus, the logical address consist : segment number and offset.
- The segment table (like page table but each entry consist limit and base register value) is used to map the logical address to physical address.

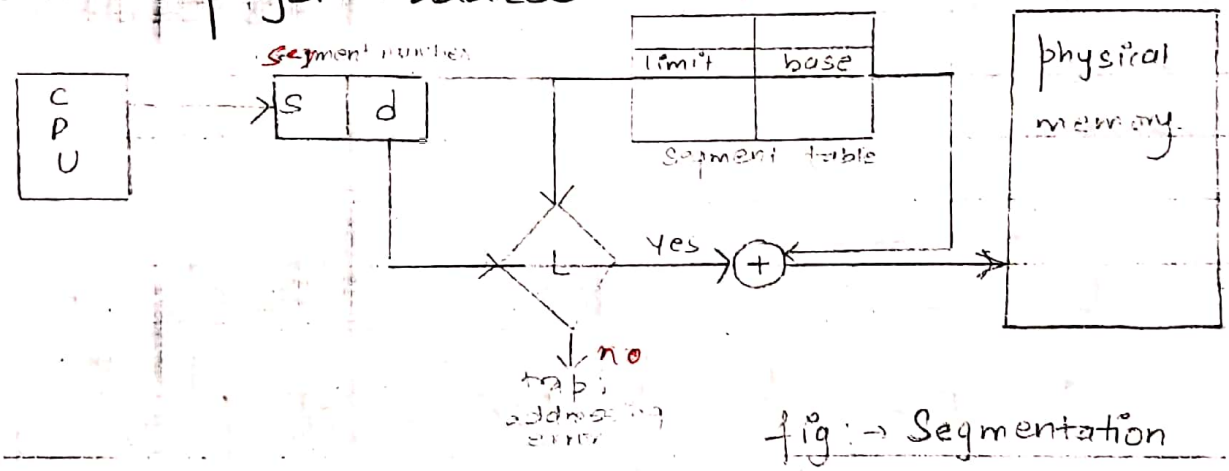


fig :- Segmentation

- The segment number used as index into the segment table. The offset d of the logical address must be between 0 and the segment limit.
- If not trap occurs, if it is legal it is added to the segment base to produce the address in the physical memory.

Segmentation with Paging :

- What happens when segment are larger than main memory?
- 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 with the segment.
- When paging is added, the segment offset is further divided into a page number and page offset.
- The segment table entry contains the address of the segments page table.

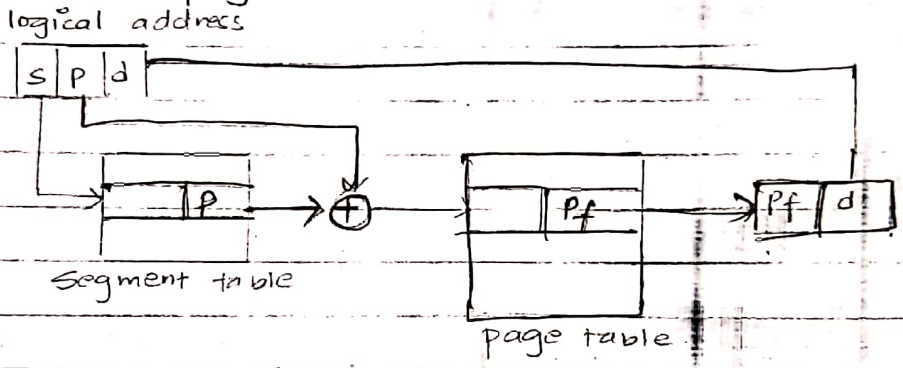


fig: Segmentation with Paging

- The Intel Pentium 80386 and later architecture uses segmentation with paging memory management.
- The maximum number of segments per process is 16k and each segment can be large as 4GB.
- The page size is 4k.
- It uses two level paging scheme.

#Virtual Memory:

- Virtual Memory is a concept that is associated with ability to address a memory space much larger than the available physical memory.
- The basic idea behind the virtual memory is that the combine size of the program, data and stack may exceed the amou of physical memory available for it.

Virtual memory -

- The OS keeps those part of the program currently in use in main memory and rest on the disk.
- Virtual storage is not a new concept, this concept is implemented by two most commonly used methods. Paging and segmentation or mix of both.
- Virtual memory is the separation of user logical memory from physical memory.
- This separation of user logical memory from the physical memory.
- This separation shows an extremely large virtual memory tobe provided for programmers when only a smallr physical memory is available.
- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available or about what code can be placed in overlays.
- Virtual memory is commonly implemented by Demand Paging.
- A demand paging system is similar to a paging system with swapping.
- Processes reside on secondary memory (which is usually a disk)
- When we want to execute a process, we swap it into memory.
- Rather than swapping, the entire process into memory, however 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 "swap" is technically incorrect.
- manipulates entire processes whereas a pager is

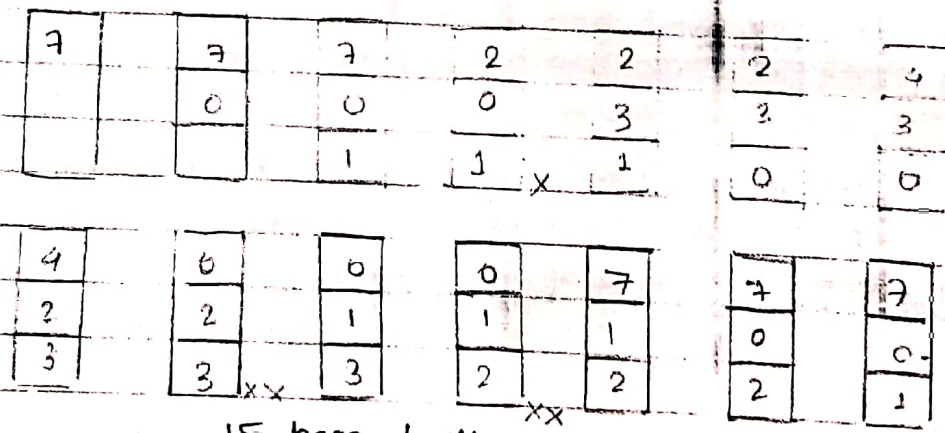
concerned with the individual pages of a process.
- We thus use pager rather than swapper in Demand Paging

V.V. Imp (long)

Page Replacement Algorithm :
- When a page fault occurs, the OS has to ~~make~~ choose a page to remove from memory to make the room for the page that has to be brought in.
- Which one page to be removed ?

ⓐ FIFO page Replacement : (oldest page is chosen).
- This associates with each page the time when the page was brought into memory.
- The page with highest time is chosen to replace.
- This can be implemented by using queue of all pages in mem.

Reference string :
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.



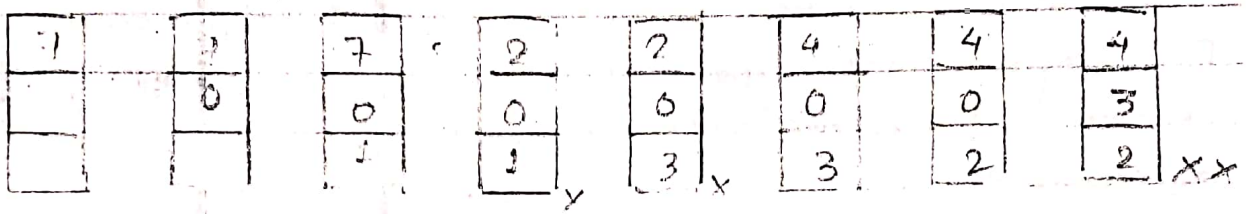
15 page fault

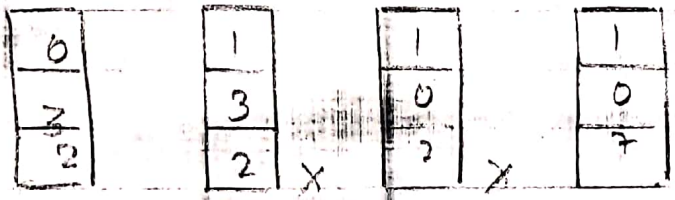
Advantages :
- Easy to understand
- Distributes fair chances to all.

Problems:
- FIFO is likely to replace heavily (or constantly) used pages and they are still needed for further processing.

LRU (Least Recently used)
(Page that has not been used for longest time)

- Reference string:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.

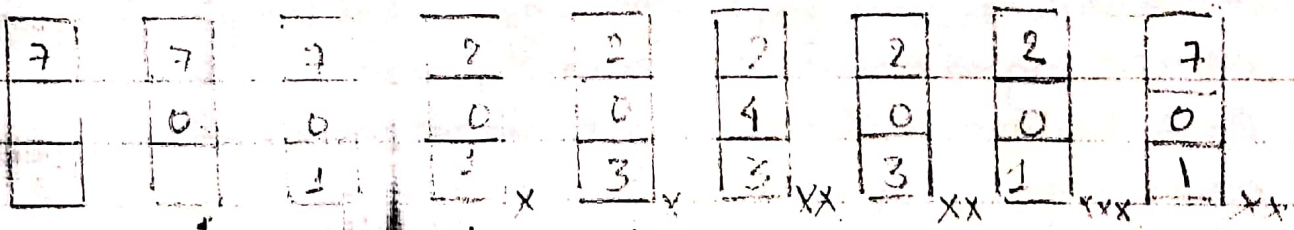




12 page fault.

Optimal page Replacement
=> Replace the page that will not be used for the longest period of time.

Reference string:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.



9 page fault.

(Clock Based Algo) (Circular queue)

Second chance Page Replacement Algorithm:

- The basic algorithm of second chance replacement is FIFO replacement algorithm.
- When a page has been selected, however, we inspect its reference bit.
- If the value is 0, we proceed to replace this page.
- If the reference bit is set to 1, however, we give that page a second chance and move onto select the next FIFO page
- When a page gets a second chance and move onto select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time.
- Thus, the page that is given a second chance will not be replaced until all other pages are replaced.
- In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.
- ~~In addition, if a page~~ One way to implement the second chance algorithm is a circular queue (clock based)
- A pointer indicates which page is to be replaced next
- When a frame is needed, the pointer advances until it finds a page with 0 reference bit.
- As it advances, it clears the reference bits.
- Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position.

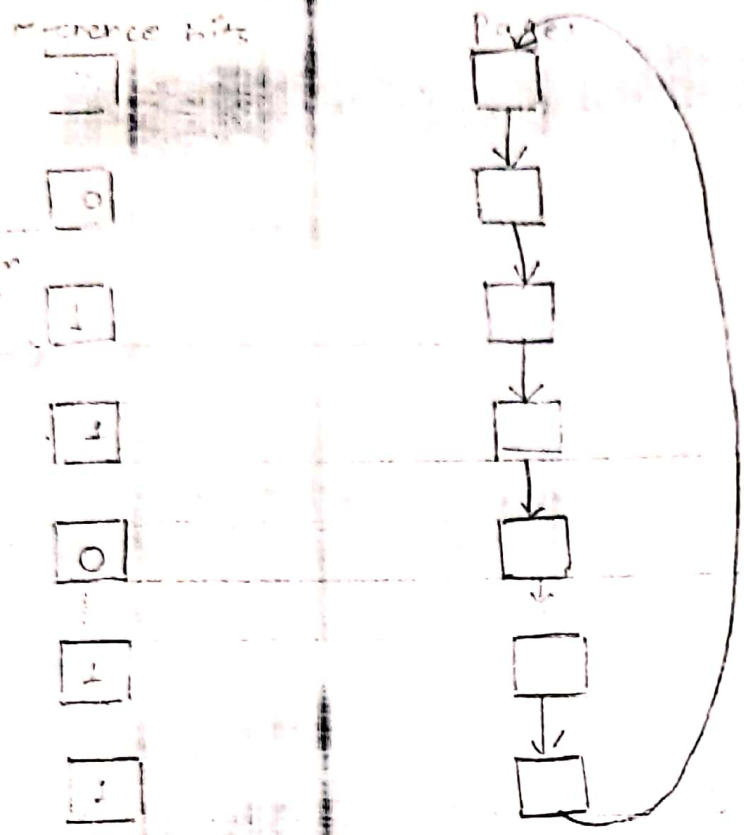


fig : Circular queue of pages

Counting based Page Replacement :

(a) LFU (Least Frequently used) :
- LFU requires that the page with the smallest count be replaced.

(b) Most Frequently used (MFU) :
- MFU requires that the page with the largest count be replaced.

Files :
- How to store the large amount of data into the computer?
- What happens when process terminates or killed using some data?
- How to assign the same data to the multiple processes?
- The solution to all these problems is to store information.

on disks or on other external media called files.
- A file is named collection of related information normally resides on a secondary storage device such as disk or tape.
- Commonly, files represent programs (both source and object forms) and data, data files may be numeric, alphanumeric or binary.
- Information stored in files must be persistent, not be affected by power failures and reboot.

File system issues:
- How to create files?
- How they are named?
- How they are structured?
- What operation are allowed on files?
- How they protect them?
- How they ~~acc~~ are accessed or used?
- How to implement?

File naming:
- When a process creates a file, it gives the file name, while process terminates, the file continue to exist and can be accessed by other processes.
- A file is named for the convenience of its human use and is referred to by its name.
- A name is ~~string of cha~~ stream of characters, that stream may be of digits or special characters. Some system differentiate between the upper and lower case characters whereas other system consider the

equivalent. Normally, the ~~stream~~ String of max. 8 characters are ~~stringed~~ legal file name in case of DOS but many systems support as long as 255 characters.
e.g. win dows.

File structure :
- Files must have structure that is understood by operating system.
- Files can be structured in several ways.
- The most common structures are :-
  (a) Unstructured
  (b) Record structure
  (c) Tree structure.

(a) Unstructure :
- Consist of unstructured sequence of bytes or words. OS does not know or care what is in the file.
- Any meaning must be imposed by user level programs.
- Provides maximum flexibility, user can put anything they want and name them anyway that is convenient.
- Both Unix and windows use these approach.

(b) Record structure :
- A file is a sequence of fixed length records each with some internal structure.
- Each read operation returns one record and write operation overwrites or append one record.

(c) Tree structured :
- File consists of records, not necessarily all the same length.

- Each containing a key field in a fixed position in the record, sorted on the key to allow the rapid searching.
- The operation is to get the record with the specific key.

File Types :

- Many OS supports several types of files.

(a) Regular files : contains user information, are generally ASCII or binary.

(b) Directories : system files for maintaining the structure of file system.

(c) character special files : related to I/O and used to model serial I/O devices such as terminals, printers and networks.

(d) Block special files : used to model disks.

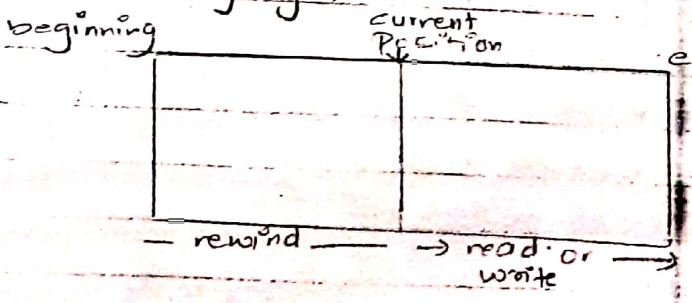
Access Methods :

(a) Sequential Access

(b) Direct / Random Access

(a) Sequential Access :

- The simplest access method, information in the file is processed in order, one record after the other.
- convenient when the storage medium is magnetic tape.
- used in early system.



(b) Direct Access / Random :
- Files whose bytes or records can be read in any order.
- Based on disk model of file, since disk allow random access to any block.
- Used for immediate access to large amounts of information.
- When a query concerning a particular subject arrives, we compute which block contain the answer and read that block directly to provide information.

File Attributes :
- In addition to name and data, all other information about file is termed as file attributes.
- The file attributes may vary from system to system.
- Some common attributes are listed here.

Attribute	Meaning
Creator	ID of the person who created file
Owner	Current owner
Protection	Who can access the file and in what way
Password	Password needed to access file.
Read only flag	0 for read/write, 1 for read only.
Hidden flag	0 for normal, 1 for do not display in listing.

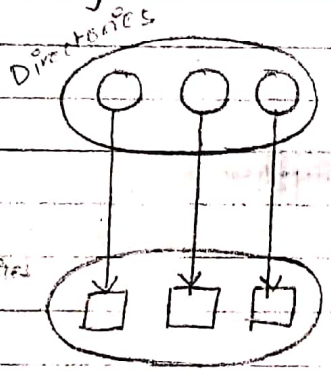
File Operations :
- OS provides system calls to perform operations on files.
- Some common calls are :

(a) Create : If disk space is available, it creates new file without data.

(b) Delete : Deletes files to free up disk space.

(c) Open: Before using a file, a process must open it.
(d) Close: When all access are finished, the file should be closed to free up the internal table space.
(e) Read: Reads data from file.
(f) Append: Adds data at the end of the file.
(g) Seek: Repositions the file pointer to a specific place in the file.
(h) get attributes: Returns file attributes for processing.
(i) set attributes: To set the user set attributes.
(j) Rename: Renaming a file.

Directory structure:
- A directory is a node containing information about files.



- Directories can have different structure.

(a) Single level directory:
- All files are contained in the same directory.
- Easy to support and understand, but difficult to manage.
- Large amount of files and to manage different users.

(b) Two level Directory:
- Separate directory for each user.
- Used on a multi user computer and on a simple network

computers.
- It has problem when users want to cooperate on some task and to access one another's file.
- It also cause problem when a single user has large number of files.

(c) Hierarchical directory:
- Generalization of two level structure to a tree of arbitary height.
- This allow the user to create their own subdirectories and to organize their files accordingly.
- To allow to share the directory for different user.

File system Implementation:
- How files and directories are stored?
- How disk space is managed?
- How to make everything work efficiently and reliably?

File system layout:
- Disks are divided up into one or more partitions, with independent file system on each partition.
- Each file occupy a set of contiguous block on the disk.
- Disk addresses define a linear ordering on the disk.
- File is defined by the disk address and length in block units.
- With 2KB blocks, a 50 KB file would be allocated 25 consecutive blocks.
- Both sequential and direct access can be supported by ~~es~~ contiguous alloration.

Allocation method :

(a) Contiguous Allocation :

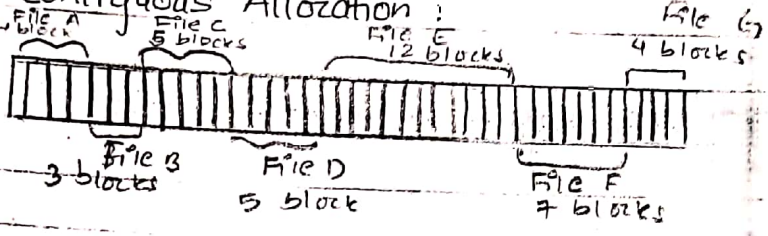


fig: Contiguous allocation method.

- Simple to implement, accessing a file that has been allocated contiguously is easy.
- High performance, the entire file can be read in single operation i.e decrease the seek time.

Problems :

- Fragmentation : when files are allocated and deleted, the free disk space is broken into holes.

(b) Linked Allocation :

- Each file is a link list of disk blocks, the disk block may be scattered anywhere in the disk.
- Each block contain the pointer to the next block of the same file.
- To create the new file, we simply create a new entry in the directory with linked allocation.
- Each block contain the pointer to the next block of same file.
- To create the new file, we simply create a new entry in the directory, with linked allocation, each directory entry has a pointer to the first disk block of the file.

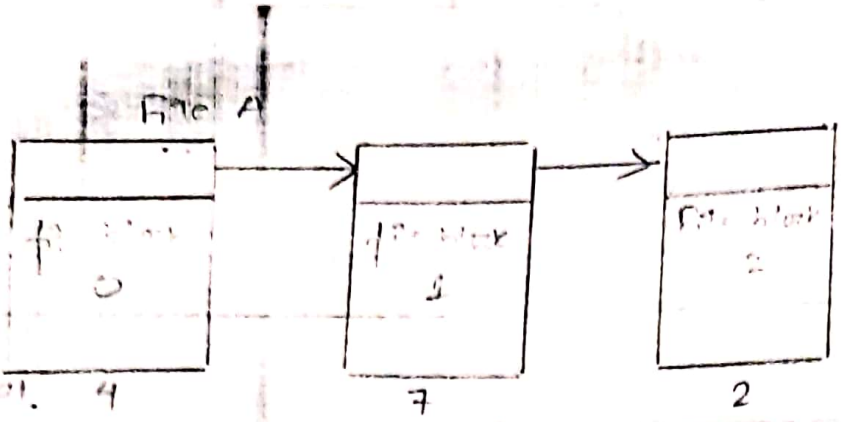


fig: Link list Allocation.

← Problems:

- It solves all problems of contiguous allocation but it can be used only for sequential access file, random access file is extremely slow.
- Each block access required disk seek.
- It also require space for pointer.

Solution: Using File allocation table (FAT).

- The table has one entry for each disk block containing next block number for the file.
- This resides at the beginning of each disk partition.
- The FAT is used as a linked list.
- The directory entry contains the block number of the first block of the file.
- The FAT is looked to find next block until a special end of file value is reached.

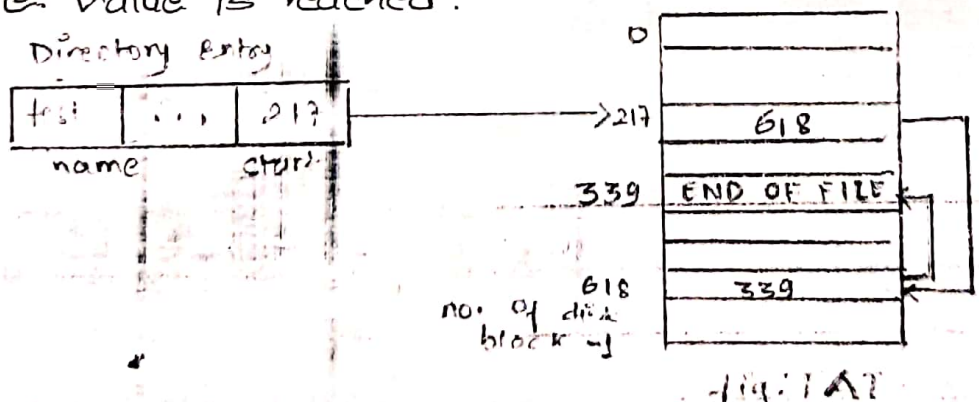


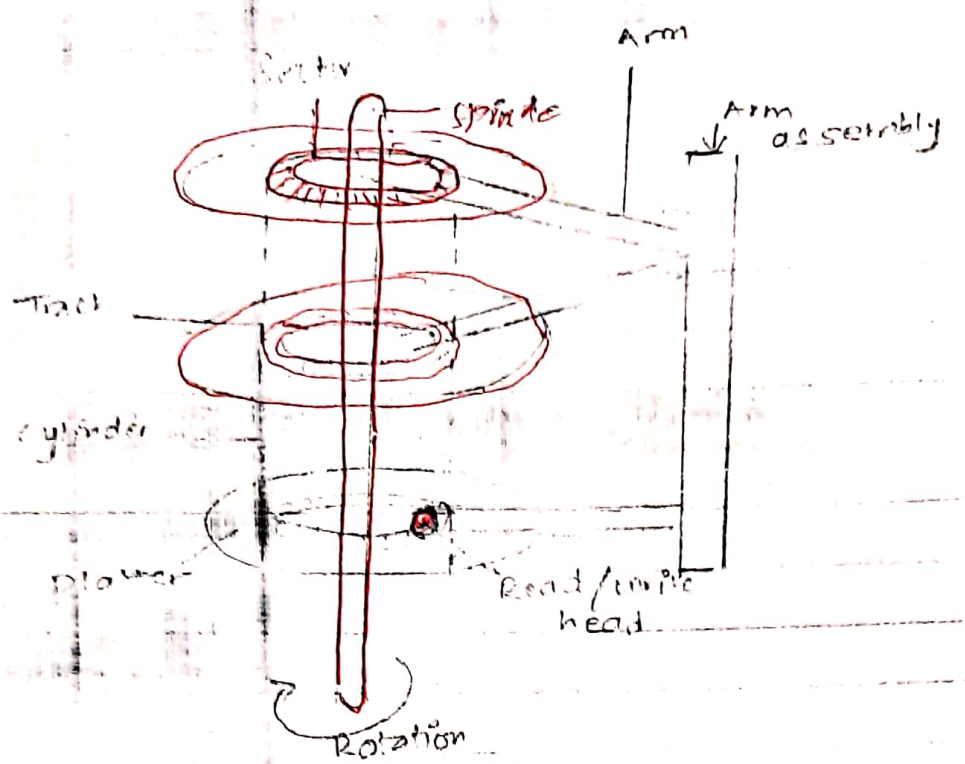
fig: FAT

Advantages:
- The entire block is available for data.
- Result the significant no. of disk seeks, random access time is improved.

© Index allocation:
- To keep track of which belongs to which file, each file has data structure (i-node) that list the attributes and disk address of the disk block associate with the file.
- each i-node are stored in a disk block, if a disk block is not sufficient to hold i-node, it can be multi-leveled.

Disk Management:
- Disk is an I/O device that is common to every computer; (Storage)

(a) Disk structures:
- Disk comes in many sizes and speeds and information may be stored optically or magnetically, however, all disks share a number of important features.
  for eg: floppy disks, hard disks, CD-ROM's and DVD's
- Disk surface is divided into number of logical block called sectors and tracks.
- The term cylinder refers to all the tracks at particular head position in hard disk.



(b) Disk operations:

- Latency Time: The time taken to ~~sat~~ rotate from its current position to a position adjacent to the read/write head.
- Seek: The processes of moving the arm assembly to new cylinder.

- To access a particular record, first the arm assembly must be moved to the appropriate cylinder and then rotate the disk until it is immediately under the read/write head.
- The time taken to access the whole record is called transmission time.

Disk Formatting

- Before a disk can store data, it must be divided into sectors that the disk controller can read and write called low-level formatting.
- The sector typically consists of preamble, data and ECC. (Error Correcting Code)

Single interleaving (one gap)

No interleaving (no gap)

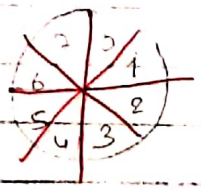
Possible double interleaving

- The preamble contains the cylinder and sector number and ECC contains the redundant information that can be used to recover from read error.
- The size depends upon the manufacturer, depending on reliability.

Preamble	Data	ECC

fig: Disk formatting. Error correcting codes.
↳ contains redundant info

- If disk I/O operations are limited to transferring a single sector at a time, it reads the first sector from the disk and doing the ECC calculation and transfers to main memory, during this time, the next sector will fly by the head.
- When transferring completes the controller will have to wait almost an entire rotation for the second sector to come around again.
- This problem can be eliminated by numbering the sectors in an interleaved fashion when formatting the disk.
- According to the copying rate, interleaving maybe no interleaving, single or double interleaving.



No interleaving

Single interleaving

Double interleaving

Error Handling:

- Most frequently one or more sectors becomes defective or most disks even come from factory with bad blocks.
- Depending on the disk and controller in use, these blocks handled in variety of ways.

1. Bad blocks are handled manually, For eg: run MS-DOS chkdsk command to find bad blocks and format command to create new block.
   - Data resided on bad blocks usually are lost.
2. Using bad block recovery: The controller maintains the list of bad blocks on the disk and for each bad block, one of the spares are substituted.

Imp

Disk Scheduling:

- The operating system is responsible for using hardware efficiently for the disk drive.
- This means having a fast access time and disk bandwidth.
- There are several algorithms exist to schedule the servicing of disk I/O requests.

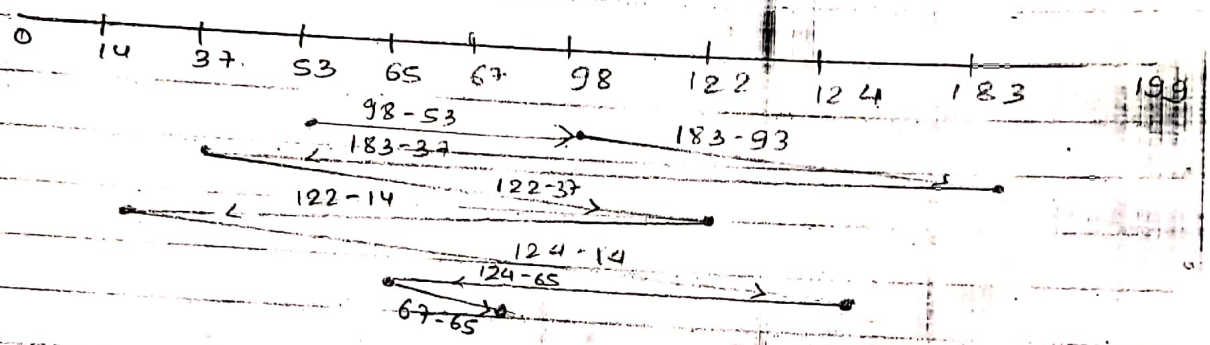
Different Disk scheduling algorithms are:

(a) First come First serve (FCFS).
(b) SSTF (shortest seek Time First)
(c) SCAN
(d) C-SCAN (Circular SCAN)
(e) LOOK
(f) C-LOOK (Circular LOOK).

(a) FCFS : (0-199)

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

Head pointer = 53



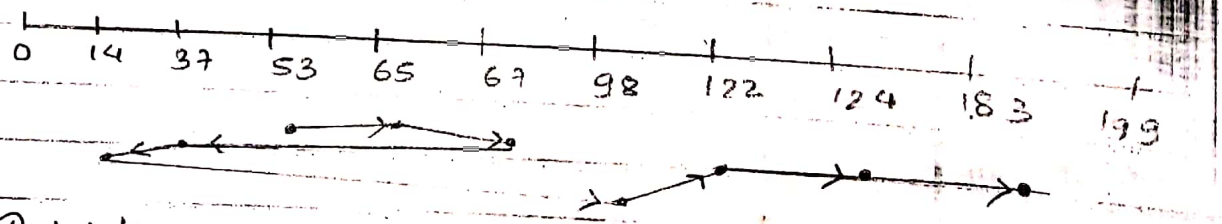
Total Head Movement = (98-53) + (183-93) + (183-37) + (122-37) + (122-14) + (124-14) + (124-65) + (67-65)

= 640 cylinders.

(b) SSTF : (0-199)

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

Head pointer = 53



Total Head Movement = (65-53) + (67-65) + (67-37) + (37-14) + (98-14) + (122-98) + (124-122) (183-124)

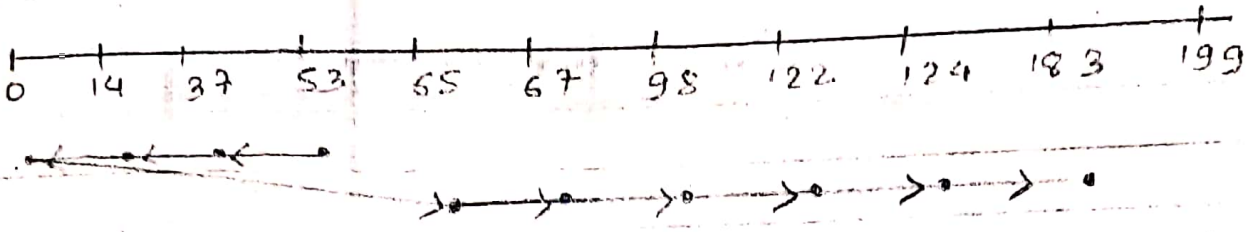
= ~~240~~ cylinders.

= 236

(c) SCAN : (0-199)

98, 183, 37, 122, 14, 124, 65, 67.

Head pointer = 53



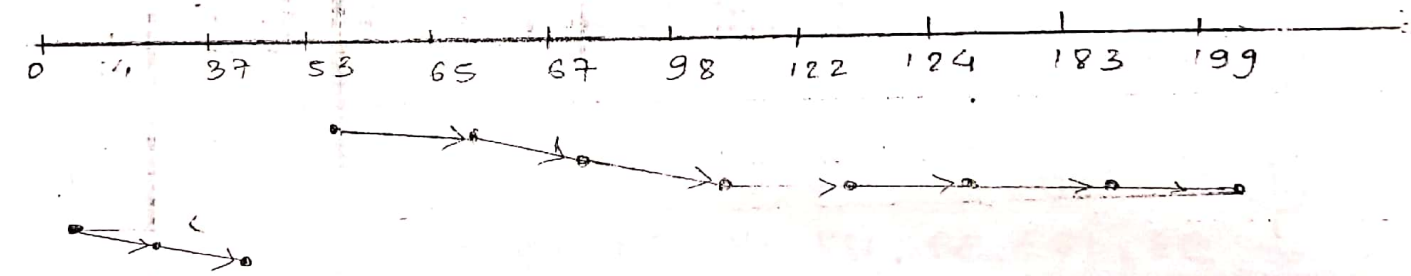
Total head Movement = (53-37) + (37-14) + ~~(14-0) + (65-0)~~ + (65-14)
(67-65) + (98-67) + (122-98) + (124-122) +
(183-124)

= ~~236~~ 208 cylinders.

(d) C-SCAN : (Circular SCAN):

98, 183, 37, 122, 14, 124, 65, 67.

head starts at 53.



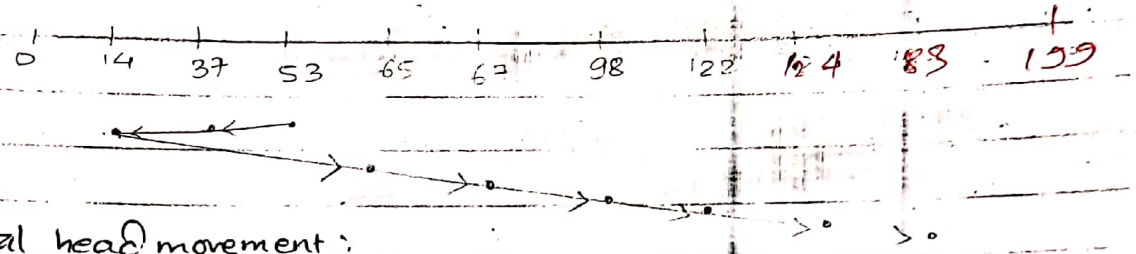
Total head movement = (65-53) + (67-65) + (98-67) + (122-98)
+ (124-122) + (183-124) + ~~(183-14)~~ + (37-14)

= 153

(e) LOOK :

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

head starts = 53



Total head movement :

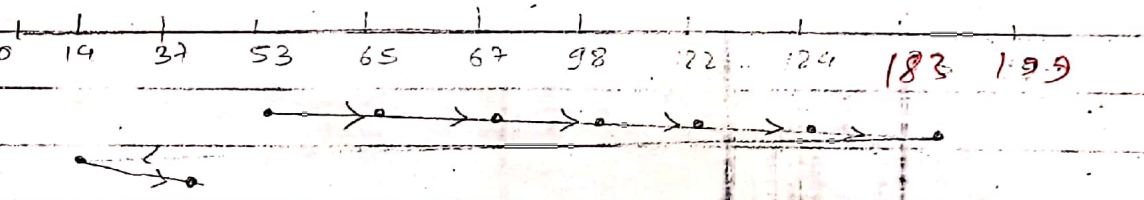
(53-37) + (37-14) + (65-14) + (67-65) + (98-67) + (122-98) + (124-122) + (183 - 124)

= 208 cylinders.

(f) C-LOOK :

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

head starts = 53



Total head movement = (65-53) + (67-65) + (98-67) + (122-98) + (124-122) + (183-124) + ~~(183-14)~~ + (37-14)

= ~~208~~ 153 cylinders.

(a) First come first serve (FCFS):

- the first request to arrive is the first one serviced.

(b) SSTF

- It selects the request with the minimum seek time from the current head position.

- SSTF scheduling is a form of SJF scheduling may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.

(c) SCAN :
- The disk arm starts at one end of the disk and moves towards the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes it is called the elevator algorithm.

(d) C-SCAN
- It provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other end servicing requests as it goes.
- When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing and requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

(e) LOOK :
- Same as SCAN except, it goes to the end point given in a queue.

(f) C-LOOK :
- Arm only goes as far as the last request in each direction, then reverses direction immediately without first going all the way to the end of the disk.

RAID (Redundant Array of Inexpensive / Independent disk)

Main issues : Disk Performance, Amount of storage required and Reliability.

- A technique of organizing multiple disks to address above issues is RAID.
- RAID allows more than one disk to be used for a given operation and allows continued operation and even automatic recovery in the case of disk failure.

RAID levels:

- There are six level of organizations called RAID levels.

=> for speed creating two or more arrays of one disk. divide a sentence and store it in array.
=> not reliable (requires enough back ups when it crashes)

@ RAID level 0 :

- RAID level 0 creates one large virtual disk from a number of smaller disks.
- Storage is grouped into logical units called strips with the size of a strip being some multiple of sector size.
- The virtual storage is sequence of strips interleaved among disk in the array.

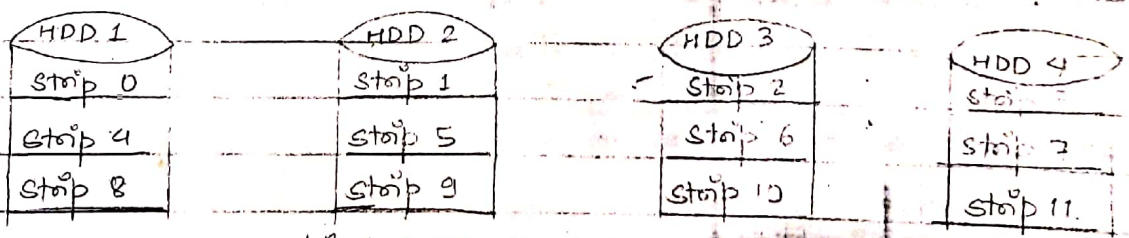


fig : RAID level 0

Advantages :

- can create large disk, performance benefits can be achieved.

Disadvantage :

- Reliability Decreases.

(b) RAID level 1 :
stores duplicate copy of each strip, with each copy on a different disk.

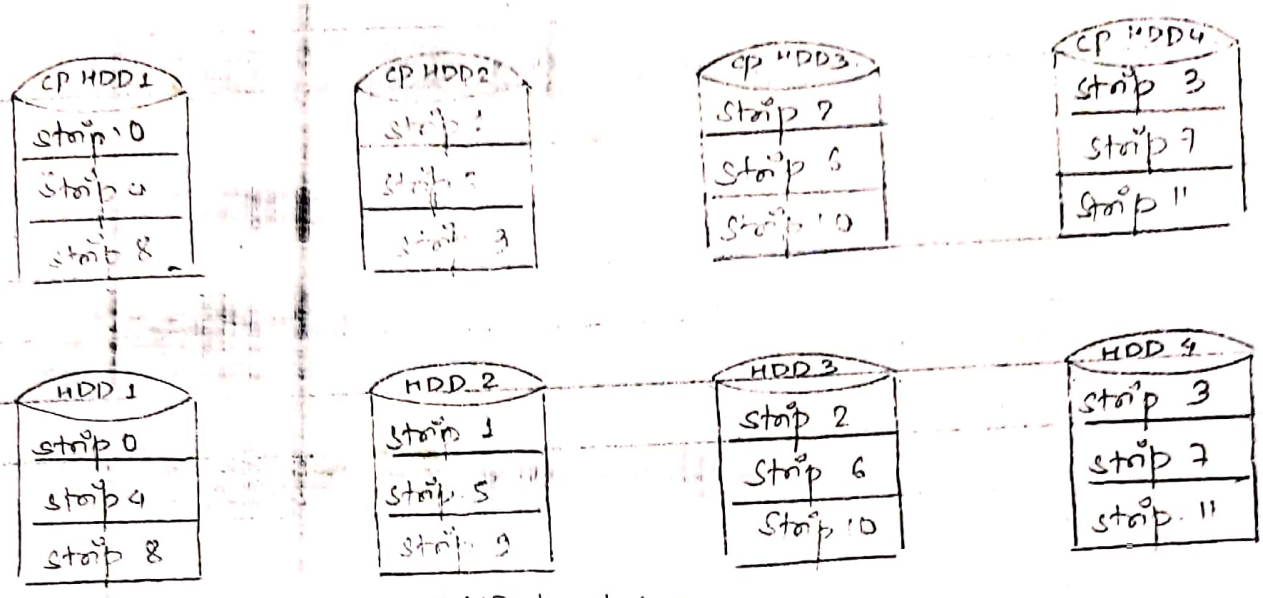


fig : RAID level 1

Advantages:
- Excellent Reliability, if drive crashes, the copy is used.
- Read Performance can be achieved.

Disadvantages:
- write performance is no better than in single drive.

(c) RAID level 2 :
- An error correcting code is used for corresponding bits on each data disks.
- Error correcting scheme stores two or more extra bits, and can reconstruct the data if a single bit gets damaged.
- for eg: the first bit of each byte is stored in disk 1, second bit in disk 2, and until eight bit in disk 8 and error correcting bits are stored in further disk.
- If one of the disk fails, the remaining bits of the byte and associated error correction bits can be read from other disks and can be

used to reconstruct the damaged data.

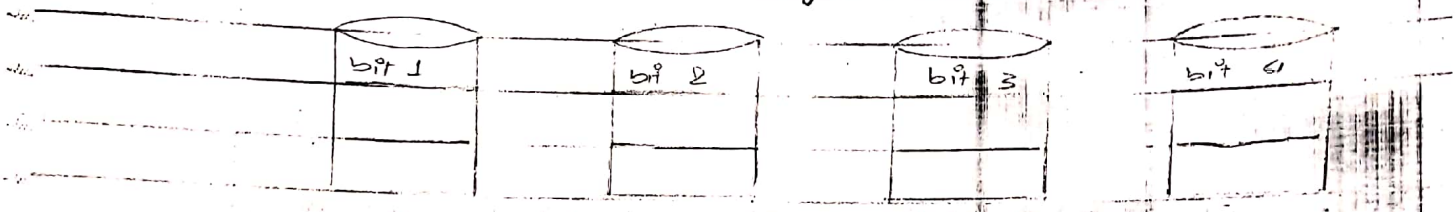


fig: RAID level 2.

Advantages:
- Total Parallelism

Disadvantages:
- Requires substantial number of drives.

(d) RAID level 3:
- Simplified version of RAID level 2.
- A single parity bit is used instead of error correcting code, hence required just one extra disk.
- If any disk in the array fails, its data can be determined from the data on the remaining disks.
- It is as good as level 2 but is less expensive in number of extra disks.

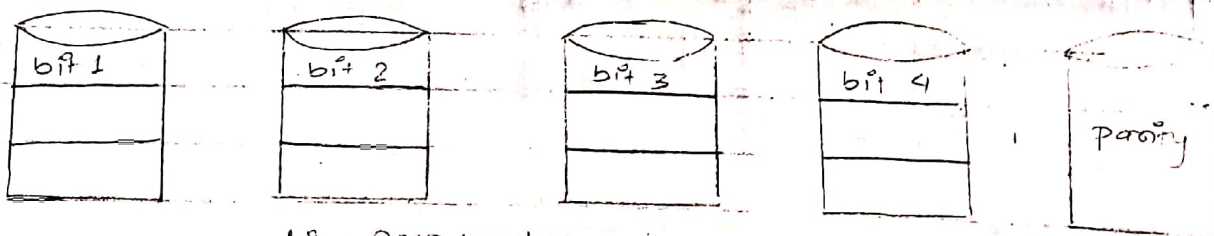
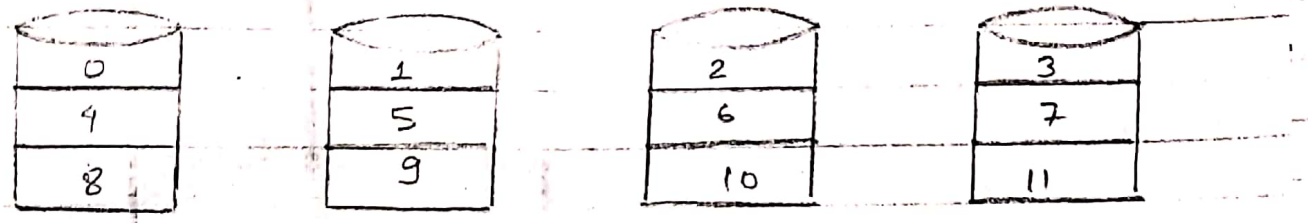


fig: RAID level 3.

(e) RAID level 4:
- It uses block level stripping as in level 0, and in addition keeps a parity block on separate disk for corresponding blocks from other disks.

If one of the disk fails, the parity block can be used with the corresponding blocks from other disks to restore the block of the fail disks.

- The transfer rate for large read as well as large write is high since reads and writes in parallel but small read and write cannot be in parallel.



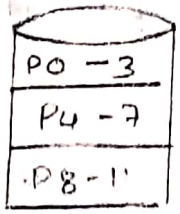


fig : RAID level 4

If one of parity of a block is removed, the parity of other parity recovers it.

Advantages
- more parallelism
- cost effectiveness

Disad : ran bottleneck parity.

(f) RAID level 5 :

- It is similar to level 4 but parity information is distributed in all disks.
- For each block one of the disks stores parity and other stores data.
- For eg : with an array of five disks, the parity for nth blocks are stored in disks.
- The nth block of the other four disks stores actual data for that block.

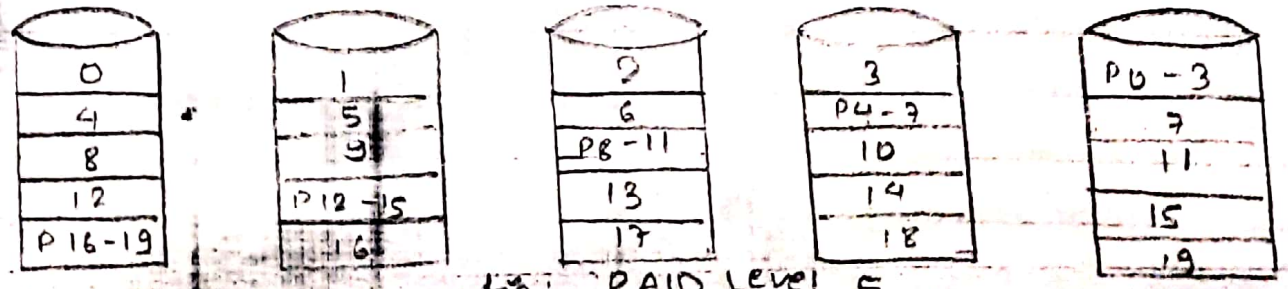


fig :- RAID Level 5

I/O Management:

- All computers have physical devices for acquiring input and producing output.
- OS is responsible to manage and control all the I/O operations and I/O devices.

Hardware organization

- The I/O devices, memory and the CPU communicates with each other by way of one or more communication buses.

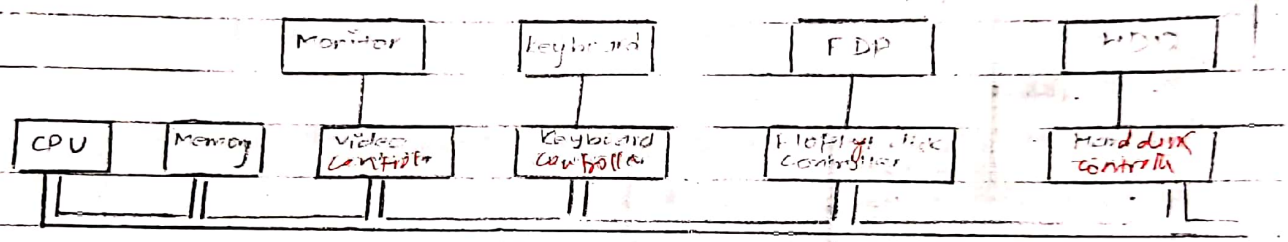


fig: single bus Architecture.

I/O Devices:

- The I/O units which consist mechanical components are called I/O devices such as hard disk, printer, etc.
- There are 2 types of devices:
  (a) Block devices
  (b) Character devices

(a) Block devices: ⇒ It stores information in fixed size blocks each one with its own address. Read or write is possible independent to other blocks direct access. eg: Disk.

(b) Character devices:

- Delivers or accepts a stream of characters without regard to any block structure
- It is not addressable.
- Eg: printer.

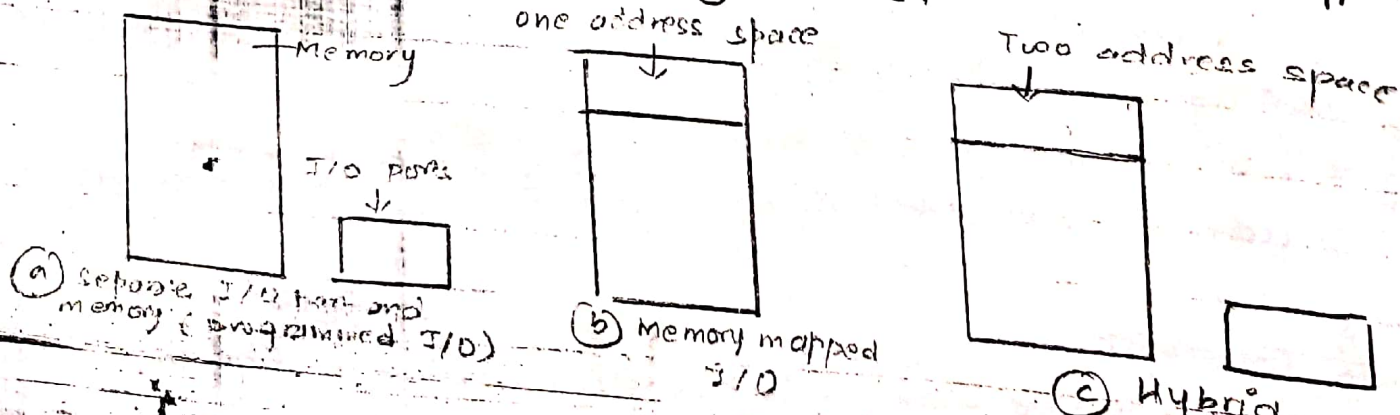
→ synchronization with processor

Device Controllers : (as a mediator to fulfil the demands of processor)

- A controller is a collection of electronics that can operate a bus or device.
- On PC, it often takes the form of printed circuit card that can be inserted into an expansion slot.
- A single controller can handle multiple devices, some devices have their own built in controller.
- The controller has one or more registers for data and signals.
- The processor communicates with the controller by reading and writing bit patterns in these registers. (data are accumulated in a buffer it keeps the data as bits.
- When transferring a disk block of size 512 bytes, the block first assembled bit by bit in a buffer inside the controller.
- After its checksum has been verified and the block declared to be error free, it can then be copied to main memory.

Memory mapped I/O : read data from device and map the data to memory via I/O is called programmed I/O

- Device controller have their own register and buffer for communicating with the CPU, by writing and reading these register, OS performs the I/O operation. data are accumulated in a buffer and directly map into memory → memory mapped I/O
- The device control registers are mapped into memory space is called memory mapped I/O.
- Usually, the assigned addresses are at the top of the address space.
- Some system (eg: Pentium) use both techniques. (Hybrid)
- The address 640k to 1m is being reserved for device data buffers in addition to I/O ports 0 through 64k.



Direct Memory Access : (DMA)

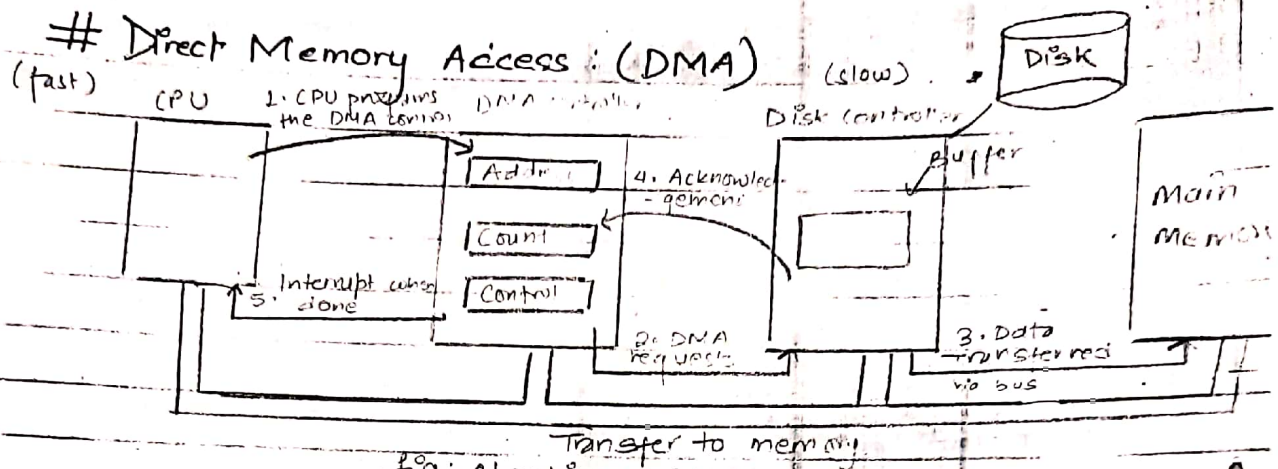


fig: Operation of DMA

Polling :

- Processing is interrupted at brief intervals to allow the CPU to check back to I/O to see if the operation has completed
- well manner way of getting attention.

Advantages :

- Simple

Problem :

- May bear long wait.
- can be a high overhead - inefficient.
- used in embedded system, where CPU has nothing else to do.

Interrupt :

- The hardware mechanism that enables a device to notify the CPU is called an interrupt.
- Interrupt forced to stop CPU what it is doing and start doing something else.

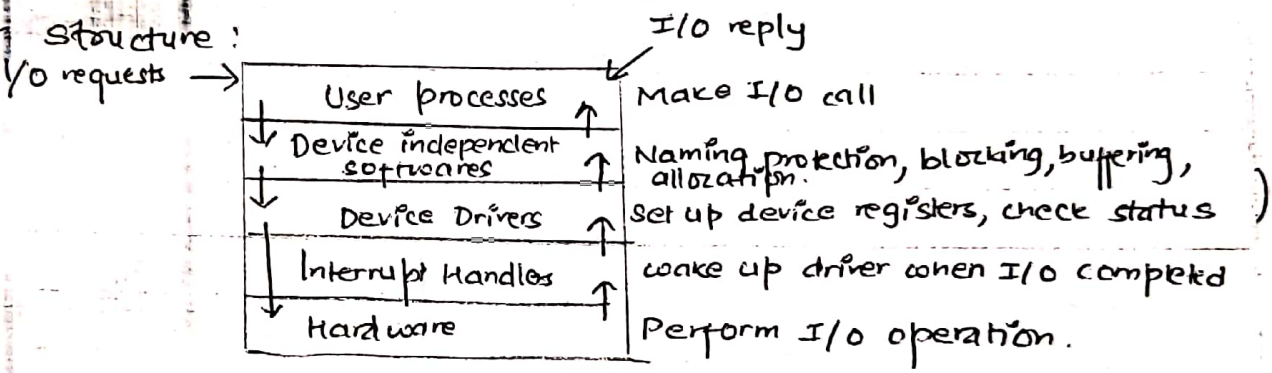
Advantages :

- It improves efficiency.

Problem:
- Because of the selfish nature, the first interrupt may not be served if the similar second happen before the time needed to serve the first.

I/O software layers:
(a) Device Independence
(b) Uniform Naming
(c) Error Handling
(d) Synchronous and Asynchronous Transfer
(e) Buffering.

Layer structure:



(a) User processes: → system calls including the I/O system calls are normally made by the user processes.
- User processes put their parameters in the appropriate place for the system calls, or other procedures that actually do real work.

(b) Device Independent softwares:
- The devices independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user level software.

(c) Device Drivers:

- Each I/O device attached to a computer, needs some device spec code for controlling it, called device driver is generally written b the device's manufacturer and delivered along with the device
- Accept read and write requests from the device independent softw above it.
- Initializes the devices if necessary.

(d) Interrupt Handlers:
- Block the driver until the I/O has completed and the interrup occur.
- The interrupt handler determines the cause of the interrupt and performs the necessary processing.

DMA:
- To explain how DMA works, let us first look at how disk reads occur when DMA is not used.
- First the controller reads the block from the drive serially bit by bit until the entire block is in the controller's intern buffer.
- Next it computes the checksum to verify that no read er have occurred.
- Then the controller causes an interrupt.
- When the OS starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop with each iteration reading one byte or word from a controller device register and storing it in main memory.
- When DMA is used, the procedure is different.
- First the CPU programs the DMA controller by setting

registers.
- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum.
- When valid data are in the disk controller buffers DMA can begin.
- The DMA controller initiates the transfer by issuing a read request looks like any other read request and the disk controller does not know or call whether it came from CPU or from a DMA controller.
- This read request looks like any other read request and the disk controller does not know or care whether it came from CPU or from a DMA controller.
- Typically, the memory address to write to its on the bus address line so when the disk controller fetches the next word from its internal buffer, it knows where to write it.)
- The write to memory is another standard bus cycle.
- When the write is complete, the disk controller sends an acknowledgement signal to the disk controller.
- DMA controller then increments the memory address to use and decrements the byte count.
- If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0.
- At that time DMA controller interrupts the CPU to let it know that the transfer is now complete.

registers.

- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum.
- When valid data are in the disk controller buffers DMA can begin.
- The DMA controller initiates the transfer by issuing a read request looks like any other read request and the disk controller does not know or call whether it came from CPU or from a DMA controller.
- This read request looks like any other read request and the disk controller does not know or care whether it came from CPU or from a DMA controller.
- Typically, the memory address to write to its on the bus address line so when the disk controller fetches the next word from its internal buffer, it knows where to writ it.)
- The write to memory is another standard bus cycle.
- When the write is complete, the disk controller sends an acknowledgement signal to the disk controller.
- DMA controller then increments the memory address to use and decrements the byte count.
- If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0.
- At that time, DMA controller interrupts the CPU to let it know that the transfer is now complete.