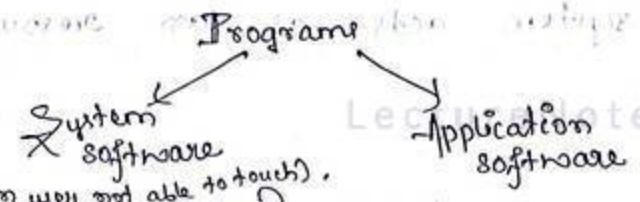
14/7/2015

Operating System

An operating system is a program that manages the computer system.

It acts as an intermediatary between user and computer system. It is nothing, but a control program that manages or controls all the activities that has been performed inside the computer system.

* Broadly, the computer system is divided into 4 parts :
  (i) computer hardware
  (ii) user
  (iii) operating system (system software) } software.
  (iv) application program.



(common user not able to touch).

Work of operating system :

(i) Process Management.
(ii) Main memory
(iii) Secondary memory.
(iv) File memory
(v) Security.

15/7/2015

OS should be a control program and a resource allocator.

*- Users v.

Users View of Operating System.

*- Operating system is designed for easy use of resources in the computer system with high performance.

*- The OS is designed to maximize the resource utilisation to assure that all available resources like CPU time, memory and I/O devices are used efficiently and effectively.

System's View of Operating System.

*- Operating system acts as an resource allocator.

*- A computer system has many resources.

*- The OS acts as the manager of all the resources and will decide which resource will be allocated to which process at what point of time.

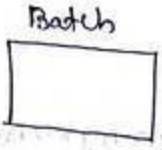
*- Operating system should behave as a control program. It manages the execution of user programs to prevent errors and improper use of resources. It provides a reasonable way to solve the problem.

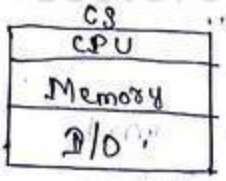
Types of Operating System.

(i) Batch processing O/S.
(ii) Multi-programming O/S.
(iii) Time sharing O/S.
(iv) Multiprocessing O/S. → separate processors one memory, LAN
(v) Distributed O/S. → WAN, separate workstations
(vi) Real time O/S.

(i) **Batch Processing O/S.** (Spooling) (as spool is present in batch).

Limited resources as one CPU, one memory. Programs are stored in batch.





Spooling: Simultaneous peripheral operations online.

O/S checks programs of similar needs & gives to CPU for output.

Drawback:

Resource utilisation is less. as when one works other remains idle.

* Output is time taking.

ii) Multiprogramming O/S.

* Instead of batch, single single programs will run, but simultaneously.

Prog 1	Prog 2
I/O	I/O
CPU	CPU

O/S gives I/O to 1, CPU to 2.

Resources don't sit idle.

iii) Batch Processing O/S.

* With this O/S, several programs of similar needs are batched together on a single input tape and processed through CPU as a group.

* The programmer sorts the programs into batches with similar requirements and runs each batch. This will give the output at some later time.

Demerits

* Here, most of the time CPU sits idle.
* Component ideal idle times due to slow manual operations.
* Turn around time from the time of submission of a program to until it gives the output is very long.
* Resource utilisation is very low.

It is also known as spooling.

16/7/2015.

Multi-programming OS.

* Use Concurrent execution of programs can significantly improve the system throughput and resource utilisation as compared to batch processing operating system.
* Multiprogramming incs. CPU utilisation by organising the programs & concurrently executing these programs so that, CPU or I/O devices are busy at all times.
* Here, OS keeps all the programs in memory simultaneously. It picks up one program at a time and execute it. During this,

execution, the program may have to wait for some other task like I/O operation.

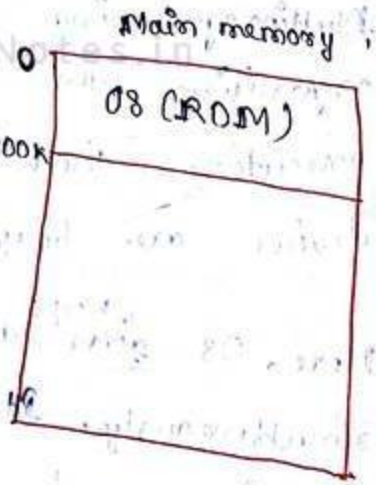
*- In non-multiprogramming environment, during this period of I/O operation CPU sits idle. But, in multi-programming environment, OS picks up another program & executes that.

*- CPU never sits idle as long as atleast one job is present.

*- When several jobs are scheduled for execution and or ready to acquire CPU at the same time the OS must choose one of them. This decision is known as CPU scheduling.

Time Sharing/Multi-tasking O/S.

*- It is nothing but, the logical extension of multi-programming. In this OS, each program gets a fixed amount of time.



*- The CPU executes multiple programs by switching among the

programs.

* Switching occurs so frequently that user can interact with each program while it is running. It provides good response time. It allows many users to share the computer simultaneously. It use CPU scheduling with multi-programming.

17/7/2015

iv) Distributed OS:

Distributed operating system is a collection of autonomous computers capable of communication & cooperation through hardware and software connections.

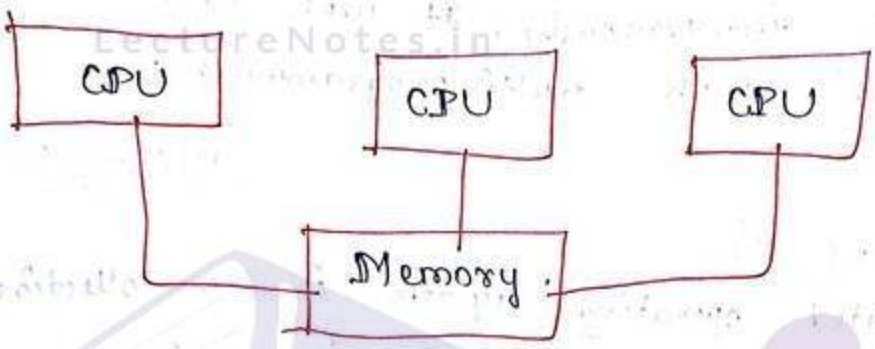
* The major objective of distributed OS is transparency.

* The component and resource distribution should be hidden from users and application programs unless explicit demand.

* Distributed OS provides the means for systemwide sharing of resources such as computational capacity, files, etc.

iv) Multi-processing / Parallel O/S.

*- More than one processor in close communication, sharing the computer bus, the clock and sometimes memory and peripheral devices.



*- The advantage of multi-processing OS is

(i) increasing throughput.

By increasing the number of processors, we can expect more work to be done from the system.

(ii) increased reliability.

If the function can be distributed properly among several processors, then the failure of one processor will not halt the system. only slows down the processor.

(iii) economic in scale.

Multi-processor systems can save more money than multiple single processor system because they can share peripherals, storage devices, etc. When several process work on same set of data, then cost of having separate components for separate processors can be reduced.

*- There are two types of multi-processors:

(i) Symmetric Multi-processor.

Each processor runs an identical copy of an OS and these copies communicate with other on demand or as needed.

(ii) Asymmetric Multi-processor.

Each processor is assigned a specific task. Here, a master processor controls the system and other processors have their own pre-defined task.

(vi) Real time O/S.

*- Real time OS are specifically designed to respond the events that happens in real

time. This includes computer system for DCU equipments, emergencies, etc.

*- A real time system have well-designed fixed time constants.

*- Processing must be done within the defined constraints, otherwise it is failed.

Responsibilities (Functions) of OS. (Function).

(a) Process Management.

*- OS is a manager of all manager (or system) resources. It performs the following fun's:

(a) Process Management.

*- It is the responsibility of OS to create new process and execute it. It also performs process scheduling and dispatching. It also suspends a process and resumes a process.

So, inter-communication process communication can be done by the OS.

-x-

21/7/2015.

I/O bound process: where most of the processes are I/O operations.

CPU bound process: most are the CPU operations.

(iii) Main Memory Management.

* In multi-programming environment, multiple programs are left kept in memory simultaneously.

* Main memory management keeps track of which segment of memory is in use and by which process. It also decides which process are to loaded into memory when space becomes available. It also performs allocation and deallocation of dynamic memory.

* Swapping of the process is also done by main memory management unit (MMU).

(iii) Secondary Memory Management.

* Secondary memory keeps all the programs and process in job pool.

* The programs will be selected by the long term scheduler to be there in main memory for execution.

(iv) Device Management.

* It grants the requests or release the devices. It also performs read, write, append operations.

v) File Management.

* File management involves creating a file, deleting a file, creating a directory and deleting a directory. It also involves opening & closing of files.

* Read, write, repositon operation can also be performed.

vi) Protection of the system.

* Protection is a mechanism for controlling the access of programs and process to user to the resources defined by the computer system. It includes protection of resources from user process, protection of user process from each other and protection of operating system from user process. This can be achieved by ensuring that all system resources are accessed by user process only through operating system.

System Call.

create() → to create new process
fork() → to create child processes.

System call is the programming interface to the services provided by the operating system which is written in a high level language or assembly language.

* Each system call is associated with a no.

* System calls are accessed by a program through a high speed application program interface (API).

* 3 most common API's are :
  i) Win32 : used for windows.
  ii) POSIX : used for linux.
  iii) Java API : JVM.

Ex: of system call to copy the content of one file to another file.
i) Acquire the input file name.
ii) Acquire the output file name.
iii) Open the input file (if file doesn't exist, abort).
iv) Create the output file.

(v) Read from the input file and write into the output file.

(vi) Close the output file.

(vii) Terminate normally.

Types of System Call.

(i) Process control system call

(ii) File manipulation system call

(iii) Device manipulation system call.

(iv) Information maintenance system call

(v) Communication system call.

O/S Services.

(i) Program execution.

(ii) Communication.

(iii) File manipulation.

(iv) Resource allocation

(v) Error detection & recovery

(vi) I/O operation.

(vii) Accounting

(viii) System protection

* OS provides an environment for execution of user program. For this, it provides certain services to the programs and user of the programs for accessing system resource

(i) Program Execution.

* The main purpose of OS is to provide an efficient and convinient environment for the execution of programs. So, OS must provide various fun's for loading of a program into RAM and execute it.

* Each executing program must terminate normally or abnormally.

(ii) File Manipulation.

* Each executing program need to create, delete and manipulate the files which is managed by OS.

(iii) Communication

* OS manages inter-process communication between the process that are executing on same system or running on different systems on a distributed environment.

* OS must provide mechanism for inter-process communication like mailbox, shared memory, etc.

iv) Resource allocation.

* When multiple users are executing the programs concurrently, resources needs to be shared among them.

* The OS will decide the allocation of resources.

v) Error detection and recovery.

* Errors may occur during the execution of a program like division by zero, memory access violations, etc.

* OS should provide mechanism for detection of such errors and handles the errors.

vi) I/O Operations.

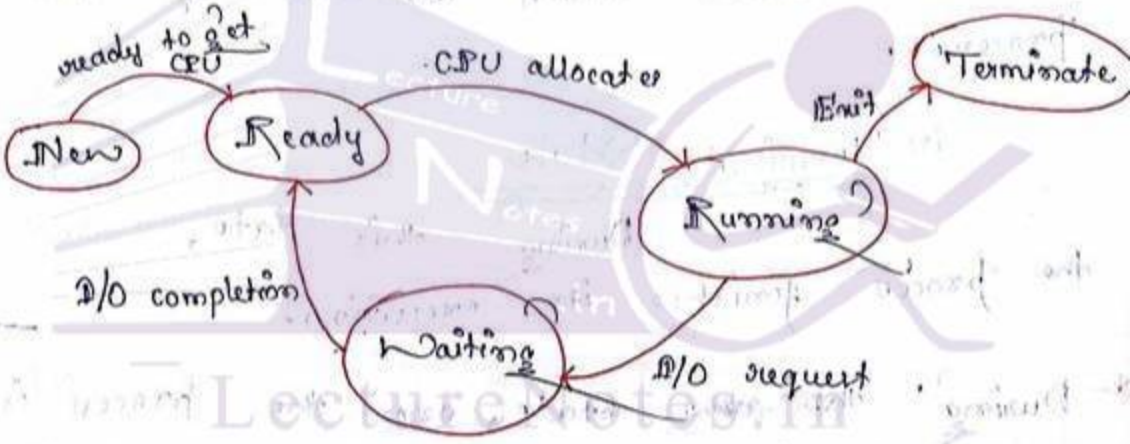
Q.

Process.

A process is an instance of a program in execution.

*- A program by itself is not a process.

*- A program is a passive entity, whereas process is an active entity, where the program counter specifies the next instruction to be executed and set of resources.



States.

i) Process State.

*- While process executes, it changes its state.

*- The state is nothing but, the current activity of the process.

*- The various states are:

ii) New State.

The process is created.

create system call is used.

(iii) Ready State.

The process in this state is ready to acquire CPU.

(iii) Running State.

During this state, the processes are executed.

(iv) Waiting State.

The processes are waiting for the resources which are held by other processes.

(iv) Terminate State.

During this state, the process finishes its execution.

* During the new state, once the process is created, it is active to acquire system resources such as processor and I/O devices.

- Only one process can be in running state at any instance of time, though many processes are available.

- When a running process becomes suspended due to interrupts or I/O requests

the operating system may schedule another process for execution.

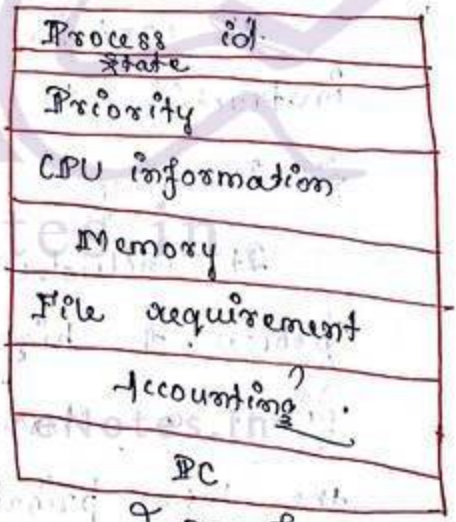
* Process is dynamic in nature that refers to a program in execution undergoing all the states.

* An executable program on the other hand, is a static process, that may give rise to one or more processes.

29/7/2015.

Q. PCB (Process Control Block)

* The OS groups all information that it needs about a particular process into a data structure is known as process control block.



* When a process is created, the operating system creates a corresponding process control block to solve the OS during the lifetime of the process. [PCB]

* When the process terminates, its PCB is also released.

A process can only be eligible for execution only when it has an active PCB associated with it. It contains many pieces of information such as

(i) process state

This may be new state, ready, running, terminate states. During the execution, the process changes its states. So, these changed states are reflected in PCB.

(ii) program counter.

It contains the address of the next instruction to be executed.

(iii) Priority

It includes the priority of a process. A high priority process is given a chance to execute first than the low priority process.

(iv) process id.

When a new process is created, a unique id is generated for that process. and this id will be stored in PCB. for identification of

that process.

(v) memory management information.

It includes the value of base and limit registers, the value of page table or segment table.

(vi) I/O status information.

It includes the list of I/O devices allocated to a process, a list of open files and access rights of files.

(vii) CPU scheduling information.

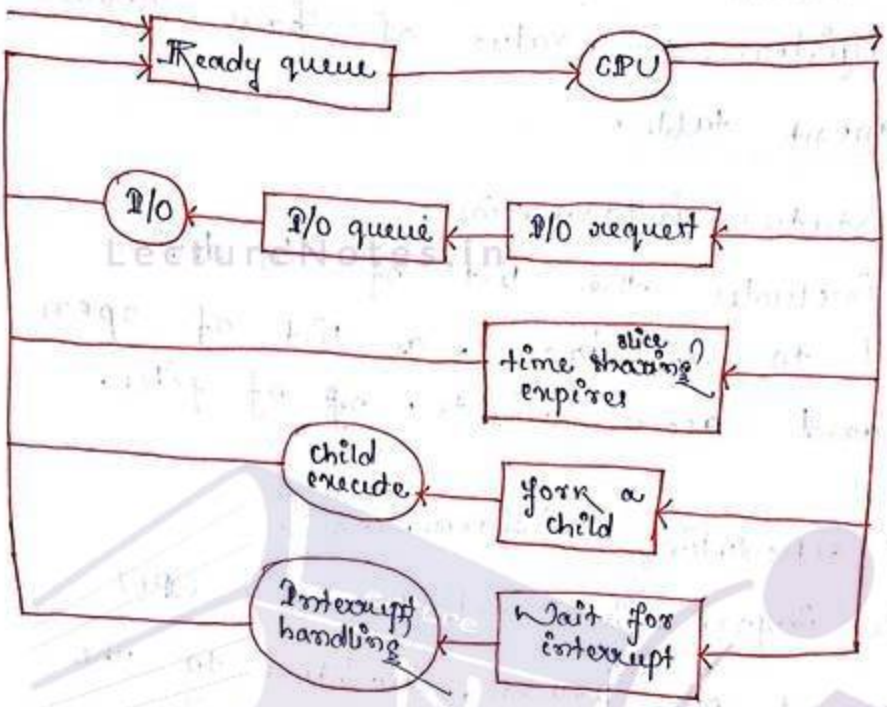
This information includes the CPU requirement of process, pointers to the scheduling tables queues. It also includes scheduling information.

(viii) Accounting

This includes amount of CPU used, time limits, account number, process number.

Process Scheduling

fork → to create child process inside a process.



Scheduling refers to the set of instructions and mechanisms that gives the order in which the processes is to executed.

Scheduler is an OS module that selects the a process to be kept in main memory and the next process to be executed.

* For process scheduling, different scheduling queues are required.

(i) Ready Queue

* The processes that are residing in main memory and are ready and waiting to

execute in CPU are kept in ready queue. This queue are implemented by a linked list.

(iii) Device Queue.

*- The list of processes waiting for a particular I/O device are kept in device queue.

(iii) Job Queue.

*- As the processes enter the system, they are kept in job queue. This queue consists of all the processes in the system.

Schedulars.

*- A process moves inbetween various scheduling queues throughout its lifetime.

*- The selection of the process for this scheduling queues are done through schedulars.

*- There are three types of schedular:
  (i) Job / Long schedular
  (ii) CPU / Short schedular.
  (iii) Medium term

(i) Job Schedular.

*- Long term schedular works with batch queue.

*- The process when submitted to a system are placed into a mass storage device known as job queue.

* The long term scheduler selects the processes from this queue and loads it to the main memory for execution.

* The primary objective of long term scheduler is to provide a balance mixed of I/O bound process and CPU bound process.

31/7/2015.

CPU Scheduler.

* Short-term scheduler allocates the CPU to the processes that are ready to execute and are in ready queue. Its main objective is to maximise the system performance. Since it is in charge of ready to running state transitions, it must be invoked for each process to select the next process to be executed.

* Short-term scheduler is invoked whenever an event occurs for changing of the state.

Medium term Scheduler.

* The running process may be suspended by executing due to I/O requests, by issue of system call, due to the interrupt occurs,

due to timeslice expires.

* A suspended process cannot make any progress towards the completion of execution and unnecessarily occupy the memory space. Hence, it is beneficial to make the main memory free by swapping out these processes from main memory.

* Medium term scheduler is responsible for swap out and swap in processes.

* When a suspended process or swap out process is again ready to acquire CPU, then medium term scheduler next takes this process and swap in this process into the memory and makes it ready for execution.

-x-

Q. Context switching?

Switching the CPU to another process requires saving of old process states and loading the saved states for the next process. This is known as context switching.

* Context switch of a process is represented by process control block or PCB of the process. It includes the value of CPU registers;

the process state and memory management information.

*- Context switch time is purely an overhead because CPU doesn't do any useful work while switching. It highly depends on hardware configuration.

Scheduling Criteria :

(i) Processor utilisation.
(ii) Throughput.
(iii) Turn around time.
(iv) Response time.
(v) Waiting time.

4/8/2015

Co-operating Process.

*- The concurrent execution of processes may be either independent process or co-operating process. 5/8/2015

A process is independent if it cannot affect or be affected by other processes executing in the system.

A process is co-operating if it can affected

or be affected by other processes executing in the system.

* There are several reasons for which we are going for process co-operation. There are

i) information sharing.

Since same information is needed by several users, we must provide the environment to allow concurrent access.

ii) Computational speedup.

If we divide a big task into smaller parts, each will be executing in parallel with others, then speed up can be acheived.

iii) Modularity:

If we construct the system in a modular fashion, then constructive system performance can be obtained.

iv) Convinience.

A user may have many task at one time, i.e., a user may be editing, printing, compiling in parallel.

Q) Dispatcher.

Dispatcher is a module which gives the control of the CPU to the process selected by the short-term scheduler or CPU scheduler.

- The function involves
  (i) switching the context.
  (ii) switch on the user mode.
  (iii) jumping to the proper location in the user program to restart that program.

- The dispatcher should be as fast as possible so that time will not be unnecessarily wasted.

- The time taken by the dispatcher to stop one process and start another process running is known as dispatch latency.

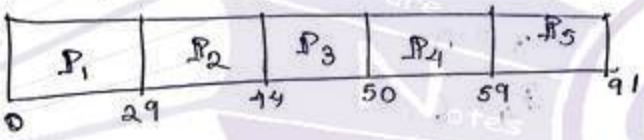
5/2/2015.

Scheduling Algorithm.

1. FCFS Algorithm (First come first serve). (non-pre-emptive)

Process	Burst time (millisec)
P_1	29
P_2	15
P_3	6
P_4	9
P_5	32

Gratt Chart



Waiting time for $P_1 = 0$

$P_2 = 29$ msec

$P_3 = 44$ msec

$P_4 = 50$ msec

$P_5 = 59$ mc.

Turn around time of

P_1 = Wt time of P_1 + execution time

= 0 + 29 = 29 msec

P_2 = 29 + 15 = 44 msec

P_3 = 44 + 6 = 50 msec.

P_4 = 50 + 9 = 59 msec.

P_5 = 59 + 32 = 91 msec.

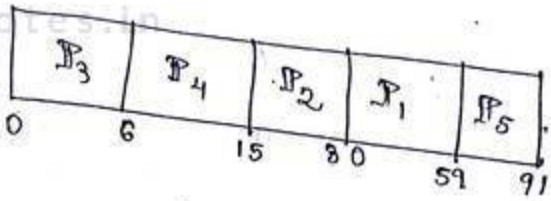
Sushri

Average waiting time

$$= \frac{0+29+44+50+59}{5} = 34.6 \text{ msec}.$$

2. Shortest Job First (SJF) . (non-pre-emptive) .

P_1	29
P_2	15
P_3	6
P_4	9
P_5	32



Waiting time of $P_1 = 30$

$P_2 = 15$

$P_3 = 0$

$P_4 = 6$

$P_5 = 59$.

Turn around time of $P_1 = 59$ msec

$P_2 = 30$ msec

$P_3 = 6$ msec

$P_4 = 15$ msec

$P_5 = 91$ msec .

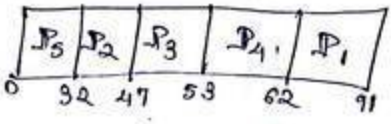
Average waiting time

$$= \frac{30+15+6+59+0}{5} = 110/5 = 22 \text{ msec}.$$

FCFS with Arrival

3. SJRF (Shortest Job Remaining First) . (Preemptive).

Process	Burst time	Arrival
P_1	29	4
P_2	15	1
P_3	6	2
P_4	9	3
P_5	32	0



Waiting time of P_1 = 62 − 4 = 58

P_2 = 32 − 1 = 31

P_3 = 47 − 2 = 45

P_4 = 53 − 3 = 50

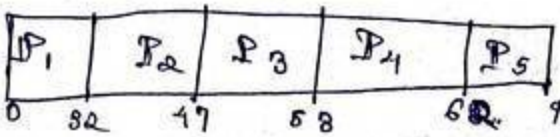
P_5 = 32 − 0 = 32

Average waiting time

$= \frac{58+31+45+50+32}{5} = 43.6$ sec.

4. Shortest Job First with same Arrival time.

P_1	29	3
P_2	15	1
P_3	6	2
P_4	9	2
P_5	32	0



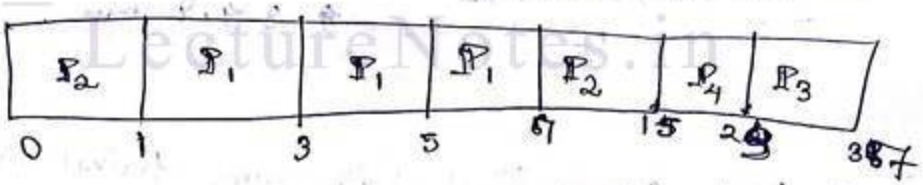
Q There are 5 process in the system i.e.,

P_1	20 msec	4
P_2	30	3
P_3	56	0
P_4	29	6
P_5	3	9

11/8/2015

5. Shortest Job Remaining First (Pre-emptive)

Process	Burst time	Arrival time
P_1	~~6~~ 7 ~~4~~ 1 ~~0~~	1
P_2	~~7~~ 2 ~~8~~ 0	0
P_3	~~1~~ 4 ~~0~~	3
P_4	~~8~~ 0	5



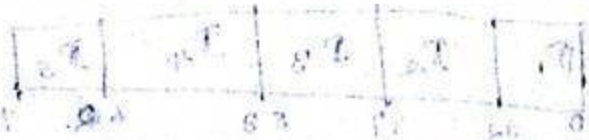
Waiting time of $P_1 = 5 - 4 = 1$

↓ at last 5 it is executed 2 times for 4 msec

$P_2 = 7 - 1 = 6$ msec.

$P_3 = 23 - 0 = 23$ msec.

$P_4 = 15 - 0 = 15$ msec.



Average waiting time $= \frac{0+14+6+23+15}{4}$

$= 45/4 = 11.25$ sec.

Turn around time of $P_1 = 1+6 = 7$

$P_2 = 6+9 = 15$

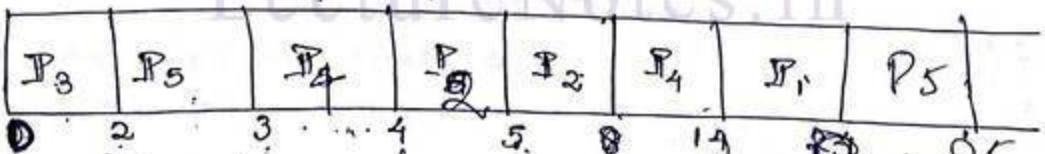
$P_3 = 23+14 = 37$

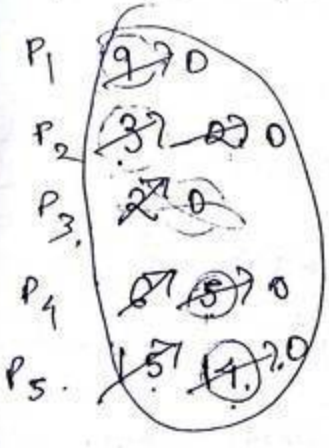
$P_4 = 15+8 = 27$.

Average turn around time $= \frac{7+15+37+27}{4} = 20.5$ sec.

Q.

Process	Burst time	Arrival time.
P_1	9	5
P_2	3 2	4
P_3	2 0	0
P_4	6 5 5	3
P_5	15 13 12	2





21
14
35

12/8/2015.

Process	Burst time	Arrival time
P_1	8	3
P_2	4	2
P_3	5	0
P_4	3	1
P_5	6	5

Ans:

FCFS (nonpreemptive)

Waiting time for $P_1 = 0$

$P_2 = 8$

$P_3 = 12$

$P_4 = 17$

$P_5 = 20$

Turn around time of:

$P_1 = 0+8 = 8$ msec

$P_2 = 8+4 = 12$ msec

$P_3 = 12+5 = 17$

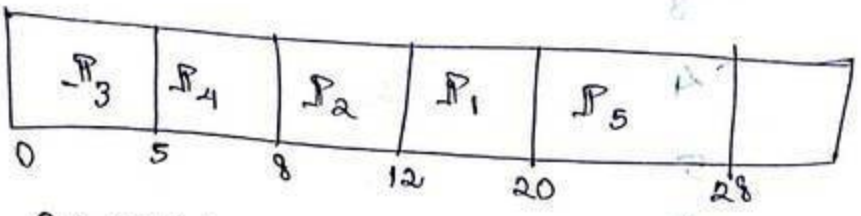
$P_4 = 17+3 = 20$

$P_5 = 20+6 = 26$

Average waiting time $= \frac{0+8+12+17+20}{5} = 57/5 = 11.4$ msec.

Average turn around time $= \frac{8+12+17+20+26}{5} = 83/5 = 16.6$ msec.

FCFS. (with)



$\frac{0+5+8+12+20}{5} = 45/5 = 9$ sec.

Turn around time of

$P_1 = 0+5 = 5$ $\quad 12+8 = 20$

$P_2 = 5+4 = 9$ $\quad 8+4 = 12$

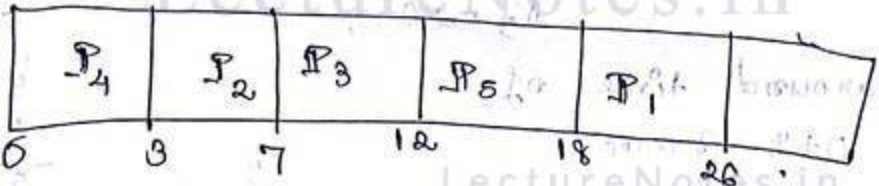
$P_3 = 0+5 = 5$

$P_4 = 5+3 = 8$.

$P_5 = 20+6 = 26$

Average $= \frac{20+12+5+8+26}{5} = 51/5 = 11.2$ sec.

SJF. (without).



$\frac{0+3+7+12+18+26}{5} = \frac{40}{5} = 8$

Turn around time of

$P_1 = 18+8 = 26$

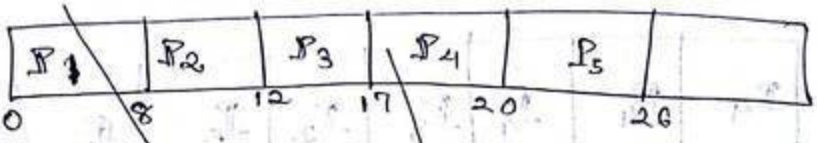
$P_2 = 3+4 = 7$

$P_3 = 7+5 = 12$

$P_4 = 0+3 = 3$

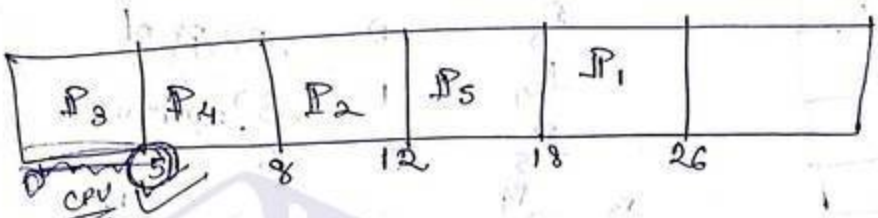
$P_5 = 12+6 = 18$ $\quad = 66/5 = 13.2$

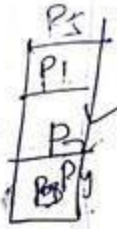
SJF (with preemption)



$\frac{0+8+12+17+20}{5} = 57/5 = 11.4$ ms.

Turn





$\frac{0+5+8+12+18}{5} = 43/5 = 8.6$.

Turn around time of

$P_1 = 18+8 = 26$

$P_2 = 8+4 = 12$

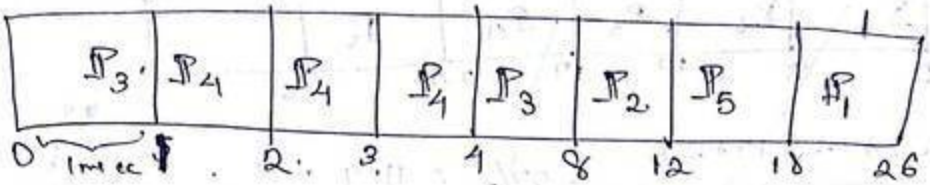
$P_3 = 0+5 = 5$

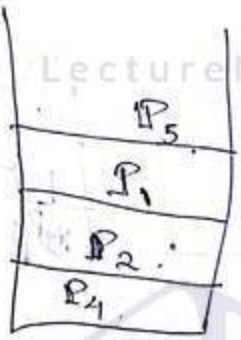
$P_4 = 5+3 = 8$

$P_5 = 12+6 = 18$

$\frac{26+12+5+8+18}{5} = 69/5 = 13.8$ ms

SJRF:





P_1	3	8
P_2	2	4 0
P_3	0	5 4 0
P_4	1	3 2 1 0
P_5	5	6

$$\frac{18 + 8 + (4-1) + (3-2) + 12}{5} = \frac{42}{5} = 8.4$$

Turn around time of

$P_1 = 18 + 8 = 26$

$P_2 = 8 + 4 = 12$

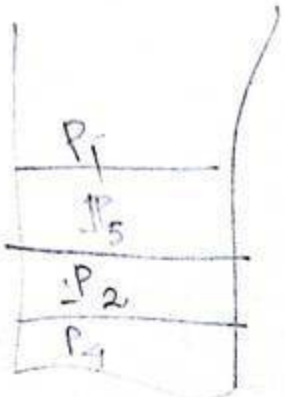
$P_3 = 3 + 5 = 8$

$P_4 = 1 + 3 = 4$

$P_5 = 12 + 6 = 18$

$= 68/5 = 13.6$ msec

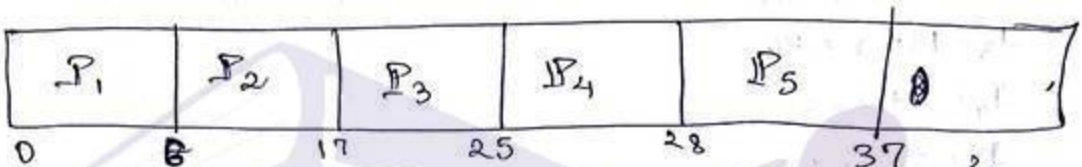
	Burst time	Arrival time
P_1	8	3
P_2	4	2
P_3	5 4	0
P_4	3 2 1 0	1
P_5	6	5



Process	Arrival	Burst time
P_1	2	6
P_2	4	11
P_3	3	8
P_4	0	3
P_5	1	9

Sol:-

FCFS (without)



$$\frac{0+6+17+25+28}{5} = 76/5 = 15.2$$

~~FCFS (with)~~

Turn around time of

$P_1 = 0+6 = 6$

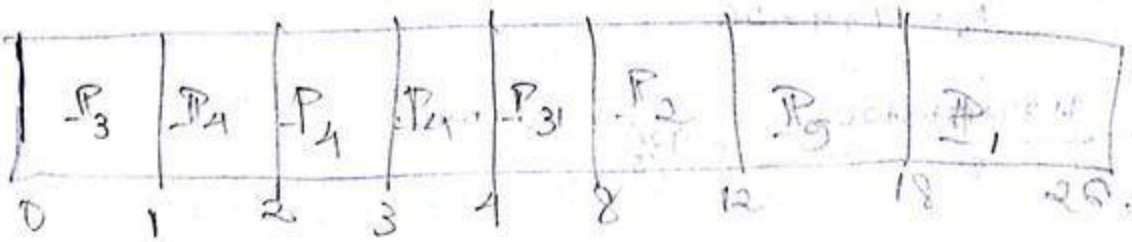
$P_2 = 6+11 = 17$

$P_3 = 17+8 = 25$

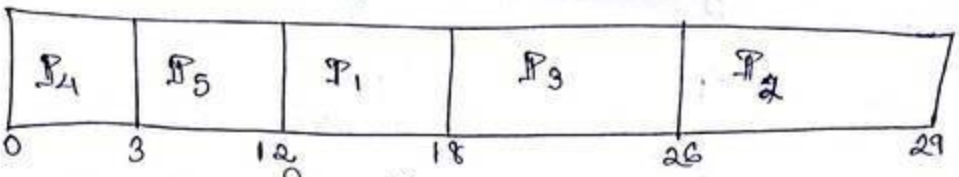
$P_4 = 25+3 = 28$

$P_5 = 28+9 = 37$

$$\frac{6+17+25+28+37}{5} = 113/5 = 22.6 \text{ ms}$$



FCFS (with)



Average waiting time

$= \frac{0+3+12+18+26}{5} = 59/5 = 11.9$ msec.

Turn around time of

$P_1 = 12+6 = 18$

$P_2 = 26+11 = 37$

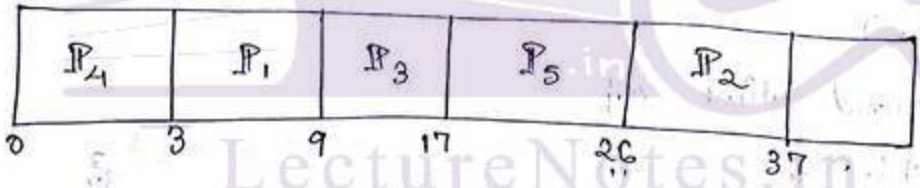
$P_3 = 18+8 = 26$

$P_4 = 0+3 = 3$

$P_5 = 3+9 = 12$

$\frac{18+37+26+3+12}{5} = 96/5 = 19.2$ msec.

SJF (without).



$\frac{0+3+9+17+26}{5} = 55/5 = 11$ msec.

Turn around time of

$P_1 = 3+6 = 9$

$P_2 = 26+11 = 37$

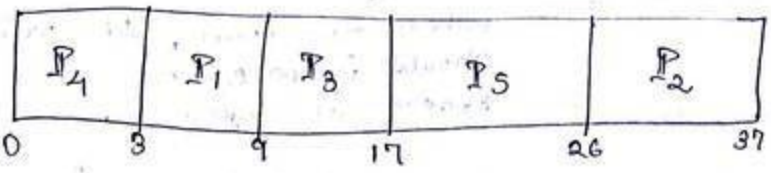
$P_3 = 9+8 = 17$

$P_4 = 0+3 = 3$

$P_5 = 17+9 = 26$

$\frac{9+37+17+3+26}{5} = 92/5 = 18.4$ msec.

SJF (with).



$\frac{0+3+9+17+26}{5} = 55/5 = 11$ msec.

Turn around time of

$P_4 = 0+3 = 3$

$P_1 = 3+6 = 9$

$P_3 = 9+8 = 17$

$P_5 = 17+9 = 26$

$P_2 = 26+11 = 37$.

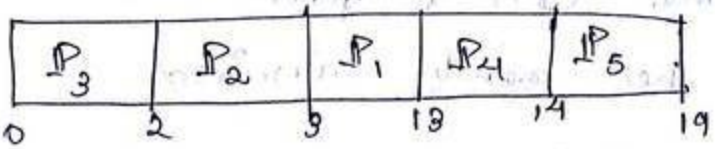
$\frac{3+9+17+26+37}{5} = 92/5 = 18.4$

SJRF.

17/8/2015

Priority Scheduling (both).

Process	Burst time	Priority
P_1	10	3
P_2	1	2
P_3	2	1
P_4	1	4
P_5	5	5



Average waiting time = 6.4 msec.

Round Robin Scheduling

Demerit of priority scheduling: indefinite blocking.

↓

when new comes, old priority should increase.

known as aging.

With this scheduling, a priority is associated with each process and CPU is allocated to the process with highest priority.

*- Equal priority processes are scheduled in FCFS manner.

*- Priorities can be either internal or external.

*- When a process arrive at ready queue, its priority is compared with priority of currently running process.

*- If the priority of currently running process is lower than the priority of newly arrived process, then priority scheduling then, pre-emptive (forcibly) the CPU & take out the CPU & give the control to the newly arrived process. The major problem with this

algorithm is indefinite blocking or starvation.

When high priority process enters the ready queue will be given the chance to be executed in CPU.

* In the process, the low priority process never get a chance to be executed, or wait for a long time to get CPU for execution. This is known as starvation.

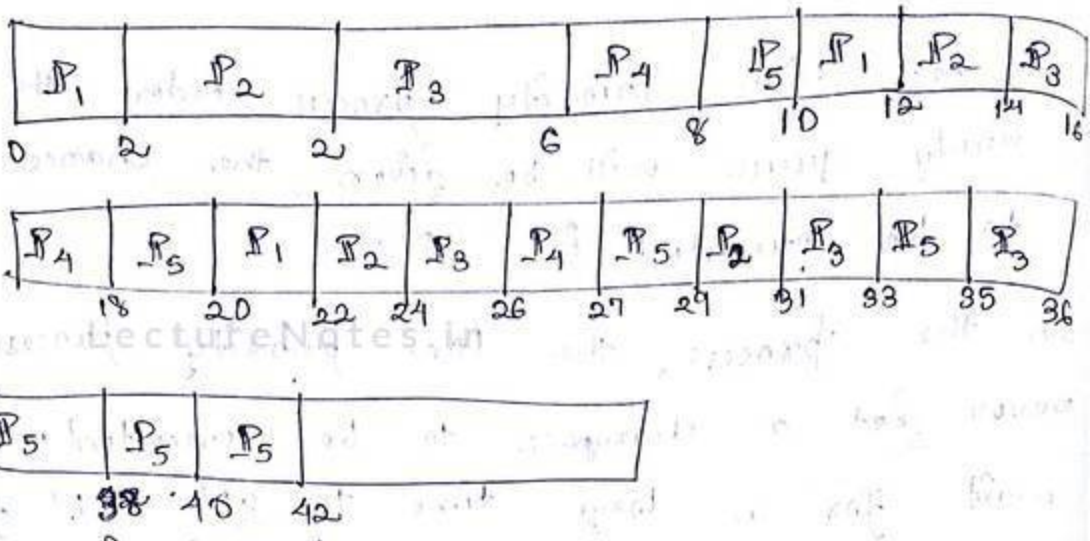
The solution to this problem of indefinite blocking of lower priority processes is aging.

Aging is a technique of gradually increasing the priority of process that wait in the system for a long time.

Round Robin Scheduling (pre-emptive).

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3
P_5	4

Time slice = 2ms



Waiting time of

$P_1 = 20 - 4 = 16$

$P_2 = 29 - 6 = 23$

$P_3 = 35 - 8 = 27$

$P_4 = 26 - 4 = 22$

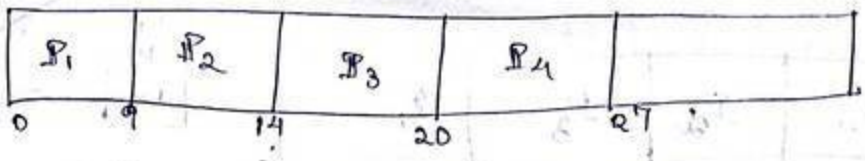
$P_5 = 40 - 12 = 28$

Average $= \frac{16 + 23 + 27 + 22 + 28}{5} = 23.2$

	Burst time	Arrival time	Priority
P_1	~~4 2 6 3~~	4	
P_2	~~5 3 2 0~~	3	
P_3	~~6 5 3 0~~	1	
P_4	~~7 7 4 1~~	0	

Apply FCFS, SJF, SJRF, Round robin to find the av. waiting time & av. turn around time where time slice = 3ms

FCFS (without).



Average waiting time = $\frac{0+9+14+20}{4} = 43/4 = 1.75$.

Turn around time of

$P_1 = 0+9 = 9$

$P_2 = 9+5 = 14$

$P_3 = 14+6 = 20$

$P_4 = 20+7 = 27$

$\frac{9+14+20+27}{4} = 70/4 = 17.5$.

FCFS (with)

Average waiting time = $\frac{0+7+13+18}{4} = 38/4 = 9.5$

Turn around time of

$P_1 = 18+9 = 27$

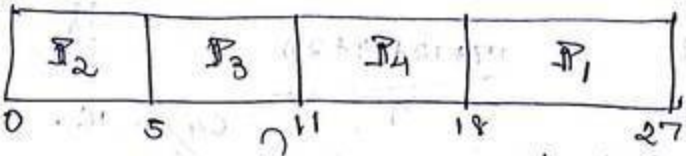
$P_2 = 13+5 = 18$

$P_3 = 7+6 = 13$

$P_4 = 0+7 = 7$

$\frac{27+18+13+7}{4} = 65/4 = 16.25$

SJF (without).



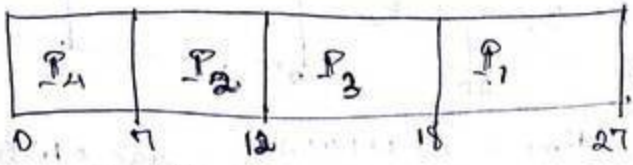
Average waiting time = $\frac{0+5+11+18}{4} = 34/4 = 8.5$

Turn around time of $P_1 = 18+9 = 27$ $P_2 = 0+5 = 5$

$P_3 = 5+6 = 11$ $P_4 = 11+7 = 18$

$\frac{27+5+11+18}{4} = 61/4 = 15.25$.

SJF (with)



Average wait time =

$$= \frac{0+7+12+18}{4} = 37/4 = 9.25$$

Turn around time of

$P_1 = 18+9 = 27$

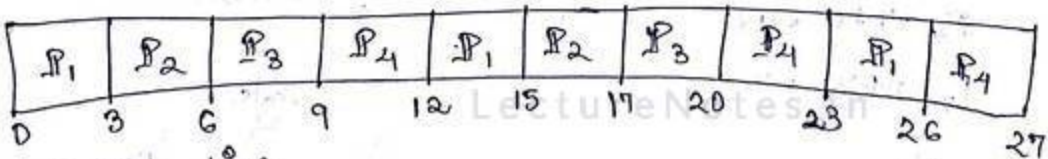
$P_2 = 13+5 = 18$

$P_3 = 7+6 = 13$

$P_4 = 0+7 = 7$

$$= \frac{27+18+13+7}{4} = 65/4 = 16.25$$

Robin round:



Waiting time

of $P_1 = 23-6 = 17$

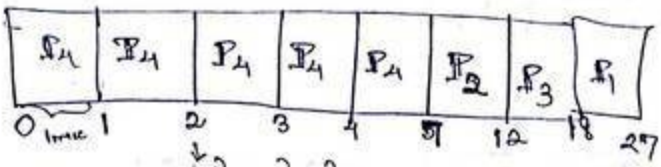
$P_2 = 15-3 = 12$

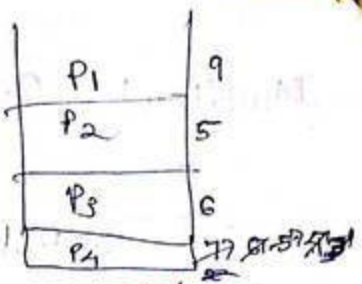
$P_3 = 17-3 = 14$

$P_4 = 26-6 = 20$

$$\frac{17+12+14+20}{4} = 64/4 = 16$$

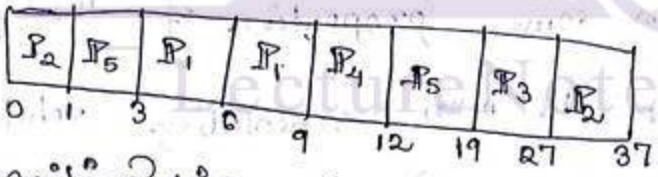
SJRF

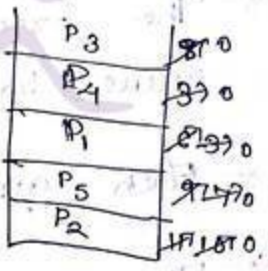




18/8/2015.

Process	Burst time	Arrival time
P_1	6	3
P_2	11	0
P_3	8	9
P_4	3	6
P_5	9	1





Waiting time of

$P_1 = 6 - 3 = 3$ ms.

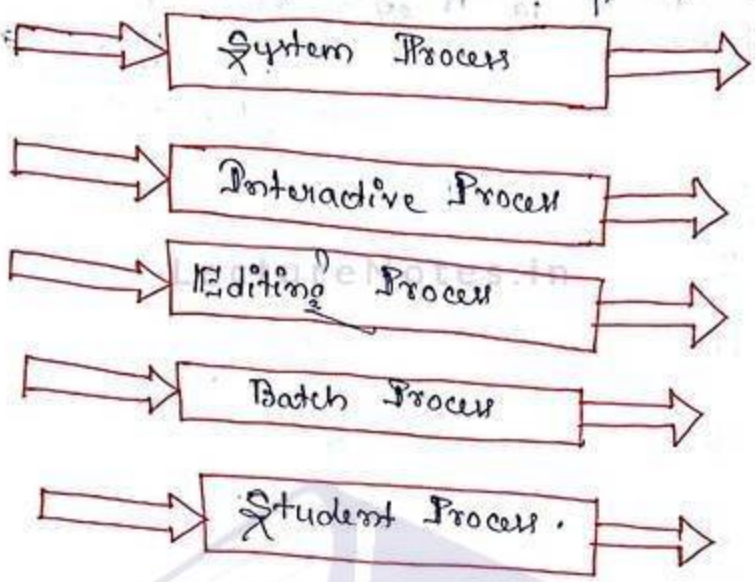
$P_2 = 27 - 1 = 26$

$P_3 = 19 - 0 = 19$

$P_4 = 9 - 0 = 9$

$P_5 = 12 - 2 = 10$.

Multilevel Queue Scheduling



This algorithm partitions the ready queue into several separate queues.

* The processes are permanently assigned to one queue based on some properties of the queue.
* Each queue has its own scheduling scheme.
* There must be scheduling among the queues also.
* Each queue has absolute priority over low priority queues, i.e., no process in the batch queue could execute until and unless all the processes in system process queue, interactive process queue,

editing? process queue has completed their execution.

19/8/2015.

Multilevel Feedback Queue Scheduling.

*- With multilevel queue scheduling algorithm, processes are permanently assigned to a queue on entry to the system.

*- Processes do not move between queues.

*- Multilevel feedback queue scheduling allows a process to move between queues.

*- If a process uses too much of CPU time, it will be moved to a low priority queue.

*- Similarly, a process that waits too long in a low priority queue may be moved to a higher priority queue.

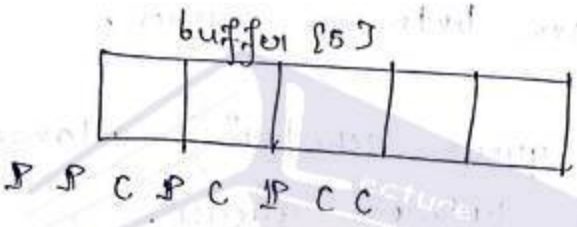
Synchronisation.

Producer process.

```
while (1)
{
    while (counter == size)
        ;
    buffer[in] = item;
    in = (in+1) % size;
    counter++;
}
```

```
While (1)
{
    While (counter != 0)
    {
        int temp;
        buff
        temp = buffer[out];
        out = out + 1;
        counter --;
    }
}
```

consumer process.



When several processes tries to access or manipulate the same data concurrently or simultaneously, the outcome of the execution depends on the order of execution in which the access takes place.

This problem is known as race condition.

*- In producer consumer problem, the producer is trying to produce an item that is stored in a buffer and customer will consume an item from

the buffer.

* The producer and consumer uses shares a common variable counter.

* Both consumer and producer process if executed separately, then outcome will not differ. But, if producer consumer problem will execute concurrently and tries to manipulate counter simultaneously, then outcome of the execution will depend on the order of execution of instructions present in both programs.

20/8/2015.

* The race condition can be solved using synchronisation tool.

* The OS has to define a critical section for each process i.e., executing simultaneously.

Critical Section Problem.

The critical section is a segment code for each process in a set of processes, i.e., from P_0 to $P_{(n-1)}$ in which the process may change its variables may change the variables, updates a table, changes a needs.

common variable, writes into a file and so on.

It is a sequence of instructions with clearly marked beginning and end.

*- The critical section problem is to design a protocol that the processes can use to co-operate.

*- The important feature of critical section is that when a process enters a critical section, it must complete all instructions there in. before any other process is allowed to enter into their critical section. This property is known as mutual exclusion.

*- The solutions to the critical section problem ~~area~~ must satisfy these three requirements:

i) mutual exclusion
ii) progress requirement.
iii) bounded waiting.

24/8/2015.

i) Mutual Exclusion.

*- If one process is executing in its critical section, then no other processes are allowed to enter into their critical section.

(ii) Progress Requirement:

* When one process wants to enter into the critical section, the permission is granted only if no process is executing in their critical section and they should not be in the remainder section also.

(iii) Bounded Waiting:

* There should be a bound on the number of times that the process are allowed to be executed.
* When more than one process wants to enter the critical section, the grant for the request should be in finite time.

Two Process Solution:

P_i:

```
while (turn == j) do no-operation;
        [ critical section ]
turn = Pj ;
```

(mutual exclusion)

P_j:

```
while (turn == i) do no-operation;
        [ Critical section ]
turn = Pi ;
```

25/8/2015.

Peterson's Solution to Critical Section.

Algorithm:

Pi:
do
{
flag[i] = true;
turn = j;
while(flag[j] && turn = j);
critical section
flag[i] = false;
remainder section;
} while(1);

(for progress)

Pj:
do
{
flag[j] = true;
turn = i;
while(flag[i] && turn = i);
critical section
flag[j] = false;
remainder section;
} while(1);

* The Petersons

Two Process Solution:

* With this algorithm, only two processes can participate in the decision of which will enter the critical section first. These two processes share a common integer variable turn.
* If turn = i, then process (P_i) is allowed to enter into the critical section.
* If turn = j, then process (P_j) is allowed to enter into the critical section.

This algorithm satisfies mutual exclusion, i.e., only one process at a time can be in its critical section. It doesn't satisfy progress requirement. As the process is not ready while its turn comes.

Peterson's Solution:

* Peterson's solution is restricted to 26/2/2015. two processes that alternate the execution between their critical sections and remainder sections.
* The two processes are P_i and P_j. It requires the two processes to share two

data items : flag[2] and turn.

*- The variable turn indicates that whose turn is to enter into the critical section. i.e.,

if (turn == i)

then process (P_i) is allowed to enter into the critical section.

*- The flag is used to indicate that, if a process is ready to enter into the critical section or not.

if (flag[i] == true)

then process (P_i) is ready to enter into the critical section.

*- To enter into the critical section, P_i makes flag[i] is true and then, turn = j (makes the turn). Thereby, entering into the critical section.

*- If turn = j, then P_j checks that whether P_i is ready to enter into the critical section or not. If yes, then P_j will allow P_i to enter into the critical section. In this way, mutual exclusion is preserved, progress requirement is satisfied and bounded waiting can be met.

20/2/2015

Thread:

Thread is a light weight process.

It is a basic unit of CPU utilization.

It consists of thread ID, program counter, register set and stack. It shares with other threads belonging to the same process, its code section, data section and operating system resources.

* A traditional process is known as heavy weight process having a single thread of control.

Difference between Thread And Process.

Similarity

1. Thread can also Process have almost same status as processes.
2. They also share CPU.
3. Each thread have its own stack and program counter.
4. Threads can create child threads.
5. If one thread is block, the other thread can run.

6. A thread executes sequentially.

Dissimilarities:

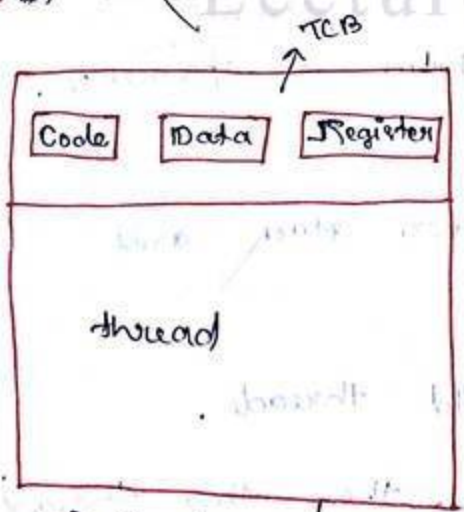
1. A process can be independent to each other but threads are not independent to each other.

2. A thread can read or write over any other thread stack, but process cannot.

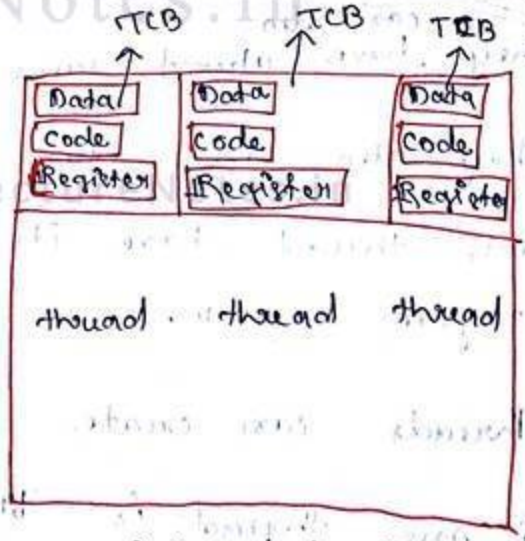
3. It takes less time to create a new thread in an existing process than to create a new process.

4. It takes less time to terminate a thread than a process.

5. It takes less time to switch between threads



Single thread context block.



Multi-threaded block

Advantages of Multi-Thread System.

1. Responsiveness.

Multi-threading may allow a program to continue running even if part of its code is blocked or is performing a lengthy operation. Thereby, increasing responsiveness to the user.

2. Resource sharing.

Process may only share resources through techniques such as message passing or shared memory. But, Threads by default shares the memory or resources of the process to which they belong.

3. Economical.

Allocating memory and resources for process creation is costly. On the other hand, because threads share the resources of the process to which they belong to, it is economical to create thread than process.

27/8/2015.

4. Utilisation of multi-processor architecture.

The benefits of multi-threading can be increased in a multi-processor architecture where each thread may be running parallel on

a different processor. It increases the CPU utilisation.

Types of Threads.

User-level Thread and Kernel Level Thread.

* User level threads are supported above the kernel and implemented by thread library.
* The library provides support for thread creation, scheduling and management with no support from kernel. Because the kernel is unware of user level threads, all thread created are scheduled in user space without kernel intervention.
* Kernel threads are supported by operating system directly.
* Kernel performs thread creation, scheduling and management in the kernel space.

Advantage of User-level Thread.

1. Fast switching among the threads are possible.
2. Thread scheduling can be application specific.

Disadvantage of User-level Thread:

1. When an user level thread executes a system call, not only that thread is blocked, but also all other threads are blocked. This is because OS is unaware of presence of threads, but only knows the existence of the process.

2. Multi-threaded application cannot take the advantage of multi-processing.

Advantage of Kernel-level Thread:

1. The OS is aware of presence of threads. Therefore, even if one thread gets blocked, the OS selects the next thread to execute either from the same or different process.

2. The OS can schedule multiple threads from the same processor.

Disadvantage of Kernel-level Thread:

1. Switching between threads is time consuming.

Multi-level Threading Model:

(i) many to one
(ii) one to one
(iii) many to many.

27/8/2015.

(ii) Many to one Model.

*- This model maps many user level threads to one kernel level thread.

*- Thread management is done in user space. So, it is efficient, but the entire process will block if the thread makes a blocking system call. Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel in multi-processor system.

(iii) One to One Model.

*- One to one model maps each user thread to a kernel thread. It provides more concurrency than many to one. It allows multiple threads to run parallel on multi-processors.

*- The only drawback is it burdens on the performance of the kernel.

(iv) M

*- Many to many model combines many user level thread to smaller or equal number

of kernel threads.

- The many to many model suffers neither of the demerits of many to one or one to one model. So, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

31/8/2015

Semaphore (Solution of critical section for multiple process)

→ Semaphore is an integer variable, apart from its initialisation is accessed through two atomic operations : wait(p) and signal(v).

Semaphore (p)

It is an atomic operation that waits for the semaphore to become positive, then decrements it by 1.

Semaphore (v)

It is an atomic operation that increments semaphore by 1.
It is also known as signal operation.

Characteristics of Semaphore.

1. Semaphores are like integers except they have no negative values.
2. Only operations possible are P and V.
3. The operations must be atomic. i.e., the instructions in each of the operation should be executed in a single execution cycle.

Signal(S) : S++;

Wait(S) : while(S <= 0); do no operation.
S--;

1/9/2015.

Mutual Exclusion on Semaphore.

```
do
{
    wait(mutex);
    - - - -
    - - - -
    Critical section
      code
    signal(mutex);
    remainder section;
} while(1);
```

mutex=1
when mutex=1, critical section is free.

Area where common variable are used : critical section.

When one process is executing, other for mutex to be 1. This is known as busy waiting or spin lock.

* When semaphores are used for mutual exclusion, mutex is initialised to 1. i.e., critical section is free.

> Wait (p) operation is applied on mutex variable by a process before it enters into the critical section.

* After the execution in the critical section, it with the process will call signal operation on mutex to make it 0. after exit

* Mutex = 1 implies critical section is free.

* Mutex = 0 implies critical section is busy.

* Suppose two processes A and B are there to execute. Now, A is executing in the critical section implies that mutex = 0. If now, B tries to enter into the critical section it cannot enter because it will have to wait till the semaphore variable mutex. This is possible only when process (A)

executes signal operation after executing in its critical section.

- The main disadvantage of this is busy waiting.

While a process is in its critical section, any other process that tries to enter into the critical section must continously look into the entry section, till mutex = 1. This leads to the wastage of CPU cycle, that might be used productively for other purpose. This semaphore variables are known as spin lock.

3/9/2015

- To overcome the problem of busy waiting, the semaphore is redefined.
- If a process executes a wait operation and finds that the value of the semaphore is currently zero, then instead of busy waiting, it can be forced to change its state as blocked state. By change in its state,

it is automatically appended to a waiting queue. This blocked process should be restarted when another process executes a signal operation by making mutex = 1.

* Wakeup() system call is used to change the state of the process from blocked to ready.

Redefination.

```
Signal(s) : s = s+1;
            if s <= 0
            then
                add this process to waiting queue;
                block;
            end.

Wait(s) : s = s-1;
          if s > 0
          then
              remove the process from waiting queue;
              wakeup(s);
          end.
```

Semaphores
- Binary
- counting.

Counting Semaphore.

Counting semaphores are free from inherent limitations of binary semaphore.

7/9/2015.

* A counting semaphore comprises of

(i) an integer variable initialized to K, where $K \geq 0$. 7/9/2015.

During operation, it can assume any value $\leq K$.

(ii) a pointer to a process queue.

The queue will hold the PCB's of all those processes that are waiting to enter their critical section.

(iii) this queue is implemented as a FCFS queue, so that waiting processes are served in FCFS.

Classical Problems in Synchronisation.

1. Bounded buffer problem.
2. Reader writer's problem.
3. Dinning Philosopher's problem.

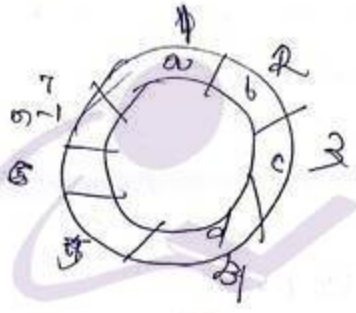
variables: mutex, full, empty

Bounded Buffer Problem. (Producer - consumer problem).

problem: item produced whenever should be stored in buffer.

Producer.

```
do
{
    produce an item;
    ...
    ...
    wait(empty);   // 0
    wait(mutex);   // 0
    ...
    ...
    add item to the buffer
    ...
    ...
    signal(mutex);   // 1
    signal(full);    // 1
    while(1);
}
```



Consumer.

decrease full as it will consume.

problem: we consume an element from buffer.

```
do
{
    wait(full);    // 0
    wait(mutex);   // 0
    ...
    ...
    remove an item from the buffer
    ...
    ...
```

```
signal(mutex);          //1
signal(empty);          //7.
...
consume an item
...
} while(1);
```

At 1st, assume

mutex = 1 ⟶ mutex free.

full = 0 ⟶ no elements

empty = n-1 ⟶ all location free

Problem : Producer & consumer when executed concurrently, then they use a shared variable for which the result will be wrong. If we will change the order of execution, then result will be different.

* We can use the semaphore to solve the bounded buffer problem.

* Assume the buffer has the capacity = n.

* Synchronization is required when producer tries to produce an item in the buffer, when buffer is full, and consumer tries to consume an item from the buffer when the buffer is empty.

Here, the solution to the synchronization can be achieved by using three semaphore variables : empty, full and mutex.

*- Initially, full = 0, empty = n and mutex = 1.

*- Mutex semaphore provides mutual exclusion for the accesses to the buffer.

*- The empty and full semaphores counts the number of full and empty buffers.

*- empty = n means initially all the buffer is empty.

*- full = 0 means the total no. of items kept in the buffer which is initially null.

8/9/2015.

Reader writer's Problem.

mutex=1 → used for readers. (for mutual exclusion)
wrt = 1 → critical section is free, (writers)
readcount=0 → count the no. of readers who were reading.

Writer.

```
do
{
    wait(wrt);      // initially 1, writer makes it 0.
    ...             other writers have to check
    ...             wrt to be 1 to write.
    writing is performed
    ...
    ...
    signal(wrt);
} while(1);
```

when one writer is writing, others are not allowed.

Reader.

```
do
{
    wait(mutex);    // 1 → later 0.
    readcount++;    // 0 → 1.
    if(readcount==1)
        wait(wrt);
    signal(mutex);  // 1.
    ...
    ...
    reading is performed;
    ...
    wait(mutex);    // 0
    readcount--;
    if(readcount==0)
        signal(wrt);
    signal(mutex);
} while(1);
```

while one is reading, others are allowed to read, but writer will be allowed after all the readers are over.

problem arises when multiple readers are there & one writer so, mutex is used.

Reader - writer's problem is when a database is to be shared among several processes concurrently.

* Some processes wants to read the database and some processes wants to write on the database. Those processes which are reading are known as reader's and those who are writing are known as writers.

* If two readers access the share data simultaneously, no problem occurs.

* If a writer and some other reader or writer procer access the database simultaneously, then problem occurs. To solve this synchronization problem is known as reader-writer's problem.

To solve this problem, we require that writers must have the exclusive access to the shared database while writing to the database.

- Here, we are using two semaphore variables mutex and wrt and integer variable readcount.

* Initially, mutex and wrt = 1 and readcount = 0.

* Now, the solution involves no reader should wait until a writer has got the permission to use the shared object / database. i.e., no reader should wait for other readers. So, wrt is used for mutual exclusion for writers.

* The mutex semaphore is used to ensure the mutual exclusion when multiple readers reads the database.

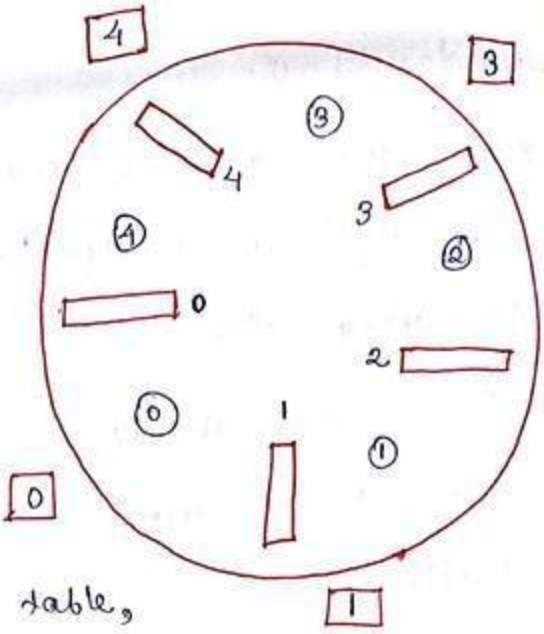
* The variable readcount is used to keep track of how many number of readers are reading the database.

After taking semaphores, if we changing the order of wait & signal → created problem.

any public/pvt/board/uni exam in India or in any foreign C. recognized as equi. to the 10+2 exam, viz graduate or 2 yr five-year post

10/9/2015.

Dinning Philosopher's Problem



*- Problem statement is :
"There are five philosophers sitting around a dinning table.

*- There are five plates placed on the table, each plate infront of each philosopher.

*- There are five chop-sticks placed between the philosophers.

*- Whenever a philosopher feels hungry, he will attempt to pick two chop-sticks which are shared with his neighbours.

*- If any of his neighbours happens to be eating, at that time the philosopher has to wait.

*- When the hungry philosopher is able to get two chop-sticks, he will take his food.

*- After he finishes eating, he places the chop-sticks back on the table and

starts to think. Now, this chop-sticks are available for his neighbours.

```
chopstick[5];
do
{
    wait(chopstick[i] %5);
    wait(chopstick[i+1] %5);
        eating is performed;
    signal(chopstick[i] %5);
    signal(chopstick[i+1]%5);
} while(1);
```

as it is around one so, after 4, 0 come so %5.

* The solution to the dinning philosopher problem is to represent each chopstick by a semaphore (chopstick[5]) where all the elements of chop-stick are initialised to 1 i.e; all chop-sticks are free.

* When a philosopher feels hungry, he will grab the chopstick by executing a wait operation on that semaphore, the releases the chopsticks by executing signal operation on that semaphore.

* The limations are :

(i) it is posible that all the philosophers may feel hungry almost at the same time and all will pick up the

chop-sticks at the same time. So, now all will keep waiting for their respective right chopsticks forever. It is a deadlock condition and progress is not satisfied.

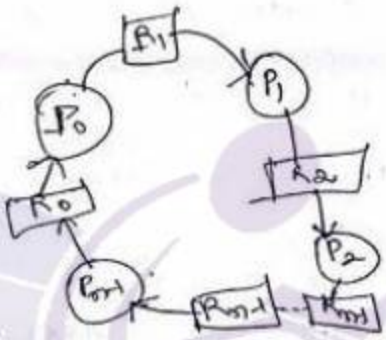
14/9/2015.

Deadlock

Characteristics:

i) mutual exclusion
ii) hold and wait.
iii) no pre-emption
iv) circular wait.

Resources:
request
release
wait.



* In multi-programming environment, several processes may compete for a finite number of resources.

* A process requests resources, if the resources are not available at that time the process enters to the waiting state.

* A waiting process never change its states because the resources it has requested are held by other waiting processes. This situation is known as a deadlock.

* A system may consist of finite no. of resources to be distributed among a number of competing processes.

* The resources are normally either physical like memory, printer, keyboard or logical like CPU cycles, clock speed, time slice.

* Under a normal mode of operation, a process may utilize a resource only in the following sequence:

(i) Request:
The process request the resources. If the resources cannot be granted immediately, then the requesting process must wait until it can acquire the resource.

(ii) Use

The process can operate on the resource.

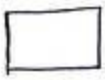
(iii) Release

The process after completion of its execution release the resources.

* A set of processes is in deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

15/9/2015.

Resource Allocation Graph

 → Resource

 → Process

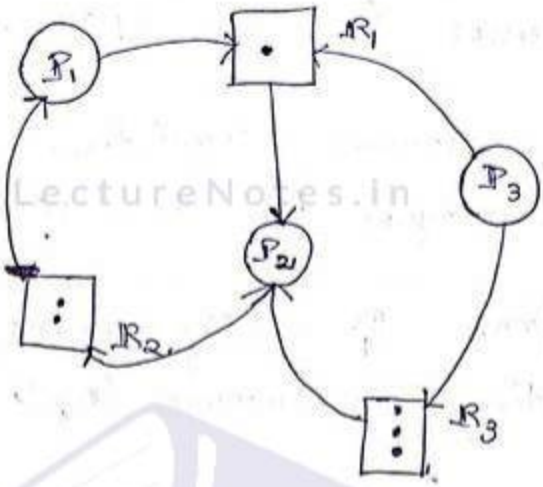
• → instance of resource (number of resources).

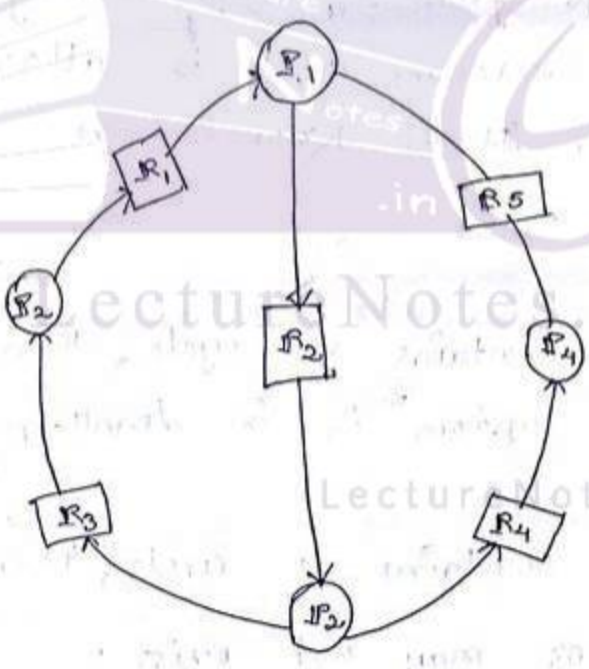
Vertices are process or resources.

edges are from process to resource or resource to process.

$R_i \longrightarrow P_i$ (allocated edge) or (assignment edge)

$P_i \longrightarrow R_i$ (request edge)





* Deadlocks can be represented by directed graph known as system resource allocation graph. This graph consists of set of vertices and set of edges.

- The set of vertices are partitioned into 2 types :
  - (i) nodes (processes) consisting of all active processes in the system.
  - (ii) nodes of resources consisting of all resource types in the system.

* A directed edge from $P_i \rightarrow R_j$ means process (P_i) is requesting a resource (R_j).

* A directed edge from $R_i \rightarrow P_j$ signifies that an instance of resource (R_i) is allocated to process (P_j). It is known as assignment edge.

* If the graph contains no cycle, then no process in the system is in deadlock.

- If the graph contains a cycle, then a deadlock may or may not exist.
- If each resource type has single instance, then cycle implies that a deadlock has occurred.

* If each resource type has multiple instance then a cycle in the graph is both necessary, but not the sufficient condition for the existence of a deadlock.

16/9/2015.

Methods of Handling Deadlock.

(i) Deadlock prevention.
(ii) Deadlock avoidance
(iii) Deadlock detection.
(iv) Recovery.

* When deadlock has occurred, we can avoid the deadlock in one of these 3 ways:

(i) We can use a protocal to prevent, or avoid deadlocks by ensuring that system will never enter a deadlock state.

(ii) We can allow the system to enter into deadlock state, detect it and recover from it.

(iii) We can ignore the problem all th together.

Deadlock Prevention.

- Deadlock prevention ensures that atleast one of the 4 necessary conditions should not occur.

(i) Mutual exclusion.

Mutual exclusion holds for non-sharable resources. If we use the sharable resources, mutual exclusion condition doesn't hold and the process cannot be involved in a deadlock because a process never waits for a sharable resource. Read only files are the example of sharable resource, whereas printer is a non-sharable resource.

(ii) Hold and wait.

To ensure that hold and wait cond^n doesn't hold in the system, we must ensure that whenever a process requests for a resource, it doesn't hold any oth resources. The protocol used for this is t each process to request and be

allocated all the resources before it begins its execution. Another protocol used here is the process requests resources only when it has no resource. The demerit of the system is

(i) resource utilization is very low
(ii) starvation may occur.

(iii) No pre-emption.

To ensure that no pre-emption condition doesn't hold, we can use the following protocol:

if a process is holding some resource and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are pre-empted that is resources are implicitly released. The process will restard only when it can regain its old resources as well as the new resources that it is requesting.

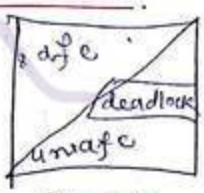
Another protocol is if a process requests some resources, we first check whether they are available or not. If they are available allocate them. If not, we check that they are allocated to some other process that is waiting for additional resource. If so, we from pre-empt the desired resources from waiting process and allocate them to requesting process.

Circular wait.

21/9/2015.

Unsafe → deadlock. (cycle).

Safe → may or may not be deadlock.



(ii) Deadlock Avoidance. (Banker's Algorithm).

Allocation			Max			Available		
A	B	C						

(iii) Circular wait.

* The fourth and final condition of deadlocks is the circular wait.
* One way to ensure that this condition never holds is to impose a total ordering of

all resource types and to require that each process requests a resource in the increasing order of enumeration.

Ex: If there are three resources in the system such as tape drive, disk drive and printer. We assign a unique integer to each resource type which allows us to compare two resources and determine whether one precedes another in our ordering.

Each process can request a resource only in an increasing order of enumeration.

(Array (available), 2 matrices)

iii) Deadlock Avoidance · (Banker's Algorithm).

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

1st P_1 executes · (given 200 + 122 = 322).

```
  3 3 2
- 1 2 2
-------
  2 1 0
+ 3 3 2
-------
  5 3 2
```

Then P_3 ·

```
  5 3 2
- 0 1 1
-------
  5 2 1
```

release (222) + 222

```
  7 4 3
```

P_4

```
  7 4 3
- 4 3 1
-------
  3 1 2
+ 4 3 3
-------
  7 4 5
```

P_0

```
  - 7 4 3
  -------
    0 0 2
+ 7 5 3
-------
  7 5 5
```

·

P_2 755
−600
155
+902
1057

	Allocation	Max	Available	Need
P_0	001	001	152	000
P_1	100	175		075
P_2	135	235		100
P_3	063	065		002
P_4	001	065		064

P_0 152
+001
153

P_2 153
−100
053
+235
288

P_3 −002
286
+065
2911

P_4 ~~3141~~
~~−0064~~

2141
064
287
+065
21412

P_1

2 14 12
− 0 7 5
2 7 7
+ 1 7 5
3 14 12

* Deadlock avoidance algorithm for avoiding deadlocks is to require additional information about how the resources are requested, with this additional information of complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait.

* Deadlock avoidance algorithm dynamically examine the resource allocation graph to ensure that a circular wait can never exist.

* A state is a safe state if the system can allocate resources to each process in some order and avoid a deadlock or that is a system is in safe state only if there exist a safe sequence.

* A sequence of processes $P_0, P_1, \ldots, P_n$ is known as safe sequence if each process (P_i), the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all other processes.

* If needed resources of P_i are not immediately available, then P_i can wait until P_j have finished. When they have finished, P_i can obtained its needed resources, complete its task and return it to OS. and terminate.

* When P_i terminates, P_{i+1} can obtained its resources and complete its execution. If no such sequence exists, the system state is said to be unsafe.

* A safe state is not a deadlock state.

* A deadlock is an unsafe state, but not all the unsafe states are deadlocks.

* An unsafe state may lead to a deadlock.

Bankers Algorithm.

i) Safety algorithm

ii) Resource request algorithm.

Resources :

i) available [m]

ii) allocation [m][n]

iii) max [m][n].

iv) need [m][n]

(e) Safety Algorithm.

1. Let Available and Finish are vectors of length m and n.

   Initially Finish[i] = false for i = 1, 2, ..., n.

2. Find i such that

   (a) Finish[i] = false

   (b) Needi ≤ Available

   If no such i exists, then goto step-4

3. Available = Available + allocation

   Finish[i] = true

   goto step-2

4. If Finish[i] = true for all then, system is in safe state.

	Allocation	Max	Available	Need
P_0	1 0 1	3 0 2	0 1 1	2 0 1
P_1	1 9 3	2 9 5		~~1 4 2~~ 1 0 2
P_2	2 2 1	5 4 3		3 2 2
P_3	2 0 0	2 1 1		0 1 1

P_3 Available 0 1 1
+ 2 0 0
2 1 1

P_0 Available = 211 + 101

= 3 1 2 .

P_1 Available = 312 +193

= 4,19,5

P_2 Available = 4 10 5
+ 2 2 1
6 12 6

The system is in safe state.

Em	Allocation	Max	Available	Need
P_0	0002	0012	1520	0010
P_1	1000	1750		0750
P_2	1354	2356		1002
P_3	0632	0652		0020
P_4	0014	0656		0642

P_0 Available = 1520
+ 0002
1522

P_3 Available = 1522 - P_2 1522
0632 1354
1154 2876

P_3 2876
0632
3 11 12 10
2 1 10 8
P_4 3 11 12 10
0 0 8 4
3 17 15 12
3 11 13 14

P_1 3 17 15 12
0 0 0 0
3 17 15 12

P_0 P_2 P_3 P_4 P_1

28/9/2015.

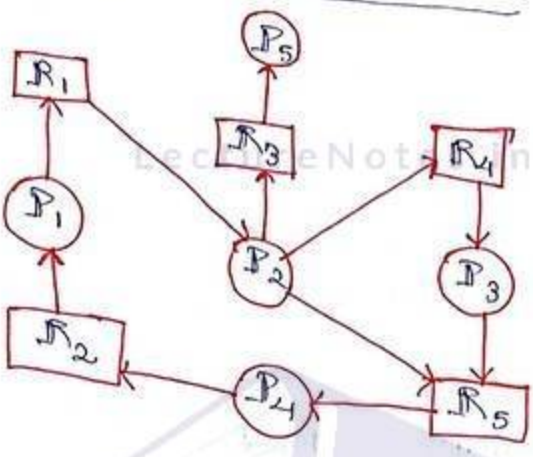
(iii) Deadlock Detection Algorithm.

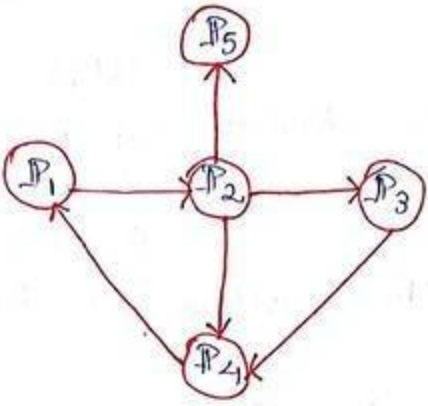
(for deadlock avoidance)

- Resource Allocation Graph.

(for deadlock detection)

Wait for Graph





* If a system doesn't employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this situation, the system may provide

(i) an algorithm that examines the state of the system to determine whether a deadlock has occurred.

(ii) an algorithm to recover from the deadlock.

* Deadlock detection algorithm exists for either single instance of resource type or multiple instance of resource type.

Single Instance of Resource Type:

When all resources have only a single instance, deadlock detection algorithm employees a wait for graph which is drawn from resource allocation graph.

* Wait for graph can be obtained by removing the nodes of resource and colapsing the appropriate edge.
* An edge $P_i \rightarrow P_j$ in a wait for graph means process (P_i) is waiting for process (P_j) to release a resource that P_i needs.
* An edge P_i to P_j exists if and only if resource allocation graph (RAG) contains two edges $P_i \rightarrow R_q$, $R_q \rightarrow P_j$ for resource (R_q).
* Deadlock exists if and only if wait for graph contains a cycle.
* Deadlock detection algorith (DDA) periodically checks the graph for a cycle.

Multiple Instance of Resource Type.

* The wait for graph is not applicable to the graph with multiple instances of each resource. For that it will use Banker's algorithm to detect the where the deadlock exists.
(Banker's safety algorithm).

(iv) Recovery.

* When deadlock detection algorithm examines the deadlock, several possibilities exists.

(i) to inform the operator that a deadlock has occurred and to lead the operator deal with the deadlock manually.

(ii) let the system recover from deadlock automatically.

* There are two options for breaking a deadlock cycle:

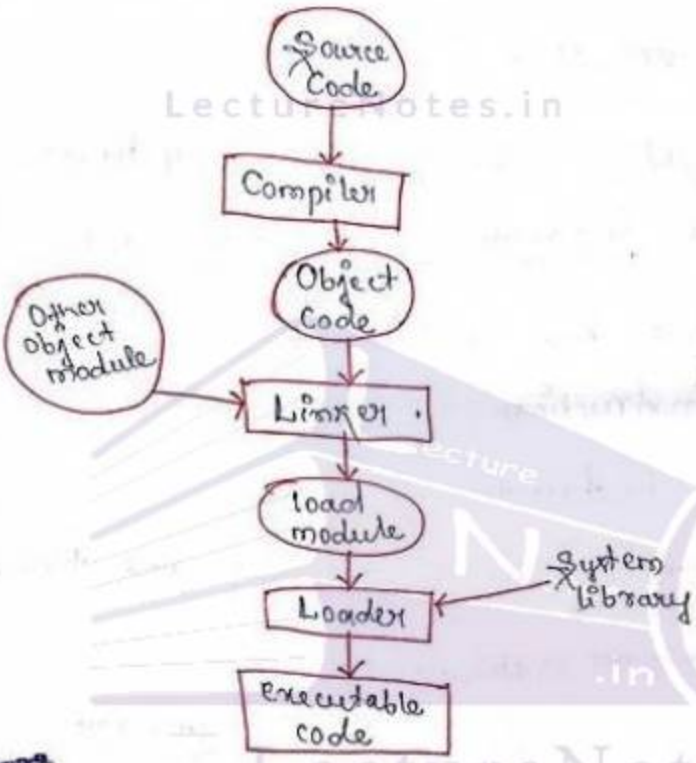
* (i) to abort one or more processes.

(ii) to break the circular wait.
(process termination).

(iii) to pre-empt some resources from one or more deadlocked processes.
(resource termination),

29/9/2015

Memory Management

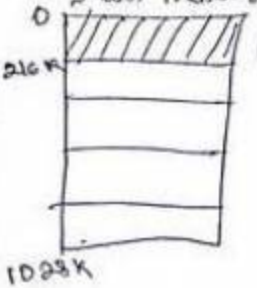


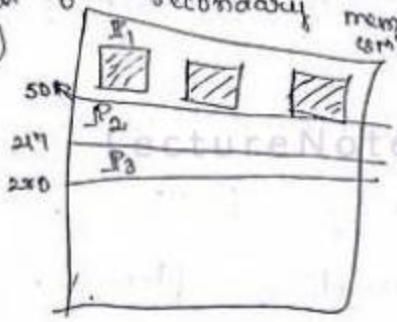
m.c (other compilation) → m.obj → m.bin → m.exe

secondary memory (backing store)

Physical (main) memory

Main memory (MM) → physical address





→ address generated by SM → logical address

* One of the major functions of the OS is memory management.

* OS controls the allocation and de-allocation of physical memory. It keeps track of memory occupancy, loading of programs into free memory space, dynamic allocation and deallocation of memory to executing processes, etc.

* Each instruction gets executed in a systematic manner known as instruction execution cycle.

* Each execution cycle has 3 steps:
  (i) fetch the instruction
  (ii) decode the instruction.
  (iii) get the operands from memory and then execute.

29/9/2015 -

* The entire execution fetch phase of a program is divided into 3 phases:
  (i) compilation phase
  (ii) loading phase
  (iii) execution phase.

* The address generated by CPU is known as logical address. Similary, the address

generated by physical memory is known as physical address.

* Address binding is a method used to map the logical address to the physical address in the memory. This binding of addresses of instructions and data to actual physical addresses can take place at 3 different times during the execution process:

(i) compile time binding.

The binding at compile time generates the absolute code. It requires that it must know at compile time that where will a process reside in memory. This is used for small and simple systems. The problem with this binding is if there is a change in starting address of a process, then the entire process must be recompiled to generate absolute address.

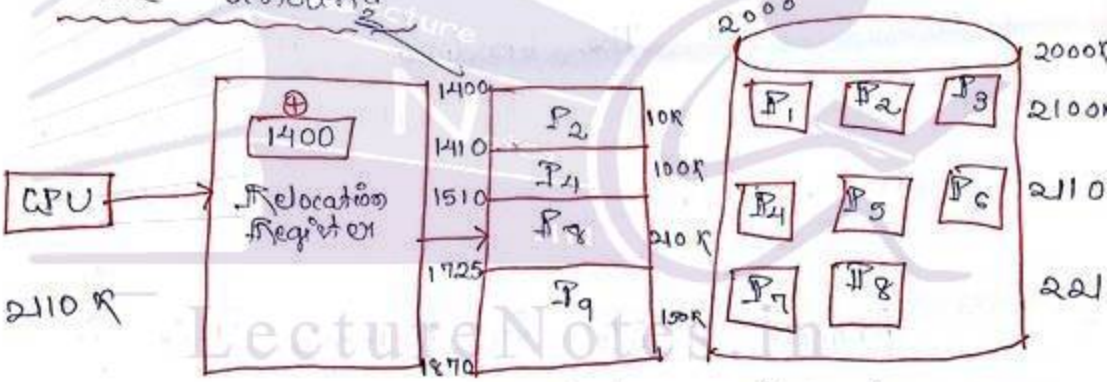
(ii) load time binding.

In this case, compiler

Generates relocatable address which are converted to absolute address at load time. Process cannot move around during execution.

iii) execution time binding:

If the process can be moved during its execution from one memory segment to another memory segment. That is known as execution time binding.



* The set of all logical addresses generated by the program is known as logical address space.

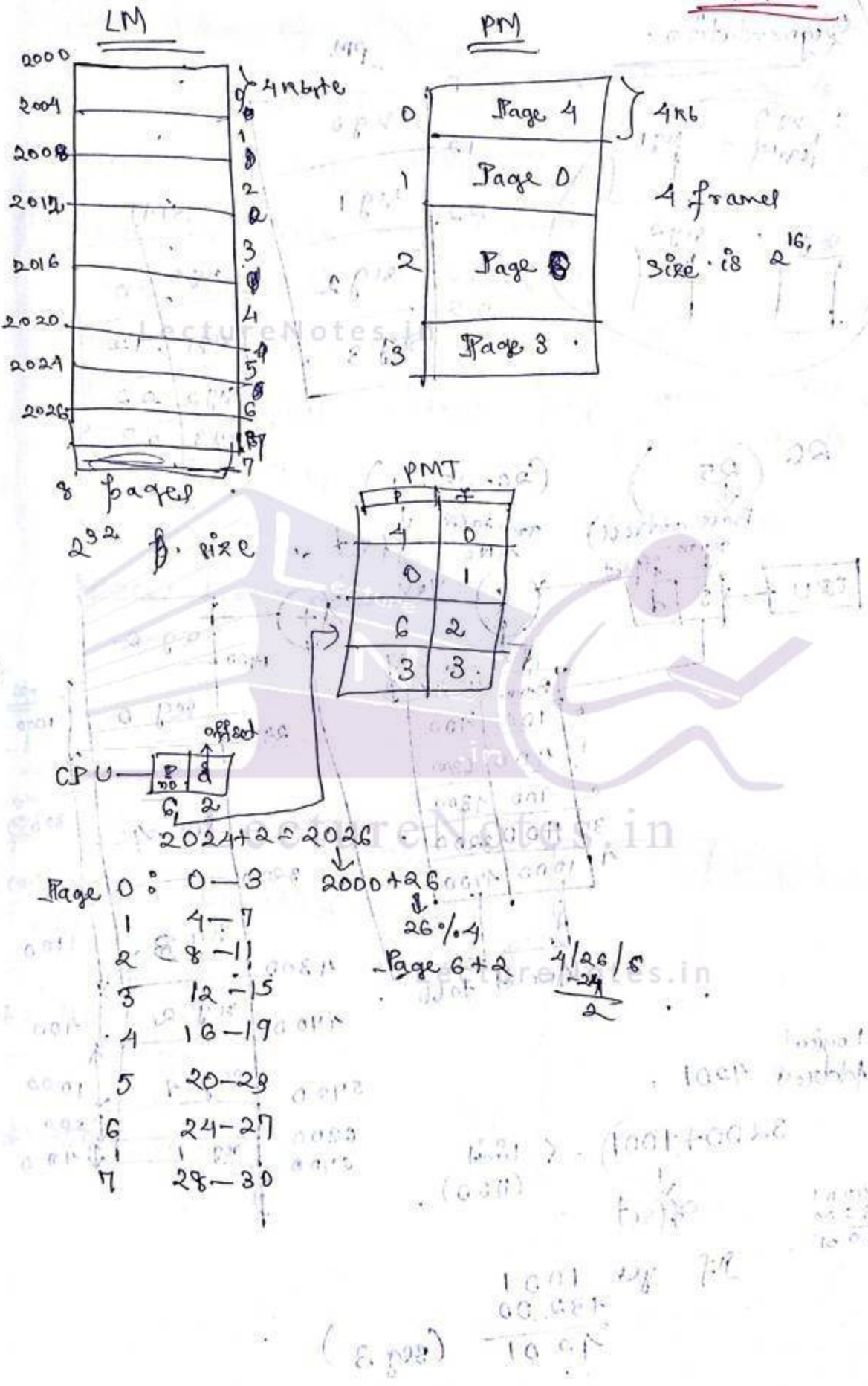
* The set of all physical addresses is known as physical address space.

* The execution time mapping from logical address to physical address is done by the memory management unit (MMU).

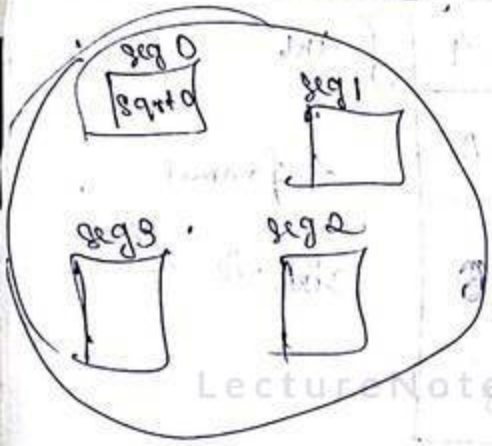
* Memory management unit consists of relocation register. It contains the value of starting (base) address of physical memory.

* The address generated by CPU is added with the value of relocation register to give the correct physical address of an instruction.

13/10/2015

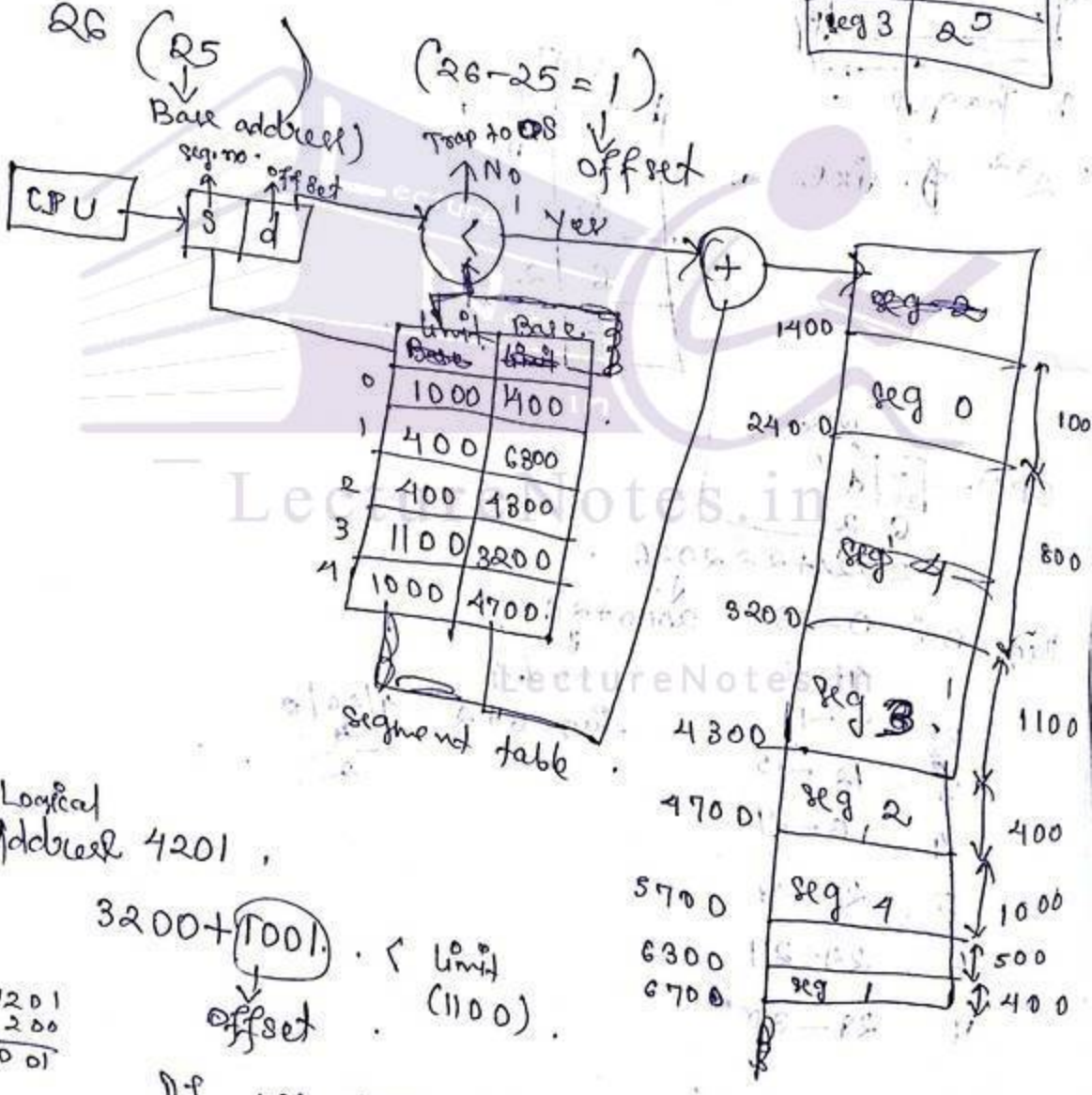

LM
PM
2000
2004
2008
2012
2016
2020
2024
2028
4Kbyte
0
1
2
3
4
5
6
7
8 pages
2^{32} B size
Page 4
Page 0
Page 6
Page 3
4K6
4 frames
size is 2^{16}
PMT
P
F
4 0
0 1
6 2
3 3
CPU
P
d
offset
6
2
2024+2=2026
2000+26
26%4
Page 6+2
4/26/6
24
2
Page 0 : 0—3
1 4—7
2 8—11
3 12—15
4 16—19
5 20—23
6 24—27
7 28—30

Segmentation.



SMT

seg 0	0
seg 1	12
seg 2	22
seg 3	25



Logical Address 4201.

3200 + 1001 . < Limit (1100).

Offset

4201
−3200
1001

If yes

1001
+3200
4201 (seg 3).

Protection of Page Table.

* For protection of page table, one bit is attached to each entry in the page table i.e., valid and invalid bit. When this bit is valid, this indicates that the associated page is in the process's logical address page and hence, a valid page. If it is invalid, it is not in the process's logical address space. Hence, it is an invalid page.

Q: Segment 2 is in which physical address?

A: 4300.

8th edition

8.21.

(a) 0430 - 100+330 < 90.

100 | 430 | 3
-400
30

(b) 110 100+10 < 90

90
+100
190.

(c) 2500 600+1900.

2500
-600
1900

14/10/2015.

Types of structures of Page Table.

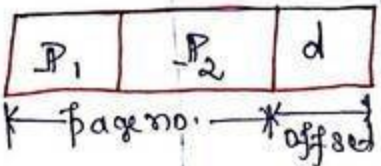
Multi-level or Hierarchical Paging.

* Most modern computer systems support a large logical address space. In such environment, page table itself becomes extremely large.

* In case of 32 bit machine, if page size is 4 Kb i.e., 2^{12} ($2^2 \times 2^{10}$), then page table has $2^{32}/2^{12}$ million of entries.

$= 2^{20}$ entries.

* Solution is to divide the page table into small pieces. One is two level paging.



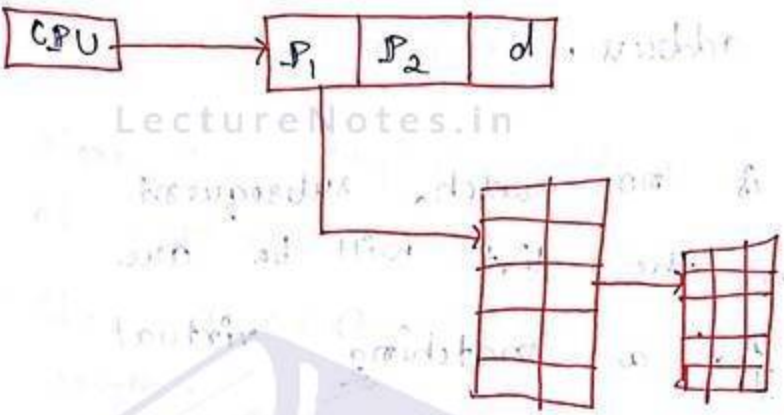
So logical address is divided into three parts:

$P_1 \rightarrow$ outer page

$P_2 \rightarrow$ inner page

$d \rightarrow$ offset.

Since we page a table, the page no. is divided into two parts:

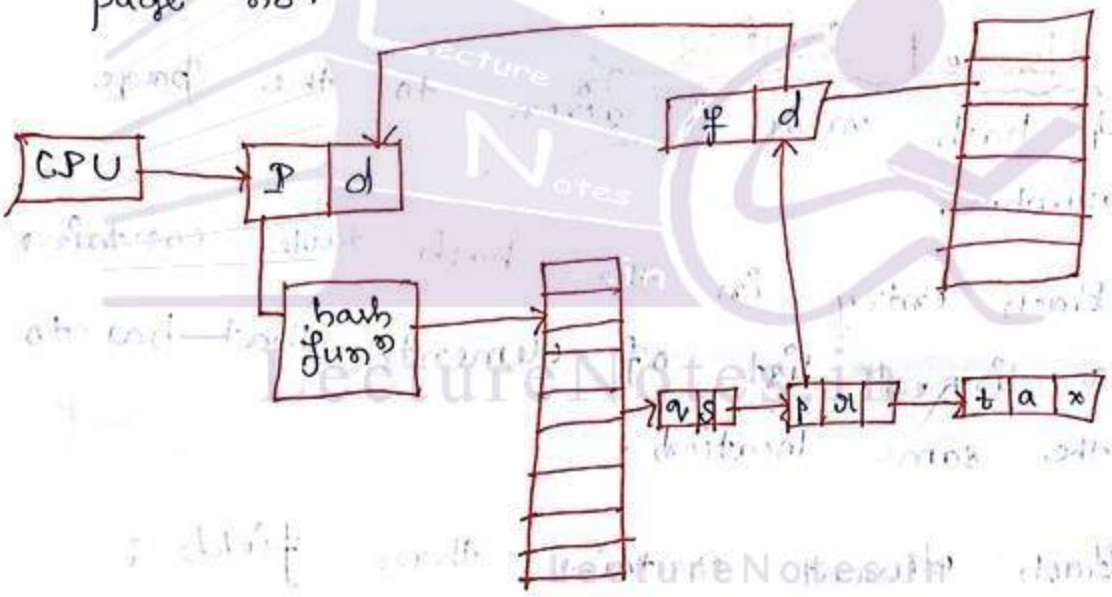


Hashed Paging

* A hash value is given to the page number.
* Each entry in the hash table contains a linked list of elements that has to the same location.
* Each element contains three fields:
  (i) page no.
  (ii) pointer to the next element in the list.
  (iii) offset.
* The virtual page no. is compared with

first element in the linked list, if there is a match, the corresponding frame is used to obtain the desired physical address.

* If there is no match, subsequent entries in the list will be are searched for a matching virtual page no.



(iii) Inverted Paging:

Segmentation.

Segmentation is a memory management scheme that supports the user's view of memory.

*- Here, the logical address is a collection of segments.

* Each segment has a name & a length.

*- A segment is a logical unit such as main program, procedure, function, method, local variables, stack, symbol table, global variable, etc. It views memory as a collection of variables sized segments rather than fixed sized pages. So, each tuple in the segment table contains segment no. and offset. (the LA is generated by CPU)

*- In paging, user specifies a single address which is partitioned by the MMU into page no. and offset.

*- Normally, the compiler automatically constructs segments reflecting the input

program.

*- The segment table maps to the one dimensional physical address.

*- Each entry in the segment table has a segment base & a limit.

*- The segment base contains the starting physical address where the segment reside in memory.

*- Segment limit specifies the length of the segment.

*- Segment table base register (STBR) points to the segment table in memory.

*- Segment table length register (STLR) indicates the no. of segments used by a program.

*- Segment no. (s) is legal if s is less than STLR.

Dynamic Loading

30/9/2015.

* To obtain better memory space utilisation, we can use dynamic loading.

* With dynamic loading, a routine or function is not loaded until it is called. All routines are kept in disk in a relocatable format.

* The main program is loaded into the memory and is executed.

* When a routine needs to call another routine, the calling routine first checks

to see whether the other routine has been loaded. If not, the relocated linking loader is called to load the desired routine into memory.

* The advantage is that an unused routine is never loaded. This method is useful when large amount of code are needed to handle infrequently occurring cases. It doesn't require special support from operating system. It is the responsibility of the user to design their programs to take the advantage of such type of systems.

Logical address

2000
2100
2101

P_1
P_2
P_3
P_4

0
100K
101
216K
217K
320K
321K
500K.

Secondary memory

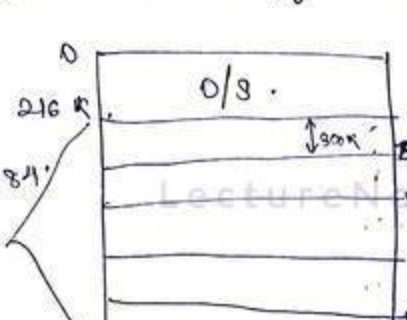
Virtual logical address is 2000.

swapping
swap in / swap out

roll in / roll out.

Memory partition: done by continuous partition.
fixed partition (fixed size is allocated).
variable partition. (variable size)

Fixed: OS occupy the highest from in memory.



300K
216 if the OS is capable of running?
216 5 process.
no. 1784/5 ≈ 300.
no.

if a process more than size comes, the process is divided into sub problems.

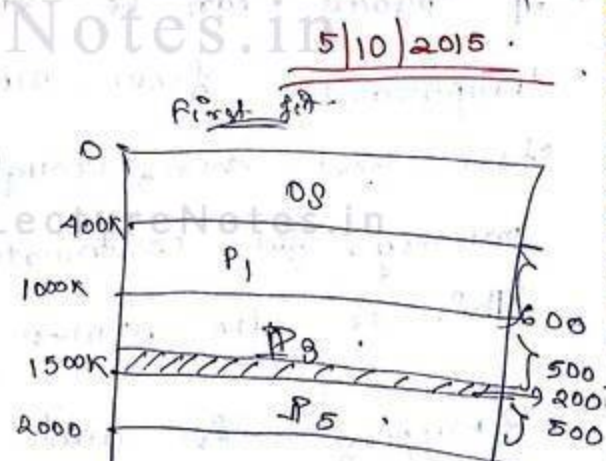
Type of resource allocation:
(i) first fit (1st fragment that can accommodate the process).
(ii) worst fit
(iii) best fit.

5/10/2015.

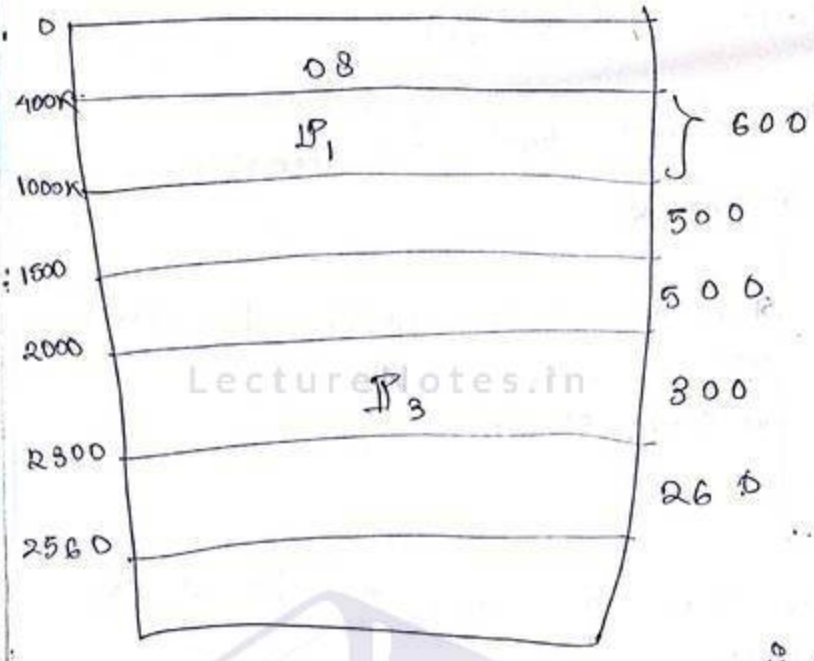
Variable partition.

Process	Memory
P_1	600 K
P_2	1000K
P_3	300K
P_4	700K
P_5	500 K

First fit.



Best fit: (best possible soln).



worst fit (the biggest space is given to 1st process
2nd biggest 2nd
3rd 3rd

Swapping

* A process can be swapped out (Secondary or Backing) temporarily from memory to a backing store and then, brought back to the memory, for continuation of execution. This is also known as roll in - roll out.

* Swapping is used for priority based scheduling algorithms i.e., lower priority process is swapped out so that higher priority process can be loaded and

executed.

* A process that is swapped out will be swapped back into the same memory space if binding is done at load time.

* A process can be swapped into different memory space if binding is done at execution time.

* Major part of swapping is the transfer time/rate of the process.

* The transfer time is directly proportional to the amount of memory swapped out & swapped in.

* Memory allocation is done in 2 ways:
  (i) continous memory allocation.
  (ii) variable partition.

(i) Continous Memory Allocation

* Here, main memory is divided into 2 partitions:
  (i) OS
  (ii) used for user process.

* Each process is stored in a single continous section of memory.

* Memory mapping is done by using memory management unit (MMU) and relocation register.

* Relocation register contains the smallest value of physical address and limit register contains the range of physical addresses.

* MMU maps the logical address dynamically into physical address by adding the relocation register value to logical address.

(ii) Variable Partition.

* When a partition is free, a process is selected from the input queue and is loaded into the free location.

* After the termination of that process, the partition becomes available for another process.

* The OS keeps track of the information about allocated partitions and

free partitions.

The free partitions are known as hole.

It is divided into 3 types:

(i) First fit

Allocate the first hole that is big enough. Searching can be started from the beginning of the holes where the previous first fit search ended. We stop searching as soon as we find a free hole that is large enough.

(ii) best fit.

Allocates smallest hole that is big enough to accommodate the process. We must search the entire list.

(iii) Worst fit.

Allocate the largest hole that is present in the list to a process.

External fragmentation:

In variable partition, we have a no. of small small holes. If they are taken together can satisfy a process

but as they are not continous, they cannot be given to a process.

Internal fragmentation:

After allocation of a fragment to a process, if a small hole is left which cannot be accommodated given to any process, then that small hole is also given to the process.

External fragmentation:

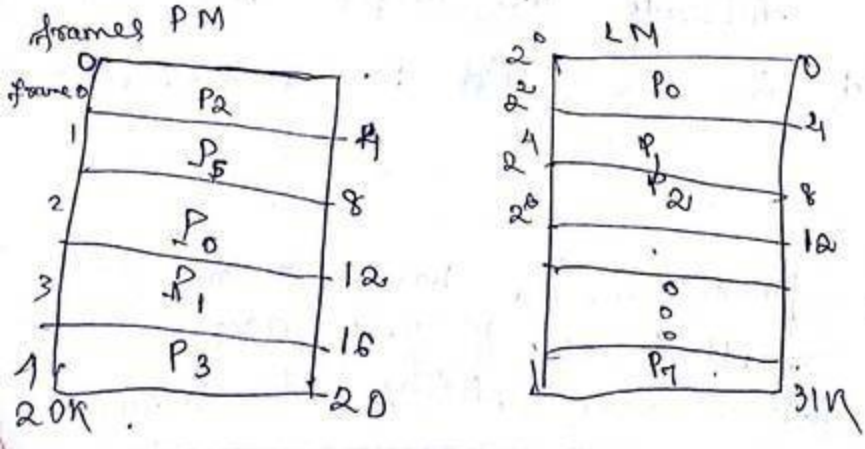
Sol: take all the used memory to one end and the unused to other end so that, the continous memory can be given to any other process. (Compaction) → costly, as have to remember.

Sol^n of external fragmentation: Paging.

6/10/2015.

Paging:

Division of logical memory → pages
physical → frames



page table maps

those pages which are in RM

process	frame
2	0
5	1
0	3
1	4
3	5

External Fragmentation

* External fragmentation exists when enough total memory space is available to satisfy a process, but it is not a continous one, storage is fragmented into a large number of small holes. This problem can be severe, in worst case we could have a block of free memory between every two process. This is also known as 50 percent rule.

* The solution to the external fragmentation is compaction.

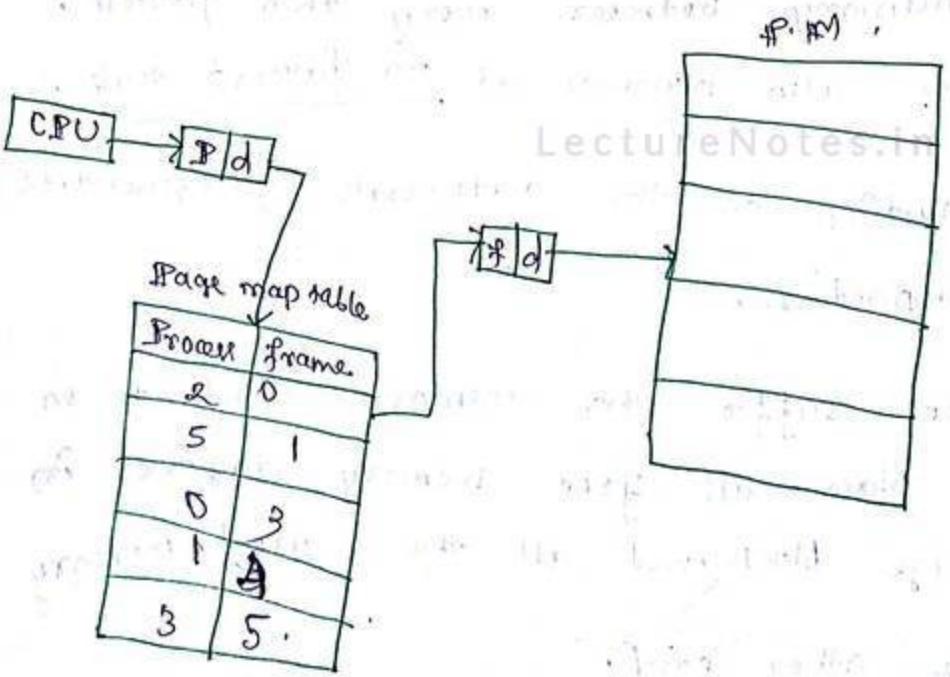
* Here, we suffle the memory contents so as to place all free memory together in one large block and all the used memory to the other end.

*- Though compaction can solve the problem of external fragmentation, it is not easy to implement. So, another solution is paging.

Internal Fragmentation

*- We usually allocate a small hole as a part of larger request.

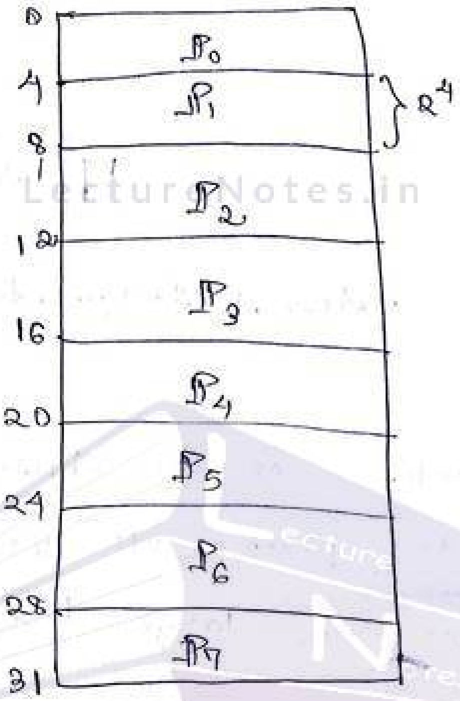
*- The allocated memory is slightly larger than the requested memory. The difference between these two numbers is known as internal fragmentation.



$2^m \rightarrow$ logical memory

$2^n \rightarrow$ size of page.

$m-n \rightarrow$ page no.



If logical address = 3, then (3-0)

$\longleftarrow \times 4 + 3$,

↓

page size

$4 \overline{)3}$ 0, 0, 3

$\Rightarrow$ $0 \times 4 + 3 \longrightarrow$ page no : 0 .

If logical address = 12, then (12-12).

$3 \times 4 + 0$

$4 \overline{)12}$ 3, 12, 0

$\Rightarrow$ page no. 3 .

If logical address = 21, then (21-20)

$\underline{5} \times 4 + 1$.

$4 \overline{)21}$ 5, 20, 1

LA = 13.

$3 \times 4 + 1$

PA = corresp. frame of page × frame + offset.

$4 \times 4 + 1 = 17$.

4|13|3
-12
1

7|10|2015-

Paging

*- The solution to the external fragmentation is compaction.

The idea of compaction is to shuffle the memory contents to place all free memory together in one large block.

*- Other solution to the external fragmentation is paging.

*- Paging avoids the problem of fitting variable memory chunks onto backing store.

*- Here, physical memory is broken into fixed sized blocks known as frames.

*- Logical memory is also broken into fixed sized blocks known as pages.

*- When a process is to be executed,

its pages are loaded into any available memory frames.

* Every address generated by CPU has 2 parts :
  (i) page no.
  (ii) offset.

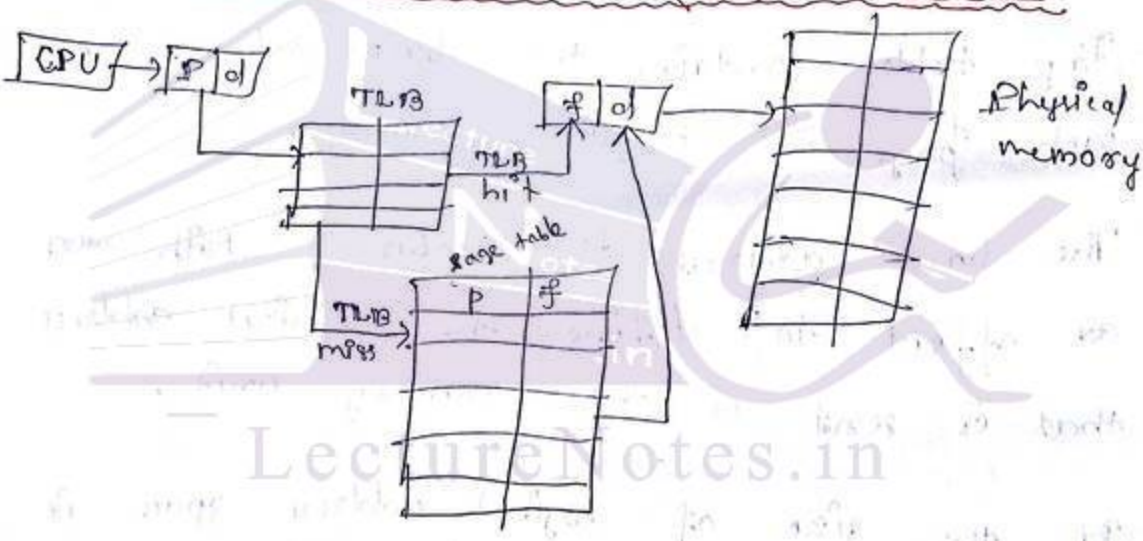
Page no. is an index into the page table.

* Page table contains the base address of each page in physical memory.
* The base address is combined with an offset to define the physical address that is sent to the memory unit.
* If the size of logical address space is 2^m and a page size is 2^n addressing bits, then $(m-n)$ bits of logical address designate the page no. and n lower bit designate the offset value.
* Suppose logical address is 2^5 Kbytes i.e., 32 bytes and each page is 2^2 = 4bytes long and physical address is 2^4 bytes

i.e., 16 bytes, then there are 8 pages in logical memory and four frames are present in physical memory.

Eg: If the logical address is 13, then its corresponding physical address is 17.

TLB (Translation Look Aside Buffer)



The percentage of time that a particular page no. is found in the TLB is called hit ratio.

Q. If it takes 20 nanosec to search a TLB and 100 nanosec to search a memory, then, what is the effective access time? Hit ratio is 80%.

Ans:-

Hit ratio = 80%

⇒ 80 hits and 20 misses.

Effective access time = 0.8 × (100+20) + 0.2 × (100+100+20) (physical address memory)

= 140 nanosec.

If 100% hit.

1 × 120 = 120 nanosec.

If 100% miss.

1 × 220 = 220 nanosec.

Q. 98% hit ratio. memory access time = 100 ns

TLB access time = 20 ns.

Ans:-

Hit ratio = 98%

⇒ 98 hits & 2 misses.

Effective = 0.98 × (100+20) + 0.02 × (100+100+20)

= 0.98 × 120 + 0.02 × 220

= 117.6 + 4.4

= 122.

*- Page table can be implemented as a set of registers. These registers are high speed registers to make the address translation efficient.

* Use of registers are satisfactory if the page table is small.

* For larger page tables, a page table base register (PTBR) points to the page table.

* Changing of a page table requires changing one of these registers (PTBR).

* The problem with this scheme is the time requirement to access an user memory location. With this scheme, two memory access are needed to access a byte.

* The standard solution is to use the special small high speed cache memory first look up known as TLB.

* Each entry in the TLB has two parts:

  (a) key
  (b) value.

* TLB contains only a few page table entries.

* When a logical address is generated by the CPU, its page number is searched

the TLB

*- If page is found, its frame no. is available and used to access the memory. It is known as TLB hit.

*- If the page no. is not found in the TLB, TLB miss will occur, then a memory reference to the page table will must be made.

*- We add the page numbers and frame nos. to the TLB dynamically so that, we can found the page quickly in the next reference.

Virtual Memory:

Virtual memory is a technique that allows the execution of the processes that are not completely in memory.

*- We can run a program larger than the physical memory.

*- The entire program is not needed in many cases such as programs have codes to handle unusual error conditions. But, that type of error doesn't happen.

*- A symbol table has an entries for 3000 symbols, but the program has less than 200 symbols.

*- If we can run a program that is partially in memory, then we have the following benefits:

(i) the program would no longer worry about the amount of physical address.

iii) since each user program could take less primary memory, more no. of programs can be executed.

* Virtual memory is a separation between user logical memory from physical memory.

15/10/2015.

Demand Paging

* Virtual memory allows files and memory to be shared by several different processes through page sharing.

* Virtual memory is commonly implemented by demand paging.

Demand paging is similar to paging with swapping.

* Process A initially resides in secondary memory.

* When we want to execute a process, we swap in this process into memory

* Rather than swapping the entire process into memory, we use lazy swapper.

* A lazy swapper never swaps a page into memory unless that page is needed.

* Since a swapper manipulates the entire process, a pager is used to manipulate the individual pages of a process.

* When a process is to be swapped in, instead of swapping the whole process, the pager brings only the necessary pages into the memory. It decreases the swap time as well as the amount of physical memory needed.

* Here, valid - invalid bit is used. When this bit is valid, it indicates that the associated page is legal and in main memory. If it is invalid, then this page is either invalid or valid but, currently in the disk.

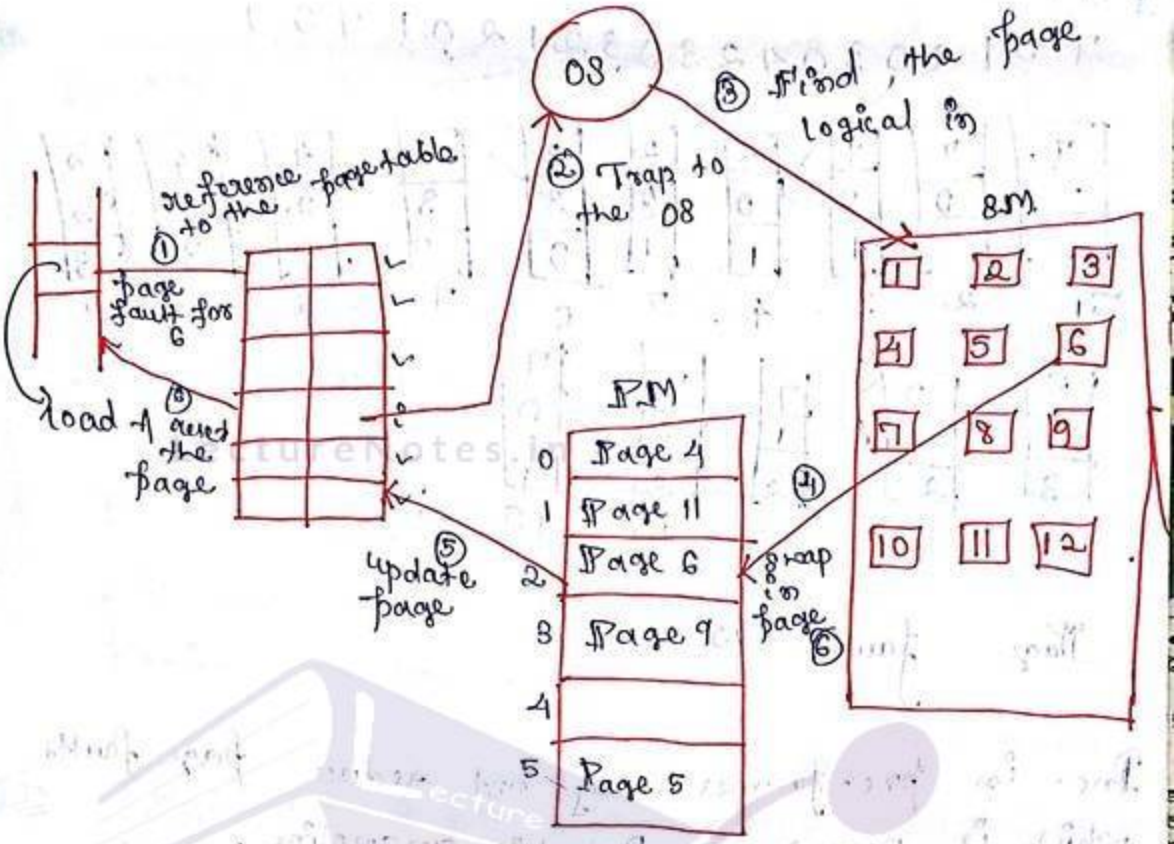
* - The page table entry for a page that is not currently in memory is marked as invalid.

* - Accessing to a page marked as invalid makes page fault trap. This trap is the result of operating system failure to bring the desired page into memory.

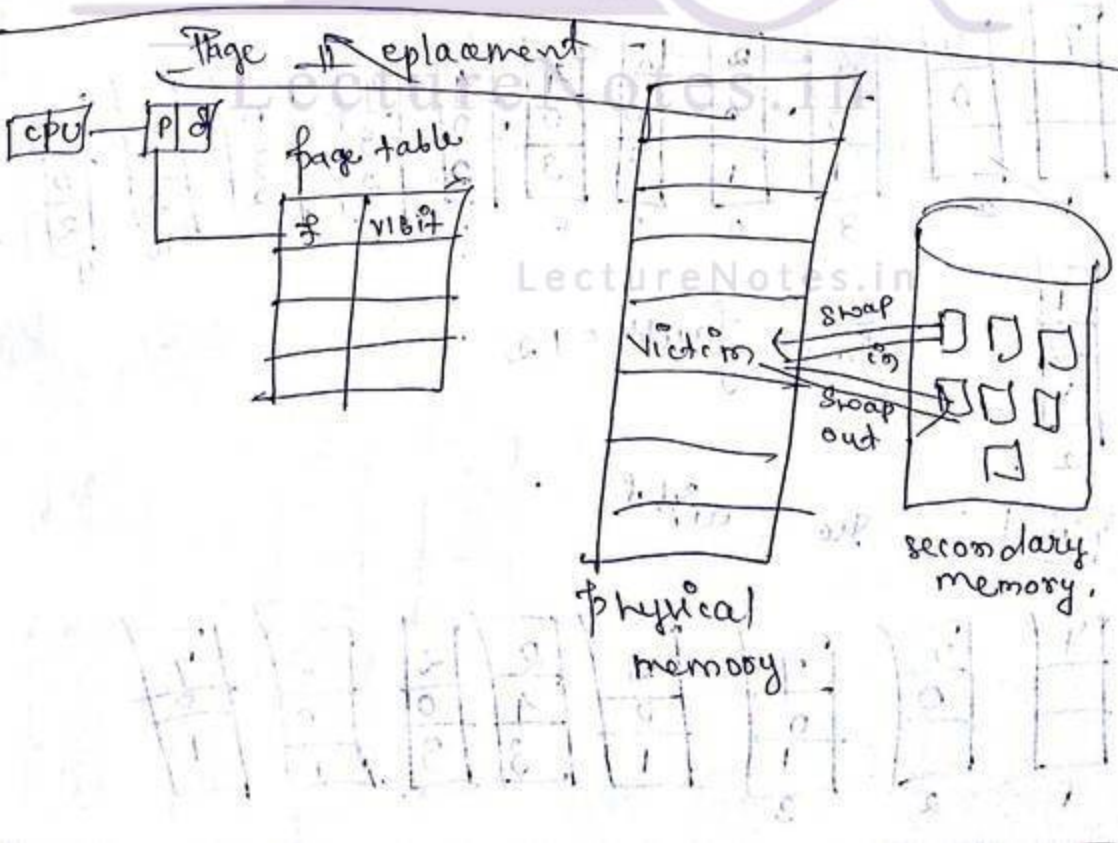
Page Fault

Accessing a page which is not present in the physical memory as well as in the page table is known as page fault.

Procedure for handling page fault

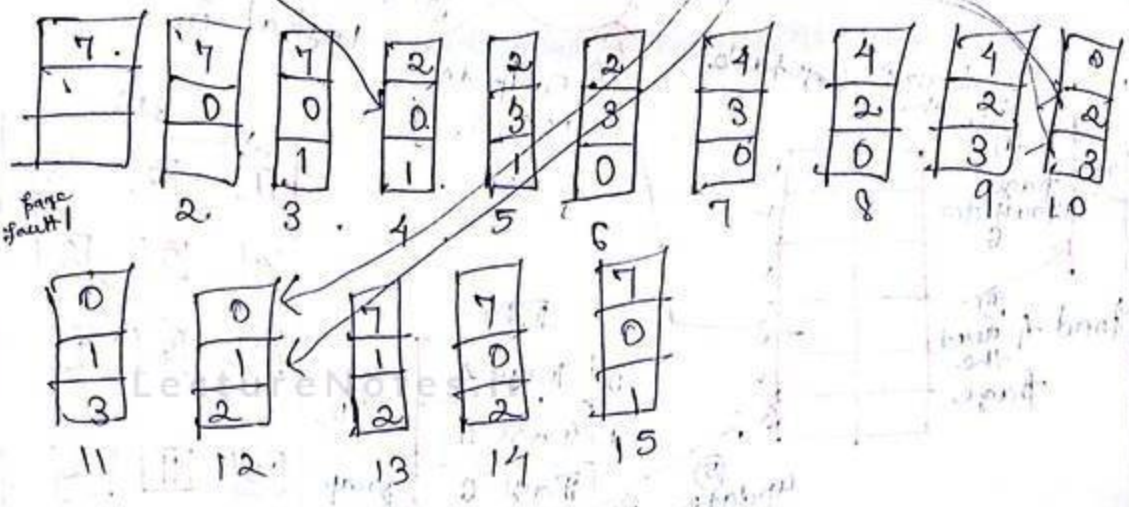

OS
③ Find the page logical in
reference to the page table
① page fault for 6
② Trap to the OS
BM
1 2 3
4 5 6
7 8 9
10 11 12
PM
0 Page 4
1 Page 11
2 Page 6
3 Page 9
4
5 Page 5
load 4 reset the page
④
⑤ update page
swap in page
⑥

29/10/2015


Page replacement
CPU
P d
page table
f visit
Victim
swap in
swap out
secondary memory
physical memory

FIFO:

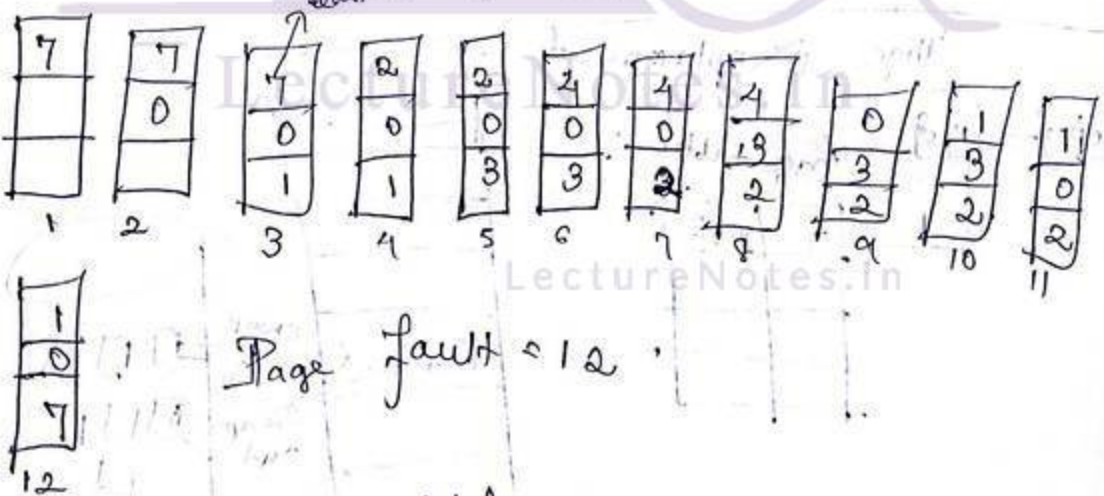
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



Page faults = 15

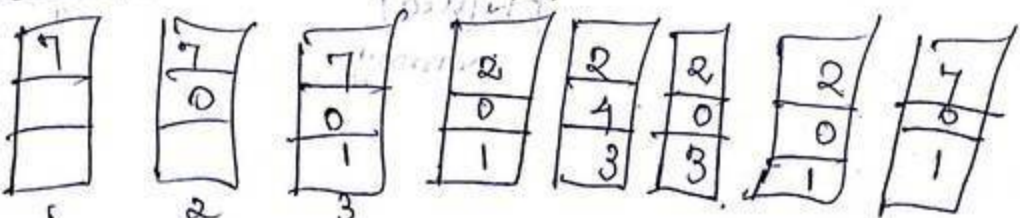
Inc. in pre-frames may not reduce page faults which is known as Belady's anamolies.

LRU: (Least Recently Used) see left.



Page fault = 12.

Optimal: see right.



In FIFO, the first page that is coming to the memory will be replaced first.

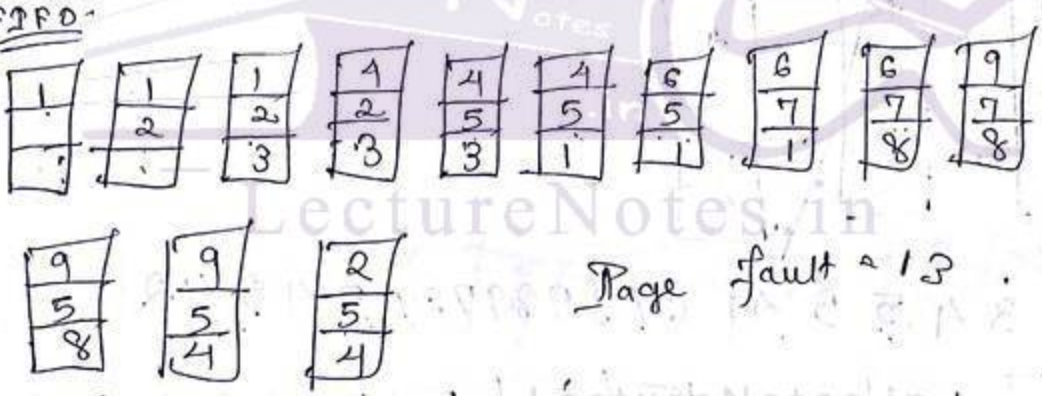
In LRU, the least recently used page will be replaced first.

In optimal, replace the page that will not be used for the longest period of time.

Eg. 1 2 3 4 5 3 4 1 6 7 8 7 8 9 7 8 9 5 4 5 4 2

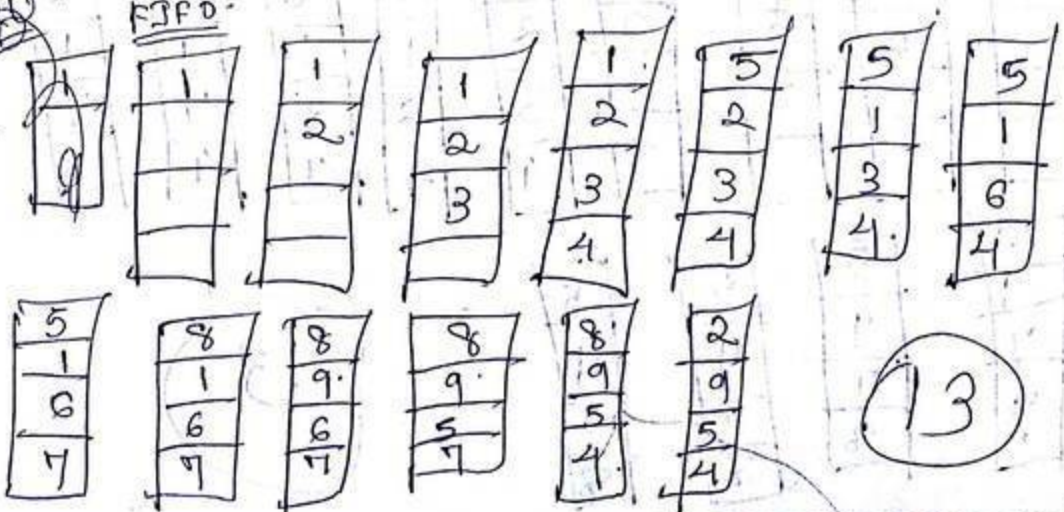
4 frames

FIFO:



Page fault = 13

LRU FIFO

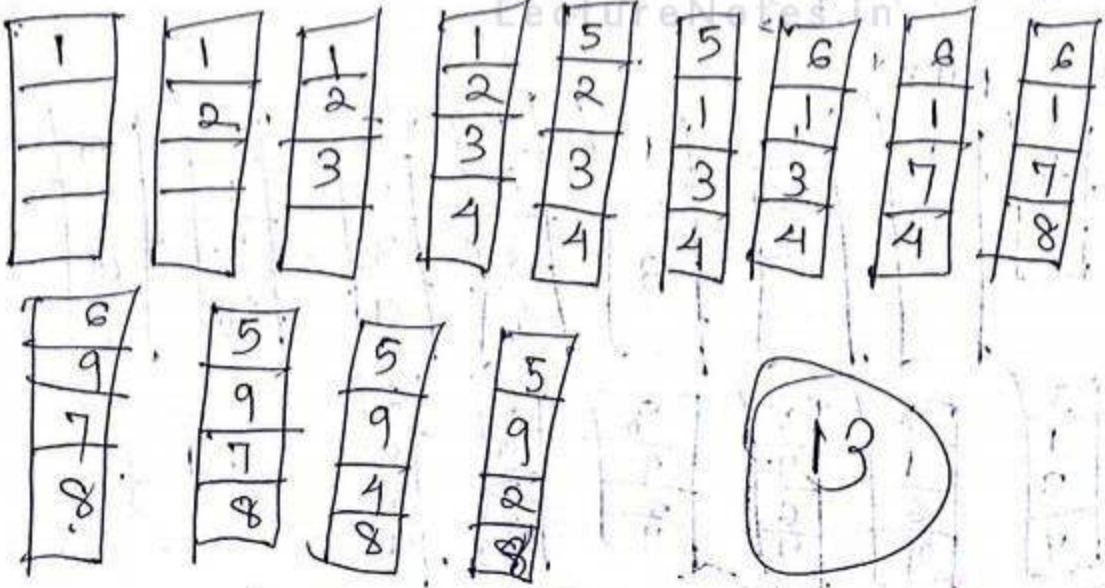


13

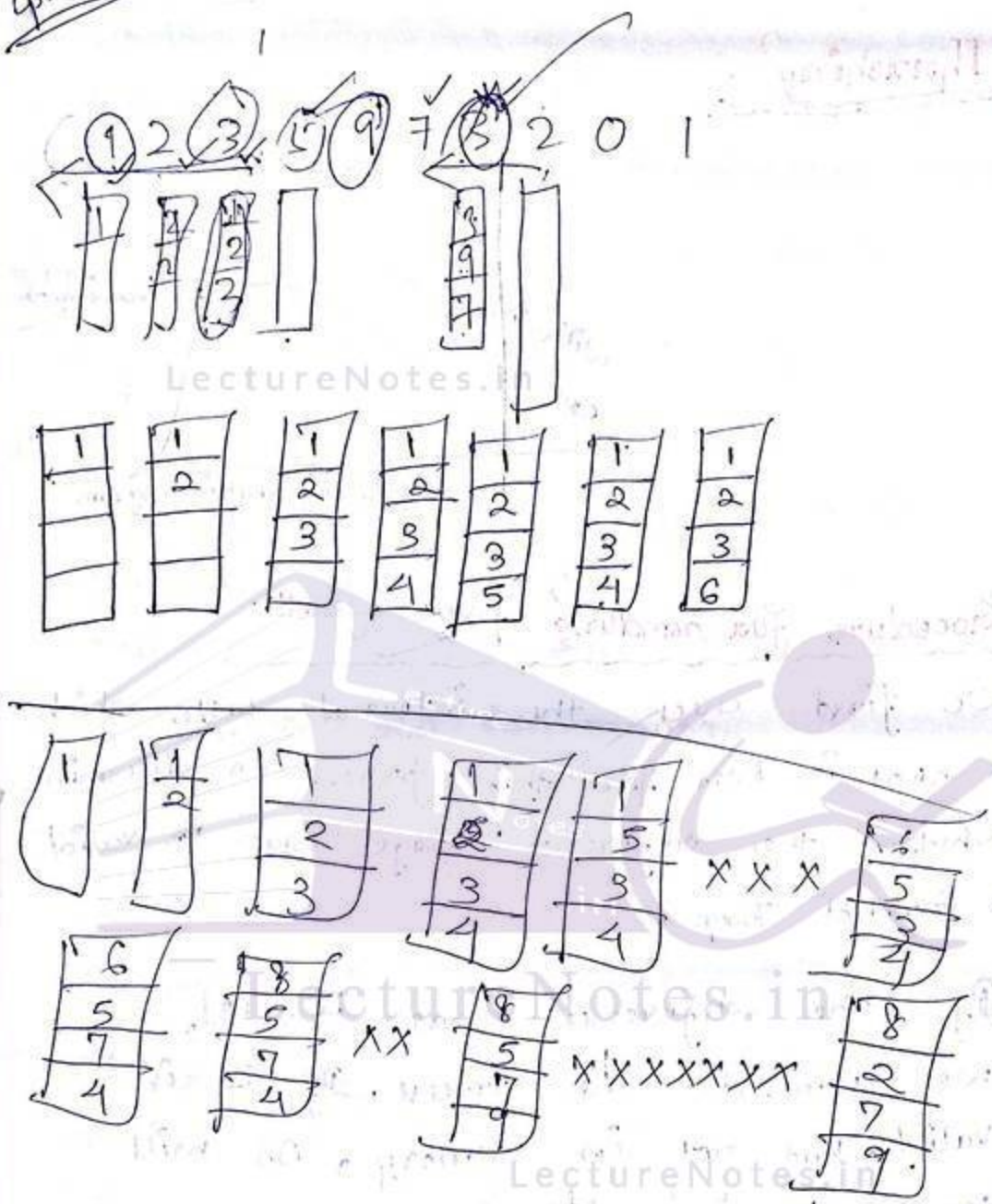
LRU

LRU

1 2 3 4 5 3 4 1 6 7 8 7 8 9 7 8 9 5 4 5 4 2

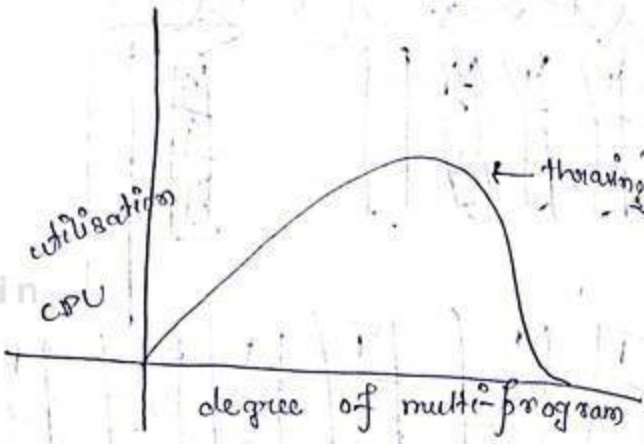


Optional:


1
1 2 3 5 9 7 3 2 0 1
1
1 2
1 2 3
1 2 3 4
1 2 3 5
1 2 3 4
1 2 3 6
1
1 2
1 2 3
1 2 3 4
1 5 3 4
x x x
1 5 3 4
6 5 7 4
8 5 7 4
x x
8 5 7 9
x x x x x x x
8 2 7 9

3/11/2015

Thrashing



Procedure for handling page fault.

1. We first check the internal table which is kept in PCB for this process to determine whether the reference page was valid or invalid page.

2. If the reference was invalid, we terminate the process. If it is valid, but not in memory, OS will search backing store or sec. memory for this page.

3. We find a free frame by taking one from free frame list.

4. Schedule the disk operation to read the desired page into the newly allocated frame.

5. Modify the page table to indicate that the page is now in memory.

6. Restard the instruction that was interrupted by illegal address trap. In this way, we are able to execute a process even though the process is not in memory.

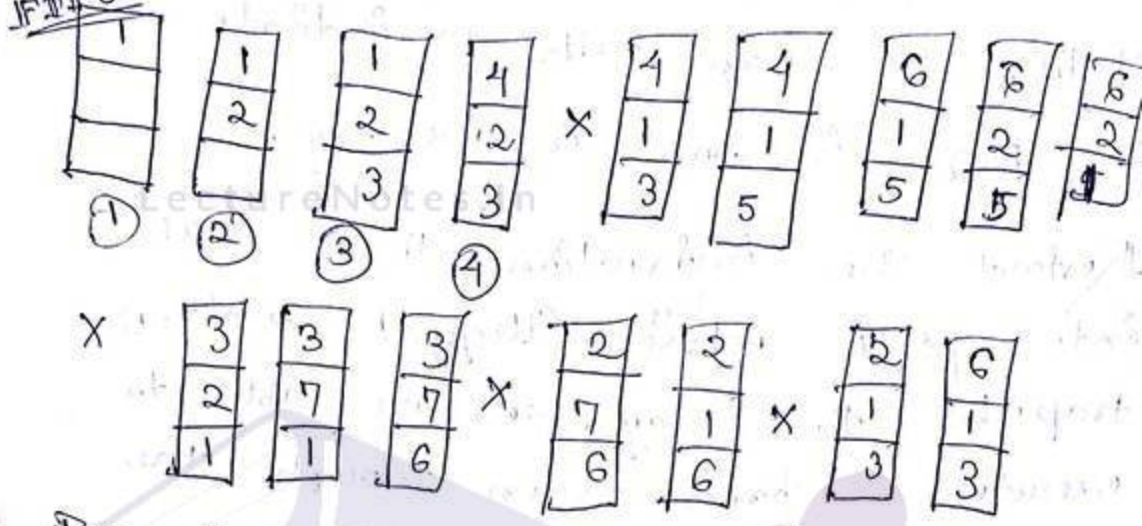
* In the extreme case, we could start executing a process with no pages in memory.

* Never bring a page into memory until it is required, it is known as pure demand paging.

Page Replacement

Ex: 1 2 3 4 2 1 5 6 2 1 2 3 7 6
3 2 1 2 3 6.

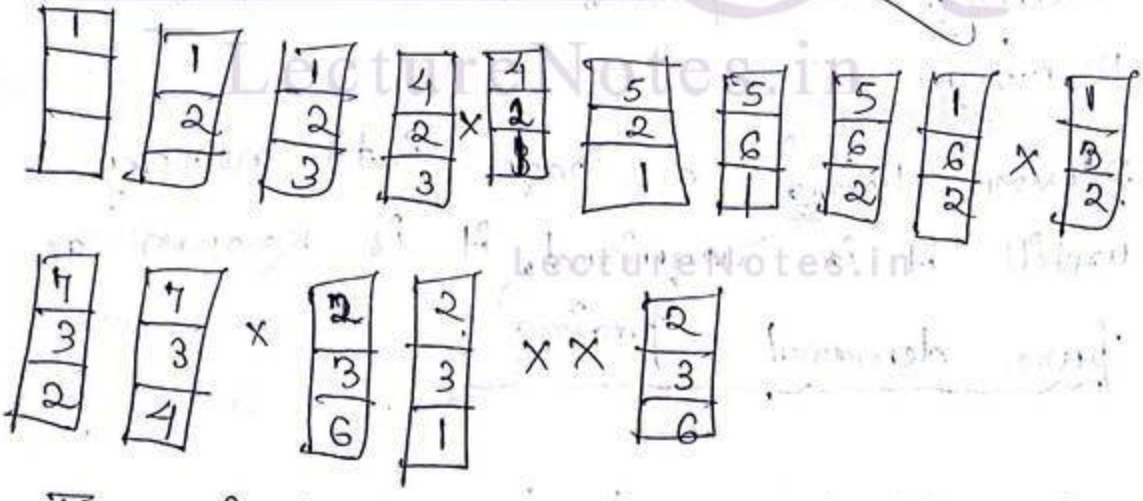
FIFO:



Page fault = 16 Total 20 try

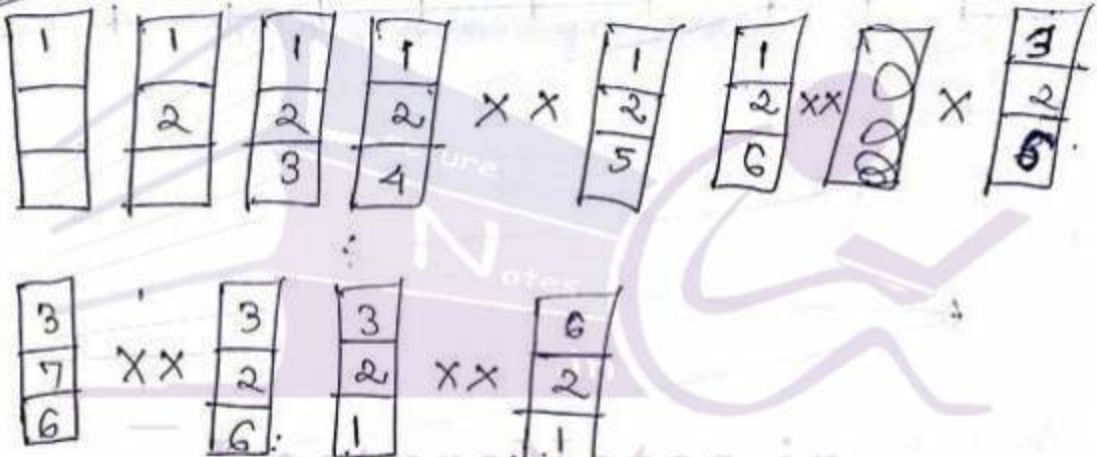
LRU:

See left side, who coming last.



Page fault = 15.

Optimal: see right, who coming last.



Page fault = 11