Operating Systems : Overview

Operating systems is one of the most used software

Few popular O.S :-

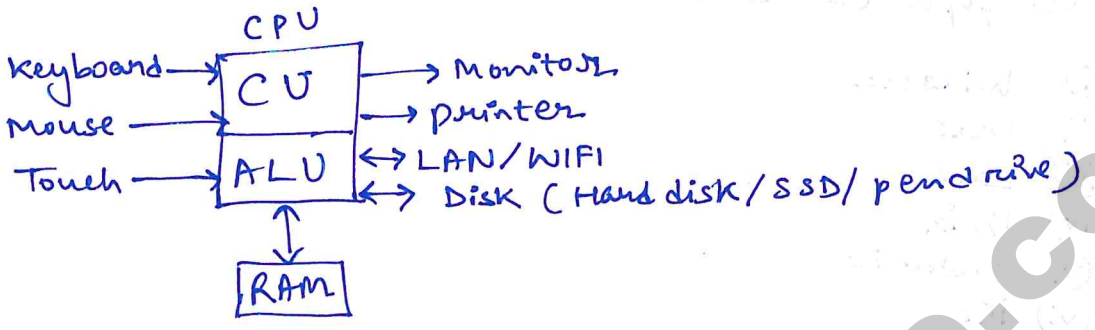
(i) Windows
(ii) Mac
(iii) Linux
(iv) Unix
(v) Android
(vi) iOS

Most of the laptops / desktops are driven by Windows / Mac. Most of the servers are powered by Linux. 99% of the mobile devices run on Android / iOS. We don't use Unix directly now-a-days. Unix is one of the earliest and very popular O.S. Mac and Linux have been inspired heavily by Unix. The core (kernel) of Mac is actually a Unix-like system. Linux is an open-source variation of Unix. The kernel of Android is a Linux or Unix based kernel.

E.g., If we are writing something to a file in C programming language, we do that using fprintf or fopen functions. Similarly we use printf to print something to the monitor. And we use scanf to get some input from the keyboard. The internal working of all these are powered by the Operating systems.

Q. What does an O.S do ?

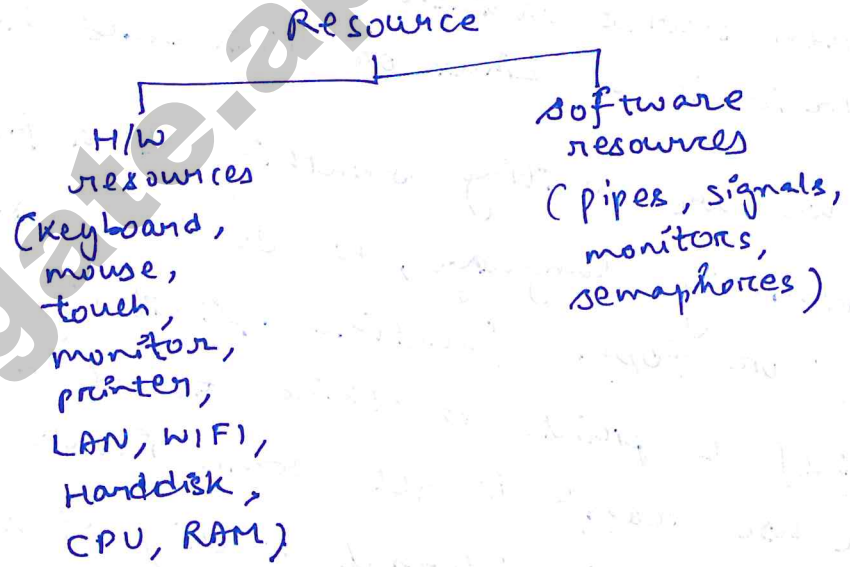
Operating systems can be thought of a resource manager.



CU is the control unit and ALU is the Arithmetic and Logic Unit.

CPU reads data from RAM effectively. It takes inputs from keyboard and executes somethings.

The CPU doesnot talk to these (keyboard/mouse,etc) directly rather CPU talks to all of these via the RAM.



Operating systems provides convenience to the users of the operating systems. O.S wants to use all the resources very efficiently.
Operating systems provides a bunch of functionalities to the software engineer to simplify application development.

Most operating systems are pieces of software written in C/C++.

In this course we will go through the following concepts:

1. Process - Management (management of CPU)
   - scheduling
   - IPC & synchronization
   - concurrency
   - Deadlocks
   - Threads

2. Memory / RAM management
   - RAM organization
   - Techniques
   - virtual memory

3. File Systems & Device management
   - Interface
   - Implementation details

File Systems → stored in HDD, SSD, pendrive

Device management → keyboard, printers, monitors

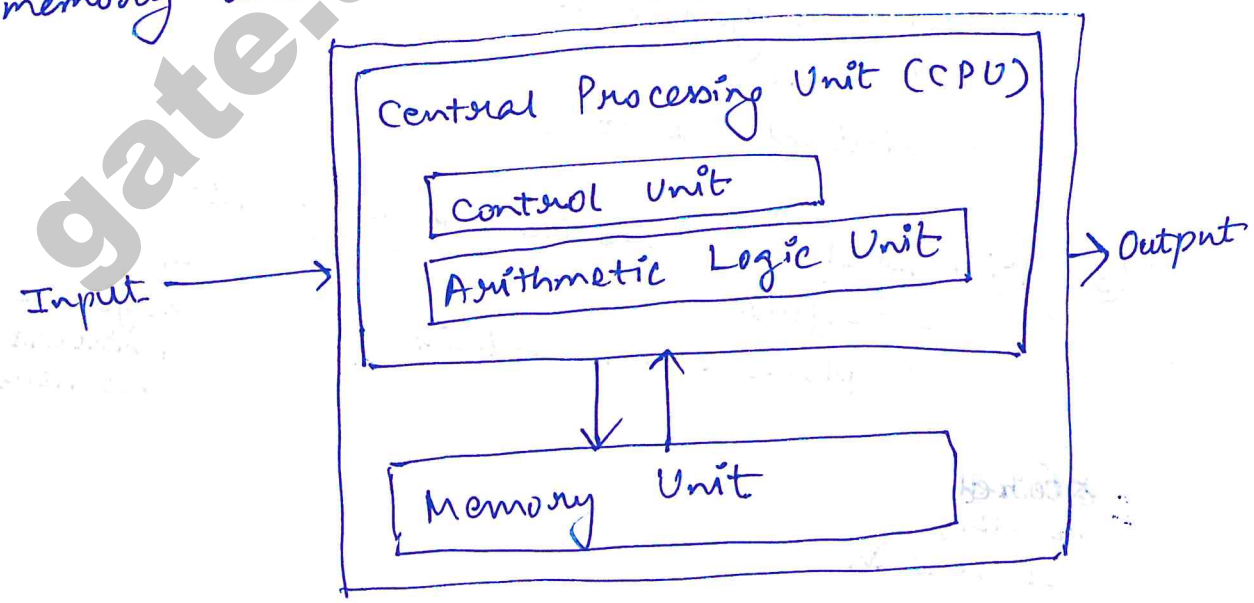
John von Neumann Architecture and History of Operating Systems.

Neumann Architecture :

John von Neumann is considered one of the greatest mathematician and computer scientist who came up with a simple and elegant architecture of what a computer is. Neumann architecture is also often referred to as Stored-program concept.

The stored program concept says any program that we want to execute will be stored in the RAM.

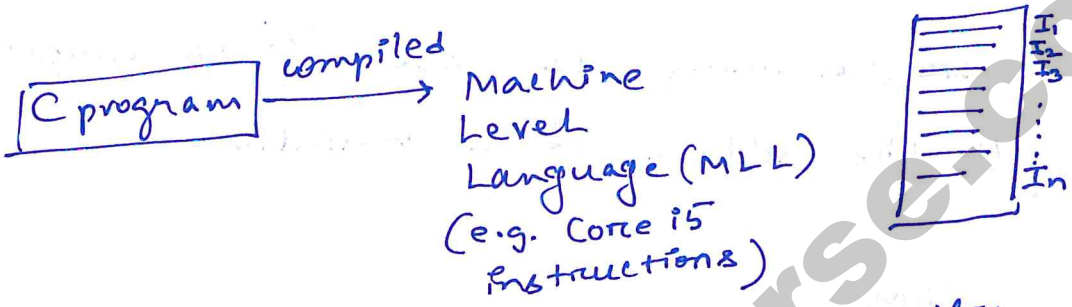
A program is a sequence of instructions that are there in machine level language. The CPU reads this instructions from the memory and executes one by one in a sequence.



Note: John Von Neumann came up with this concept in 1940s. It was revolutionary way of thinking about it.

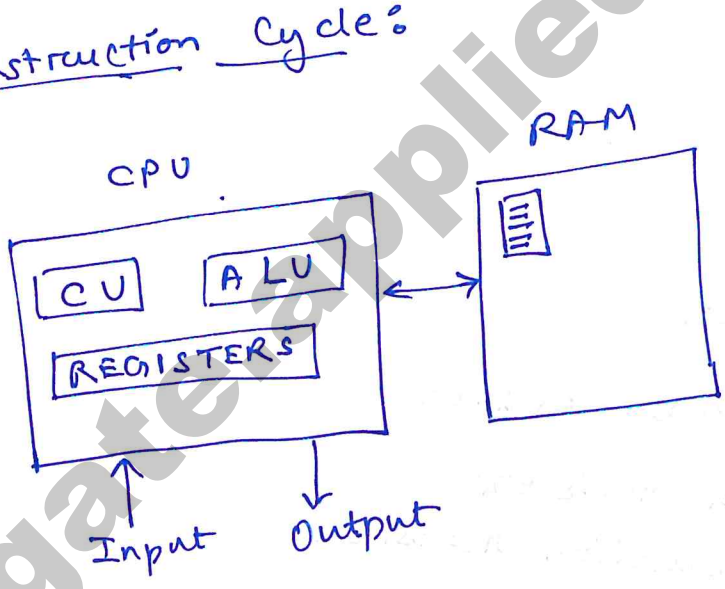
In the CPU, there are also other components like the registers.

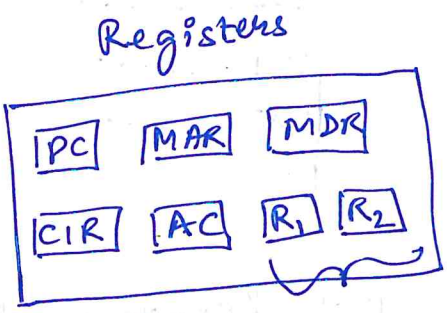
Note:



These sequence of ~~instructions~~ specific to a microprocessor is stored in the RAM.

Instruction Cycle:





We can store values of variable in registers

A register is basically a bunch of memory which is part of the CPU chip itself, therefore, it is much faster to read data from the registers when compared to reading data from the RAM.

How does the CPU execute a bunch of instructions? That is what is explained by an Instruction cycle.

RAM is basically a sequence of addresses. We can store data in RAM and we can access them using addresses.

In C programming, we use '&' or ampersand to get the address of a variable.

RAM

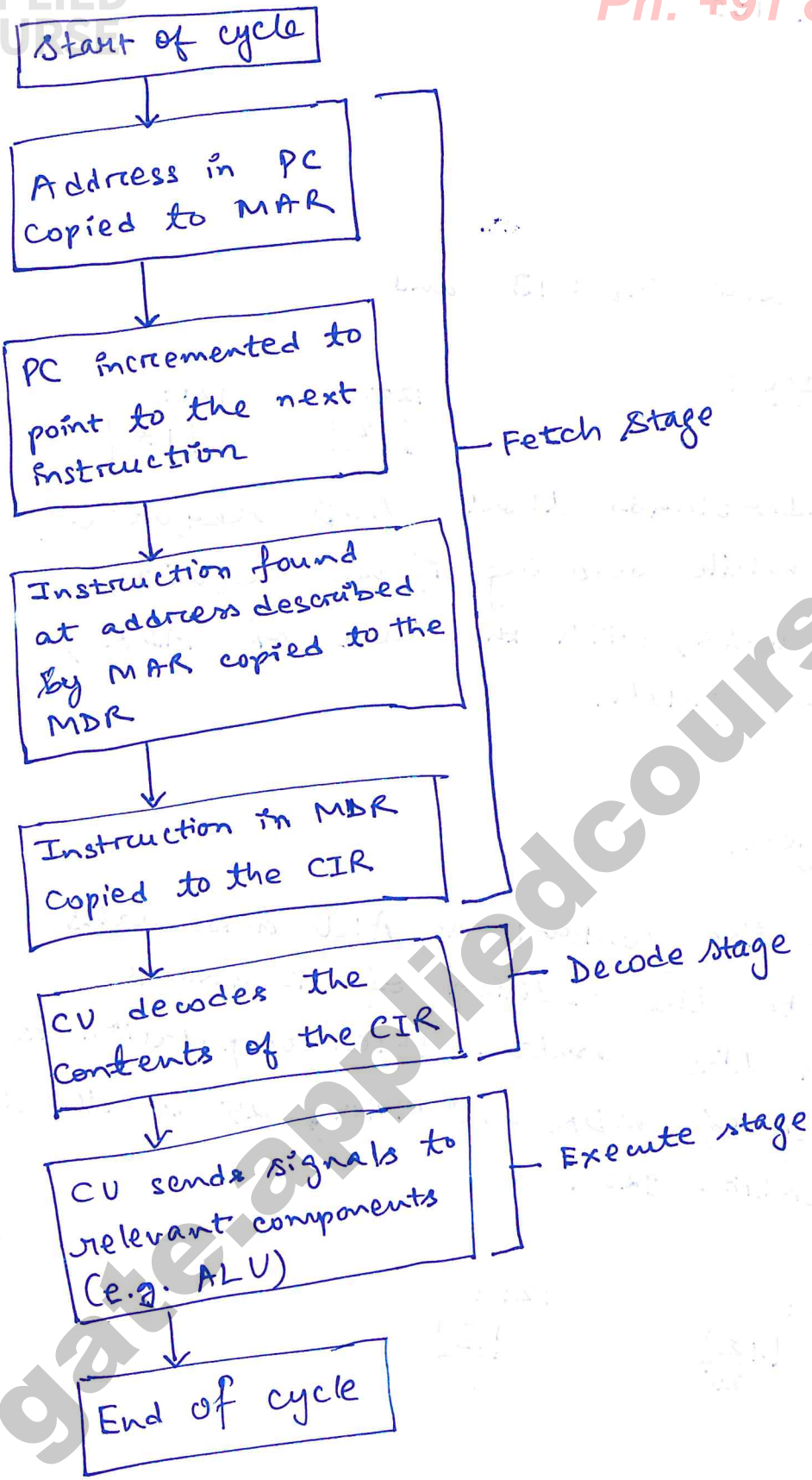
100 :	
101 :	
102 : ADD (120) (121)	← MLL instruction
103 :	
104 :	
105 :	

PC : Program Counter

MAR : Memory address register

MDR : Memory Data register

CIR : Current Instruction register



In Fetch cycle :

Let, PC ~~102~~ 103

MAR : 102

MDR : ADD

CIR : ADD

Let, R_1 : 12 and R_2 : 13

In Decode cycle :

120 : 12 | 121 : 13

The CU understands that ADD requires the ALU. While decoding if we require additional data, with the help of MDR, we perform fetch.

In Execute cycle:

The CU gives the control to ALU and asks it to compute the addition of 12 and 13 and return the result. After computation is over, with the help of MDR and MAR, 25 is stored in the location 120.

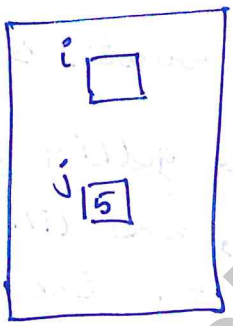
120 : ~~12~~ 25 | 121 : 13

Interrupts: We take a C program example to understand interrupts.

```
Line 1:  main() {
Line 2:      int i;
Line 3:      int j = 3+2;
Line 4:      j = j*2;
Line 5:      scanf("%d", &i);
Line 6:      i = i/j;
Line 7:      printf("%d", i);
Line 8:  }
```

gets converted to → MLL instructions

In RAM:

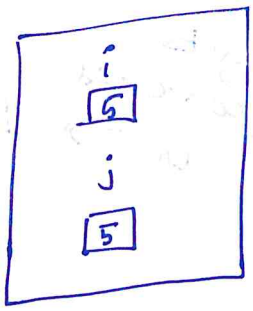


The lines 2, 3 and 4 and their corresponding machine level instructions gets executed.

Note that, each of lines 2, 3 and 4 will individually form (translated) into multiple instructions (MLL instructions) before they get executed.

Line 5 tells the CU and the CPU that we have to wait for a input from the keyboard. The becomes idle as it waits for the input. This creates an interrupt. Interrupt is basically interrupting the flow where in we are expecting some input to arrive into the system.

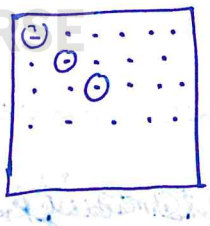
Therefore, the flow of execution has been interrupted waiting for something external to the CPU to work out.



Say the user entered input as 5. Therefore 5 gets copied into i.
Next line 6 will be executed without waiting. Line 7 has to print to the monitor but even here we can have interrupt. Say, there are a lot of things that are getting printed on the monitor right now, and till the time line 7 is printing i on the monitor, we want to wait. There can also be cases where outputs don't create interrupts.

History of Operating Systems:

(1) 1st-generation (40's & 50's): No O.S; punch cards were used as there was no secondary storage. People used to write codes in machine level language, punch holes into punch cards.



Every line would represent some set of instruction.
They would punch a hole and use electronic circuitry to read these and convert into a program.

Even the 1st generation systems were using the stored program because program was now stored on a punch card. And loaded into the memory for processing. Main memory was in the form of magnetic drum.

② 2nd - generation (60's and 70's): NO O.S; People started using magnetic tapes as the permanent way to store data. An example of magnetic tapes is walkman cassette tapes. Data would be copied from magnetic tapes to the RAM and then processed. So, instead of punch cards, we can now store more programs on the magnetic tape. This era was called as batch processing era.

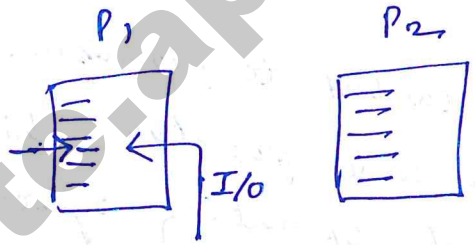
③ 3rd - generation (80's and 90's): O.S started to flourish. And also the disk technology became popular. We will focus on 3rd generation of operating systems in this course. O.S like MS-DOS, Unix, early Windows OS, early Linux OS etc, were built in this generation. Here we started to have the concept of Hard disk. The RAM size was increasing and the modern O.S were being designed in this age. Companies like Microsoft and Apple started during this era. Uniprogramming and multi-programming O.S concepts became popular.

④ 4th generation: (since 2000's) These are OS like Network OS.

Also called distributed OS, where we have hundreds of computers connected in the network. We also see Real time O.S (RTOS) like in IOT, mission critical applications like satellite launch. We even see parallel processing O.S, etc. We usually study about these O.S in a graduate level or research level.

Uniprogramming vs Multiprogramming

In uniprogramming environment, one program is executed at a time. It is a huge wastage of resources like CPU.



Suppose CPU is executing program P_1. At some instruction the program P_1 gets interrupted and is waiting for some keyboard input etc. In the uniprogramming environment, CPU is still running P_1 and program P_2 is waiting to get CPU.

The whole idea of multiprogramming is can we increase the efficiency by having multiple programs executed simultaneously?

In multiprogramming environment, the RAM will have multiple programs and CPU will allocated to another program if the currently running program gets blocked.

All the modern OS implements multiprogramming, e.g., Linux, Windows 10, Android etc.

Example of uniprogrammed OS is MS-DOS.

The shift from uniprogramming to multiprogramming happened as in the last 30-40 years the size of RAM has increased and the cost of RAM has decreased. Similarly, CPU speed and disk size has increased.

Multiprogramming vs Multitasking

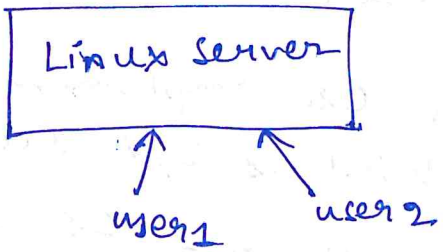
Multiprogramming is a Unix terminology.

Multitasking is a term used by Windows as a marketing tool. Both of these terminology actually mean the same thing.

Types of Multiprogramming :

(1) Single User vs MultiUser :

In a Linux server or a Unix server, multiple users can simultaneously login and use it.



In 70's and 80's, since everyone did not have personal computer, the companies would have powerful computers where multiple people can login, submit their work which will be executed and results would be returned back.

Incase of a single user O.S, only a single user can login and use the system.

(2) Preemptive vs Non-preemptive :

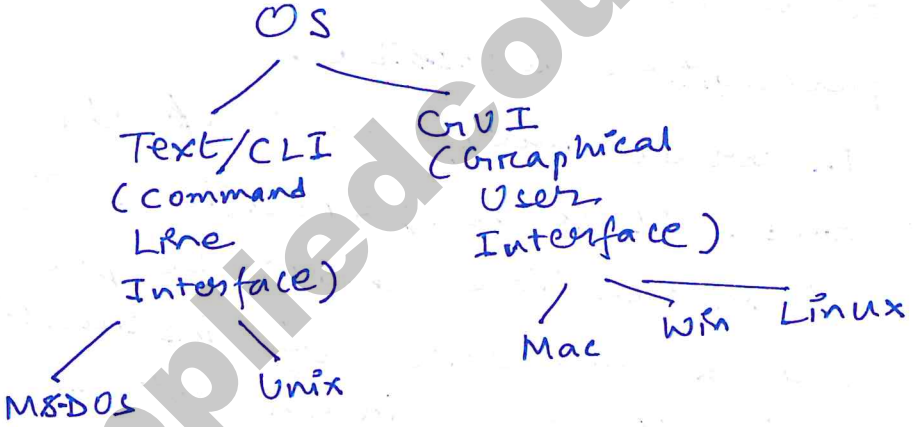
In a preemptive system, the O.S can force a process to give away control of some resources like CPU.

Preemption Basically means forcefully removing something.

In a non-preemptive system, the process has to give away control itself. The process might give away upon completion or if there is an I/O event.

Examples of Non preemptive O.s are windows 3.0 and windows 3.1. And examples of preemptive O.s are Windows 10, Linux, Android etc.

Note: There are also Text based Os and GUI based Os.



The GUI based Os also have support for CLI like command prompt (windows) and terminal (Mac, Linux).

Modes of CPU execution + FORK()

Modes of execution on a CPU:

There are two modes of execution — user mode and the kernel mode.

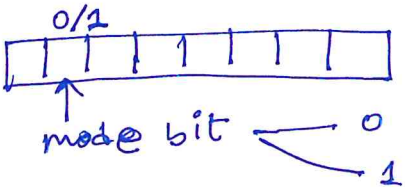
The kernel of an O.S is basically the core piece of the software which enables the O.S.

Every user submitted / user given program runs in the user mode. In a user mode, preemption is possible.

In kernel mode, most of the key functionality runs. It is non preemptive. If a sequence of instructions cannot be preempted, then such a sequence of instructions are said to run in an atomic fashion or atomic execution of instructions.

Mode bit in Process Status Word (PSW):-

For every process, there is a data structure that the O.S maintains called the Process status word (PSW). Within the PSW, there is one bit called mode bit.

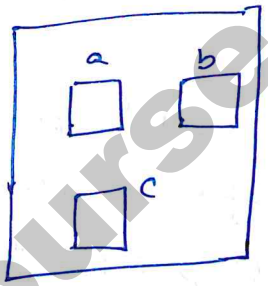


Special Data structure

The other bits hold important information too.

E.g.1)

```
#include <unistd.h>
#include <stdio.h>
main()
{
1. int a,b,c; — user mode
2. c=a+b; — user mode
3. fork(); — kernel mode
4. printf("Hi"); — user mode
}
```

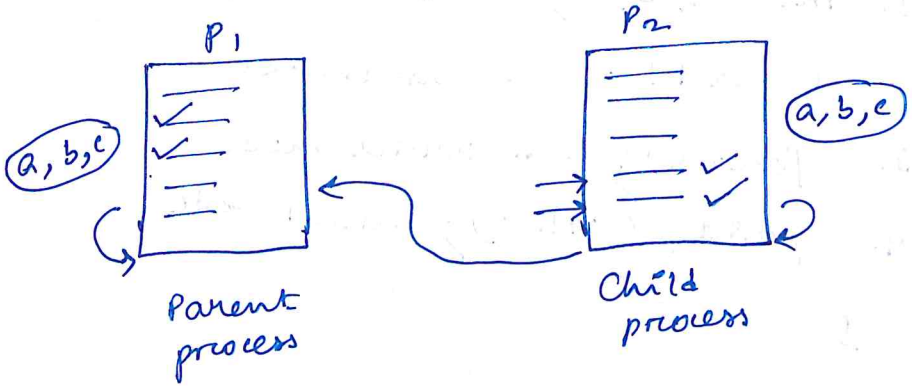


We declare 3 variables, ie., in the memory we have 3 places to a, b, c respectively.

fork() is often called a system call or sys call. The function is declared in unistd.h. This function eventually calls the O.S. Fork creates a new process. It copies the instruction and the variables.

While executing line 3. fork(), the O.S takes the control and it goes into kernel mode and takes all the instructions and memory associates with the parent and creates a copy (child process). And control is transferred to child process and child process execution begins on whatever is there after the fork() line (line 4). After finishing the execution of the program (child), control is

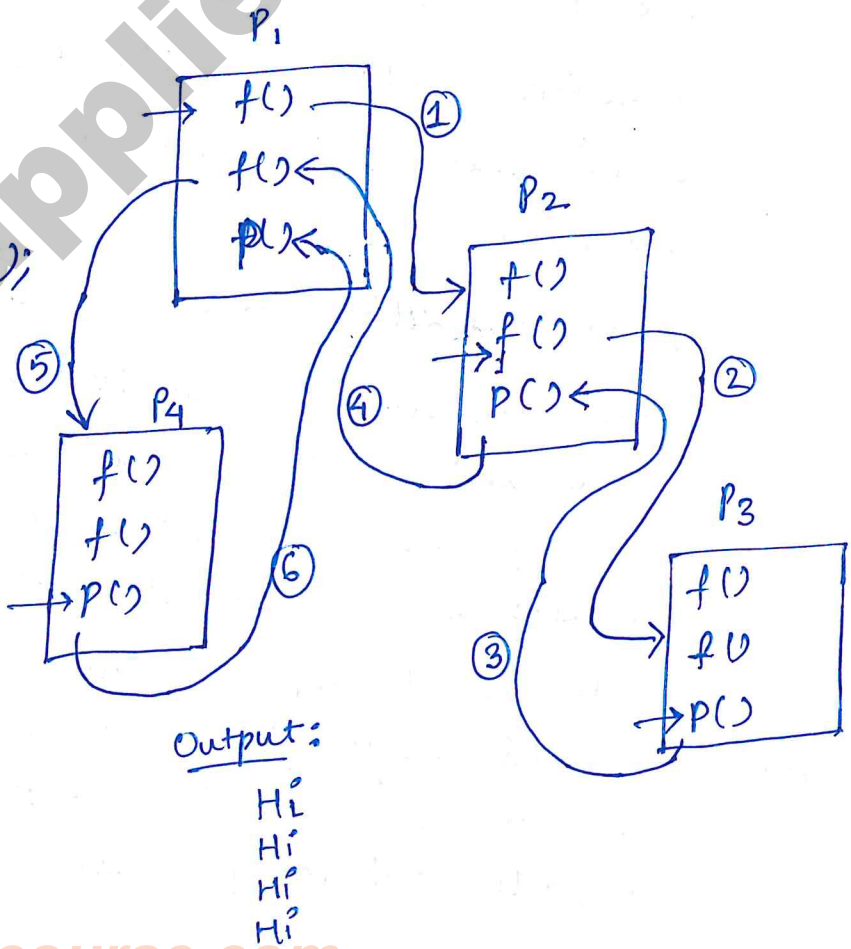
transferred back to the parent process (to the line after the fork()).



The child prints "Hi" and then when control is transferred back to the parent process, the parent prints "Hi".

E.g. 2)

```
#include <unistd.h>
main() {
  fork();
  fork();
  printf("Hi");
}
```



Output:

Hi
Hi
Hi
Hi

For 2 forks, 2^2 processes got created, including the parent process.

Similarly, if we have n forks, we will create 2^n processes including the 1st parent processes.

Eg 3)

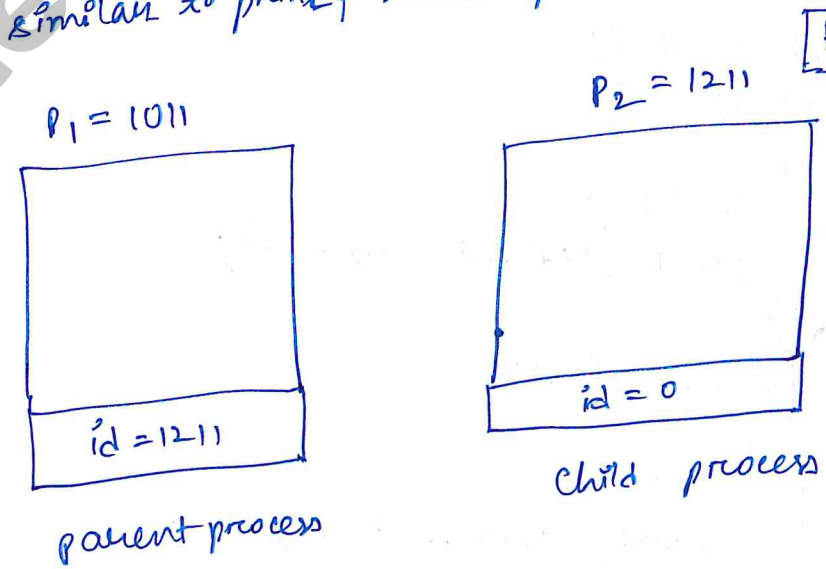
```
main ( ) {
int id;
id = fork ();
if ( id == 0 )
{
  ==
}
else if ( id > 0 )
{
  ==
}
(when id < 0) {else
{
perror ("Unable to fork");
}
```

executed by the child process

executed by the parent process

executed when O.S could not fork a new process.

similar to printf which prints errors.



Let the process id of parent process $P_1 = 1011$ and the process id of the child process $P_2 = 1211$.

The variable id in the parent process gets the value from value returned by the fork(). Basically, the process id of the child is updated in the variable id of the parent.

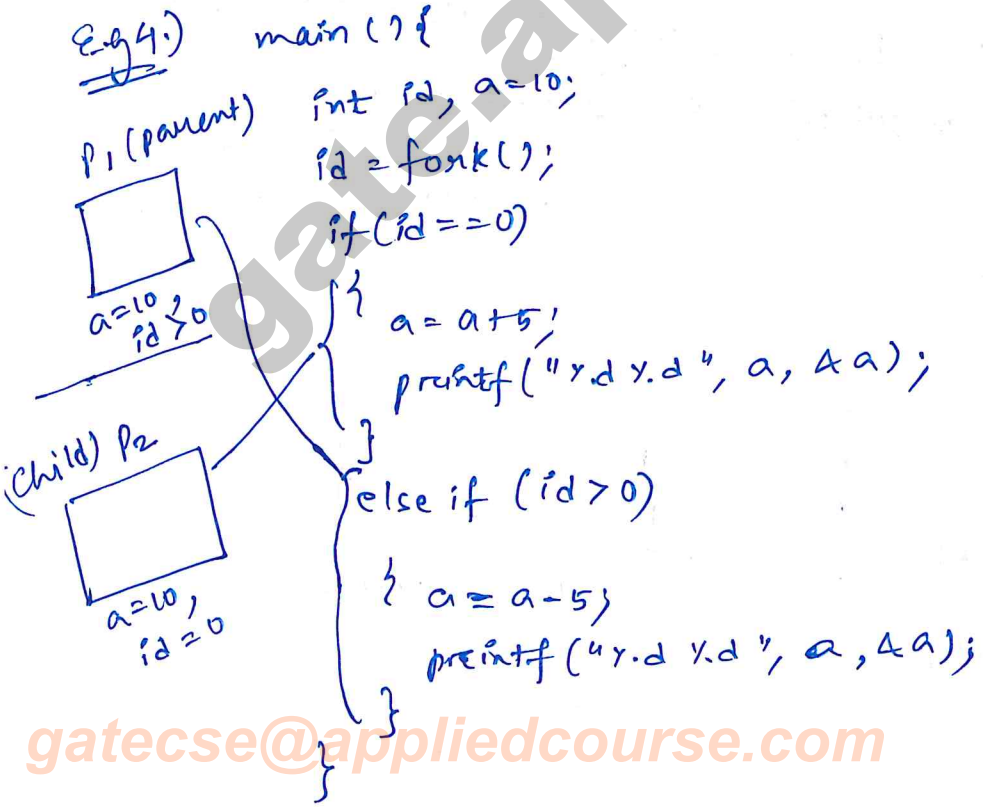
∴ id in parent will become 1211.

While variable id in the child will be updated to 0.

Note: The process ids are always greater than 0.

PID > 0

Eg 4:)



If u & v are values printed by parent and x & y by child, then

a) $u = x + 10$; $v \neq y$

b) $u = x + 10$; $v = y$

c) $u + 10 = x$; $v = y$

d) $u + 10 = x$; $v \neq y$

u v
Child : 15 , &a

parent : 5 , &a
x y

fork() returns 0 in the child process and process id of the child process in parent process.
In child process, $a = a + 5$ is executed.
In parent process, $a = a - 5$ is executed.

$\therefore x = u + 10$.

The physical addresses of 'a' in parent and child must be different. But the program accesses virtual addresses (assuming we are running on an OS that uses virtual memory. The child process gets an exact copy of parent process and virtual address of 'a' doesnot change in child process. Therefore we get same addresses in both parent and child. $\therefore v = y$. Option (C) is correct.

Solved Problems

Q. The following C program is executed on a Unix/Linux system.

```
#include <unistd.h>
int main()
{
  int i;
  for(i=0; i<10; i++)
    if(i%2 == 0)
      fork();

  return 0;
}
```

0 9

0, 2, 4, 6, 8 } 5 times fork() } fork is called

The total number of child processes created is 31.

Since fork() will be called 5 times, we will have 2^5 processes in total and $(2^5 - 1)$ child processes.

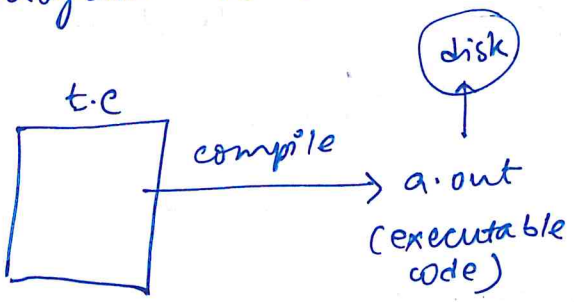
And $2^5 - 1 = 31$.

Process Management - Basics + Scheduling algos

What is a Process ?

Process Management:

Program vs Process



A process is a program under execution. A program after it is compiled is stored on disk (secondary storage).

A process is stored in the main memory (RAM) with resources allocated to it.

A single program could create multiple processes using fork() or spawn().

A process is basically a unit of execution.

A program is a passive entity as it is not being executed, it is sitting on the disk.

While process is an active entity, because the process is being executed right now.

Process (in memory)

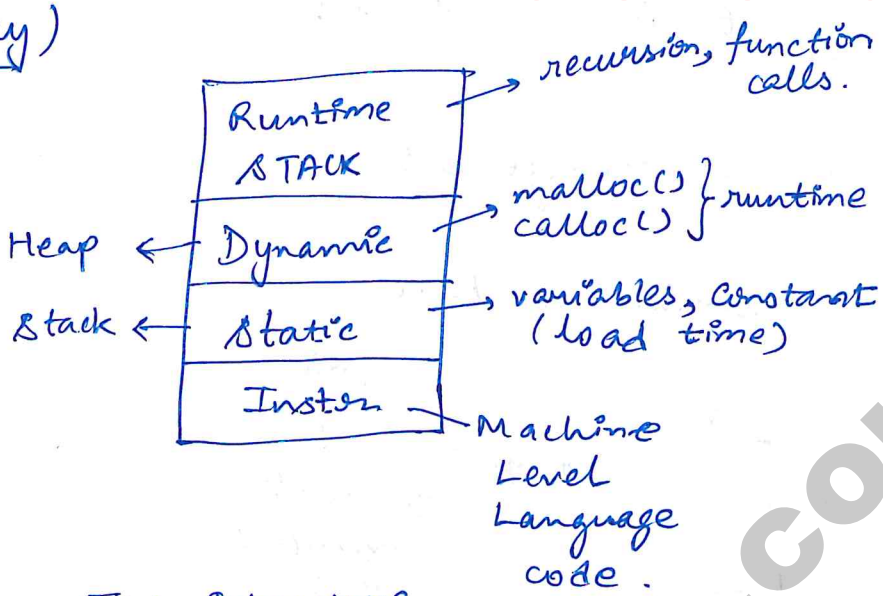


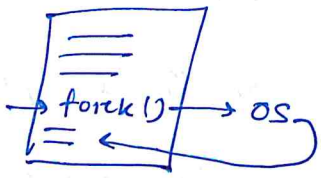
Fig: Structure of the process in RAM.

Operations performed on a process:

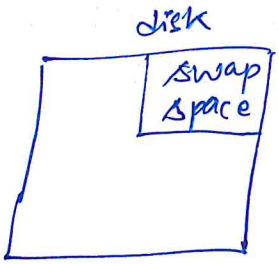
1. Create : Allocate resources
   Devices initialized.

2. Schedule : on CPU

3. Block : I/O, System calls.



4. Suspend : Process suspension means remove the process from RAM and store it temporarily into Disk (swap space).



5. Resume: start scheduling it for resources like CPU to be executed again. (Revoke or swap into memory from disk).

6. Terminate: Resource Reallocation.

Attributes associated with the process:

1. Process identification: process id (PID), parent Pid (PPID), Group id (gpid).
2. CPU related: program counter (PC), General purpose registers (G.P.Rs), priority, states, type.
3. Memory related: memory limits, page tables
4. File related: file ids, list of open files
5. I/O related: list of open devices (e.g. keyboard)
6. Protection related: mode (in PSW), permission

All the above attributes are stored in a table called PCB (process control block).

Every process has a PCB. A PCB is a data structure. It is a big look up table.

PCB is also called as process descriptor.

Process context is the content of the PCB.

And process context is also called process environment.

Process States & Queues & Schedulers

Process states :

There are 3 states diagram as well as 5 state diagram.

3 states — Ready, running, block/wait

5 states — Ready, running, block/wait, suspend ready, suspend block.

It is to be noted that the 'new' state and the 'terminated' state are not counted in the number of states.

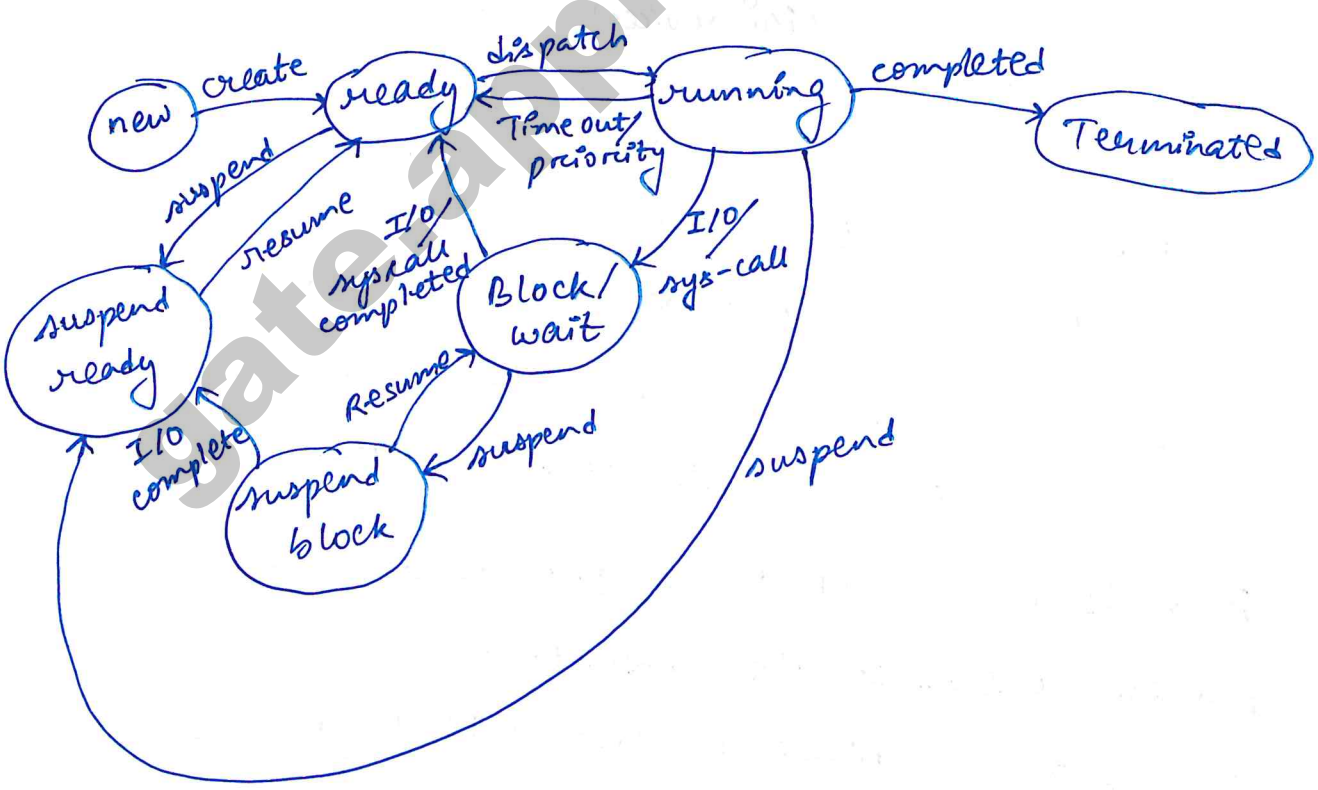


Fig: 5 state diagram

Initially the process is new. It gets created. Then it reaches the ready state. By ready state, we mean is, the process is now in main memory or RAM.

The create operation takes all the instructions of the program, assigns all the static variables and some space for recursion calls and dynamic memory allocation and places the process in RAM itself. Then the process reaches the ready state.

There is a module in the O.S called dispatcher, which takes a process in ready state and moves it to the running state. Running state means CPU (one of the most important resources) is assigned to the process.

Also if the RAM is running out of memory, the process might have to be suspended by moving it from ready state to suspend ready state. When in suspend ready state, the process is actually in secondary storage as there is no space in RAM.

While in running state, the process may complete its execution and reach terminated state. Resources like memory, CPU, I/O devices, etc are given up when the process reaches the terminated state.

While in running state, the process may have to give up CPU due to priority, when a new higher priority process is scheduled.

The process may have to give up CPU due to timeout where, some algorithms in O.S tells the process to use the CPU for a fixed time limit. And when the process runs out of time the process is moved to the Ready state.

Again while running the process in CPU, the process can get blocked on an I/O call (like waiting for some input from the keyboard).

Also, the running process can get blocked on system call (like fork). On getting blocked the process state changes from running to Block/wait state.

Now, there is one possibility that while running on the CPU, there is a high priority process arrived in the Ready queue and gets scheduled. But there is no space in the RAM. Then the process running in the CPU directly gets suspended to the secondary storage and the state becomes

suspend ready.

When process is in the Block/wait state, it is waiting for the completion of system call or some input to happen. When these events are satisfied, the process is moved from block/wait state to the ready state. There is also another possibility that I/O may have happened or may not have happened, but there no space in the RAM, then, from block/wait state the process is moved to the suspend block state as the process gets suspended. Suspend block means the process is in the secondary storage. If memory is available the process state can change back to block/wait state.

But while in suspend block state, if I/O completes, then the state of the process changes to suspend ready. In the suspend ready state, the process is ready to be resumed and given space in RAM (ready state)

Swap space :

A computer has sufficient amount of physical memory but most of times we need more, so we swap some memory on disk. Swap space is a space on hard disk which is a substitute

of physical memory. Swap space helps the O.S in pretending that it has more RAM than it actually has. The interchange of data between virtual memory and real memory is called swapping.

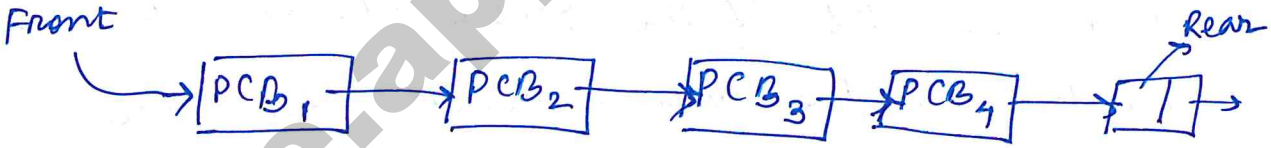
Scheduling Queues:

- Ready Queue
- Block Queue
  - Device Queue
  - Event Queue

A process can be uniquely identified by its PCB.

Ready Queue:

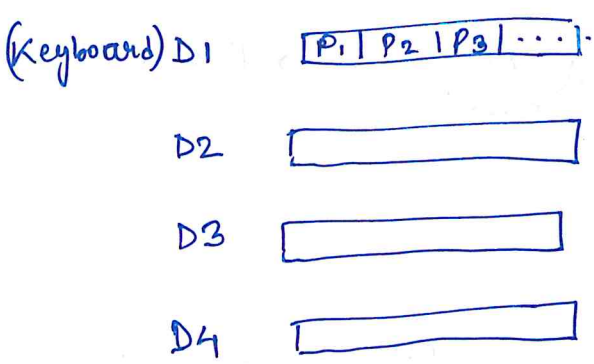
Queue of PCBs of process which are in Ready state.



Event Queue:

An event can be anything. It can be a software event like completion of execution of the child process for the control to be transferred back to the blocked/waiting parent process (on fork() sys call).

Device Queue:



Each of these processes are waiting for keyboard input.

Note: All the queues are stored in the RAM.

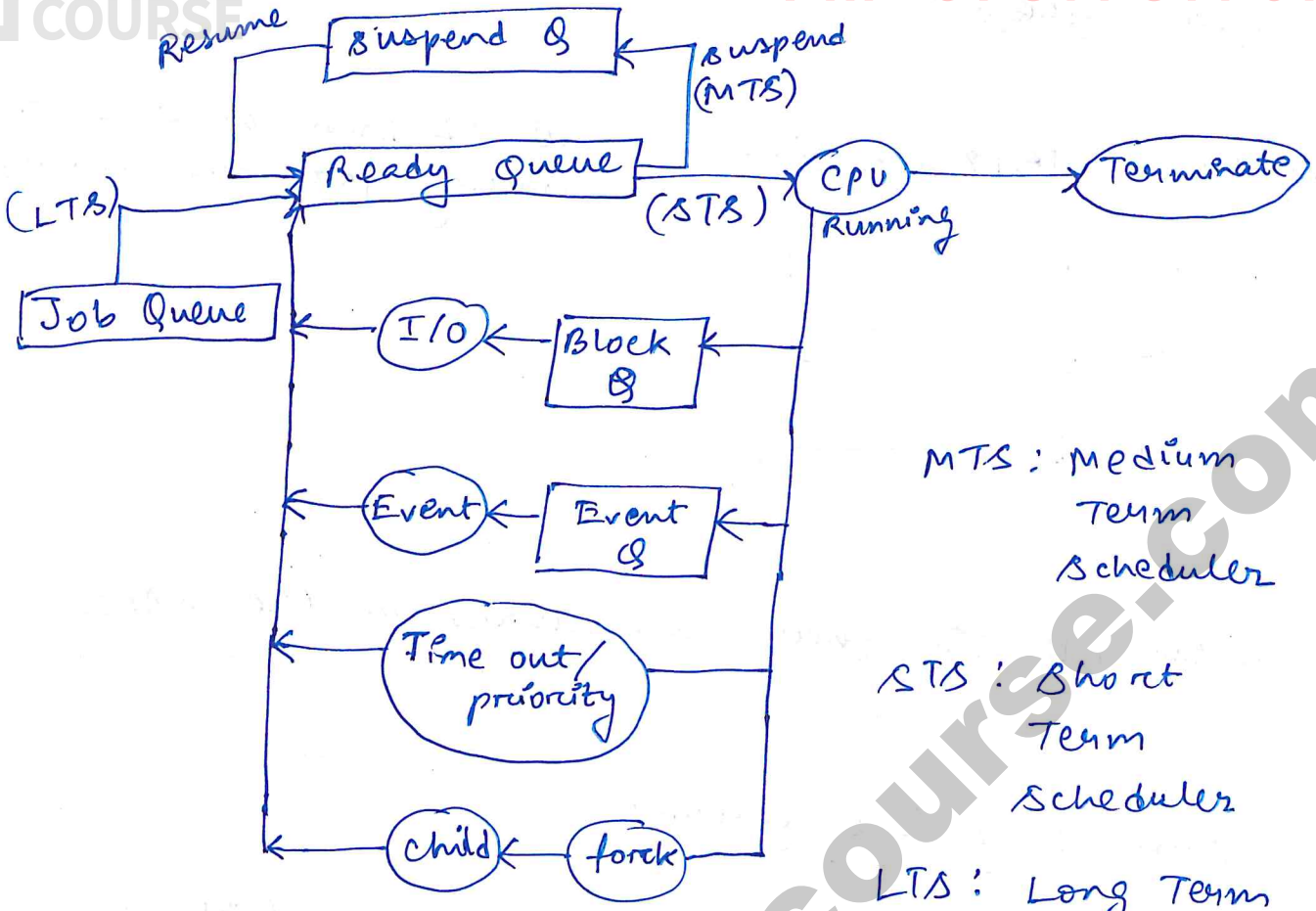
Suspend & Job Queue:

Suspend Queue has all the processes which are in suspend state. Suspend queue is stored in secondary memory.

Job Queue / Input Queue consists of all the programs waiting to be loaded in the main memory into a process.

Process State Queuing Diagram:

When we have many processes which are lined up in the queues, we will look at process states in the queuing diagram perspective.



MTS : Medium Term Scheduler

STS : Short Term Scheduler

LTS : Long Term scheduler

The time that each process gets on the CPU is quite small, therefore it is called short term scheduler.

The time that each process spends being in the ready queue is quite long. Therefore it is called long term scheduler.

The short term scheduler / CPU scheduler will have some algorithm using which it will decide which of the process (from ready queue) it will give CPU to.

E.g: Priority scheduling algorithm is a very popular algorithm used by operating systems like Android. Every process has a priority. A call can have much higher priority while reading from a browser, may have much lower priority.

Once the short term scheduler picks the process to be scheduled, the dispatcher gives the process control to the CPU.

From running state, the process can go to the terminate state. Or, it could go into one of the device queues or event queues. On completion of the I/O or the event, the process will be moved to the ready queue.

While a process is running in the CPU, a higher priority process may arrive, then the currently executing process will be moved back to the ready queue.

While a process is running in the CPU, there maybe a fork call, due to which the parent will get blocked till the child completes.

Now, while the process is in ready queue, the system can run out of memory, the process can be suspended. When suspended, the process goes to suspend ready queue.

There are two types of suspend queues:-

(i) suspend ready queue
(ii) suspend block queue (Device block queue)

I/O bound vs CPU bound processes:

I/O bound [less CPU; more I/O] → Disk (Reading 2GB files from the disk and copy to another location on disk)

CPU bound [less I/O; mostly CPU resources] → lots of CPU → scientific computation → logical & arithmetic

There can also be third type of process that is neither totally I/O bound nor totally CPU bound. It has a good mix of both CPU B.Ts and IO Burst times.

Dispatcher & Context switching:

The dispatcher gives control of the CPU to a process selected by short term scheduler (STS).

The roles of dispatcher:

① On completion of a process:

A process is defined by its PCB. Once the process completes, the PCB corresponding to the process is deleted (destroyed).

Now, the CPU becomes idle. Dispatcher informs the STS to give the dispatcher a new process.

The STS gives a new process to be executed on the CPU. The dispatcher copies the process attributes (e.g., PC, GPR, ..) from the PCB into the CPU registers and the dispatcher gives the control to the control unit of the CPU to start the execution of the process.

② A process running on CPU requires I/O:

The dispatcher immediately realizes that process has to go from running state to blocked state. Therefore dispatcher saves the PCB and move the corresponding PCB into the block queue.

Then the dispatcher waits for the STS and when STS gives a new process, then the dispatcher loads the new process into the CPU.

Context switching: The current execution context gets switched from one process to another. The dispatcher performs context switching.

③ Process preemption:

Process preemption can happen due to priority, when there is a higher priority process arriving in the system while there is a low priority process running in the system. In such cases, STS forces preemption. The dispatcher saves the current state in PCB, removes the register values and loads the new state of the process in the CPU. Then dispatcher gives the control to the CPU.

Q). m processes and n CPUs.

What is the lower & upperbound on # processes in

	Lower	Upper
Ready :	0	m
Running :	0	n
Block :	0	m

Probably none of the m processes are in the ready state. All of them could be in the suspend state. Therefore minimum number of processes in the ready state is zero.

The maximum number of processes that could be in the ready state is m. If we have lots of RAM, then all the m processes can be in the memory.

If all the m processes are I/O bound, then all the m processes could be in blocked state, and they are waiting the I/O requests to be serviced. Therefore the minimum number of running processes is 0.

Since it is not mentioned in the question that it is a multicore processor, we will

consider it is a unicore (single core processor) system, where one process can run at any time.

Therefore, at max, we can have n running processes.

The minimum number of blocked processes could be zero because if all of the processes are completely CPU bound and they don't require any I/O, they will never be blocked on it.

The maximum number of processes that can be blocked is m.

CPU Scheduling (STS) timings:

The task of the STS is to pick which process in the ready queue should be given control of the CPU.

The goal of the STS is to maximize the CPU utilization. It wants to ensure that the processes complete their execution as fast as possible.

The following time related metrics help us measure the max. CPU utilization :

(i) Throughput : It is the amount of work that is completed per unit time. This is basically the number of processes completed per unit time by the scheduler. The short term scheduler wants high throughput.

Throughput = n/L ; n = # of processes ; L = Schedule Length.

(ii) Arrival / Admission / Submission Time (AT) : (A_i) Σ i^{th} process

This is time at which the process arrived in the ready queue.

(iii) Wait time (WT) : < CPU wait time : We want to decrease it (↓)
< I/O wait time : STS doesnot control it

A CPU wait time is the time that the process waits in the ready queue without being assigned a processor or a CPU to work on.

The STS is bothered about CPU wait time only. STS can't control how much time a process is blocked in a blocked queue / event queue. It can only control how much time a process is waiting in the ready queue.

STS wants to reduce the CPU wait time (WT).

(iv) Burst / Service Time (BT) : (x_i) :

This is the time duration for which the process is in the execution / running state.

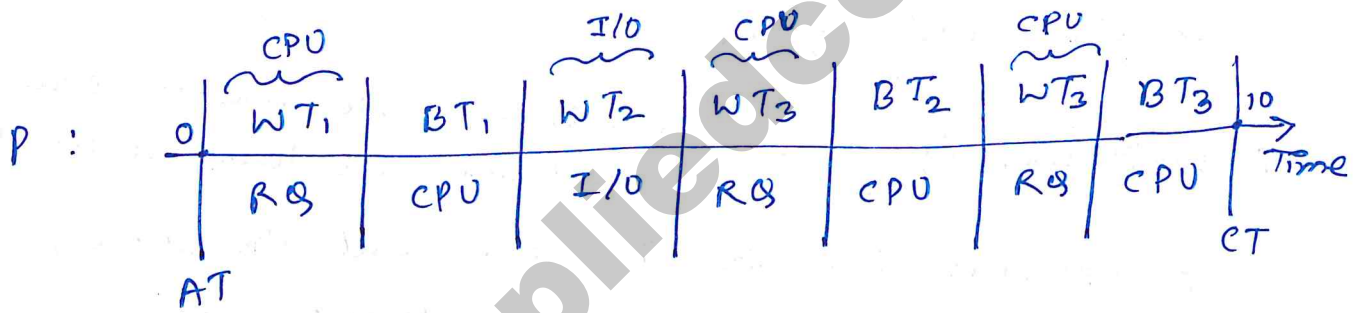
The process is assigned a processor (CPU) and it is utilizing the CPU. We want to increase the Burst time (B.T).

(v) Completion time (CT) : (CT) :

It is the time when the process completes.

(vi) I/O Burst wait time :

This is the wait time spent Blocked on I/O.



(vii) Turn-around Time (TAT) : CT − AT

The TAT for the above process P is

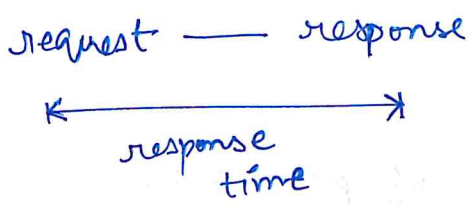
TAT = CT − AT
= 10 − 0
= 10 units of time

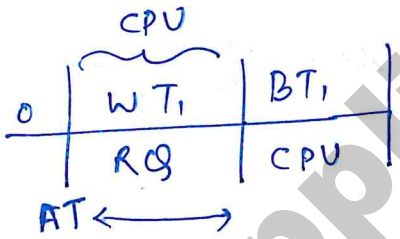
(viii) Response Time :

Suppose we have a process in the ready queue waiting for the STS to assign the CPU (resource) to the process.

After sometime based on the STS algorithm and dispatcher, the CPU is assigned to the process. The first time the CPU is assigned to the process, it is called response.

When a process comes to the ready queue, it requests access of the CPU and the first time it gets a response, this whole period is called the response time.





We want to reduce the response time.

(ix) $\underline{\text{Avg. TAT}} = \frac{1}{n}\sum_{i=1}^{n}(C_i - A_i) = \frac{1}{n}\sum_{i=1}^{n} TAT_i$

It is the simple mean / average of the TAT of all the processes.

We want to reduce the average TAT and TAT_i.

(x) Weighted TAT = $\frac{C_i - A_i}{x_i} = \frac{TAT_i}{x_i}$: How much of the total time is spent on the CPU.

TAT_i = Total time the process spends

x_i = B.T = total time the process spends executing.

(xi) $WT_i = (\underbrace{C_i - A_i}_{TAT_i}) - x_i$

(CPU wait + I/O wait)

where WT_i = wait time of process i.

(xii) Avg. WT = $\frac{1}{n} \sum_{i=1}^{n} (C_i - A_i - x_i) = \frac{1}{n} \sum_{i=1}^{n} WT_i$

(xiii) Schedule : $\underbrace{\{P_1, P_2, \ldots P_n\}}_{\text{set of } n \text{ processes}}$

We can sequence/schedule in any order we want.

$P_1\ P_3\ P_2\ P_4 \ldots\ldots$ } one possible sequence

For n processes, we can have n! schedules.

$\underline{n} * \underline{n-1} * \underline{n-2} * \ldots\ldots$ } n!

↑ n possibilities for first place.

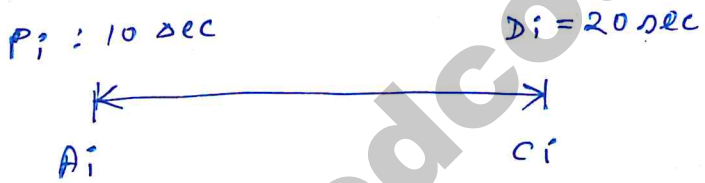
(xiv) Schedule Length (L) = Total time to complete all processes in a schedule.

$L = \max(C_i) - \min(A_i)$

= maximum complete time − minimum arrival time.

(xv) Deadline (D_i)

It is the timestamp before which the process should complete.



Deadline overrun:

If $C_i > D_i$

then $C_i - D_i > 0$.

It means the process didn't complete before the deadline. Such a case is called deadline overrun.

Deadline underrun:

If $C_i < D_i$

then $C_i - D_i < 0$

The process execution completes before the deadline associated with the process.

Scheduling Algorithms : FCFS — Non preemptive

A.T : criteria.

(First come First served)

Assumptions :

(i) No I/O (Input-Output)

(ii) Context switching (C.S.) Time is negligible

E.g 1)

Process id	A.T	B.T	ST	CT	TAT	WT
1	0	4	0	4	4	0
2	1	5	4	9	8	3
3	2	6	9	15	13	7
4	3	8	15	23	20	12
5	4	2	23	25	21	19
6	5	4	25	29	24	20

A.T = Arrival Time

B.T = Burst Time

S.T = Scheduling Time

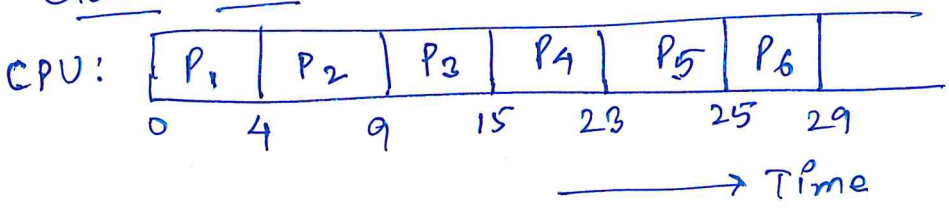
C.T = Completion Time

TAT = Turn Around Time

W.T = Wait time

Scheduling Time is the time at which the process is scheduled the first time.

Gantt chart :



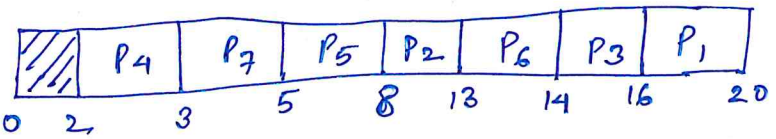
FCFS is a non-preemptive scheduling algorithm. That is, once a process starts scheduling, the O.S doesnot preempt it. O.S lets the execution of the process continue till the time the process completes its execution.

Arrival time is the scheduling criteria. Whichever process has the lowest arrival time will be scheduled first.

E.g 2.) From the following data, calculate the schedule length.

PID	AT	BT	ST	CT	TAT	WT
1	8	4	16	20	12	8
2	5	5	8	13	8	3
3	7	2	14	16	9	7
4	2	1	2	3	1	0
5	4	3	5	8	4	1
6	6	1	13	14	8	7
7	3	2	3	5	2	0

CPU:

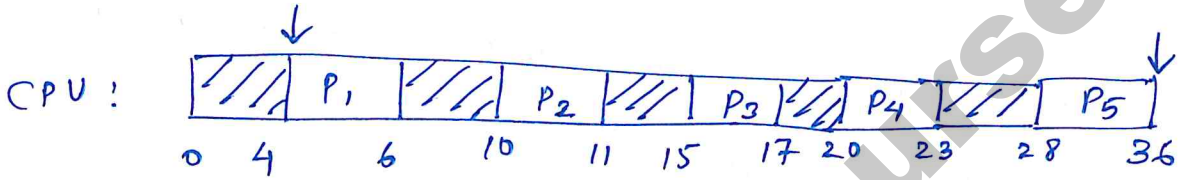


Schedule length (L) = 20 − 2 = 18

= Max(Ci) − min(Ai) = 18

E.g. 3)

PID	AT	BT
1	4	2
2	10	1
3	15	2
4	20	3
5	28	8



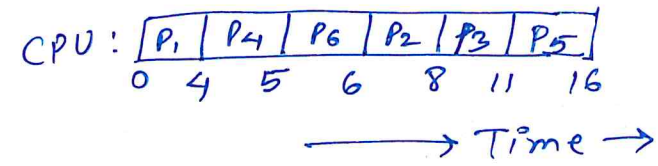
∴ Schedule Length (L) = 36 − 4 = 32

Scheduling Algorithms : SJF & SRTF

Shortest Job First (SJF)

↳ Non preemptive
↳ Criteria : Burst Time

PID	A.T	B.T
1	0	4
2	1	2
3	2	3
4	3	1
5	4	5
6	5	1



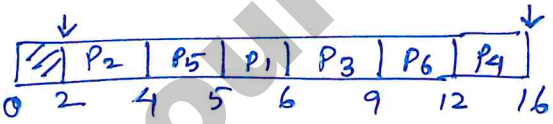
$$\text{Throughput} = \frac{n}{L}$$

The rationale behind the SJF is whichever job we can complete first, let's complete that first, because it will improve the Throughput. Improving throughput is one of the objectives of the scheduler.

E.g.2)

PID	AT	BT
1	5	1
2	2	2
3	3	3
4	2	4
5	4	1
6	3	3

CPU:



$L = 16 - 2 = 14$

If new processes keep arriving with very low B.T requirements, high B.T processes like P_4 will never get a chance to be executed on the CPU. Thus, it will lead to the starvation problem of P_4. It is a major problem in SJF.

Incase where 2 processes have same B.T, the tie breaking mechanism will be mentioned. It can be the lower process id, arrival time etc.

Note: FCFS doesnot suffer from starvation problem.

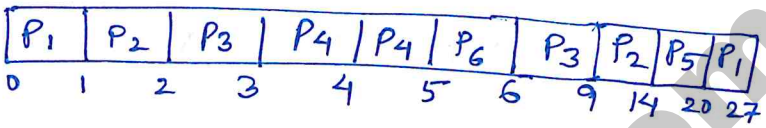
Shortest Remaining Time First (SRTF) = SJF + preemption

↳ Preemptive scheduling algorithm

E.g.1)

PID	AT	BT
1	0	8
2	1	6
3	2	4
4	3	2
5	4	6
6	5	1

CPU:



TAT (P_3) = ?

$= C_3 - A_3$

$= 9 - 2$

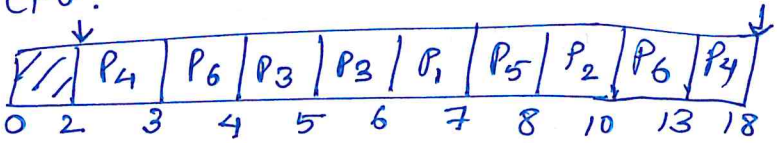
$= 7$ units of time

Note: At every unit of time, we have to look at the possibility that a new process came in whose remaining time left is less than the current process and we have to preempt accordingly.

E.g.2)

PID	AT	BT
1	5	1
2	7	2
3	4	2
4	2	6
5	6	1
6	3	4

CPU:



SRTF also could lead to starvation problem.

SJF:

Advantages	Disadvantages
① Throughput ↑ Avg WT ↓ TAT ↓	① Starvation ② SJF / SRTF / FCFS Assumption ← Burst time is known ∴ We need to predict the B.T.

Predicting Burst Times:

① Using size of process:

old: 100KB → 10 sec } Historical data

new: 100KB → ~10 sec
Estimated data.

B.T = f(size of process)
function

10KB	2 sec
20KB	4 sec
100KB	10 sec
⋮	⋮

This table is kept in memory

It is a very simple methodology, but it doesnot work very well in practise.

Because we can have the same amount of size but we could have very complex loops.

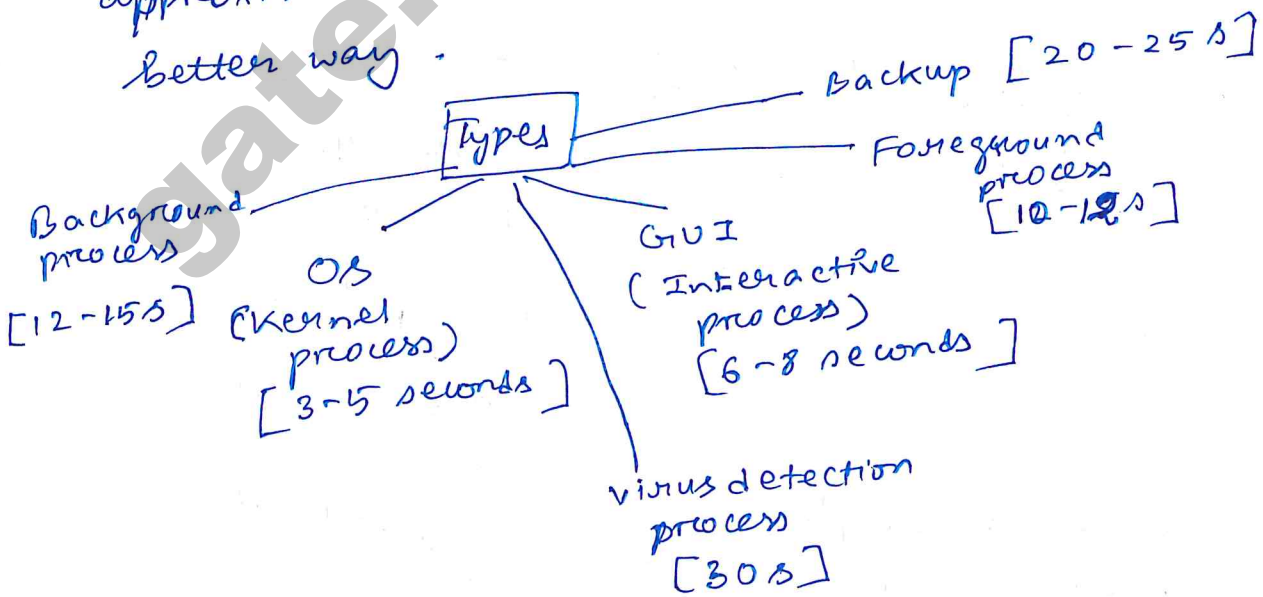
Imagine we have machine level language. The number of instructions, the amount of memory, all of them maybe same between two processes. But one process could have very complex loop and the other could have a simple loop.

In one case it could loop from instruction 1 to 10 while in the other case, it can loop from instruction 1 to 1000.

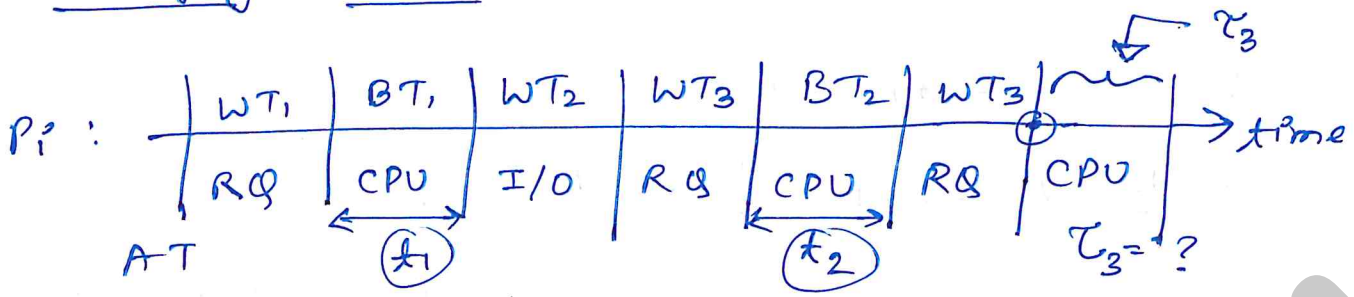
Therefore size itself is not the best predictor. This strategy maynot work that well in practise. But it is a good enough simple approximation.

(2) Using Type of the process:

Type of the process can be used to approximate the estimated Burst times in a better way.



③ Averaging previous CPU-bursts:



t_1, t_2, t_3 = actual BT

τ_1, τ_2, τ_3 : estimated B.T

$$\tau_3 = \frac{t_1 + t_2}{2}$$

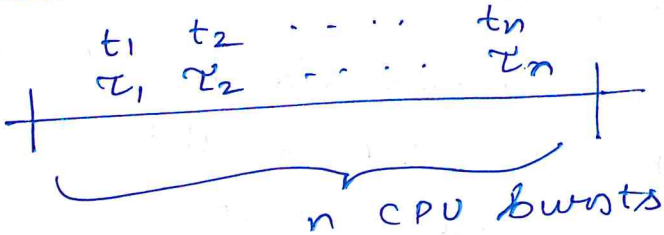
$$\tau_n = \underbrace{\frac{1}{n-1} \sum_{i=1}^{n-1} t_i}_{(n-1) \text{ actual bursts}}$$

Here we do historical averaging for the same process only.

④ Exponential averaging / Ageing-algo:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \qquad 0 \le \alpha \le 1$$

Recursive equation



$\alpha = 0 \Rightarrow \tau_{n+1} = \tau_n$

$\alpha = 1 \Rightarrow \tau_{n+1} = t_n$

$\alpha = \frac{1}{2} \Rightarrow \tau_{n+1} = \frac{1}{2} t_n + \frac{1}{2} \tau_n$

$= \frac{1}{2} t_n + \frac{1}{2} \{ \frac{1}{2} t_{n-1} + \frac{1}{2} \tau_{n-1} \}$

$= \frac{1}{2} t_n + \frac{1}{2^2} t_{n-1} + \frac{1}{2^2} \tau_{n-1}$

$= \frac{1}{2} t_n + \frac{1}{2^2} t_{n-1} + \frac{1}{2^2} \{ \frac{1}{2} t_{n-2} + \frac{1}{2} \tau_{n-2} \}$

$= \frac{1}{2} t_n + \frac{1}{2^2} t_{n-1} + \frac{1}{2^3} t_{n-2} + \frac{1}{2^3} \tau_{n-2}$

LRTF, HRRN & Priority based scheduling

Longest Remaining Time First (LRTF):
[Preemptive scheduling algo; criteria: BT(max.)]

SRTF suffers from starvation where the larger B.T process get starved from getting the CPU.

Therefore one simple fix for it is instead of using the shortest remaining time first, we use the longest remaining time first.

PID	AT	BT
0	0	4 1 1 0
①	0	4 3 2 1 0
2	0	8 4 3 2 1 0

CPU : | P_2 | P_1 | P_2 | P_1 | P_2 | P_0 | P_1 | P_2 |
0 4 5 6 7 8 9 10 11

P_0	P_1	P_2
11	12	13 14

Lots of context switching.

$$\text{Avg. TAT} = \frac{12 + 13 + 14}{3} =$$

LRTF minimizes starvation.

Highest Response Ratio Next (HRRN) : Non preemptive
Criteria : max(RR)

↳ At any time t, the Response ratio of any process is $\left(\frac{WT + BT}{BT}\right)$

We will pick a process for scheduling which has higher response ratio (↑).
(RR)

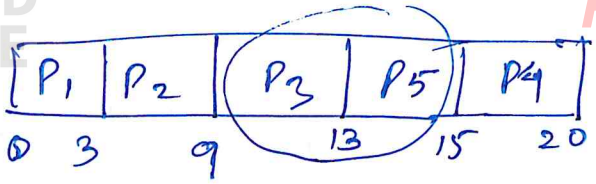
PID	AT	BT
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

SJF : | P_1 | P_2 | P_5 | P_3 | P_4 |
0 3 9 11 15 20

We compare SJF with HRRN.

HRRN also avoids starvation as it takes the W.T into consideration.

HRRN:



At t=9,

$P_3 \leftarrow RR_3 = \frac{5+4}{4} = 2.25$

$RR_4 = \frac{3+5}{5} = 1.6$

$RR_5 = \frac{1+2}{2} = 1.5$

At t=13,

$RR_4 = \frac{7+5}{5} = 2.4$

$RR_5 = \frac{5+2}{2} = 3.5$

→ Preemptive or Non preemptive; criteria: Priority score;
Priority-based scheduling algorithm:
(used in Linux, Android)
(Linux kernel)

Imagine we are reading a webpage, and we receive a call. Since call has a higher priority, the call window should take over the screen.

The priority score can be static and dynamic.

Priority score

[Static] → Priority score is assigned once @ start and is not change

It has an issue. Static priority can result in starvation.

[Dynamic] → Priority score can change during the life cycle of the process.

We can change the priorities the way we want. A dynamic priority scheme could be: If a process is waiting too long, increase its priority, so that it gets scheduled the next time.

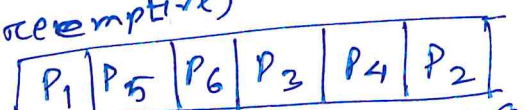
Typically on Linux and Android O/S, priority based scheduling algorithms are non preemptive and both static and dynamic priority scores can be assigned.

E.g. Static priority based scheduling:

PID	AT	BT	P
1	0	4	4
2	1	5	5
3	2	3	8
4	3	6	6
5	4	9	12
6	5	2	10

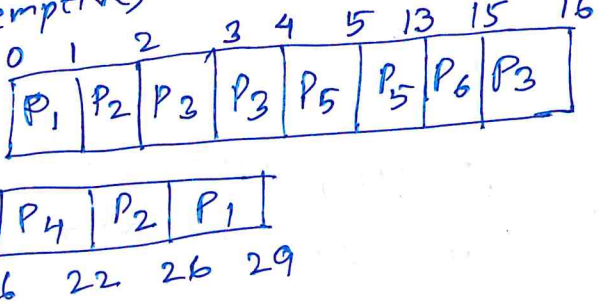
(Non preemptive)

NP:



(preemptive)

P:



Round Robin & Multi-Level Queuing Algorithms:

Round Robin : Preemptive, Time Quanta/slice
(OS, CN)

It is also called as preemptive FCFS

The multiprogramming, multi-user Time sharing O.S use round robin as a strategy to ensure that we don't have starvation.

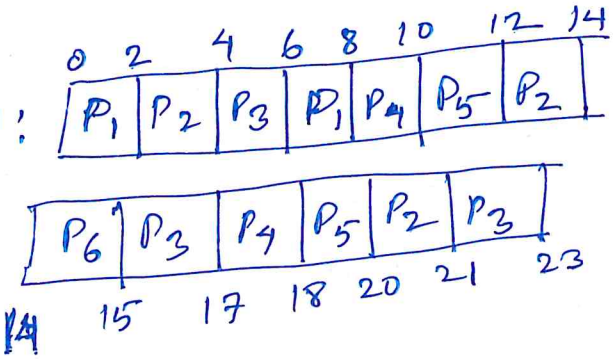
∴ Round Robin Algorithm is starvation free.

PID	AT	BT
1	0	~~4~~ 2
2	1	~~5~~ ~~3~~ 1
3	2	~~6~~ ~~4~~ ~~2~~
4	3	~~3~~ 1
5	4	~~4~~ 2
6	5	1

TQ = 2.

RQ: ~~P1~~ ~~P2~~ ~~P3~~ ~~P1~~ ~~P4~~ ~~P5~~ ~~P2~~ ~~P6~~ ~~P3~~ ~~P4~~ ~~P5~~ ~~P2~~ ~~P3~~

CPU:



Every process will get a fair share of CPU and nothing will get starved.

Performance vs Time Quanta

v.small : lots of context switching; efficiency is low

small : more C.S ; improvement in responsiveness.

large : lesser C.S; responsiveness could suffer

v. large : FCFS

(Q)

PID	AT	BT
A	0	4
B	0	1
C	0	8
D	0	1

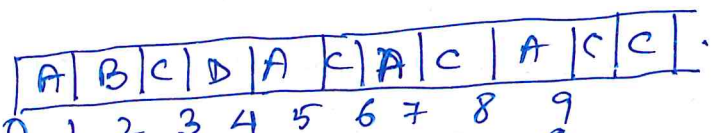
TQ = 1 unit

CT(A) = ? = 9 units of time.

Even though the A.T of all the processes are 0. It is given that process A arrives slightly before B. B arrives slightly before C and similarly for C, C arrives slightly before process D.

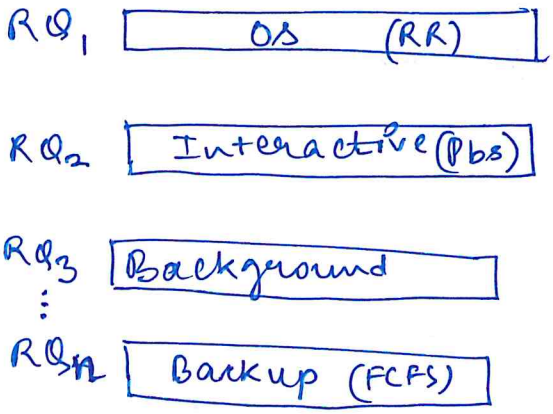
RQ : ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~A~~ ~~C~~ ~~A~~ ~~C~~ ~~A~~ ~~C~~

CPU :



Multi-level - Queues :

Let's have multiple ready queues, one ready queue corresponding to each process.



RR : Round Robin

Pbs : Priority based scheduling

We have priorities based on the type of processes.

Priority (RQ_1) > Priority (Interactive) > Priority (Background) > Priority (Backup)

If only the RQ_1 is empty then the CPU will be able to run a process from RQ_2 and similarly for the remaining queue. Therefore priority is maintained.

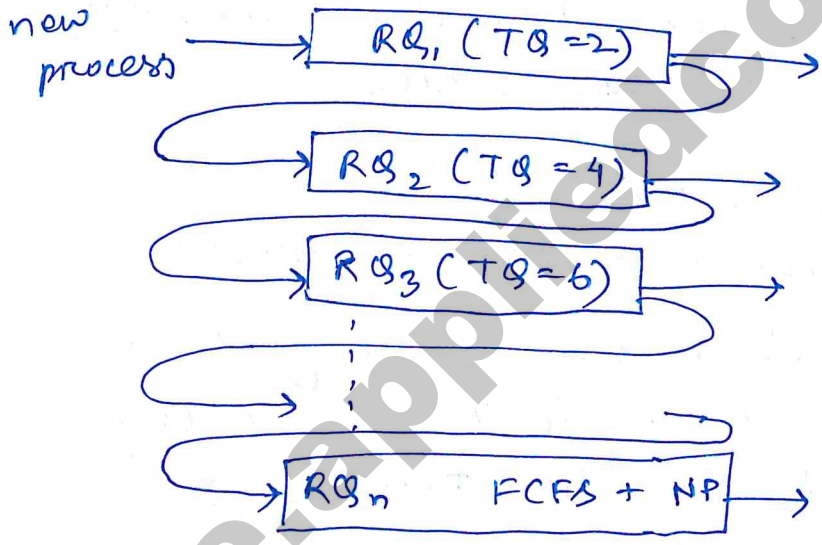
Within OS processes, we can run any scheduling algorithm we want. Similarly for the remaining queues.

The last / lowest priority queue usually implements FCFS scheduling algorithm.

There is a problem of starvation for lower priority processes if higher priority process keep arriving in the system.

Multi-level Feedback Queues:

(Round Robin + MLQ)



A typical process joins RQ_1. If it can finish in 2 units of time, it will finish, or else it will be moved to the next level (lower priority queue with larger TQ) RQ_2. This way we are trying to balance between high BT and low BT processes getting the highest priority RQ_1 all the time. Thus allowing lower BT processes to complete as soon as possible.

But starvation can still happen here.

Higher BT processes can be starved as they will be moved to the lower priority ready queues with high TQs, while a burst of O/s processes keep arriving in the system.
Similar to MLQ, in MLFQ, only if the higher priority ready queues are empty, the lower priority ready queue processes will get scheduled in the CPU.

Processes like backup and virus check processes take long B.Ts to complete. Thus can suffer from starvation in this strategy.

Note: An addition to MLFQ:
We can avoid starvation of processes in lower level queues by moving those processes to a higher level queue if they wait for too long in a low level queue using an ageing mechanism where the priority of the waiting processes are increased.

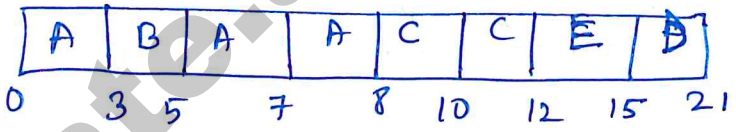
Note: No scheduling algorithm is perfect or the best. It depends on the OS that we are designing. Based on which, we will consider the best possible scheduling algorithm. In Linux, Android, priority based scheduling is widely popular.

Solved Problems - I

Qn) Consider the following set of processes that need to be scheduled on a single CPU. All the times are given in milliseconds.

Process Name	Arrival Time	Execution Time
A	0	6
B	3	2
C	5	4
D	7	6
E	10	3

Using the shortest remaining time first scheduling algorithm, the average process turnaround time (in msec) is : 7.2 msec



The TAT for each processes :-

A : 8
B : 2
C : 7
D : 14
E : 5

Avg. TAT

$= \frac{36}{5} = 7.2$ msec.

Qn 2)

Three processes A, B and C each execute a loop of 100 iterations. In each iterations of the loop, a process performs a single computation that requires t_c CPU milliseconds and then initiates a single I/O operation that lasts for t_{io} milliseconds. It is assumed that the computer where the processes execute the sufficient number of I/O devices and the OS of the computer assigns different I/O devices to each process. Also, the scheduling overhead of the OS is negligible. The processes have the following characteristics:

Process Id	t_c	t_{io}
A	100 ms	500 ms
B	350 ms	500 ms
C	200 ms	500 ms

The processes A, B and C are started at times 0, 5 and 10 milliseconds respectively, in a pure time sharing system (round robin scheduling) that uses a time slice of 50 milliseconds. The time in milliseconds at which process C would complete its first I/O operation is 1000 ms

Soln.

100 iterations
{
compute ⟹ t_c
I/O ⟶ t_{io}
}

B C

Time
A : 50
→ B : 50
✓ C : 50
A : 50 ← A : I/O : 500 ms
→ B : 50
✓ C : 50
→ B : 50
✓ C : 50
→ B : 50
✓ C : 50 ← C : I/O : 500 ms

C started its I/O operation at 500 ms and C requires 500 ms of I/O time.

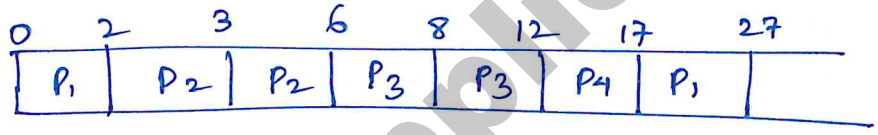
∴ C completes its first I/O operation at 1000 ms

Note: I/O can happen parallelly, because it requires another device, not the CPU.

Qn) An Operating system uses shortest remaining time first scheduling algorithm for pre-emptive scheduling of processes. Consider the following set of processes with their arrival times and CPU burst times (in milliseconds):

Process	Arrival Time	Burst Time
P1	0	12
P2	2	4
P3	3	6
P4	8	5

The average waiting time (in milliseconds) of the processes is 5.5 msec



P1 : 4 + 6 + 5 = 15

P2 : 0 = 0

P3 : 3 = 3

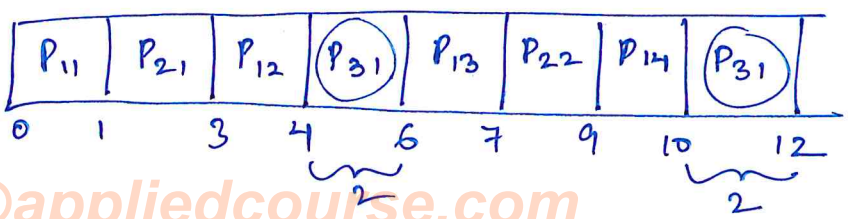
P4 : 4 = 4

Avg. WT = $\frac{22}{4}$ = 5.5 msec.

Q.n) Consider a uniprocessor system executing three tasks T_1, T_2 and T_3, each of which is composed of an infinite sequence of jobs (or instances) which arrive periodically at intervals of 3, 7 and 20 milliseconds, respectively. The priority of each task is the inverse of its period and the available tasks are scheduled in order of priority, with the highest priority task scheduled first.

Each instance of T_1, T_2 and T_3 requires an execution time of 1, 2 and 4 milliseconds, respectively. Given that all tasks initially arrive at the beginning of the 1st milliseconds and task pre-emptions are allowed, the first instance of T_3 completes its execution at the end of 12 milliseconds.

AT		Period	BT	ATs: 0 3 6 9 12
0	T_1 →	3	1	$P_{11}, P_{12}, P_{13}, P_{14}, P_{15}$...
0	T_2 →	7	2	0 7 14 21 > $P_{21}, P_{22}, P_{23}, P_{24}$, ...
0	T_3 →	20	(4)	0 20 40 60 > $P_{31}, P_{32}, P_{33}, P_{34}$, ...



P_3 runs on CPU in 2 bursts to complete its B.T of 4 units. It completes execution in 12 milliseconds (The 1st instance of T_3 completion Time).

Qn) The maximum number of processes that can be in Ready state of a computer system with n CPUs is:

1. n
2. n^2
3. 2^n
4. Independent of n.

The number of processes in the ready state could be any number. It is independent of the number of CPUs.

Solved Problems II

Qn) For the processes listed in the following table, which of the following schedule schemes will give the lowest average Turn Around Time?

Process	Arrival Time	Processing Time
A	0	3
B	1	6
C	4	4
D	6	2

(A) First come First Served

(B) Non-preemptive Shortest Job First

(C) Shortest Remaining Time First

(D) Round Robin with Quantum value two.

Soln.

Typically SRTF has the lowest average Turn Around Time. [If all the processes arrive at the same time]

FCFS: $3 + 8 + 9 + 9 = 29/4 = 7.25$ (A, B, C, D)

SJF: $3 + (9-1) + (11-6) + (15-4) = \frac{27}{4} = 6.75$

SRTF: $3 + 4 + 4 + 14 = \frac{25}{4} = 6.25$

RR: 8.25

Qn) Consider an arbitrary set of CPU-bound processes with unequal CPU burst lengths submitted at the same time to a computer system. Which one of the following process scheduling algorithms would minimize the average waiting time in the ready queue?

(A) Shortest remaining time first

(B) Round Robin with time quantum less than the shortest CPU burst.

(C) Uniform random.

(D) Highest priority first with priority proportional to CPU burst length.

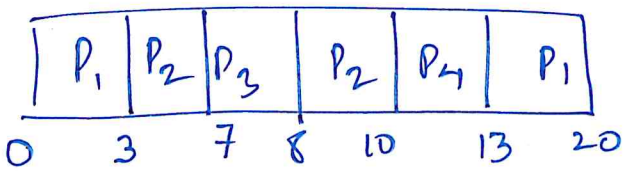
Soln. Shortest Remaining time first minimizes the average waiting time.

Qn) Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is preemptive shortest remaining time first.

The average turn around time of these processes is 8.25 milliseconds.

Process	Arrival Time	Burst Time
P_1	0	10
P_2	3	6
P_3	7	1
P_4	8	3

Soln.



P_1 : 20
P_2 : 7
P_3 : 1
P_4 : 5

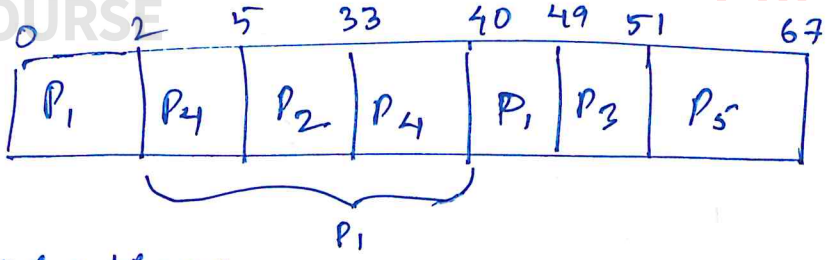
$\frac{33}{4} = 8.25$

Qn: Consider a set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.

The average waiting time (in milliseconds) of all the processes using preemptive priority scheduling algorithm is _____.

Process	Arrival Time	Burst time	Priority
P_1	0	11	2
P_2	5	28	0
P_3	12	2	3
P_4	2	10	1
P_5	9	16	4

Soln.



Waiting times:-

P_1 : 38

P_2 : 0

P_3 : 37

P_4 : 28

P_5 : 42

145

$\therefore \frac{145}{5} = 29$ msec

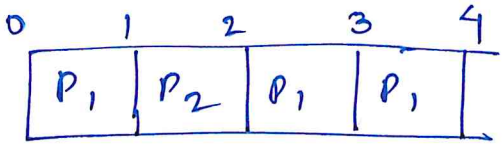
(Qn). Consider the following four processes with arrival times (in milliseconds) and their length of CPU burst (in milliseconds) as shown below:

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	4
CPU Burst time	3	1	3	2

These processes are run on a single processor using preemptive shortest Remaining Time First scheduling algorithm. If the average waiting time of the processes is 1 millisecond, then

the value of z is 2 ms.

Soln:



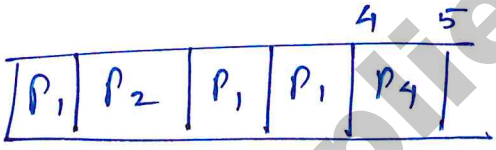
$WT(P_2) = 0$

$WT(P_1) = 1$

$WT(P_3) = 1$

Total waiting time
= # process * Avg. waiting Time
= 4 * 1 ms
= 4 ms

If z = 1,
then P_4 gets scheduled,



$WT(P_4) = 0$

$WT(P_3) = 1 + 1 = 2$

$\therefore$ Total WT = $WT(P_1) + WT(P_2) + WT(P_3) + WT(P_4)$

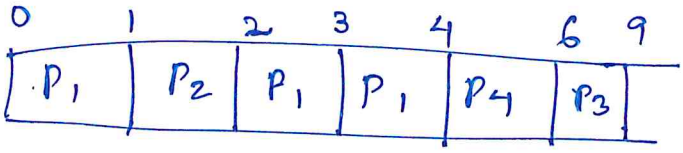
$= 1 + 0 + 2 + 0$

Total WT = 3 ms

But we are given that Avg WT = 1 ms
and we got **Total WT = 4 ms**

$\therefore z \neq 1$.

If $z = 2$ →



$WT(P_1) = 1$
$WT(P_2) = 0$
$WT(P_3) = 1+2$
$WT(P_4) = 0$

4 ms

It matches with the required Total W.T

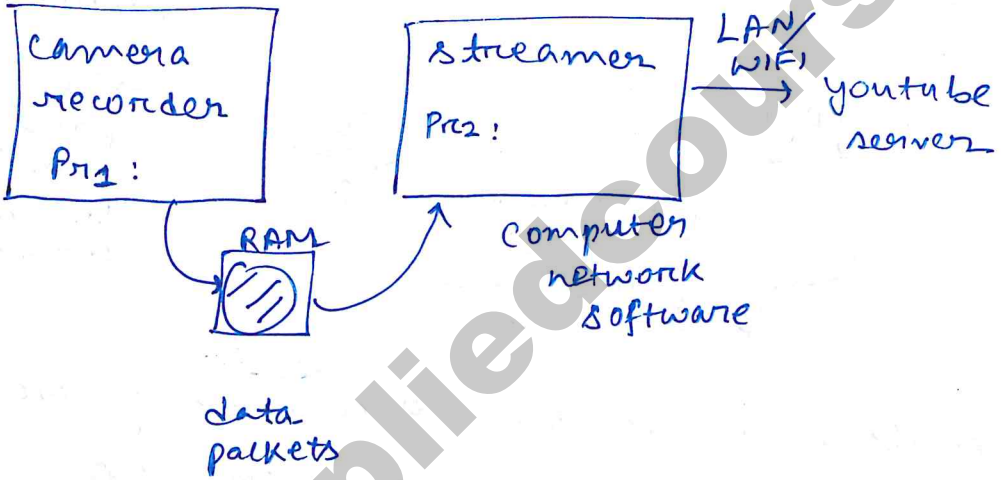
∴ $\boxed{z = 2 \text{ ms}}$ ← Answer

IPC + Synchronization : An introduction

IPC & Synchronization :

↳ widely used by software development engineers, machine learning scientists.

E.g. When we do youtube live :



The moment the camera records 1 second of data, it has to send it via packets to the streamer. There has to be a synchronization between Process Pr_1 and process Pr_2.

We can think about storing the video in a file using a process that writes to the file. And the streamer process reads and then sends it.

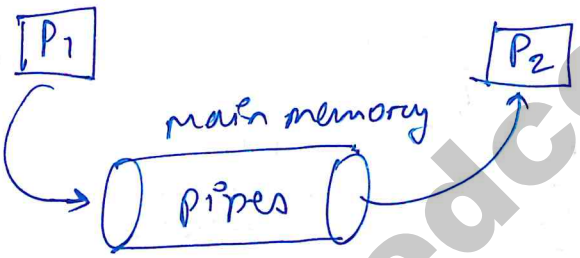
But since disk read/write is extremely

Now, it makes more sense to store the data in the RAM. But the problem is whatever is there in the dynamic memory of Pr1 cannot be accessed by the process Pr2 and vice-versa.

There we need inter process communication with the above constraint in place.

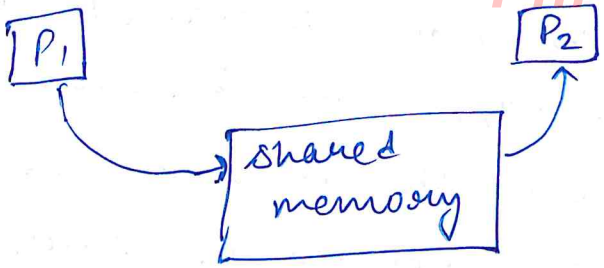
Communication strategies: (Inter process communication)

↳ Two or more processes running on the same computer.



A pipe is a file in the main memory or some space in the main memory which both process P_1 and P_2 can access. It is used for one-way communication only. For two way communication, two pipes can be set up. Pipe is a very popular concept in Unix, Android and Linux etc.

Shared memory is another concept where both P_1 and P_2 can access simultaneously. multiple processes can infact access simultaneously.



Also, there is another concept called message passing. There are standard queues (FIFO) for communicating between P_1 and P_2.

Intra process communication:

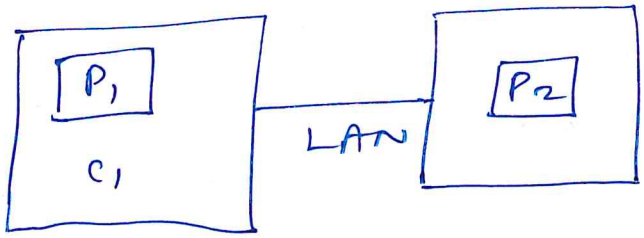
Intra process communication is the communication between the same process.

Let's say we have a process, which has the following snippet of code:

```
// global variables, parameters.
main()
{
    f();
}
f()
{
}
```

Intra process communication is done with the help of global variables and function parameters.

Note: Interprocess communication (in the real world) can happen in between two process where the processes belong to different computers (in distributed systems)



Note: There is a concept called Remote procedure calls in Java. (RPC)

Synchronization

cooperative

Multiple processes are cooperating to get the task done

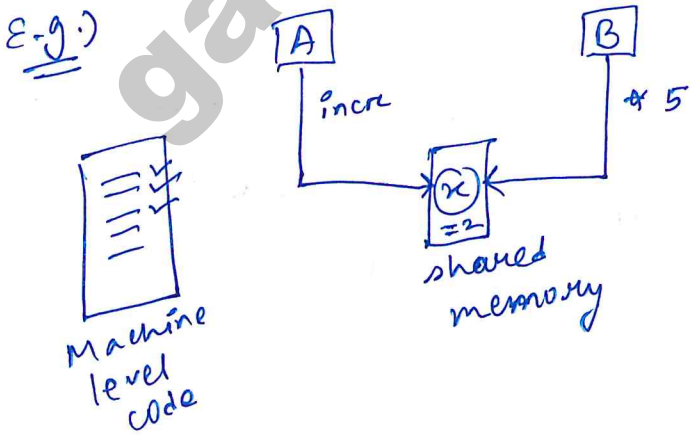
E.g.: Youtube streaming

competing

Multiple process are competing to get hold of the resources.

E.g. Two processes competing for the pointer (resource).

E.g.)



Order of execution:

① A: x = x+1 ③
B: x = x*5 ⑮

② B: x = x*5 ⑩
A: x = x+1 ⑪

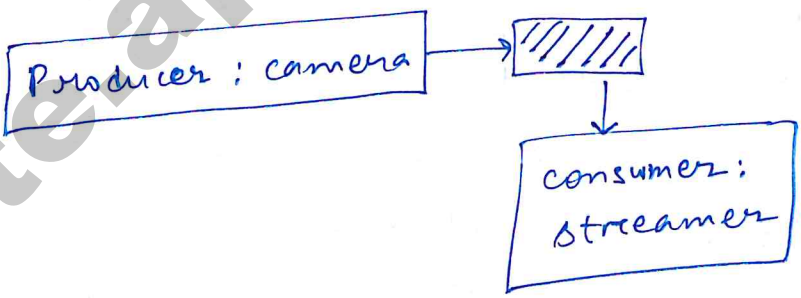
Depending on the sequence / order of execution, the output changes and these could sometimes be disastrous as we can't assume in which order, it will be executed. If A finishes only then B can start and vice-versa as variable x is stored in a shared memory.

Also, code A may be written by software engineer 1 and code B may be written by software engineer 2.

The other challenge we need to think of is while A is changing the variable how do we ensure that B doesnot change it too.

Producer - Consumer Problem : Challenges

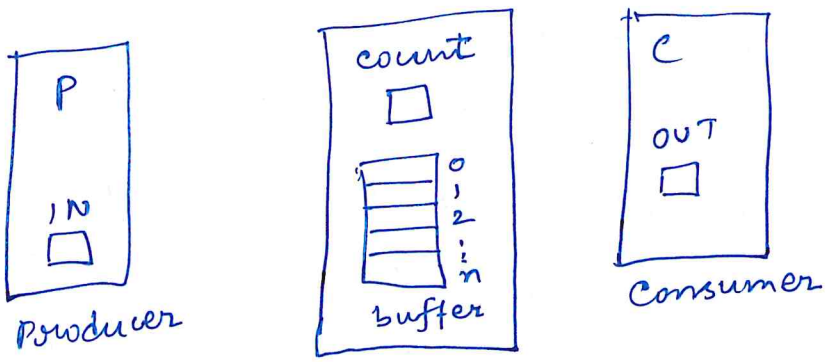
↳ {video streaming



In the video streaming, the camera process produces video data (records it and compresses the data). Say it produces 1 second of data.

The other process called the streamer (consumer process) which consumes this data.

The system's view / abstract view of producer consumer problem:



Assumption: preemptive O/S

Producer process has a local variable called IN. Consumer process has a local variable called OUT.

COUNT is a variable in the shared memory. There is an array which we can think of as a buffer.

IN variable content tells us where to place the content (item) that will be generated next. Once placed COUNT will be incremented.

COUNT tells us how many items are present in the buffer.

Using the content (value) of OUT variable, the consumer will read the item from the buffer and decrement the count.

Linux shared memory:

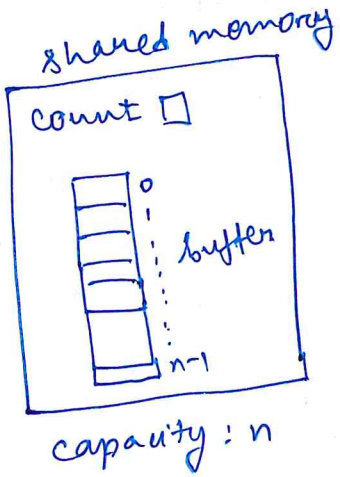
shmget() : shared memory get.
shmat() : shared memory attach.
shmdt() : shared memory detach.
shmc() : shared memory control.

In we write code in C programming in Linux, Linux has the above four functions (OS utilities) to create shared memory.

```
void Producer(void)
{
    int itemp, in=0;
    while (true) {
        P1. produce item (itemp);
        P2. while (count == n);
        P3. buffer [in] = itemp;
        P4. in = (in+1) mod n;
        P5. count = count + 1;
    }
}
```

infinite loop

```
void consumer(void) {
    int itemc, out=0;
    while (true) {
        while (count == 0);
        itemc = buffer[out];
        out = (out+1) mod n;
        count = count - 1;
        consume item (itemc)
    }
}
```



P2 : If we have n elements in the buffer and the consumer hasnot consumed them, then there is no space in the buffer to place a new item. So, the while will force us to be stuck in the infinite loop. This step is a busy waiting step. This is also called as spinlock.

Only when the count becomes less than n, we will come out of the loop.

P3 : If there is empty space in the buffer, whatever item (itemp) that is currently produced is placed in the buffer. (buffer[in]).

P4 : We increment 'in'. Using 'mod n' makes the buffer behave like a circular queue.

C1 : count == 0 means there is no item in the buffer. This is also a busy waiting state as consumer keeps on waiting for the producer to produce an item.

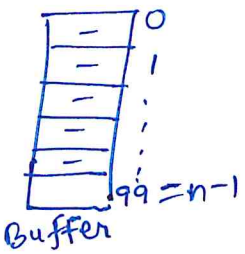
C2 : If there is an item in the buffer, then place the item in the itemc from buffer[out].

C3 : We increment 'out'.

C4 : count is decremented

Tricky case: (Failure case of the producer-consumer)

count = 5



RQ: [P] [C]

Producer:

P5: count = count + 1

P51: Load R_p, M[count]
P52: INCR R_p
P53: STORE M[COUNT] R_p

} Equivalent machine level codes (instructions)

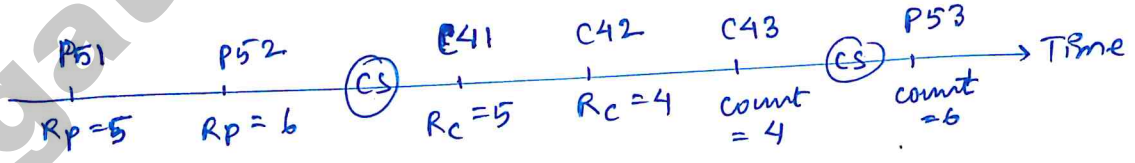
Consumer:

C4: count = count - 1

C41: Load R_c M[count]
C42: DECR R_c
C43: STORE M[COUNT] R_c

[Assumption: Preemptive O.S / preemptive multiprogramming O.S.]

Case 1: P: P51, P52

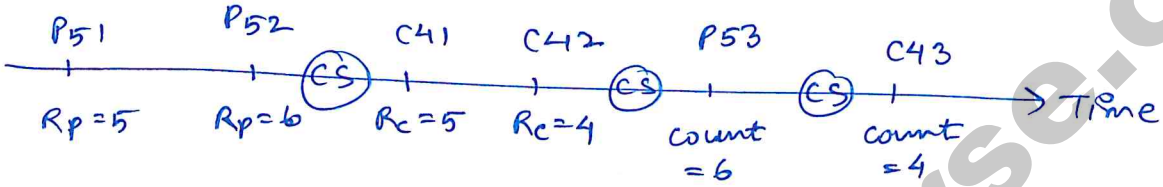


If context switch didnot happen, when producer code executes, count becomes 6 and then when consumer consumes (executes its code), count becomes 5. And viceversa sequence would have

resulted the final result as count = 5.

Because of context switches, the final value of count becomes 6 and is therefore inconsistent.

Case 2:



Because of the context switches and the preemptive nature of the operating systems, final value of count can be either 6 or 4.

This is inconsistent and incorrect.

These are user processes, context switch can happen anytime, anywhere.

Therefore, this issue has to be resolved.

Synchronization Mechanisms I

Necessary conditions for synchronization problems / inconsistencies:

(1) Critical section (C.S):

This is the portion of the code where shared resources are accessed.

[Non-critical section (NCS): The portion/part of the code/program where shared resources are not accessed.]

If there is no critical section, then there is no synchronization problem to start with.

∴ Synchronization problem needs critical section to be present.

(2) Race condition:

Processes must be racing/competing to access the shared resources. Access of shared resource happen in the critical section part.

If there is no race condition, there is no synchronization problem.

(3) Preemption:

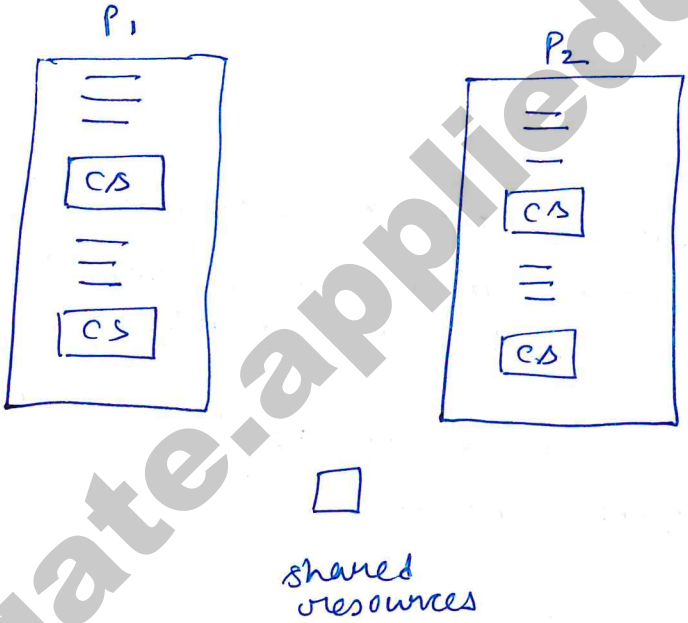
Preemption can occur. If the scheduling algorithm is non preemptive, then there is no problem at all. There is no issue of synchronization because one process will execute completely and then the next process execution will start.

Therefore, the existence of critical section, the existence of race condition and having a preemptive scheduler — these 3 are the necessary conditions for synchronization problem.

To avoid synchronization problem, there is an idea called mutual exclusion (ME).

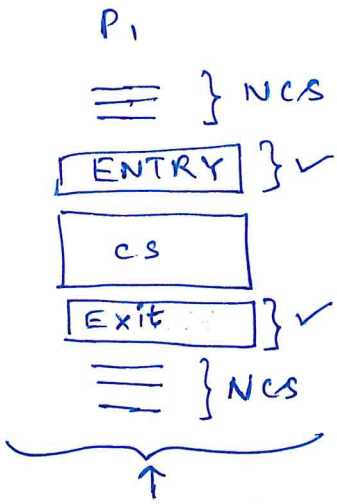
Mutual Exclusion (ME) :

We are going to exclude one process from accessing the resource while the other process is accessing the resource.

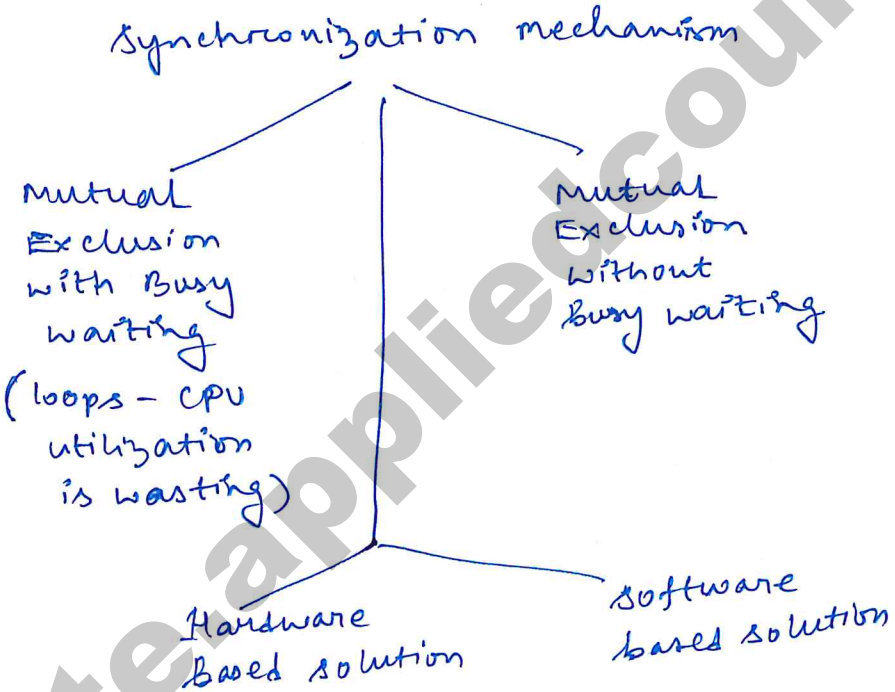


Only one of the competing process can be in its critical section at any point of time.

How is M.E actually implemented?



Mutual Exclusion Model.



Conditions for synchronization mechanisms:

① Guaranteed Mutual Exclusion: While one process is in its critical section, other processes should not go inside the critical section. It should guarantee that.

② Progress: Process that are currently executing in the Non-critical section may not block other processes which want to enter the critical section. [overall system progress should be satisfied].

Note: M.E and progress are primary conditions.

③ Bounded waiting:

No process has to wait forever to access the critical section. It means the waiting time is always bounded (limited).

④ No assumptions about speed & CPUs:

This is very important for the design of O.S. We can't assume anything about the speed of processors and the number of processors. By not making these assumptions, we can build a general purpose operating system.

Note: 3rd & 4th conditions are secondary conditions.

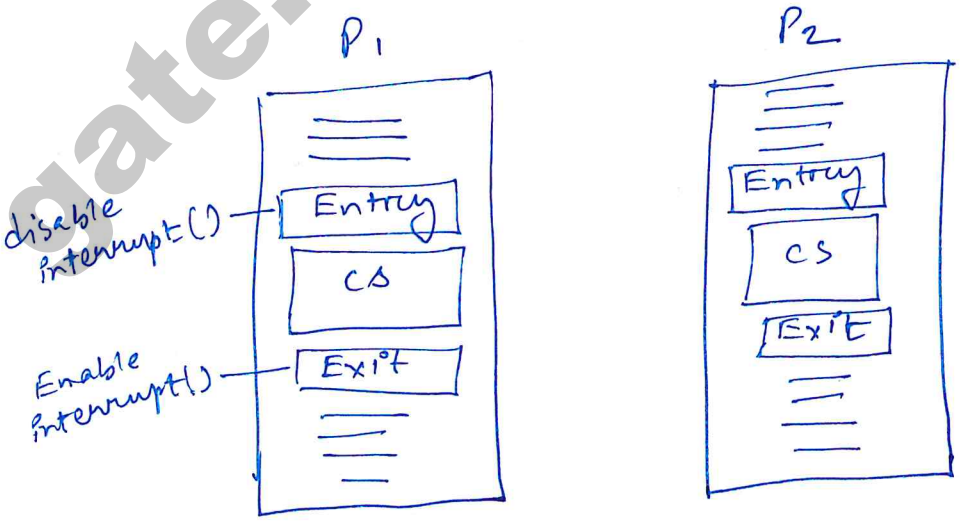
Every synchronization mechanism should satisfy the primary conditions (mandatory conditions). The secondary conditions are what we ideally want to have.

Note: We would like to avoid busy waiting, if we can. Because it will improve the throughput if CPU cycle is not wasted.

Mechanism 1: Disable Interrupts (H/W mechanism + O.S utility)

This is a hardware mechanism. It has to be supported by the CPU. CPU might have one register which says if it is 0 then the interrupts are disabled.

In the entry section, we will write one line of code that says disable interrupts. It will call the O.S utility and the O.S changes on the hardware.



Suppose P_1 has disabled the interrupt and entered the critical section.

Now P_2 tries to enter its critical section, but can't progress anymore now. As when P_2 tries to disable the interrupt, since the interrupts are already disabled, it will be understood that the process P_1 is already inside the C.S. Therefore, P_2 gives up the control / relinquishes control of CPU to other processes.

In the exit section, we write one line of code called enable interrupt. Once enable interrupt happens, whenever the turn is for process P_2, then it will try to disable the interrupt and enter the C.S.

Note! There is no busy waiting. It will never enter a busy waiting loop because whenever a process is in its critical section, it can't preempted.

When we are solving through Disable Interrupt mechanism we want to make sure that the critical section doesnot take too much time. Because if critical section takes too much time then the whole O/S is blocked. Otherwise having disabled interrupt makes it as a non preemptive scheduler.

This has to be implemented in the kernel mode.

Mechanism 2: Lock variable:

Software mechanism. It could have busy waiting we can execute it in the user mode.

Lock is a shared variable.

```
entry section:
    while (lock != 0);
    lock = 1;

exit section:
    lock = 0
```

The solution does not guarantee mutual exclusion. Here lock is just another user variable just like count in producer consumer problem.

Since lock does not guarantee M.E, it is not a valid solution.

In 1970's, 1980's tons of research went into this field but many mechanisms turned out to be incorrect.

Mechanism 3: Strict Alternation:

It is a software mechanism. It could have busy waiting, 2-process solution.

P_1

```
=
NCS
=
while (turn != 0); // ENTRY
=
 CS
=
turn = 1;   // EXIT
=
NCS
=
```

P_2

```
=
NCS
=
while (turn != 1); //Entry
 CS
turn = 0; //Exit
=
NCS
=
```

Here there is a shared variable called turn, which can either be set to 0 or 1.

It guarantees M.E.

P_1 and P_2 can enter their C.s alternately and they can't enter their own C.s again, without making sure the other process enters the C.s.

$P_1 \xrightarrow{CS} P_2 \xrightarrow{CS} P_1 \xrightarrow{CS} P_2 \rightarrow \ldots$

when turn = 0,
P_1 enters the C.s and P_2 goes into busy waiting.
In the exit section of P_1, turn is made to 1.
Now, P_2 can enter its critical section.

The problem with strict alternation is that it doesnot guarantee progress.

Turn = 0

P1:

≡ NCS

ENTRY : while (turn != 0)

CS

EXIT : turn = 1

≡ NCS

P2 :

≡ NCS

ENTRY : while (turn != 1)

CS

EXIT : turn = 0

≡ NCS

P_1 in the NCS is blocking P_2 to enter the C.S. P_2 will wait till P_1 makes turn = 1. Infact, if P_1 doesnot show any interest to execute its C.S, P_2 will forever wait to enter its own C.S.

Detailed explanation about Disabling interrupt

[Source: William Stallings]

Disable interrupts

```
While (true) {
    /* disable interrupts */ ; ← Entry section
    /* critical section */ ;
    /* enable interrupts */ ; ← Exit section
    /* remainder */ ;
}
```

infinite while just like in producer consumer problem

Peterson Solution & TSL based synchronization:

Peterson's solution: (1981)
↳ 2 process solution

(1990)
↳ multiprocess solution
(wikipedia)

→ It uses turn and flags.
(boolean variable)

→ software based solution

→ Mutual exclusion

→ Busy waiting.

```
int turn
int flag[2] = {False, False}   } shared variables
              0      1
```

P_0:

```
Entry [ 1. flag[0] = true; // atomic & immediate
        2. turn = 1; // atomic
        3. while (flag[1] == true && turn == 1);

        CS

Exit  [ 4. flag[0] = false  // atomic
```

P_1:

```
Entry [ 1. flag[1] = true;
        2. turn = 0
        3. while (flag[0] == true && turn == 0);

        CS

Exit  [ 4. flag[1] = false.
```

These flags are actually called as intent flag. Because it indicates whether P_1 or P_2 wants to go into the C.S.

		Machine Instruction
flag [0] = true ;	≈	STORE M[FLAG[0]], 1
turn = 1 ;	≈	STORE M[TURN], 1

With a single machine instruction we can make sure that line number 1, 2 and 4 are atomic and immediate.

Let's assume P_0 is in the entry section. P_0 is changing the value of flag[0] to true, indicating that it wants to go to the critical section. Then P_0 update value of turn to 1.

The while loop is a busy waiting loop. The condition inside while loop means if process P_1 wants to go inside the critical section and the turn is 1, then P_0 gets stuck in the while loop.

Suppose there is preemption and control goes to P1. P1 updates flag[1] = true and turn = 0. Then P1's while loop also leads to busy waiting as flag[0] is true and turn = 0.

The say P1 gets preempted and control comes

comes back to P_0. The it checks the flag[1] value and turn value. Since turn = 0, therefore the while condition is false, as process 1 wants to go to critical section the current turn is of process 0. Therefore process P_0 goes inside the critical section. If during the execution of C.S, P_0 gets preempted, and control goes to process P_1, but since while condition is True for P_1, P_1 continues to be blocked. Therefore, only when P_0 comes out of C.S and update flag[0] to false. Now if P_0 gets preempted and control goes to P_1, P_1 will be go into the C.S as the while condition will be false (∵ flag[0] = False). Which means at any point in time, either P_0 is in critical section or P_1 is in the critical section.

Bounded waiting is also satisfied here as none of the processes will get stuck waiting to go to the critical section forever. The wait time for any process to enter the C.S is bounded or limited.

Progress is also satisfied. In terms of M.E + one of Busywaiting, Peterson's solution is the best solution.

Test & Set Lock Instruction (TSL):

↳ Hardware mechanism & support.

flag = 0 shared variable

ENTRY
1. TSL r, M[flag] ⇒ r = flag ; flag = 1
2. CMP r, 0 (or) COPY ACC r
3. JNZ 1

CS

EXIT
4. STORE M[FLAG], 0

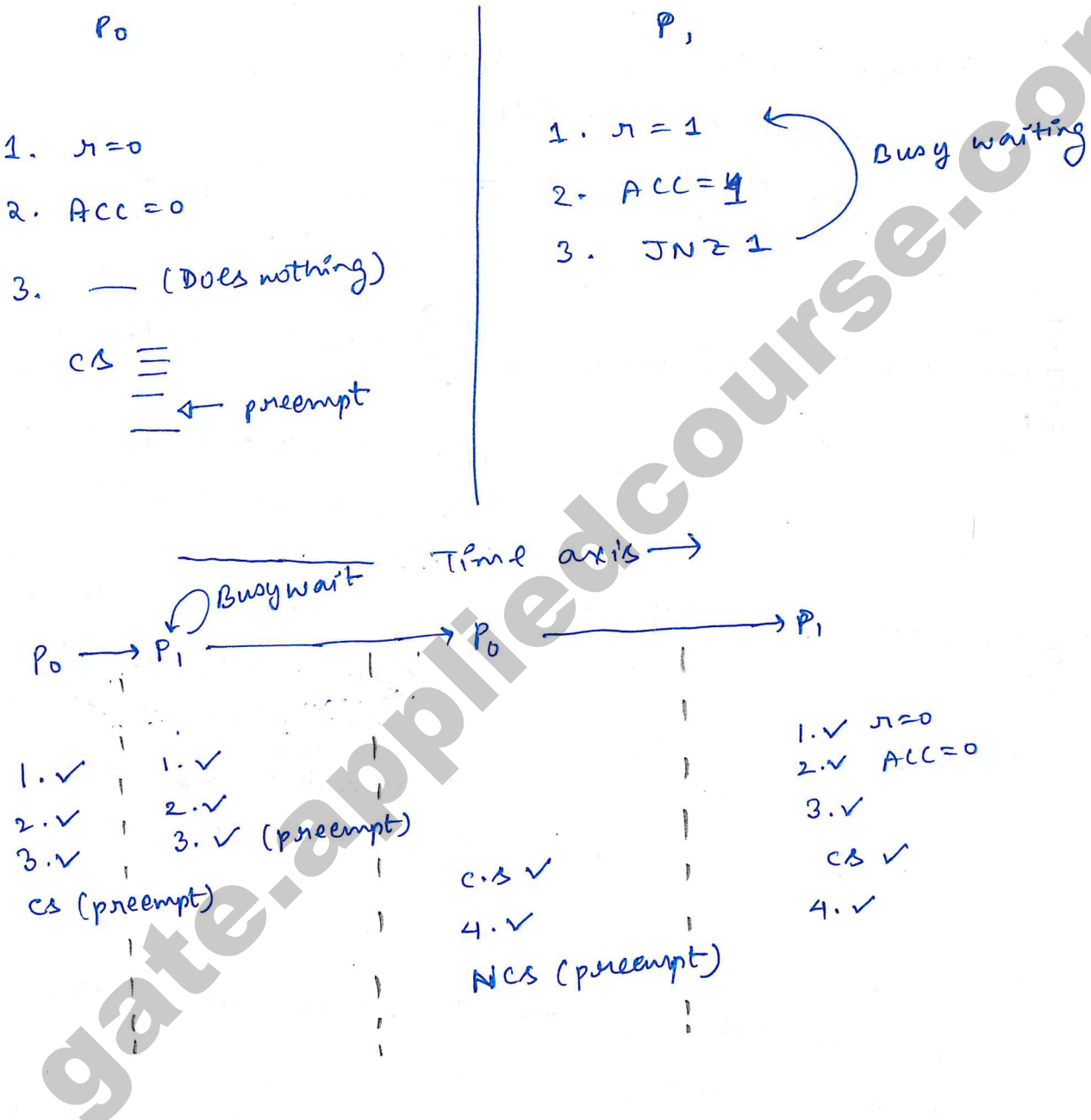
preempt →
SET r M[FLAG]
STORE M[FLAG] 1

But TSL is one instruction. It is a special instruction that the CPU provides us, which executes atomically. Because with two instructions, if used, there is a chance of preemption occurence after the 1st instruction.

Note:
flag indicates if another process is in its critical section or not.
TSL instruction executes atomically. If it is one instruction, whole instruction is either executed or not executed.
Every machine level instruction is atomic in nature. Some CPUs give this special instruction called TSL but typical CPU requires a machine

level instruction.

flag = Ø Ø 1



TSL ensures Mutual Exclusion.

Let's check if TSL satisfies bounded waiting.

Bounded waiting means a process always has a finite amount of time it waits before it goes into the critical section.

Let's assume the CPU is with P_0, P_0 already entered its critical section and is executing the critical section. While it is executing the critical section, P_1 is waiting. But as soon P_0 comes out of the C.S, the flag becomes 0.

Now, we might say, as soon as the flag becomes zero, P1 can enter into its critical section, very true. Therefore we can argue that there is a bounded waiting (finite amount of time that P_1 is waiting to get into the C.S).

→ This is an incorrect analysis. In the real world we may not only have two processes, we could have 100s of processes waiting (busy waiting) to get into their C.S.

Now, as soon as P_0 comes out of the C.S, any other process could be scheduled. It is not guaranteed that P_1 will be scheduled. In the world of operating systems, new processes keep coming in. It could just happen that P_1 is stuck busy waiting, thus leading to an unbounded waiting for P_1.

The other name of unbounded waiting is starvation. It is starvation because P_1 is being starved and not given the C.S while other processes keep getting the CPU.

Just like peterson's algorithm, this also suffers from Busy waiting problem. It is actually wasting the CPU cycles (stuck in infinite cycle / loop)

T&L guarantees progress.

Priority Inversion Problem : [Preemptive & priority scheduler.]
(Peterson, TSL)

Let's assume,

Priority (P_1) > Priority (P_0)

Ready queue : | P_0 | P_1 |

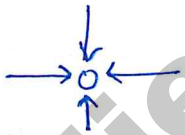
Let's assume P_0 was executing for a while, then P_1 arrives. Therefore P_1 gets control of the CPU. P_1 takes the control of the CPU when (say) P_0 was actually executing its C.S. Then, P_1 executes its Non C.S and tries to enter the C.S. Because there is a busy waiting

approach used in both Peterson and T&L →

P_1 gets stuck in busy waiting in the entry section, waiting for P_0 to give up on the C.S (or come out of its C.S). But P_1 will never give the control of execution back to the P_0 because P_1 has higher priority in comparison to P_0. While P_0 is waiting for P_1 to give the control of execution or CPU, so that P_0 can complete its execution.

This condition is called Deadlock.

This is like a traffic signal jam where nobody can move forward.



Mutual Exclusion without busy waiting : Sleep & Wakeup

Mutual Exclusion without busy waiting is a good solution as CPU cycles are not wasted.

Actually sleep() and wakeup() are two system calls (that are part of the O/S).

Whenever a process calls sleep(), the process gets preempted and enters a sleep state. The process is asking the scheduler to preempt it.

wakeup() is a function that can wakeup other processes that are sleeping.

P_0 :

ENTRY :
=
while (cond); → if (cond) sleep();
=
[CS
EXIT :
—
wakeup(P_1)
=
=

P_1 :

ENTRY :
=
while(cond);
=
[CS
EXIT :
—
wakeup(P_0)
=

Since sleep() and wakeup() are system calls they are executed in the kernel mode.

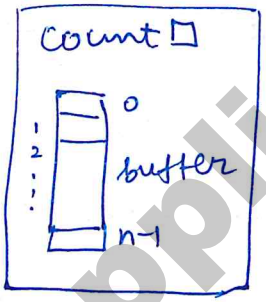
When a process goes into sleep mode, it relinquishes the control of the CPU and gives the other process a chance to complete its critical section.

(Q). Producer-consumer + sleep-wakeup

```
void Producer (void)
{  int itemp, in = 0;
   while (true) {
      P1. produceItem(itemp);
      P2. if (count == n)
              sleep();
      P3. buffer[in] = itemp;
      P4. in = (in+1) mod n;
CS.   P5. count = count + 1;
      P6. if (count >= 1) wakeup(consumer);
   }
}
```

```
void consumer (void){
    int itemc, out=0;
    while (true){
        c1. if(count==0)
                sleep();
        c2. itemc = buffer[out];
        c3. out = (out+1) mod n;
        c4. count = count-1;
        c5. if (count<n)
                wakeup (producer);
        c6. consumeItem(itemc);
```

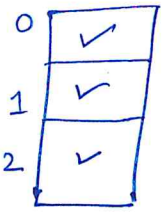
Shared Memory



Even this solution suffers from inconsistency as M.E is not guaranteed, and producer & consumer both can access the C.S together.

Therefore just by introducing sleep and wakeup to a non mutual exclusion solution doesnot guarantee anything.

Inconsistent : Both processes can be in CS simultaneously



BUFFER

n=3

COUNT 0 1 2 3

consumer : just before sleep()

producer : item 1
item 2
item 3

Assume the consumer process starts executing. Just before going to sleep the control goes to the producer function. The producer produces item1, item2 and item3. The count value = 3.

The wakeup(consumer) will only work if the consumer had gone into the sleep mode. But the consumer got preempted just before the sleep.

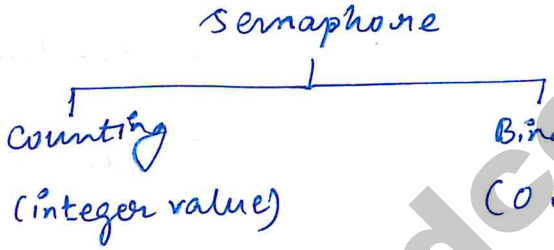
Let's assume the control is still with the process, since n=3, the producer goes into sleep.

Once the producer goes into sleep mode, the producer gets the control. And it resumes its execution and executes the sleep() and goes to sleep.

Now, the producer and consumer both go to sleep and both expect that the other will wake them. This is Deadlock.

Semaphores

Semaphores are some of the most widely used O.S tools / functionality even in modern O.S. These concepts are from 1960s. It was proposed by Djikstra's. Semaphore is basically a software resource that is part of the O.S. The O's kernel implements semaphore. Semaphores can store integer values. All the semaphore operations are atomic.



Operations on semaphores: UP & DOWN (signal/V, wait/P)

COUNTING SEMAPHORE

```
struct Semaphore {
    int value;
    Queue L;
}
```

```
INIT(semaphore, int x)
{
    S.value = x;
}
```

```
DOWN(semaphore S)
{
    S.value--;
    if(S.value < 0){
        add(curr_proc, S.L)
        sleep(current process)
    }
    else
        return
}
```

```
UP(semaphore S)
{
    S.value++;
    if(S.value <= 0)
    {
        p = get(S.L)
        wakeup(p)
    }
    else
        return
}
```

L is a List. It is actually a queue of PCBs.

INIT, UP and DOWN are atomic operations.

value is +ve means number of available resources that can be allocated to new processes.

Binary semaphores:

```
struct BSemaphore
{
  enum value {0, 1};
   Queue L;
}
```

```
INIT(BSemaphore S, int x)
 {
   S.value=x;
 }
```

```
UP(BSemaphore S)
 { if(S.L is not empty)
   {
     P = get(S.L);
     wakeup(P);
   }
  else
   {S.value = 1;
    return;
   }
```

```
DOWN(BSemaphore S)
{ if(S.value == 0)
  {
   add(currproc,
              S.L);
    sleep(currproc)
  }
 else
  {
   S.value = 0;
   return;
  }
```

Binary semaphores are weaker counterparts of counting semaphores. We can easily simulate a binary semaphore using a counting semaphore. Binary semaphores are also called mutexes.

Note: The implementation of a binary semaphore and implementation of a mutex could be very similar. The way mutex are applied and used in the real world are slightly different from how binary semaphores are used.

If on down operation, s.value = 0 already, the binary semaphore adds the process to the queue and forces it to go to sleep.

So down basically means the binary semaphore goes to 0 or gets blocked.

Up basically means if there are some processes in the queue then get the process from the queue and wake it up. And if the queue is empty, then change the s.value to 1 and return.

Note:

Binary semaphore value cannot tell us how many processes are blocked.

Counting semaphore if the value is +ve, it tells us how many resources we have and if it is negative, it tells us how many processes are there in the queue.

There are successful and unsuccessful cases:

① Successful case:

The else case in DOWN operation in the semaphores.

② Unsuccessful case:

The if() condition in DOWN operation in the semaphores. If the if() is satisfied, the process goes into the sleep mode and doesnot get the resource.

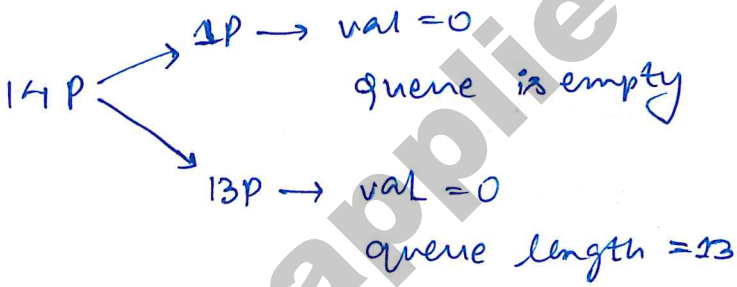
(Q) In an application, the value of binary semaphore is 1.

Operations : 14P, 8V, 6P, 2V, 10P, 4V

What is S.value & size of S.L?

P : DOWN
V : UP

	14P	8V	6P	2V	10P	4V
S.val :	0	0	0	0	0	0
Queue length :	13	5	11	9	19	15



Out of 13 processes (on queue), 8 of them will be woken up.

6 more processes get added to the queue

⋮

At the end, 15 processes wait in the queue and s.val = 0.

The four cases of Binary semaphore:

Case 1: INIT(S,1) S.val = 1

DOWN(S) S.val = 0

State: Successful
(as we get the resource)

Case 2:

INIT (S,0) S.val = 0

DOWN(S) S.val = 0

State: Unsuccessful

Case 3: INIT (S,1) S.val = 1

UP(S) S.val = 01

Case 4: INIT (S,0) S.val = 0

UP(S) S.val = 1

Producer - consumer using Semaphore

```
Producer() {
    int in=0, itemp;
    while (true) {
        P1 : produce item(itemp);
Entry   [ P2: DOWN(E)
        [ P3: DOWN(b)
CS      [ P4: Buffer[in] = itemp;
        [ P5: in = (in+1) mod n;
Exit    [ P6: UP(b);
        [ P7: UP(F);
    }
}
```

```
Consumer() {
    int out=0, itemc;
    while (true){
Entry   [ C1: DOWN(F);
        [ C2: DOWN(b);
CS      [ C3: itemc = Buffer[out];
        [ C4: out = (out+1) mod n;
Exit    [ C5: UP(b);
        [ C6: UP(E);
          C7: consumeItem(itemc)
    }
}
```

n = 100

Buffer[100]

Shared variable

Semaphore E = n

Semaphore F = 0

BSemaphore b = 1

The importance of binary semaphore b is that if a process is in C.S, another process cannot get into the C.S.

Semaphore E is helping producer determine if there are any empty cells or not. If there is no empty cell, it will go to sleep. If there is even a single empty cell, we can go in.

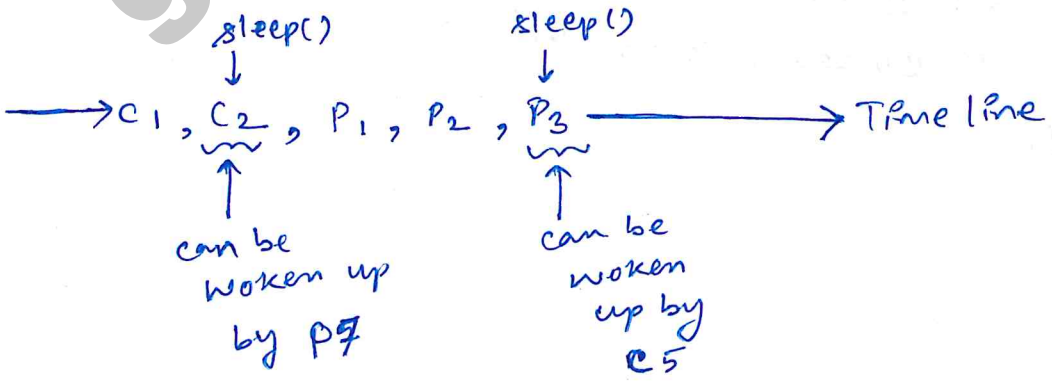
Semaphore is basically helping the consumer to determine even if there is a single buffer that is full, so that it can consume it.

If none of the buffers are full, that is all the buffers are empty, it will go on to sleep.

Note: A simple mess up or re-ordering of the operations could lead to a deadlock.

E.g.

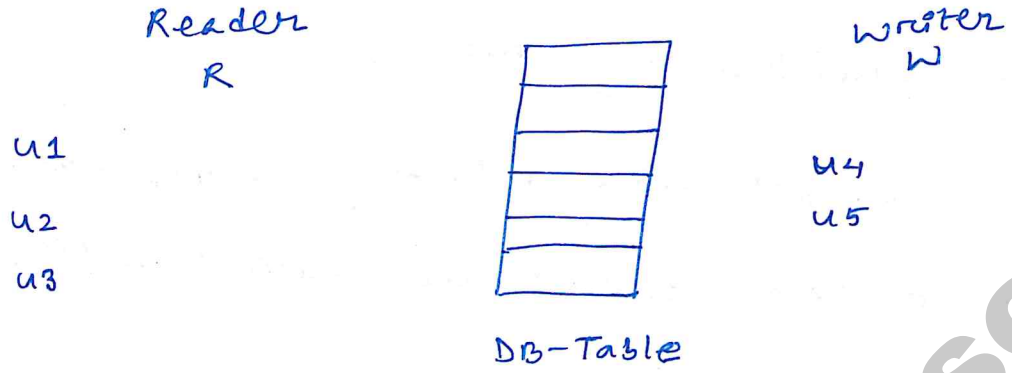
swap
C_1 : ~~Down(F);~~ Down(b);
C_2 : ~~Down(b);~~ Down(F);



∴ Deadlock.

Reader - writer problem + Semaphores

This problem can be seen in databases.



Note: When some process is writing in the database, no reader process should be allowed to write

There can be multiple readers reading simultaneously. But while a reader is reading, a writer can't start writing. The writer has to wait for the readers to finish the task. Once the readers finish the task, then the writer can get control of the database.

Database can be thought of as a software resource, where multiple processes are trying to access it, read, write and update it etc. So, from an O.S viewpoint, a database is nothing but a software resource where there are multiple jobs / processes which are trying to

read or write.

From a pure database perspective, we can have multiple processes read.

There are multiple cases :-

Case 1: 1 Reader & 1 writer:

We can just have a binary semaphore which controls access to the critical section

Entry : Down(b)

—CS—

Exit : UP(b)

Case 2: n-readers & 1 writer:

When any of the n readers are in the C.S, then a writer cannot go in the C.S.

Note: N readers can simultaneously go into the C.S.

case 3: n-readers & n writers:

In general, n-readers can go in the C.S.

But if any of the readers are in the C.S, no writers can go in the C.S.

If any one of the writers are in the CS, no other writers can go in the C.S.

We will solve this problem using semaphore.

```
int rc = 0
Bsemaphore m_rc = 1;
Bsemaphore rw = 1;
```

```
Writer() {
  while (true)
  {
Entry → DOWN (rw);
  cs: writeDB();
Exit → UP(rw);
  }
}
```

```
Reader()
{
  while (true)
  {
    DOWN (m_rc);
    rc = rc + 1;
    if (rc == 1)
      DOWN (rw);
    UP (m_rc);
    ReadDB();
    DOWN (m_rc);
    rc = rc - 1;
    if (rc == 0)
      UP (rw);
    UP (m_rc);
  }
}
```

Note: rc : readers count

m_rc : mutex for controlling rc.
or
mutex to access rc.

rw : mutual exclusion between readers and writers and amongst the writers.

In reader process:

① Before we access / modify rc, we want to get control of rc using m_rc.

② If (rc==1) is true, then the 1st reader has arrived.

③ If (rc==0) is true, then it is the last reader. Then do up(rw), so that the writers can get the control.

In the real world we will encounter lots of examples of readers writers problems. Like multiple processes are trying to read or write a file at the same time.

Solved Problems-1

Q. Consider the procedure below for the Producer consumer problem which uses semaphores:

Which one of the following is True?

(A) The producer will be able to add an item to the buffer, but the consumer can never consume it.

(B) The consumer will remove no more than one item from the buffer.

(C) Deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.

(D) The starting value for the semaphore n must be 1 and not 0 for deadlock-free operation.

```
semaphore n=0;
semaphore s=1;

void producer()
{
    while (true)
    {
        produce();
        semWait(s);
        addToBuffer();
        semSignal(s);
        semSignal(n);
    }
}

void consumer()
{
    while(true)
    {
        semWait(s);
        semWait(n);
        removeFromBuffer();
        semSignal(s);
        consume();
    }
}
```

Soln:

(A) Consumer is not blocked here. [False]

(B) consumer can remove more than one item from the buffer. [False]

(C)

```
semaphore n=0; -1
semaphore s = 1; 0 -1
```

[blocked] semWait(s) ← semWait(n) [blocked]

semSignal(n) → semSignal(s)

∴ Deadlock occurs [True].

(D) False.

Q. The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

```
P1()
{
   C = B-1;
   B = 2*C;
}

P2()
{
   D = 2*B;
   B = D-1;
}
```

The number of distinct values that B can possibly take after the execution is 3

Soln. B = 2

D = 2*B = 4 B = D-1 = 3 C = B-1 = 2 B = 2*C = 4	C = B-1 = 1 B = 2*C = 2 D = 2*B = 4 B = D-1 = 3	C = B-1 = 1 D = 2*B = 4 B = D-1 = 3 B = 2*C = 2
C = B-1 = 1 D = 2*B = 4 B = 2*C = 2 B = D-1 = 3	D = 2*B = 4 C = B-1 = 1 B = D-1 = 3 B = 2*C = 2	D = 2*B = 4 C = B-1 = 1 B = 2*C = 2 B = D-1 = 3

∴ B = {3, 2, 4}

Qn) Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes.

Process X

```
/* other code for process X */
while (true)
{
  1. varP = true;
  5. while (varQ == true)
     {
6.     /* critical section */
       varP = false;
     }
}
/* other code for process X */
```

Process Y

```
/* other code for
   process Y */
while (true)
{
  2. varQ = true;
  3. while (varP == true)
     {
4.     /* critical
          section */
       varQ = false;
     }
}
/* other code for
   process Y */
```

Here varP and varQ are shared variables and both are initialized to false. Which one of the following statements is true?

(A) The proposed solution prevents deadlock but fails to guarantee mutual exclusion.

(B) The proposed solution guarantees mutual exclusion but fails to prevent deadlock.

(C) The proposed solution guarantees mutual exclusion and prevents deadlock.

(D) The proposed solution fails to prevent deadlock and fails to guarantee mutual exclusion.

Soln.

The numbering 1 to 6 shows the order of execution.

Both the processes can enter the C.S, therefore, M.E is not guaranteed.

We know that M.E is a necessary condition for deadlock. That is deadlock requires mutual exclusion.

∴ Option (A) is correct.

Qn) Consider the following two-process synchronization solution.

Process 0	Process 1
Entry: loop while (turn == 1);	Entry: loop while (turn == 0);
(critical section)	(critical section)
Exit: turn = 1;	Exit: turn = 0;

The shared variable turn is initialized to zero. Which one of the following is TRUE?

(A) This is a correct two-process synchronization solution.

(B) The solution violates mutual exclusion requirement.

(C) The solution violates progress requirement.

(D) The solution violates bounded wait requirement.

Soln.

This solution is a Strict alternation.

(A) This not a correct 2 process synchronization solution. Strict alternation has its own set of problems.

∴ False

(B) Strict alternation guarantees mutual exclusion.

(C) Progress requirement. Processes which are in their Non CS should not block the processes which are there in the NC-S and wants to execute their C.S.

The process 0 while executing Exit Section (a Non CS) blocks process 1 from entering into its C.S.

∴ True

(D) The wait time of every process in the ready queue is bounded.

In strict alternation, the order of execution is maintained. Bounded waiting is guaranteed.

∴ False

Q.n) Consider the following solution to the producer-consumer synchronization problem. The shared buffer size is N. Three semaphores empty, full and mutex are defined with respective initial values of 0, N and 1. Semaphore empty denotes the number of available slots in the buffer, for the consumer to read from. Semaphore full denotes the number of available slots in the buffer, for the producer to write to. The placeholder variables, denoted by P, Q, R and S, in the code below can be assigned either empty or full. The valid semaphore operations are: wait() and signal().

Producer:

```
do{
    wait(P);
    wait(mutex);
    // Add item to
    // buffer
    signal(mutex);
    signal(Q);
}
while(1);
```

Consumer:

```
do{
    wait(R);
    wait(mutex);
    // Consume item from
    // buffer
    signal(mutex);
    signal(S);
} while(1);
```

Which of the following assignments to P, Q, R and S will yield the correct solution?

(A) P: full, Q: full, R: empty, S: empty

(B) P: empty, Q: empty, R = full, S: full

(C) P: full, Q: empty, R = empty, S: full

(D) P: empty, Q: full, R: full, S: empty.

Soln.

Here, the initializations are done opposite for Full and Empty.

Here, E = 0 and F = N.

wait(P) is actually wait(Full)

signal(Q) is actually signal(Empty)

wait(R) is actually wait(Empty)

signal(S) is actually signal(Full)

∴ Option (C) is correct.

Qn:) Consider three concurrent processes P_1, P_2 and P_3 as shown below, which access a shared variable D that has been initialized to 100.

P_1	P_2	P_3
:	:	:
D = D + 20	D = D − 50	D = D + 10
:	:	:

The process are executed on an uniprocessor system running a time-shared operating system. If the minimum and maximum possible values of D after the three processes have completed execution are X and Y respectively, then the value of Y-X is _____.

Soln.

X: min
Y: max

$D = D + 20$ =

Load operation
- copy R_1 M[D]
- ADD R_1 20
- STORE M[D] R_1

} Non-atomic (can be preempted)

For minimum value only D = D-50 can subtract to give minimum value.

The final value of D depends on whichever of the processes store the last value.

P_2	P_1	P_3
1. $R_2 = 100$	3. Load	6. Load
	4. Add	7. Add
2. ⟶ preempted	5. Store	8. Store
9. Add		
10. Store		

The numbering gives the orders of execution.

Minimum value of X = 50.

For maximum value (Y):

1. P1 reads D=100, preempted
2. P2 reads D=100, executes, D = D-50 = 100-50 = 50.
3. Now, P1 executes D = D+20 = 100+20 = 120
4. And now, P3 reads D = 120,
   executes D = D+10, D = 130.
   P3 writes D=130 final value.

∴ Y-X = 130 - 50 = 80.

Qn) Consider the following snapshot of a system running n concurrent processes. Process i holding X_i instances of a resource R, $1 \le i \le n$. Assume that all instances of R are currently in use. Further, for all i, process i can place a request for at most Y_i additional instances of R while holding the X_i instances it already has. Of the n processes, there are exactly two processes p and q such that $Y_p = Y_q = 0$.

Which one of the following conditions guarantees that no other process apart from p and q can complete execution?

(A) $X_p + X_q < \text{Min}\{ Y_k \mid 1 \le k \le n, k \neq p, k \neq q \}$

(B) $\text{Min}(X_p, X_q) \geq \text{Min}\{ Y_k \mid 1 \le k \le n, k \neq p, k \neq q \}$

(C) $\text{Min}(X_p, X_q) \le \text{Max}\{ Y_k \mid 1 \le k \le n, k \neq p, k \neq q \}$

(D) $X_p + X_q < \text{Max}\{ Y_k \mid 1 \le k \le n, k \neq p, k \neq q \}$

Answer (A)

X_i = Holding resources for process P_i.

Y_i = Additional resources for process P_i.

As process p and q doesnot require any additional resources, it completes its execution and available resources are $(X_p + X_q)$

There are $(n-2)$ process P_i $(1 < i < n, i \neq p, q)$ with their requirements as Y_i $(1 < i < n, i \neq p, q)$

In order to not execute process P_i, no instance of Y_i should be satisfied with $(X_p + X_q)$ resources, ie, minimum of Y_i instances should be greater than $(X_p + X_q)$.

Q1. Consider a counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12 V(S) operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _____

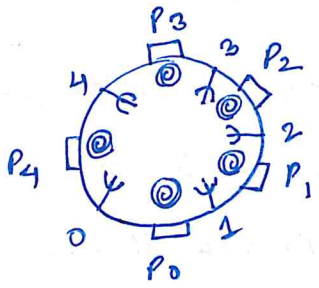
(A) 7
(B) 8
(C) 9
(D) 10

Soln.

20 − 7 = 13 will be in the blocked state, when we perform 12 V(S) operations makes 12 more process to get chance for execution from blocked state. So one process will be left in the queue (blocked state) here we have considered that if a process is in CS, it will not get blocked by other process.

Dining philosopher's problem.

↳ By Dijkstra.



Note that semaphore as concept was introduced by Dijkstra. He wanted to understand the power of semaphore and how they are to be used. It is a theoretical problem But very interesting problem.

It doesnot map exactly to real world problems, but it tells us about the complexity that is involved, in designing proper or well synchronized processes which do not have deadlocks using constructs like semaphores.

we have a bunch of philosophers,

$P_0, P_1, P_2, \ldots P_{n-1}$; $n \geq 2$

we need atleast 2 philosophers

The philosophers can do 2 things :-

1. They can think
2. They can eat

They think on some theoretical philosophy problem.
Whenever they are hungry, they eat.
To eat, they need to pick the fork that is to the left of him/her, they also need to pick the fork that is to their right.

They need both the forks to eat the food.
We can think of the forks as resources.
There are 5 philosophers and five forks.
Therefore, every philosophers can't eat.
Note: There are 5 plates with food on the table.

We want to come up with a deadlock free synchronization mechanism for this problem.

Solution 1:

```
Philosopher(int i)
{
  while (true)
  {
    think(i);
    take fork(i);
    take fork((i+1) mod n);
    eat(i);
    put fork(i)
    put fork((i+1) mod n);
  }
}
```

Suppose i=0,

→ i starts to think
→ i gets hungry, picks up left fork
→ then i picks up right fork
→ then i eats
→ once done with eating, put left fork down
→ then put right fork down.

Let's check if the solution is good.

In the world of multiple philosophers, say philosopher 0 picks up left fork after he think and when he gets hungary.

Let's say P_0 gets preempted now and control goes to P_1.

P_1 starts thinking and when hungry P_1 takes the left fork and gets preempted and control goes to P_2. In this similar fashion, P_3 and P_4 also takes the left fork, say control comes back to P_0.

∴ P_0 is waiting for P_1 to give the right fork. P_1 is waiting for P_2 to give the right fork. And similarly for the rest. And P_4 is cyclically waiting for P_0 to give up P_4's right fork.

∴ All of the processes are in a deadlock.

Therefore, we would like to think of solutions without using semaphores.

Non-semaphore based solutions:-

pick right fork then pick left fork.

This solution doesnot work. Even this leads to deadlock.

Another Non-semaphore based solution:

Say 0 to n-2 philosophers pick left then they right fork

While the $(n-1)^{th}$ or the last philosopher picks right fork and then left fork.

In this scheme, when philosopher 3's turn comes again, he picks the right fork and finishes eating and places both the forks back. Then P_2 can pick the right fork and complete his meal and then place both the forks back on the table. This way, we can avoid the cyclic wait.

This scheme has no deadlock.

Third non-semaphore based strategy:

Let odd number philosopher pick left fork and then right fork.

While let even numbered philosopher pick Right fork first and then left fork.

Here,

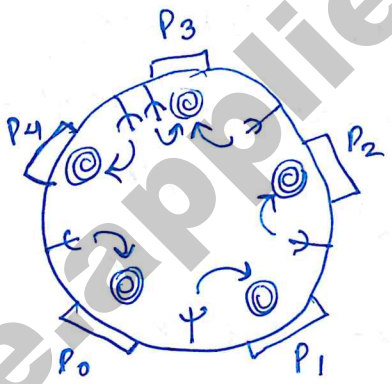
P_1 is depending on P_0 to give up.

P_2 picks up Right fork and then left fork.

In this fashion we can avoid deadlock.

Q. In standard solution of dining philosophers problem how many more forks is needed to avoid a deadlock?

Soln.

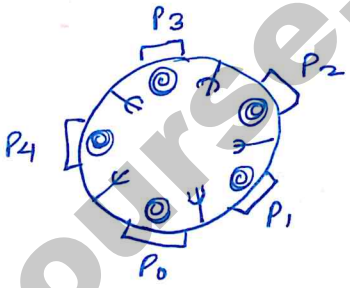


By adding one more fork in between any two philosophers, we can avoid deadlock.

Semaphore based solution:

```
int left(i)
{
  return (i-1+n) mod n;
}
int right(i)
{
  return (i+1) mod n;
}
```

```
void Philosopher(int i) {
  while(true) {
    think(i);
    takeforcks(i);
    eat(i);
    putforcks(i);
  }
}
```



n = 5

THINKING = 0

HUNGRY = 1

EATING = 2

BSemaphore m = 1

BSemaphore S[n] = {0}

int state[n] = {THINKING}

shared variable

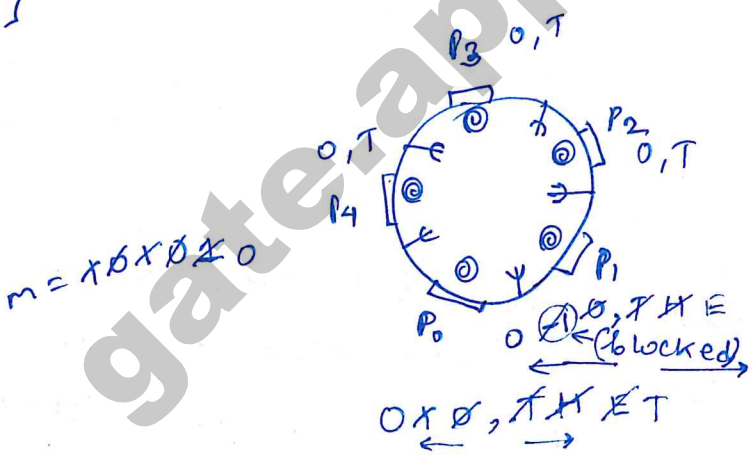
left	right
$P_4 \leftarrow P_0$	$\rightarrow P_1$
$P_3 \leftarrow P_4$	$\rightarrow P_0$

We have to be careful about takeforcks() and putforcks() because forcks are the resources due to which we can encounter deadlock like situations, if allocated in an incorrect manner.

```
takeforks (i)
{
    DOWN(m);
    state[i] = Hungry;
    test(i);
    UP(m);
    DOWN(S[i]);
}
```

```
test(i)
{
  if state[i] == HUNGRY
  AND state[left(i)] != EATING
  AND state[right(i)] != EATING)
  {
     state[i] = EATING;
     UP(S[i]);
  }
}
```

```
Putforks(i)
{
    DOWN(m);
    state[i] = THINKING;
    test[left(i)];
    test[right(i)];
    UP(m);
}
```



The semaphore based implementation is deadlock free.

The DOWN(m) and UP(m) in takeforks() and putforks() make sure no to philosophers can take or put forcks simultaneously.

The test() logic is to block a process (philosopher) if its left or right is not available.

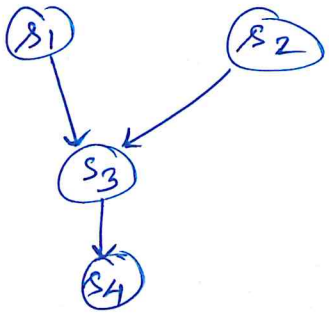
Concurrent Programming: Parbegin-Parend Model

Concurrent Programming: If we can execute two or more statements parallelly. Imagine we have a system with multiple CPUs and some of the CPUs are sitting idle, in that scenario, S1 could run on one processor while statement S2 can run concurrently run on another processor, as S1 and S2 have no dependencies.

S1: a = b+c → CPU 1
S2: d = e*f → CPU 2
S3: k = a+d
S4: l = k*m

Note: S1 and S3 can't be executed on 2 different CPUs as S3 requires variable a which has been changed by S1.

At the end of the day, the result that we get from sequential execution, should be the same result given by the concurrent execution of the statements.



precedence graph

Precendence graph is a visual way of understanding what statements can be run parallelly and what statements can't be run parallelly / concurrently.

Bernstein's conditions for concurrency:

Readset and writeset concept:

① S1: a = b + c;
Readset = { b, c }
writeset = { a }

② S2: a += ++b * --c;
Readset = { a, b, c }
writeset = { b, c, a }

③ S3: scanf("%d", &a);
Readset = ϕ, writeset = { a }

Note:

Readset are all the variables that are being read and used in the statement.

writeset consists of all the variables to which we write to.

The 3 conditions of Bernstein are :-

① $R(S_i) \cap W(S_j) = \phi$

② $R(S_j) \cap W(S_i) = \phi$

③ $W(S_i) \cap W(S_j) = \phi$

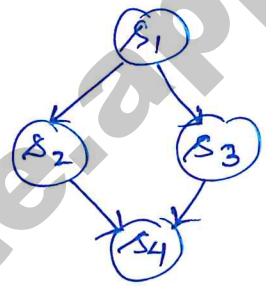
} If these 3 conditions are met, then we can concurrently execute S_i and S_j.

Note:

$R(S_i) \cap R(S_j)$ need not be ϕ.

That is, S_i and S_j can read the same variable.

E.g) Precedence graph:



Multi-CPU system:

CPU 1	CPU 2
S1	
↓ S2	S3
S4	

Single CPU system:

S1
↓ S2 or S3 } S2 or S3, S3 or S2
S3 or S2 }
S4

Programming constructs for concurrent programming:

[parbegin-parend (or) cobegin-coend]

I:

parbegin → parallel
parend
cobegin → concurrent
coend

Different languages give us different programming constructs.

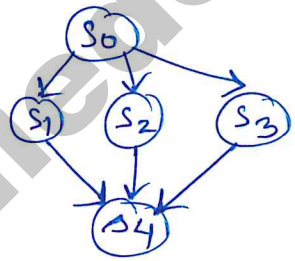
e.g:)

```
{                begin
   S1;                S1;
   S2;                S2;
   S3;                S3;
}                end
```

---

e.g:)

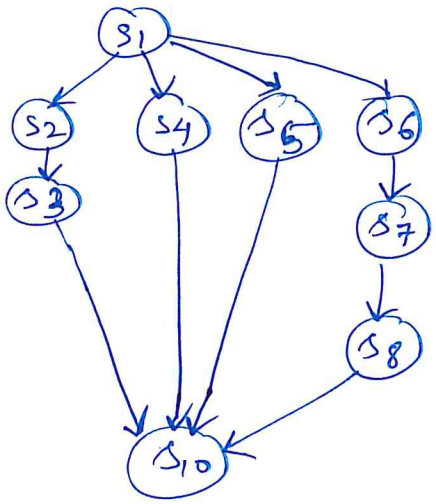
```
S0;
parbegin:
        S1;
        S2;
        S3;
parend
        S4;
```



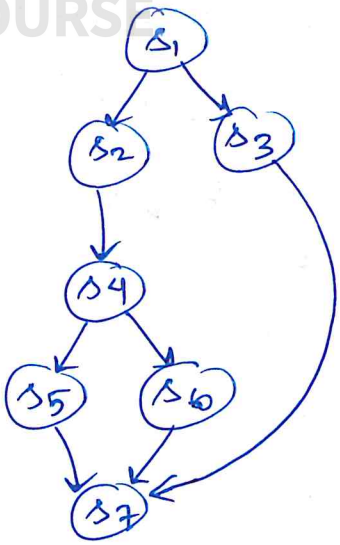
---

e.g:)

```
S1;
parbegin:
   begin S2;S3; end
        S4;
        S5;
        begin:
              S6;
              S7;
              S8;
        end:
parend:
   S10;
```



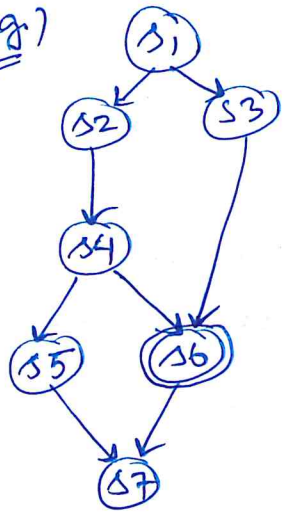
e.g)




```
S1;
parbegin
    S3;
    begin
        S2;
        S4;
        parbegin
            S5;
            S6;
        parend
    end
parend
S7;
```

---

e.g)



We can't write the standard parbegin-parend.

But using semaphores along with parbegin-parend we can write concurrent/parallel programming code for solving this problem.

II. parbeginAparend + semaphores:

Bsemaphores a, b, c, d, e, fg = 0;

P: DOWN
V: UP

parbegin:

```
Begin S1; V(a); V(b); end
begin P(a); S2; S4; V(c); V(d); end
begin P(b); S3; V(e); end
begin P(c); S5; V(f); end
begin P(d); P(e); S6; V(g); end
begin P(f); P(g); S7; end
```

parend

(Q) void P()
{
A;
B;
C;
}
void Q()
{
D;
E;
}
main()
{
parbegin
P()
Q()
parend
}

which sequences are possible?

① A, B, C, D, E ✓ } P() Q()
② D, E, A, B, C ✓ } Q() P()
③ A, B, D, C, E ✓
④ D, A, E, B, C ✓
⑤ A, B, E, C, D ✗
⑥ E, C, D, B, A ✗

(Q) int x=0, y=0;

cobegin
begin
x=1;
y=y+x;
end
begin
y=2;
x=x+3;
end
coend

what are the possible outcomes?

① x=1; y=2 ✗
② x=1; y=3 ✓
③ x=4; y=6 ✓

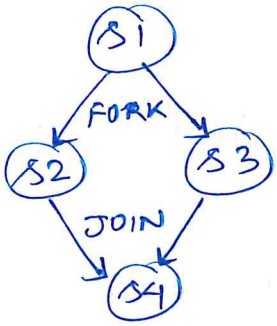
x	y
3	2
1	3

sequence:
y=2
x=x+3
x=1
y=y+x

x	y
1	2
4	6

sequence:
x=1
y=2
x=x+3
y=y+x

III. Fork and Join Model of Concurrency



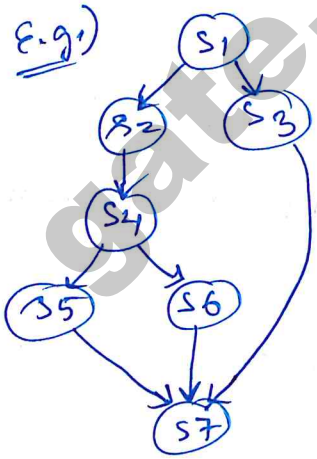

```
P1:                          P2:
count = 2;
S1;
fork L;
S2;
goto L1;
L: S3;                       S3
L1: join count
S4;
```

Implementation of Join could be different on different operating systems and different programming languages.

join count or join P_1, P_2 (process ids)

where count is the number of concurrent processes that we can join.

E.g.:

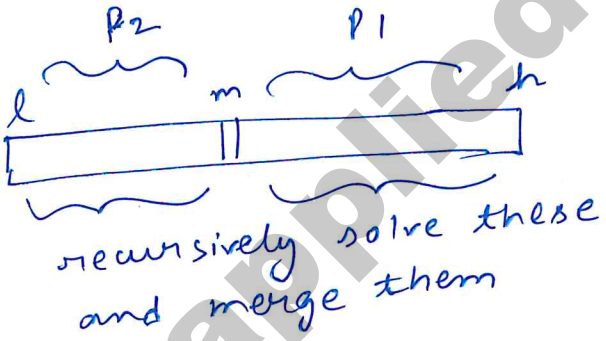



```
P1:                  P2:          P3:
COUNT = 3;
S1;
fork L1;
S3;
goto L3;
L1: S2;              S2;
    S4;              S4;
    fork L2;
    S5;              S5;
    goto L3;
L2: S6;                           S6;
L3; join count
S7;
```

Real world example :

(source: wikipedia)

```
mergesort(A, lo, hi):
    if lo < hi: //atleast one element of input
        mid = [lo + (hi - lo)/2]
        fork mergesort (A, lo, mid)
            // process (potentially) in parallel
            // with main task
        mergesort (A, mid, hi)
            //main task handles second
            //  recursion
        join
        merge(A, lo, mid, hi)
```



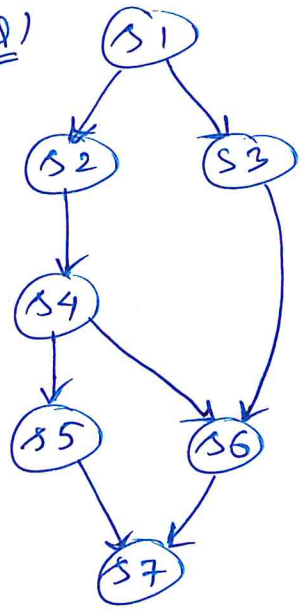
The sorting of l to m will be done on P2 and sorting of m to h will be done by process P1. It is possible because the sorting of the above sub parts are independent of each other.

When P2 and P1 completes sorting, they both will arrive at the join.

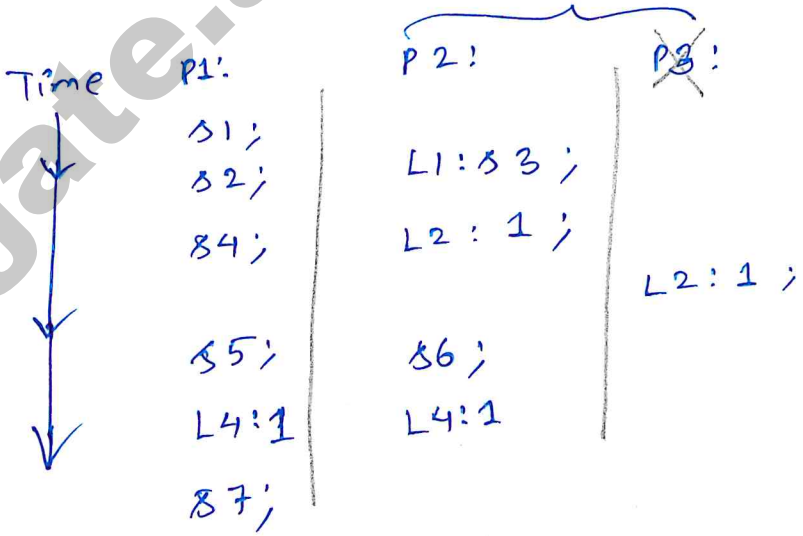
Then P1 will do the merge.

Lot of parallel processing and distributed systems use something very similar to this.

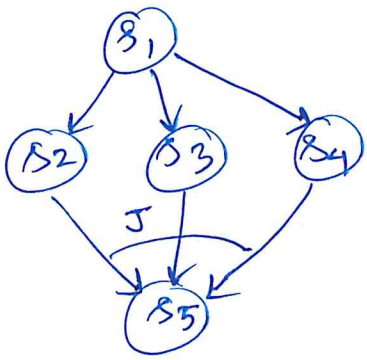
Q)



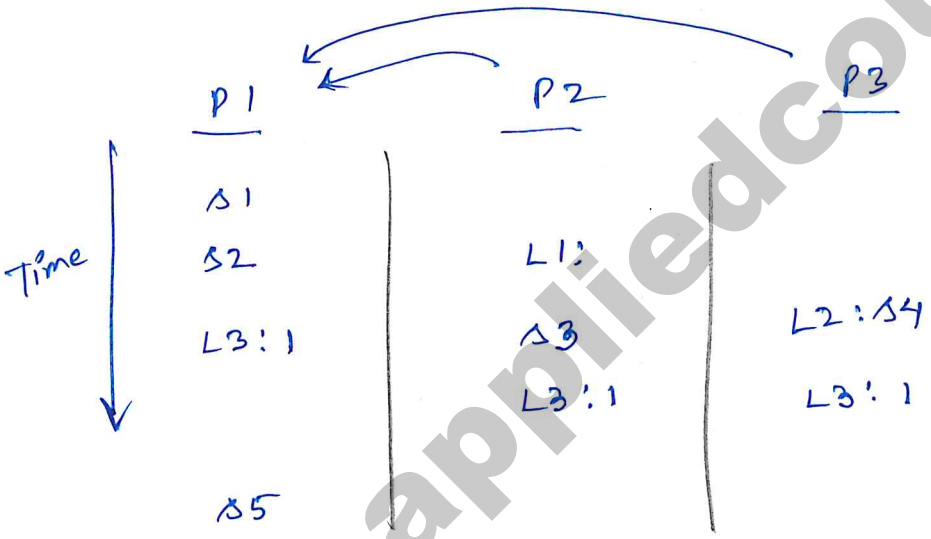

```
count1 = 2
count2 = 2
S1;
fork L1;
S2;
S4;
fork L2;
S5;
goto L4;
L1: S3;
L2: join count1;
S6;
L4: joint count2
S7;
```



Q.1



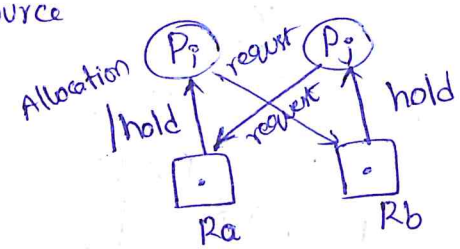

```
count = 3
S1;
fork L1;
S2;
goto L3;
L1: fork L2;
S3;
goto L3;
L2: S4;
goto L3;
L3: join count
S5;
```

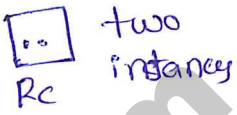


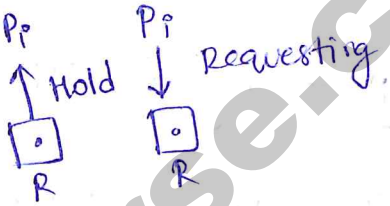
Process Management - Deadlocks and Threads

Deadlocks: cpu, Resource → s/w, H/w

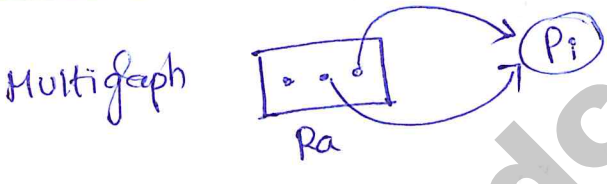
↳ Two or more processes wait forever for an Event to happen which will never happen.



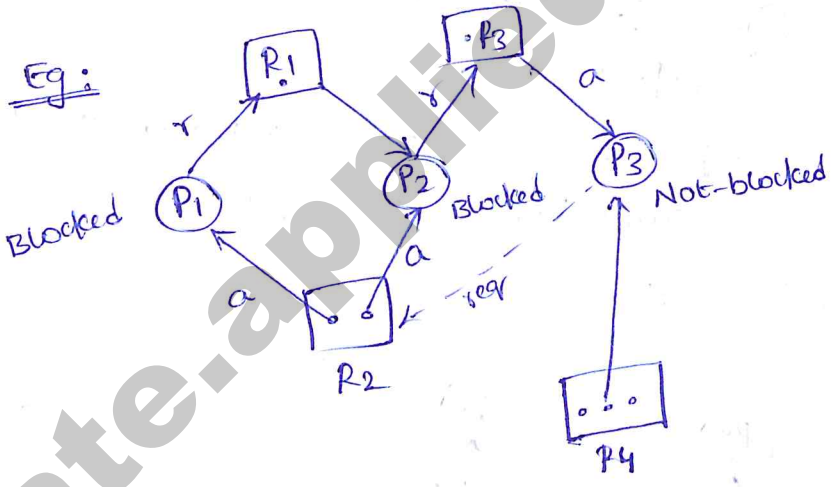




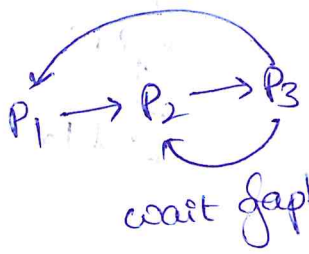
Resource Allocation Graph: (RAG)

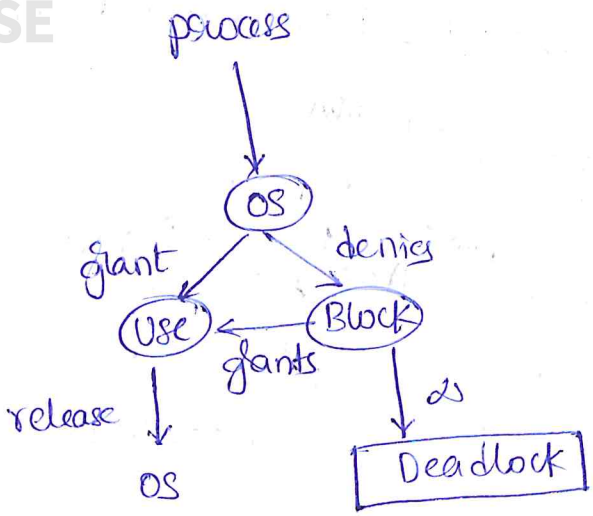


Eg:



∴ If P3 requests instance of R2 then P3 will also in Blocked state.





Necessary Conditions for Deadlocks:

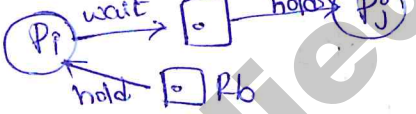
Dead Lock ⇒ 1,2,3,4

1,2,3,4 ⇏ Deadlock

① Mutual Exclusion → CS

{ shared-resources
(cpu, var, printer, semaphore)

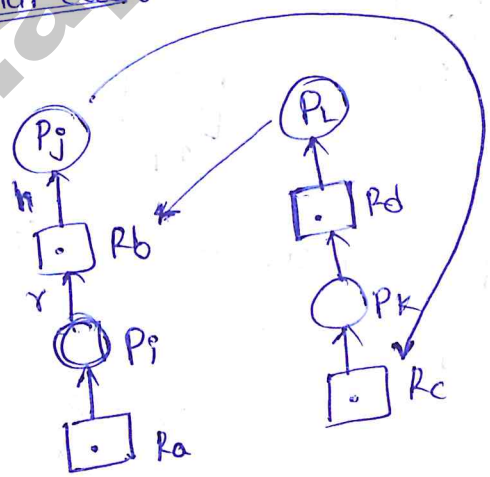
② Hold and wait:

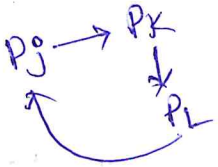


③ No preemption: processes are not allowed to grab resources held by other processes.

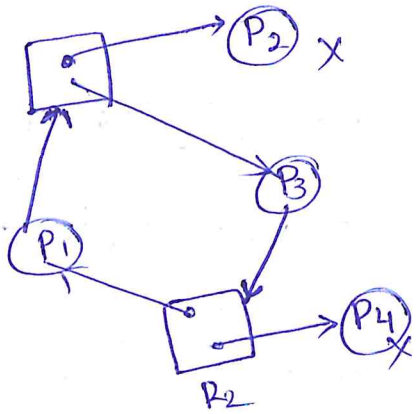
④ Circular wait

Eg:





Eg:



∃ cycle
No deadlock

P_2 is not blocked. As soon as P_2 is completed, it releases the resources. Then P_1 will finish, followed by P_3 and so on.

→ RAG with Multi-instance resource, then cycle in RAG need not imply deadlock.

→ If RAG with only single instance resource, then cycle is a necessary and sufficient for deadlock.

Deadlock-handling Methods:

1. prevention
2. Avoidance
   ↓
   Banker's Algorithm.

Detect & Recover
Ignore — early unix, Minix

deadlocks can happen

Ignorance : Ostrich Algorithm.
↓
Bury their Heads

Prevention:

1. Mutual Exclusion
2. Hold & wait
3. No preemption
4. Circular wait

Deadlock (1 & 2 & 3 & 4) } logic

¬(1 & 2 & 3 & 4) ⇒ ¬ Deadlock

¬1 V ¬2 V ¬3 V ¬4

A → B
¬B → ¬A

⇒ ¬ Deadlock

Mutual Exclusion : Shared Resources .. CS.

→ NOT dissatisfiable.

Prevention : ! (Hold & wait) = ! Hold V ! wait
(a)

(a) Resource Allocation @start

P1 : ≡ ← request R1, R2, R3
request R1
request R2
≡
release R1
request R3
≡
release R2 & R3

R2, R4
P2
P3
P4
⋮

phase1
R2 phase2

Highly inefficient

STARVATION

No wait

(b) Release All the resources before requesting new resources.

P1 : ≡
request R1
request R2
≡
release R1
request R3
≡
release R2 & R3

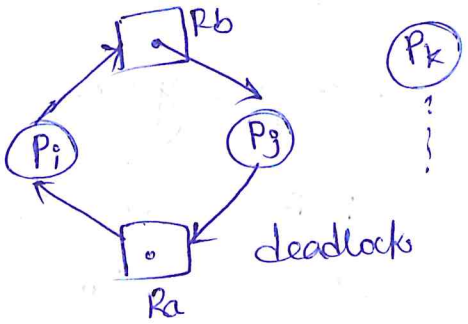
R1, R2 phase1
R2 & R3
R2

STARVATION
P2, P3

Release both R1 & R2 then request for both R2 & R3

Prevention: !(No preemption) = preemption

(a) Forceful preemption



(b) Graceful/ self-preemption

{ STARVATION

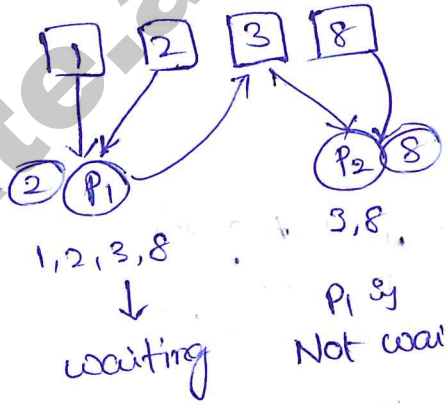
Prevention: !(circular-wait) : STARVATION

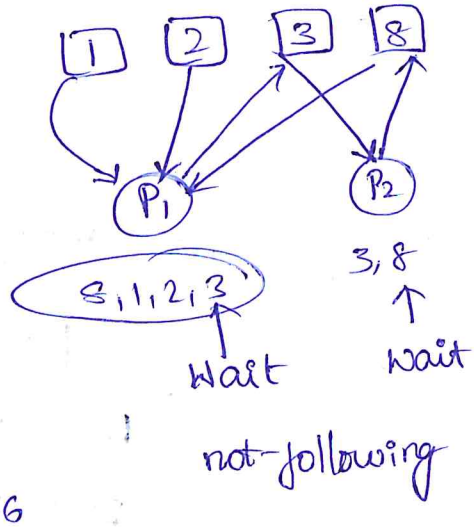
R1 : 4
R2 : 8
R3 : 9
R4 : 3
R5 : 5
R6 : 12
R7 : 10

Pi : R4, R3, R7, R2 → processes
3 < 9 < 10 8

Pj : R4, R2, R3, R7
3 8 9 10

Request resource in increasing order of ID's.





Total order : 1 < 2 < 3 < 4 < 5 < 6
↳ among the processes
$P_1 < P_2$
2 8

Avoidance: Banker's Algorithm

① State of a System:

$P_1, P_2, \ldots P_n \Rightarrow n$ processes

$R_1, R_2 \ldots R_m \Rightarrow m$ resources

- max [n][m]
- Allocation [n][m]
- Need [n][m]
- Available [m]

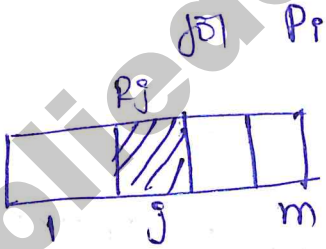
max[i][j] — P_i R_j — maximum number of Resource

Limitations

Allocation[i][j] ⇓ Number of instance of R_j is allocated to process P_i

Need[i][j]: Instance of type R_j is still needed

Available[m]:



How many resources of R_j is available now.

Eg: n=5, m=1

	MAX R	Alloc R	Need R	Available R
				(2)
P_1	10	5	[5]	
P_2	5	2	[3]	
P_3	12	6	[6]	
P_4	20	10	[10]	
P_5	8	4	[4]	

Safe & Unsafe States

Unsafe state.

With the available resource, we cannot complete any one of the process.

Safe & Unsafe State

If the needs of all processes can be satisfied in some order. → Safe Sequence

Eg:

	MAX A B C	Alloc A B C	Need A B C	Available A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2
✓ P_1	3 2 2	2 0 0	1 2 2	5 3 2
P_2	9 0 2	3 0 2	6 0 0	7 4 3
✓ P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

Safe-State

$(3,3,2) \geq (1,2,2)$

Avail ≥ Req1

Safe sequence { P_1, P_3, P_0, P_2, P_4 ; P_1, P_3, P_2, P_4, P_0

Safety Algorithm:

1. Initialize work = available
2. Finish[i] = False ; i = 1 to n
3. Find i such that
   Finish[i] = false
   and Need[i][j] ≤ work[j] ∀ j = 1 → m
   → if not found goto 5:

4. Work = Work + Allocation [i]

Finish [i] = TRUE

GOTO 3

5. if finish [i] = TRUE ∀i then

safe-state

else

unsafe state

Resource Request Algorithm:

1. Request$_i$ ≤ Available ✓
2. Request$_i$ ≤ Need$_i$ ✓ sanity check
3. Assume we satisfy the request
   - (a) Available = Available − Request$_i$
   - (b) Need$_i$ = Need$_i$ − Request$_i$
   - (c) Alloc$_i$ = Alloc$_i$ + Request$_i$
4. Run Safety-algorithm
5. If safe, grant request$_i$ else Block(P_i)

	MAX A B C	Alloc A B C	Need A B C	Available A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 7 → 2 3 7 → 2 1 0
P_1	3 2 2	2 0 0	1 2 2	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

t1: (a) P_1 asks for Allocate (1,0,2)

t2: (b) P_4 asks (3,3,0)

t3: (c) P_0 asks (0,2,0) blocked

unsafe

Bankers Algorithm:

→ Always in Safe-State

deadlock ⇒ Unsafe ⇒ ¬ Unsafe ⇒ ¬ deadlock

Unsafe ⇏ deadlock

⇓

Safe

Limitations:

① MAX[n][m] → Not very realistic

② n changes dynamically

⇓

Processes

More Solved problems-1

① In a System, there are three types of resources: E, F and G. Four processes P_0, P_1, P_2 and P_3 execute concurrently. At the outset, the processes have declared their maximum resource requirements using a matrix named Max as given below. For example MAX[P_2, F] is the maximum number of instances of F that P_2 would require. The number of instances of the resources allocated to the various processes at any given state is given by a matrix named allocation.

Consider a state of the System with the allocation matrix as shown below, and in

which 3 instance of E and 3 instance of F are the only resource available.

From the perspective of deadlock avoidance, which one of the following is True?

(A) The System is Safe State

(B) The System is not in Safe State, but would be safe if one more instance of E were available.

(C) The System is not in Safe State, but would be safe if one more instance of F were available.

(D) The System is not in Safe State, but would be safe if one more instance of G were available.

Sol:

Need

3 3 0 ←
1 0 2 ←
0 3 0 ←
3 4 1 ←

Allocation

	E	F	G
P_0	1	0	1
P_1	1	1	2
P_2	1	0	3
P_3	2	0	0

MAX

	E	F	G
P_0	4	3	1
P_1	2	1	4
P_2	1	3	3
P_3	5	4	1

P_0, P_1, P_2, P_3 ⟹ Safe Sequence

Available

E F G
~~3 3 0~~
~~4 3 1~~
5 4 3

(2) Consider a system with 3 processes that share 4 instances of the same resource type. Each process can request a maximum of k instances. Resource instances can be requested and released only one at a time. The largest value of k that will always avoid deadlock is

---

Sol:

	max	Alloc	
P_1	k	k-1	1
P_2	k	k	2
P_3	k	k-1	1

R[4]

$k \geq 1$

$k \leq ?$

$3k - 2 \leq 4$

$k \geq 6/3 \leq 2$

$k = [2]$ No deadlock

(3) An operating system uses the Banker's Algorithm for deadlock avoidance when managing the allocation of three resource types X, Y and Z to three processes P_0, P_1 and P_2. The table given below presents the current system state. Here the allocation matrix shows the current number of resources of each type allocated to each process and the max matrix shows the maximum

number of resources of each type required by each process during its execution.

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in a safe state.

Consider the following independant requests for additional resources in the current state.

REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z

REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z.

Which of the following is TRUE?

(A) Only REQ_1 can be permitted

(B) Only REQ_2 can be permitted

(C) Both REQ_1 and REQ_2 can be permitted.

(D) Neither REQ_1 nor REQ_2 can be permitted.

	Allocation			MAX			Need	Avail
	X	Y	Z	X	Y	Z	X Y Z	
P_0	0	0	1 3	8	4	3	(8,4,0)	3 2 2
P_1	3	2	0	6	2	0	(3,0,0)	2 2 0
P_2	2	1	1	3	3	3	(1,2,2)	

Now Avail

6 4 0 P1

REQ1: if P_0 : (0,0,2) is allocated

P_0 & P_2 cannot be completed. ⇒ unsafe sequence / state

REQ 2: $P_1(2,0,0)$ Avail: (3,2,2)

$\Downarrow$

P_1 5 2 0

		Alloc	MAX
Need		X Y Z	X Y Z
(3,4,2)	P_0	0 0 1	8 4 3
(1,0,0)	P_1	5 2 0	6 2 0
(1,2,2)	P_2	2 1 1	3 3 3

~~P_1~~ ~~P_2~~ P_3

Avail: 6,4,2

8,5,3

(4) A system containg three programs and each requires three tape units for its operation. The minimum number of tape units with the system must have such that deadlocks never arise. is

	MAX	Alloc
P_1	3	2
P_2	3	2+1
P_3	3	2

R ?

$R \geq 3$

if we have 7 Resources.

3×2+1

(5) A System has 6 identical resources and N processes competing for them. Each process can request atmost 2 resources. which one of the following values of N could lead to a deadlock?

X (A) 1

X (B) 2

X (C) 3

X (D) 4

	MAX	Alloc
P_1	2	1
P_2	2	1
P_3	2	1
P_4	2	1
P_5	2	1
P_6	2	1

	MAX
P_1	2
P_2	2
:	:
P_N	2

Avail 6

$\Rightarrow$ P=6 Lead to a deadlock.

(6) Consider the following policies for preventing deadlock in a System with mutually exclusive resources.

1. processes should acquire all their resources at the begining of execution. If any resource is not available all resources acquired so far are released. — ! (Hold & wait)

2. The resources are numbered uniquely, and processes are allowed to request for resources only in increasing resource number. — ! (Circular wait)

3. The resources are numbered uniquely, and processes are allowed to request for resources only in decreasing resource number.

4. The resources are numbered uniquely. A process is allowed to request only for a resource number larger than its currently held resources.

which of the above policies can be used for preventing deadlock?

✗ (A) Any one of I and III but not II or IV.

✗ (B) Any one of I, III and IV but not II

✗ (C) Any one of II and III but not I or IV

✓ (D) Any one of I, II, III and IV

(7) A system shares 9 tape drives. The current allocation and maximum requirement of tape drives for 3 processes are shown below.

Which of the following best describes the current state of the system.

(A) Safe, Deadlocked

(B) Safe, Not Deadlocked

(C) Not Safe, Deadlocked

(D) Not Safe, Not Deadlocked.

Process	Current Allocation	Max Requirement	Need
P_1	3	7	4
P_2	1	6	5
P_3	3	5	2

Avail : 2 3 3

P_3, P_1, P_2

Deadlock ⇒ unsafe

Safe ⇒ ¬ Deadlock

Detection & Recovery :

(1) Single Instance Resource → Cycle in Wait-for Graph

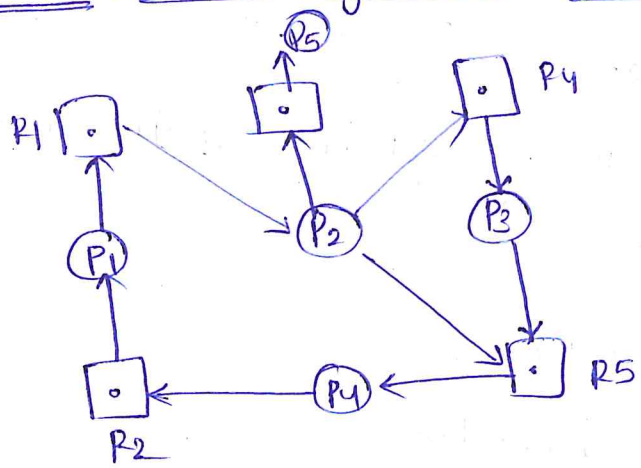
(2) Multi Instance Resource. → Modified safety Algorithm

When to apply deadlock detection ?

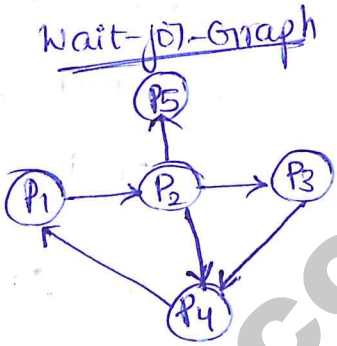
↳ Symptoms : low cpu utilization

Majority of processes are blocked.

Detection with Single Instance Resource



RAG



If the wait for Graph has a cycle

↓

Deadlock

Detection with Multi-instance Resource:

		Alloc A B C	Req A B C	Avail A B C
	P_0	0,1,0	0 0 0	0 0 0
NB →	P_1	2,0,0	2 0 2	
B →	P_2	3,0,3	0 0 0	
NB →	P_3	2,1,1	1 0 0	
B →	P_4	0,0,2	0 0 2	

Total Resource A B C
7,2,6

Safty Algorithm

3-B
2-NB
} Majority of processes are blocked.

safe ⇒ No-dead Lock.

Safe Sequence P_0, P_2, P_3, P_4, P_1

Avail 0 0 0
(0+1+0) 5,2,6
~~3,1,3~~
~~5,2,4~~

Small changes in Table

	Alloc A B C	Req A B C	Avail A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Total Resource

7, 2, 6

Safe sequence: P_0

Avail: 0 1 0

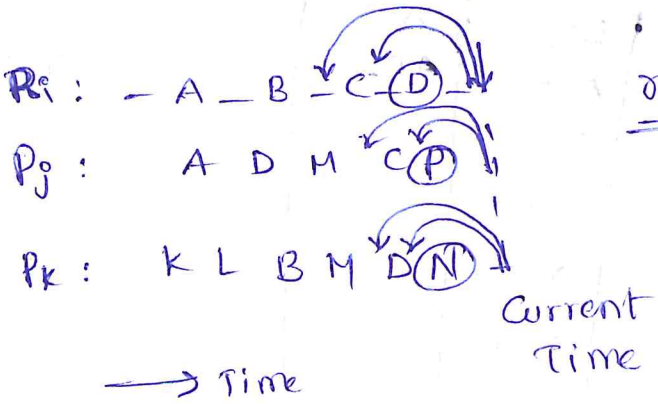
No safesequence: With the available resource No process willbeable to finish the execution.

Recovery Mechanisms:

- Process Termination
  - Killall → restart $P_0, P_1 \ldots P_4$, drawback: Utilization ↓ starvation
  - Kill one at a time
    - $P_1, P_2, P_3 \ldots [P_i] \ldots P_{10}$
    - cpu utilization
    - recency
    - kills & Restarts
- Resource preemption → Force processes to give up resources. → STARVATION

STARVATION

Recovery : Preemption



rollback

After one Rollback, if still deadlock exist in the system pre-empt the last two and rollback.

(Q) $P_1, P_2 \dots P_n$

R : 6

Each process needs 2 instances of R.

n, n+1 forks

Dining philosophers

Find Max-value of n to avoid a deadlock.

5

n=5 P_1 P_2 P_3 P_4 P_5 ⇒ No deadlock

1 1 1 1 1

⇒

1

If n=6

P_1 P_2 P_3 P_4 P_5 P_6

1 1 1 1 1 1

6 ⇒ deadlock

5 ⇒ No deadlock

(Q) $P_1, P_2 - - - P_n$

R : 6

each process needs 3 instances of R

Find max value of n to avoid a deadlock.

P_1 P_2 P_3 $\quad n=3$

2 2 2

$P_1, P_2 \Rightarrow R \div 2$

2 2 $\quad N=2$ No deadlock

$N=4$

P_1 P_2 P_3 P_4

1 1 2 2

Largest value of n, where there is no deadlock $= \boxed{2}$

(Q)

	Max R	
P_1	10	$\rightarrow 9$
P_2	15	$\rightarrow 14+1$
P_3	3	$\rightarrow 2$
P_4	8	$\rightarrow 7$
P_5	12	$\rightarrow 11$

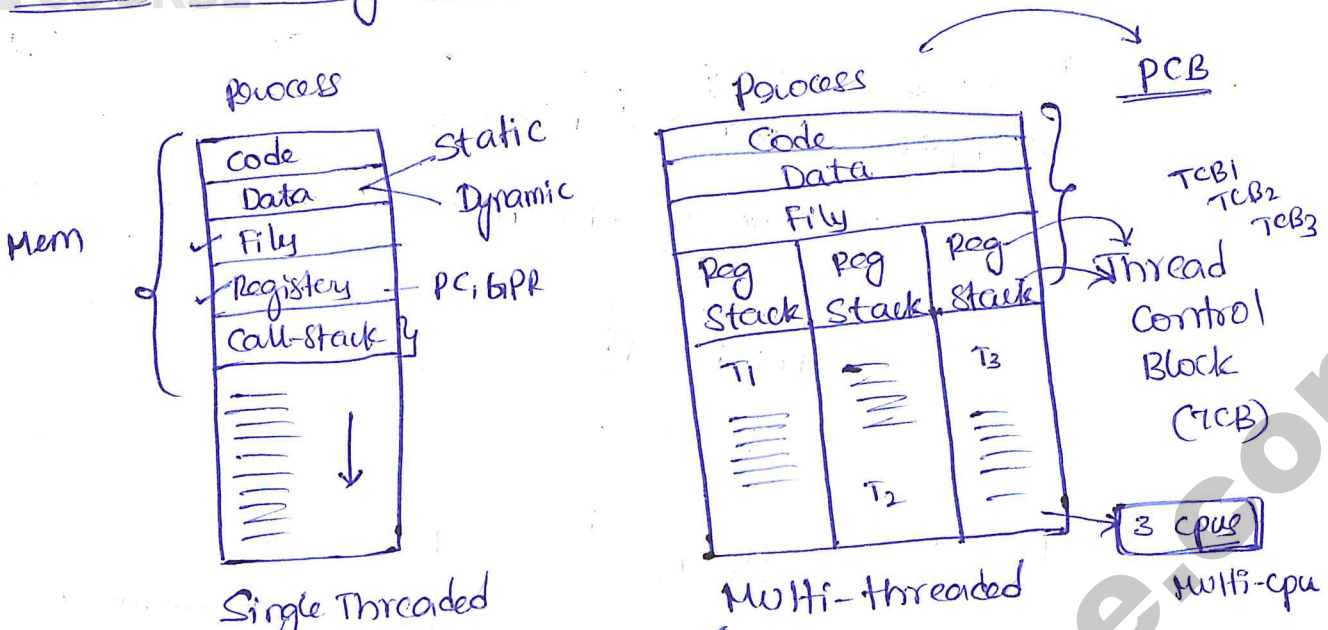
$43+1 = (44)$

Min # of instances of R to avoid a deadlock.

Ex

Min # of instance of R to avoid a deadlock is $\boxed{44}$

Multi-Threading : An Introduction



Google Chrome
100 MB
Running 30 threads parallely.

Multi-cpu System

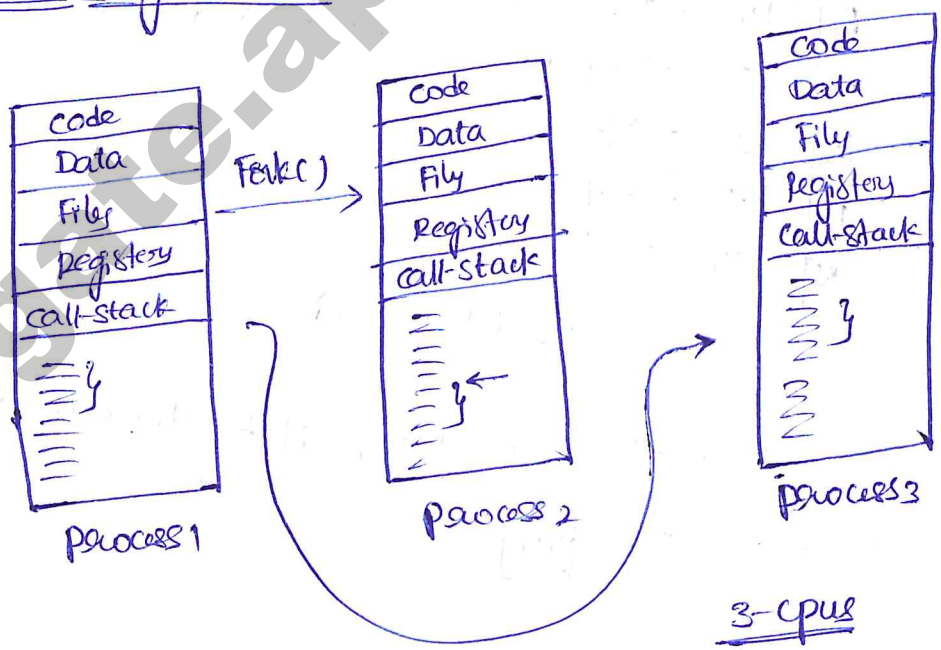
Intel 50%

1990's JAVA
Inbuilt Multithreading

Multi-tasking : Fork()

shared-Memory

Semaphores → S/W Resource



Multi-tasking vs Multi threading (1990's)

1. code & Memory sharing
2. Better cpu utilization
3. Easier access to shared-data.
4. Faster to Context-Switch.
5. programmer-control & ease

Assume single CPU

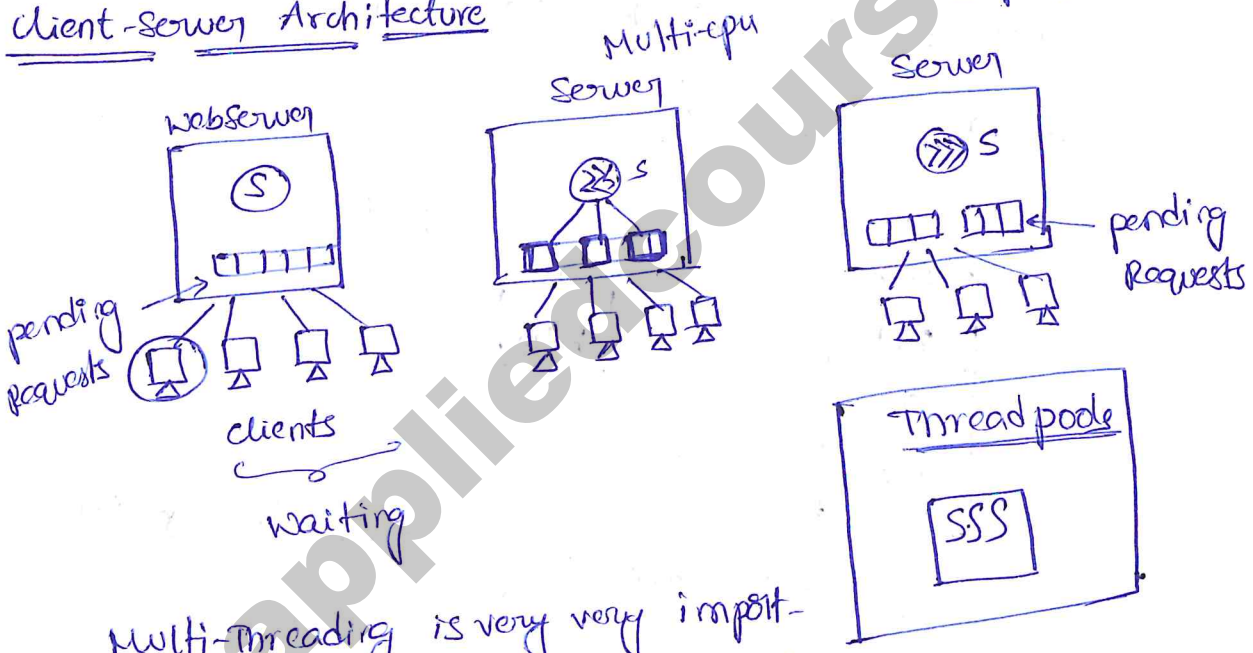
No Need to Context switch between the processes. Other threads will Continue their Execution.

⇓

More user friendly and Convinient to the user

Client-Server Architecture

C: POSIX



Multi-Threading is very very important and (or) Concurrent programming plays crucial role in current requirements.

User & Kernel Level Threads: [Light-weight-processes]

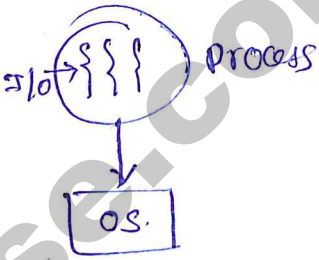
User Threads

→ OS is unaware of the threading

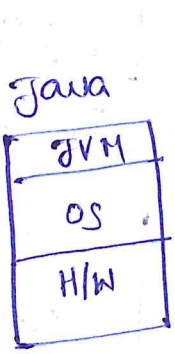
→ OS thinks Everything is a single process.

→ Thread-Management is Completely taken care by the user/programmer.

User mode



If one thread is blocked for I/O whole process will blocked (OS thinks that the process is blocked)



→ JVM will take care of all the threads in Java.

Kernel Level Threads: OS is multi-threaded.

→ Thread Scheduling is taken care by OS.

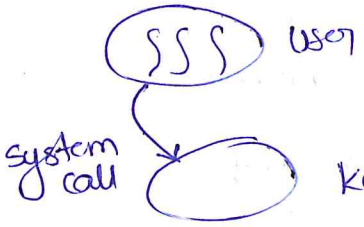
system calls → Kernel

- Kernel knows 3 threads are running.
- If one blocked for I/O OS schedule the remaining threads.

advantage

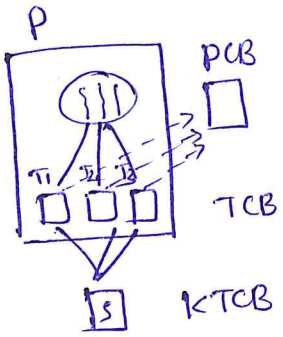
⇒ Efficiency-Increase

disadvantage:



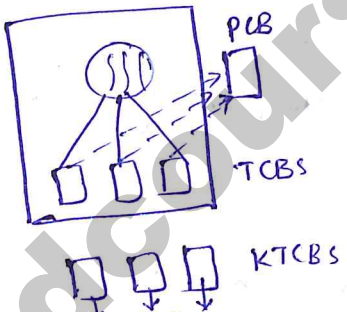
switching via kernel is more expensive

Hybrid-Threads : Linux (Most of the modern os use Hybrid Threads (many-many).



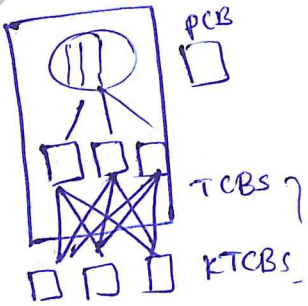
(a)

Many to one Mapping



(b)

one-one



(c)

Many-Many

More Solved problems-II

(1) which one of the following is FALSE?

TRUE → (A) user level threads are not scheduled by the kernel.

TRUE → (B) When a user level thread is blocked, all other threads of its process are blocked.

TRUE → (C) Context switching between user level threads is faster than context switching between kernel level threads.

FALSE (D) Kernel level threads cannot share the code segment.

(2) Threads of a process share

(A) Global variables but not heap.

(B) Heap but not global variables.

(C) Neither global variable nor heap.

(D) Both heap and global variables.

Heap → Dynamically Allocated Data

Global variables → Statically allocated Data

(3) Which of the following is/are shared by all the threads in a process?

X 1. program counter

X 2. Stack

✓ 3. Address space → Code, Data

X 4. Registers

(A) I and II only

(B) III only

(C) IV only

(D) III and IV only

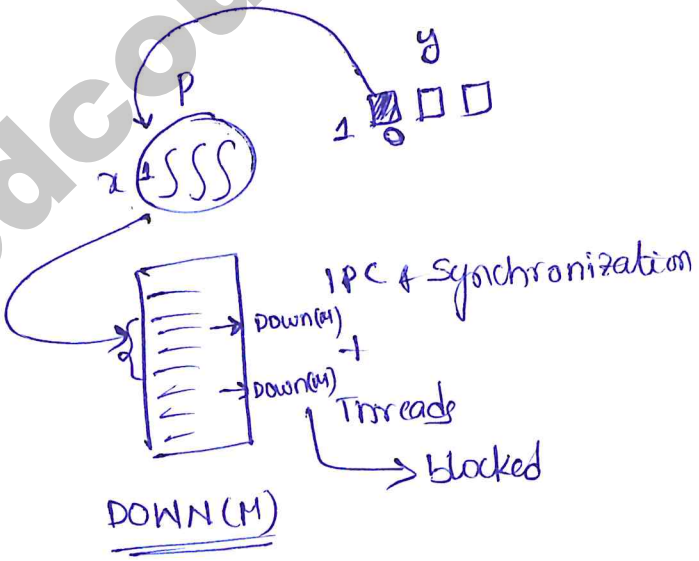
(4) A multi-threaded program P executes with (x) number of threads and uses (y) number of locks for ensuring mutual exclusion while operating on shared memory locations. All locks in the program are non-reentrant, i.e if a thread holds a lock, then it cannot re-acquire lock I without releasing it. If a thread is unable to acquire a lock, it blocks until the lock becomes available. The minimum value of x and the minimum value of y together for which execution of P can result in a deadlock are:

(A) $x=1, y=2$

(B) $x=2, y=1$

(C) $x=2, y=2$

(D) $x=1, y=1$



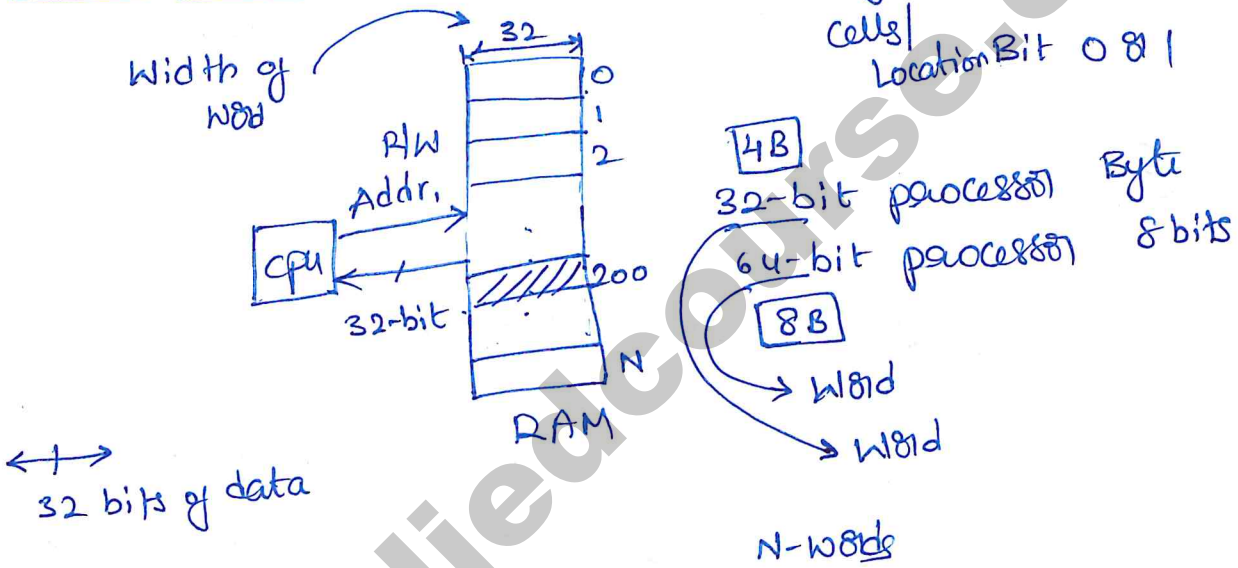
Memory Management

① Main-Memory - RAM - 4GB, 8GB

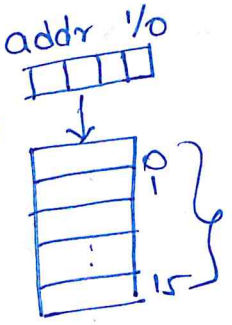
② DISK, Solid State Drive

{ 256GB
512GB
1TB }

Abstract View of Memory: bits, bytes, words, chips



Addressing:



RAM
Random Access Memory.

① N=16 = # cells/Loc/ n-bit address

4-bits ⇒ $n = \log_2 N$

② N=256 ⇔ $n = \log 256 \Rightarrow 8$ $\boxed{2^n = N}$

③ N=4GB ⇒ Assume 1 cell & 1 word = 1 Byte

$n = \log 2^{32} = 32$

$= 2^{32}$ W/ cells

$1K = 2^{10} \approx 1024$
$1M = 2^{20} \approx 1M$
$1G = 2^{30} \approx 1$ Billion
$1T = 2^{40}$

(4) $N = 500GB$

1Word = 1B ⇒ 8 bit processor

⇒ 500GB

⇒ 500×2^{30} cells/W $512 = 2^9$ $256 = 2^8$

$\lceil \text{Log } 500 \times 2^{30} \rceil$

$= 39$

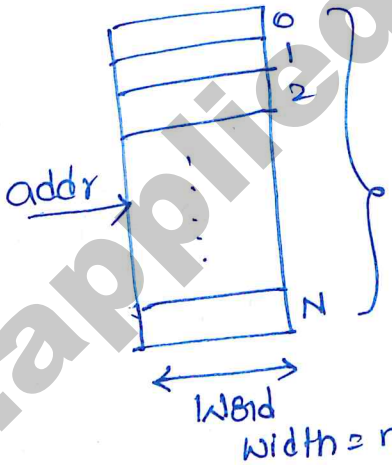
$2^9 \times 2^{30} > 500 \times 2^{30} > 2^8 \times 2^{30}$

$n = 39$ $n = 38$

(5) $n = 27$

$N = ?$

⇒ $2^{27} = 2^7 \times 2^{20} = 128MW$



Width = m bits

N = # Cells

n = address-size

m = word-width

$n = \lceil \text{Log}_2^N \rceil$

Total Memory = N × m

(6) n = 23 bits

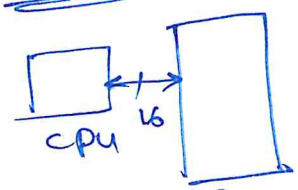
word-width = m = 16 bits. = 2B

words = 2^{23} words

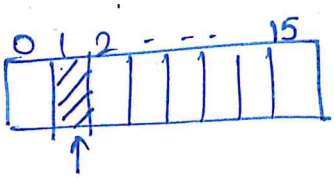
bytes = $2^{23}W = 2^{23} \times 2B = 2^{24}B$

bits = $2^{23} \times 2 \times 8 = 2^{27}$ bits

Addressability

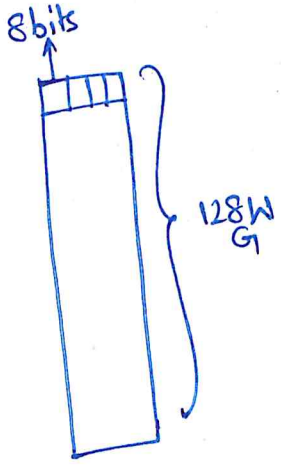


↳ programming.



(Q) N = 128GW

m = 32 bits. = 4B



RAM capacity	Addressability
Word : 128G	$n = 2^7 \times 2^{30} = 37$ bits. (n)
Byte : 512GB	$n = 2^9 \times 2^{30} = 39$ bits (n)
bits : 4Tb	$n = 2^2 \times 2^{40} = 42$ bits (n)

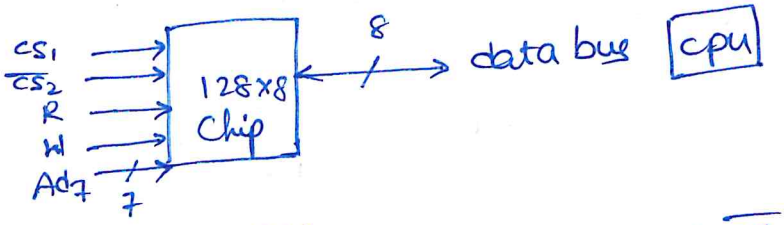
(Q) Mem = 64Gb

m = 16 bits

RAM capacity	Addressability
Word: 4GW	$2^2 \cdot 2^{30} = 32 \Rightarrow n = 32$
Byte: 8GB	$2^3 \cdot 2^{30} = 33 \Rightarrow n = 33$
Bit : 64Gb	$2^6 \cdot 2^{30} = 36 \Rightarrow n = 36$

RAM chips & Organization:

CS1 - control signals



{ 5V → 1
{ 0V = GND → 0

N = 128

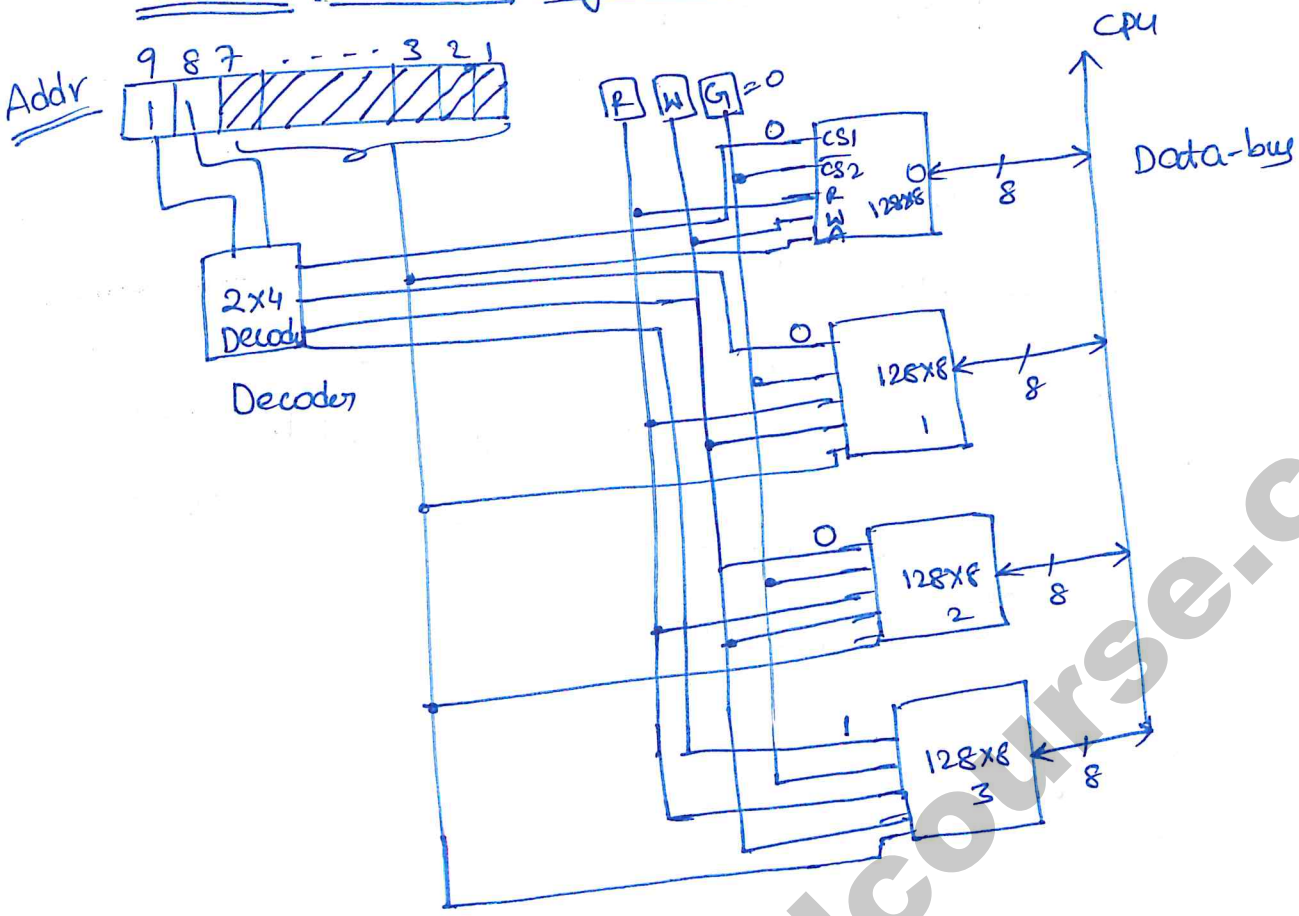
m = 8

n = 7

$CS_2 = 0 \Rightarrow \overline{CS_2} = 1$ ⇓ RAM

$CS_2 = 1 \Rightarrow \overline{CS_2} = 0 \Rightarrow$ ROM

Build 512B RAM using 128B chips:



$$\frac{512B}{128B} = 4$$

$N = 512W \Rightarrow \boxed{n=9}$

00 — Select First RAM chip
01 — " Second RAM
10 — " Third RAM
11 — " Fourth RAM chip

Q) 1GB RAM using 128MB chips

$$\#\text{ chips} = \frac{1GB}{128MB} = \frac{2^{30}}{2^7 2^{20}} = 2^3 = 8$$

$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$

Vertical - Increasing: Constant word-size

16 MB RAM using 128 x 8 chips

1W = 8 bits = 1B

16 MB → 16 MW

128 words x 8 bit / word

$$\frac{16\text{MW}}{128\text{W}} = \#\text{ chips}$$

N = 16 MW

$= 2^4 \cdot 2^{20}$

$= 2^{24}$

$\Rightarrow n = 24$

$= \frac{2^4 \times 2^{20}}{2^7} \times W$

$= 2^{17}$ chips

$= 128\text{K}$

decoder : $17 \times 2^{17} \Rightarrow x \times 2^x$

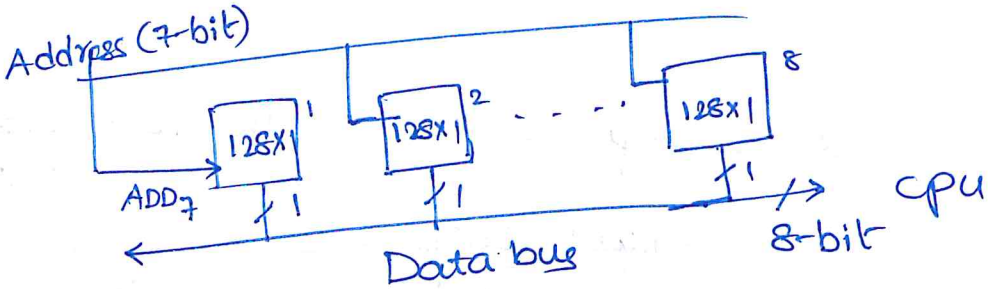
1W = 8 bits = 8B

Address: 17 7 = 24

17 → Decoder, 7 → chips

Horizontal Increasing: change word size

128B RAM using 128x1 RAM



$\frac{128 \times 8}{128 \text{ b}} = 8$

128W ⇑ 128B 1W = 1B = 8 bits.

128×1 1W = 1b

128B RAM
1B = 1W
⇓
128W RAM
7 bit address.

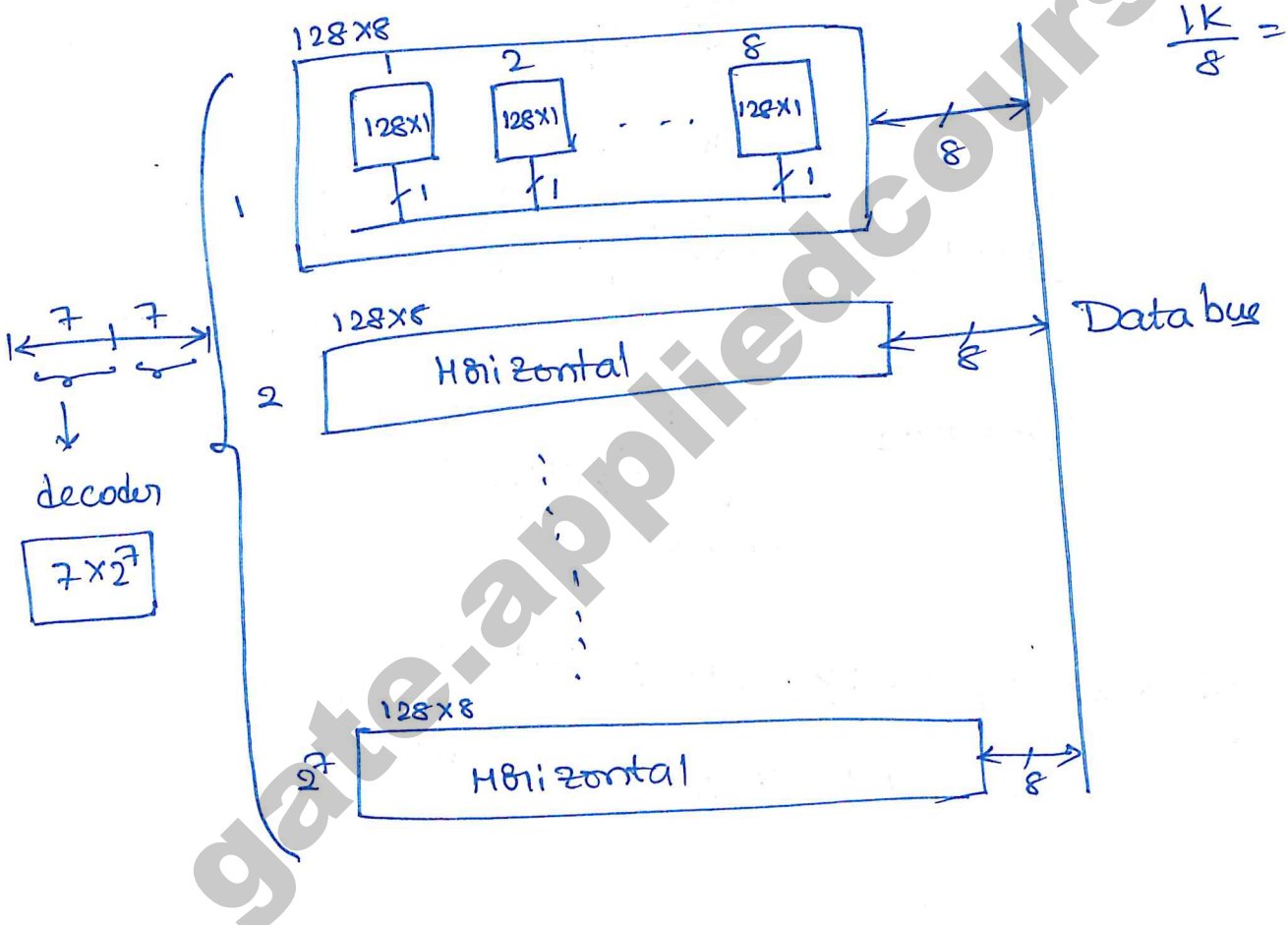
Vertical + Horizontal Increasing:

16KB RAM using 128×1 chips

1W = 1B = 8 bits.

$$\#\text{chips} = \frac{16\,KB}{128\,b} = \frac{2^4 \times 2^{10} \times 2^3}{2^7} = 2^{10} = 1K \text{ chips}$$

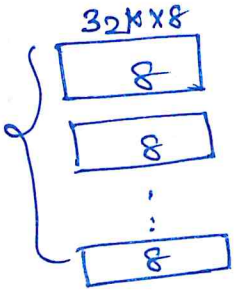
$$\frac{1K}{8} = 2^7$$



(2) 32K×1 chips, 256 kB 1W = 1B = 8 bits.

chips: $\frac{256kB}{32K} = 2^6 = 64$

Decoder: 3×8

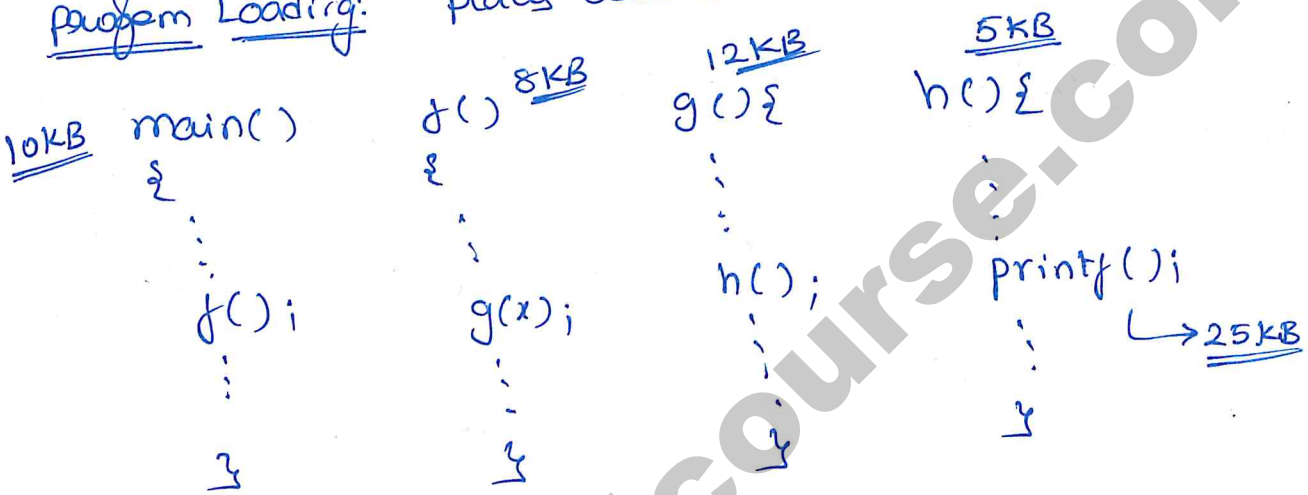


$32K \times 1 \xrightarrow{8} 32K \times 8$

Address: 3 (Decoder) + 15 (32K) ⇒ 18 bits

Program Linking and Loading:

Program Loading: places code into RAM



{ header: func declaration
Library: Func definition

program → Compiler → MLL → Linker/Loader (O.S) → process

60KB

Static & Dynamic Loading:

⇓

{ All modules of code are loaded into main memory before runtime.

Disadvantage → Bad memory utilization. ⇓ Lower Multi-processing

Advantage

Speed Execution of

Dynamic loading:

→ Only load modules of code that are currently needed.

→ 10KB

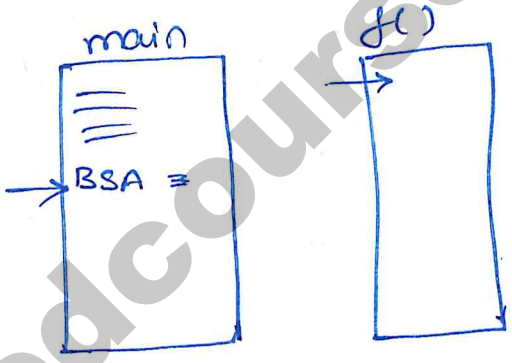
→ disadvantage: Speed of Execution is very low.

→ advantage: Efficient memory utilization.

Linking: Static vs Dynamic

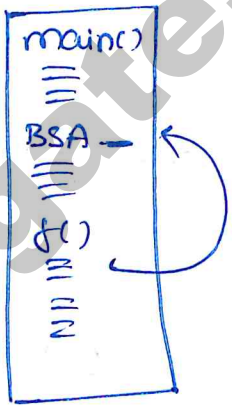
```
main()          +    f()
{                    {
  ≡                    ≡
  f(); BSA             ≡
  ≡                    ≡
}                    }
```

Branch and Save return Address



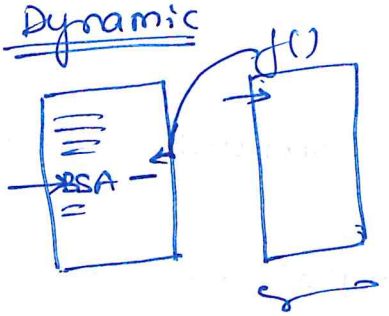
Binding the addresses of called functions in the calling functions MLL code.

Static Linking



Static loading

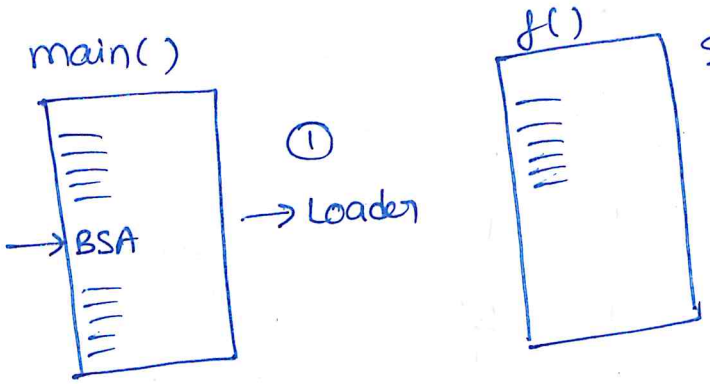
Dynamic



[Before Runtime]

.dll (dynamically linked libraries)

Dynamic Linking + Dynamic Loading



Adv
optimal memory utilization

disadv:
Extra time

Programming Flexibility

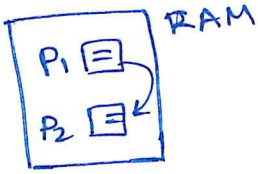
.dll
Library → func. definitions

Memory Management : Objectives & functions

4GB | 8GB | 16GB

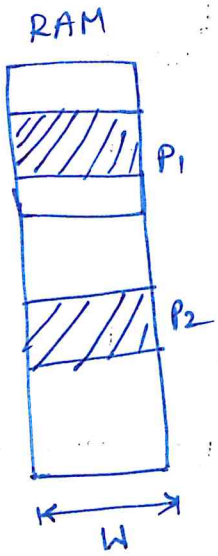
1. Minimize wastage of Memory (8) Better utilization of main memory.
2. Increase degree of multi-tasking
   ⇓
   Better cpu utilization
   ↑
   more processes in memory

1. Memory allocation
2. Memory deallocation
3. Managing the free Space.
4. Protection



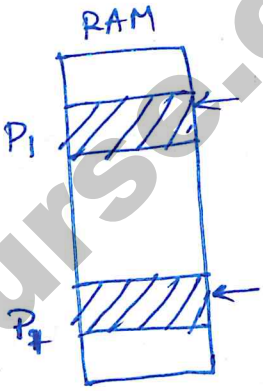
Contiguous

1. Overlays
2. Partitioning
3. Buddy System

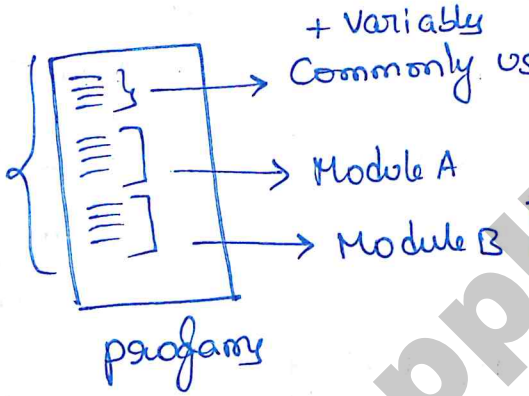


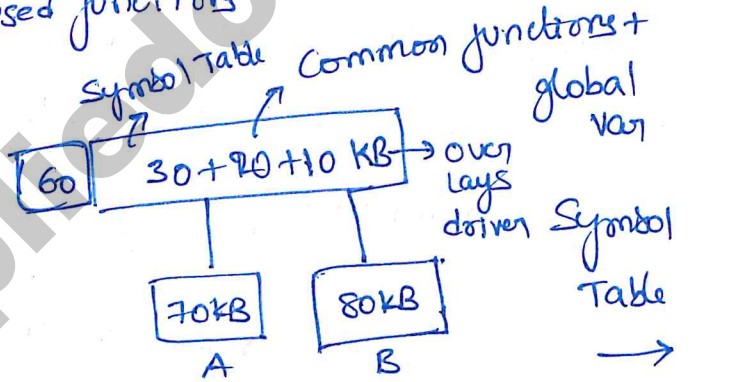
Non-Contiguous

1. Paging
2. Segmentation
3. Segmented paging
4. Virtual memory & Demand paging.



Overlays: (Contiguous)





Necessary Condition:

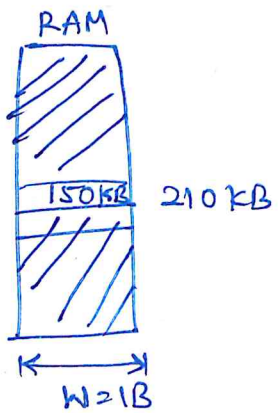
Modules A & B are Indep

Let free space = 150KB

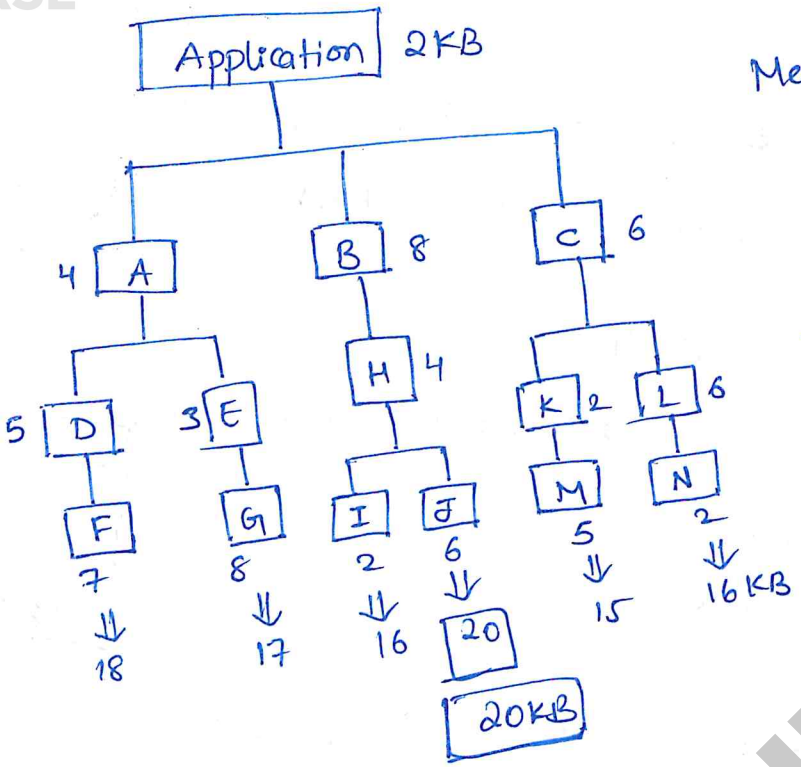
overlays free

150 < 60 / B: 80KB

150 + 60 = 210KB



Overlay Tree:



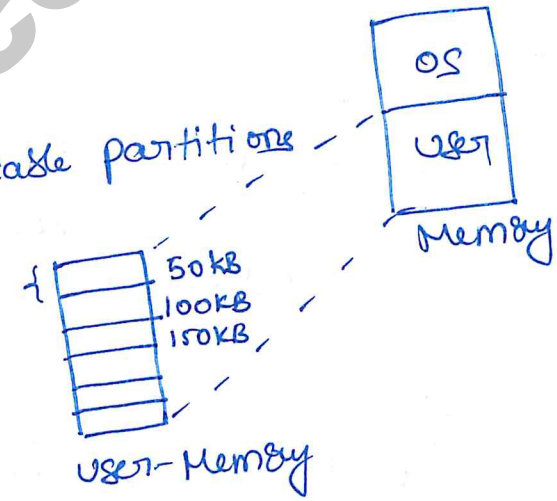
Min Memory-needed = ?

2
18
20
30
70 KB

⇓ 20 KB

Contiguous partitioning:

- Fixed partitions
- variable partitions



partitions can be same-size or variable size different

P_1 : 40KB

P_2 : 270KB

MFT → Multiprogramming with Fixed number of Tasks.

MVT → Multiprogramming with variable number of Tasks.

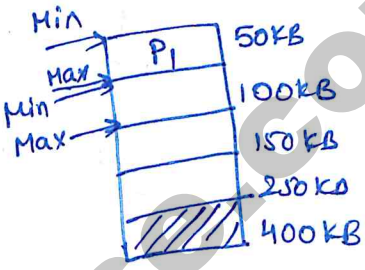
Fixed-partitions:

→ max # processes @ any time = # partitions.

→ (Q) Which available partition should we place a new process-in.

→ Protection: ∀ partition

↳ Limit Registers



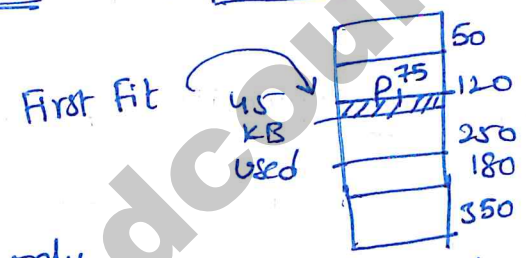
P_1 : 40KB

P_2 : 270KB

Partition Allocation Methods:

P_1 : 75 KB

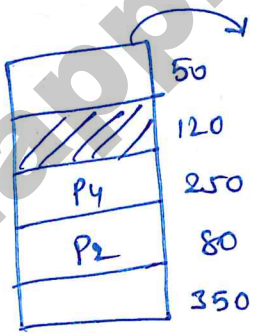
→ First Fit:



→ Best-Fit: ← Best Fit only

Good Method

Best available Fit



Wastage is minimized

Best-Fit

P_1 : 75

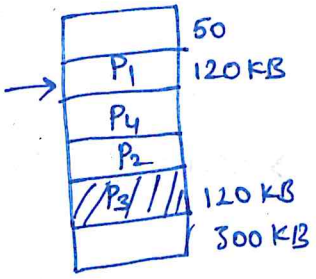
P_1 will wait for P_2 to free the memory location.

Best-available Fit: place it in 120 KB partition.

Next-Fit:

↓

Faster

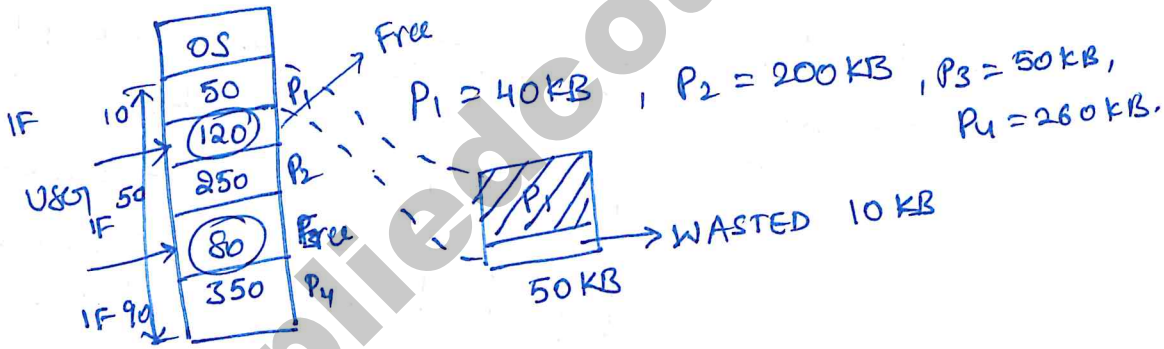


P_3 : 40KB place next to the most recent partition.

Worst-Fit: → Wastage is Maximal. (waste)

Performance of MFT: (Fixed partitions)

① **Fragmentation** < Internal / External



$P_1 = 40KB$, $P_2 = 200KB$, $P_3 = 50KB$, $P_4 = 260KB$.

⇒ $P_5 = 200KB$

Free Space = 120 + 80 = 200

(Non-Internal Fragmentation)

External Fragmentation

⇓

Inability to load a process into memory, as the free space in the memory is Non-contiguous.

② **Degree of Multi-programming:**

⇓

Fixed | Limited by # partitions.

(3) Process-size Limitations:

$P_6 = 500\,KB \rightarrow$ cannot accomdate

Partitioning: Variable-Sized (MUT): Multiprogramming with Variable number of Tasks.

↳ No internal Fragmentation

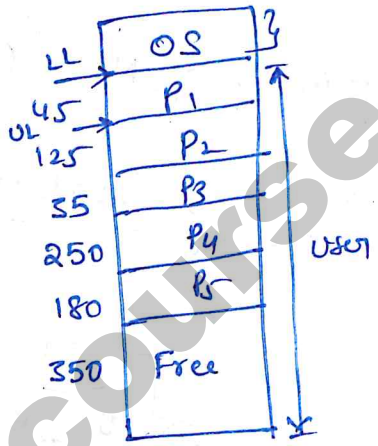
Protection — PCM:

Partition Control Memory.

PID	LL	UL	size
P_1	..	.	..

LL: Lower limit

UL: Upper Limit



P_1: 45KB P_4: 250KB

P_2: 125KB P_5: 180KB

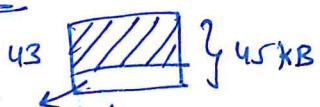
P_3: 35KB

Partition Allocation

1. First Fit — $P_6 \rightarrow 45KB$ < 43, 2

Large enough freeblock to accomdate

2. Best Fit

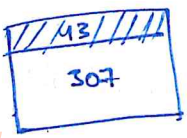


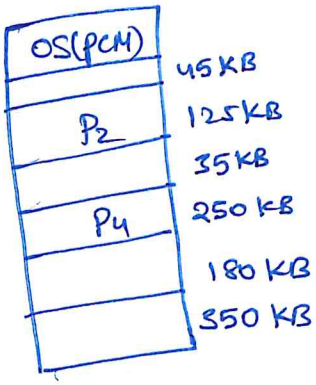
External Fragmentation

3. Worst Fit — 350KB < 43KB, 307KB

⇓

Good Strategy For Variable partitioning.





P_1, P_3 & P_5 are Completed.

P_6: 45KB

Performance of variable partitioning:

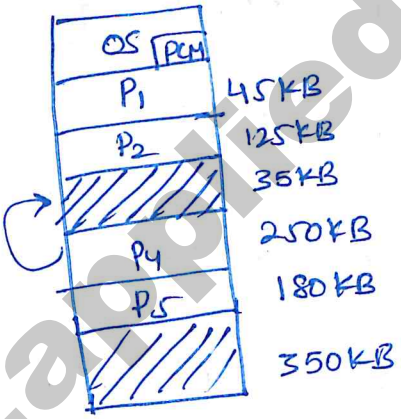
1. ~~First Fit~~: Fragmentation: → No Internal Fragmentation.
   → External Fragmentation.

2. Degree of Multi-programming: Increases
   overhead: OS (Book-keeping)

3. # processes: Not limited.

Solution to External Fragmentation:

Compacting / coalescing:

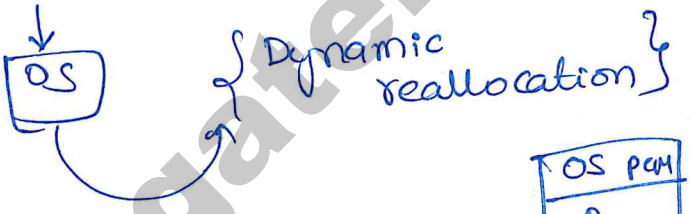


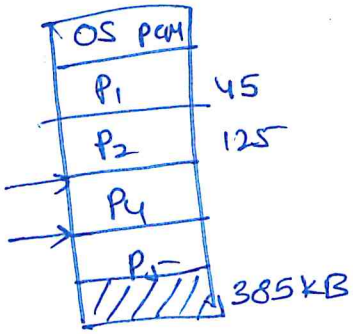
385KB

P1, P2, P4 & P5 are running

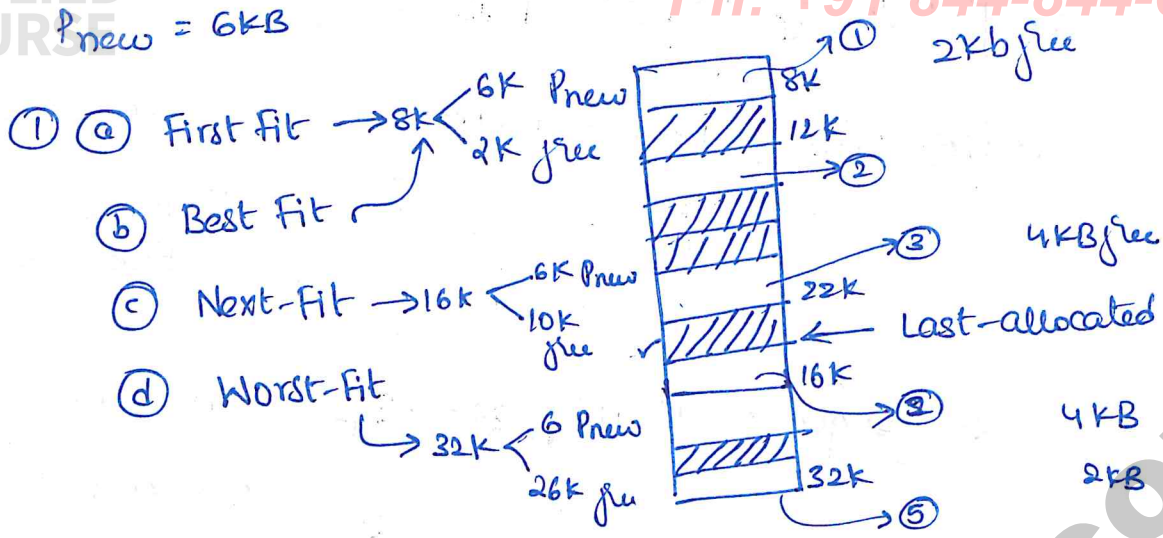
P3: Completed

P6: 380KB





(Q) P_{new} = 6KB



② Howmany new 6KB processes be allocated using MVT?

⇓ Variable Size

1+2+3+2+5

= 13 [6KB] processes

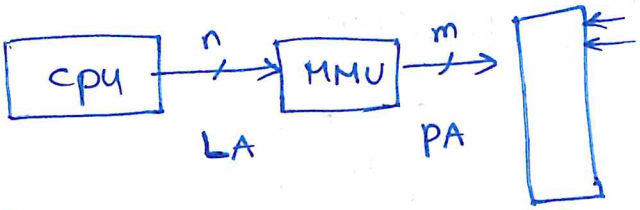
MVT + Compacting

13 + ②

Non-Contiguous Memory Allocation:

set of words/Locations

Logical Address Space (LAS) & physical Address Space (PAS)



P_i

LAS ⇒ 2^{20} words → 20 bit address → n → Logical Address

PAS ⇒ 2^{16} words → 16 bit addr → m → physical Address

$n \geq m$

LAS ≥ PAS

Eg: LAS = 16KB ⇒ 2^{14} B ⇒ LA: 14 bits ⇒ 1W = 1B

PAS = 4KB ⇒ 2^{12} B PA: 12 bits

Non-contiguous memory management → Organization of LAS

2. Organization of PAS

3. MMU

4. Translation.

- ✓ Simple-paging
- ✓ Segmentation

Cases:-

① LAS > PAS → Very good ⇒ Degree of Multiprocessing

Efficient use of Mem

② PAS = LAS → No improvement

③ LAS < PAS → worse.

Simple paging | Paging:

① Organization of LAS: → View of the process

LAS: 8KB ⇒ LA: 13b

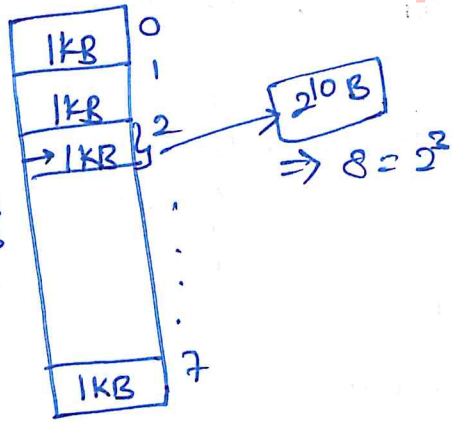
PAS: 4KB ⇒ PA: 12b

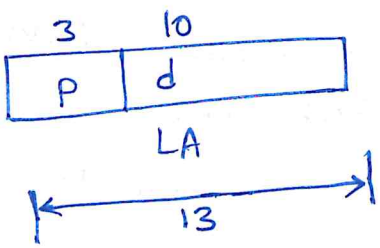
→ Pages: Equi-sized

pages : 2^x

$N = \frac{LAS}{PS} = \frac{8KB}{1KB} = 8$

page size : 1KB





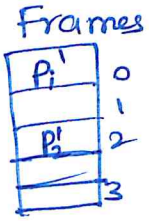
010 $\Rightarrow 2^{nd}$ page

p: page number

d: page offset

② Org of PAS:

4KB $\Rightarrow$ PA: 12 bits

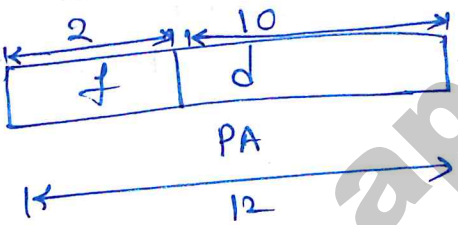


→ Frames | page-frames

→ Frame size = page-size

FS = 1KB

Frames = M = $\frac{4KB}{1KB} = 4$



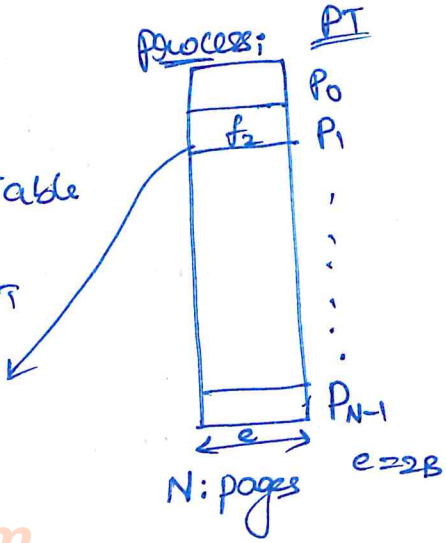
F: Frame Number

d: Frame offset

③ MMU-organization:

→ Page Table | page Map Table

→ Every process has a PT

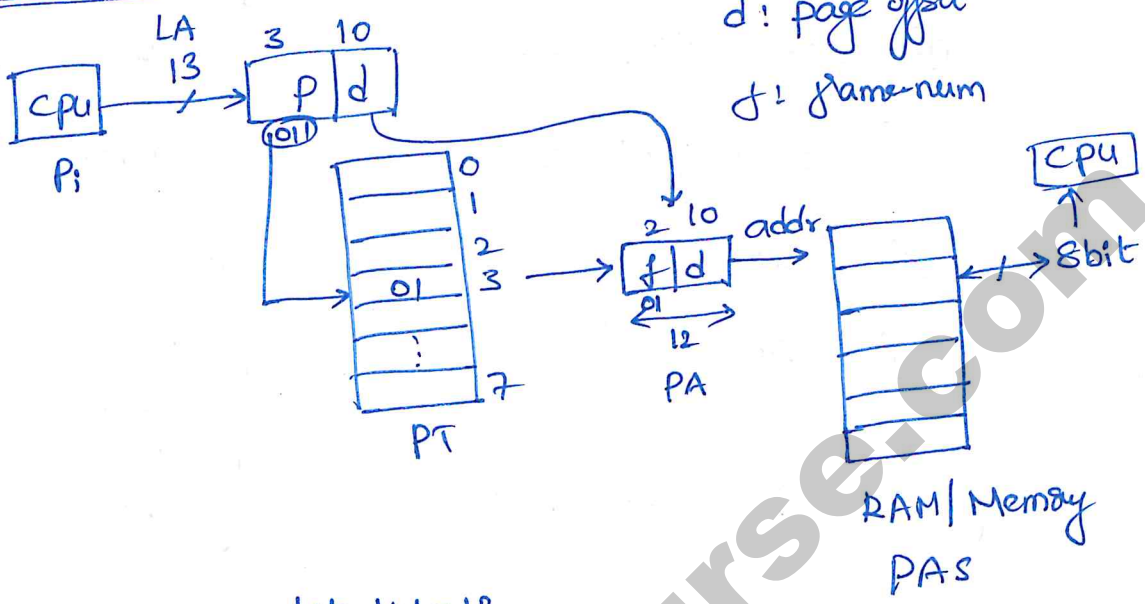


00000000 00000010

Page-Table size = NXe

④ Translation:

p: page num
d: page offset
f: frame-num



Ex :- ①

Let 1W = 1B

LA = 29 bits ⇒ LAS = 2^{29} B

PA = 21 bits ⇒ PAS = 2^{21} B

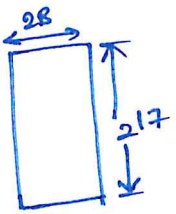
PS = 4KB ⇒ 2^{12} B

ⓐ N = # pages = $2^{29}/2^{12}$ = 2^{17} pages

ⓑ LA | p | d | p: 17, d: 12

ⓒ PA | p | d | p: 9, d: 12 (9, 12)

ⓓ e = 2B ; PT-size



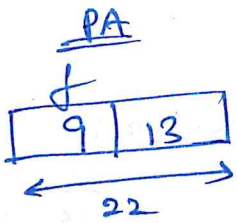
2^{17} x 2B

= 2^{18} B

= 256KB

Q2 2K pages & 512 Frames

PA: 22 bits



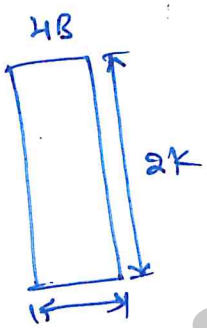
FrameSize = 2^{13} = ps

(a) e = 4B ; PT-Size

(b) page-offset (d)

(c) LAS

(d) PAS

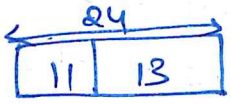


1B ≠ 1W

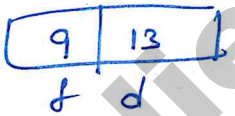
(a) PT-Size = 2K × 4B = 8KB

(b) d = 13

(c) LAS:



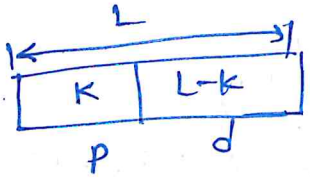
(d) PAS:-



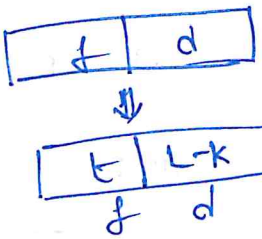
Q3 LA = L bits # frames = 2^t

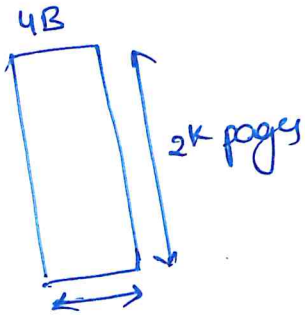
page-bits = K

e = 4B



1B = 1W





(a) PAS: 2^{t+L-K} B

(b) PT-Size: $2^K \times 4B = 2^{K+2} B$

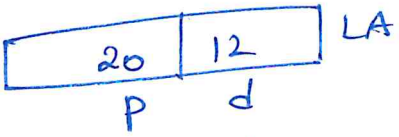
(Q4) LA = 32bit

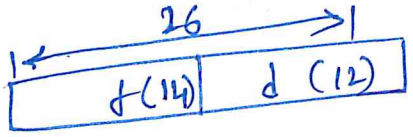
PS = 4KB (pagesize)

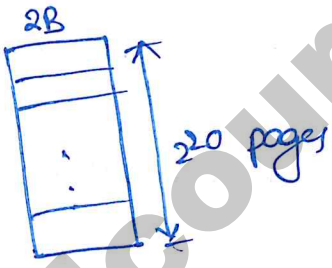
PAS = 64MB $\Rightarrow 2^6 \times 2^{20} \Rightarrow 2^{26}$

e = 2B

PT-Size = ?





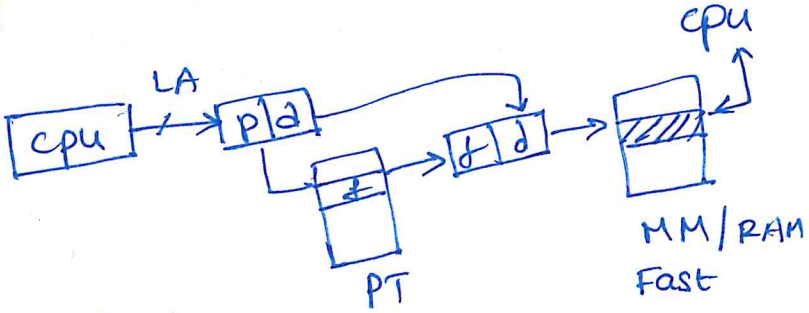


$\Rightarrow 2 \times 2^{20} B$

$\Rightarrow$ 2MB pagetable size

Performance of Simple-paging:

Effective Memory Access time [EMAT] = m + m = 2m (PT, Word)



Memory-Access time = 'm' ns | ns

(Memory | RAM)

PA

Disk

slow

How to reduce the Effective memory Access time?

Solution: Paging + TLB (Translation lookaside Buffer)

cache

V. fast to V.V. fast

Amount

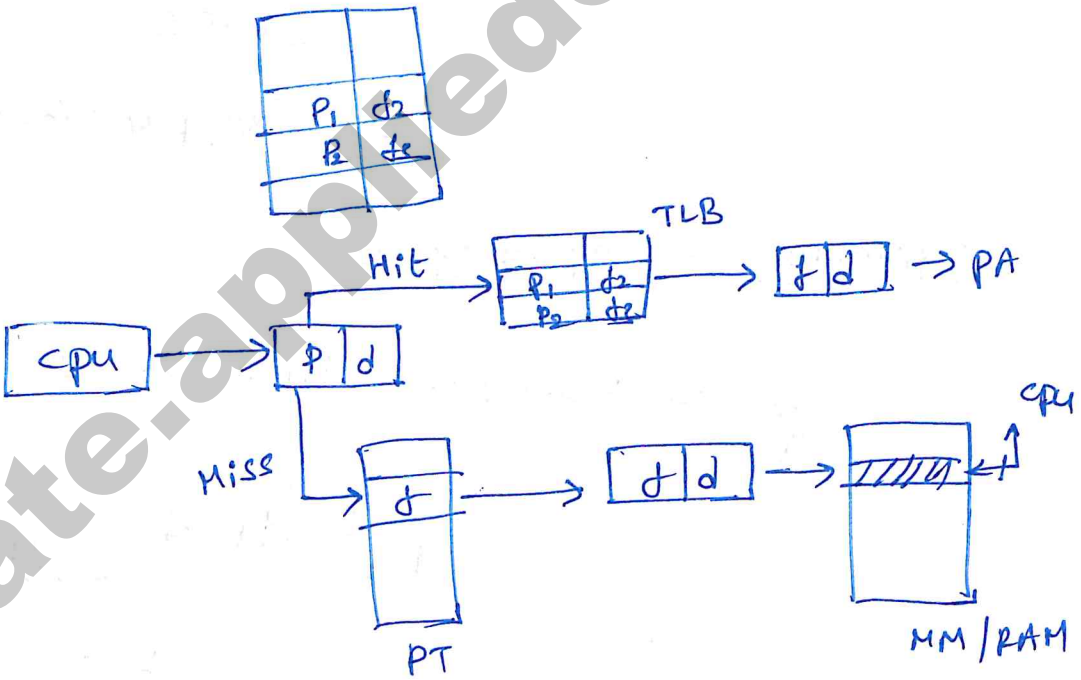
Reg < L_1 < L_2 < RAM < DISK

L_1 cache L_2 cache

Speed:

Reg ≥ L_1 ≥ L_2 > L_3 > RAM > disk

Cache Access time = [c ns << m ns]



(a) $m = 100ns$

$c = 20ns$

$x =$ TLB-Hit rate $= 90\%$ Simple paging.

(a) $EMAT_{sp} = 2 \times m = 2 \times 100 = 200ns$

(b) $EMAT_{sp+TLB}$ => $x = 0.9$ TLB Hit rate

TLB Miss rate $= 0.1$

$= x(m+c) + (1-x)(2m+c)$

$= 0.9(120) + 0.1(220)$

$\Rightarrow$

(a) What TLB-hit-ratio is required to reduce EMAT from 280ns (without TLB) to 180ns (with TLB) assuming cache-access-time of 40 ns.

$c = 40ns$

$EMAT_{sp} = 280ns = 2 \times m \Rightarrow \boxed{m = 140ns}$

$EMAT_{sp+TLB} = 180 = x(180) + (1-x)(40+280)$

$x = 1 \Rightarrow \boxed{100\%}$ TLB Hit

Multi-Level paging:

Largept ← performance Issues of Sp.

LA = 32bit

ps = 4KB

e = 4B

pages = $2^{32}/4K = 2^{32}/2^{12} = 2^{20} \cong 1M$

PT-Size = $2^{20} \times 4B = 4MB$

10 processes: 40MB

let physical Mem: 64MB

64 − 40 = 24 only left for remaining processes

Solution 1: Increase page-size

⇒ Reduction in # pages.

⇓

PT-Size reduces.

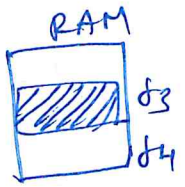
⇓

{Internal Fragmentation

program-size = 1026 B

Case (a) ps = 1KB = 1024B

case (b) ps = 2B



P_1 : 1024

P_2 : 2 → 1022 B Wasted

$= \frac{1026}{2}$

= 513 pages. No Internal Fragmentation.

⇓

Page Table-Size Increases.

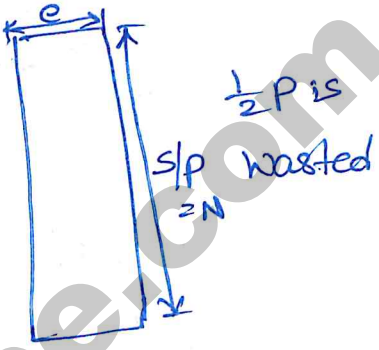
(a) calculate optimal page size assuming half a page is wasted per process on Average.

optimal → minimizing Internal Fragmentation

LAS = S Bytes

PT_entry_size = e B

pageSize PS = P Bytes.



Min Frag = P/2 ⇒ min P

$$\min(\text{Frag} + P.\text{TSize}) = \frac{P}{2} + \frac{S}{P} * e = f(P)$$

$$\min(df) = \frac{df}{dp} = \frac{1}{2} - \frac{Se}{P^2} = 0$$

$$P = \sqrt{2Se}$$

Solution 2: Multi-Level paging (Multi level Indexing)

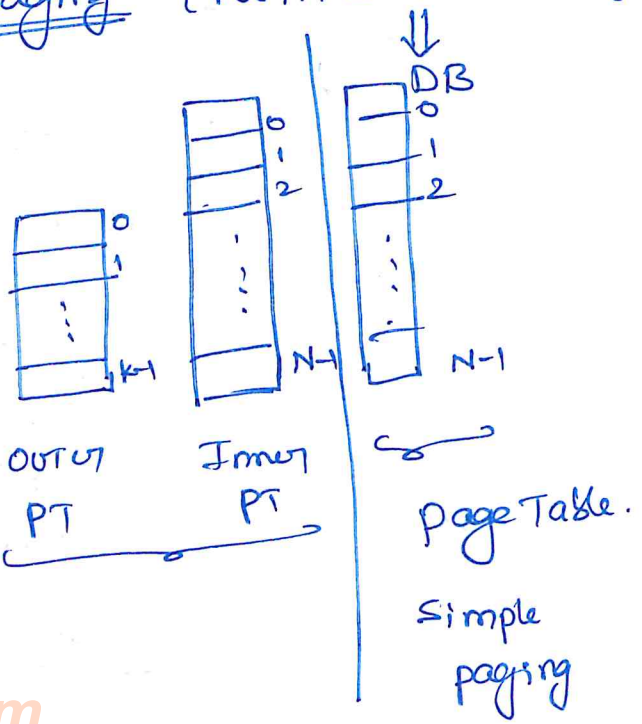
LAS = 4GB

PS = 4KB

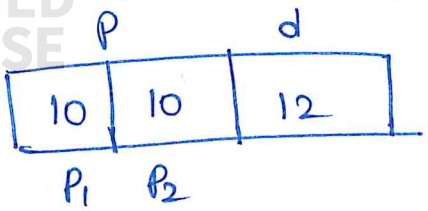
LA:

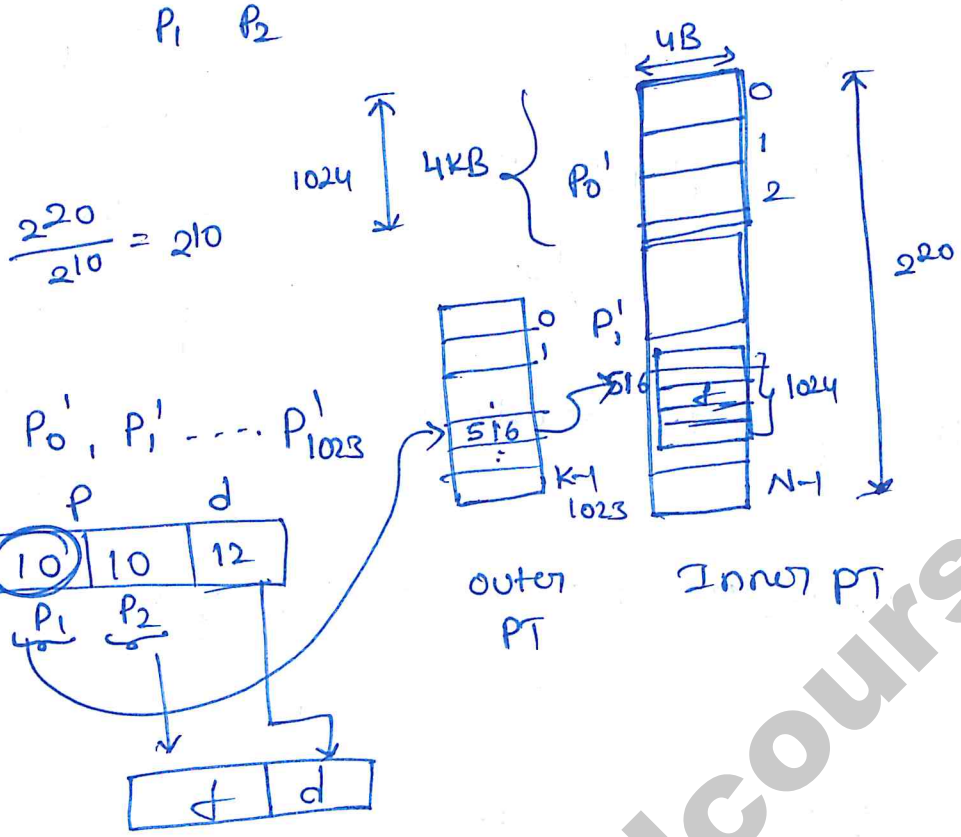
$$\#\text{pages} = N = \frac{2^{32}}{2^{12}} = 2^{20}$$

p	d
20	12

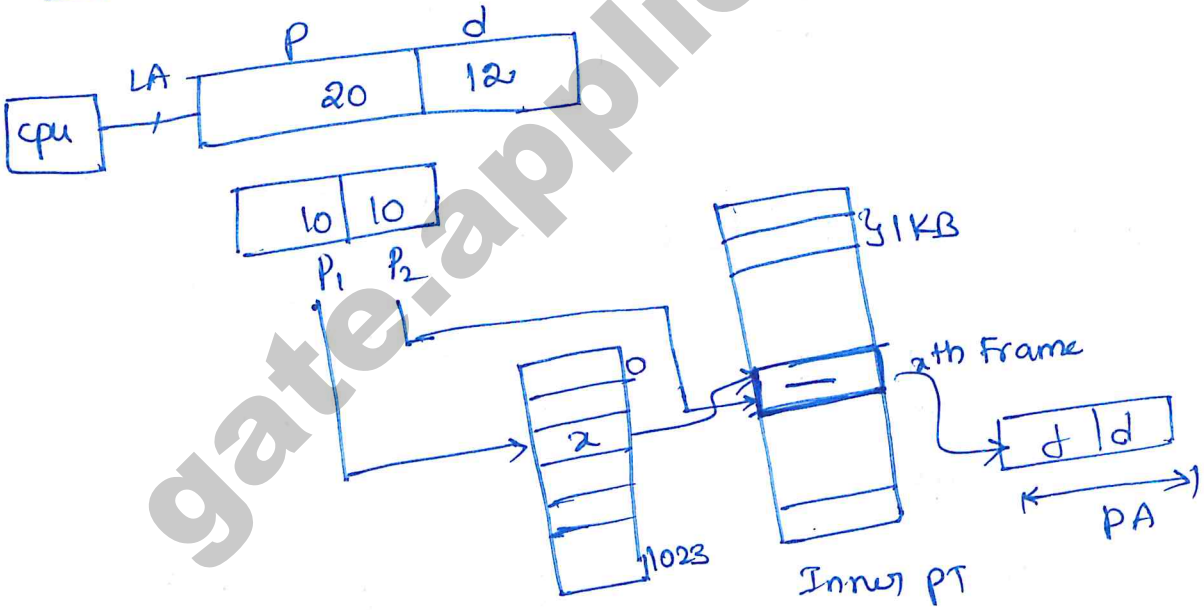


Two-Level paging



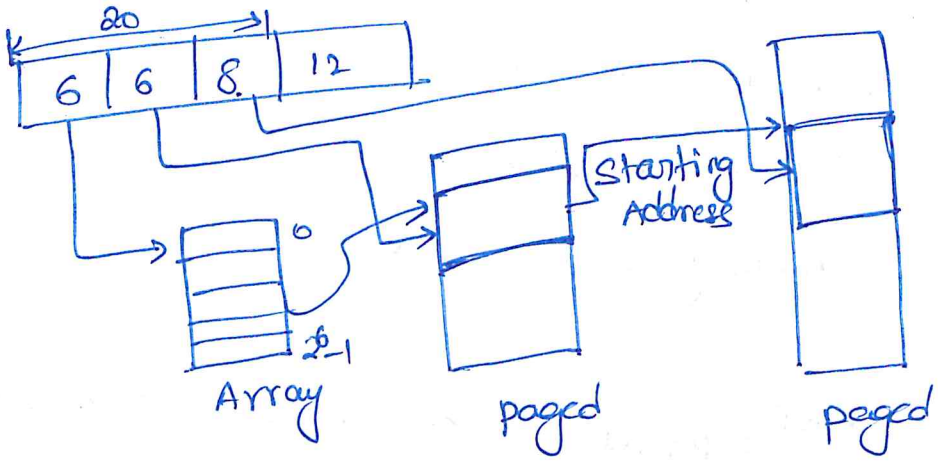


Block Diagram



2-level paging

3-Level paging

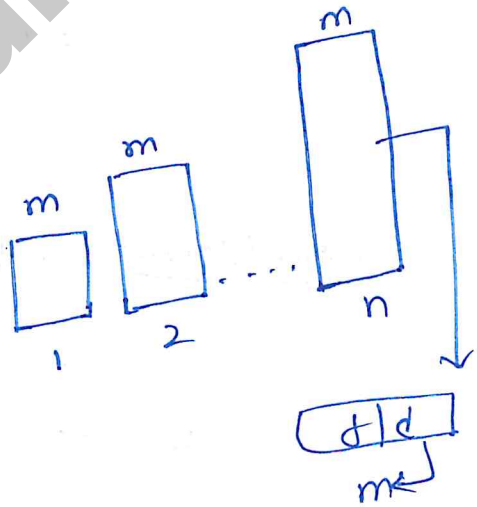


Intel 5-level-pages

Performance of Multi-level paging:

$EMAT_{2LP} = 3 * m$ ns

$EMAT_{nLP} = (n+1) * m$ ns

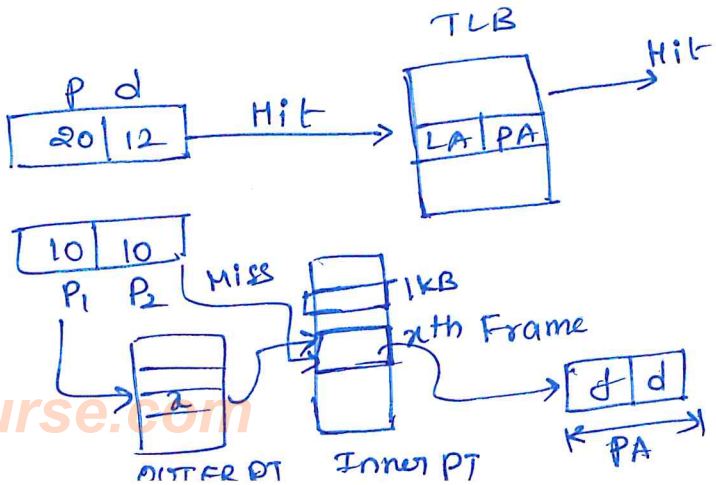


5-level paging ⇒ (6) m ns

Multi-level paging - TLB & Hash-based paging

MLP + TLB:

2LP + TLB



(c) MLP + TLB

TLB-hit-rate = x

TLB-access-time = c

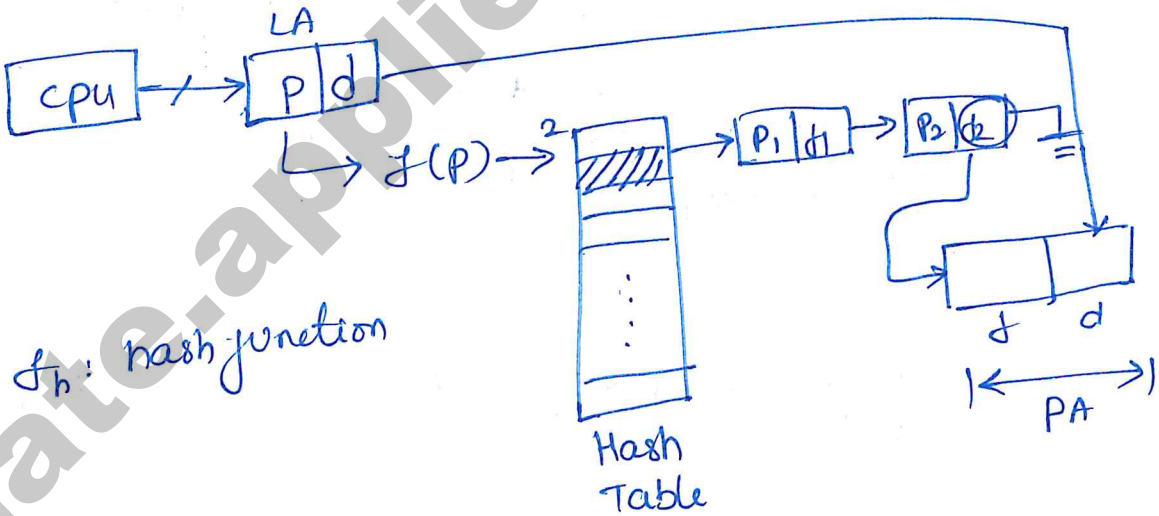
memory access-time = m

$EMAT_{2LP+TLB} = x(c+m) + (1-x)(c+m+m+m)$

$= x(c+m) + (1-x)(c+3m)$

$EMAT_{nLP+TLB} = x(c+m) + (1-x)(c+(n+1)m)$

Solution3: Paging with Hashing → Data structure



Segmentation:

Fixed & Variable size

User's view of memory allocation is not preserved with paging.

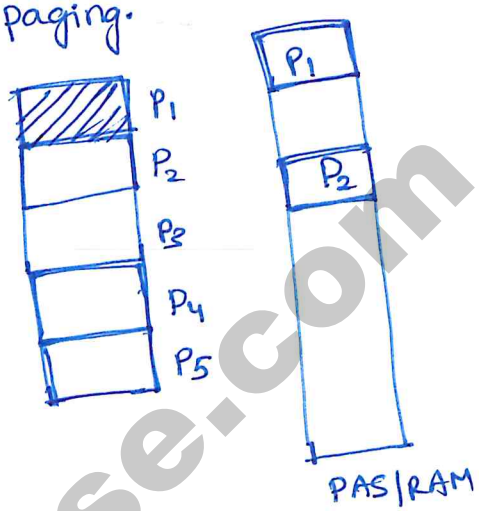
Limitations of paging:

Eg: PS = 1KB

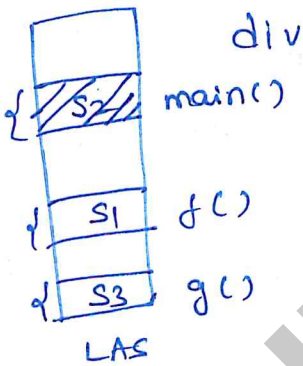
```
main()
{
  f()
}
```

segment — 5KB

```
f()
{
}
```



Organization of LAS:



divided in unequal sized segments.

page: LAS: Equi sized

Segment: LAS: unequisized

Frame: PAS: equi-sized

Physical Address space organization:

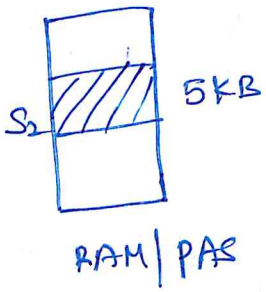
→ Variable-sized partitioning / MUT → Partition Allocation Methods

→ Segment is loaded into contiguous memory.

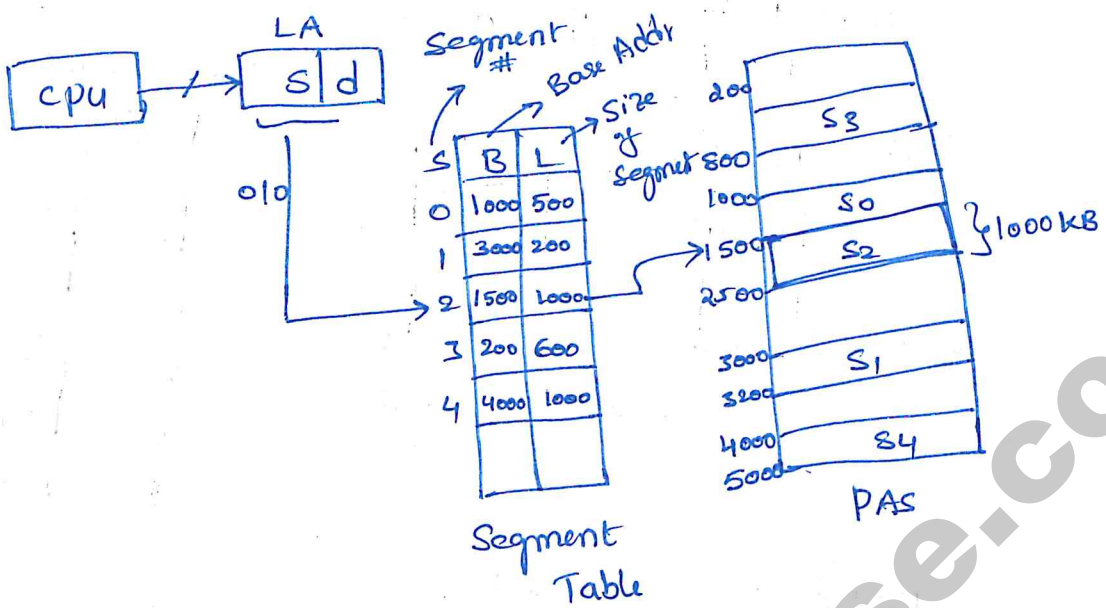
→ No Frames

⇓

Best Fit
Worst Fit
[illegible]
First Fit



Segmentation: MMU-org + Translation:



Implementation: paging on Segments / Segmented paging

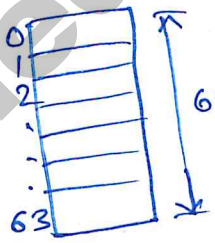
page-size (PS) = 1KB

Segment : 64KB

pages in a Segment:

$= \frac{64KB}{1KB}$

$= 64$



PA

1500 + d

let d = 128

1500 + 128

1628

PA

1500 + d

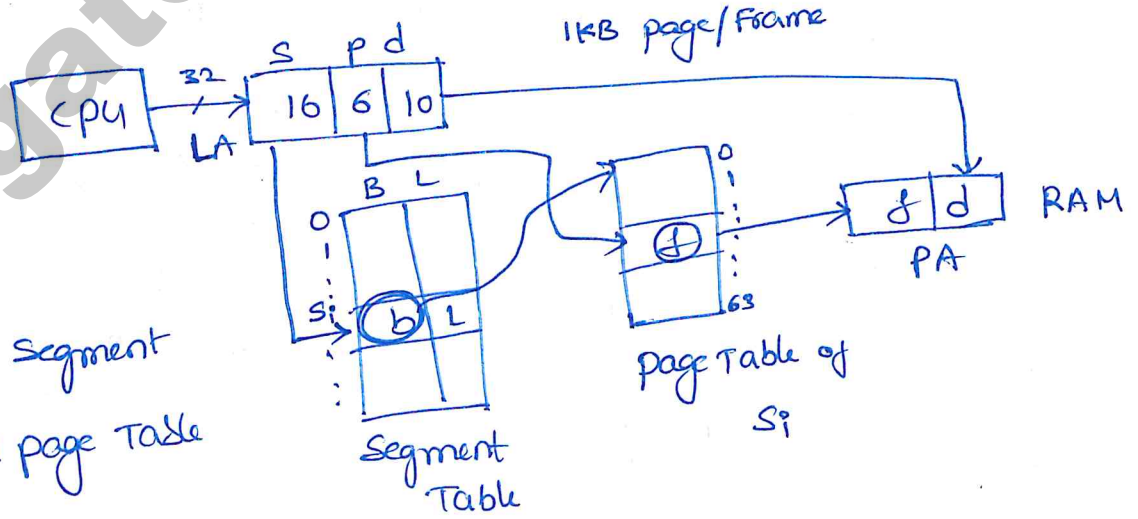
let d = 1280

2780

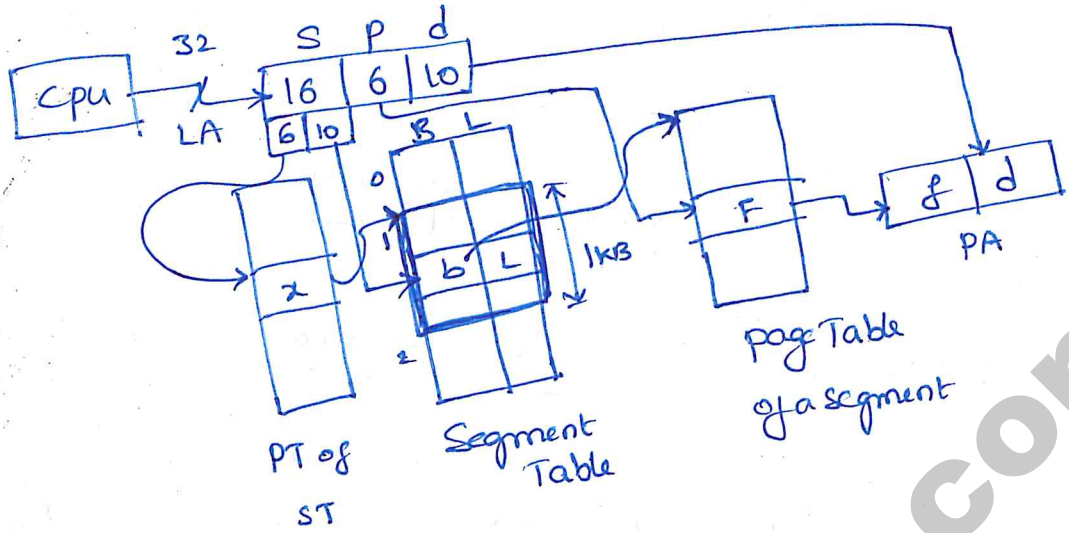
PA

error id if d > L

1000



Paging the Segment Table:



(a) LAS = PAS = 2^{16} B

LAS: 8 equi-sized Segments.

paging on each Segment

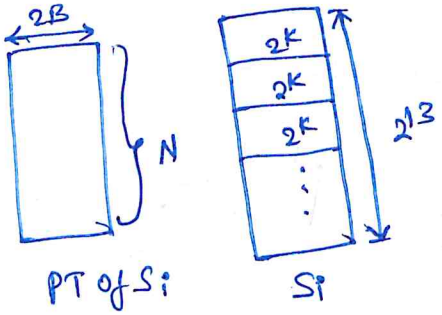
PT-size-entry-size = 2B

page size = 2^K

what should be the page size of a Segment so that PT of Segment exactly Fits in one-page.

Sol:

Size of Segment = 2^{13} B



$\Rightarrow \frac{2^{13}}{2^K}$

$\Rightarrow 2^{13-K} = N$

pages in a Segment $= 2^{13-K}$

size of page table = N X 2

$= 2^{13-K} \times 2 = 2^{14-K}$

$\Rightarrow 2^{14-K} = 2^K$

$\Rightarrow 14-K = K$

$\Rightarrow K+K = 14 \Rightarrow \boxed{K=7}$

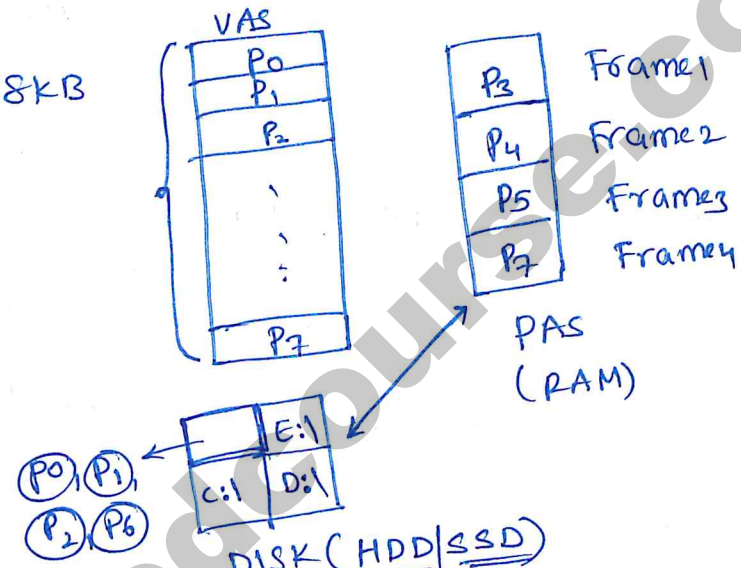
Virtual Memory & Demand paging

LAS = VAS = 8KB

PAS = 4KB

PS = 1KB

pages =



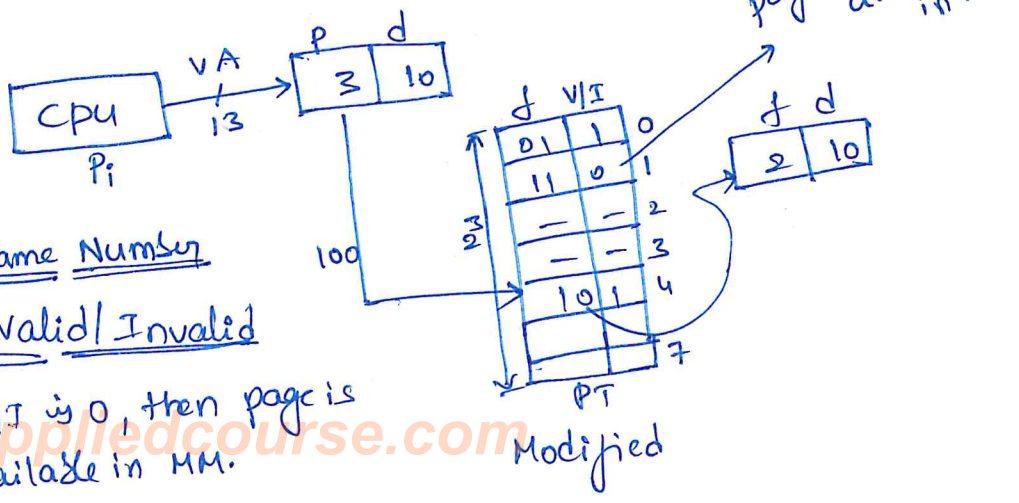
Swap Space → Linux, Unix, Android, Mac

Mac Box

- Physical Memory: 16.00GB
- Memory Used: 12.00GB
- Cached Files: 2.49GB
- Swap used: 1.04GB

PS = 1KB $\Rightarrow 2^{10}$ B

Assume 1W = 1B



F: Frame Number

V/I: Valid/Invalid

If V/I is 0, then page is not available in MM.

Modified

Page-fault: Required page is not available in MM.

page-fault-service:

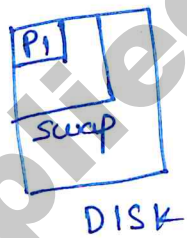
1. Current process is blocked.
2. Kernel-Mode (Mode-bit)
3. Virtual Memory Management System (OS) → DISK → RAM

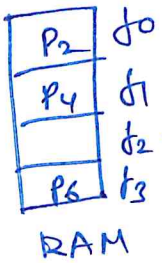
Disk Access ↓ ms

ns ns ns/μs ms
Reg ≥ L1 ≥ L2 ≥ L3 ≥ MM ≥ DISK
HDD, SSD

case1: Empty frame available in physical memory.

→ P_1 moved from swap to f_2 → Disk Access
→ PT is updated → MA Memory Acc
→ P_i from blocked to ready State

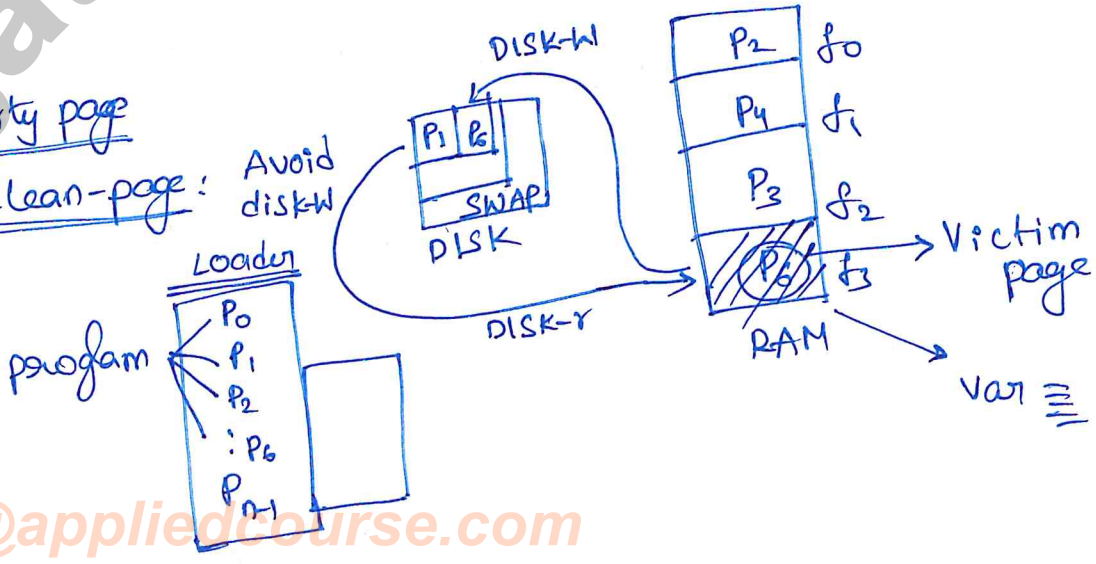




→ Empty Frame

case2: No Empty Frame [page-replacement-algo]
↓
FIFO, LRU, MRU

↳ Dirty page
↳ clean-page: Avoid disk-W



Demand paging:

- Pure Demand paging [Default]
- prefetch Demand paging → Some pages are preloaded into PAS before the start of process.

Performance of Virtual-Memory + Demand paging

① $EMAT_{DP} = p(s+3m) + (1-p)2m \approx ps + (1-p)(2m)$

MMAT = m ns

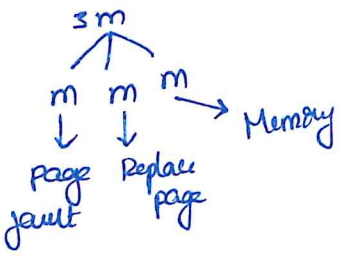
page fault service time: PFST = s ms

$s >> m$

page-fault rate = p

$0 \le p \le 1$

page-hit-rate = 1-p



② PFST (Page fault Service Time) = 10 MS = S = 10,000 ns

MMAT = m = 1 µs = 0.001 ms

page-fault rate = p = 0.01% = 0.0001

$EMAT_{DP} = 0.0001(10,000+3) + (0.9999)2$

≈ 2.9999 µs

(a) Instruction-time = i μs

Time for page-fault = j μs j >> i

PFST = i + j μs

one in k instructions has a page-fault

PFR = 1/k

$EMAT_{DP} = \frac{1}{k}(i+j) + (1-\frac{1}{k})i$

$= (i + j/k)$

i = Normal Instruction Execution time.

j = Additional execution overhead (time) in case of page fault.

Page Replacement Algorithms:

page-reference-string:

P_i : $VA_1, VA_2, VA_3, VA_1, \ldots\ldots$ (Sequence of Reference)

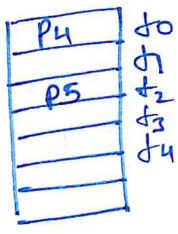
→ time

4, 5, 5, 4, ... Sequence of page Numbers

pure-demand page

L : Length

n : # unique pages



main-memory

Frame Allocation policies:

VMM : MP | MT

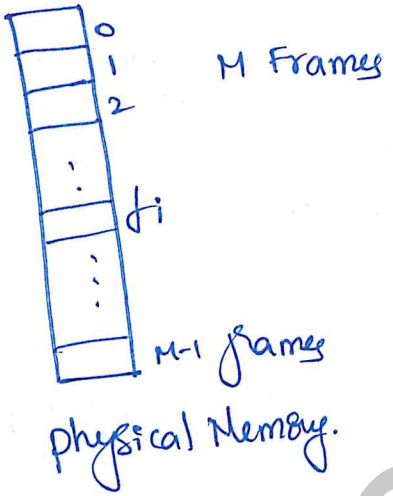
$Pr_1, Pr_2, Pr_3, Pr_4, Pr_5$

$S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5$: Demand

$\sum S_i = S > M$

$Pr_1 \; Pr_2 \; Pr_3 \; - - \; Pr_5$ $\boxed{M=40}$

Frames in physical-memory



i	S_i	a_i equal alloc	a_i prop. alloc	a_i 50% rule
Pr_1	11	8	$11/80 * 40 = 6$	6
Pr_2	4	8	$4/80 * 40 = 2$	2
Pr_3	20	8	.	
Pr_4	15	8	.	
Pr_5	30	8	.	
	$S=80$		M	

$S = \sum S_i$

May not be equal to M.

a_i = # frames allocated to Pr_i

proportional allocation

$$a_i = \frac{S_i}{S} * M$$

(Q) Reference String = 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1,
7, 0, 1

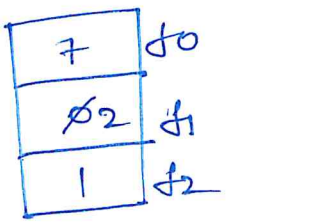
$L = 20$

Set of pages = {7, 0, 1, 2, 3, 4}; $n = 6$
unique

pure demand paging:

P_0

page Replacement Algorithm



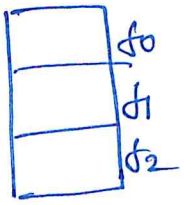
① First in First OUT: (FIFO)

	f	V/I	TOL
0			
1			
⋮			
7			

Time of Loading / AT

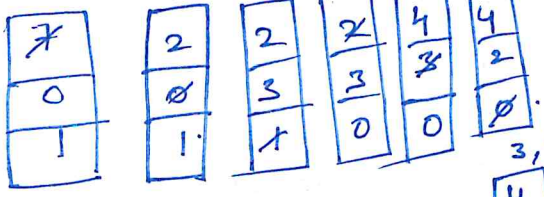
String: 7, 0, 1, [2], 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

PT of P_i



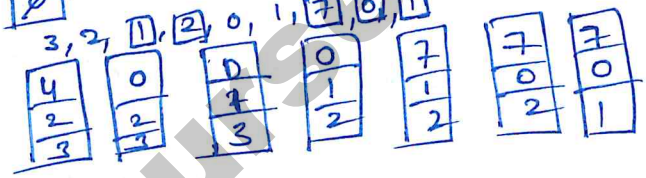
Physical Memory

7, 0, 1, [2], 0, [3], [0], [4], [2], [3], 0



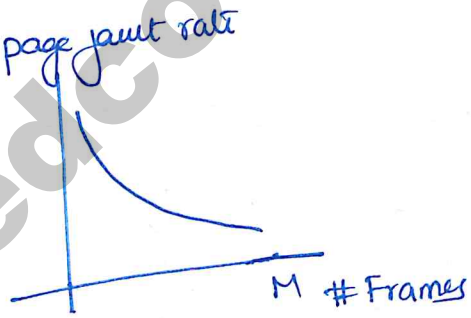
page faults = 15

3, 2, [1], [2], 0, 1, [7], [0], [1]



Frames = 4

page faults = 10



Eg: Ref-string = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

ⓐ # Frames M = 3, # page-faults: 9

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4	4	4	5	5	5						
2	1	1	1	3	3						
3	3	2	2	2	4						

ⓑ M = 4, # page-faults: 10

1	2	3	4	1	2	5	1	2	3	4	5
5	1	2	3	4	5						
4	5										

Frames are Increased page faults are also increasing.

Belady's Anamoly

(2) Optimal Replacement (min # of page faults)

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

M = 3

Replace the page that is not needed longest into the future.

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

page faults = 9

drawback: Not Implementable.

{NO Belady's Anamdy}

(3) Least Recently used (LRU):

	f	V/I	AT	TOR
0	f1	1	✓	
1	f2	1	✓	
2		0		
:		0		
:				
7	f0	1	✓	✓

TOR → Time of Reference

r.s = 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

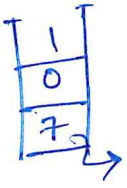
→ Time

PT of Pi:

f0	7	2	4	0	1
f1	0		3	0	
f2	1	3	2	7	

page faults = 12

Implementation: stack + Linked list



Optimal & LRU : Stack Algorithms [No Belady's Anomaly]

PRA condition: [Set of pages in Memory with n-frames ⊆ Set of pages in memory with n+1 frames]

LRU

(n)

(n+1)

(Pi)

(4) Most-Recently-Used (MRU):

M = 3 r.s = 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

→ Time

7	7	0																	
0		0	1																
1			1	2															

page-faults = 16

(5) Counting/Frequency Algorithm: → Least frequently used / → Most frequently used.

	d	V/I	AT	TOR	FC
0	d1				10
1	d2				6
2	d0				1
:					
7					

→ Frequency Count

updating the FC for every memory reference.

PT of Pi

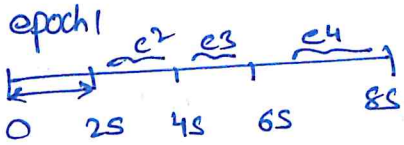
⑥ LRU-approximation Algorithm [Very Widely used]

ⓐ

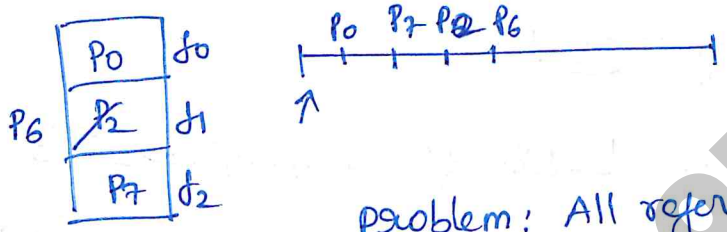
	f	V/I	AT	R
0	0	1	30	1
1	–	0	–	–
2	1	1	2	1
:				
n	2	1	0	1

PT of P_i

R: Reference bit



→ @ Start of new epoch make Rbit = 0 ∀ pages.



problem: All reference bits = 1

ⓑ Second Chance / Clock Algorithm:

→ Just like the reference-bit methods.

→ If R=1 ∀ pages
then FIFO on Arrival time

ⓒ Enhanced Second chance: [Modified second chance]

PT: f, V/I, AT, R, M

P_i

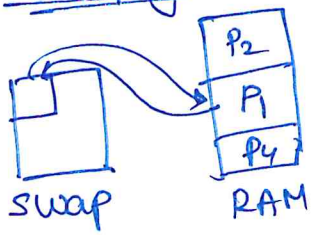
0 0

↙ Ist chance

cpu
P_i

	F	V/I	AT	R M
P_1		1		0 0 → 1st
P_2		1		0 1 → 2nd
P_3		1		1 0 → 3rd
P_4		1		1 1 → 4th

clean page → disk-r
dirty page → disk-w + disk-r



VMM

(a) Ref-string-Length = L

0, 1, 2, 3, 4, ... L

n = L

Unique-pages = n

0, 1, 0, 1, 0, 1 ...
n = 2

frames allocated to P_i = m

calc the lower and upper-bound on the # page-faults using any page-replacement algorithms.

Let m = 1 

m ≥ n

max # pag faults = L

min # page faults = n

(c)

	TOL	TLR	R	M
P0	126	279	0	0
P1	230	260	1	0
P2	120	272	1	1
P3	160	280	1	1
	AT			

{ Which page is optimized when page-fault occurs.

TOL: Time of Loading

TOR: Time of Last reference.

(a) FIFO : P_2 is replaced

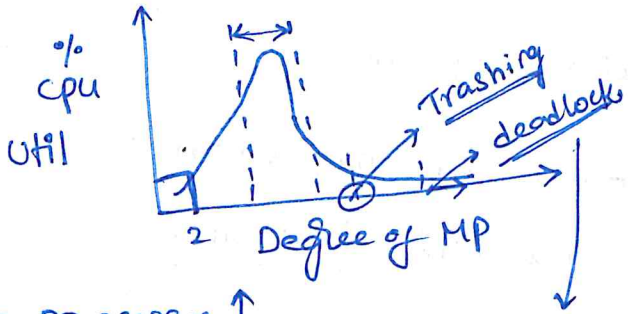
(b) LRU : P_1 is replaced

(c) Second-chance : P_0

(d) Enhanced Second Chance : P_0

Trashing:

Excessive page-activity
↳ v. high page fault rate.



vmm → Lots of time handling page faults.

{ # processes ↑ → PAS ↓

Extreme case

[Low cpu util]

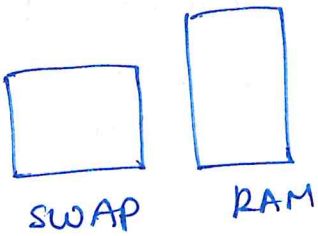
Reasons: Primary → too Few Frames, Too many processes

Secondary → page Replacement Algorithm, Small page size, program, DS

Thrashing-control-strategies → Prevention # process ≤ k
→ Detection & Recovery (degree of MP)

Detection of Thrashing:

→ Low cpu utilization, high-degree of MP
→ high-paging-disk util



Recovery
↓
{ Suspend/block
a few process

Secondary-reasons

page-size → Too small ⇒ Pri has to handle too many pages.
↓
4KB → too-large ⇒ Contiguous memory allocation

→ Lesser chance of Thrashing

→ Internal Fragmentation.

Program structure & Design Techniques

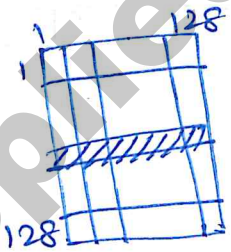
int A[1---128][1--128]; Let ps = 128B

① for i = 1 to 128
  for j = 1 to 128
    A[j,i] = 1

② for i = 1 to 128
  for j = 1 to 128
    A[i,j] = 1

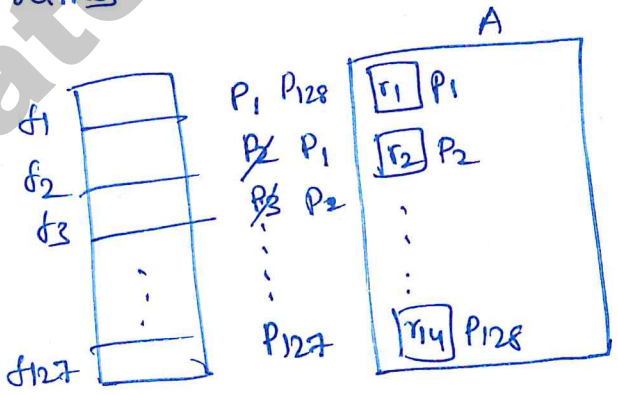
	P_1	P_2		P_{127}	P_{128}
1st col	A[1][1],	A[2,1]	A[3,1] ---		A[128,1]
2nd col	A[1,2],	A[2,2],	A[3,2] ---		A[128,2]

Row1: A[1,1] A[1,2] --- A[1,128]
P2: A[2,1] A[2,2] --- A[2,128]



128×128 B

Frames Allocated = 127



Row-major:

A[i][1-----128] P_i

Pure-demand paging FIFO

P_1 A[1,1] A[1,2] ---------- A[1,128] ← row1

P_2 A[2,1] A[2,2] ---------- A[2,128] ← row2

Program1: 128 X 128 → Too many page faults.
FIFO

Program2: 128

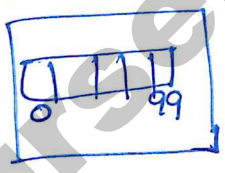
Eg: Linked List vs array

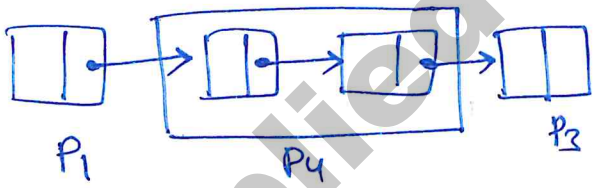
↓ less Wastage.

↓ wastage of space

Higher chance of page faults.

int a[100]





(a) VAS = PAS = 2^{16}B ⇒ PA = VA = 16 bit

PS = 512B page-offset (d) = 9 bits.

PT entry size = 2B = 16 bits

PT: f, V/I, R, M, protection, Age How old page is MM
(1) (1) (1) (3) (x)

3 bits

Find x

f + 6 + x = 16

frames = $\frac{2^{16}}{2^9} = 2^7 =$ 128

7 + 6 + x = 16 ⇒ x = 3

More solved problems:

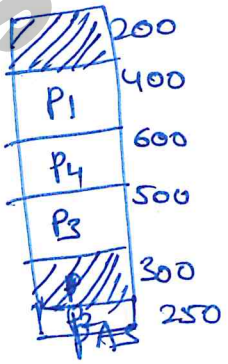
(1) Consider six memory partitions of size 200KB, 400KB, 600KB, 500KB, 300KB, and 250KB, where KB refers to Kilobyte. These partitions need to be allotted to four processes of sizes 357KB, 210KB, 468KB and 491KB in that order. If the best fit algorithm is used, which partitions are not allotted to any process?

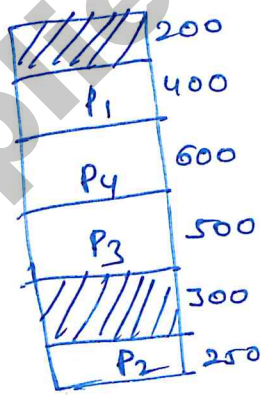
(A) 200KB, and 300KB

(B) 200KB and 250KB

(C) 250KB and 300KB

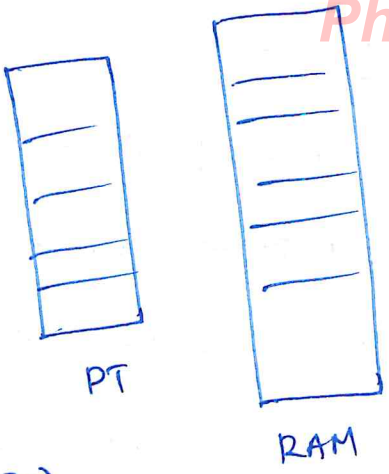
(D) 300KB and 400KB





PAS

(2) Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 ms to search TLB and 80 ms to access the physical memory. If the TLB Hit ratio is 0.6, the effective memory access time (in ms) _____



100 ns - TLB cache

80 ns - RAM

122 MS

EMAT = 0.6 (10+80) + 0.4 (10+80+80)
TLB+PT

= 54 + 68

= 122 ns

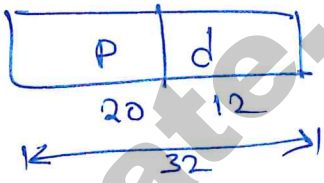
(3) Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabits 4MB

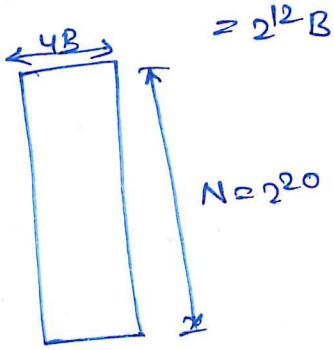
1 W = 1 B

LA = 32 b

PS = 4 KB ⇒ d = 12 bits

= 2^{12} B





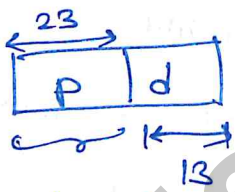
= 2^{20} × 4B

= 4MB

④ A Computer system implements 8 kilobyte pages and a 32-bit physical address space. Each page table entry contains a valid bit, a dirty bit three permission bits, and the translation. If the maximum size of the page table of a process is 24 megabytes the length of the virtual address supported by the system is _____ bits.

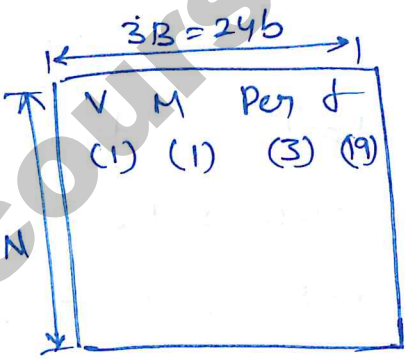
PS = 8KB = 2^{13} B

PA = 32 bit



$\log_2 N$

$\Rightarrow$ 36 bits



$= \frac{24 \times 2^{20} B}{3B}$

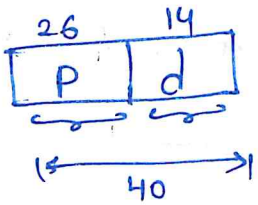
$= 8 \times 2^{20}$

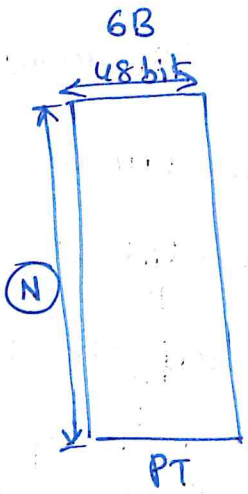
$= 2^{23}$

⑤ Consider a Computer System with 40-bit virtual addressing and page-size of 16 kilobytes. If the Computer System has a one-level page table per process and each page-table entry requires 48 bits, then the size of the per-process page table is _____ megabytes.

VA : 40 bit

PS : 16 KB
$= 2^{14}$ B





$N = 2^{26}$

Size of PT $= 2^{26} \times 6B$

$= 2^{26} \times 6 \times 2^{20} B$

$= 64 \times 6 MB$

$= \boxed{384 MB}$

(6) Consider a process executing on an operating system that uses demand paging. The average time for a memory access in the system is M units if the corresponding memory page is available in memory, and D units if the memory access causes a page fault. It has been experimentally measured that the average time taken for a memory access in the process is X units.

Which one of the following is the correct expression for the page fault rate experienced by the process?

(A) $(D-M) / (X-M)$

(B) $(X-M) / (D-M)$

(C) $(D-X) / (D-M)$

(D) $(X-M) / (D-X)$

$X = r(D) + (1-r)M$

$X = rD + M - rM$

$r = \frac{X-M}{D-M}$

(7) Assume that in a certain Computer, the virtual addresses are 64 bits Long and the physical addresses are 48 bits Long. The memory is word addressable. The page size is 8KB and the word size is 4 bytes. The Translation Look-aside buffer (TLB) in the address translation path has 128 valid entries. Atmost how many distinct virtual addresses can be translated without TLB Miss?

(A) 16×2^{10}

(B) 8×2^{20}

(C) 4×2^{20}

(D) 256×2^{10}

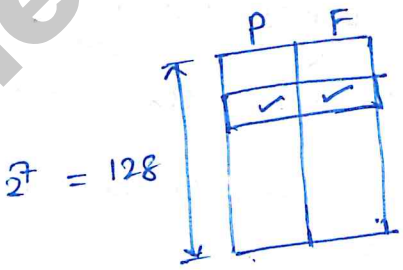
VA : 64

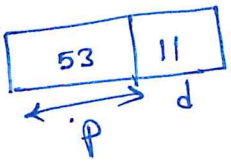
PA : 48

PS : 8KB $= 2^{13}$ B $= 2^{11}$ W

1W = 4B $= 2^2$ B

Words per page $= 2^{11}$





1page $= 2^{11}$ virtual addresses

$\Rightarrow 2^7 \times 2^{11}$ B

$\Rightarrow 2^{18}$

$\Rightarrow 256 \times 2^{10}$

More Solved problems-II

(1) Assume that there are 3 page frames which are initially empty. If the page reference string is 1, 2, 3, 4, 2, 1, 5, 3, 2, 4, 6, the number of page faults using the optimal page replacement policy is 7

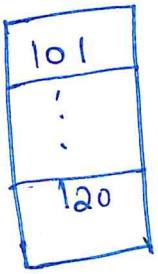
pure-demand paging

1 2 3
2
3 4

page faults: 1, 2, 3, 4, 5, 3, 6

(2) A Computer has twenty physical page frames which contain pages numbered 1, 2, ... 100 in that order, and repeats the access sequence THRICE. which one of the following page replacement policies experiency the same number of page faults as the optimal page replacement policy for this program?

Now a program accesses the pages numbered 101 through 120.



1, 2, ... 100,
1, 2, ... 100,
1, 2, ... 100

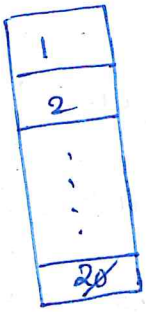
(A) Least-recently-used

(B) First-in-First out

(C) Last-in-First-out

(D) Most-recently-used.

1, 2, 3 . . . 100 1, 2, 3 . . . 100 1, 2 . . . 100



1 2 3 20 21 22 23 . . . 100

MRU
Most Recently used.

In this given problem optimal page replacement algorithm works similar to MRU.

(3) A system uses 3 page frames for storing process pages in main memory. It uses the Least Recently used (LRU) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below?

4, 7, 6, 1, 7, 6, 1, 2, 7, 2

(A) 4
(B) 5
(C) 6
(D) 7

4 1
7 2
6 7

(4) Consider a main memory with five page frames and the following sequence of page reference: 3, 8, 2, 3, 9, 1, 6, 3, 8, 9, 3, 6, 1, 2, 3. which one of the following is true with respect to page-replacement policies First-in-First-out (FIFO) and LRU (Least recently used).

(A) Both incur the same number of page faults.

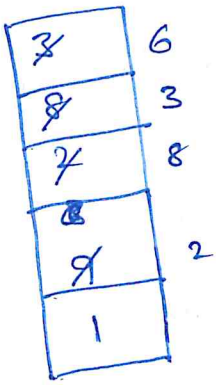
(B) FIFO incurs 2 more page faults than LRU.

(C) LRU incurs 2 more page faults than FIFO

(D) FIFO incurs 1 more page faults than LRU.

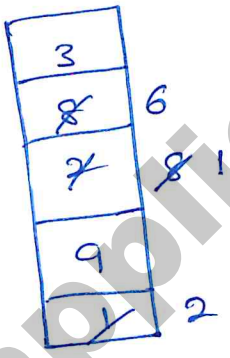
Pure-Demand paging.

FIFO:



5+4 = 9

LRU:



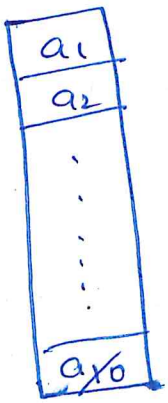
5+4 = 9

(5) Consider a Computer System with ten physical frames. The System is provided with an access sequence $a_1, a_2 \ldots a_{20}, a_1, a_2 \ldots a_{20}$), where each a_i number. The difference in the number of page faults between Last-in-first-out page replacement policy and the optimal page-replacement policy is ____ (1)

Sol:

Optimal

$a_1, a_2 \cdots a_{19}, a_{20}\ a_1, a_2, \cdots a_{19}, a_{20}$



$a_{11}\ a_{12} \cdots a_{20}$

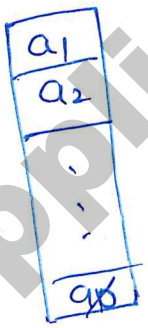
for $a_1, a_2 \cdots a_{20}$ — 20 page faults.

From $a_1, a_2 \cdots a_9$ — No page faults.

$a_{10} \cdots a_{19}$

Total # page faults = 20 + 10 = 30.

LIFO:



$a_{11}\ a_{12}\ a_{10}\ a_9\ a_{11} \cdots a_{19}\ a_{20}$

First part: $a_1 \cdots a_{20}$ — 20 page faults.

Second part: $a_1 \cdots a_9\ a_{10}\ a_{11} \cdots a_{20}$

$a_1 \cdots a_9$ — NO PF

$a_{10}\ a_{11} \cdots a_{20}$ — page Faults.

page faults = 20 + 11 = 31

Difference = 31 − 30 = 1

(6) In which of the following page replacement algorithms it is possible for the page fault rate to increase even when the number of allocated frames increases?

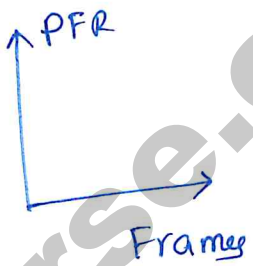
(A) LRU (Least-Recently-used)

(B) OPT (Optimal page Replacement)

(C) MRU (MOST Recently Used)

(D) FIFO (First in First OUT)

⇓ Belady's Anamoly



(7) Recall that Belady's anomaly is that the page-fault rate may increase as the number of allocated frames increases. Now consider the following statements.

S1: Random page replacement algorithm (where a page choosen at random is replaced) suffers from Belady's Anamoly.

S2: LRU page replacement algorithm suffers from Belady's Anamoly.

which of the following is correct?

(A) S1 is true, S2 is true

(B) S1 is true, S2 is False

(C) S1 is False, S2 is True.

(D) S1 is False, S2 is False.

{subset/ set of pages with n fams ⊆

set of pages with n+1 fams }

Files and Disks : Hardware Internals

File-system & Disks: We will get a high level overview of how the CPU works with the disk via the File system.

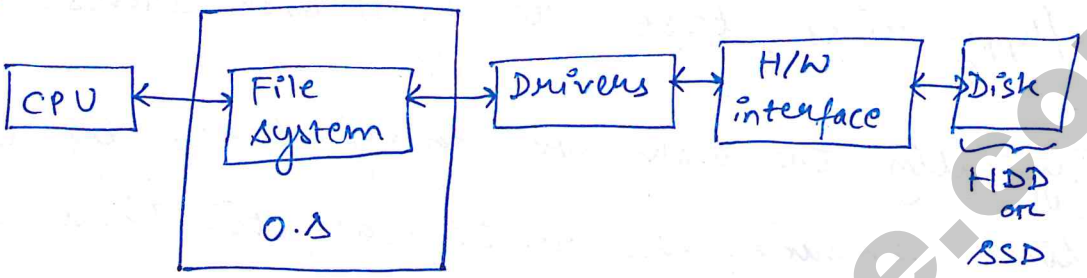


Fig: Block diagram

We have seen:

C:\> } Drives or volumes
D:\> } (Windows) (Linux, Unix, Mac)

File system is a visible component of the operating system. It is widely used component.

Most of the current literature that we study is in the context of Hard disk drive (HDD). The more recent technology solid state drive (SSD) will not be covered under the scope of GATE exam.

Within the operating system there is a piece of software called the File system. The File system basically manages all the data that is permanently stored on the disk. In comparison,

the data is RAM is volatile i.e., as soon as we turn off the power, all the data in the RAM is erased or vanishes.

There has been huge revolution in the disks over the last three to four decades. We went from floppy disks, tapes to hard disk drives.

Typically the disks are manufactured by companies like Samsung, Western Digital, etc. These companies provide us the device driver. The device driver is basically a snippet of code that can talk to the file system and that can also talk to the hardware interface. The hardware interface is what gets connected to the actual physical disk.

Sometimes the device drivers are also part of the operating system (the general purpose device driver). But sometimes the device driver is external to the operating system (the manufacturer gives the device driver). Device drivers are different for different operating systems.

The hardware interface is also called as controller (or interface card).

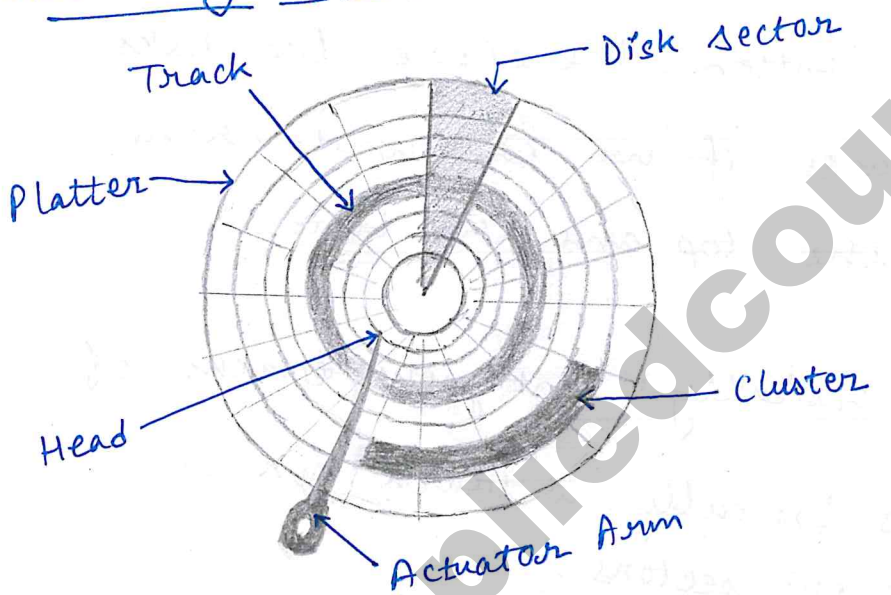
E.g., SATA is a type of hardware interface which is faster than the older technologies.

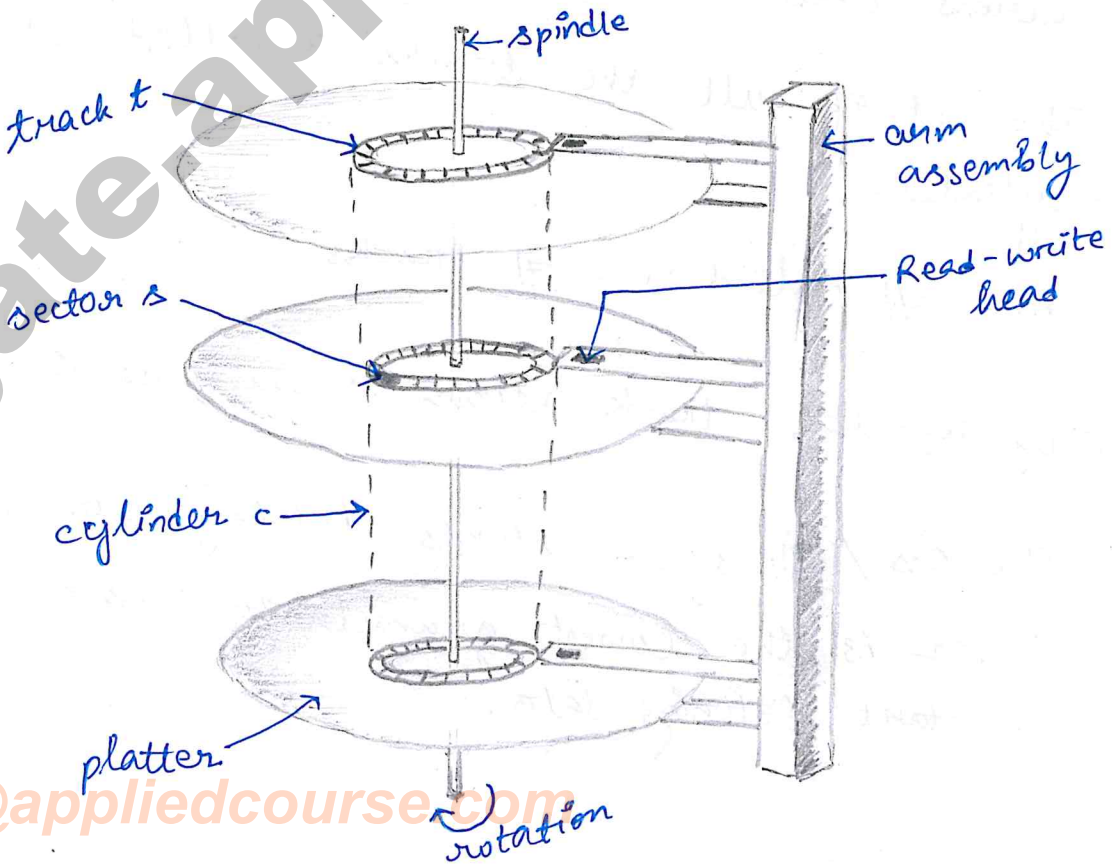
Another technology called SCSI (older technology) is also a hardware interface.

The hardware interface could be built on the motherboard itself.

IDE is another hardware interface technology.

Geometry of a disk:





The physical spindle rotates and along with it all the platters also rotate.

The read write head can read from any of the platters. Read write head has multiple arms, one arm for each of the platter.

We can think of these platters as disks.

For some disks, we can write both at the top and the bottom. Therefore, the disks have 2 surfaces if we can write/read both from the top and the bottom.

A cluster is basically a track spread across contiguous sectors.

The set of all the tracks is called as cylinder.

And # cylinders = # tracks

By default, track sector is same as sector.

The OS / file system starts fetching data per sector. Sector is the lowest granularity in which we start fetching data.

What is the data that we store in every sector?

We store sector number, data and Error correcting codes (ECC).

ECC is a concept from electronics engineering.

Hard disks when slightly older get corrupted, means the data is no more useful.

The hard disks rotates very fast. The hard disk is an electro-mechanical device. The read/write head touches the disk to read the data on the disk. The disk rotates too many times and due to this there can be physical damage (over long duration), and the data in the sector could be corrupted.

Therefore to determine whether or not the data in the sector is corrupted, we use error correcting code.

In a high level overview, what ECC does is :-

We compute a function on the data. The function could be hash function, etc.

Say, f(data) = 10 bits value, the function returns us a 10 bit value which is what store as ECC in the sector information.

Whenever the data on the sector modifies, we recalculate ECC using the function f.

If both the 10 bits numbers match, then the data which we have written is not corrupted. The corruption can be of data as well as of the ECC number. In both the scenarios, the sector data is lost.

Important terminologies:

I. Seek time (S.T): Time to move from one track to another track.

II. Rotational latency / Latency time (L.T):

If R = time for one rotation,

then $LT = \frac{R}{2}$ } on an average

Rotational latency is the time taken to get the right disk sector in place.

III. Transfer time (TT): The time taken to read from the sector from disk to the main memory.

The Working:

The arm reads the data, then sends the data to the hardware interface. Then the driver takes the data and gives it to the O.S. Then the O.S stores it in the memory. Then the data is accessed by the CPU.

Note: The HDD is very different from the SSD.

The SSD doesnot have all of the electro-mechanical parts. The SSD is made of chips (electronic chips). The moving parts of the HDD is completely avoided in the SDD. That's why the SSD's are significantly more expensive than the HDD.

Both HDDs and SSDs are widely used today and slowly people are moving towards SSD because they are more reliable, they last longer and they are significantly faster than the HDD.

HDD's are extensively used because they are significantly cheaper compared to SSDs and HDD can store far more at a lower cost than the SSDs.

The SSD architecture is not the same as that of the HDD. SSD comprises of a

bunch of chips like what are there on the smartphones. The 16 GB, 64 GB etc storage that we get on the smartphone is the SSD.

Disk I/O time = Seek Time + Latency Time + Transfer time

= ST + LT + TT

Qn Imagine a disk that has :-

16 platters
2 surfaces
2K tracks
512 sectors
2KB / sector

Average seek time = 30 ms
Disk rpm = 3600 rpm (revolutions per minute)

(a) What is unformatted capacity of disk?

When we format the disk, some of the disk space is taken up to store some information about the disk.

For e.g., for every sector we have to store sector number, data and ECC. Out of these, sector number and ECC are the book-keeping data (as these are not the actual data)

$\therefore$ $16 \times 2 \times 2K \times 512 \times 2KB$ (16 platters, 2 surfaces)

$= 64 GB$ = Total storage of the total disk.

(b) What is I/O time for reading a sector?

To read one sector, time required is

$ST + LT + TT$

$ST = 30 ms$

$\text{avg } LT = \frac{R}{2} = \frac{1}{60 \times 2}$

$= \frac{1s}{120} = 8.3 ms$

$60s \longrightarrow 3600 r$

$1s \longrightarrow 60r$

$1r \longrightarrow \frac{1}{60}$ seconds

When the disk rotates once completely, we can read everything that is there in the track.

In each track we have 512 sectors.

In each sector we have 2KB / sector

$\therefore$ One track has $512 \times 2KB$ amount of data.

$= 1 MB$ data, when disk rotates once.

To rotate once, disk takes

$\frac{1}{60}$ seconds.

It means, disk takes $\frac{1}{60}$ seconds to transfer 1MB data.

$\frac{1}{60}$ seconds ⟶ 1MB

$\frac{1}{60} \times \frac{1}{1M} \times 2K$ ⟵ 2KB

$\approx 33.3\ \mu s$ = Transfer Time

$\therefore$ ST + LT + TT

$= 30\ ms + 8.3\ ms + 33.3\ \mu s$

$= 30\ ms + 8.3\ ms + 0.0333\ ms$

(c) Effective data transfer rate is Bytes/seconds

In $\frac{1}{60}$ seconds, we can transfer 1MB

In 1 second ⟶ $1MB \times 60 = 60\ MB/second$

$= 60\ MB/s$

[on an average]

(8) How long does it take to load 64KB program from disk whose average seek time = 30 ms, the disk takes 20 ms for one rotation. The track size = 32KB and its page size = 2KB.

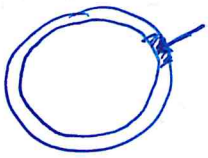
Assume pages are spread across randomly around the disk.

64KB means $\frac{64KB}{2KB}$ = 32 pages.

For each page, we have,

1 page = ST + LT + TT

= 30 + 10 + x = 30 + 10 + 1.25 ms = 41.25 ms



In 20 ms we can complete one rotation.

∴ In 20 ms how much data can we transfer?

The data that is there in the whole track = 32KB

∴ In 20 ms we can transfer 32KB.

For 2KB page size, the time required is $\frac{20}{32} \times 2 = 1.25$ ms

∴ x = 1.25 ms

We have 32 pages.

∴ For 32 pages (64KB program), the loading time = 32×41.25 ms.

(Q) # surfaces = 6

Inner diameter = 4 cm

Outer diameter = 12 cm

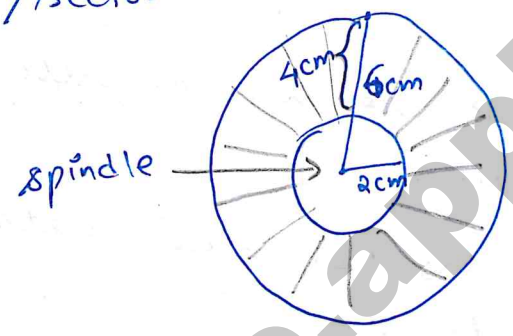
Inner track space = 0.1 mm

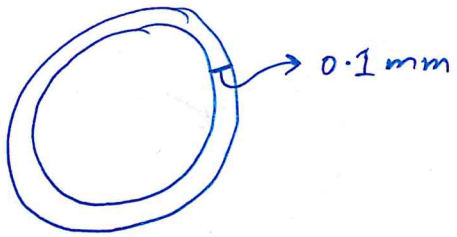
sectors / track = 20

Avg ST = 20 ms

3600 rpm

Data / sector = 4KB





(a) Capacity of disk

0.1 mm ⟶ 1 track } 1 mm ⟶ 10 tracks, 1 cm ⟶ 100 tracks

4 cm ⟶ 400 tracks

We have 20 sectors / track.

∴ Total we have $400 \times 20 = 8000$ sectors

There are 6 surfaces

∴ Total number of sectors across 6 surface

= 6 × 8000 sectors

In each sector we can store 4KB data,

∴ In total, capacity of the disk

= 6 × 8000 × 4KB

(b) Sector I/O time (How much time does it take to read data from a sector)

A track has 20 sectors.

A sector consists of 4KB.

∴ 1 track has 80KB

Given 3600 rpm,

i.e., $\frac{1}{60}$ seconds ⟶ 1 rotation

Note: In one rotation, we can cover 1 track.

∴ $\frac{1}{60}$ seconds ⟶ 80 KB

Now, 1 track is 4KB

∴ For 4KB, the time required = $\frac{1}{60} \times \frac{1}{80} \times 4$

$= \frac{1}{1200}$ seconds

∴ The sector transfer time = $\frac{1}{1200}$ seconds

Sector I/O time = ST + LT + TT

= 20 ms + $\frac{1}{2}$ r + $\frac{1}{1200}$ seconds

= 20 ms + $\frac{1}{2} \times \frac{1}{60}$ seconds + $\frac{1}{1200}$ seconds

Logical structure of a disk : Partitions, Files and Directories.

Since we have seen the physical and geometrical structure of a disk, now we will look at the logical structure.

Logical structure is what the O.S or the File system looks at a disk to be.

We know about formatting of a disk.

By formatting a disk we are creating logical structure in the disk, which is what we see (e.g directories, volumes, drives, etc).

When we format we are basically creating a bunch of data structures/tables in which we will store data.

When we format, we partition the whole hard disk into multiple drives/volumes. The volumes (in Linux) is mounted on some directories.

Partitions are of two types:

(i) Primary partition
- Bootable
- O.s + data
- some O.s also can store user data (Windows)

(ii) Secondary partition
- Non-bootable
- data (user data)

Note: A system can have multiple primary partition and multiple secondary partition

In multi-boot systems we can have:

Note: Bootloader (e.g., GRUB in Linux) keeps track of all the Operating systems that are installed and the partitions in which they are installed.

Master Boot Record (MBR):

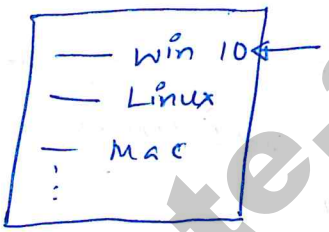
When partitions are created, O.s creates MBR which is stored on the first primary partition.

When the computer is turned on, the Bootloader looks at the first chunk of the first partition.

MBR stores the partition table. The partition table tells us what all partitions are there, which all clusters, cylinders, surfaces, sectors is each of these partitions stored.

The MBR also consists of the Bootloader (like GRUB). Bootloader is a piece of software which starts fetching data from the first partition from the MBR itself.

The GRUB based bootloader gives us a GUI (Graphical User Interface) with options of which operating systems we want to load.



If MBR is corrupted, the disk is gone. This can be corrupted due to electro-mechanical failures.

Structure of a partition:

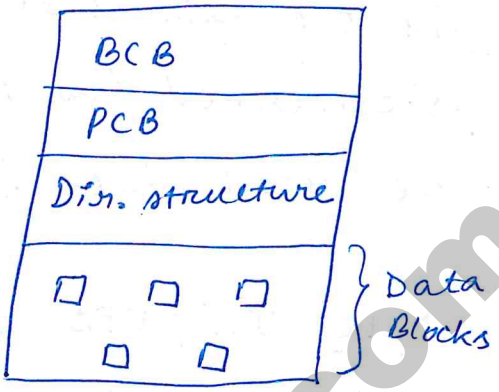
1. Boot Control Block (BCB):

It contains the kernel of the Operating system.

Only primary partitions (bootable) contains a Boot control Block because BCB enables us to boot the kernel of the operating system.

The GRUB will transfer the control to BCB. Within the BCB, we have a bunch of code that loads the operating system and starts everything.

logical structure



Note: Implementation of each of these components differs from one file system to other file system (E.g. NTFS, FAT32, EXT4, JFS, UFS, ZFS)

2. Partition control Block (PCB):

This stores lot of information about the partition, e.g., what is the size of the partition, how many free blocks are there in this partition, OS blocks.

The PCB is called as Master file table (in Windows based file system like NTFS and FAT 32)

A data block is basically cluster of sectors.

All the data that we read, as far as O.S is concerned (as far as logical view is concerned) we read data in blocks.

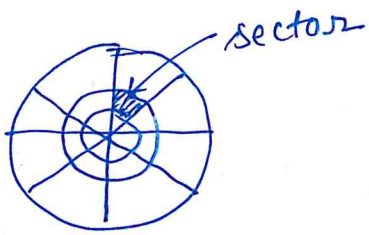
Note: In UFS file system, PCB is called as super block. UFS is popular in Sun micro-systems O.S called Solaris } It is no more popular today.

3. Directory structure :

There are multiple types of directories :

(i) Single level
(ii) Two level
(iii) Hierarchical
(iv) DAG (Directed Acyclic graph based directories).

4. Data Blocks (Cluster of sectors)



A file is stored on multiple blocks. Each block corresponds to a cluster

File: It is a collection of data blocks. } Physically.

Logically { From implementation perspective, we can think of a file as a data structure, which stores lot of information about physical data blocks (about the size of the file etc).

File is what the O.S sees. At the end of the day, everything is a sector.

The information that is stored in the file data structure are:

size of a file, name of a file, etc.

All these are attributes of a file.

The file data structure also contains operations that we can perform:

create a file, read a file, write a file.

For file handling in C programming, we see operations like:

fopen(), fseek(), fclose(), fprintf(), fscanf(), etc.

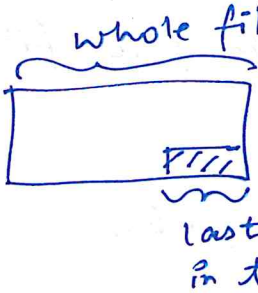
Structure of the file –

- Table
- Tree (B+ trees)
- series of bytes
- Flat size

File Operations :

create
open
read
write
modify
truncate
seek

copy
move
rename
permissions ← user u_1, u_2, u_3
close
delete



Truncation is removing last few bytes from the file.

close - remove the file from main memory but it should be available in the disk.

delete - remove the file even from the disk.

For most of these file operations, there are two levels:

(i) system calls / program level access

(ii) user level access.

Say in Windows O.S, we can copy a file from one location to another using the GUI. In Linux, we use command called cp.

All of these commands are implemented in the O.S as a bunch of system calls. For e.g., in Unix based system, there are functions calls like open(), read(), close() etc.

It is O.S which has control over the file system. O.S controls the physical disks via the drivers.

File attributes :

- Name
- extension
- type (executable, etc)
- size
- owner
- path (logical path)
- timestamps (when was it created/last modified)

- location (block numbers)
- permissions
- link-count (These are software links or pointers)

All the file attributes are stores in a place called File control Block. File control Block is also a part of the data associated with the File.

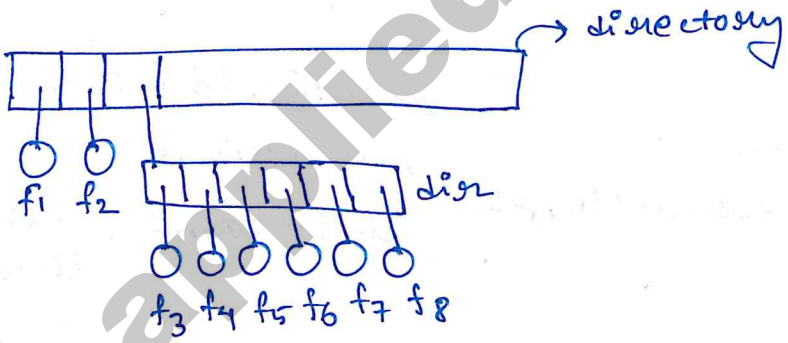
Directory:

Internally, a directory is also stored as a file. This file doesnot have data associated with it. It has meta-data and some details about the directory (structure) itself.

∴ Directory is a special type of file.

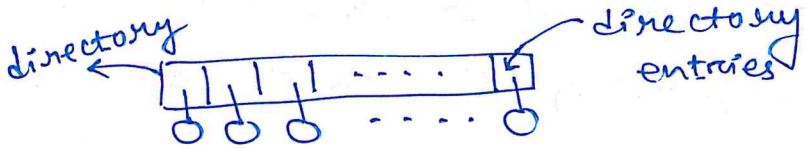
Abstract view of directory by the O.S –

The O.S views a directory as a series of directory entries. These entries are also referred to as words in some operating systems.



Directory structure:

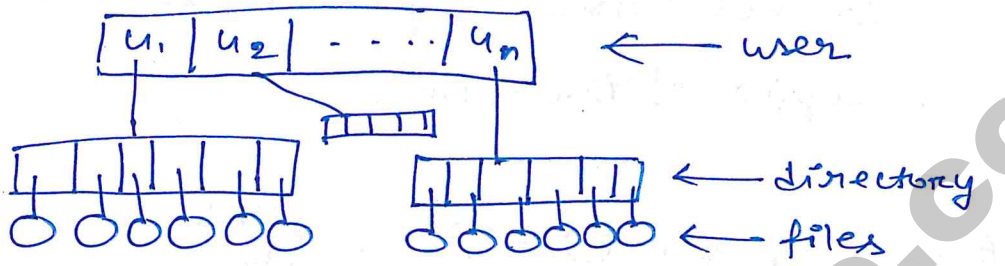
1. Single level:



It is not very popular now-a-days. It was popular in very early days of O.S

2. 2-level :

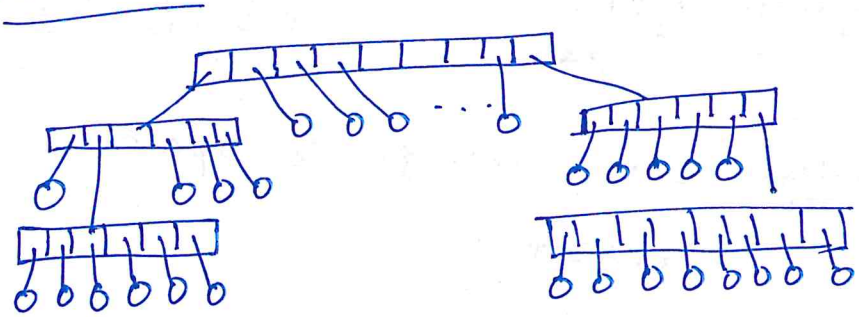
When multi-user O/s popped up (Unix based), (developers) they created a series of users.



Disadvantage with this approach is, whatever file is allocated to a user, it won't be visible to the other users. If other users have to access those files, they have to copy these files and store it under their own directories.

This type of directory structure is no more popular today. It was famous during the first iterations of Unix.

3. Tree / hierarchical structure :



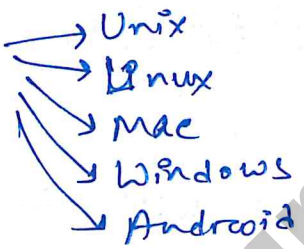
Each of the files as leaf node. It can be thought of as n-ary tree.

Disadvantages:

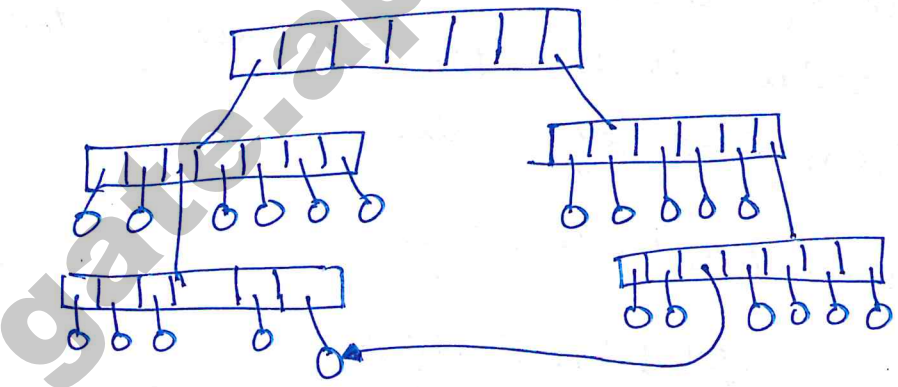
① Wastage of disk space (copy and paste to have access to files)

② Inconsistency — The same modification is not visible / updated to all the copies of the file.

4. Acyclic graph (DAG):

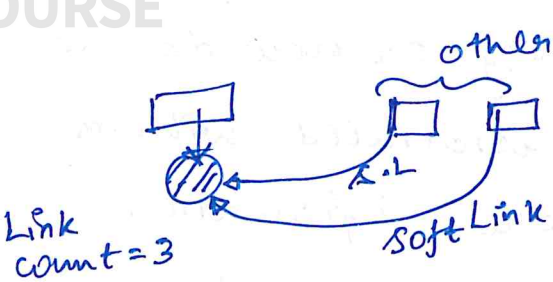


This method is similar to creating a soft link. In windows also, we can create a link to the file and place it in a different folder.



Advantage: Less wastage of space

Consistency.



If the owner of file decides to delete the file, we will decrement the link count whenever one of the links to the files is removed. We don't actually delete the file. The file will be available on the physical disk, it is just that the in the original directory, we don't see the file to be available anymore.

Only when the link count becomes 0, the actual file is deleted.

This method/type of directory structure is most popular and widely used today.

File System Implementation

Implementing a File System:

One of the most important aspects of a File system is how do we allocate various blocks to a file. How do we deallocate (when we delete a file or when we make all the

link count of a file to zero, how do we deallocate a file completely or how do we remove all the blocks associated with a file. We also need to store information about the file itself, how do we store that. Also, we want to ensure protection. By protection, it meant that if a file is to be accessed by one user only, then it should not be accessed by other users. These are all the key operations we have to think of when we are implementing a file system.

The most important thing is allocation. It's easy to store information. It is easy to ensure protection by saying which users have access to it. Once we know allocating, deallocating will also be straightforward.

The most important aspect of implementing file system is how do we allocate various data blocks in a partition to a file.

For every data block in a partition, there are two things:

1. DBA: Data Block Address — which is literally an address to each of these data blocks. It is also called as Disk block address.

Note: A block is a cluster of sectors.

For every block, there is an address associated to it.

2. DBS: Disk Block Size — every block has a size associated with it. In many Operating system, this is often constant.

Allocation Methods:

What is the objective of allocation methods?

Imagine we are creating new file, how do we decide which data block to assign or allocate to a file? That is the task of allocation methods.

A comparison:

Instead of pages and frames in main memory, here we have blocks. Just like a frame is a unit of storage in main memory, a block is a unit of storage in the disk.

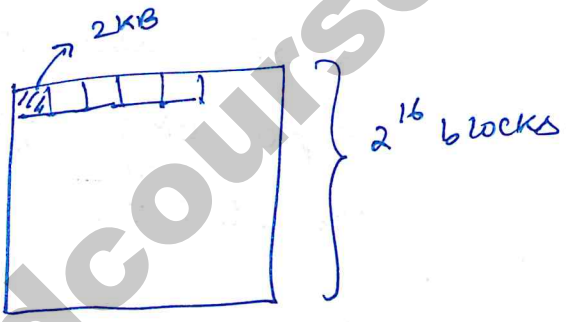
To be more precise, here blocks are data blocks, which is on the partition or the hard disk itself.

(Q) DBA : 16 bits

DBS : 2 KB

Max-file size possible on the disk = ?

DBS = 2^{16}



The maximum disk space that we can address using DBA where each block occupies 2KB is

$= 2^{16} \times 2KB$

$= 2^{16} \times 2^{11} B$

$= 2^{27} B$

$= 128$ MB

Now, maximum file size possible on the disk = maximum disk space (size)

= 128 MB

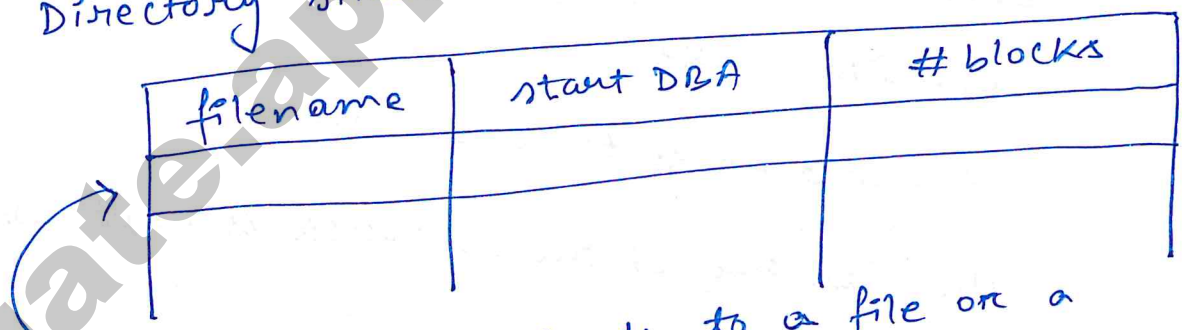
Allocation strategies / methods:

1. Contiguous allocation:

In main memory, we have seen contiguous and non contiguous allocation of frames. Here we see contiguous allocation of blocks.

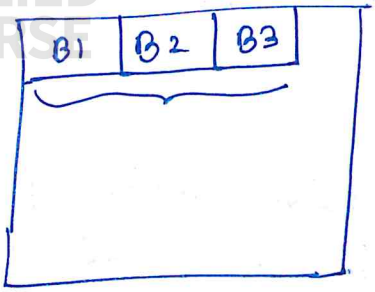
Let's assume we are creating a file named file1.
And let's assume we require 3 blocks to store the data in the file.
We need to have a directory structure.
A directory structure stores the information the information about all the files and all the subdirectories within the directory.

Directory structure:



Each row corresponds to a file or a subdirectory.

Since file1 requires 3 blocks, so amongst all the data blocks, we will look for 3 contiguous blocks which are next to each other.



filename	start DBA	#blocks
file 1	1	3

There are problems that will arise here:

(a) Internal fragmentation:

Here internal fragmentation occurs with respect to blocks.

E.g.,

Let, file1 has size 5KB

Let, DBS = 2KB

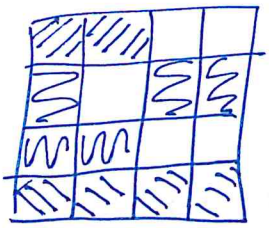
We need 3 blocks to store 5KB

The last half of the 3rd block is empty since 3 blocks has a maximum capacity of storing 6 KB.

We cannot assign the 1 KB to any other file because the block has been completely assigned to file 1. Therefore, we have internal fragmentation.

The concepts that we have learnt in main memory also translates to problem and issues in disk.

(b) External fragmentation:



Data Blocks

Imagine we need 3 blocks for a file1.
We have 5 free blocks but we can't assign 3 contiguous blocks.
Therefore contiguous allocation of blocks also suffer from external fragmentation.

(c) Increasing the file size:

Suppose we are writing new data into a file and let's say we need 3 more blocks. But we don't have 3 contiguous blocks available. So, increasing the file size doesnot work out.

Note: In reality, we donot use contiguous allocation. Because in reality, we want to keep writing data into files and it should work.

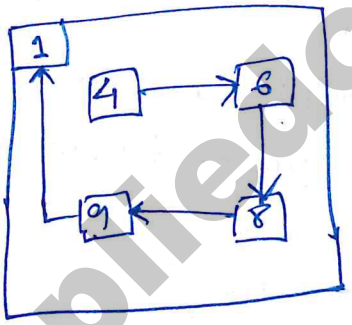
① Types of access:

If we have contiguous allocation, we can both sequentially and randomly access a file. We can seek to any location in a file, just go there and access it.

2. Linked allocation:

<filename, startDBA, endDBA>

<file1, 4, 1>



This is similar to linked list, at the end of each block, some space will be kept to store the next block address. The linked list is stored on the main memory, while this is on the disk.

4 → 6 → 8 → 9 → 1

Because it also stores the endDBA, we know that we have reached the end and we don't have to use the

last few bits and interpret them as address.

(a) Internal fragmentation :

Let, 1 Block = 2KB + pointer space.

file 1 size = 9KB

∴ We need 5 blocks.

The first 4 blocks are completely full, only the 5th block is half full.

∴ Internal fragmentation can occur.

(b) External fragmentation :

Since we can point to any block using the pointer, we don't have external fragmentation in Linked allocation.

(c) File size Increase :

Yes, it can be done. We just add one more block at the end and update the last pointer. We then update the end DBA number in the directory structure.

(d) Type of access :

We can access only sequentially. We can't do random access to any block because we use pointers to traverse to the next block.

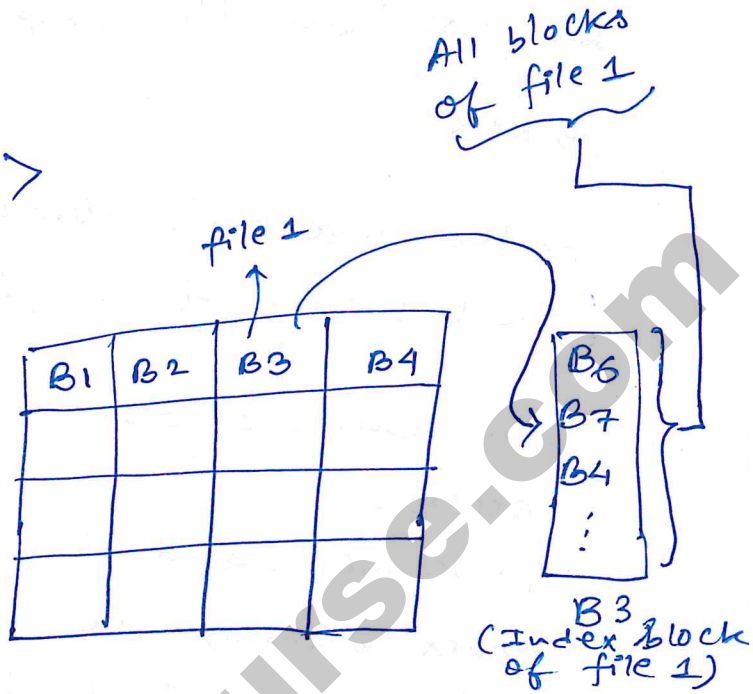
Having access to endDBA, helps us access the pointer of the last block and add new block and update the link to the newly added last block. Incase we didn't have access to endDBA, to increase the size of a file, we would have to sequentially traverse through all the blocks (one by one) until we get a null entry in the next block link.

Thus, endDBA increases the efficiency of file size increase. It also speeds up file size increase.

Imagine if one of the block gets corrupted, then the pointer information also gets corrupted/modified/null, then the link is also lost. Then all the data following that link is lost. And all these data are wasted in the main memory. Therefore by storing the endDBA, we can also check for the file consistency if all the pointers are perfectly valid or not. Because if we can start our traversal from the first block and reach the last block, then we know that these

Blocks are not corrupted. There is wastage of space due to usage of pointers. But it is negligible.

③ Indexed Allocation



This is a very popular strategy. Modified versions of indexed allocation is what is used in windows, Linux etc.

ⓐ Internal fragmentation : Yes

DBS = 2KB
File size = 9KB

∴ We get 5 blocks. Last block will have 1KB internal fragmentation.

ⓑ External fragmentation : cannot happen

Because if there is any empty block, we can assign to any file by just changing the file's index block.

(c) File size increase — very easy to perform this. we just add the empty block at the end of the index block.

Index block keeps track of all the blocks that belong to a file.

We can do random and sequential access.
First we read the index block, in the index entries, we can directly jump to the required block.

(d) Index-block (disadvantage):
Suppose the index block is corrupted, all the information is lost.
we are using additional space to hold the block itself to store the information about all the blocks that correspond to a file.
The index block is now tied to file 1.
we can't use index block as data blocks anymore.

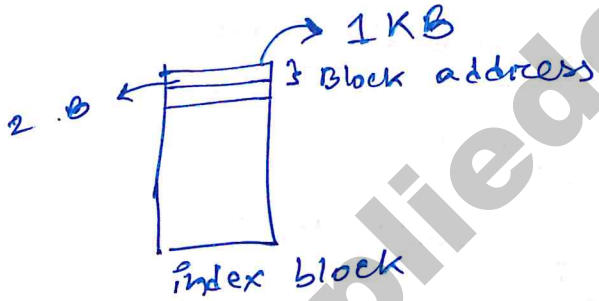
In pratise, we end up using indexed allocation or modified forms of indexed allocation.

(Q) DBA : 16 bits ; Indexed allocation;

DBS : 1KB ;

Max file size possible ?

Index block is a data block that is specifically allocated to one file to store the indexes of all the blocks that is there.



How many block addresses can be stored here?

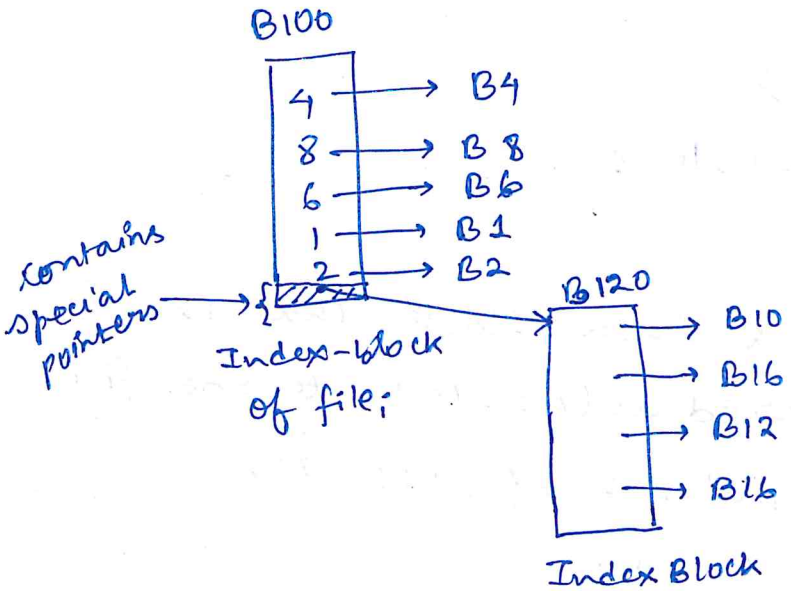
$\frac{1KB}{2B} = \frac{2^{10}}{2} = 2^9 = 512$ data block addresses

The file has 1 index block associated with it.

∴ we can have 512 blocks, each block is 1KB, ie,, the maximum file size is 512KB.

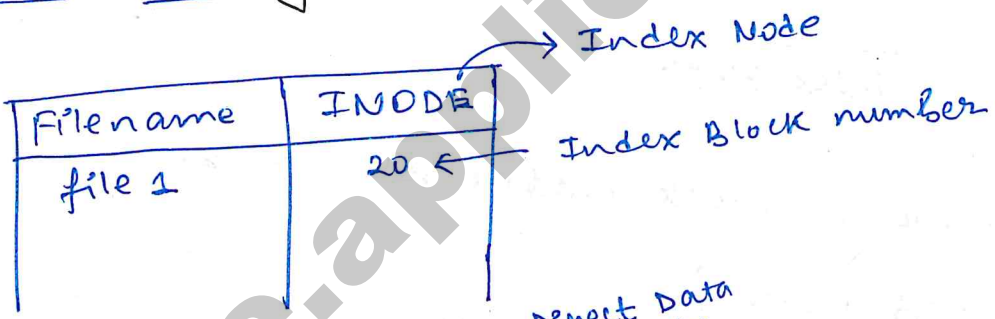
(4) Multi-Level index allocation → { Unix, Linux, Windows, Android }.

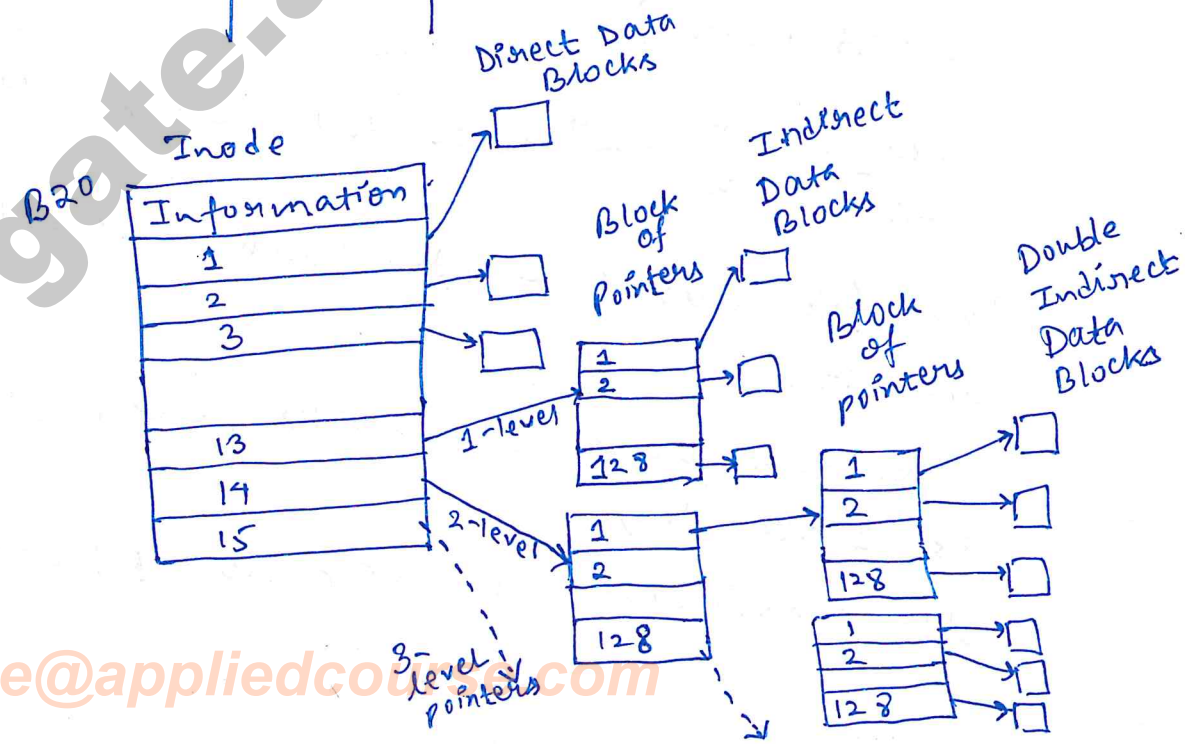
Let file; have the index block at Block 100.



Unix and DOS Implementations

Unix directory-structure:





The inode consists of some general information.

The structure of the Unix inode itself has been evolving over time.

The inode structure provided here is one of the most popular used inode structures.

In B20, the few bytes are allocated for all the attributes, like extension, type of the file, size of the file, permissions of the file, etc.

Then B20 has 12 block pointers. Each of these point to data blocks (called Direct Data Blocks). That is, B20 stores 12 Direct Block addresses.

The 13th pointer in B20 points to block of pointers, which point to 128 Indirect Data blocks.

The 14th pointer in block B20 points to block of pointers, where each of 128 pointers points to block of pointers. And, each pointers in of those 128 blocks of pointers, point to Double Indirect Data Blocks.

∴ 12 direct block addresses
One 1-level indirect block pointer
One 2-level indirect block pointer
One 3-level indirect block pointer
15 pointers in total.

If the 1st 12 pointers can satisfy the requirement of all the data that we want to store in the file, then the remaining three pointers will simply point to Null.

If the 1st 12 direct pointers are not sufficient, then we go for one level indirect pointer.

After all the one level indirect pointers are also filled, then we go for two level indirect pointer. Likewise, if needed, we go for the 3-level indirect pointer.

Current Unix system use the 15 address inode structure. The older unix systems used other types of inode structures.

Older unix inode

General attribute

10-direct-DBAs

1-single-indirect-DBA (1-level)

1-double-indirect-DBA (2-level)

12 address unix Inode

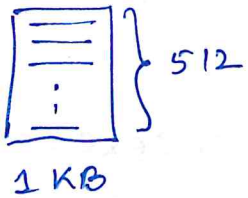
Q) Using the above older unix inode structure,

DBA = 16 bits = 2B

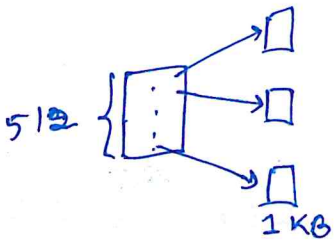
DBS = 1 KB

Note: Blocks are on the disk
pages are in the virtual address space
Frames are in the physical address space.

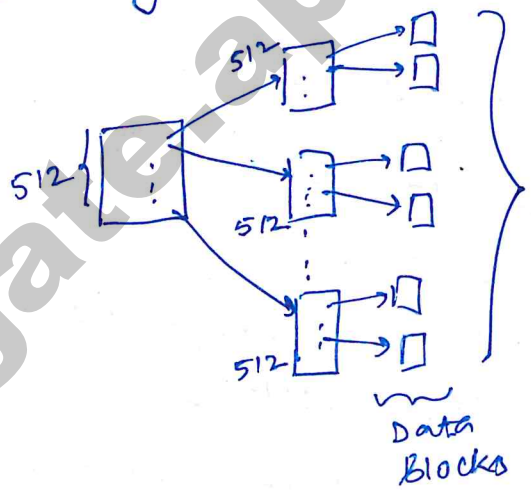
∴ # addresses per block = $\frac{2^{10}}{2}$ = 512 address can be stored per block.



In 10 direct DBAs, we can store 10 × 1 KB
= 10 KB



In 1 single indirect-DBA, there are 512 addresses pointing to 512 blocks.



∴ Total space we can address using 512 × 512 pointers is 512 × 512 × 1 KB = $2^{18} \times 2^{10}$.
= 256 MB

∴ Total addressable space

= 256MB + 512KB + 10KB

≅ 256.522 MB

∴ In the older Inode, maximum possible size of a file is 256.522 MB

Extended - indexed - allocation

DBA = 16 bits

DBS = 1KB

Max. file size on disk = ?

vs

Max. file size using inode = 256.522MB

With 16 bits, the total number of blocks = 2^{16}

Bytes / space = $2^{16} \times 1KB = 64MB$

∴ Maximum file size on the disk = maximum disk space.

= 64MB (physically we won't have more than 64MB)

This is called extended indexed allocation where-in the actual file size that we can have using inode is more than the actual number of blocks and actual storage that we have.

For the given hard disk, using inode of 12 addresses is an overkill, because we can store anything more than 64 MB.

Inode in its full potential allow us to use upto 256 · 22 MB but we can't really use it in this given scenario.

DOS directory structure:

General attrib: name, type, size, timestamp, ...

Note: DOS = Disk operating system. It is a precursor to windows 3.1. It is a command line operating system. It is not a GUI (graphical user interface) Based O.S. It is an old O.S. It's important to understand how an old O.S works.

The directory structure only stores the 1st DBA. Say the 1st DBA is 101. It literally stores the block address of the 1st block.

Let the entry in the table be:

<file1, ..., 101>

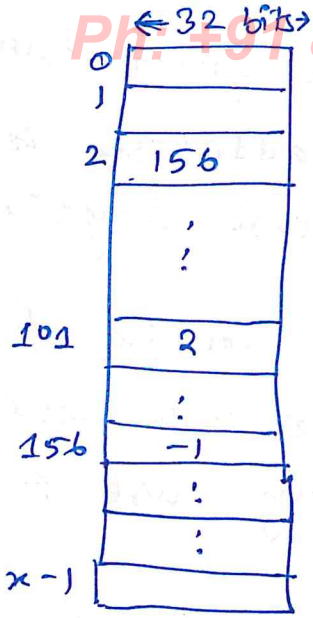
DOS along with the directory structure table also has a File Allocation Table (FAT).

The 1st data block address given is 101. Therefore the file starts at 101.

FAT is stored as an array internally.

At location 101, value 2 is available.

It means from 101 block there is a link to block 2.



FAT32 Table.

x = # blocks in a partition.

This method of storing pointers is called Tabular Linked allocation. Instead of storing the physical links, at the 101 entry in the FAT, the next location is 2.

sequential access } disadvantage.

Note: There is some form of random access possible.

We will keep a copy of the FAT in the RAM, so that we can look it up very quickly. We don't need to fetch every block, rather we can just traverse through the FAT to finally fetch the required block.

Note:
In case of simple linked allocation, we had to load the entire block, read the entire block, go to the end of the block to get the pointer to the next block.

Reading the whole block to memory takes time. The moment we know that the file starts at 101, we just use the FAT to randomly jump to any location that we want.

FAT (32) ← No. of bits in the DBA / the size of each DBA entry.
FAT (16)
FAT (8)

Disk Free-space Management

Disk free space management

On any disk there is some amount of free space at any point. How do we keep track of the free space? How do we keep track of all the blocks that are currently not allocated to some file or the other?

Only when the disk management system or the file system itself knows what blocks are free it can allocate these blocks for new files. What data structures / methods are used to keep track or manage the free space on a disk.

E.g.,

Disk size = 20 MB

DBA = 16 b = 2B

DBS = 1 KB

#blocks = 2^{16}

Maximum possible disk size = $2^{16} \times 1$ KB

$= 2^{16} \times 2^{10}$ B

$= 64$ MB

But on the disk with 20MB size and disk block size as 1KB, # of blocks = $\frac{20\text{ MB}}{1\text{KB}}$

$= 20 \cdot 2^{10}$

$= 20$K Blocks.

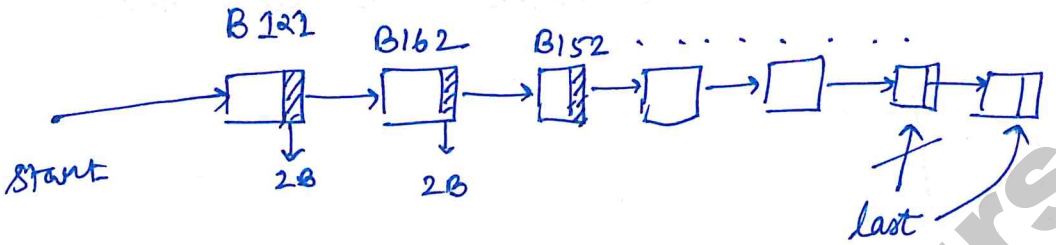
Initially all the 20K Blocks are empty.

Now we have to keep track of which all blocks are empty.

How to keep track of free-blocks.

① Free linked list approach: (*Not very popular)

Simplest of the strategies. Imagine all the blocks which are not currently allocated to any file.



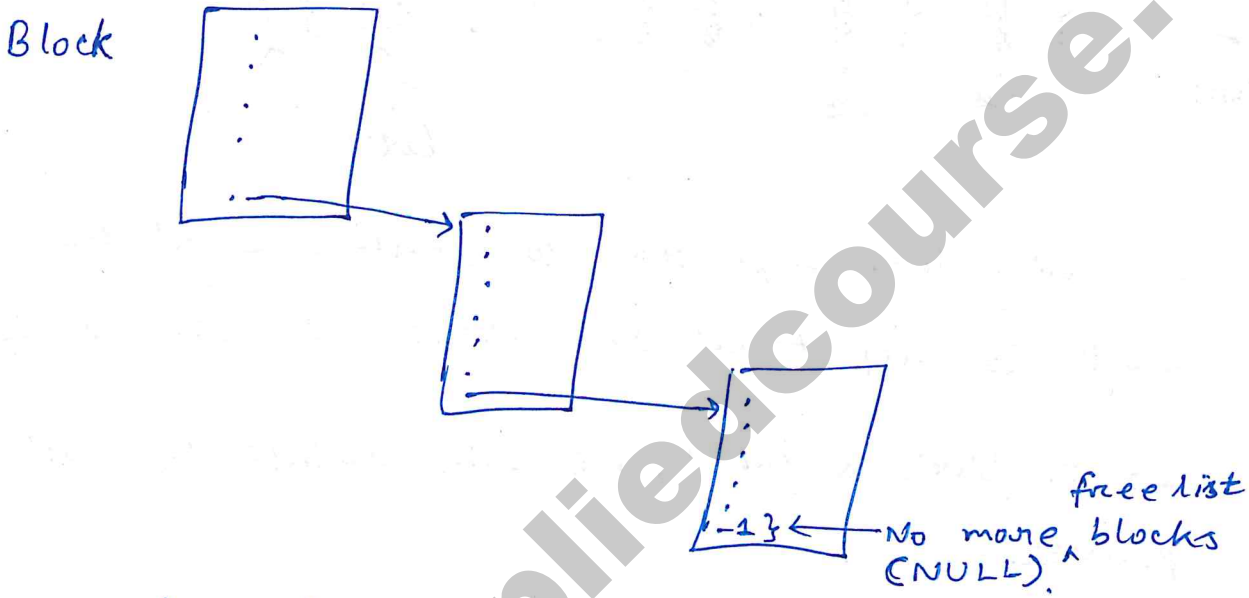
We can store a pointer to start and at the last to keep track of first and last blocks.

If a block is freed, we will update the last pointer.

The overhead in the free linked list approach is we have to go for each of free blocks and add the address of the next free block. And we can't allocate whichever block that we want. We have to start allocating only in the sequence. We can't directly jump and allocate a block which is available to a file.

② Free list : (* popular approach)

In this approach we will create one block called free list block. In this block we will store addresses of all blocks. If all the blocks are full then, the last entry will point to another block and this continues.

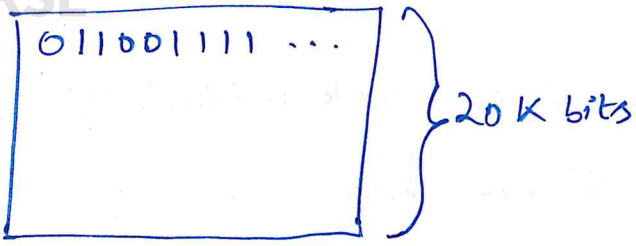


The free list blocks are special purpose blocks as they are only meant to store list of addresses of blocks that are free.

③ Bitmap : (*popular approach)

Let's assume at the very beginning we have 20K free blocks.

We create a bit vector and we write 1 bit per block. If bit value is 0, means the block is free. If the bit value is 1, that means the block is in use by some file.



The whole bit length is going to be 20K bits.

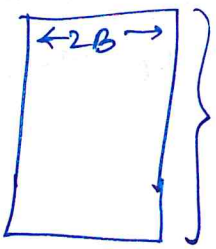
∴ Total storage required = 20K bits

# of bits required is not dependant on the number of free blocks. Whether the block is in use or free, we have to mention it here (in the bit vector). The bit vector is a fixed size data structure.

(Q). To store data about free-blocks using free-list method, # blocks needed = ?

Given, DBS = 1KB
DBA = 16b
#Blocks = 20K (from previous example)

The block size is 1KB and the address is 16 bits = 2B



$\frac{2^{10}}{2}$ = 512 entries = 2^9 entries

Max. # of free blocks = 20K

To store these 20K free block addresses, how many blocks do we need?

We need $\frac{20K}{2^9} = 20 \times 2 = 40$ blocks

It is an approximation as we didnot store pointer information.

(Q) Same as above.

blocks needed using bitmap-method=?

20K blocks $\Rightarrow$ 20K bits

$\because$ 1 KB $= 2^{10}$ B $= 2^{13}$ bits

We have 20K blocks, for each block there is a bit, therefore we need 20K bits.
And each block can hold 1KB. And 1KB $= 2^{10}$B $= 2^{13}$ bits

$\therefore$ 2^{13} bits is the capacity of 1 block.

And we need 20K bits,

$\therefore$ $\frac{20 \times 2^{10}}{2^{13}}$ = # of blocks we need using bitmap approach.

$= \frac{20}{2^3}$

$= \lceil \frac{20}{8} \rceil$

$= 3$ blocks [as 2 blocks won't suffice, 2 blocks will only give 16 K bits]

Bitmap approach therefore needs only 3 blocks in comparison to that of free list approach (where we needed 40 blocks).

Also doing bitwise operations are very efficient.

(Q) # blocks = B

free blocks = F

DBA = D bits

DBs = S Bytes

(a) What is the actual disk size

B blocks * S Bytes / block

(b) What is the maximum possible disk size

$\underbrace{2^D}_{\text{blocks}}$ * S Bytes / block

[Maximum possible number of blocks = 2^D]

Note:

$B \leq 2^D$

The number of blocks in a system has to be less than or equal to 2^D.

(c) Under what condition does free list use less space than bitmap.

In the bitmap method, we have B blocks.

∴ We need B bits.

In the free list method,
because there are F free blocks,
for each of the F free blocks, the address is D bits.

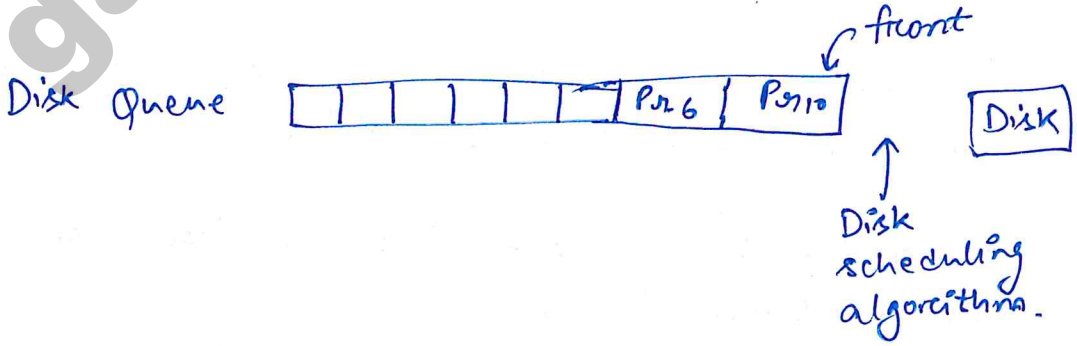
∴ # of bits needed (in total) = F * D bits

∴ $\boxed{F*D < B}$ whenever this condition is true, then the free list approach uses less space than the bitmap approach.

Disk / Device Scheduling

Just like the CPU scheduling algorithm, where CPU is a resource,
the devices / disks also are resources of the system. Therefore the processes requesting for I/O also needs to be scheduled.

For any disk / device, there is a queue.



The objective of the CPU scheduler is to improve the throughput of the CPU.

In ready queues, we put the PCBs of the processes.

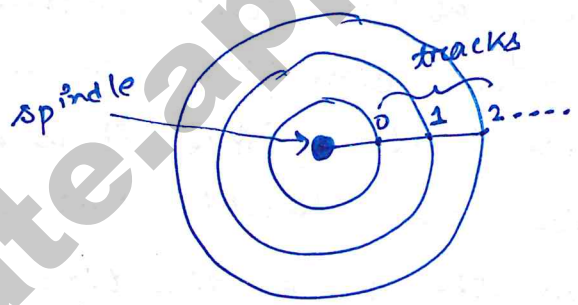
Similarly in the Disk Queue, we put the PCBs of the processes.

The objective of the disk scheduling algorithm is to minimize the seek time.

Disk / Device scheduling algorithms:

I/O time = Seek Time + Latency Time + Transfer Time

- Seek Time: major component ∴ we try to minimize it to reduce the overall I/O time.
- Latency Time + Transfer Time: Depends on the rotational speed of the disk.



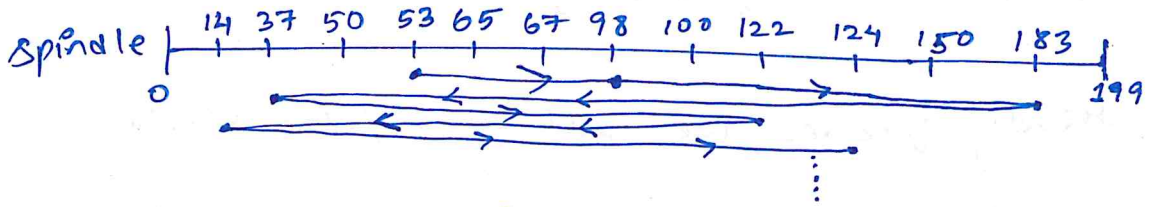
For one disk, we have multiple tracks like this

From the perspective of multiple disks, we have each track corresponding to a cylinder.

Seek time is the time it takes to go from one track to another. This is physical movement and it takes much longer than just spinning the disk.

E.g.

tracks = 200



Track sequence : 53, 98, 183, 37, 122, 14, 124, 65, 67

→ time axis →

① FIFO : (FCFS)

$45 + (183-98) + (183-37) + (122-37) + \ldots\ldots$

$= 640$ tracks

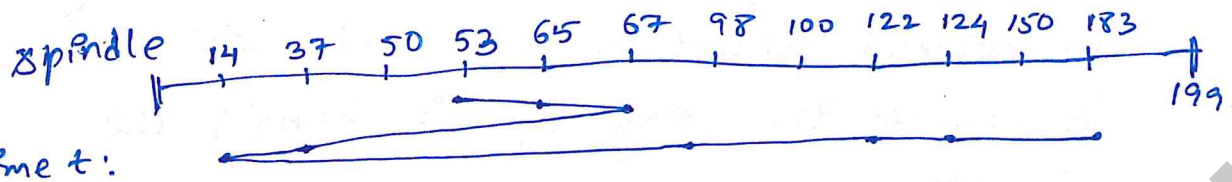
If the movement over tracks reduces, that implies that the total time spent as seek time also reduces.

FCFS is one of the least efficient algorithm for disk scheduling as we don't continue to move in the same direction. We keep jumping back and forth. Since it is a mechanical device we should rather keep moving it in the same direction.

Advantage: There is no starvation
No long wait times, as we are going to service the requests in the order they arrive.
But the overall efficiency of the disk scheduling algorithm could be terrible.

② SSTF:

(Shortest seek time first)



@ time t:

Track-seq: 53, 98, 183, 37, 122, 14, 124, 65, 67.

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

$= 236$

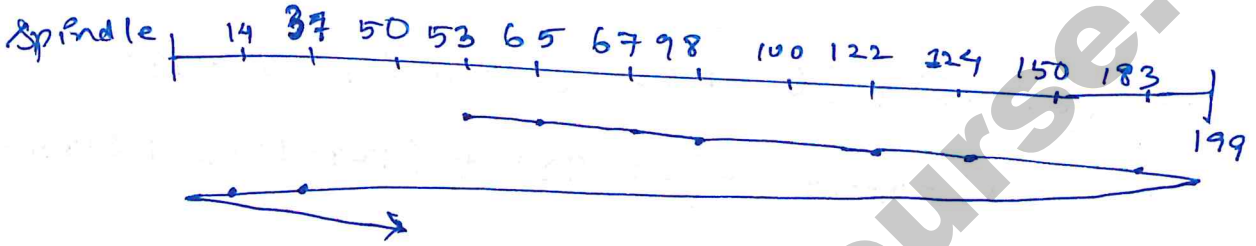
Disadvantage : starvation.

If there are lots of new incoming requests where the track sequence requests are very close to one another, therefore the read/write head keeps servicing those requests and continues to ignore older requests that are far away from the current location of the read-write head.

Also there may be variance in performance from one request to the other. We want to minimize the variance also. SSTF doesnot utilize the direction sense.

③ SCAN (elevator):

If an elevator or a lift is moving in upward direction, it continues to move in upward direction and only when it reaches the very top it changes the direction.



@t:

Track sequence: 53, 98, 183, 37, 122, 14, 124, 65, 67

This algorithm can also suffer from very long wait time.

After completing 183 and 14 we still have to go till the end and only the read write head changes the direction. That is why it is inefficient.

Say, we got a request as track 14, but currently read write head is servicing at 183 and we keep getting new requests from the process of tracks between 183 and 199, then servicing of track 14 gets delayed by a long duration (finite time). Thus, wait time increases for track 14 to be serviced.

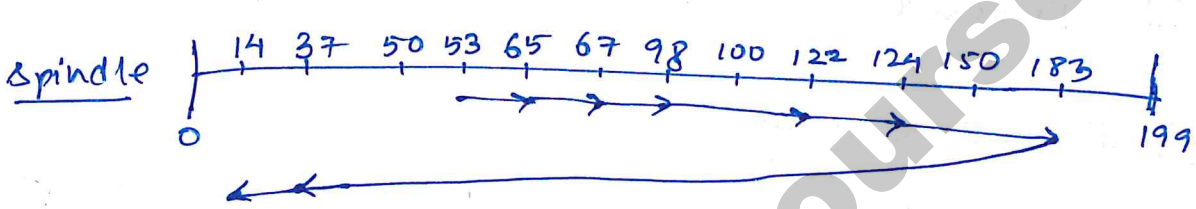
The case of starvation is also possible, if the keep getting track requests like

190, 190, 190, 190, 190, 191, ...

then track request of 14 will be starved.

④ A small modification to the SCAN algorithm.

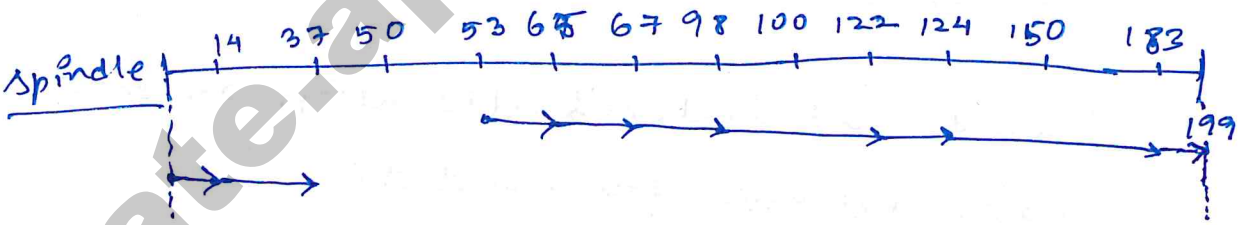
LOOK:

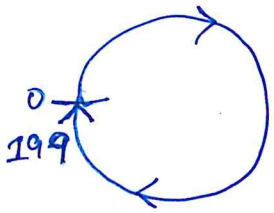


@Time t:

Track-Seq: 53, 98, 183, 37, 122, 14, 124, 65, 67.

⑤ CSAN:





After servicing 183, it goes upto 199, then it directly jumps to 0 and continues in the same direction like a circle.

More Solved problems :

Q.n) In a file allocation system, which of the following allocation scheme(s) can be used if no external fragmentation is allowed?

I. Contiguous

II. Linked

III. Indexed

(A) I and III only

(B) II only

(C) III only

(D) II and III only

Soln. Contiguous allocation has external fragmentation.

Linked and Indexed allocation do not have external fragmentation.

∴ (D) is the answer.

Qn) The index node (inode) of a Unix-like file system has 12 direct, one single indirect and one double indirect pointer. The disk block size is 4KB and the disk block addresses 32-bit long. The maximum possible file size is rounded off to 1 decimal place) ____4____ GB.

Soln.

DBS = 4KB

DBA = 32b = 4B

In one disk block, # addresses:

$$= \frac{4KB}{4B} = 1K$$

∴ # address/block = 2^{10}

In 12 direct block pointers, we can have

$12 \times 4\,KB$

In 1 single indirect block pointer, we can have

$1 \times 2^{10} \times 4\,KB$

In 1 double indirect block pointers, we can have

$1 \times 2^{10} \times 2^{10} \times 4\,KB$

∴ Total size (addressable) = $48\,KB + 2^{10} \cdot 4KB + 2^{20} \cdot 4KB$

$= 48KB + 4MB + 4GB = 4.0...$ GB

$\cong 4GB$

Qn) Suppose a disk has 201 cylinders, numbered from 0 to 200. At some time the disk arm is at cylinder 100, and there is a queue of disk access requests for cylinders 30, 85, 90, 100, 105, 110, 135 and 145. If shortest seek time first (SSTF) is being used for scheduling the disk access, the request for cylinder 90 is serviced after servicing ___3___ number of requests.

Soln.

R/W head
↓
100, 105, 110, 90
3 requests

Qn) Suppose the following disk request sequence (track numbers) for a disk with 100 tracks is given: 45, 20, 90, 10, 50, 60, 80, 25, 70.

Assume that the initial position of the R/W head is on track 50. The additional distance that will be traversed by the R/W head when the Shortest Seek Time First (SSTF) algorithm is used compared to the SCAN (Elevator) algorithm (assuming that SCAN algorithm moves towards 100 when it starts execution) is ___10___ tracks.

track# ⟶ track#

1 100

SSTF: $50 \xrightarrow{5} 45 \xrightarrow{15} 60 \xrightarrow{10} 70 \xrightarrow{10} 80 \xrightarrow{10} 90 \xrightarrow{65} 25 \xrightarrow{5} 20 \xrightarrow{10} 10$

$= x$

SCAN: $50 \xrightarrow{10} 60 \xrightarrow{10} 70 \xrightarrow{10} 80 \xrightarrow{10} 90 \xrightarrow{10} 100 \xrightarrow{55} 45 \xrightarrow{20} 25 \xrightarrow{5} 20 \xrightarrow{10} 10$

$= y$

$|x - y| = 10$ tracks ← Ans

Qn Consider a disk queue with requests for I/O to blocks on cylinders 47, 38, 121, 191, 87, 11, 92, 10. The C-look scheduling algorithm is used. The head is initially at cylinder number 63, moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is 346.

Soln. CLOOK is a circular variation of LOOK.

$63 \rightarrow 87 \rightarrow 92 \rightarrow 121 \rightarrow 191 \quad 199$

$0 \rightarrow 10 \rightarrow 11 \rightarrow 38 \rightarrow 47$

$$
\begin{array}{r}
191-63 \\
+\ 191-10 \\
+\ 47-10 \\
\hline
346
\end{array}
$$

CLOOK: keep going in one direction till the last request is there, but we don't have to go to the last cylinder/track. From the last request directly jump to the earliest track request and keep going in the same direction.

Qn) A FAT (file allocation table) based file system is being used and the total overhead of each entry in the FAT is 4 Bytes. in Size is 10^3 Bytes, the maximum size of a file that can be stored on this disk in units of 10^6 bytes is 99.6 MB

Note:

$10^3 \approx 2^{10}$

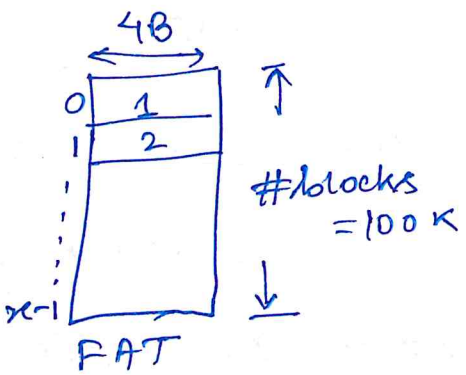
$10^6 \approx 2^{20}$

$10^9 \approx 2^{30}$

Disk capacity = 100MB

DBS = 1 KB

$\#\text{blocks} = \frac{100M}{1K} = 100K$



Space required for FAT = 100K * 4B
= 400 KB.

Among the 100 MB disk space, we will have to keep 400KB aside for FAT.

400 KB ≈ 0.4 MB

∴ The maximum size of a file that can be stored on the disk in units of 10^6 Bytes is = 100 MB - 0.4 MB = 99.6 MB.

Qn.) Consider a disk pack with a seek time of 4ms and rotational speed of 10000 rotations per minute (RPM). It has 600 sectors per track and each sector can store 512 bytes of data. Consider a file stored in the disk. The file contains 2000 sectors. Assume that every sector access necessitates a seek, and the average rotational latency for accessing each sector is half of the time for one complete rotation. The total time (in milliseconds) needed to read the entire file is 14020 ms.

Soln

Seek time = 4ms

10K rpm

10K revolutions —— 60 seconds

1 revolution —— $\frac{60}{10000}$ s

= 6 ms

R = 6 ms

$LT = \frac{1}{2}R = 3$ ms

The Total time (in ms) = ST + LT + TT

= $2000 \times (4 + 3 + 0.01)$ = 14020 ms ← Answer

6 ms → 600 sectors = 1 track

$\frac{6}{600}$ → 1 track

$\frac{1}{100}$ ms = 0.01 ms = Transfer time

Qn) Consider a typical disk that rotates at 15000 rotations per minute (RPM) and has a transfer rate of 50×10^6 bytes/sec. If the average seek time of the disk is twice the average rotational delay and the controller's transfer time is 10 times the disk transfer time, the average time (in milliseconds) to read or write a 512 Byte sector of the disk is 6.1 ms

Soln).

15K R ⟶ 60 sec

$R \longrightarrow \frac{60}{15K} = 4ms$

Rotation delay or Latency time = 2ms
(RD or LT)

ST = 4ms

Transfer rate = 50×10^6 bytes / seconds.

For 512 Bytes, we need time = $\frac{512}{50 \times 10^6}$

∴ Disk transfer time = 0.0102 ms

Given controller transfer time = 10 x Disk transfer time

= 0.102 ms

∴ Total Time = 4ms + 2ms + 0.102 ms

= 6.102 ms

≈ 6.1 ms

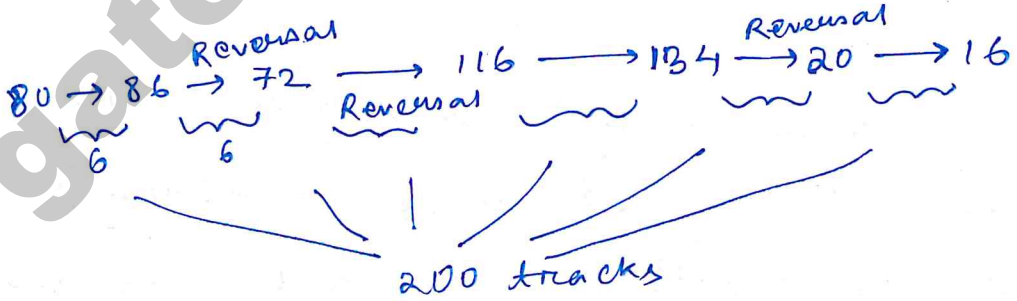
Qsn.) Consider a storage disk with 4 platters (numbered as 0, 1, 2 and 3), 200 cylinders (numbered as 0, 1, ..., 199) and 256 sectors per track (numbered as 0, 1, ..., 255). The following 6 disk requests of the form [sector number, cylinder number, platter number] are received

by the disk controller at the same time:

[120, 72, 2], [180, 134, 1], [60, 20, 0], [212, 86, 3], [56, 116, 2], [118, 16, 1]

Currently head is positioned at sector number 100 of cylinder 80, and is moving towards higher cylinder numbers. The average power dissipation in moving the head over 100 cylinders is 20 milliwatts and for reversing the direction of the head movement once is 15 milliwatts. Power dissipation associated with rotational latency and switching of head between different platters is negligible.

The total power consumption in milliwatts to satisfy all of the above disk requests using the shortest seek Time First disk scheduling algorithm is 85 milliwatts



100 — 20 m watts
200 — 40 m Watts

reversals — 15 m watt
3 reversals = 45 m watts

∴ Total power consumption = 40 + 45 = 85 milli-watts.

Qn) The following are some events that occurs after a device controller issues an interrupt while process L is under execution.

(P) The processor pushes the process status of L onto the control stack

(Q) The processor finishes the execution of the current instruction

(R) The processor executes the interrupt service routine.

(S) The processor pops the processor status of L from the control stack.

(T) The processor loads the new PC value based on the interrupt.

Which of the following is the correct order in which the events above occur?

✓(A) QPTRS

(B) PTRSQ ✗

(C) TRPQS ✗

(D) QTPRS

Soln. It has to start with Q.

∴ Option (B) and (C) are invalid.

P comes before T as we have to move the currently running process back

to the stack. Then the new process's PC will loaded after that.