PAGE NO. :
DATE : / /

Operating Systems

Ref. Books :

- Operating System Concepts by Galvin
- OS internals & design principles by William Stallings
- Modern OS by Tanenbaum
- OS : 3 Easy Pieces

Gate Syllabus :

- System Calls, process, threads, Interprocess communication
- Concurrency & Synchronization
- Deadlock.
- CPU & IO scheduling
- Memory Management & Virtual memory
- File Systems.

OS is study of software & COA deals with hardware.

OS is important to execute even a small program else we will have to manually load program in memory, mange resources, update counters & many other responsibilities. e.g. to OS.

- we will have to bookkeeping of program in the main memory locations
- we will have to keep track of the resources if in use / free.

And many more works we'll need to do.

PAGE NO.:
DATE: / /

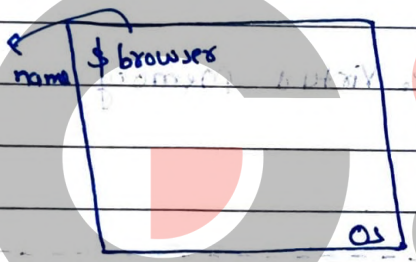
⇒ Needs of an OS:

→ Manually too hectic

→ These tasks are common, everyone needs them.

→ w/o OS not everyone will be able to use computer. Only comp. scientist will be able to do so.

first version of unix was succesfull attempt of OS development. It was developed by 2-3 people only; that too in 2-3 years. windows, Mac, linux all are UNIX inspired.



After writing file name, OS search for file in secondary storage & loads in MM.

It kills shell, runs browser, kills browser, run shell again. This all is done using fork & exec. We will soon see them.

OS provides us virtualisation in many ways. One of them is we will feel that our OS is running the programs concurrently but they don't run simultaneously OS runs 1 at a time, but swap them in nanoseconds So we get an illusion. (CPU virtualisation)

→ OS is like Govt. It performs no useful function by itself. It simply provides an environment within which other programs can do useful work

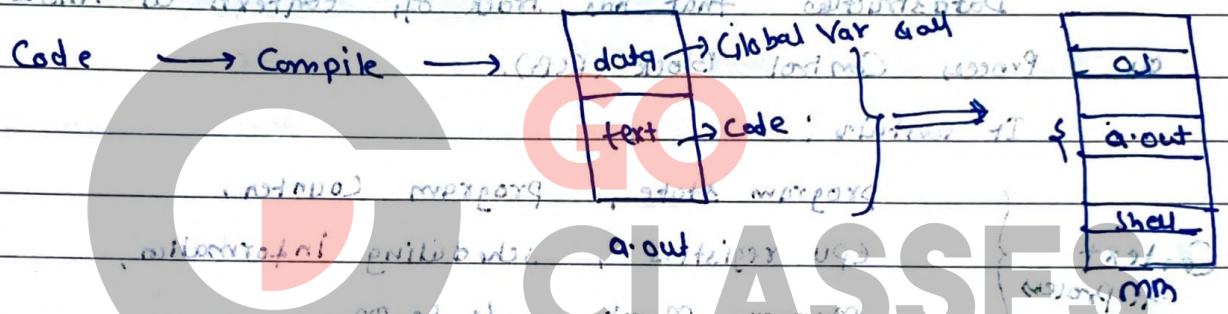
-Galvin

PAGE NO. :
DATE : / /

→ We already have the compiled code of OS in hard disk (exec. file).

* How a program is run on machine?

- User writes a program
- Compiler converts the program into executable code (byte code)
- User requests OS to run this byte code.
- OS loads the program in memory
- OS sets register properly & starts executing the code.



OS is always in mm (loaded by bootloader at start of computer), but it don't means it is always running. Some code like printf need interference of OS bcoz we need to see if user program can access monitor.

O.S. on request by user to run byte code does following two steps

#1 create space for bytecode (in mm)

#2 Load code in that space.

Now these two steps are done using fork & exec. So, we will study both of them & then come back to this

OS ≡ Complex Exercise of D.S.

PAGE NO. :
DATE : / /

Process: Informally, it is program in execution. A process competes for the resources.

Context of a process: OS has to keep track of everything (P.C, Registers, Open files, etc) of the process. And this everything is called the context (current state) of the process.

In C, we can (for OS) use struct to keep note of various details called Context of the process.

Data structure that has track of contexts is known as Process Control Block (PCB).

It contain:

Context of process { program state, program counter, CPU registers, scheduling information, Memory Mgmt, & so on.

Ex:

```
struct pcb {
    pid_t pid;
    long state;
    struct files_struct *files;
    .
    .
    .
};
```

→ After loading, it's not necessary that we'll get chance to immediately run it. It will move to something called ready queue (taught later).

PAGE NO.:
DATE: / /

Qu: After putting code in mm, shall we call it program or process?

→ It's in grey area. Some author call it process while some say it should run atleast once to be process.

Something practical now:

☆ **fork():** It's a system call used to create a new process. It creates an exact replica of the calling process.

It copies the exact context including PC so next line of both child & parent process run are same.

Ex:

```
main()
{
→ fork();    // line 8
  pf(a);
}
```

Parent.

```
main()
{
→ fork();
  pf(a);
}
```

Child

↳ Created after exec. of line 8 from parent got executed.

PAGE NO. :
DATE : / /

So fork() creates exact same copy in mm. Parent & child will start executing after the line which had fork().
fork() returns 0 in child process & process id of child (Non-zero) to parent process.

Ex: main()
{ int pid = fork(); // Now two process in mm
if (pid == 0) pf("child"); // only true in child process
else pf("Parent");
}

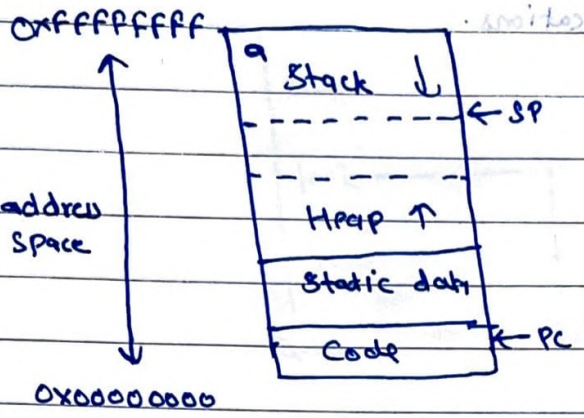
O/p can be Parent Child or Child Parent.
Value of pid in child Process is zero & in parent is non-zero.

Que: main() {
pf("A");
int pid = fork();
if (pid == 0)
pf("B");
else
pf("C");
pf("D");
}

→ A, B, C → once
D → Twice

PAGE NO.:
DATE: / /

Process looks as:




```
int main()
{
    int a=10;
}
```

This a will have some logical address & printf always prints the logical address.

Actually in mm it will have d/f address known as physical address which we do not care about. It depends upon where the process is loaded in main memory.

```
main()
{
    int a=10;
    fork();
    pf("%a");
}
```

Same address will be printed twice bcz printf prints logical address & fork creates exact replica, so logical address is same.

However their physical address will be different always bcz they are loaded at d/f locations in mm.

So, manipulating one's pointer will not effect other bcz actually they are at d/f locations, Their "logical/virtual address looks identical.

In opcodes statements also we deal with logical address.

PAGE NO. :
DATE : / /

It will not matter if it is a global variable bcz. they are diff & available at diff locations.

Gate CS or

Que: Consider the following code:

```
if(fork()==0)
{
    a=a+5;
    pf(a,&a);    //x,y
}
else
{
    a=a-5;
    pf(a,&a);    //u,v
}
```

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

⇒ v & y will always be same as logical address is printed. So v=y

diff b/w u & x will always be 10, as both are diff variables so u+10=x

So, Change in one will not reflect in other as both are diff physically.

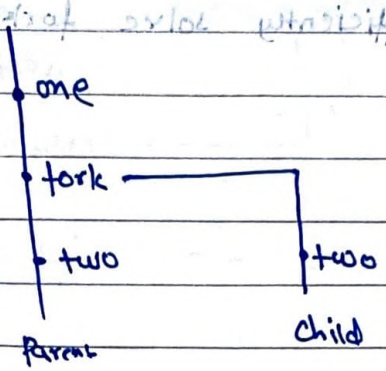
Que: Solve the following fork based question:

a) `pf("one");`
`fork();`
`pf("two");`

o/p: one
two
two

PAGE NO. :
DATE : / /

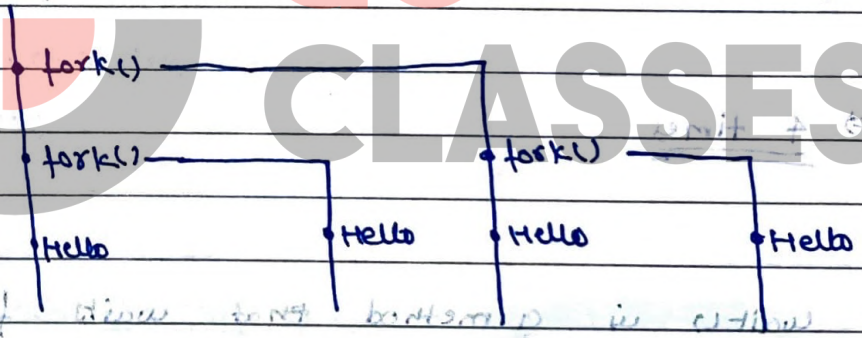
1. Process graph



(b)

```
fork();
fork();
pf("Hello");
```

⇒

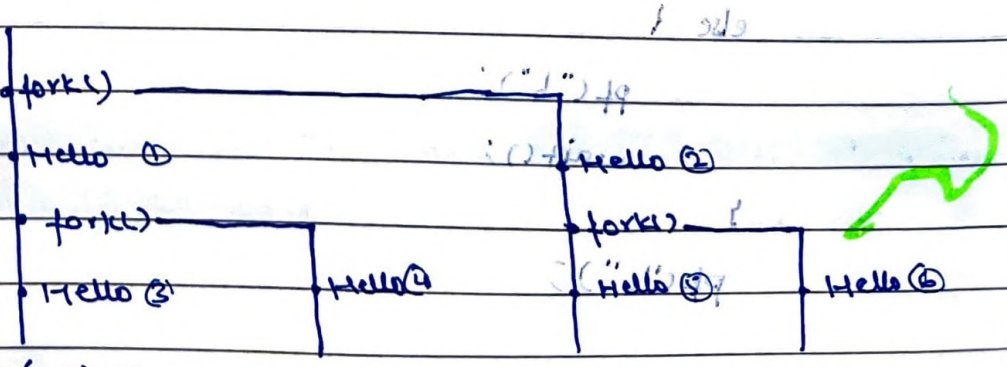


So Hello printed 4 times
(4 process are created)

(c)

```
for(int i=0; i<2; i++){
    fork()
    pf("Hello")
}
```

⇒



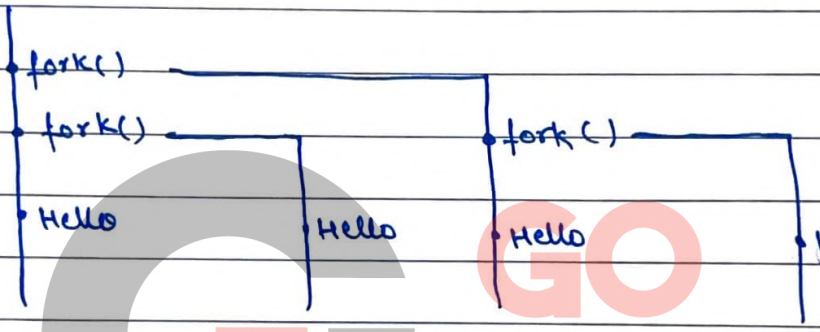
2+4 times = 6 times

PAGE NO. :
DATE : / /

So to create process graph we can write equivalent code.
Process graph is a tool to efficiently solve fork() que.

d)

```
for(int i=0; i<2; i++){
    fork();
}
pf("Hello");
```



⇒ 4 times

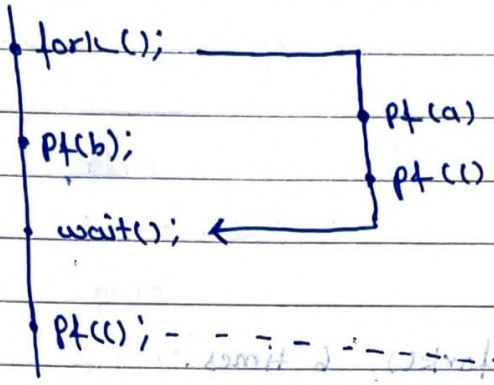
NOTE wait() is a method that waits for child to finish first.

e) write all possible patterns:

```
if(fork()==0){
    pf("a");
}
else {
    pf("b");
    wait();
}
pf("c");
```

By logical memory we give an illusion memory is for you.

PAGE NO. :
DATE : / /



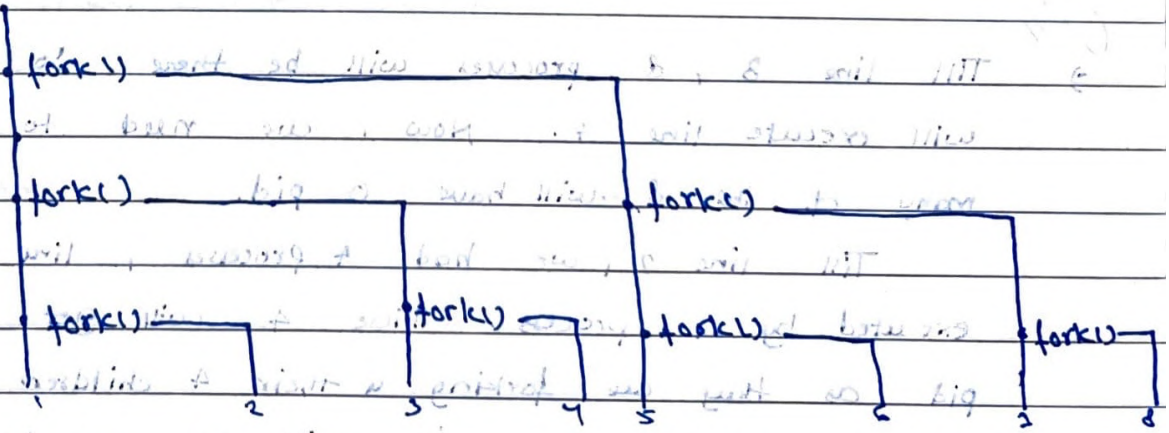
So, possible patterns will have c at last () this one.
& child's a & c will be printed ~~one~~ one after other.

bacc, acbc, abcc

↳ 3 patterns possible

Q. for(i=1; i<=3; i++)
fork();

⇒



8 processes will be there i.e. 7 new processes are created.

NOTE: n fork() will create $2^n - 1$ new processes & 2^n processes will be there in total.

PAGE NO. :
DATE : / /

g)

```
for (i=1; i<=3; i++)
{
    fork();
    fork();
}
```

→ Code is eq. to write fork() 6 times.

So # processes = $2^6 = 64$

new processes = $2^6 - 1 = 63$

h)

```
int pid = fork();
pid = fork();
pid = fork();          //line3
if (pid == 0)
    fork();
fork();                //line6
```

$\frac{4 \times 2}{12}$

a) //

⇒ Till line 3, 8 processes will be there, so 8 processes will execute line 4. Now, we need to know how many of the 8 will have 0 pid.

Till line 2, we had 4 processes, line 3 was executed by 4 process. These 4 will get a non-zero pid as they are forking & their 4 children will have pid as zero. So, if will be executed by 4 children. So, 4 will go in if & 4 will not go. The 4 children in if will come as if & 4 were there directly.

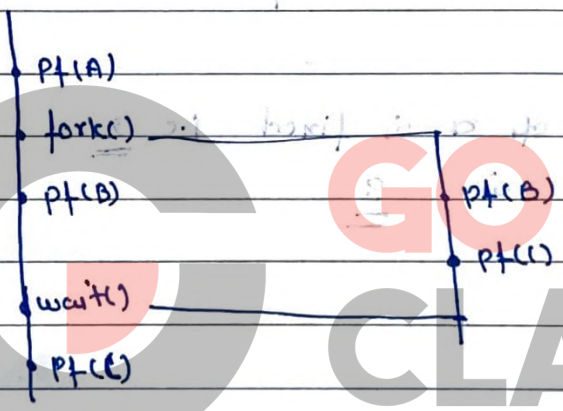
Finally 12 proces will execute line 6. At the end 24 processes will be there

PAGE NO.:
DATE: / /

i) All patterns possible:

```
pf(A)
int ch= fork();
pf(B)
if (ch)
    wait();
pf(C)
```

⇒

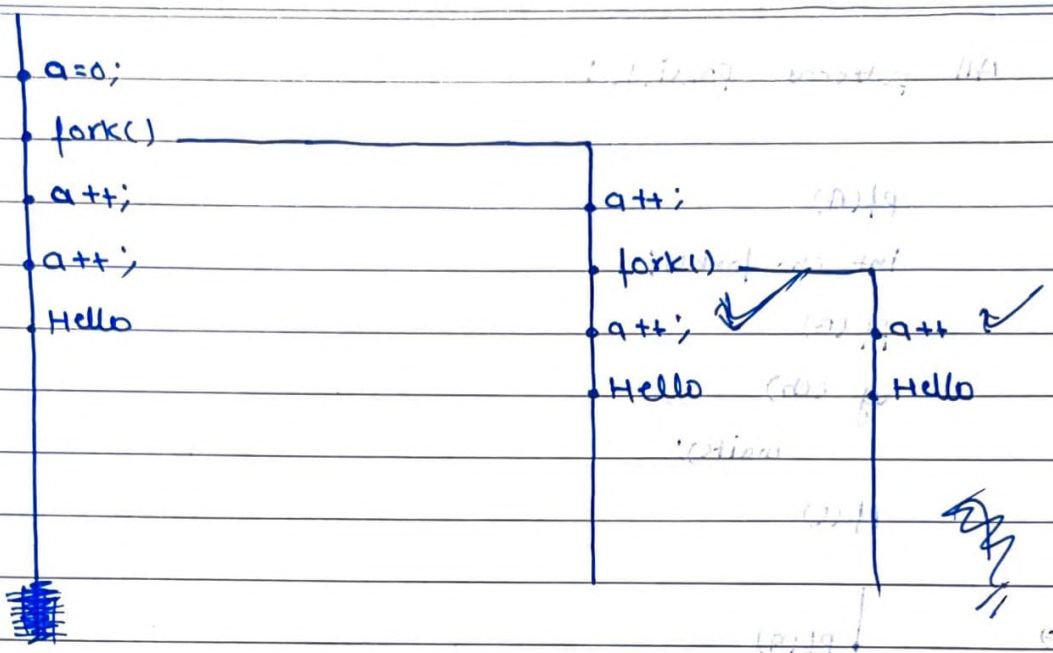


Possible Outputs: ABCBC, ABBCC

ii) # times hello printed & max. value of 'a' possible?

```
int a=0;
int rc= fork();
a++;
if (rc==0) {
    rc= fork();
    a++;
}
else { a++; }
pf("Hello");
pf("a");
```

PAGE NO. :
DATE : / /



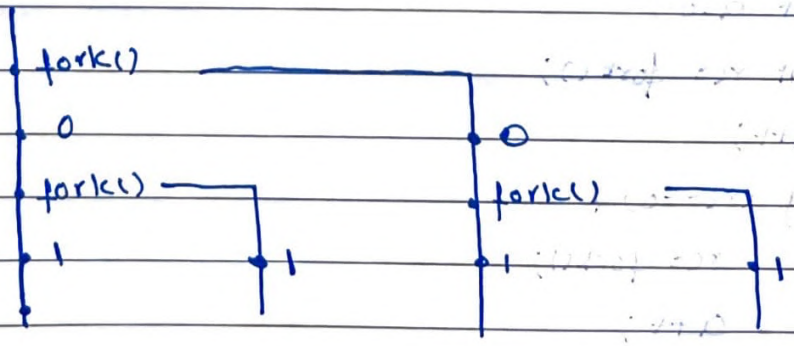
Max & Min value of a is fixed i.e 2.

times hello is 3

k) Possible patterns:

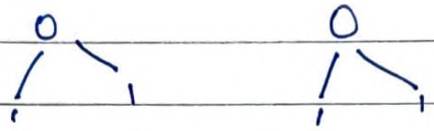
```
int i=0;
while(i<2){
        fork();
        pf(i);
        i++;
}
```

⇒



PAGE NO.:
DATE: / /

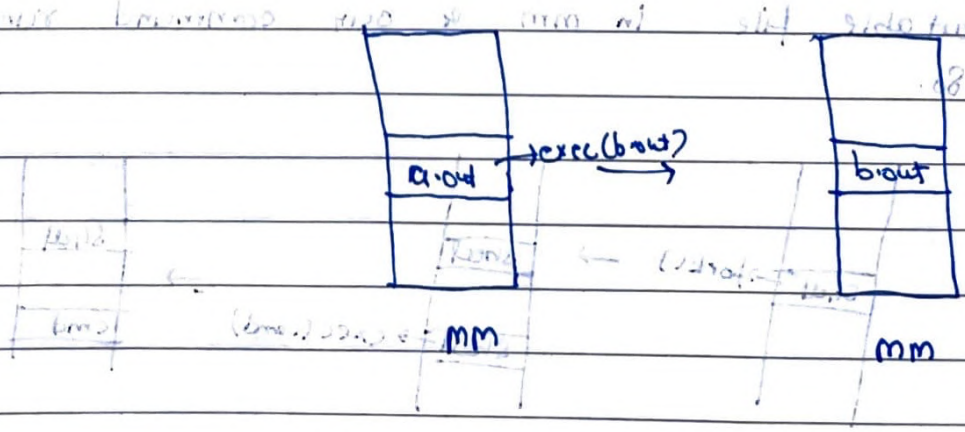
Dependency graph ii



So possible are 001111, 011011, 010111, only 3 patterns

Now in order to understand how process is created, we will go with another system call exec.

☆ exec system call: It receives one executable file name in it's parameter & when executed the exec in code, it replaces its own process data in mm with the executable file's details in mm. However, process id remains same for both prev. & the new file passed as argument. bcz it replaces itself from mm. No new space is needed to search.



a.out gets replaced with b.out after executing the instruction exec("b.out");

PAGE NO. :
DATE : / /

Que: O/P?

a.c:

```
pf(1);
exec(b.out);
pf(2);
```

b.c:

```
pf(3);
```

O/P: 1 3

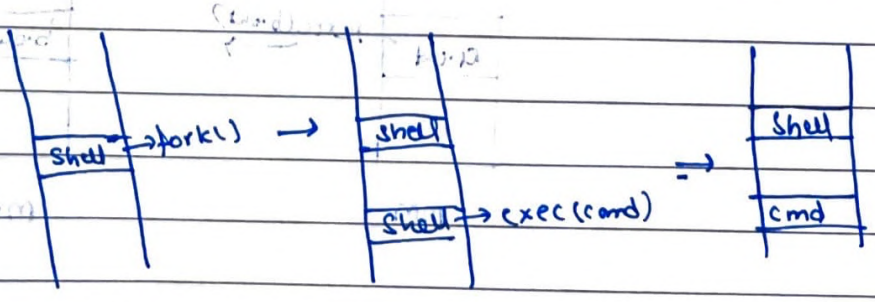
Note that 2 is not printed.

☆ Process Creation: Now we will understand how the process is created using fork & exec.

We need to create space for new space in mm & then load that process in that space of mm.

If we run a.out in terminal then a.out gets executed. Basically fork() is run by terminal which creates it's exact copy & then exec is run which replaces the executable file in mm & our command runs.

So.



This is how a process look in mm.

PAGE NO. :
DATE : / /

shell code might look like this:

```
while(true)
{
    scanf("%s", &cmd);
    pid = fork();
    if (pid==0)
        exec(cmd);
    else
        wait();
}
```

NOTE: If we write exec before fork, it will not work like earlier obviously. Shell will get killed after executing one program. As fork() will be replaced & never executed...

Ques:

```
a.c:
main(){
    int i=0;
    pf(i);
    i++;
    if(i==2)
        exit();
    exec(a.c);
}
```

Op: 0 0 0 0 ... 0 (forever)

NOTE: After exec() process ID do not change, it remains same only content gets changed, registers are reset.

PAGE NO. :
DATE : / /

★ System Calls:

→ **User Trust issue:** Most of the users are not experts in computers and do not understand how hardware works. Thus they mayn't know how to properly manage the resources.

An user (expert/not) can make incorrect operation may be intentionally or by mistake & can lead to system failure.

Solution for this is that never trust a user while dealing with such critical tasks by do not allowing user to directly access h/w.

These critical task which need OS help are called privelleged task & other are called normal tasks. (Ex: a=b+c)

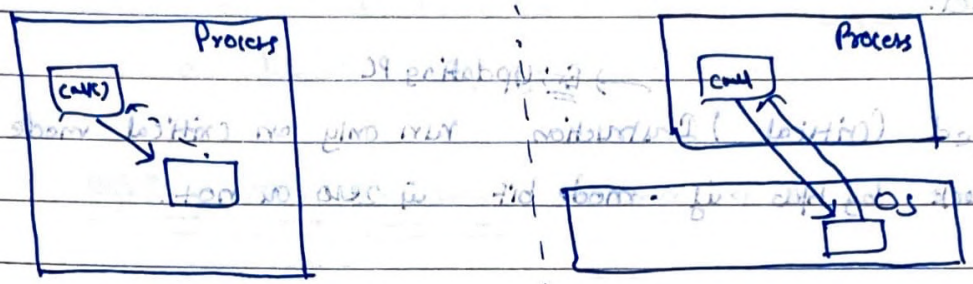
User wants to do both type of tasks, so normal task are done in user mode & privilleged tasks are done in kernel (or privelleged) mode.

There is a bit for it to tell CPU which mode we are currently in called mode bit (0 for kernel mode & 1 for user mode)

User asks for OS services i.e. previlleged tasks using system calls. These are entry point to the kernel. We call System call (written in OS code) to touch the h/w.

PAGE NO.:
DATE: / /

⇒ function calls v/s system calls:

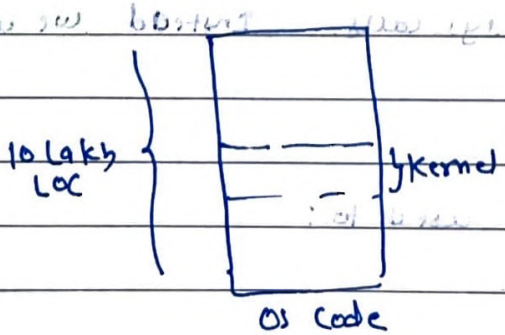


function call | System call

Caller & callee are in the same process (same address space)

Callee is written in OS code

⇒ Kernel: Some imp piece (part) of OS code (where all system calls are written)



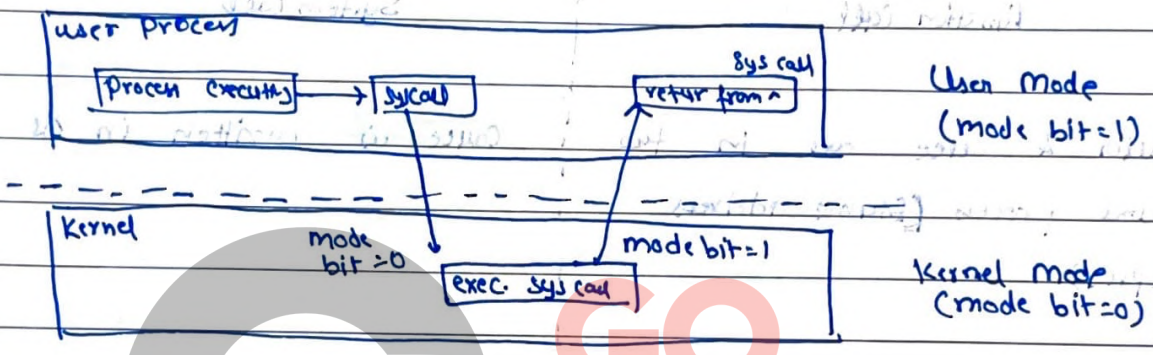
Kernel contains many function in which some are system calls (~400 in linux) & some are not (~600). These are not allowed for user to call. They are called inside OS & system call.

PAGE NO. :
DATE : / /

→ System calls, Interrupts (in (OA) & exceptions are entry points of kernel.

→ Ex: Updating PC

→ Privelleged (critical) Instruction run only on critical mode i.e. check by h/w if mode bit is zero or not.



We call sys call in C using libraries. fork() was not sys call but fun. fork may be in our OS which fork() may be calling in library.

Printf() is not syscall but write() is which is called by printf().

We don't directly invoke sys calls. Instead we use API (libraries of C).

Ques: System calls must be used to:

a) Modify global variables
b) Call a user written function
c) write to a file
d) All of the above.

PAGE NO. :
DATE : / /

Ques: What is system call?

→ Request to kernel to peform a service

→ Call a function (programmer's prespective)

* System Call Execution: (Same as Interrupt Exec. in COA)

→ Each system call has a unique numeric identifier (say 57 for fork(), 23 for write(), etc)

→ OS has a system call table that maps number to functionality requested.

→ We move unique no. to register & then execute instruction syscall which changed mode bit & now kernel checks that register & from table it executes requested call.

Ex: fork() { mov rax, 57; syscall; }

If we have some argument also, then we put them in a stack which can be accessed by kernel (in sys call). (Not reg. bcz arguments can be many).

Ex: printf("Hello"); { mov rax, 23; mov stack, "hello"; syscall; }

If syscall is returning something we put that in stack & then in user mode process accesses it.

PAGE NO.:
DATE: / /

⇒ **Steps in System Call execution:**

1) Programmer calls wrapper function in C library.
(Ex: printf, fork)

2) Wrapper function moves arguments in stack or register and puts unique sys call number into register.

3) Wrapper function flips CPU to kernel mode (user → kernel) by machine instruction (e.g. syscall). Now CPU can touch h/w.

4) Kernel executes syscall handler:
4.1) Invokes service routine (actual system call) corresponding to syscall no. & generates result.
4.2) Places return value in register or stack.
4.3) Switches back to user mode, passing back controll to wrapper. (kernel → user).

⇒ System Call table (i.e. mapping of unique no. to sys call) is usually implemented as an array of function pointers where each function implements one system call.

PAGE NO. :
DATE : / /

⇒ Inside the "syscall" machine instruction:

a) Saves old SP (stack pointer) value.
b) Switches SP to kernel stack
c) Saves old PC (program counter)
d) Saves old privellege mode
e) Sets privilege mode to 0.
f) Sets new PC using System Call table.

Note that privellege mode can be 0 for kernel & 1, 2, 3, ... for user which repr. severity of task.

⇒ Overheads of System calls are:

→ They are very slow.
→ They can take tens to hundreds of thousands of clock cycles.
→ This is due to:
- Changing mode,
- argument validation,
- Cache effects, etc.

So, programs should make fewer system calls when practical.

There are various kinds of system calls & they may have dif name in dif OS like- fork() in unix & CreateProcess() in windows. Types are

Process Control, File Manipulation, Communication, Protection, Device Manipulation, Info. maintainance, etc.

PAGE NO. :
DATE : / /

☆ **Threads:** In certain situations, a single application may be required to perform several similar tasks. In such situation we may not want to perform have different processes. The we create Threads.

Example can be notepad where we need to write, as well as check spelling, may be translation etc.

for the similar tasks if we have diff process instead of threads then:

- Inter process Communication (IPC) is required.
- Process context switching is costlier than thread context switching.

→ Threads are about Concurrency & parallelism.

(Concurrency: one after other; parallelism: parallely)

For parallelism we need multi-core system.

T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 - - - -

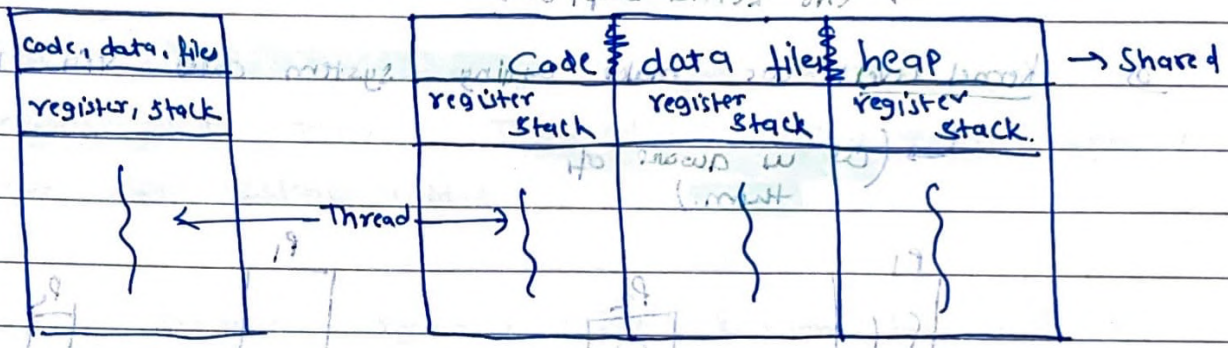
Concurrent Exec. of Single Core

Core 1	T_1 T_3 T_2 T_3 - - -
Core 2	T_2 T_4 T_3 T_4 - - - -

Concurrent & Parallel in Dual-Core.

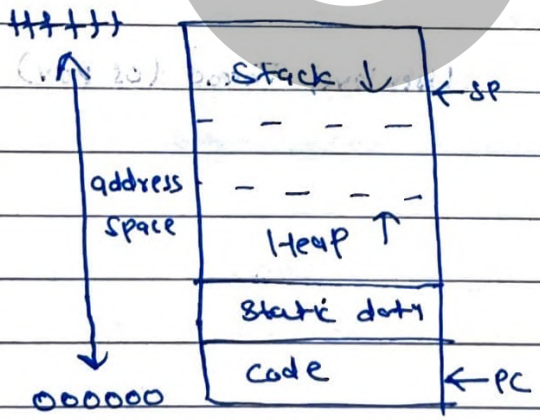
PAGE NO. :
DATE : / /

→ Things that threads must not share : Stack, register values
& things that they must share : code & data & heap.

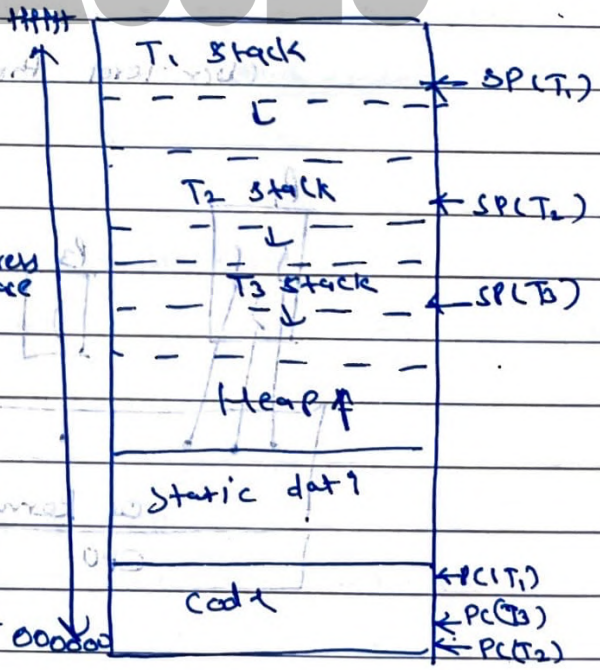


Single Threaded Process Multi Threaded Process

→ Some authors call threads lightweight processes because they have their own stack but can access shared data.



Process w/o threads.



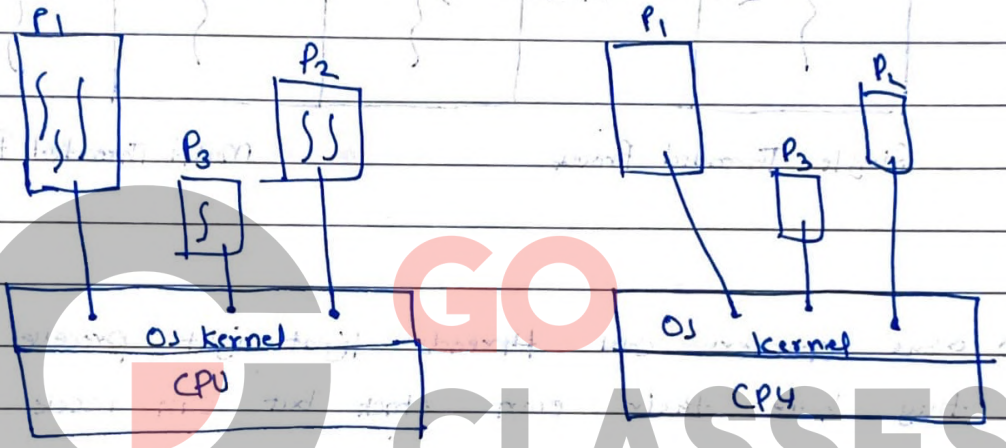
Process with threads.

PAGE NO. :
DATE : / /

⇒ There are two types of threads:

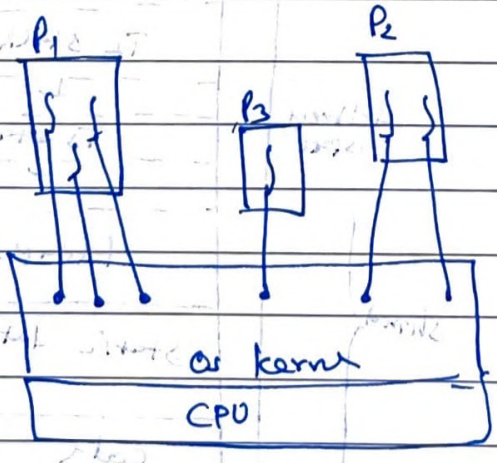
a) User Level: We make using function calls via library
(No kernel support)

b) Kernel Level: We make using system calls via library
(OS is aware of them.)



User level thread.

User level Thread (OS POV)



Kernel Level Thread (any POV)

PAGE NO. :
DATE : / /

→ User level threads are small and can be 10-100 x faster as compared to kernel threads. They are maintaine entirely by user level library (having no system call)

→ Each thread has a PC, register, stack and a Thread control block (TCB). Thread id is unique within proces but not system wide.

→ OS schedules only the proces & kernel threads, not the user level threads. Programmer manages (ie. create, schedule, sync, etc) user level threads by its own using libraries.

→ Advantage of user level thread is that they are fast since no OS involvement & disadvantage is that since no OS involvement, so if one threads block, then all other are also not scheduled by OS.

Que: Can user level threads run kernel code?

→ Yes, as we can have system call in user thread. No issue.

Que: Assume P_1 was running which now goes for I/O. Now we want to schedule P_2. For this we need to execute scheduler. How to schedule the scheduler?

→ We have a instruction written in P_1 which schedules the scheduler (sched;)
↳ Instruction.

These system call are just sw interrupts.

PAGE NO.:
DATE: / /

→ Also called traps

Interrupts (COA topic)

- S/w
  - System Call
  - Faults
    - → Page,
    - → seg,
    - → /by 0, etc
- H/w (Timer, Mouse Click, etc)

NOTE ↔ Any process transition (ready → running, or any) in b/w states is done using system call only. like we scheduled scheduler.

Clue: Arrange them in increasing order of cost:

a) New Process
b) Kernel Thread
c) User Thread

⇒ a > b > c

a → most costly
c → least costly

→ Kernel level threads are implemented in the kernel. OS can schedule them. If one thread (kernel level) is blocked, then OS known about it & can run other threads from that process.

→ Kernel level threads are pretty expensive as all thread operation like creating, scheduling are system call.

PAGE NO. :
DATE : / /

→ Switching b/w user level threads is faster than b/w kernel level threads since a context switch is not required.

Quiz: T/F

a) OS provides illusion to each thread that it has its own address space.

→ false

b) With kernel level threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously.

→ True.

c) If a user level thread is blocked for I/O, then kernel will run another user level thread from same process.

→ false.

d) Many to One multithreading model uses one kernel, so it cannot use multiple processors.

→ True.

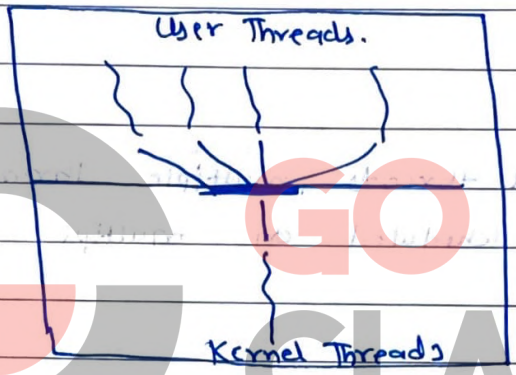
e) In multiple kernel level threads, various threads associated with a single process must share a common thread state.

→ false.

PAGE NO. :
DATE : / /

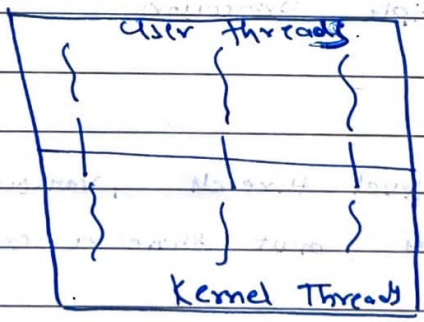
⇒ **Multithreading Models :**

a) **Many - to - One :** Various user level threads are mapped to single kernel thread. It is default if we don't have kernel threads.



Many to one

b) **One - to - one :** Each user thread maps to kernel thread. It is default if we ~~don't~~ have only kernel level threads

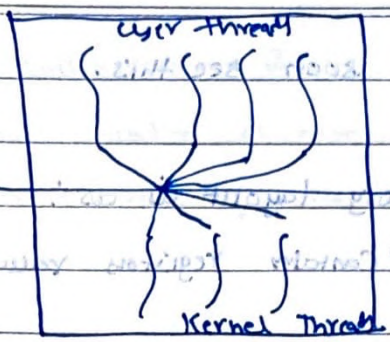


One to one.

c) **Many - to - Many :** Allows many user level threads to map to many kernel level threads.

These are not very common.

PAGE NO. :
DATE : / /



So, with this multithreading approach, multiple threads within the same application can run in parallel on multiple processors & a blocking system call need not block the entire process. If properly designed, this approach should combine the advantages of pure user & kernel level threads while minimizing the disadvantages.

→ We don't map all ULT to KLT bcz it's more overhead to OS to manage KLT.

★ Context Switching :

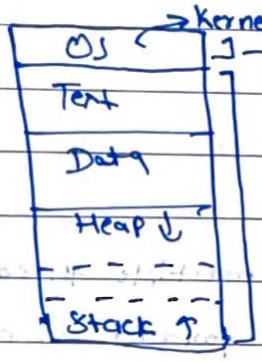
When a CPU starts to execute another function instead of the previous one, then we first save the registers of previous process somewhere in memory and then before starting execution of that process again, we restore the value of registers from the memory. This is known as Context switching.

→ Saving state of a process before switching to another process & restoring original process state when switching back.

PAGE NO. :
DATE : / /

→ How do we do it? We'll soon see this.

→ for every process, memory layout is as:



So when context switch happens, data of registers, PCB are stored in privileged area of memory called kernel stack.

→ Context switch has following steps:

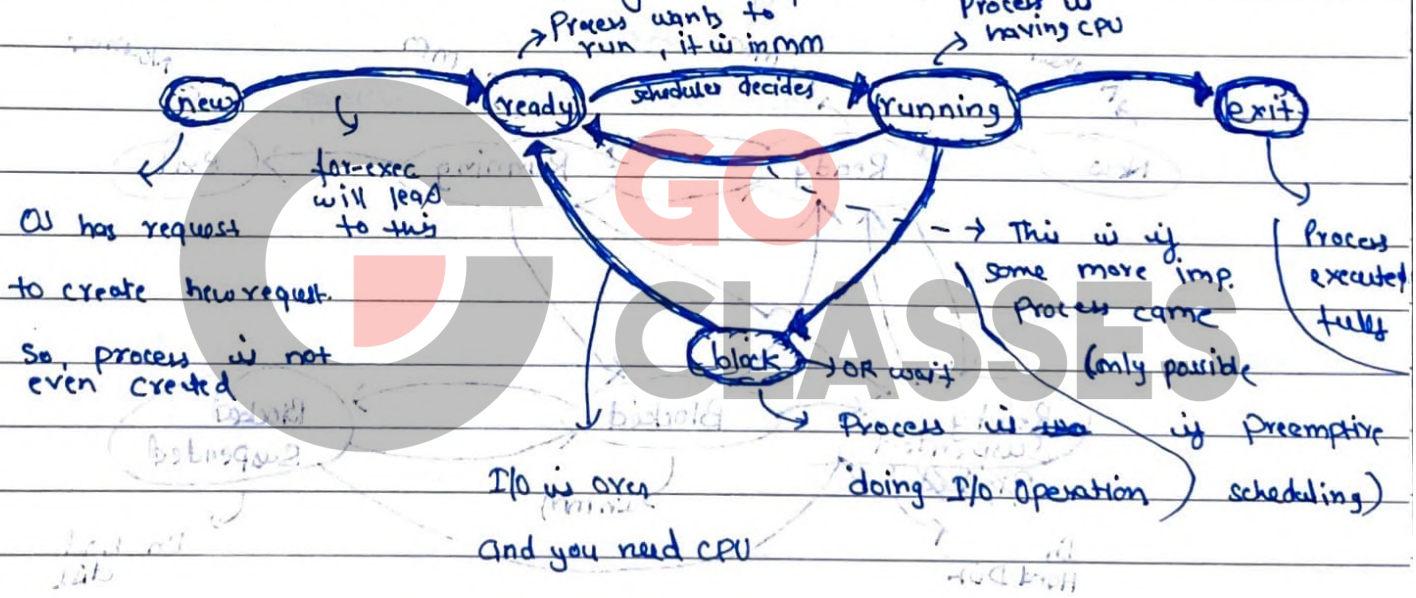
a) P_1 is running in user mode
b) Something happened & interrupt called.
c) P_1 is running in kernel mode
d) Saves context in stack (context switch)
* e) Now, scheduler starts to run. & another process runs & now again after sometimes P_1 gets the chance.
f) P_1 is running in kernel mode
g) Restore the context
h) P_1 is running in user mode.

So, context switch happens when process is already in kernel mode.

PAGE NO.:
DATE: / /

→ Thread Context Switch v/s Process Context Switch: Main d/f b/w both is that during a thread switch the virtual memory space remains same, while it changes in process switch. (Not Imp.)

★ **Process States:** There are various models to repr. various states of process. Here there is no hard and fast rule for the transitions, but some of the transitions are literally not possible.



Five State Diagram

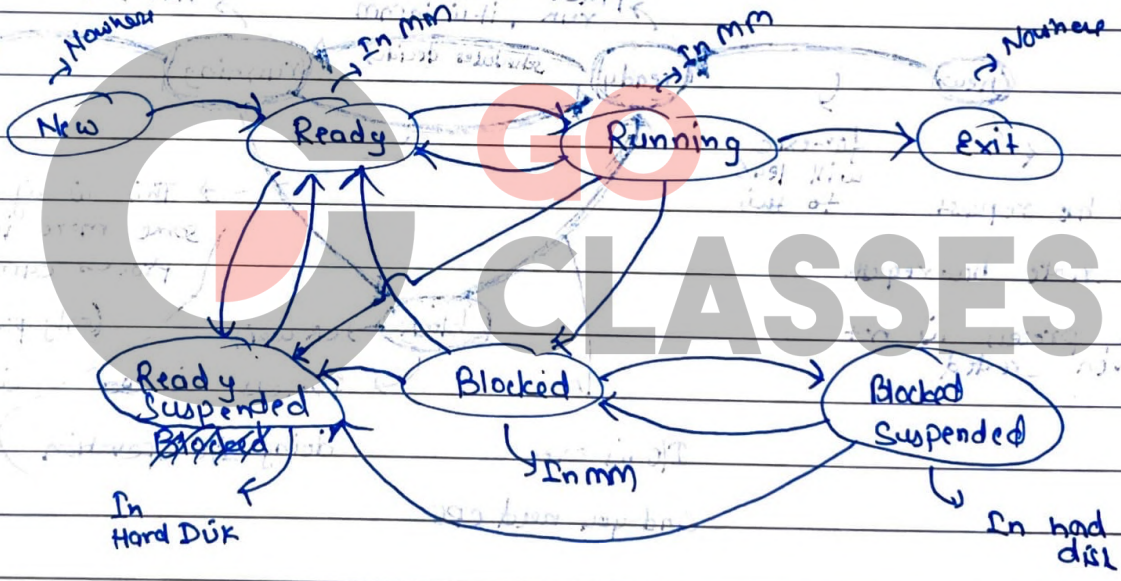
This is one of the possible set of states, OS designer based on his needs can have more/d/f set of states.

For Ex: There can be situation that OS want to put some process in Hard Disk from MM if MM is full (mostly in virtual memory). Then the process may go in new state called suspended

PAGE NO. :
DATE : / /

If we want to implement this then we need to new states i.e. ready suspended and blocked suspended. to distinguish where the process was before this.

NOTE: We follow van-neuman arch. (studied in COA) which says that we need process in MM only if we want to exec. instructions in that i.e. when it need CPU. and I/O can be done even if process is in secondary memory. (obviously I/O can be done in MM) Not mand.



7-State Diagram

★ Schedulers: There are mainly 3 types of schedulers where name depicts the frequency at which they run.

a) Long-Term Scheduler: It contains less frequent tasks & runs once in long time. Mostly it schedules new task to ready task.

PAGE NO. :
DATE : / /

(It is very rarely required in OS BTW). It runs after long time because we as a human create new task after very long time as compared to CPU movements.

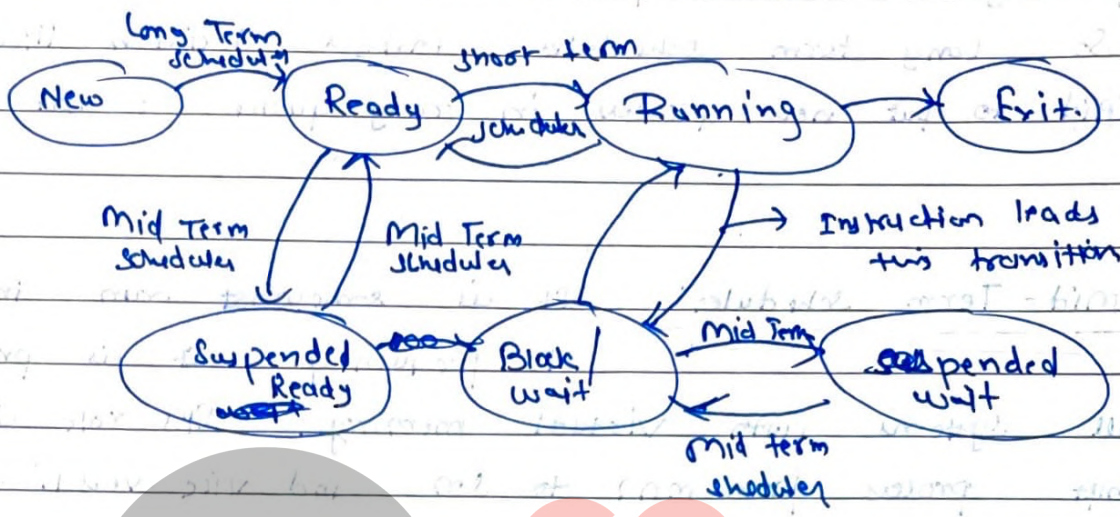
So. Long term scheduler makes decisions like when to put new process in ready queue or not put it.

b) Mid-Term scheduler: It is somewhat run in moderate frequency. It is present in all systems with virtual memory. It's role is to shift process from MM to SM and vice-versa. i.e. ready or blocked to ready suspend or blocked suspended respectively and vice versa.

c) Short-Term scheduler: We need to run this scheduler very frequently again and again. It is responsible for ready to running transition. Whenever CPU becomes idle, OS selects one of the processes from ready queue to be executed. This is carried out by short term scheduler also known as CPU scheduler. It chooses one of the process from MM that are ready to execute & allocates this CPU to that process. It is invoked frequently, so it needs to be fast and sometimes it is only scheduler in a system.

→ Long term scheduler controls degree of multiprogramming

PAGE NO. :
DATE : / /



Schedulers & States.

☆ **Scheduling:** Here we talk about short term CPU scheduling i.e. on what basis it will select process from ready queue to execute. It depends on the following criteria ie. what do we require:

→ Max. CPU utilization.

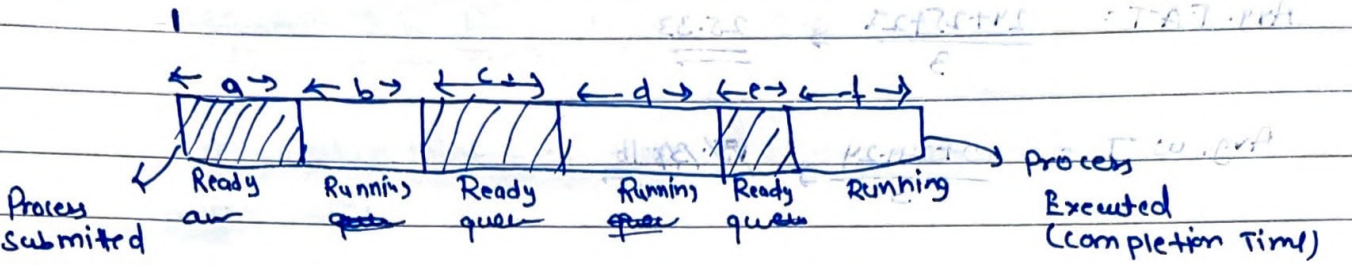
→ Max. Throughput i.e. # process completed executed per unit of time.

→ Min. Avg. Turn around time i.e. # time to execute process.

→ Min. Avg. Waiting time i.e. # time process waiting in ready queue.

→ Min. Avg. Response time ie. # time # process in ready queue till it got first response.

PAGE NO. :
DATE : / /



T.A.T. = a+b+c+d+e+f

Response Time = a

Waiting Time = a+c+e , Burst Time = b+d+f (# CPU needed)

→ Note that response time is same as waiting time in case of non-preemptive scheduling (i.e. once scheduled, CPU will not be taken back until it's fully executed).

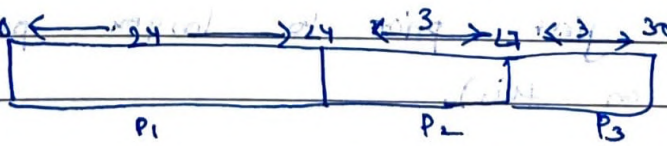
T.A.T. = Completion Time (C.T.) - Arrival Time (A.T.)

W.T. = T.A.T. - Burst Time

★ Scheduling Algorithms:

I First Come First Serve (FCFS): Simplest CPU scheduling algorithm.

Ex:	#P	B.T.	A.T.	C.T.	T.A.T.	W.T.
	P_1	24	0	24	24	0
	P_2	3	2	27	25	24
	P_3	3	3	30	27	24



✓

PAGE NO. :
DATE : / /

Avg. TAT = $\frac{24+25+27}{3} \Rightarrow \underline{25.33}$

Avg. WT = $\frac{0+24+24}{3} \Rightarrow$ ~~17.33~~ 16

If the same processes arrived in order P_2, P_3, P_1 then Avg. Times are reduced ~~waiting~~ very significantly. ($\simeq 3$)

Thus, the average waiting time in FCFS algorithm is generally not minimal & may vary if burst time vary greatly.

→ Convoy Effect: Short B.T. process waiting for long sized B.T. processes to complete.

Ex: if ferrari → lambo → Tractor in single lane, then supercars are undergoing convoy effect

So, FCFS undergoes convoy effect.

$x \gg y$.

Convoy effect obviously reduces throughput very significantly.

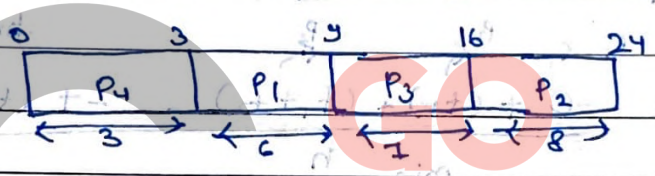
→ The solution to convoy effect seems to be to put shorted process first prior to longer processes. (SJF is based on this)

PAGE NO. :
DATE : / /

II : Shortest Job first (SJF): Assign CPU the process with min CPU burst process available at that point of time. We may break the tie using FCFS.

Ex:

#P.	B.T.	A.T	C.T.	T.A.T.	W.T
P1	6	0	9	9	3
P2	8	0	24	24	16
P3	7	0	16	16	9
P4	3	0	3	3	0



Avg. W.T: $\frac{3+16+9}{4} \Rightarrow \frac{28}{4} = 7$

If we do schedule above processes by FCFS we get the avg. waiting time = 10.25 which is more than SJF.

→ Always, SJF performs better as compared to FCFS. (No case when it doesn't). We can also prove it; but not required. Idea, I am writing:

Moving short process before long process will decrease waiting time of short process more than it will increase W.T. of long process & so, avg. W.T. decreases.

PAGE NO. :
DATE : / /

If $t_1, t_2, t_3 \ldots t_n$ are for processes $P_1, P_2, P_3 \ldots P_n$ respectively and $t_1 < t_2 < t_3 \ldots < t_n$, then SJF will schedule them as $P_1 \to P_2 \to P_3 \ldots P_{n-1} \to P_n$ (at last).

If A.T is same then

Avg. W.T = $0 + t_1 + (t_1 + t_2) \ldots$

Skippable

→ If $t_1 \ldots t_n$ are B.T. for process $P_1 \ldots P_n$ and $t_1 < t_2 \ldots < t_n$ then SJF will schedule P_1 first & P_n at last. If A.T is zero for all n processes then

Avg. W.T. = $\dfrac{0 + (t_1) + (t_1 + t_2) \ldots + (t_1 + t_2 \ldots + t_{n-1})}{n}$

(for P_1, P_2, P_3, for P_2)

(Max. time, Min. time)

$= \dfrac{(n-1)(t_1) + (n-2)(t_2) + \ldots + 1(t_{n-1})}{n}$

(Not Required)

But, SJF has got two main problems:

a) Starvation: Some long job may never get chance if smaller jobs keep coming.

b) How will we know the B.T. in advance w/o running on CPU? We don't know this in advance, that's why SJF is not possible practically. However, we can predict it based on historical data.

PAGE NO.:
DATE: / /

We use previous burst times to predict the expected burst time of next process by averaging them.

Let S_5 repr. predicted B time for process 5 and t_1, t_2, t_3, t_4 are B.T. of P_1, P_2, P_3 and P_4 processes.

Then $S_5 = \frac{t_1+t_2+t_3+t_4}{4}$

Assume actual B.T. comes out to be t_5 for process 5.

then $S_6 = \frac{t_1+t_2+t_3+t_4+t_5}{5}$

$= \frac{t_1+t_2+t_3+t_4}{5} + \frac{t_5}{5}$

$= \frac{4}{5}(S_5) + \frac{t_5}{5}$ $\left(\text{as } S_5 = (t_1+t_2+t_3+t_4)/4\right)$

Similarly $S_7 = \frac{5}{6}(S_6) + \frac{t_6}{6}$

S_n depends on S_{n-1} & t_{n-1} i.e. actual & predicted B.T.s of previous processes

Here we are giving more weightage to previous prediction (5/6) & less to actual (1/6) B.T.

$$S_n = \frac{(n-2)}{(n-1)} S_{n-1} + \frac{t_{n-1}}{(n-1)}$$

Instead of fixing these weights, we can use a factor of how much we trust real data by;

$$S_n = \alpha S_{n-1} + (1-\alpha) t_{n-1}$$

$\alpha > \alpha^2$ if $0 < \alpha < 1$

PAGE NO. :
DATE : / /

Earlier we were rigid in the sense that we were always giving less weight to the recent actual time & more weight to the predicted time, but now we are using variable 'α' for this.

Case I : $\alpha = 0$, $S_n = t_{n-1}$ same as actual prev

Case II : $\alpha = 1$, $S_n = S_{n-1}$ same as predicted prev.

→ If we unroll it:

$$S_n = \alpha^3 s_{n-3} + \alpha^2(1-\alpha)t_{n-3} + \alpha(1-\alpha)t_{n-2} + (1-\alpha)t_{n-1}$$

(unroll this too)

then one thing that can be noticed is that, this formula gives more weightage to recent actual time $[(1-\alpha)t_{n-1}]$ and lesser to previous time $[(\alpha^2(1-\alpha))t_{n-3}]$.

→ So, SJF is most ~~prac~~ optimal, but it's not practical we have to rely on prediction.

NOTE ↔ If we are not using CPU to execute an run process, then we are not effectively using CPU.

CPU Efficiency = $\frac{\text{time exec. jobs}}{\text{time exec. jobs} + \underset{\text{time}}{\text{Context switch}} + \underset{\text{time}}{\text{other}}} \times 100\%$

PAGE NO. :
DATE : / /

Qu: We have following jobs:

#Pid	A.T.	B.T.
P_1	0	8
P_2	0	7
P_3	0	4
P_4	6	16

We use SJF, then calculate averaging time S_4 with $S_1 = 10$ & $\alpha = 0.5$.

$\Rightarrow$ $S_2 = 0.5 * (10)^{S_1} + 0.5 * (4)^{t_1}$

$= 7$

$S_3 = (0.5 * 7) + (0.5 * 7)$

$= 7.$

$S_4 = (0.5 * 7) + (0.5 * 8)$

$= 7.5.$

Ans.

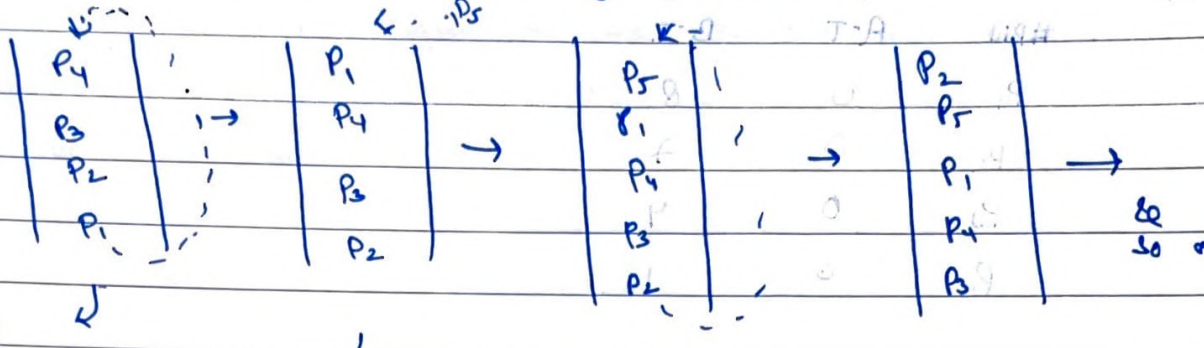
III Round Robin Scheduling Algorithm (RR): It is a preemptive scheduling algorithm. Preemption means stopping execution of one process to execute another. Preemption is required to give user an illusion that several apps are running parallelly.

With each preemption, we do a context switch.

In RR we have a time quanta for which we run each process one by one.

PAGE NO. :
DATE : / /

→ It can be implemented using circular queue.



P_1 executed for its time quanta

before P_1's exec. P_5 arrived in Ready queue.

P_2 executed

So if some process is coming in b/w then we park it at the end of circular queue (may be LL).

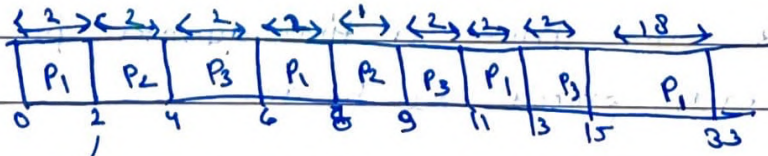
→ Quanta of RR is a timer maintained by H/w. So it's not a s/w interrupt.

→ Scheduler here needs to be very fast bcz many switches will occur & it's fast bcz we just do push & pull in queue.

Ex:

#Pid	A.T.	B.T.	T.A.T	W.T	C.T
P_1	0	~~2 4 2 6~~ 18	33	9	33
P_2	0	~~5 x 0~~	9	6	9
P_3	0	~~6 4 x 0~~	15	9	15

Quanta = 2



In b/w CS time is there

Avg W.T. = $\frac{9+6+9}{3} \Rightarrow 8$

↳ forming for this que.

PAGE NO. :
DATE : / /

NOTE: If we do not have any other process in ready queue then we will skip context switch and continue executing current process. (However, depends on implementation)

→ So, In RR no process is allocated CPU for more then 1 time quantum in a row unless it is the only runnable process.

Que: If a system using RR algo with quanta = 195 ms has C.S time 5 ms. then, CPU utilisation is?

⇒ $\frac{195}{195+5} * 100 \Rightarrow \frac{195}{200} * 100 \Rightarrow 97.5\ \%$

→ It will be near 100% if C.S. time is about zero or very less then time quantum, so that quanta ≈ quanta + CS time.

(Ex: $(10^6 / 10^6 + 0.01) = 100\ \%$)

→ If we keep large quanta then RR behaves like FCFS & we may fell sick of convoy effect.

→ If we keep small quanta then CPU efficiency decreases due to context switch times.

Typically,

→ RR provides higher avg. TAT then SJF, but lesser response time then SJF.

→ Avg waiting time is lower if job lengths vary widely, but higher if they have similar lengths (worse then FCFS).

PAGE NO. :
DATE : / /

→ Choosing right quantum is a tradeoff.

Ques We have process P_1 & P_2, P_1 is running & P_2 is ready to run. OS uses RR. If P_1's quantum is about to expire & P_2 will be running next. List the sequence of events that will occur:

→ a) A timer ~~etc~~ interupt occures.
b) P_1's user level stack is saved in trapframe (part of kernel level thread).
c) OS scheduler runs
d) Context switch from P_1 to P_2
e) P_2's user level state is restored from trapframe.

Ques We have P_1 & P_2. P_1 has 3 threads & P_2 has 2 threads OS uses R.R. scheduling. What % of processor's time will be spend for running P_1's thread if all threads are a) user level: → 50 %
b) Kernel Level: → 60 %

6

→ FIFO (or SJF) is preferable for batch apps where user submits the jobs, goes away, and then come back to get results.

→ RR is preferable for Interactive Applications.

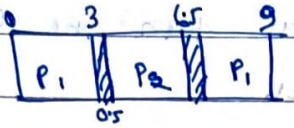
PAGE NO. :
DATE : / /

Qu: Throughput? , CPU utilization = ?

Quanta = 3
B.T.

P_1 5

P_2 3



⇒ 2 jobs per 9 unit of time.

= 2/9 jobs per unit of time. (Throughput)

⇒ 8/9 = 88.89% is CPU utilisation.

Qu: If all jobs have same length, a RR scheduler (with time slice less than length) provides better avg. T.A.T than FIFO?

⇒ False, with RR, each job will finish at nearly same time (one quanta before each other), which will give poor T.A.T.

Ex: 10 jobs with each taking 100 second.

In FCFS, jobs will finish at 100, 200 ---, 1000 sec. respectively

In RR, with quanta = 1 second, jobs will finish at 991, 992 ... 1000 seconds respectively.

So, Avg. TAT of RR is worse if identical length jobs are there with less quanta.

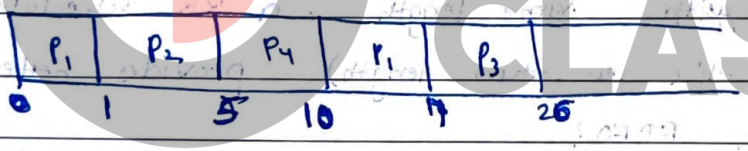
PAGE NO. :
DATE : / /

IV **Shortest Remaining Time First (SRTF):** It is a preemptive version of SJF. So, if the newly arrived process has a shorter running time than what is left of the currently executing process then the OS will preempt the current process.

Ex:

Process	A.T.	B.T.	T.A.T	W.T.	C.T.
P_1	0	~~8~~ 7 0	17	9	~~17~~
P_2	1	~~4~~ 4 0	4	0	5
P_3	2	9	24	15	26
P_4	3	~~2~~ 1 0	7	2	10

→



Avg. W.T. = $\frac{9+0+15+2}{4} = \frac{26}{4} = 6.5$

→ SRTF can be implemented using min heap, whenever a new job enter in ready queue we insert that in min heap, decrease keys of current scheduled job with remaining B.T. and then pick the job with min. B.T.

→ FCFS can be implemented by queue

→ SJF " " " min heap

→ RR " " " circular queue / LL.

→ LJF/LRTF " " " Max heap

↳ comingsoon.

PAGE NO. :
DATE : / /

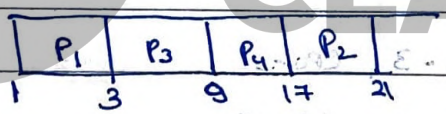
V. **Longest Job first (LJF)**: Opp. of SJF. In this longer burst jobs will be scheduled first. (Non preemptively). Can lead to starvation.

VI. **Longest Remaining Time first (LRTF)**: Preemptive version of the LJF algorithm. Can lead to starvation.

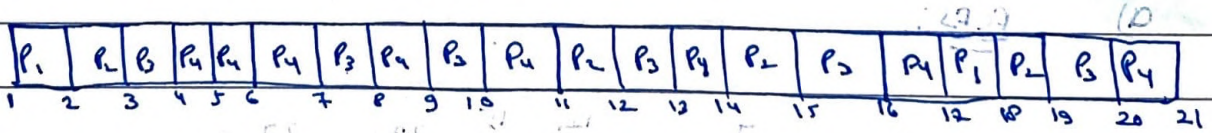
Ex:

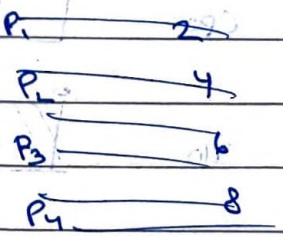
Process	A.T.	B.T.
P_1	1	2
P_2	2	4
P_3	3	6
P_4	4	8

a) LJF.



b) LRTF





(Same as Gate 2006 (CSE))

PAGE NO. :
DATE : / /

★ **Scheduling with Mixed Workloads:** Practically, process do not do only CPU.

It also goes for I/O operation in b/w i.e. blocked state.

Ex: P_1: CPU → IO → CPU →

P_2: CPU → IO → CPU → IO →

Note that with a single CPU, two process can't do CPU together, but they can do I/O with overlap by default unless explicitly given in the question.

If process do more I/O (less CPU) than process is called I/O bound.

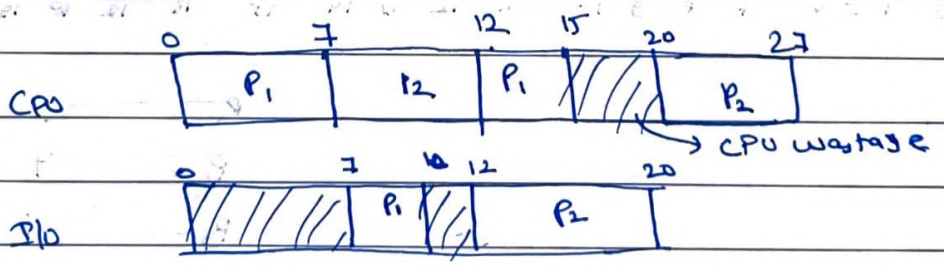
If " " " CPU (less I/O) " " " " CPU bound.

Ex: P_1: CPU - 7, I/O - 3, CPU - 3.

P_2: CPU - 5, IO - 8, CPU - 7.

Draw Gantt chart for below algorithms:

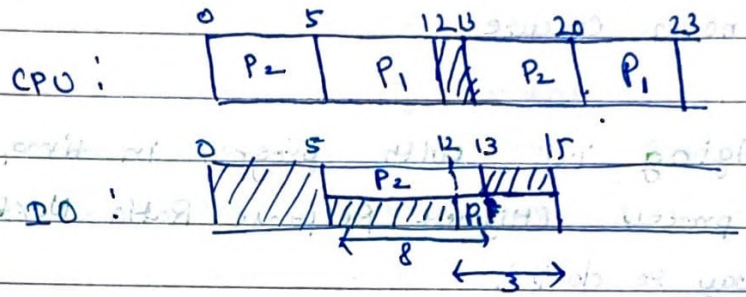
a) FCFS:



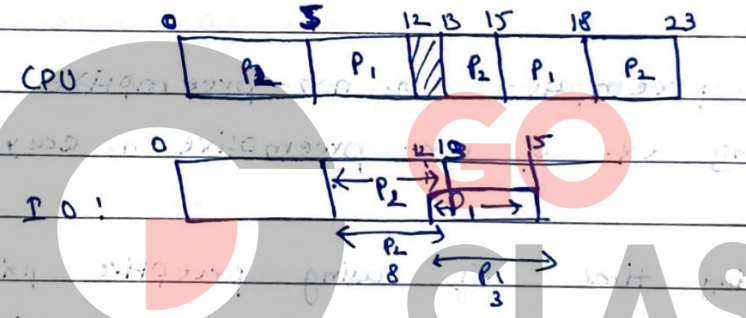
Here we choose process which came first in ready queue either from new state or from blocked state, it doesn't matter. We just do FCFS.

PAGE NO. :
DATE : / /

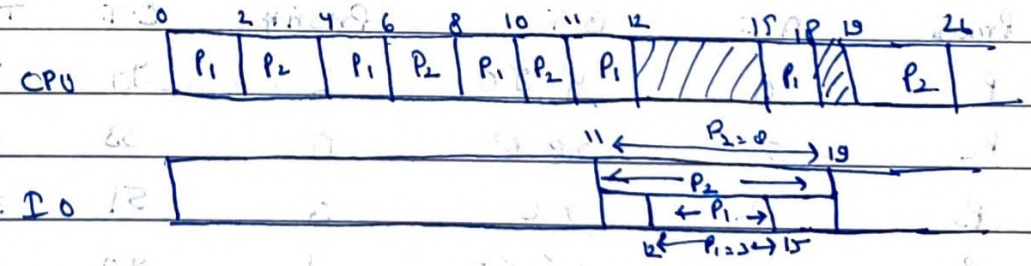
b) Shortest Job first.



c) SRTF:



d) RR with quanta = 2



☆ Priority Scheduling: A priority number is associated with each process. CPU is allocated to the process with highest priority

→ SJF is priority scheduling algorithm where priority is inverse of predicted next CPU burst time.

Generally, less number ↔ Higher priority
↳ Default

PAGE NO.:
DATE: / /

→ It may also suffer from starvation as low priority processes may never execute.

→ Solution is Aging i.e. with progress in time, increase priority of process. (Highest Response Ratio Next is one of the way to do it).

→ Priority scheduling is static if priority once given do not change, else it is dynamic.

→ It can be preemptive or non-preemptive.
(Not writing ex. of non-preemptive as easy).

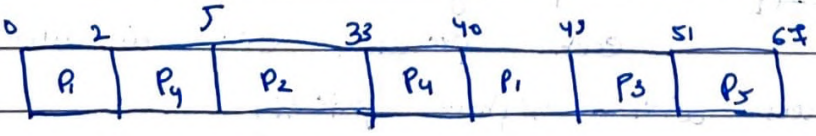
GO CLASSES

Gate 2017 CS
Que: The avg. waiting time if using preemptive priority scheduling algorithm (0 as highest priority) is:

Part of syllabus

Process	A.T.	B.T.	Priority	C.T.	T.A.T.	W.T
P_1	0	~~8~~ 10	2	49	49	38
P_2	5	~~28~~ 0	0	33	28	0
P_3	12	~~2~~ 0	3	51	39	37
P_4	2	~~10~~ ~~7~~ 0	1	40	38	28
P_5	9	~~16~~ 0	4	67	58	42

⇒



Avg W.T. = $\frac{38+37+28+42}{5} = \frac{145}{5} = 29$

PAGE NO. :
DATE : / /

Que: Suppose a processor uses a priortized round robin scheduling policy. New process are assigned an initial quantum of length q. Whenever a process uses its entire quantum w/o blocking its new quantum is set to twice its current quantum. If a process blocks before its quantum expires, its new quantum is reset to q. Assum every proces terminates. Is starvation possible?

a) If scheduler gives higher priority to process with larger quanta.

→ No, because process will terminate someonne.

b) If scheduler gives higher priority to process with smaller quanta.

→ Yes, if new process keep coming with quanta q, then old with $2q, 4q$ will starve.

Gate 2015 CSE

Que: The maximum no. of processes that can be in ready state for a computer system with n CPU, is:

→ Independent of n.

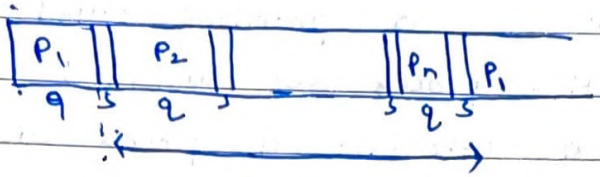
(→ 'n' in running state.)

Gate 98 CSE

Que: Consider n processes sharing CPU in RR fashion. Assuming that each process switch takes s seconds, what must be the quantum size 'q' such that each process is guaranteed to get it's turn for CPU in atleast every 't' seconds?

PAGE NO. :
DATE : / /

⇒



t can be atleast this. Greater will work bcz if 't' is more P_1 will get its turn within 't' time.

$$t \geq (q)(n-1) + (s)(n)$$

$$t \geq qn - q + sn$$

$$\frac{t - sn}{n-1} \geq q$$

$$\boxed{q \leq \frac{t - sn}{n-1}}$$ (a)

(Many PYQ's from this)

← Try Them

PAGE NO. :
DATE : / /

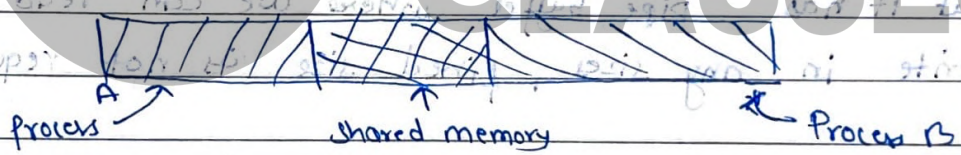
☆ **Interprocess Communication (IPC):** (No PYQs till now, so not much imp. so will not go in details)

→ If two process within a system wants to communicate, then they will take help from O.S. but if they are in d/f system, then along with OS, CN help is also required.

☆ OS provides various facilities for IPC such as:

a) **Shared Memory:** This is easiest way for IPC.

There is shared memory in user space where one process writes and other reads.



→ One process will create an area in RAM which the other process can access for this, they need help from OS. (i.e. for creation)

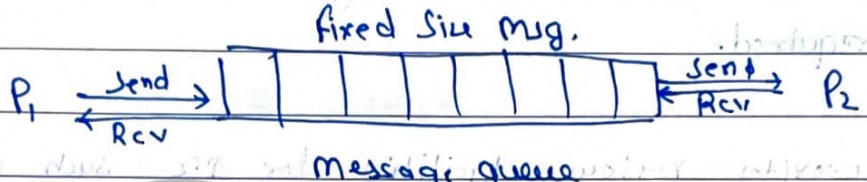
→ Now both can access that shared memory like a regular memory w/o OS intervention i.e. in user mode bcz shared memory is in user space.

→ This shared memory is not heaps of any process nor stack, it is d/f memory.

PAGE NO. :
DATE : / /

b) **Message Passing:** In this shared memory is created in the kernel space. So system calls are needed for the communication.

It's benefit is they do not share any address space for communicating, so memory isolation abstraction is maintained.



message queue in kernel space.

Not imp.

c) **Pipes:** It can only be implemented within parent & child. Almost same as message passing but it has pipe buffer where we can read & write in any sizes, fixed size is not required.

in kernel space

All of these can be implemented with support from libraries. (sir showed C code).

Module-2

PAGE NO. :
DATE : / /

☆ Synchronisation:

→ If multiple processes try to read & write in a shared memory then we should be carefull, because it may lead to inconsistency which we did not expect.

→ Need of synchronisation with the help of this example:

"The too much milk problem"

time	You	Your Roommate.
3:00	Arrive Home	
3:05	Look in fridge, no milk	
3:10	Leave for Grocery	
3:15		Arrive home
3:20	Buy Milk	Look in fridge, no milk
3:30		Leave for grocery
3:35	Arrive Home with milk	
3:40		Buy Milk
3:45		
3:50		Arrive home with milk.

→ Oh, now "extra milk's money is wasted.

Something similar can be the issue in OS if you & your roommate are process (or threads) & fridge is shared memory.

PAGE NO. :
DATE : / /

For Ex: $x = 50;$
$x = x+1;$

Suppose these two loc are run by two processes, & x is shared. If both of them run it concurrently then x will be 52 by sure at the end of both processes' execution.

But, if we allow interleaving & they run concurrently final value of x can be 51 or 52.

→ Serial execution & many other scenes.

→ How?

⇒ $x = x+1$; looks like this in low level: (1000 is loc)

1) LOAD R_1, 1000
2) ADD R_1, 1
3) STORE R_1, 1000

(will not change ans. if preempted after 2nd line)

a) Now assume P_1 runs line 1. & is gets preempted. Its Register R_1 has value 50.

b) Now P_2 gets scheduled & it completes execution of all three lines (BTW R_1 i.e. Registers for both P_1 & P_2 will be different despite having same name), & writes 51 in x.

c) Now P_1 again gets scheduled from 2nd line, it's register contain 50. It updates it to 51 locally & then stores back 51 in x.

& x finally has 51.

So, this is where Synchronisation is required.

→ This situation is called Race Condition.

PAGE NO. :
DATE : / /

→ **Race Condition**: A situation where several processes access & manipulate the same data concurrently & the outcome of the execution depends on the particular order in which the access takes place is called a Race condition.

→ **Critical section**: Area of Code / Part containing the shared variable (Ex: n=n+50, in prev. example)

→ The important feature of the system is that, when one process is executing in its critical section no other process is allowed to execute in its own critical section.

→ Very naive way to achieve this is using sequential execution, but it reduces CPU efficiency & not beneficial if using preemptive scheduling algorithms.

→ An analogy that can be developed here is using washroom (analogue to critical section) having free & engaged ~~tags~~ tags.

→ Note that critical sections in different threads are not necessarily the same code segment.

→ General structure of process

```
do{
    entry section
    critical section
    exit section
    remainder section
}while(true);
```

PAGE NO. :
DATE : / /

→ And we have to write entry and exit codes for synchronisation. For this we will try various attempts.

Before that we will go over some important necessities we need in our solution for synchronisation.

** **Que:** What can be the minimum & maximum value of count?

```
count=0;

Thread1:
for(i=0; i<10; i++) { count++; }

Thread2:
for(i=0; i<10; i++) { count++; }
```

⇒ **Maximum : 20**

↳ when we execute serially.

Minimum : 2

↳

T1 → load Instruction only.
So register is having 0.

T2 → All Instructions 9 times
So value of count is 9.

T1 → Increment & store in count.
So value of count is 1.

T2 → Load Instruction only
So register is having 1.

T1 → Complete all of its execution (9)
Count is 9.

T2 → Increment & store in count, (finish exec)
count is 2

CSe = Critical Section
CSw = Context Switch

PAGE NO.:
DATE: / /

→ These aptitude type of question, just hit and trial will work here.

☆ Requirements / Necessities of Critical Section Solution:

a) Mutual Exclusion: At most one process in C.Se.

```
while(1);          while(1);
  CSe code           CSe code
```

→ This code is providing ME as none will be able to enter CSe, which satisfies above requirement of at most 1.

→ As we can see alone ME cannot provide solution to Critical Section. We have other requirements too.

b) Progress: It says there should be some progress i.e. Atleast one process in C.Se (means no deadlock)

Now ME & Progress → Critical Section Solution.
flag[1] and flag[2] → = 1

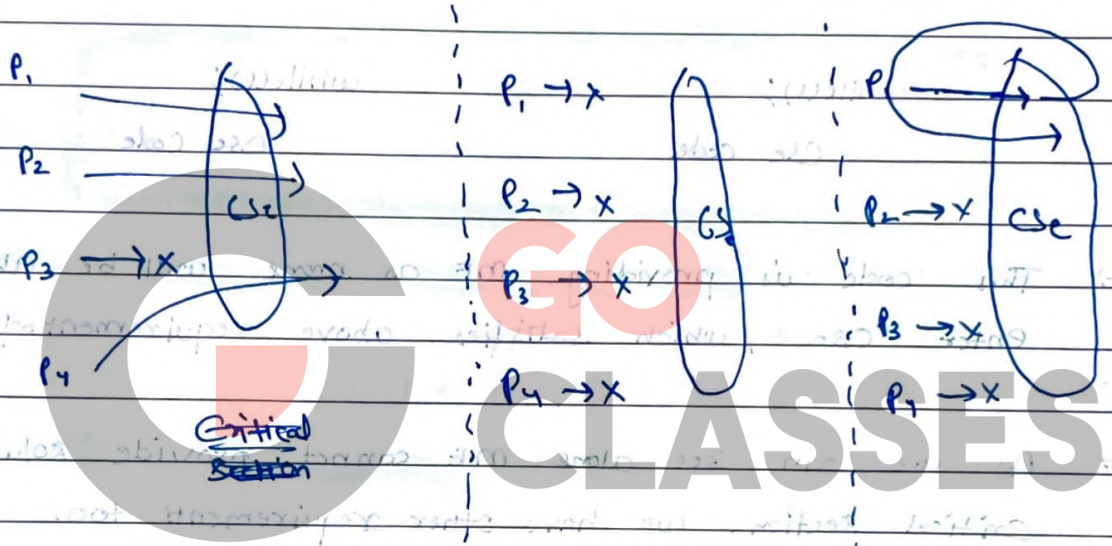
→ So both these condition says exactly one process is allowed to be in critical section.

→ Another requirement can be:

BW ≡ Bounded Waiting.
ByW = Busy W ≡ Busy waiting

PAGE NO. :
DATE : / /

c) **Bounded waiting**: If P_1 executed CS i.e. P_2, P_3, P_4 are also in req. to execute Critical section then P_1 should not be immediately allowed consecutively to execute critical section. In case noone is waiting then it can be allowed.



Violations: Mutual Exclusion ; Progress ; BW

→ So any solution following ME + Progress + BW conditions, then it is a valid CSe solution.

→ Now, there are three ways to implement CSe solution.

a) via s/w: write user code.

b) via OS support: code with system calls.

c) via H/w: special h/w instructions help.

PAGE NO. :
DATE : / /

⇒ **Formal Definitions:** (By Galvin)

a) **Mutual Exclusion:** If process P_i is executing in its CSe, then no other process should be exec. in their CSe.

b) **Progress:** If no process is executing in its critical section & some process wish to enter their CSe, then only those process that are ... (tough)
↳ Not writing.

* In laymen terms, the purpose of this condition is to make sure that either some process is currently in CSe & doing some work, or, if there is at least one process that wants to enter the CS, it will & then do some work. In both cases, some work is getting done & therefore all processes are making progress overall.

c) **Bounded Waiting:** There exists a bound, or limit, on the no. of times that other process are allowed to enter their CSe after a process has made a request to enter its CSe & before that request is granted.

In laymen terms, process P should not be bypassed by P'.

Here, formally reenter won't be issue if we are giving a guarantee on the no. of times it can reenter immediately. But, by default we keep this no. of times as 1.

PAGE NO. :
DATE : / /

→ Goodway

**

☆ To solve questions, check following conditions (if can't do these, then not a CSE solution)

1: Mutual Exclusion.

1.1: One process in CSE, another process tries to enter → Show that second process will block in entry code.

1.2: Two (or more) processes are in the entry code → Show that at most one will enter CSE.

2: Progress (== Absence of Deadlock)

2.1: No process in CSE, P_i arrives → P_i enters.

2.2: Two (or more) processes are in the entry code → Show that atleast one will enter CSE.

3. Bounded waiting (== Fairness)

3.1: One process in CSE, another process is waiting to enter → Show that if first process exits the CSE & attempts to reenter, Show that waiting process will enter.

⇒ Now try to find counter example of above conditions, if able to do, then respective requirement not satisfied & CSE soln is not a good soln.

while (intrested) ≡ while (intrested == 1)

intrested ↔ busy ↔ lock (used interchangebly)

PAGE NO.:
DATE: / /

** Critical Section solutions: (Software Based)

⇒ Attempt 1:

```
intrested = 0;   // shared.

P1:                                  P2:
1) while (intrested);                while (intrested);
2) intrested = 1;                    intrested = 1;
3) // CSe                           // CSe
4) intrested = 0;                    intrested = 0;
```

1:2 : Satisfied ?
↳ No.

If P_1 is preempted before line 2, after executing while & before updating intrested and P_2 is scheduled, then both will enter in CSe ie Mutual Exclusion is not guaranteed.

1:1 : Satisfied ?
↳ Yes

No issue if P_1 is in CSe then P_2 to enter; it will be blocked.

So, attempt failed as no ME.

⇒ A CSe soln is acceptable if ∀ executions all three requirmts are satisfied.

PAGE NO. :
DATE : / /

2.1 : Satisfied
↳ Yes. as new process can always enter if nobody in CSe & busy will be zero.

2.2 : Satisfied
↳ Yes

3.1 : ~~Not~~ Satisfied
↳ No, as P_1 can enter again & again w/o asking P_2.

So, in this Soln ME → ✗, Progress → ✓, BW → ✗

Attempt 2:

int turn = 0; // (or 1) shared var

P_0:	P_1:
while (turn != 0) ;	while (turn != 1) ;
CSe	CSe
turn = 1;	turn = 0;

→ Note that it will provide strict alteration, $P_0 P_1 P_0 P_1 \ldots$
(Assume this only for above two processes)

1.1 → Satisfied ✓

1.2 → Satisfied ✓

2.1 → Satisfied ✗
↳ No, assume turn is zero, & P_1 want to execute, it can't.

PAGE NO. :
DATE : / /

i = 1-j

i	j
1	0
0	1

3:1 → Satisfied → Yes → bcz alternate process will go now.

So, this soln has ME ✓, Progress X, BW ✓

Attempt 3:

```
int want[2] = {F, F};   //shared var

P0:                              P1:
1) want[0] = T;                  want[1] = T;
2) while (want[1] == T);         while (want[0] == T);
3) CSe;                          CSe;
4) want[0] = F                   want[1] = F;
```

ME: 1:1 → Satisfied ✓
1:2 → Satisfied ✓

Progress: 2:1 → Satisfied ✓
2:2 → Not Satisfied ✗
↳ If P0 preempts after line 1 & P1 starts exec. deadlock will occur.

BW: 3:1 → Satisfied.
↳ Do not forget that if P1 wants to enter & P0 is in CS then want[1] is True. & BW is clearly satisfied.

PAGE NO. :
DATE : / /

NOTE ↔ Deadlock is one of the reason due to which progress halts. Other reason is when single intrated process is not allowed.

GE 2010 Question:

P1 :
```
while(S1==S2);
CS
S1 = S2;
```

P2 :
```
while(S1 != S2)
CS
S1 = not(S2)
```

S1 & S2 are shared boolean variables.

⇒ ME ✓ Progress ✗ BW ✓

Similar to Attempt-2 i.e. Strict Alteration.

* → Progress is not satisfied when any process which is not interested to enter into the CSe will not allow other interested process to enter into the critical section.

NOTE ↔ Process's code need to be surrounded in infinite loop to be able to run again & again. Take care of this, else there can be a huge problem.

PAGE NO. :
DATE : / /

Attempt - 4:

Peterson's Solution: It is a combination of Attempt 2 & Attempt 3.

```
//Shared variables:
int want[2] = {F,F};
int favored = 0;
```

Process 0:

```
want[0] = True;
favored = 1;
while (want[1] && favored==1);
CS;
want[0] = false;
```

Process 1:

```
want[1] = True;
favored = 0;
while (want[0] && favored==0);
CS;
want[1] = false;
```

Here favored (favour) var. represents that I allow other process before me if it arrived before me set want var as True. Process will not run if want of other process is True & we have favored it.

It is somewhat analogus to trial room where two people want to go but asking each other that you go first you go. Person saying last that you go will not go & other person will go as first person said at last.

Note that w/o favored it is Attempt 3 & having deadlock when both want are True.

PAGE NO. :
DATE : / /

favored will break tie if both want are True i.e. both are interested. Process setting favored at last will not execute & will favor the other person to execute.

It is satisfying all (1.1, 1.2, 2.1, 2.2, 3.1) conditions. So it has MP, Progress & Bdy.

So, it is a good s/w based solution.

Busy Waiting: Situation when process is waiting in "while loops" while wasting CPU cycles instead of sleeping. Also know as "spinning".

All of above attempts do busy waiting & this is a kind of drawback.

Ques: Good

turn & flag are shared variables

P0:
```
while(1) {
    flag[0]=1;
    turn=1;
    while(flag[1]==1 && turn==1);
    <CS>
    flag[0]=0;
    <RS>
}
```

P1:
```
while(1) {
    flag[1]=1;
    turn=0;
    while(flag[0]==1 && turn==0)
    <CS>
    flag[1]=0;
    <RS>
}
```

PAGE NO.:
DATE: / /

a). If P_i is in CS, then value of shared variables:

→ flag[0] = Anything
flag[1] = 1
turn = Anything

b). If P_i is in RS, then:

→ flag[0] = Anything
flag[1] = 0
turn = Anything

c). If P_0 is in CS & P_1 is in RS, then:

→ flag[0] = 1
flag[1] = 0
turn = 1

⇒ All solution we have seen till now are s/w based solutions out of which only peterson solution worked but it has following disadvantages:

- Code are slower than hardware.
- Peterson sol" was for 2 processes. However we can generalise for n processes but n must be known in advance.
- Busy waiting i.e. wasting CPU cycles.
- Hard to convince that solution is correct because exponential no. of interleavings are possible.

for each processor, we have dedicated int

PAGE NO. :
DATE : / /

** Hardware Supported CSE Solutions:

Attempt 5: Disabling Interrupts

We can disable interrupts (when interrupts are disabled no interleaving happens) when process is in CS.

This is a working approach but it becomes inefficient when we are on a multiprocessor environment. Then either we have to disable interrupts of all processors which is a costly operation or process can go to other processor & inconsistency can come.

It is also unwise to give user processes the power to turn off interrupts. (what if one of them did it & never turned them on again?).

** OS + Hardware Supported CSE Solution

Attempt 6: Atomic Operations

Hardware itself provides some atomic operations to execute read-modify-write operations in one Assembly instruction.

These are H/w dependent. Some of them are common:

a) test_and_set (type *ptr): sets *ptr to 1 & returns previous value.

b) fetch_and_add (type *ptr, type val): adds val to *ptr & returns prev. value.

PAGE NO. :
DATE : / /

Not imp.

y. Compare-and-swap (type *ptr, type oldval, type newval):

If *ptr == oldval, set *ptr to newval & return True, else return false only.

→ In exam, proper definition of these instruction will be give, no need to bybeart these. (just keep in mind, common ones).

→ Remember our first attempt in S/w based CSE soln. It was unable to provid ME bcz both process could have busy = 1 if initially busy = 0 and preemption at line 1.

```
y) while (busy);
z)     busy = 1;
3)     count++;
y)     busy = 0;
```

→ We can remove this issue if we have some way to atomise line 1 & 2. And that can be done using TSL (Test & Set) lock.

→ TSL(x) will return 0 if x is 0
1 if x is 1
& set x to 1.

It may look like TSL Rx, LOCK as machine instruction in which lock's data is moved in register Rx and lock is set to non-zero value.

PAGE NO. :
DATE : / /

→ Now the implementation will look as:

```
while (test & set (&lock));
    //cse
    lock = false;
```

It satisfies ME + Progress.
But BW ✗ is not satisfied.

→ It is a good approach. Its advantages are:
- It is applicable to any no. of processes on single or multi processor environment.
- Simple & easy to verify.
- Multiple cse can be supported by their own variables.

→ But it's main disadvantage is, "Bounded waiting is not satisfied", which can lead to Starvation. And, similar to peterson solution it also have Busy waiting.

⇒ All of these problems can be solved using Semaphores.

Sphr ⇔ Semaphore

PAGE NO.:
DATE: / /

** Semaphores (Attempt-7) :- Given by dijkstra in 1965.

It can be considered as normal int variable which can be initialised, incremented & decremented (atomically & or w/o race condition) only.

There is no way to read the values of semaphores.

Two operations supported other than initialisation are:

- P() / Down() / ~~Signal~~ wait() → decrement by 1
- V() / Up() / Signal() → increment by 1.

Derived from words dutch: prober & verhoog.

→ There are mainly two types of semaphores, counting sphr (can take any integer value) and binary semaphore (can take 0 or 1). → Also called Mutex

→ Binary semaphores provides mutual exclusion & so also called ~~mutex~~.

→ Counting sphrs can represent a resource with ~~fixed~~ multiple instances. (Reading Writing Ex.)

→ P & V codewise works as:

```
P(S){
  while(S==0);
  S=0;
}
```

```
V(){
  S=1;
}
```

For Mutexes

PAGE NO.:
DATE: / /

(with CS)

Now code can be viewed as:

S = 1

P1:
P(S)
CS
V(S)

P2:
P(S)
CS
V(S)

→ Previous code of mutex suffers from busy waiting (CPU wastage). This can be avoided as:

```
P(S) {
    if (S==0) {
        put in queue;
        sleep();
    }
    S=0;
}
```

```
V(S) {
    if something in queue wake up it;
    S=1;
}
```

This is how actually mutex are implemented.

→ Similarly counting sphrs can be implemented as:

```
P(S) {
    S->val --;
    if (S->val < 0) {
        put in queue;
        sleep();
    }
}
```

```
V() {
    S->val ++;
    if (S->val <= 0) {
        remove from queue;
        wakeup();
    }
}
```

This will work for both type of sphrs.

Initial value of S->val determines max no. of process we allow. i.e. S->val = 1 for mutex always.

PAGE NO.:
DATE: / /

P.T.R.

→ P() basically allows only if s→val is non-zero. (If zero or negative then blocks). → in start → Not after decrementing

→ V() never blocks, It always values, but just increases s→val.

→ Positive Value means no. of res available now & Negative value means no. of threads waiting for resource.

→ Write P() before CS & V() ~~before~~ after CS & to initialise s→val with one for Mut.exclusion. (Binary Sphrs).

** Qus! We want stmt A2 from process A to complete before stmt B4 from process B begins. (A2 < B4). Implement using mutex.

A:	B:
A1;	B1;
A2;	B2;
A3;	B3;
A4;	B4;
A5;	B5;

wait() signal()

⇒ Just write P() before B4 & V() after A2 & initialise value with 0.

PAGE NO. :
DATE : / /

CS 92

Que: At a particular time of computation, the value of a counting semaphore is 7. Then 20 P & 15 V operation were done on semaphore. final value of sphr is:

⇒ 7 − 20 + 15 = 2

CS 93

Que: Shared variable 'k' is initialised to zero & operated on by four concurrent processes w, x, y, z as follows: Before x reads 'k' from the memory, it increments by 1. S is initialised to two. Maximum value of 'k' possible is?

⇒

w	x	y	z
P(S)	P(S)	P(S)	P(S)
Read k	Read k	Read k	Read k
k++;	k++;	k−=2;	k−=2;
write k	write k	write k	write k
V(S)	V(S)	V(S)	V(S)

(If S was initialised by 1 then mutex was available & no consistency so min = max = −2)

As S is initialised by two, two process can enter in CSe at same time.

So max = 2
↳ overwrite Y & Z by w & x

min = −4
↳ overwrite w & x by Y & Z.

PAGE NO.:
DATE: / /

Question: fill blanks, i/p desired is ABABAB...

P:

```
while(1){
    //blank1
    print(A);
    //blank2
}
```

Q:

```
while(1){
    //blank3
    print(B);
    //blank4
}
```

→ We will have two sphrs S & T, S=0, T=1

blank1 → P(T)
blank2 → V(S)
blank3 → P(S)
blank4 → V(T)

CS 10

Que: Binary sphrs initialised as S0=1, S1=0, S2=0;

P0:

```
while(true){
    wait(S0);
    print(0)
    release(S1);
    release(S2);
}
```

P1:

```
wait(S1);
release(S0);
```

P2:

```
wait(S2);
release(S0);
```

How many times will process P0 print '0'?

→ It has to print once w/o problem, then it can print one or two more times on the basis of scheduling of P1 & P2. So **Atleast twice.**

↳ Ans.

(It would be **exact thrice**, if it was counting sphr instead of binary with same initialisation)

PAGE NO. :
DATE : / /

→ Note in prev. question only P_0 was surrounded by while loop. If all were surrounded then 0 printed 90 times.

CS 97

Que: Each process P_i, $i = 1 \ldots 9$ is coded as

```
while(T) {
        P(mutex)
            CS
        V(mutex)
}
```

The code for P_{10} is:

```
while(T) {
        V(mutex)
            CS
        V(mutex)
}
```

Maximum no. of processes that can be inside CSe at any moment?

⇒ All of them can be there because P_{10} is surrounded by while loop. So despite binary sphr, all 10 can be in C.S. at same time.

(If while was not around P_{10} & sphr is initialised by one then atmost 3 can be there at same time and one of them was P_{10}).

PAGE NO.:
DATE: / /

CS 2003

Que: Suppose we want to synchronise two concurrent processes P & Q using binary semaphores S & T. The code for P & Q is:

P:
```
while(1) {
  //w
    print('0');
    print('0');
  //x
}
```

Q:
```
while(1) {
  //y
    print('1');
    print('1');
  //z
}
```

What can we fill at w, x, y & z, such that it will ensure that the o/p string never contains a substring of the form 01^n0 & 10^n1, where n is odd?

⇒ Basically we want consistency, no interleaving should happen. Only interleaving can print above format string? This can be done by:

w: P(S) y: P(S)

x: V(S) z: V(S)

S initially 1 & T initially anything.

Que: Which of the following can lead to deadlock, mutex a, b = 1

a)

P_0	P_1
P(a)	P(a)
P(b)	P(b)
CS	CS
V(a)	V(a)
V(b)	V(b)

b)

P_0	P_1
P(a)	P(b)
P(b)	P(a)
CS	CS
V(a)	V(a)
V(b)	V(b)

PAGE NO. :
DATE : / /

P_0:	P_1:
P(a)	P(b)
P(b)	P(a)
CS	CS
V(a)	V(b)
V(b)	V(a)

c)

P_0:	P_1:
P(a)	P(b)
P(b)	P(a)
CS	CS
V(a)	V(b)
V(b)	V(a)

d)

⇒ b & c deadly.

(V's don't matter in deadlock)

Gate13
Que: Tough (DIY once) (Lecture - 20 OS)

⇒ We need to make the sphrs P() & V() code atomic too, else we are again stuck if any interleaving happens. We do this using TS lock H/w instruction. Then we have busy waiting but we already know P() & V() are small codes so much CPU is not wasted. Issue arises when busy waiting causes too much time waste & other process who is exec. CS is taking too much time. Here too much time is not taken. So busy waiting is there but minimal.

↳ Bcz we use TSL to sync. P() & V().

→ To sphrs are better than usual TSL instruction bcz TSL can be too long but sphr wastes CPU for very short operations.

Basically by using sphrs instead of TSL, we have shifted busy waiting from CSe code to fixed small code.

PAGE NO. :
DATE : / /

Classic Synchronisation Problems: There are various classic problems that represent a class of synchronisation situations. Once we know the "generic" solution to these problem templelate, we can use them according to our future needs. We have mainly three problems in our syllabus.

a). Producer Consumer Problem: In this we have:

- → Multiple producer threads
- → Multiple consumer threads
- → All threads modify same buffer.
- → No production when buffer is full.
- → No consumption when buffer is empty.
- → Only one thread, be it producer or consumer, one thread can only modify the buffer (cse) at any time.

Producer:

gen. item locally

fill buffer with item

Consumer:

get item from buffer

use it locally.

These do access buffer so, it's a cse.

→ We can cover CSe with P & V of a mutex but, overflow & underflow conditions are not fulfilled then, they also needed to be handled.

PAGE NO. :
DATE : / /

→ To solve this let's have two counting sphrs empty (represents free space for items in buffer), & full (represents full space for items in buffer). by

→ Initial value of empty sphr is N & that of full smphr is 0, (N is size of buffer).

Final Implementation with all conditions satisfied is:

Producer:

```
do{
    /* produce an item locally */
(1)* P(empty);
(2)* P(mutex);
    /* CSe i.e. update item to buffer */
(3)* V(mutex);
(4)* V(full);
} while(true);
```

Consumer:

```
do{
(5)* P(full);
(6)* P(mutex);
    /* CSe ; i.e- cut item from buffer */
(7)* V(mutex);
(8)* V(empty); → /* Consume item locally */
} while(true);
```

PAGE NO. :
DATE : / /

Here we have three sphs:

Sphr Name.	Type	Initial Value
empty	Counting	N
full	Counting	0
mutex	Binary	1

Qu: What if we no full sphr?
→ Underflow condition i.e. Consumer will consume even if buffer is empty.

Qu: What if we don't have empty sphr?
→ Overflow i.e. Producer will keep on producing even if buffer is full.

Qu: What if we swap 1↔2? (or 5↔6.) LOC.
⇒ It may lead to deadlock if buffer is full (i.e. empty=0) or buffer is empty (full=0)

Qu: What if we swap 3↔4 (or 7↔8) LOC.
⇒ No problem at all.

PYQs
Gate IT 08, IT 04, IT 06, CS 99, CS 2002, CS 14 (Set 2)

DIY.

PAGE NO. :
DATE : / /

b) **Reader - Writer Problem** : In this we have:

→ A reader reads the data but won't change it.
→ A writer can change the data.
→ Multiple reader & writer threads.
→ Allow many readers at a time but 0 writer.
→ Allow single writer at a time & 0 readers i.e. if writer is in cse then no other writer and no reader. and if reader is in cse, then no writer only readers are allowed.

⇒ If we surround cse by P & V of single mutex, then all condition are achieved except that multiple readers are not allowed at a time.

⇒ If we have mutexes only on writer then all conditions are satisfied except that 1 writer & readers are in cse at same time.

⇒ To solve this problem we can stop writer when first reader goes in & allow it when last reader comes out.
We can use counter for it.

⇒ It can be achieved by following code:

PAGE NO.:
DATE: / /

binary sph mutex = 1;
binary sph wrt = 1;
int readcount = 0;

writer:

```
do{
/*1*/  p(wrt);
/*2*/  /* writer cse */
/*3*/  V(wrt);
}while(1);
```

reader:

```
do{
/*4*/  p(mutex);
/*5*/  readcount++;
/*6*/  if(readcount==1)
/*7*/      p(wrt);
/*8*/  V(mutex);
/*9*/  /*reader cse*/
/*10*/ P(mutex);
/*11*/ readcount--;
/*12*/ if(readcount==0)
/*13*/     V(wrt);
/*14*/ V(mutex);
}while(true);
```

first reader

Last Reader

→ We need to protect readcount else inconsistency may create because there are multiple readers

Question: What if we swap following lines:

a) **6 & 7 outside mutex protection**

→ Preemption just after V(mutex) may never make readcount as 1, it will go from 0 to 2 for if condition & writer are not blocked despite reader being inside code. However, **NO deadlock only inconsistency.**

PAGE NO. :
DATE : / /

b) 12 & 13 line outside P or V of mutex.

⇒ ~~It can be possible that readcount goes negative bc~~

⇒ It can be possible that many readers are prompted just ~~after~~ before if condition & after V(mutex), then many readers will have readcount = 0 & then many writers can come (if semaphore is not binary), bcz. wrt will be more than one as many V's will be executed.

⇒ There can be situation that no reader execute this if and writer remains blocked and deadlock may occur.

How?

↳ 3 readers are preempted just befor if & readcount was zero.

Now new reader enters & updates readcount to one. But it can't execute line 7 i.e P(writ) bcz. wrt is already zero.

Now neither readers can read nor writer can write and deadlock.

c) "readcount++" outside the mutex protection i.e line '5' before line '4'.

⇒ Deadlock can happen.

Assume that last reader (when readcount is 0) gets preempted just after readcount--;

PAGE NO. :
DATE : / /

Writer cannot perform bcz we have not done V(wrt)
Now assume many readers do readcount++ sing it's outside protection le value of readcount will be 1 due to race condition but many readers.
Now the last reader is scheduled again le since readcount is 1 (it will finish w/o V(wrt).
Now any reader is scheduled, it will stuck at P(wrt) bcz readcount is 1.

Now all readers are stucked at P(mutex) bcz one reader is inside.
And, so deadlock.

★ c) Dining Philosopher Problem: In this we have.

→ N philosophers on a round table.
→ one chopstick b/w each philosopher.
→ In order to eat we need two chopsticks.
→ Philosopher can pick chopstic 1 by 1, not both at once.
→ once chopstick grabbed, it will not be released before eating.

- We need a solution which is deadlock free & starvation free.

fork ⇔ chopstick
↳ used interchangeably

PAGE NO. :
DATE : / /

a)

Attempt 1 Naive Solution:

for all Philosophers P_i.

P_i :

grab left fork;
grab right fork;
eat();
release right fork;
release left fork;

⇒ It may cause deadlock if everyone gets prempted just after grabbing left fork.
↳ Everyone will have one only & wait for another, but no one will release, & so deadlock due to circular wait.

So, to get this done we have to break hold & wait, so it is done in following two ways: (Attempt 2 & Attempt 3)

Attempt 2 1. Impaster (knowledgeable) Philosophy.

In this All except one grabs left then right but one of them will grab right then left.

P_i $\forall i-1 i=n$ (All except last)	P_n (last)
grab left;	grab right;
grab right;	grab left;
eat();	eat();
release right;	release left;
release left;	release right

PAGE NO.:
DATE: / /

Now, we can never have deadlock.

Attempt 3: Both or none

Allow philosophers to grab chopsticks only if both are free. (This need CSe protection to test if free before grabbing them)

Attempt 4: Allow only N-1 philosophers for N forks.
(or buy 1 more forks for N philosopher).

Attempt 2, 3 & 4 are deadlock free. & starvation free.

⇒ Important Terms.

BW ↛ Progress
Progress ↛ Bounded waiting

So, both BW & progress are independent.

Progress → No deadlock
No deadlock ↛ Progress.

Progress & deadlock are related.

BW & deadlock are not related.

→ Starvation free means if a process is trying to enter its CSe it will eventually enter. It's turn will come bakka.
Progress + BW imply starvation free.
(Above two can't imply)

PAGE NO. :
DATE : / /

Module-3

Memory Management (Another Unit)

L-23

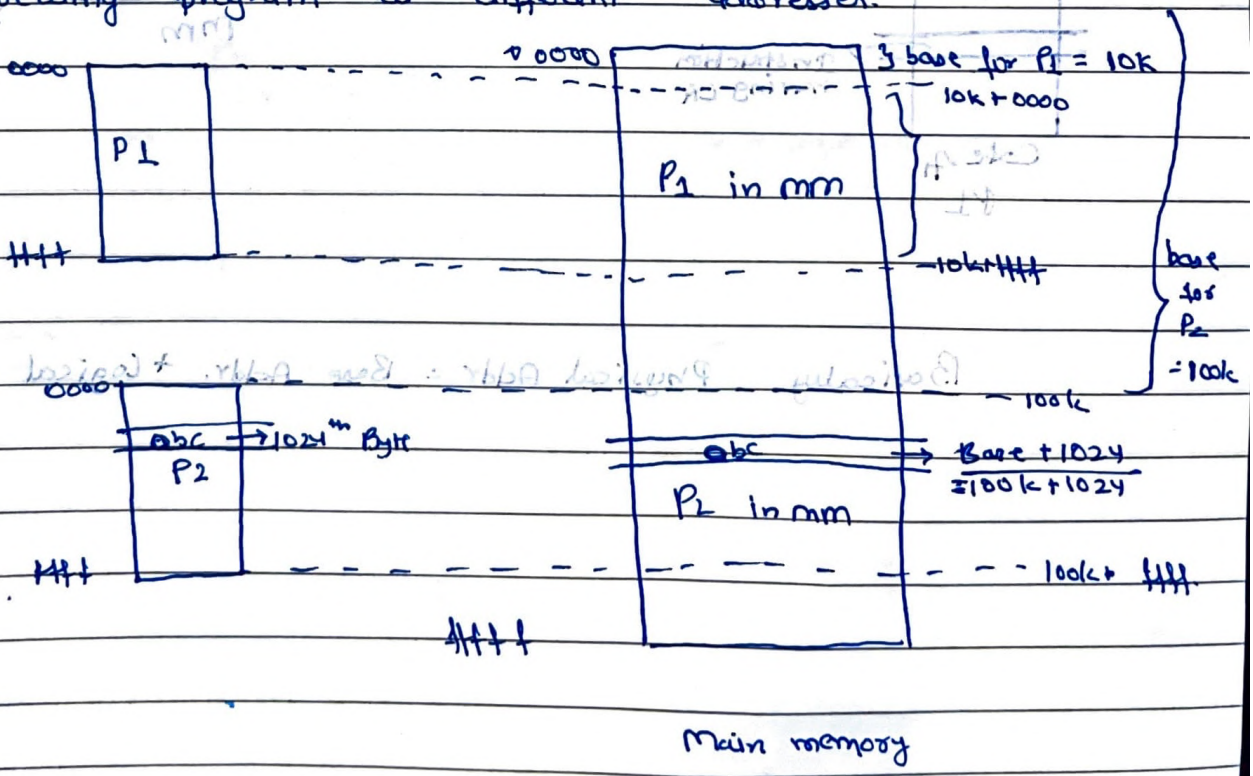
→ We already know that while the program is in compilation phase, it do not know about physical memory. It assumes its own space from 0 to x Bytes and assumes this whole memory is for it only. But in actual, many processes reside in memory.
(more details on this in Program Journey lecture).

→ If we load the compiled process with these virtual addressed in the instruction (starting from zero), then we can not have 2 programs at same time in main memory.

→ Also, we can not ask compiler to directly generate physical address because it do not know after how many decades you will run it & at that time which part of memory is available.

→ One of the way to achieve this is by relocating program at different addresses.

Ex:

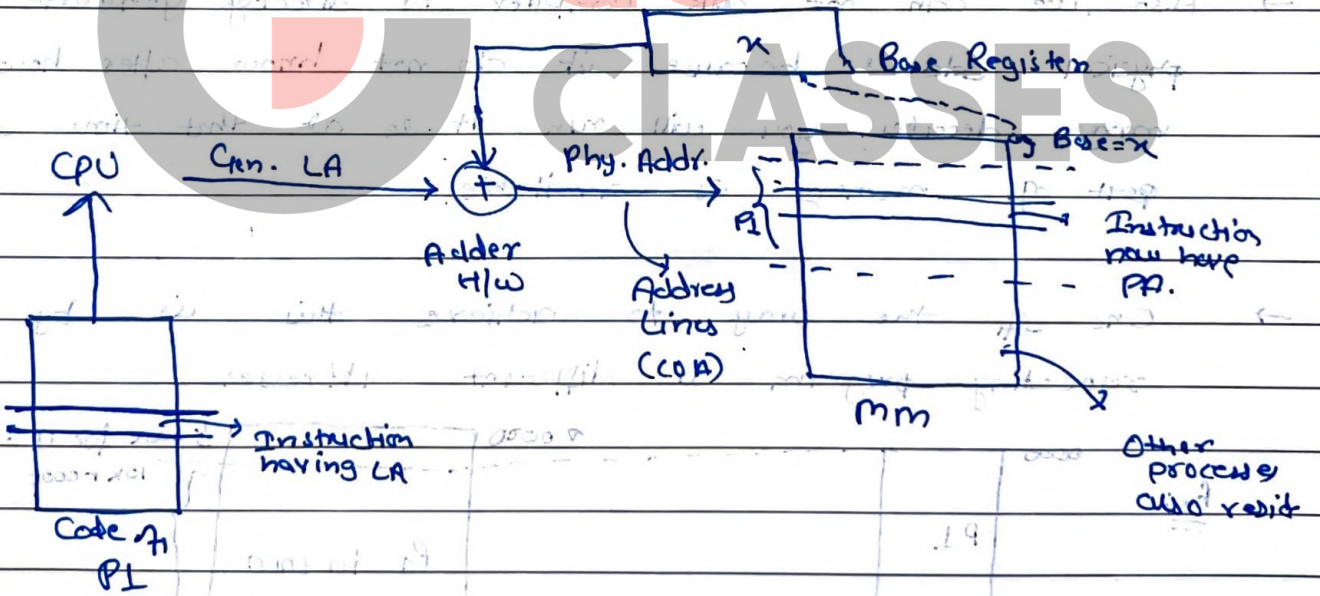


PAGE NO. :
DATE : / /

Here we assume:

→ Process is contiguously stored in main memory.
→ Process start from base address
→ Each process has its own base address.
→ Base address of currently scheduled process is in H/w register called based register. (Stored in PCB when not scheduled and reloaded in register on context switch like normal registers).

⇒ Address Translation for this Base Setup:



Basically Physical Addr = Base Addr. + Logical Addr.

PAGE NO. :
DATE : / /

⇒ Logical v/s Physical Address :

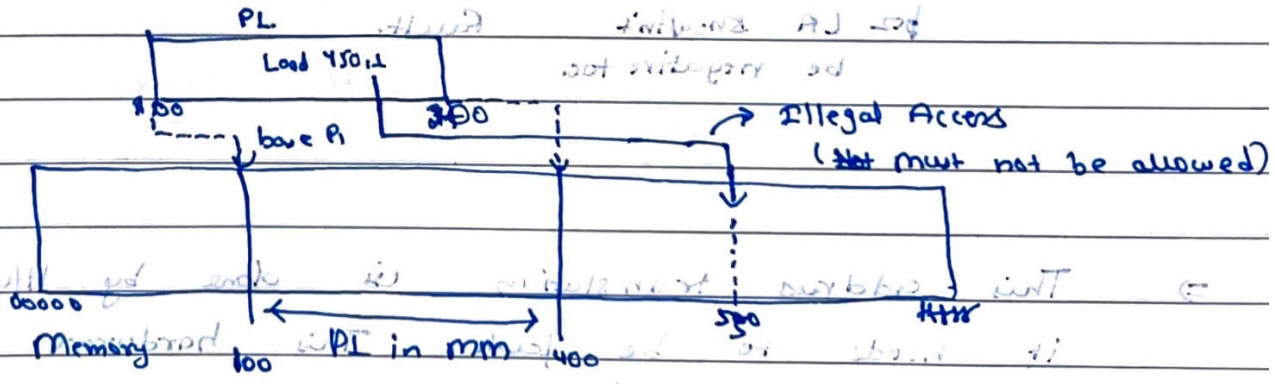
→ Logical Address is Generated by CPU & programmer sees this only (printf prints this only) ; he never see/ read physical address.

→ Physical Address is address sent to RAM (or IO) for R/W operation.

NOTE : for two diff programs in mm at the same time, logically there address could be same, but there Physical address will always be diff.

⇒ Drawback of this Base Register setup is process can generate some illegal address and may do some opn in memory of other process which it shouldn't be allowed to.

for Ex:



[ ] ( )
Incl. Excl.

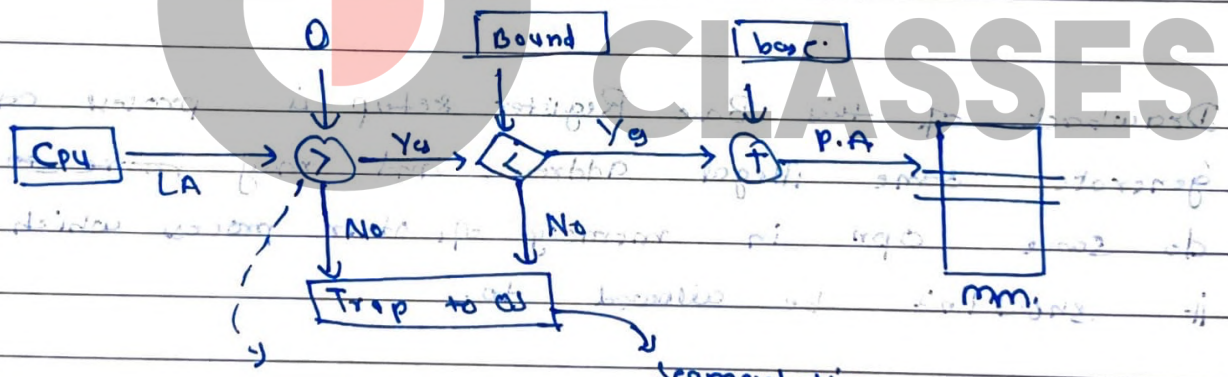
PAGE NO. :
DATE : / /

→ So to overcome this problem, we come up with another hardware register "Bound". also known as Limit Register.

→ Now a process is allowed to access physical memory in [base, base + bound).

→ Bound contains size of the process. So now,
LA space is = [0, bound-1] or [0, bound)
PA space is = [base, base + bound - 1] or [base, base + bound)

→ Address Translation in Base Bound setup:



⇒ This address translation is done by H/w bcz it needs to be fast. This hardware is known as Memory Management Unit (MMU).

PAGE NO.:
DATE: / /

→ Advantages of Base register approach:
- Dynamic Relocation
- Simple & Inexpensive & fast H/w implementation.
  - → Little logic
  - → Few Registers.
- Simple context switching (Save & Restore few registers).

→ Advantages of Base + Bound:
- All of Base Register.
- Provides Protection of Address Space.
  - → Not in Base Reg. Approach

→ Disadvantages of Base + Bound:
- We have to move whole process in MM (Resolved by demand paging).
- External fragmentation (told 4 pages later)
- Each process must be loaded contiguously in physical memory. (Solved by Segmentation).

★ Segmentation: Here, we will divide process into some chunks (of any size) & store chunks independently in MM. (Not in hard disk).

→ Segment is variable sized area of memory. It's just natural extension of base & bound, there we just had 1 segment per process. Here we have multiple segments for each process, & one pair of base & bound for each segment.

PAGE NO. :
DATE : / /

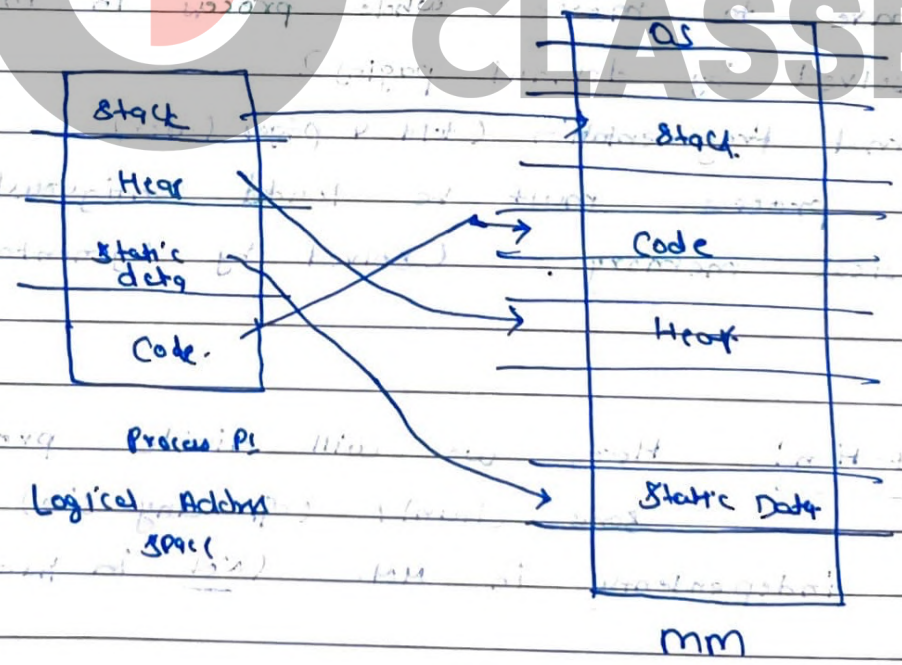
→ We have one segment map per process which contains values of base & bound, permission & other seg. wise detail for each segment.

→ On context switch, values of base & bound are restored from this table present in kernel.

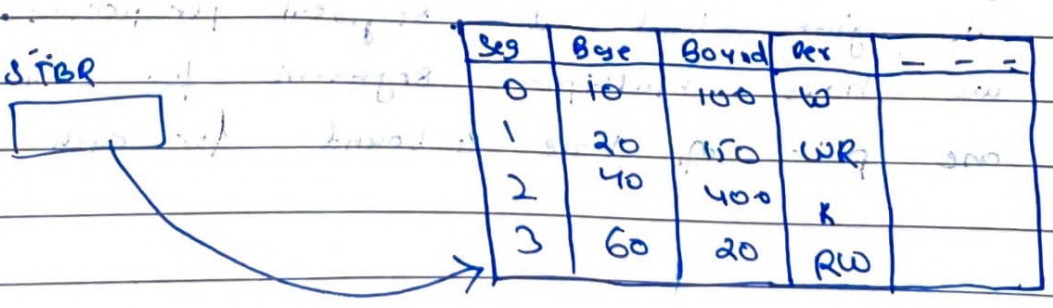
→ We have segment table base register which holds address of this segment map of current process

→ We can form segment of any size, but it's done logically like one seg. for code, one for stack etc. but it's not hard & fast rule to do so.

Ex:



So, process stored non contiguously in MM.



Offset: Location in Segment / Page.

→ After Reaching that segment, where req. data?

→ Remains same always.

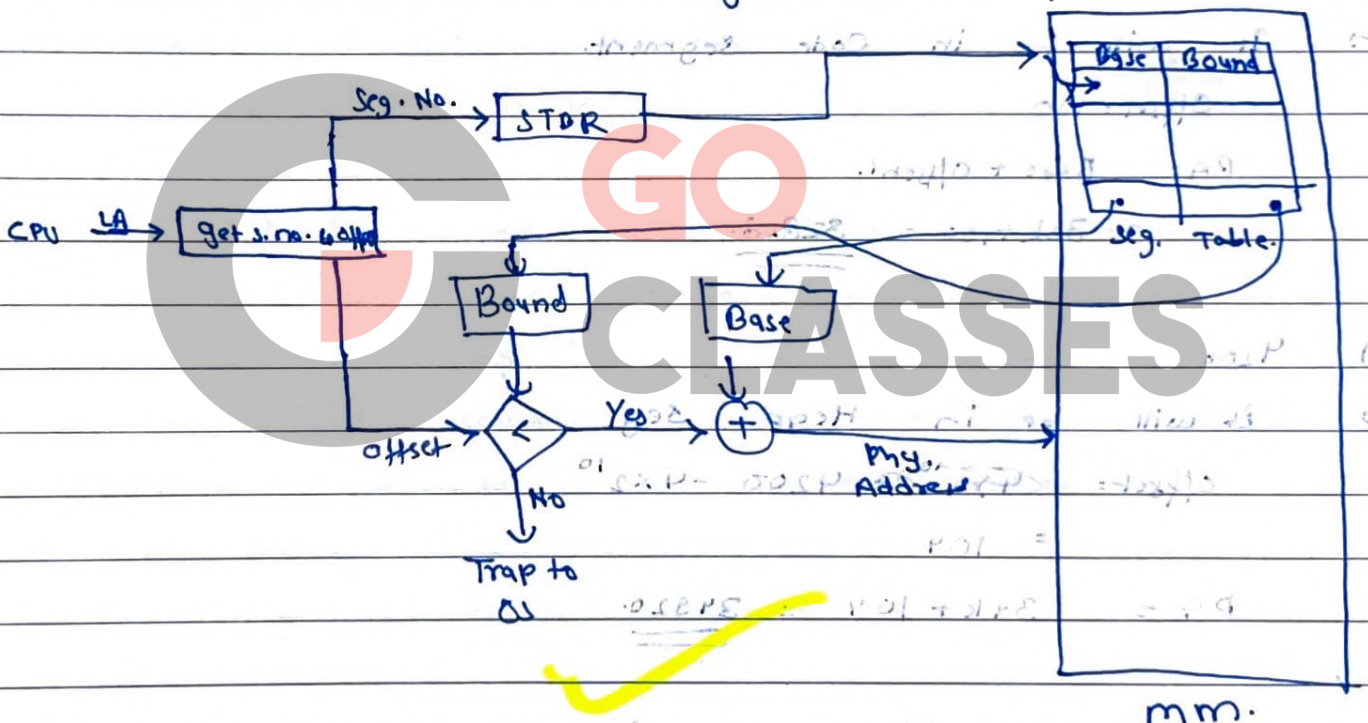
→ In LAS if we have 2^n segments then Most sign. n bits represent the segment no. and remaing total - n represents offset.

Ex: LA: 01 00011011

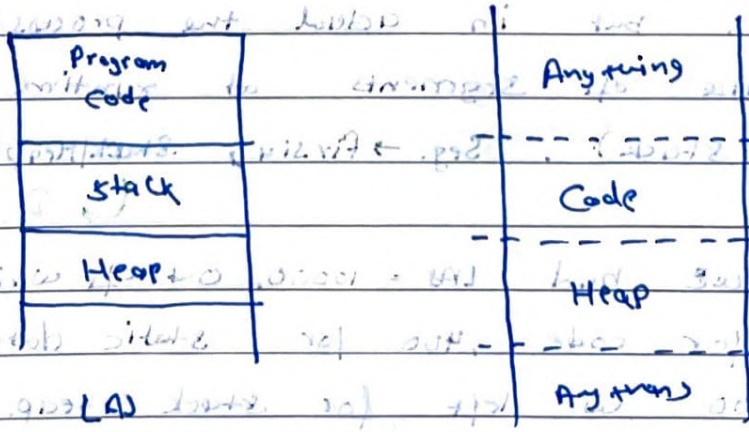
01 → Seg No.

00011011 → within Seg., how much to go?

→ Address Translation in Segmentation:



Ex:



PAGE NO.:
DATE: / /

Seg. Table ⇒:

Segment	Base	Size
Code	32K	2K
Heap	34K	2K

Find Out PA corresponding to LA:

a) 100

→ It will be in Code segment.

Offset = 100

PA = Base + Offset.

= 32K + 100 = 32868.

b) 4200

→ It will be in Heap Seg.

Offset = $4200 - 4 \times 2^{10}$

= 104

PA = 34K + 104 = 34920.

c) In Segmentation we fix the size of segments created once, but in actual the process may have diff size of segments at runtime (like heap of stack). Seg. → fix size, Stack/Heap → Not fixed.

↳ Problem :

Ex: Assume we had LAS = 10000, out of which 600 is for code, 400 for static data. Now 9000 is left for stack & heap which may or may not be required.

PAGE NO. :
DATE : / /

Now how do we split segments :

a) Give 4500 - 4500 to both, then may be stack need 6000 & heap need 3000.
OR. we are also wasting mm if they do not require that much.

b) Give 500 - 500 to both, but we have already promised programmer we have 9k space for stack + heap

c) Put stack & heap at same place, this again boils down to old problem of contiguous memory req.

d) Give fix size and then increase segment → if mm is available then update bound → if mm not there, then swap to diff loc. and update base & bound.
This solution seems to be working but relocating or swapping entire segment takes time.

⇒ External fragmentation (fitration) when space is avlbl in mm but not contiguously.
(Problem in Base & Bound, Segmentation bound)

Ex!

we have 90 KB, but still can't bring seg. in memory.

PAGE NO. :
DATE : / /

One solution for External fragmentation is to move all used space continuously, but this is inefficient.

So, External frag. & fix size of seg. are problems that make segmentation inconvenient.

→ In base & bound we have no way to avoid unnecessary unused space (alloted to stack & heap) to be loaded in mm. We load entire process there.

★ Practice on Base & Bound Segmentation:

Que: No. of base registers in system with 1 CPU?
⇒ 1. (4 for 4 CPUs)

Que: Base = 1000, Bound = 100. # memory accesses for instruction "LOAD 10, R1".
⇒ 2 (one for fetching instr & one for loading value at address 10).

Que: VA = 10, Base = 1000, Bound = 100, PA?
⇒ 1010.

Que: If bound = 100, then # legal addresses?
⇒ 100 (0 to 99).

PAGE NO.:
DATE: / /

Que: Let's have following process table:

[Process: A; base: 100; bound: 10]
[Process: B; base: 1000; bound: 20]
[Process: C; base: 500; bound: 50]

a) Legal PAs for process B:

⇒ 1000 to 1019.

b) When switching b/w Process A to B, os forget to update bound register, but updates base register correctly, then:

⇒ B could only access 1000 to 1009, but should be allowed from 1000 to 1019 so might fault unexpectedly.

Que: Consider following segmt. table:

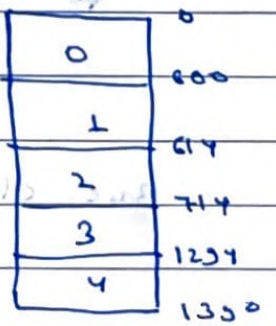
Seg.	Base	Limit
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

a) LA=1000, PA=?

⇒ This would be in Seg 3.

Offset = 1000 - 714
= 286

PA = 1327 + 286
⇒ 1613



PAGE NO. :
DATE : / /

b) PA = 1375 ; LA = ?

⇒ It would be in seg 3
as it ranges from 1327 to 1327 + 500 - 1

Offset = 1375 - 1327

⇒ 48

LA = 714 + 48

⇒ 762

c) LA = 2400 ; PA = ?

⇒ 2400 is beyond CAS, so Addressing Error.

Que: Consider following LA → PA

0: 776 → 14611

1: 597 → 14432

2: 929 → Seg. Violation.

* what can be said about base & bound.

⇒ from 2: , bound ≤ 929 (Equal bcz if bound is 929 then address = 0 to 928 only).

from 0: bound ≥ 776

so 776 < bound ≤ 929

LAS < 929

Base clearly is 14611 - 776 = 13835

OR 14432 - 597 = 13835

PAGE NO. :
DATE : / /

Qu: Assume following simple seg. system that supports two segments: one (+ive growing) for code & heap, and one (-ive growing) for stack.
LAS = 128 B (7 bit), PAS = 512 B.

	Seg.	Base	Bound
Positively Grows	0	0	20
Negatively Grows	1	512	20

→ Which of the following are valid LA?

a) 29 → No.
b) 123 → Yes
c) 16 → Yes
d) 90 → No.
e) 508 → No. (Yes if it was PA)

LA 0 ... 19 are valid &
127 ... 108 are valid (Last 20 bytes of LAS).
Here it do not matter where we map bcz LA is given

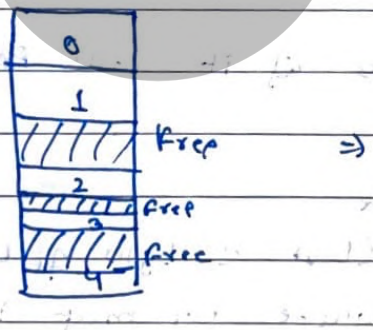
PAGE NO. :
DATE : / /

☆ Paging:

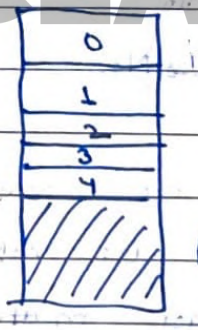
→ free space mgmt., while allocating a new seg. was a problem in segmentation, (First fit or Best fit). Another problem was external fragmentation obviously.

→ What if we keep all the segments to be equal sizes? ⇒ Paging.
↳ No external frag. will be there.

→ External fragmentation needs to be avoided bcz it leads to either space wastage or time wastage.
Compaction: → Relocation of free parts ↙
will save space but waste time.



⇒



Compaction
↳ Saves space but time consuming.

→ Paging is not really a new idea, it is somewhat similar to our books where we have page, Page number, index etc.
Analogy

PAGE NO.:
DATE: / /

→ It solves the Ext. frag. by using fix size units in both physical and logical memory.

→ It solves internal frag. by keeping this unit small.

→ It is modern technique and used in current computers.

→ LAS is divided in pages & PAS is divided in frams

& Page Size = Frame Size.

Ex:

LAS MM (PAS)

Then page table will be an array looking like

1	11	9	311
0	1	2	3

→ Each process has its own page table.

Ques: LAS = 1024 B.

Pg. Size = 8 B

PA LA = 90

PA = ?

Page	Frame
0	3
1	6
!	!
!	!
113	10
!	!
!	!
127	!

⇒ No. of pages = $\frac{1024}{8}$ ⇒ 128 (0 to 127)

// → Integer Div

%. → Modular

PAGE NO. :
DATE : / /

→ 905 LA will be in page
905 // 8 = 113th page.
& offset will be 905 %. 8 = 1 B.

Now 113th pg is in 10th frame
So PA = (10 × 8) + offset
= 81 (82 for 906)

Note that for LA 904, 905 ... 911, page no. will be 113.

* Address Translation:
So. Page No. is LA // Page size
& then offset is LA %. Page Size
Now from pg. table get Frame No corresponding to pg. No.
PA = (Frame No. * Pg. size) + offset

⇒ Some maths matical things:

1KB ⇒ 2^{10} B
1MB ⇒ 2^{20} B
1GB ⇒ 2^{30} B
1TB ⇒ 2^{40} B
1b ⇒ 8 bits

{ 10^3 B, 10^6 B, 10^9 B } In CN, while measure data speed

B → Byte
b → bits.

PAGE NO. :
DATE : / /

Que: LA = 807 B. , Pg. size = 8 B , LAS = 1024 B.
Frame No (100) = 3.

⇒ 807 = (100 * 8) + 7
= So 7th byte in 100th page
100th page in 3rd frame
So, (3 * 8) + 7
⇒ 31

⇒ In decimal.
1234 // 10 ⇒ 123 , 1234 % 10 ⇒ 4
1234 // 100 ⇒ 12 , 1234 % 100 = 34

Similarly in Binary (Not Intuitive, but Compare)
11001 // 2 ⇒ 1100 , 1100 % 2 ⇒ 1
11001 // 2^2 ⇒ 110 11001 % 2^2 ⇒ 01

⇒ So dividing n bit number with 2^p will give p bits for offset and 'n-p' bits for page no.

* Imp.

← n-p →	← p →
Pg. No.	offset

n bit LA

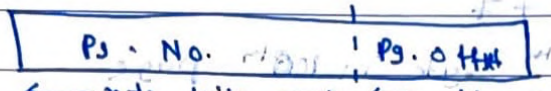
pg. size = 2^p

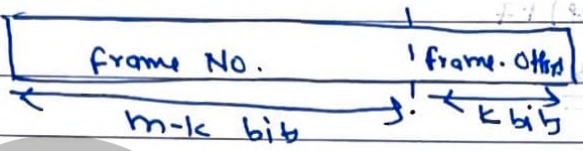
→ Also if LA is of n bits then LAS = 2^n Bytes (if Byte Addressable).

PAGE NO.:
DATE: / /

P.T.R:

⇒ LAS = n bit, PAS = m bit, Pg. size = 2^k B





Note that offset bits remain same.
↳ bcz page size = frame size

Qu: VA = 8 bits, PA = 10 bit, Page size = 2^6 B.

a) # Pages = $2^8 / 2^6 \Rightarrow 2^2$ Pages

b) # frames = $2^{10} / 2^6 \Rightarrow 2^4$ Frames

c) # entries in page table = # pages = $2^2 = 4$

d) Given page table = [2, 5, 1, 8], VA = 241
PA = ?

⇒ Pg. No: $241 / 2^6 = 3$
Offset: 241 % 64 = 49
PA: (8 * 64) + 49 ⇒ 561
(8 = frame No from pg. table)

Using Binary

241 = 11 110001
(Pg no. | offset)
= 1000 110001 = P.A
561

Note that pg. table is already ...

PAGE NO. :
DATE : / /

Using paging, we have solved problem of space mgmt, while allocating a new chunk.

But this single Level paging (seen till now) has another issue. Consider Ex:

Pg size = 4KB, 32 bit LA.

So 2^{20} pages.

and so 2^{20} PTEs; if each PTE is 4B,

page table size = 4MB.

↳ Huge.

This is for one process. May be process is of 4B as of now, remaining space is wasted b/w stack and heap. Still pg table is 4MB. So we have huge pg table even for small process (compiler assumes huge LAS). We cannot afford huge table for every process. So we have to reduce Pg. Table size.

This is done by Multi-Level Page Tables.

→ Understant Multi Level Paging then we will see how it's saving space.

PAGE NO. :
DATE : / /

→ No. of bits in LA ≠ No. of bits in PA.

However, offset remains same in both.

→ In page table, along with frame no, we may have other info. also:

Ex: • Valid bit = Present bit • Protection bit
• Present bit • Ref. bit ≡ Accessed bit
• Dirty bit = Mfd. bit (will taught later).

So, crux is PTE ≠ bits req. for frame No.
↳ This is minimum req.

Practice Single Level Paging

Ques: LA = 256 KB ⇒ 2^{18} B.
PA = 64 KB ⇒ 2^{16} B.
Ps size = 4 KB ⇒ 2^{12} B.

a) No. of pgs = $2^{18} / 2^{12} = 2^6$

b) No. of bits in pg. offset = 12.

c) No. of frams = $2^{16} / 2^{12} = 2^4$

d) PTE = 2^P B, Size of pg. table = $2^P \times 2^6 = 2^{6+P}$ B.

PAGE NO.:
DATE: / /

Que: LAS = 1TB = 2^{40} B.
#page = 2^{12}
#frame = 2^{10}

a) PAS = ?

⇒ pg. size = $2^{40} / 2^{12} = 2^{28}$.
PAS = frame × pg. size
= $2^{10} \times 2^{28}$ ⇒ 2^{38} B.

Que: Consider following pg. table.

0	1	2	3	4
3	10	9	2	0

Pg size = 1024 B

a) VA = 697, PA = ?
⇒ 697 is offset pg. no. 0
⇒ (3 × 1024) + 697 = 3769

b) VA = 1024, PA = ?
⇒ 0 is offset pg no 1.
= (10 × 1024) + 0 = 10240

c) PA = 2075, LA = ?
⇒ 2nd frame and 27
So 2nd frame has 3rd page (Pg. table)
= (3 * 1024) + 27
⇒ 3099

(This type of que, is mostly asked in binary, so people are used to of that only).

PAGE NO. :
DATE : / /

→ Even though VA of two d/f processes (Not sharing anything) can be same, but PA will be surely d/f bc2. same page no. will be mapped to d/f frames.

Que: Consider a single level paging system with 12 bit VA, 24-bit PA & Pg. size = 2^8.
Max. No. of page table entries possible.

⇒ 2^4 (bc2 there are # pages).

Que: VAS = PAS = 32 bits long.
Pg Size = 2^{12} B.
Pg. Table is:

0	1	2	3	4
0x00788	0x00249	0x00237	0x000ace	0x00bcd

Frame No. in Hexadecimal.

a) LA = 0x00001a60 , PA = ?

⇒ Pg. No. : 20 bits of left = 5 char. in hexadecimal
= 00001 = 1.
So 1st page in 0x00249th frame
& offset remains same.
PA = 0x00249a60.

b) LA: 0x00005ffH. PA = ?

⇒ Pg. no = 00005 → Frame No. not Given

Translation not possible with given info.

0x 09ce ≡ 0x009ce = Any no. of zeroes

PAGE NO.
DATE : / /

c) PA = 0x009ce H/c , LA = ?

⇒ 3 char are for offset

↳ remaining show frame no. (in PA)

So frame → 0x009ce

↳ It hold 3rd page.

So VA = 0x003CE

d) 0x000008d7

⇒ frame no = 0

we don't know which pg. 0th frame holds

∴ TRANSLATION NOT POSSIBLE.

Gate 2001:

Ques: VAS = 32 bit, MM = 64MB, Pg. Sz = 4KB

Pg table = ?

⇒ # PTE = $2^{32} / 2^{12} \Rightarrow 2^{20}$ = # page

frame = $2^{26} / 2^{12} \Rightarrow 2^{14}$

PTE size = 14 bits

Page table size = # PTE * PTE Sz

$= 14 * 2^{20}$

$\Rightarrow 14 * 2^{20}$ bits $\simeq$ 2 MB.

PAGE NO. :
DATE : / /

Gate 2016

Que: VAS :- 40 bits, Pg size = 16 KB ⇒ 2^{14} B.
PTE = 48 bits. , Pg Table size = ?

⇒ # PTE = $2^{40}/2^{14} \Rightarrow 2^{26}$ Page Table Entries.
P.T.S = $48 * 2^{26}$ bits.
⇒ $3 * 16 * 2^{26}$ bits
⇒ = 384 MB.

Gate 2017 set 2. Que 47

Que: D.I.Y.

As we observed previously that page table is too large for each process to be accomodated in MM. So we come up with multilevel paging.

(Page tables were large because we had too many invalid pages (empty) to maintain,)

☆ Multi Level Paging :

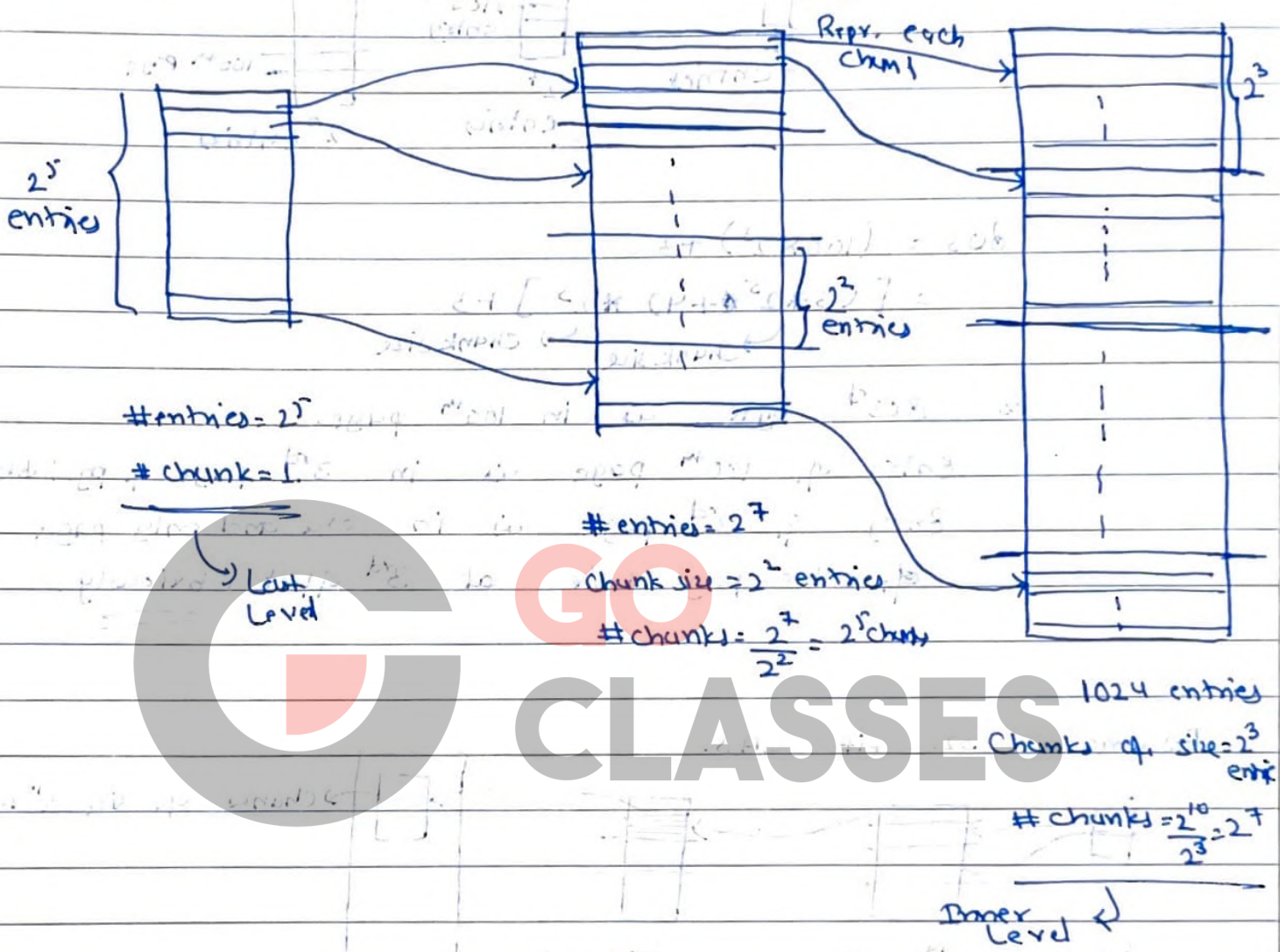
⇒ Why this ?

→ We may not find continuous space to store page table in main memory.

→ We have seen most of the pages are empty, then why to store entries corresponding to pages which are empty.

PAGE NO. :
DATE : / /

* A generic repr. of Mult Level paging.



→ If we consider Inner two levels only, then this is exactly equal to single level paging where Inner most level is LAS & next level contains frame no. of pages of LAS.

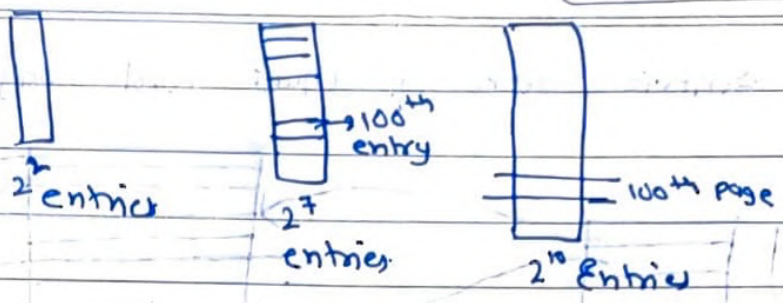
Now if we divide this page table also in pages of specific size than next level contains page frame no. for corresponding page of page table.

And we can go on like this until we remain with single chunk.

Entries at inner most level repr. [illegible]

PAGE NO. :
DATE : / /

Ex:



$803 = (100 \times 2^3) + 3$

$= [(3 * 2^5 * + 4) * 2^3] + 3$

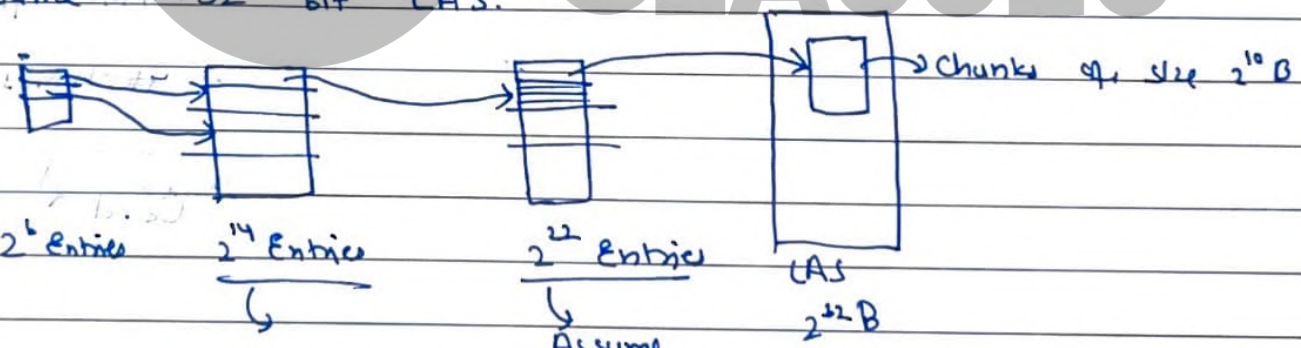
↳ Chunk size ↳ Chunk size

So 803^{rd} byte is in 100^{th} page.

Entry of 100^{th} page is in 3^{rd} page of pg. table

Entry of 3^{rd} page is in one and only page of outer page table at 3^{rd} offset Obviously.

Assume 32 bit LAS.



Level 0 Level 1 Level 2 Level 3

	0	1	2	3
#chunks	1	$2^{14}/2^8 = 2^6$	$2^{22}/2^8 = 2^{14}$	$2^{32}/2^{10} = 2^{22}$
Chunk Size	2^6	2^8	2^8	2^{10}
#entries	2^6	2^{14}	2^{22}	2^{32}

LAS ≡ VAS

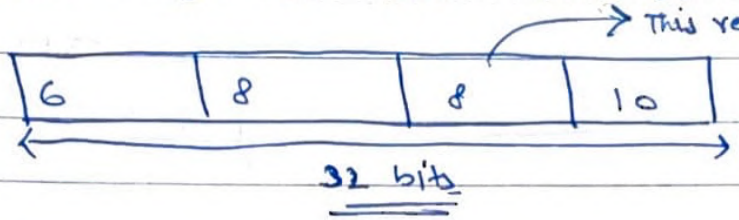
↳ for us as of now know

↳ Minute d/t taught later.

PAGE NO :

DATE : / /

And thus 32 bit VAS can be shown as:



This is called address split in multilevel page table.

	6 bits	8 bits	8 bits	10 bits
# chunks	1 ($2^6/2^6$)	2^6 ($2^{14}/2^8$)	2^{14} ($2^{22}/2^8$)	2^{22}
Size of each chunk (same)	2^6	2^8	2^8	2^{10}
Entries / chunk	2^6	2^{14}	2^{22}	2^{32}

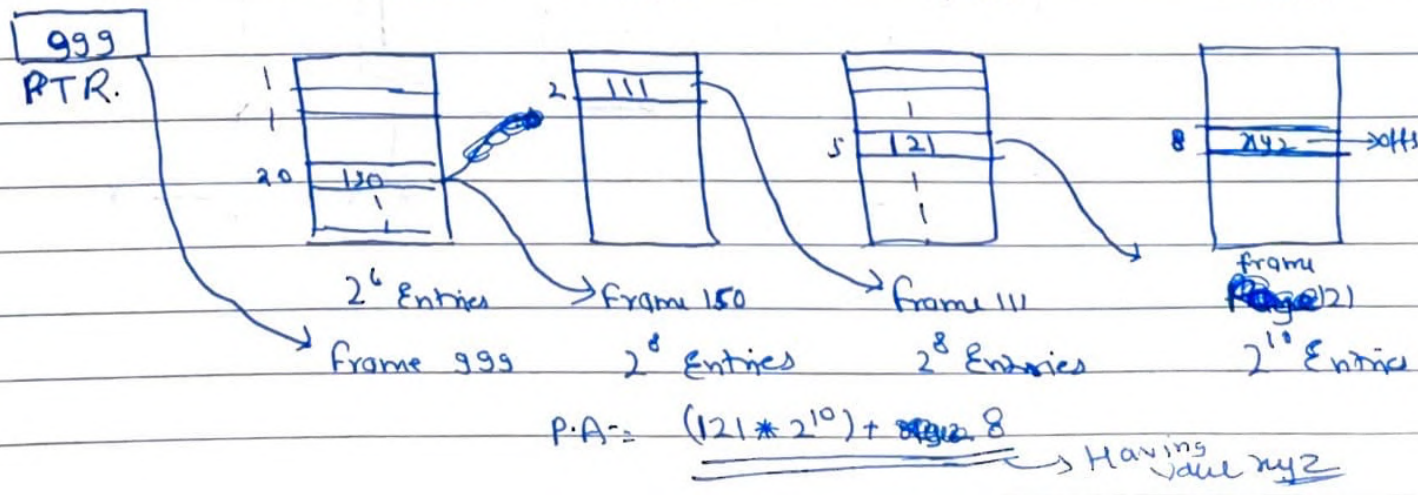
↳ Each entry points to one chunk of next level.

↳ Each entry at this level just contain 1B.

Ex: LA = 6 | 8 | 8 | 10

0 1 0 1 0 0 | 0 0 0 0 0 0 1 0 | 0 0 0 0 0 1 0 1 | 0 0 0 0 0 0 1 0 0 0

$(20)_{10}$ | $(2)_{10}$ | $(5)_{10}$ | $(8)_{10}$



P.A = $(121 * 2^{10}) + 8$ → Having value xyz

PAGE NO. :
DATE : / /

⇒ Just remember this :

Address Split :	6 bits	8 bits	8 bits	10 bits (innermost)
# chunks	1	2^6	2^{14}	2^{22}
Entries/chunk.	2^6	2^8	2^8 ↳ 1 Entry/PTE size	2^{10} ↳ 1 Entry/Byte

→ Each entry in P.T at any level points to chunk at next level. At innermost level, each entry represents 1 B.

Ques : VAS = 32 bits, LA = 0X12345678 and we are using 5 level pg. tables. If innermost pg. table the frame no. is 0X134 then PA = ?, give pg. size = 2^8 B.

⇒ 0X12345678 → offset remains same.

** main

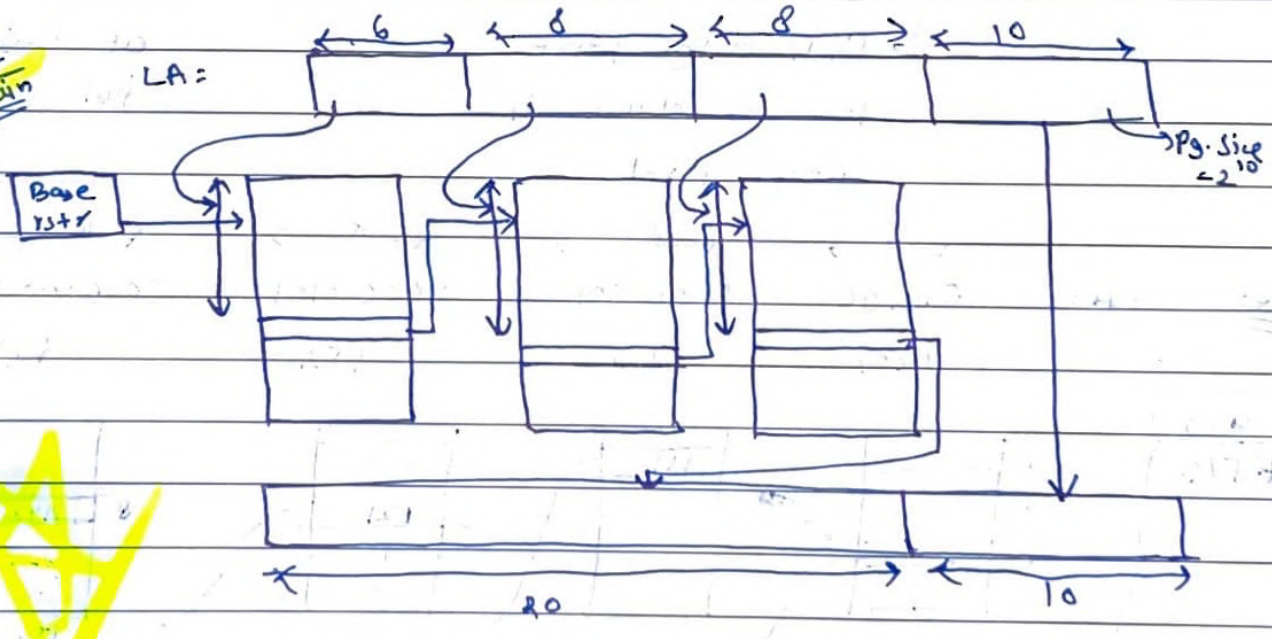


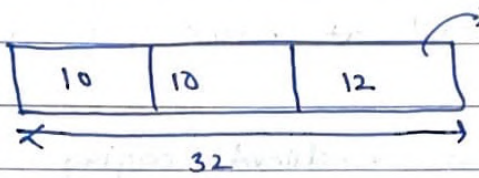
Fig. Address Translation. (Imp.)

PAGE NO.:
DATE: / /

CS Q2

Que! VAS = 32 bit, PAS = 32 bits. Pg Size = 4 KB = 2^{12} B.
Levels = 2. Equal no. of bits should be used for indexing first level and 2nd level of pg. table. PTE = 4 B.

a) Address split:



b). No. of PTE per page?

⇒ 2^{10} Entries.

c) Extra bits available in PTE?

⇒ Bits for frame No. = No. of bits needed to dist. frames
(Given) 20 ($2^{32}/2^{12} \Rightarrow 2^{20}$)

Available = ~~64-20~~ 32-20

⇒ 12 bits

2^{12} bcz $2^{10} * 2^2$ (chunk size)

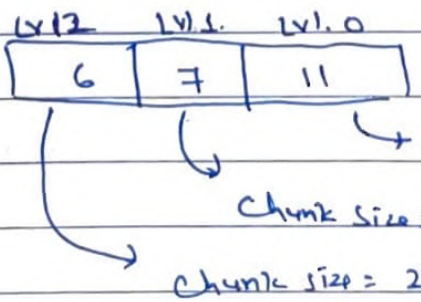
Not due to page size

NOTE ↔ It is not mandatory that Innermost Page is also needed in inner levels, it can be diff at each level. (Coincidentaly same chunk size in above que)

But it can be diff too

coincidence is so common, that not happening coincidence is a coincidence.

Ex. PTE = 2B



PAGE NO. :
DATE : / /

** NOTE: Also diff no. of bits will be req. for frame nos bcz it is not the case that we have hard division of physical memory in frames of page size. That was just for sake of understanding. Diff chunk sizes lead to diff division of a mm. Ex: In prev. ex. at level 1, we need 9 bits at 2nd level 9 bits and at level 0, we need 5 bits given PAS = 16 bits.

This implies that multilevel paging is a choice (Explained Later).

Qu: Given: VAS = 42 bits, PAS = 40 bits, Pg. size = 64KB = 2^{16} B.
PTE = 2^3 B.

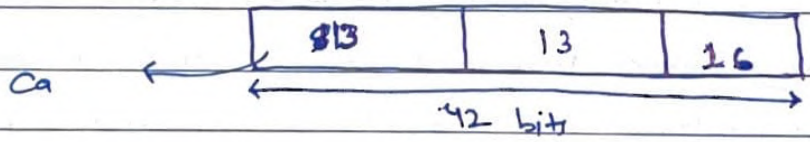
a) Memory needed for pg. table in single level pg. table?

⇒ # PTE = $2^{42} / 2^{16} = 2^{26}$ Entries.

PTS = $2^{26} * 2^3$ B ⇒ 2^{29} B = 512 MB.

b) In multi level paging, Give address split given size of pg chunk at any level is size of physical pg.

⇒



Entries / page = $2^{16} / 2^3 \Rightarrow 2^{13}$ Entries.

PAGE NO.:
DATE: / /

Que: 2 level page tables, VAS = 42 bit, PAS = 40 bits
PTE = 2^3 B, Page Size = 2^{16} B, Chunk Size = 2^{16} B.

Which LA use the same first level (outer level) page table entry as 0x00 123 456 789

⇒ Address Split:

13	13	16

Last ~~three~~ 13 bits must be same for this: (All other two)

same
If should have 0 0000 (0) | 0001 (1) | 0010 (2) | 0011 (3)

0x00123... to 0x00123...
3...9
↳ 28 bits

Ex: ← These must be same →
0x 0000 | 0000 | 0001 | 0010
0 | 0 | 1 | 2

So. 0x 0012.... to 0013....

⇒ For 2nd Level last 26 bit must be same.

0x 0000 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101
0 | 0 | 1 | 2 | 3 | 4 | 5

Start with 0x00012345....
↳ Ans.

PAGE NO. :
DATE : / /

Que: Give Address split if pg size = 256 B, PTE = 2 B &

a) Level 1 page table has 16 Entries
" 2 " " " 128 "
" 3 " " " 128 " .

⇒ | 4 | 7 | 7 | 8 |

b) Level 1 pg. table has 16 B. ⇒ 8 Entries
" 2 " " " 128 B ⇒ 64 Entries
" 3 " " " 128 B ⇒ 64 Entries

⇒ | 3 | 6 | 6 | 8 |

(Do not think it like VAS: 64E × Pg. Size → Question not well framed)
Concept Correct.

Gate CSE 2008.

Que: PAS = 36 bits, VAS = 32 bits, Pg Size = 4KB = 2^{12} B.
PTE = 4B = 2^2 B.

Three level pg. table is used for VA → PA translation, where VA is used as:

| 2 | 9 | 9 | 12 |

The no. of bits req. for addressing the next level page table ?

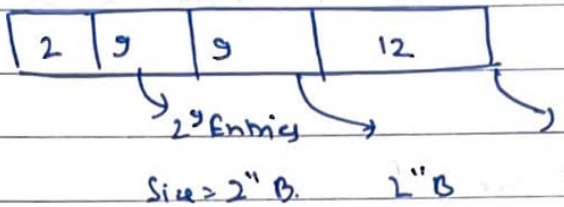
A. 20, 20, 20
B. 24, 24, 24
C. 24, 24, 20
✓ D. 25, 25, 24

PAGE NO. :
DATE : / /

Also matching this ques, at last level final PAS = 36 bits i.e 12 bit offset than 24 bit frame no.

⇒ Basically its asking no. of bits req. for frame size. As at each level frame size will be d/f bcz. chunk size is d/f at each level. (Note ** on prev. pg)

So frame sizes are :



So, bits req. to distinguish frams ⇒ $2^{36}/2^{11} \Rightarrow 2^{25}$,
$2^{36}/2^{11} \Rightarrow 2^{25}$
$2^{36}/2^{12} \Rightarrow 2^{24}$

$2^{25}, 2^{25}, 2^{24}$ (D) (Not it's not B)
Ans.

⇒ Main thing to infer from this que is mm is not hard partitioned by fix frame size. frame size vary acc. to chunk size and so d/f frams req. d/f no. of bits to be distinguished.

Qui: Why we do not put byte no. as entry in pg. table and we only put frame no. as the entry.

⇒ Bcz its convinient for H/w to replace pg. no. with frame no. w/o touching offset. Else it will need complex h/w.

→ Note again chunk size need not be same as page size. But mostly in que. it is given as same.

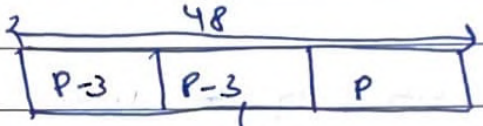
PAGE NO. :
DATE : / /

Similar to Gate CS 6

Ques: VAS = 48 bits. 4 levels of page tbl is used
PAS = 52 bits. PTE = 2^3 B. Chunk size = Pg size
↳ said as "at every level Page table must fit into a page"
If the page size is T, no. of levels can be reduced.
~~If~~ Pg size = ? If we want 2 levels.

⇒ Let page size be 2^P B

Address split:



2^P is size then
$\frac{2^P}{2^3}$ is No. of Entries
So P-3 bits

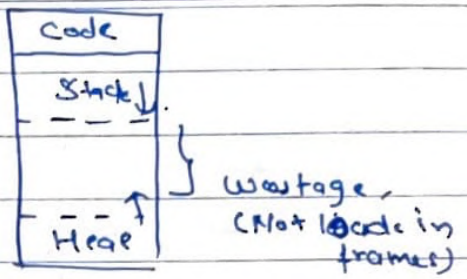
$P + (P-3) + (P-3) = 48$
$3P - 6 = 48$
$3P = 54$, $P = 18$
So page size = 2^{18} B.

All varieties of questions are told

Now coming back to our motivation of multi-level paging, i.e. Savings, where are the savings?
It looks like we have T size instead of d it by adding more page tables.
But,
Ruko zara, Sabar karo.

PAGE NO.:
DATE: / /

As we know our most of the space was wasted here, bcz, these entries were not of any use. They were invalid.

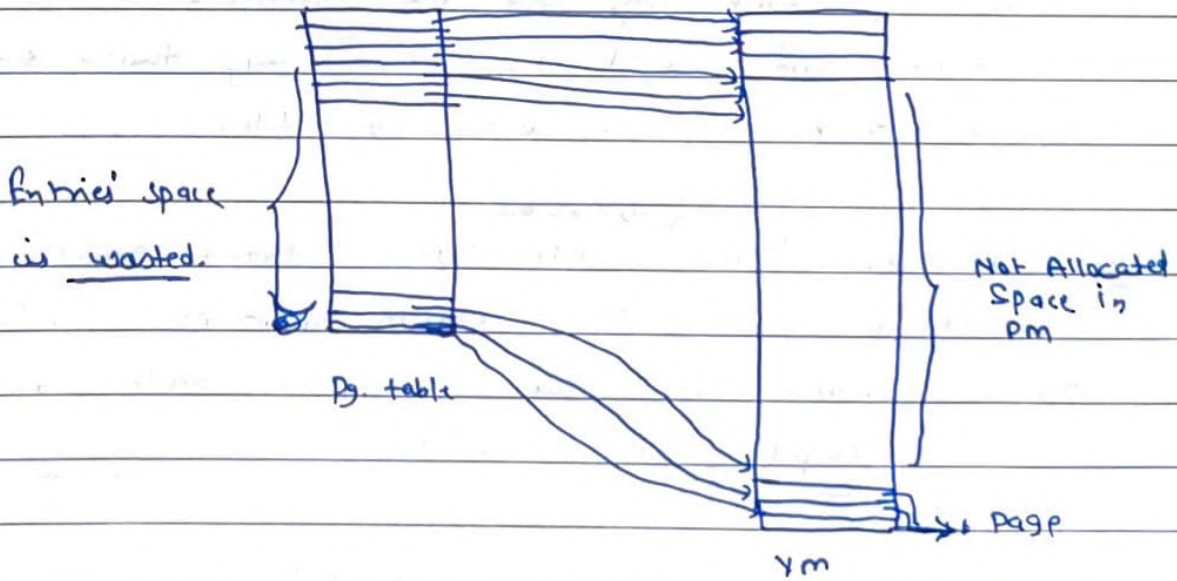


So PTE is invalid.

One thing to not that this space is not even allocated mm, so any request to this space can be given as seg. fault. (Code things). But we have to maintain PTE for it and write Invalid instead of frame no.

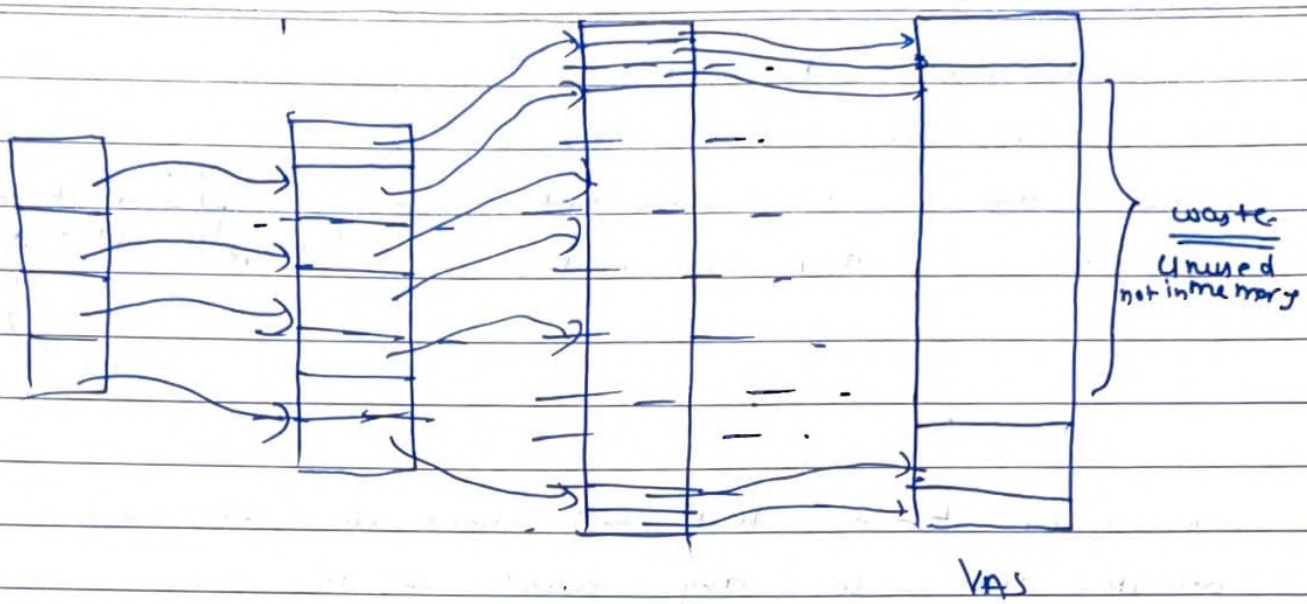
Now, in single level paging we need to store whole page table (including invalid PTE,) in the main memory.

But, if we divide the page table into multiple pages then we have choice to save chunks independently, hence, we can avoid saving invalid entry pages into physical memory.



Single Level paging.

PAGE NO. :
DATE : / /



1st Level 2nd Level 3rd Level

Store only those pages in mm of all page tables which refer to some valid frame no. in 3rd level pg. table.
(Will be clear from questions).

→ If x pages are useful in VAS then only x Entries are useful and as we have now multi level paging these x entries can be present in a single chunk or can be spreaded in 5-6 chunks, then we need to store only those 5-6 chunks in memory (Not whole pg. table).

Que: VAS = 32 bits
Pg. Size = 2^{12} B, PTE = 4B, Actual ~~pages~~ memory useful is 2^{16} B, starting from 0. ~~Size of~~

a) Calculate size of all page tables in mm.

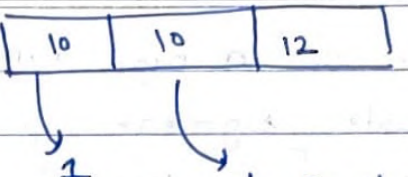
⇒ Useful pages = 2^{16} B / 2^{2} ⇒ 2^{4} pages (Continuously from zero.)

So 2^4 Entries are useful only.

Note that in most of the new OS, multi-level paging is used.

PAGE NO.:
DATE: / /

Address Split:



useful chunks of Pg table → 1

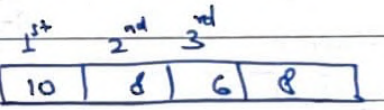
1 or 1 pg. can have 2^{10} Entries but we need 2^4 only (Diff if useful pages were scattered)

So, $(2^{10} * 4)B + (2^{10} * 4)B$

$\Rightarrow$ 8 KB useful.

(It was 4 MB when using single level Page table).

Que: PTE = 2B, Address split is:



a) Pg table size for a process that has a 256 K of memory starting at address 0.

$\Rightarrow$ Page Size = 2^8 B.

useful pages = $256 * 2^{10} / 2^8 \Rightarrow 2^{10}$ pages

Entries useful in 3rd level = 2^{10} Entries

Chunks req. for 2^{10} Entries = $2^{10}/2^6 \Rightarrow 2^4$ Chunks.

Entries useful in 2nd level = 2^4 Entries

1 chunk is useful for 2^4 Entries in 2nd level

1 Entry is req. in 1st level, still 1 chunk will be needed.

total = $(1 * 2^{10} * 2) + (1 * 2^8 * 2) + (2^4 * 2^6 * 2)$

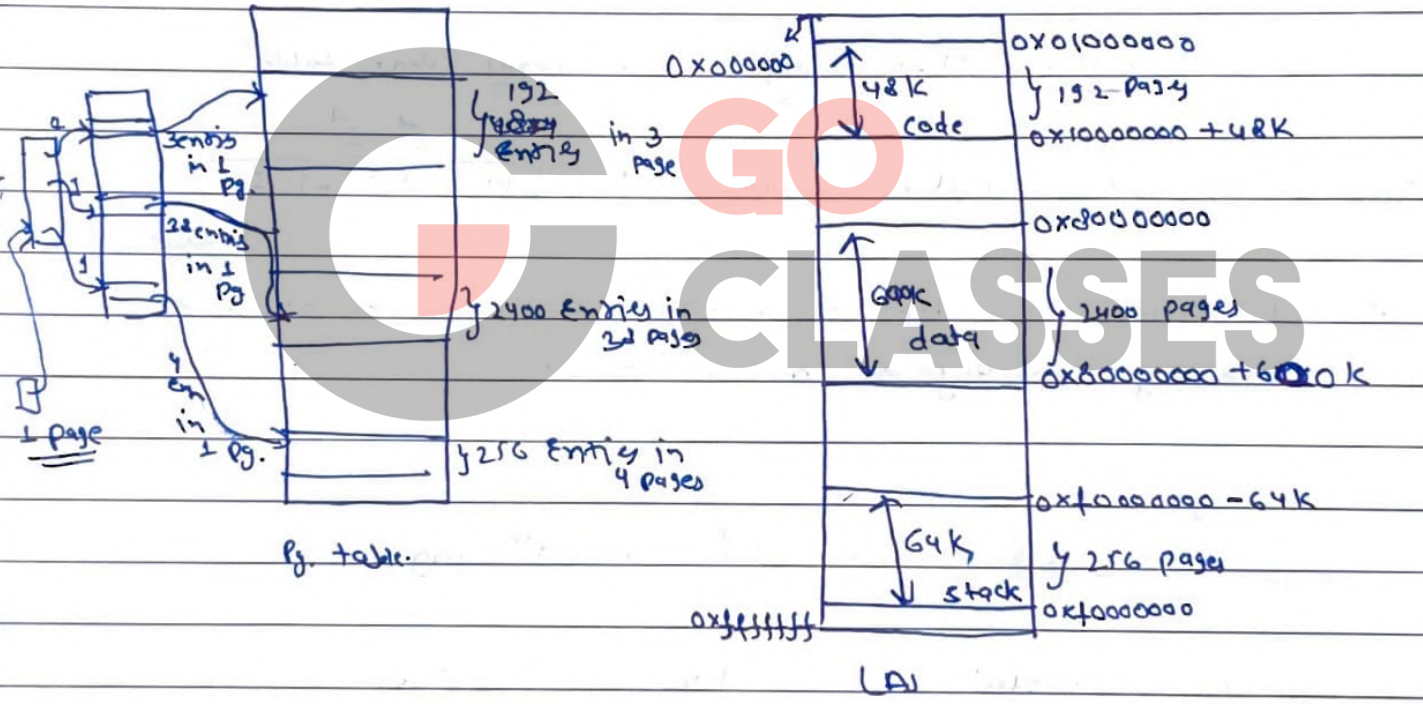
$\Rightarrow 2^{11} + 2^9 + 2^{11}$

$\Rightarrow 2^9 (1 + 8) \Rightarrow 9 * 2^9 = 4608$ B.

PAGE NO.:
DATE: / /

b) What is the size of a page table for a process that has a code segment of 48K starting at address 0x1000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf0000000 & growing upward.

⇒ Here useful pages are not contiguous



	Code	Data	Stack.
# useful pages = # entries in 3rd level	$\frac{48*2^{10}}{2^{12}} \Rightarrow 48*4$ pages	$\frac{600*2^{10}}{2^{12}} \Rightarrow 600*4$ pages	$\frac{64*2^{10}}{2^{12}} \Rightarrow 64*4$ pages
# page in 3rd level ⇒ 2^6 entries/page	$\frac{48*4}{2^6} = 3$	$\frac{600*4}{2^6} \Rightarrow 38$	$\frac{64*4}{2^6} \Rightarrow 4$
# pages in 2nd level ⇒ 2^{10} entries/page	1	1	1
	1 for all		

PAGE NO.:
DATE: / /

Total useful page's Entry:

$[(3+38+4)*2^6] + [3*2^8] + [3*2^{10}]$

⇒ $(45*64) + (3*256) + (38*1024)$ Entries

⇒ $2880 + 768 + 38912$ Entries

⇒ 4672 Entries

⇒ 9344 Bytes

In this qu. we assumed all three segments are for enough such that their page entries do not come under same chunk at 2nd level i.e. b/w the entries in 2nd level, there is atleast 1 Chunk

It can be checked by checking address i.e. if there are 2^{10} entries b/w the entries in 2nd level. (Very Complex).

↳ But need to do.

Qui: GATE CSE 2003 | question: 79.

↳ Must
Must visit on Gateoverflow for clear understanding of this topic.

Not writing Here

PAGE NO. :
DATE : / /

Summary:

→ We have seen 4 ways to get PA given LA:
• Base • Base & Bound • Segementating • Paging.

Each had it's own advantages & own disadvantages

→ Advantages of Paging:

a) No External fragmentation
b) Fast to allocate (no need to scan) & No compaction.

→ Disadvantage of Paging:

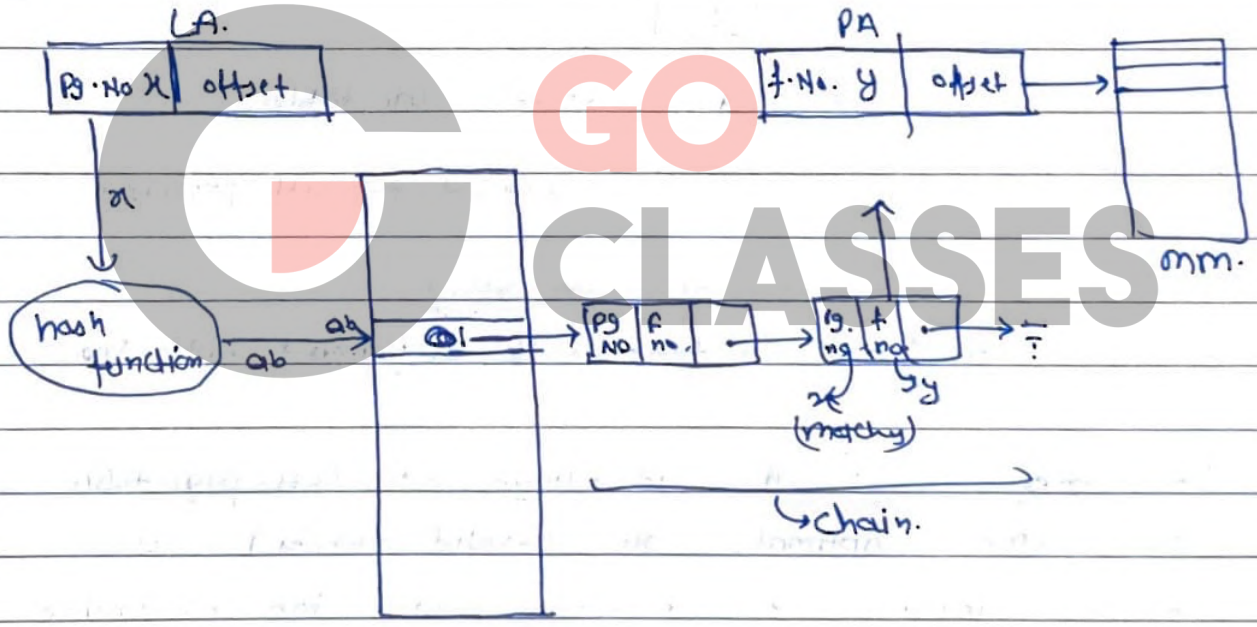
a) Additional memory req. for pg. table.
b) Internal Frag. (i.e. space with a page is wasted, can be avoided by, decreasing page size, but it increases page table size, it's a trade off).
c) Required space for page table may be substantial (Avoided by using multilevel paging).

★ Hashed Page Table: Alternate to multilevel paging, used for reducing size of page table. Idea here is to reduce page table size w/o using multilevel paging.

PAGE NO.:
DATE: / /

We cannot directly store only valid pages entries bcz, we have to also know the page no. This can be done by having 2 cols page no & frame no. But, it reduces efficiency as we can not directly go to frame no., we have to do binary search. So huge time complexity (O(1) → O(log n))

So, the final solution can be having which takes O(1) to search. Hashing with chaining can be used.



Hashed Pg. Table.

Fig. Address Translation in Hashed Pg. Tbl.

NOTE: Most of the systems use Multi Level Paging.

Global Pg Table → Inverted Pg. Table

PAGE NO. :
DATE : / /

★ **Inverted Page Table:** It is another option to do paging. Here we use global page table unlike before where we had 1 page table / process.

Here Index is frame no and each entry included process no. as well as page no. that frame is holding.

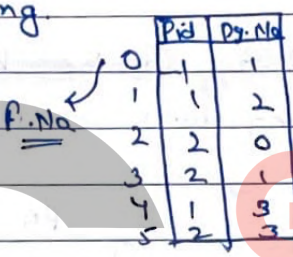


Fig: Inverted Page table.
↳ Global for all processes

Size of Invtd. Pg. table = No. of frames * Entry Size.

→ It comes with its advantage i.e. Less page table size (most optimal, No Invalid entries) and its disadvantage i.e. searching is too expensive.
Need to traverse whole page table to know respective frame number.

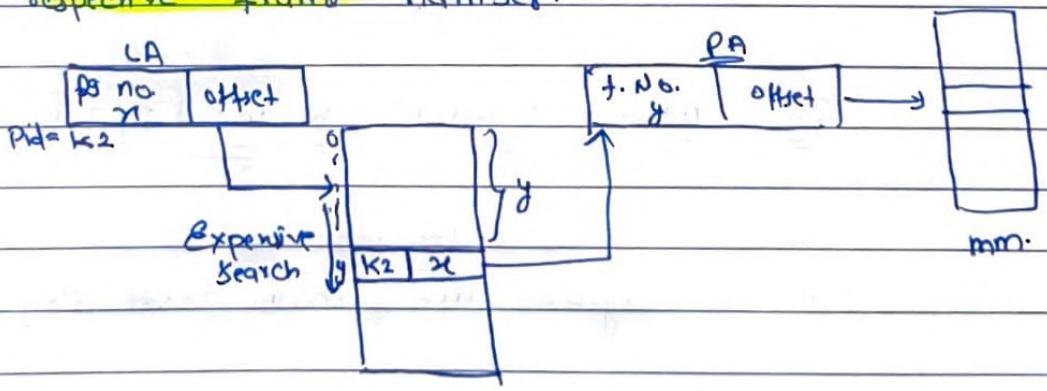


Fig: Address Translation

PAGE NO.:
DATE: / /

Que: Size of inv td. pg. table grows with size of VAS?

⇒ **false.** , it grows only if we decrease frame size or T mm.

Que: T/F:

a) Base-Bound is faster than single level Paging

→ True.

b) Invtd. page table is fast then single level.

→ False.

c) Single level is faster than multi level.

→ True

d) Multi level is faster than inverted.

→ False Gen. but may or may not.

NOTE ↔ We can use hashing for Inverted page table also.

Memory ↓ Time ↑ in paging as any memory mgmt.

★ Segmentation with Paging: Similar to multilevel paging. We divide process in segments & then segments in fixed size pages.

V.A.

Seg. #	Page #	offset

PAGE NO. :
DATE : / /

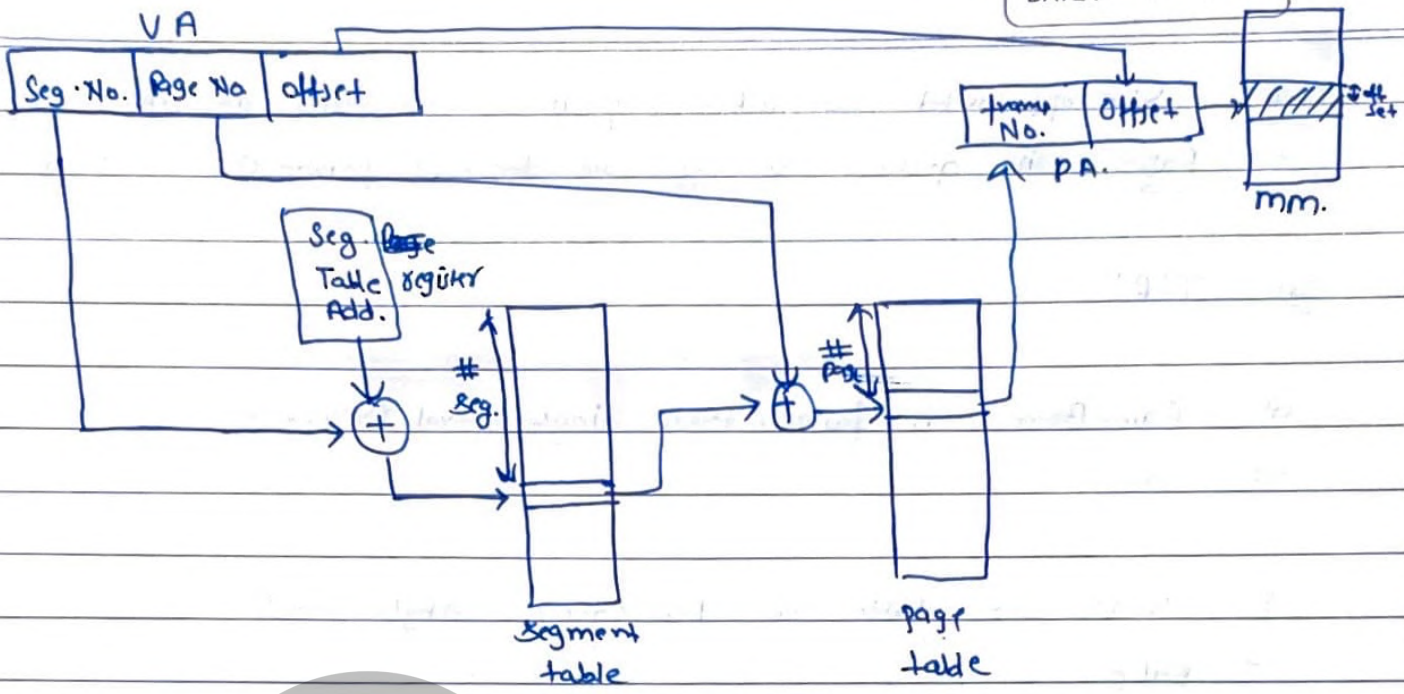


Fig.: Address Translation for paging x seg.

Qesn: Assume that a task is divided into four equal sized segments & that the system builds an 8 entry page table for each segment. Page size is 2 KB. System uses comb of pging & seg.

a) Max. size of each seg.?

⇒ 1 x 8 x 2 KB ⇒ 16 KB

b) max. logical address space for task?

⇒ 4 x 16 KB ⇒ 64 KB

X c) PA = 00021ABC , LA = ?

Incomplete Given Ignore

offset = 11 bits (2^{11} B is page size)

PAGE NO. :
DATE : / /

Gate IT 06
Que:

Process	Total Size in KB	#segments
P_1	195	4
P_2	954	5
P_3	45	3
P_4	364	8

Pg. Size = 1KB = 2^{10} B.
PTE = 4B = 2^2 B (Pg. Tbl Entry)
STE = 8B = 2^3 B (Seg. Tbl entry)
Max. Seg. Size = 256KB = 2^{18} B

P: Storage overhead for 2 level paging
S: " " " Segmentation
T: " " " " with paging.

Tell the relation b/w P, S, T ?

⇒ ~~Page table for all pages~~

for 2 level paging:

	P_1	P_2	P_3	P_4
#pages for	195	954	45	364
#PTs for	195×4	954×4	180	364×4

Now we assume chunk size for inner page table as page size then # pages required are

P_1	P_2	P_3	P_4
1+1	1+1	1+1	2+1

⇒ 9 KB

So P = 9 KB

PAGE NO. :
DATE : / /

By Segmentation

	P_1	P_2	P_3	P_4
# seg.	4	5	3	8
# Seg. table size =	4*8	5*8	3*8	8*8

So, S = 32 + 40 + 24 + 64
= 160 B.

By Paging with Segementation.
(~~Assuming divide each process in equal segments~~)

Max. Seg. size is 256 KB, and page size is 1KB
so need 256 PTE for each segment = 1024 B (256*4)

	P_1	P_2	P_3	P_4
Seg. table size	32	40	24	64
Page table	4*1KB	5*1	3*1	8*1

Total size, T = 4KB + 5KB + 3KB + 8KB + 160B
= 20640B

So $S < P < T$ (B)
AS

PAGE NO.:
DATE: / /

★ Common flags of Page Table Entry:

- **Valid**: Indicating whether the particular translation is valid.
- **Protection**: Indicating page permissions
- **Present**: Indicating page is in mm or on disk (swapped out)
- **Dirty**: Indicating if page has been modified or not.
- **Reference**: Indicating that a page has been accessed.

Virtual Memory:

Till now, we assumed all pages of a process are in main memory before they execute.

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

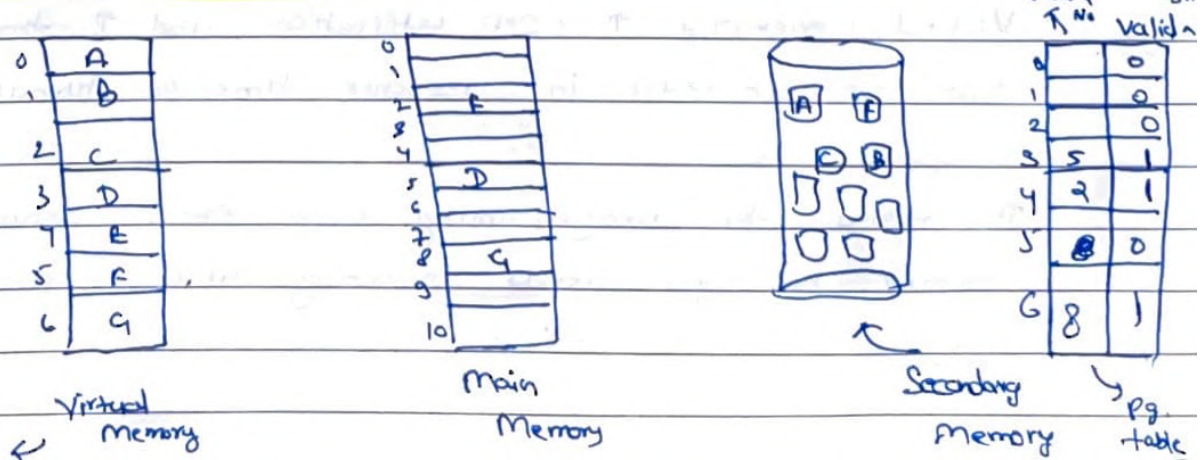
One major advantage of this scheme is that program can be larger than physical memory.

Virtual memory abstracts main memory into an extremely large continuous storage.

★ **Demand Paging**: It says that even if we can load only one page in the main memory then also job can be executed.

Since, we are not putting all pages in mm, hence some of the pages will also reside in Disk.

Ex:



PAGE NO. :
DATE : / /

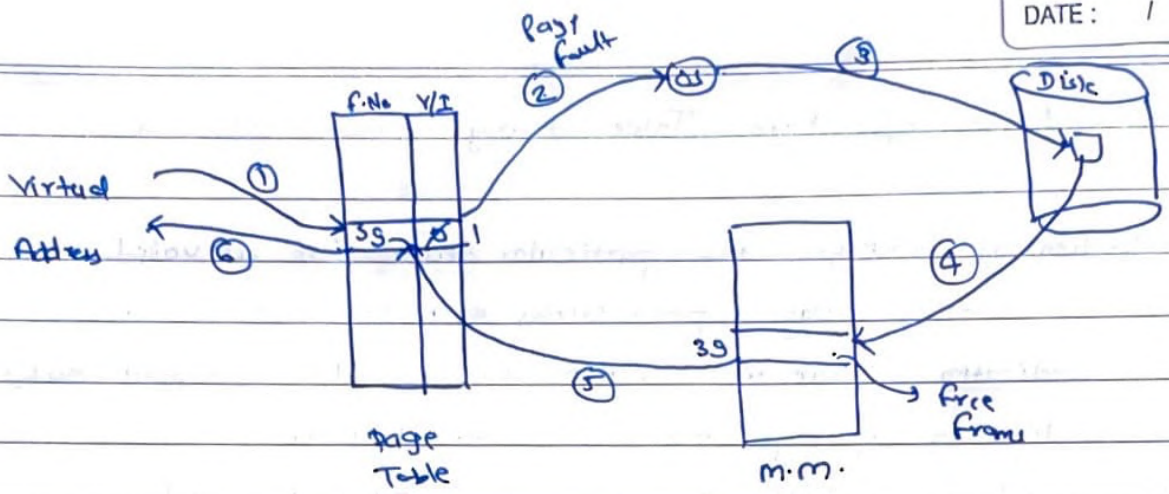


Fig: Seq. of execution when page is not in mm.

★ Logical Memory: We say to compiler that given memory is contiguous to you. We don't talk about size.

★ Virtual Memory: It say that logical memory can be bigger than physical memory.

↳ Implemented by → Demand paging
→ Demand segmentation

Virtual memory helps us to achieve higher degree of multiprogramming (many process) and obviously allows execution of process that may not be completely in mm.

Virtual memory ↑ CPU utilization and ↑ throughput but no increase in response time or turnaround time.

It makes the programming much easier because programmer no longer needs to worry about amount of physical memory available.

PAGE NO. :
DATE : / /

→ If page is not present in memory then CPU generates an interrupt called page fault interrupt.

→ In pure demand paging we start with zero pages in MM. all pages in disk initially.

Gate CSE 95

Qu: Add. seq. gen. by tracing a particular program exec. in pure demand paging with 100 records/page with 1 free MM frame is as follows:

0100, 0200, 0430, 0499, 0510, 0530, 0560, 0120, 0220, 0240, 0320

page faults?

⇒ Page No. will be record %100 bcz 100 record/page. Page faults are underlined. i.e. 7

When we bring 0100, then 1st page in MM is record 100-199 in MM.

Gate IT 07

Qu: 100 byte/page. Add. seq. is : (with pure demand paging)

0100, 0200, 0430, 0499, 0510, 0530, 0560, 0120, 0220, 0240, 0260, 0320, 0410.

Suppose that memory can store only 1 page & if address 'x' causes pg. fault. then bytes from x to x+99 are located in MM. # page faults?

⇒ DIY (Ans: 7) (Diff from above qu)

PAGE NO. :
DATE : / /

★ Performance of demand paging:

Effective access time EMAT for demand paged memory can be computed as:

$$EMAT = (1-p)(m) + (p)(pf)$$

where p is prob. of page fault,
m is memory access time,
pf is page fault service time.
↳ Includes everything from SM to MM & then memory access.

Ex: $m = 200\ ns = 200 * 10^{-9}$ sec.
$pf = 8\ ms = 8 * 10^{-3}$ sec.

$$EMAT = 200 + p(7,999,800)\ ns$$

If $p = \frac{1}{1000}$, i.e. if 1 in 1000 page causes pg. fault
EMAT = 8.2 μs.

This is slowdown of 40 even when prob. is so less. Our goal for next couple of pages is to reduce this prob. of page fault.

After page fault, we load page in MM, but if MM doesn't have space, we may need to replace one of the page.

PAGE NO. :
DATE : / /

So, to choose the page to be replaced, we need an algorithm and the performance of algorithm will be best if result in minimum page faults. Such algorithms are called Page Replacement Algorithms, which we will see now.

Note that we need to write the victim page (page we chose to replace) to write back in Disk before we replace it if and only if it's been modified i.e. dirty flag is True.

So, If we incurr a page fault:

→ If there is an unused frame, get it.

→ If no unused frame available, then:

- find a used page frame (using algorithm)
- If modified write it to disk
- Invalidate its current PTE and TLB entry
- get page in this frame now
- update PTE of this page

} Page Replacement

A) Optimal Algorithm: Obviously, not possible to implement, only for comparison purpose, Replace that page which we'll not use in futures.

bc2. don't know what we need in future

Ex: 4 page frames.

Ref. String: 1 2 3 4 1 2 5 1 2 3 4 5

1	5
2	
3	
4	4

6 page faults
↳ minimum Possible.

PAGE NO. :
DATE : / /

B) First in first Out (FIFO): Throw out the oldest page loaded. (Not oldest used)

Ex: 4 page frames.

Ref. string: 1 2 3 4 1 2 5 1 2 3 4 5

4	3
3	2
2	1 5
1	3 4

1	2	3	4	3	1	2	3	4	5

Queue
~~Stack~~ for Rough use

10 page faults.

Qn: What if we had 3 page frames or any more lesser than # page faults?

⇒ might increase. ~~Or~~ may decrease also. (Counterintuitive)

It will increase mostly but may decrease when lesser frames are no more subset of more frames & ref. strings now contain pages that of less frames.

This is known as Belady's Anomaly.

↳ Page fault ↓ with ↓ frames.

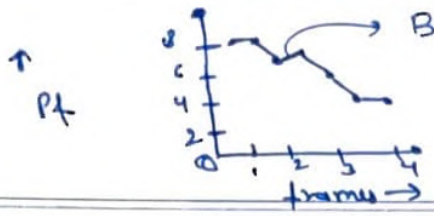
Ex: 3 page frames

Ref string: 1 2 3 4 1 2 5 1 2 3 4 5

3	2 4
2	1 3
1	5

1	2	3	4	1	2	5	1	2	3

9 page faults.



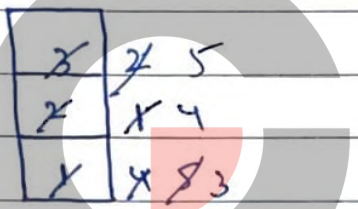
PAGE NO.:
DATE: / /

Belady's anamoly occurs when fewer frames are no more subset of more frames and useful page gets removed

c). **Least Recently Used (LRU)**: Replace the page used least recently. (Not loaded like FIFO)

Ex: 4 page frames.

Ref. String: 1 2 3 4 1 3 5 1 2 3 4 5



(stacks with update each use →)

1 2 3 4 1 3 5 1 2 3 4 5

8 page faults.

→ Linked list can be used to implement this.

→ There are many more variations of these algorithms like MRU, LIFO, etc. which can be practiced directly by questions.

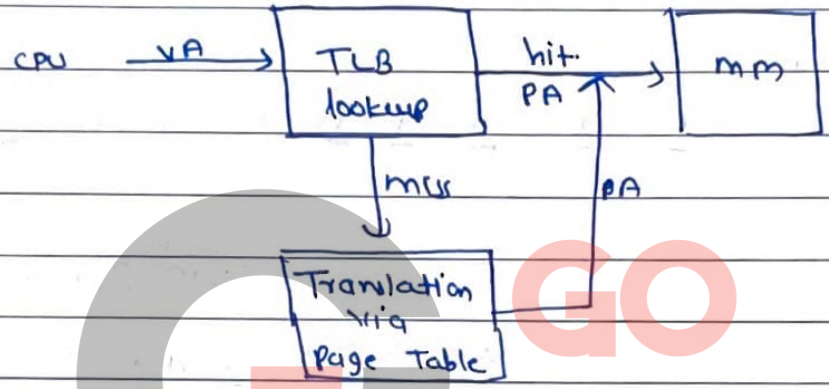
D). **Least Frequently Used (LFU)**: It keeps counter of no. of ref. made for each page and replaces page with smallest count.

→ Practice GATE PYQs from this topic ←

PAGE NO. :
DATE : / /

★ **Translation Lookaside Buffer (TLB)**: TLB is a cache memory (way faster than mm) which contains some of the page table entries (Not whole page table or page)

It says. "Address Translation ke lie ja rahe ho? Home bhi dekhle phle. (Sideeye se)".



⇒ Why TLB works? Due to locality.

⇒ **Locality**: Programs tend to use data & instruction with addresses near or equal to those they have used recently.

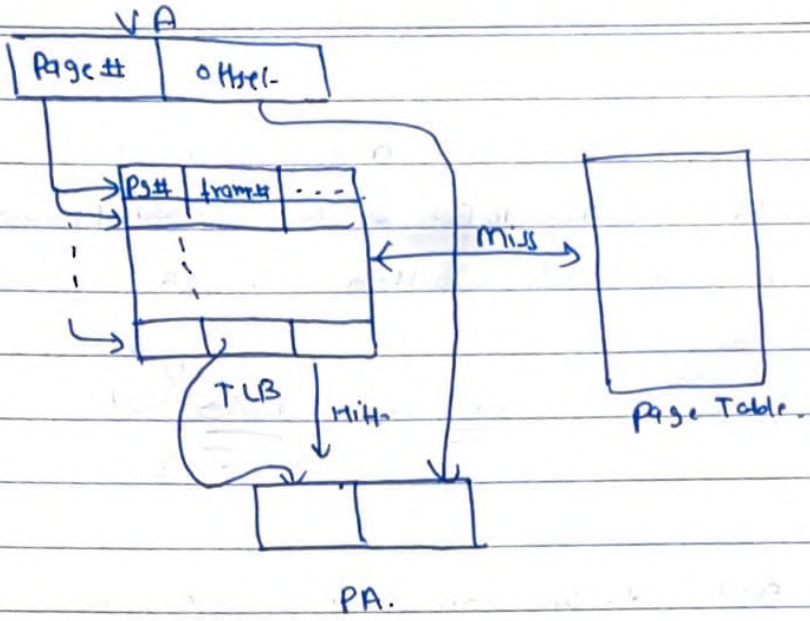
It is of two types:

a) **Temporal Locality**: Recently accessed ~~Pages~~ items are likely to be ref. again in near future.
So same PTE will be used again.

b) **Spatial locality**: Items with nearby addresses tend to be ref. again. They are in same page so same PTE will be used again.

→ All caches take advantage of this.

PAGE NO. :
DATE : / /



TLB gets PTE after it misses it, due to its belief in locality.

Qu:

```
Sum=0;
for(i=0; i<n; i++)
    Sum+= a[i];
```

Which of these memory location has temporal and spatial locality in above code?

a) i — Temporal
b) sum — Temporal
c) array elements a[0], a[1].. — Spatial.
↳ Nearby Addresses.

Qu: If array a[] has 1024 items, each item is 4B, Page size is 4KB, code is:

```
int sum=0;
for(i=0; i<1024; i++)
    sum+=a[i];
```

Then no. of TLB miss?

⇒ Each page can have 1024 items of array ⇒ $\frac{4\times1024\ B}{4B}$

TLB miss # = 0 (If all entries in 1 page of PTE in TLB), 1 (If all entries in 1 Page of PTE not in TLB), 2 (If all entries spanned in 2 Pages)

TLB is just like any other cache. (fully / set / direct associative)
↳ Typically not more than 128 or 256 entries even on Gaming PC
↳ faster than cache.

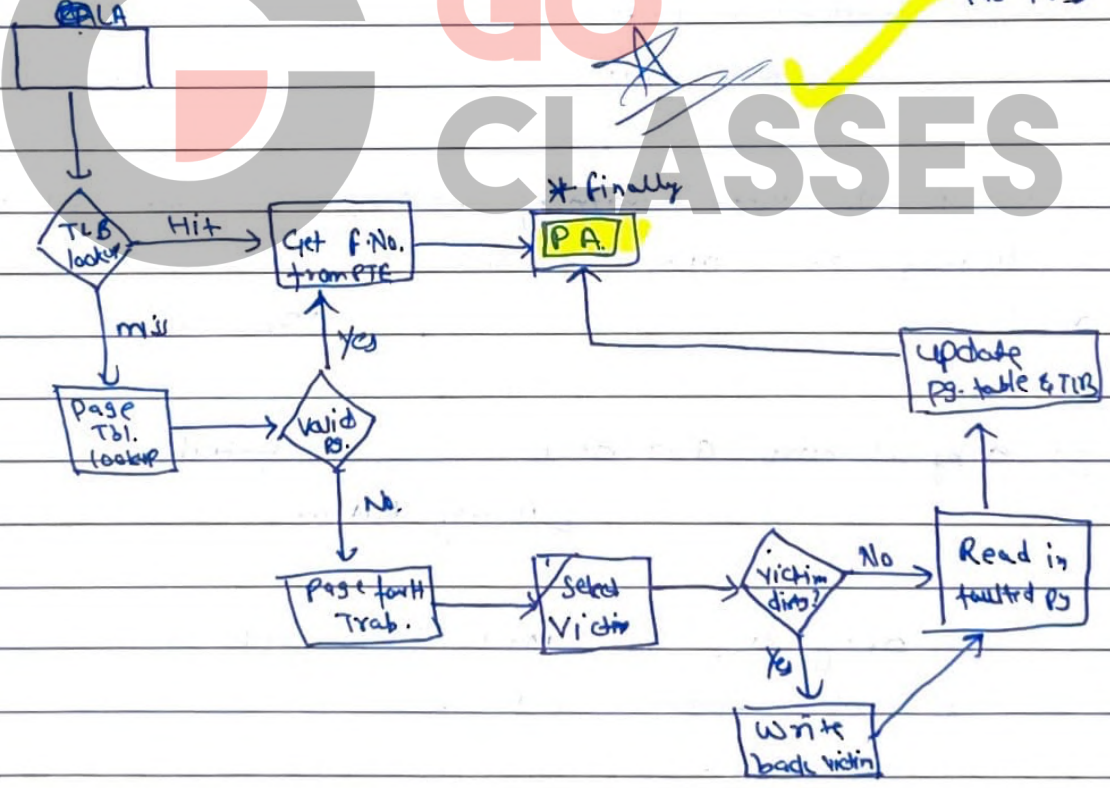
PAGE NO.:
DATE:

Que: Consider a m/c with 64 bit VAS and pg size = 8KB = 2^{13} B
TLB has 512 entries, mm = 4GB = 2^{32} B. What fraction of PM can be accessed w/o suffering a TLB miss?

⇒ $512 * 2^{13}$ B ⇒ 2^{22} B ⇒ 4MB $\quad \frac{1}{1024} * 100\%$
↳ Ans.

" Knuth says that 90% of program's time get spend in 10% of program. "

Imp:
(Note that if TLB hit, then No PF.)



Gate 2014

Que: Consider a paging h/w with TLB. Assume no page fault. It takes 10 ns to search TLB and 80ns to access PM. If TLB hit prob. is 0.6, EMAT is:

PAGE NO. :
DATE : / /

⇒ Approach 1:

$P(TLB\,Time + MAT) + (1-P)(TLB\,Time + MAT + MAT)$

(Pg. table → Memory → data)

where P is hit ratio

Approach 2:

$MAT + TLB\,Time + (1-p)(MAT)$

(for data; we always look here; If not in TLB, then see pg. table)

Approach 3:

VA → PA + Data Access → MAT

$P(TLB) + (1-p)(TLB + MA) +$

Use any, answer is:

$80 + 10 + 0.4 * 80$

⇒ 122 ms

Similar qu. can be made for Page faults.

Gate 92

Que: If an instruction takes 'i' ms and a page fault takes additional 'j' m&conds, the effective instruction time if on an avg. a page fault occurs every k instruction is:

⇒ $i + \frac{1}{k}(j)$

NOTE: Servicing a Pg. fault includes everything (Hard Disk → MM + Data Access)

PAGE NO.:
DATE: / /

GateCSE

Qu: Suppose time to service a page fault is on the avg. 10 ms while a mat take 1 μs. Then a 99.99% hit ratio results in avg. MAT?

⇒ .0001 (10 × 10^-3 s) + .9999 (10^-6 s)

⇒ 1 μs + .9999 μs

⇒ 1.9999 μs

→ Ans.

★ Dynamic Allocation Strategies: Also called Heap Allocation Strategy because with malloc & free requests, holes starts to be created in Heap memory (fragmentation). So we can't satisfy request even after having free space.

There is not such issue in stack bcz it will grow or shrink in a fixed manner.

If there was no option of free in Heap then this issue would not be present in Heap also.

There are three algorithms by which we allocate these requests in heap:

a) Best fit:
- Search whole free space list
- Choose smallest hole satisfying request
- Can stop search if exact size hole found.

b) First fit:
- Choose first hole/block that can satisfy request

c) Worst fit:
- Search whole list to choose largest block.

→ These strategies can also be used in Segmentation or any variable sized partitioning.

PAGE NO.:
DATE: / /

→ Next fit is also strategy similar to first fit, but instead of starting from start, they start from last pointer/hole.

→ first fit & Best fit are found to operate better than worst fit in terms of speed & storage utilization.

Gate 1)
Qu: Consider 6 memory partition 200, 400, 600, 500, 300 & 250 KB These partitions need to be alloted to four processes of size 357, 210, 468, & 491 KB in that order. If the best fit is used which partitions are not alloted to any proces?

⇒

Proces	Partition
357 →	400 KB
210 →	250 KB
468 →	500 KB
491 →	600 KB

So, 200 & 300 KB partitions are not used.

Gate CS 2020 | 53 Qu.
Qu: TLB pg fault qu.

Consider a paging system that uses 1 level page table residing in mm & TLB for addres trans. MA = 100 ns. TLB time = 20 ns, Page Transfer to/from disk = 5000 ns TLB hit ratio = 95%, Page fault rate = 10%, 20% pages are dirty TLB update time is negligible EMAT ?

⇒ Page fault service time = 5000 + (.2)(5000) + 100 → Data acces
(read always, write when dirty)
= 6000 ns + 100 ⇒ 6100 ns

Data Access Time = 100 ns (.9) + (.1)(6100)
↳ No pg. fault

Catch here is if in TLB then no pg. fault

PAGE NO. :
DATE : / /

EMAT = TLB Hit Ratio (Time Hit) + TLB miss Ratio (Time Miss)

Time Hit = TLB Time + MAT

= 20 + 100 = 120 ns

Here, he assumed if TLB hit, then no page fault., bcz if page fault, then TLB would not have an Entry)

Time Miss = TLB Time + pf (Time) + nopf (Time with Service)

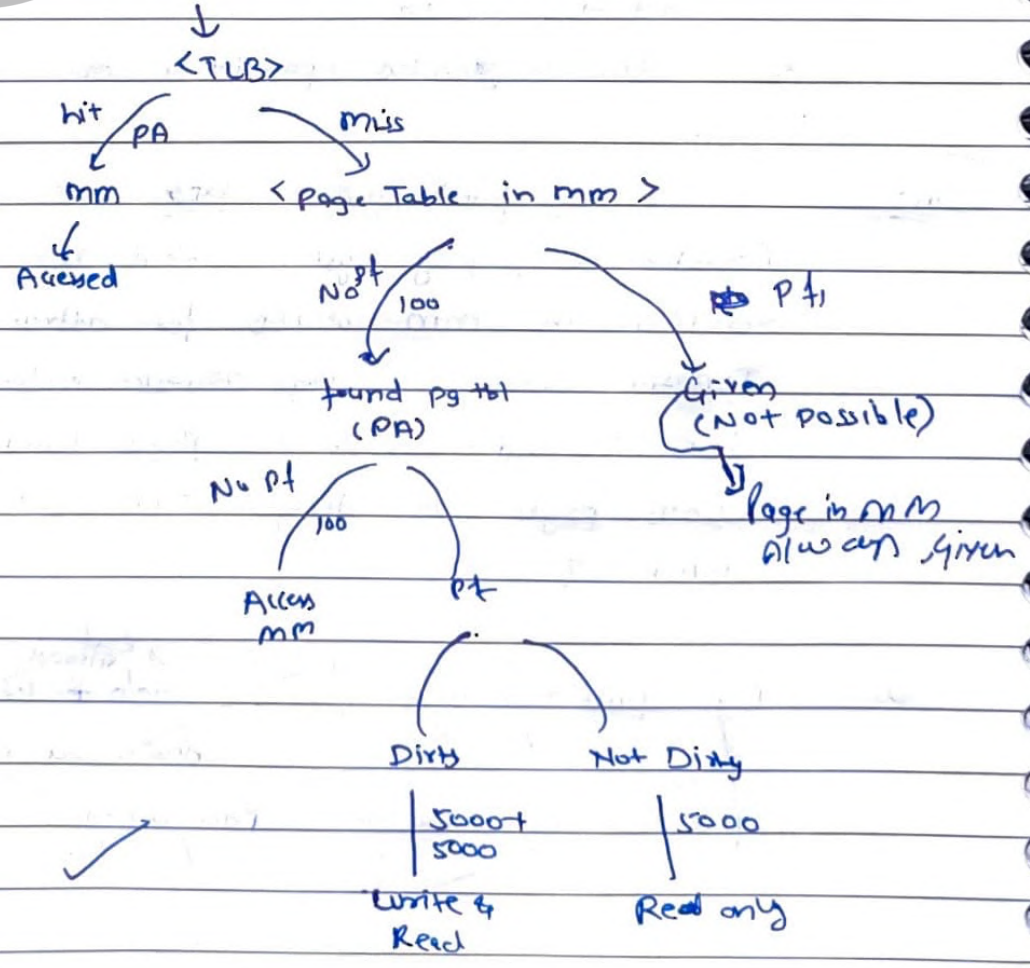
= 20 + [0.90 (100 + 100) + 0.10 (100 + 100 + 0.80 (5000) +
(No Pf; Pg. table; Data; Pf; Pg tbl; Data; Pg. not Dirty; Read)

+ 0.20 (5000 + 5000))
(Dirty Page; Read; write.)

= 820 ns

EMAT = 0.95 (120) + 0.05 (820) = 155 ns
(114; 41)

Flow:



PAGE NO. :
DATE : / /

Deadlock & File System written in Another NB.

→ Thrashing in an OS is when a computer spends too much time swapping data b/w its physical memory & virtual memory.
Causes of thrashing are:
- Insufficient physical memory for the workload.
- Overallocation of memory to multiple processes.
- Inefficient memory allocation strategies.

Precise
→ Thrashing occurs when size of all locality (set of pages used together) across all the running program exceeds the memory size.
- ↑ degree of multiprogramming will ↑ thrashing
- Priority page replacement can limit thrashing.

→ In dynamic loading in OS, libraries are loaded only when the functions in them are actually called during runtime. So it is useful in system with less main memory.

→ • Best fit reduces internal frag, but can't avoid it completely.
- Best & First fit strategies suffer from external fragmentation.
- Neither best fit nor first fit is clearly better than each other in terms of storage utilization, but first fit is usually faster.

Made By - Karan Agrawal (GATE CS 2024 AIR 102)

PAGE NO. :
DATE : 10/10/23

Lecture - 34, 35

★ **Deadlock**: It is a problem in which, A set of blocked processes each holding a resource and waiting to aquire a resource hold by another process in that set.

Ex: A system has 2 disk drives, P1 & P2 each hold one disk drive and each needs another one.

Fundamental cause of deadlock is "Lack of Resources" which can't be solved.

We have already seen deadlock situations in semaphores

Ex:

P1:	P2:
wait(A);	wait(B);
wait(B);	wait(A);

→ One sarcastic example where you are in deadlock is, "You can't get job w/o experience ; you can't get experience w/o job."

★ **Conditions for Deadlock**: 2 Rtd to Rsrc (a,c) and 2 to process (b,d)

a) **Mutual Exclusion**: It says resources are not sharable.

b) **Hold & wait**: It says process holds resource & waits for resource.

c) **No preemption**: It says resources (→ Not process) can't be preempted.

d) **Circular wait**: Processes waiting for resources in circle. P_0 waiting for P_1, P_1 for P_2 ... P_n for P_0.

PAGE NO. :
DATE : / /

→ These four conditions are like four legs of table. If any leg is broke, deadlock is not possible

* These conditions are necessary conditions but not sufficient.

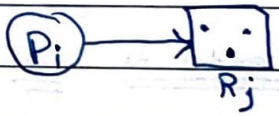
i.e. Deadlock → All 4 conditions satisfied.
All 4 conditions satisfied → May or may not be deadlock
Any 1 condition not satisfied → Deadlock Impossible.

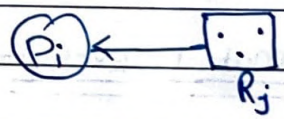
→ Deadlock can not occur among processes that need atmost one resources each, because "Hold & wait" is not possible in those processes.

* Resource Allocation Graph:

Process → O

Resource with 4 instances → 

Pi request instance of Rj → 

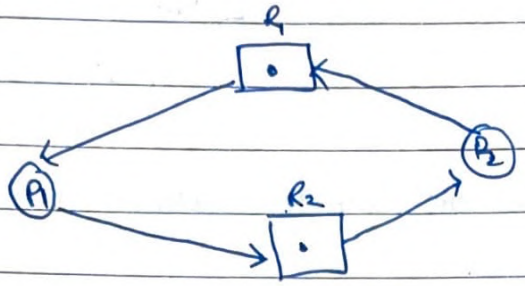
Pi holding instance of Rj → 

We have a formal method to detect deadlocks, but till then use Intution

PAGE NO. :
DATE : / /

Que : Does the following system shown by RAG have deadlock ?

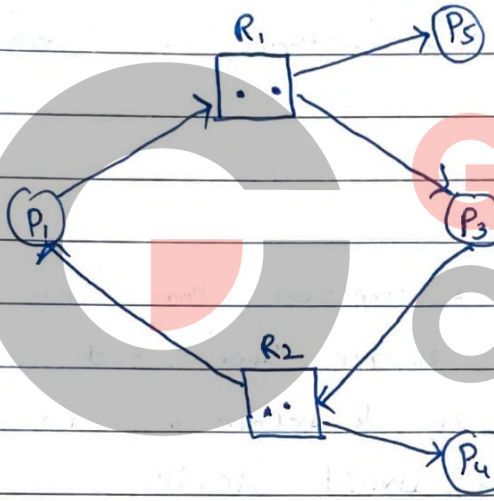
a)



Cycle & Deadlock

→ They are clearly in deadlock.

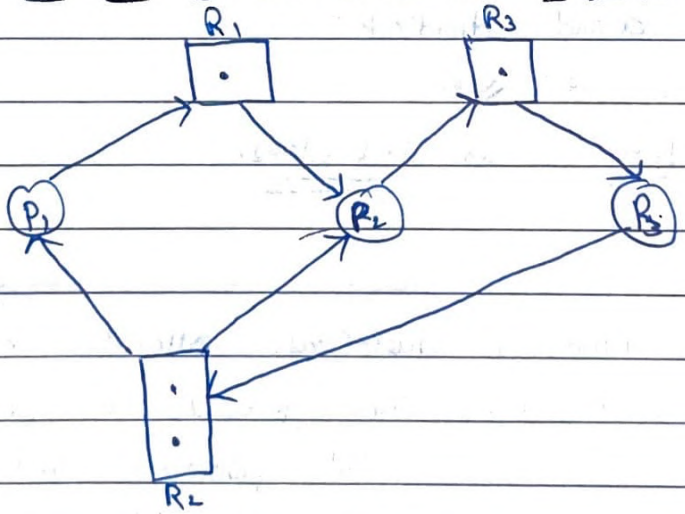
b)



Cycle & No Deadlock

→ Not in deadlock, P_5, P_4, P_3, P_1 is possible exec. sequence

c)



cycle & Deadlock

→ Clearly no one can move, So in Deadlock.

PAGE NO. :
DATE : / /

- If graph has no cycle ⟶ No Deadlock
  ↳ bcz. no circular wait
- Graph has cycle & Resources are Single Instance ⟶ Deadlock
- Graph has cycle & Resources are multi instance ⟶ Deadlock possible

→ So, if we have multiinstance per resource type, then we need deadlock detection method, we can't judge by cycles.

Gate CS'21
Qu: Consider a system with 3 processes that share 4 instances of the same resource type. Each process can request a maximum of k instances. Largest value of 'k' that will always avoid deadlock is:

→ Assume we give max-1 to each process & then have 1 extra to avoid deadlock:

$$3(k-1)+1 \leq 4$$

$$k \leq 2 \quad \text{so } k=1, k=2.$$

NOTE → To solve this kind of questions, allocate max-1 to each process and then give 1 to break deadlock. If we have 'n' processes and max_i is maximum resources required by i^{th} process then to avoid deadlock:

→ Pigeonhole principle

$$\text{resources} \geq \left[\sum_{i=1}^{n} (max_i - 1)\right] + 1$$

↳ ***

To be not in Deadlock | Maximum ~~Request~~ Process / Minimum Resources

↳ GATE for. que.

PAGE NO.:
DATE: / /

CS 92

Que: A computer sys. has 6 tapes with n processes competing for them. Each process may need 3 tapes. Max. value of n to guarantee no deadlock is:

→ $2n+1 \leq 6$

$n \leq \frac{5}{2}$

$n = 2$

Ans

Gate CS 93

Que: Consider a system having 'm' resources of same type. These resources are shared by 3 processes A, B, C which have peak demands 3, 4, and 6 respectively. For what value of m deadlock will not occur.

⇒ $(3-1)+(4-1)+(6-1)+1 \Rightarrow \geq 11$

i.e. till m=10, possibility of deadlock is there.

Gate CS05

Que: Suppose n processes share m identical resource unit. Max. req. of Pi is s_i, where $s_i > 0$. Which one of the following is sufficient for ensuring no deadlock.

⇒ $\sum_{i=1}^{n} s_i < (m+n)$

↳ From prev. pg. formulae

$\sum_{i=1}^{n}(s_i - 1) + 1 \leq m \Rightarrow \sum_{i=1}^{n} s_i - n + 1 \leq m \Rightarrow \sum_{i=1}^{n} s_i < m+n$

PAGE NO. :
DATE : / /

Gate CS 06

Que: Consider snapshot of system running 'n' processes. Processes 'i' is holding 'xi' instance of resource R. Currently all instances of R are occupied. Further for all i, 'i' has placed a request for 'yi' instances while holding the xi instances it already has. There are exactly two processes P & q such that $y_p = y_q = 0$. Which one is necc. condition to guarantee system is not approaching a deadlock?

→ $\exists_k (x_p + x_q \geq \min_{k \neq p,q} y_k)$

However, still deadlock in future is possible.

☆ There are mainly four strategies to ded with deadlocks:

a). Just ignore the problem altogether.

- Restart the system manualy if system seems to be in deadlock.
- Used by most OS, including UX.
- Also called Ostrich Algorithm.

I am while ostrich driving

↳ It puts head in sand & pretend there is no problem at all.

b). Deadlock Prevention: We can prevent deadlock by negating one of the 4 necessary conditions for deadlock.

PAGE NO. :
DATE : / /

i) Attacking Mutual Exclusion : Not possible, because nature of Resource may not be sharable like Speaker.
So, Not all Devices can be shared/ Spooled.

ii) Attacking the hold & wait condition : There are two ways to do so:

a) Hold but never wait : Can be achieved by getting all resources at start and then never ask, but we don't know req. in advance.
And also it will reduce efficiency as we may not need all resource at same time.

b) Wait but never hold : When you want to wait, release the resources you are holding. This is also not feasible. Possible, but low rsrc utilization.

iii). Attacking No Preemption Condition : This is not a viable option, b/c ~~process~~ resource maybe halfway it's job like printer printing.

NOTE that Resource related conditions can't be attacked at all. So just say sorry for them directly.

iv) Attacking ~~Hold and~~ Circular Wait Condition : This can be attacked by using a trick/ algorithm we will discuss now.
Impose a total ordering of all resource types & require each process to request resource in ↑ order.

PAGE NO. :
DATE : / /

In simple words, give no. to each resource and process having i^{th} numbered resource can't request less than i^{th} resource, it can only request greater than i^{th} resource. This protocol will never lead to circular wait and hence we have prevented deadlock.

Practical & Easy to apply.

c) **Deadlock Avoidance:** It is third technique to deal with deadlock. It is bit relaxed than deadlock prevention. It does not attack any condition.

It avoids deadlock in such a way that before granting any resource, it makes sure system is still in safe state (i.e. can not go in Deadlock).

Here we can have all 4 conditions and still no possibility of Deadlock.

We will do this by Banker's Algorithm.

↳ Reason?

↳ There could be a cycle with multiple instances.

→ It requires that each process declare the max. no. of resource of each type that it may need.

↳ This a priori info. must be available.

→ It dynamically examines the resource allocation state to ensure that there can never be circular wait condition.

PAGE NO.:
DATE: / /

⇒ Checking if system in safe state or Not.

→ Safe sequence is a sequence of process in which if we fulfil their request then we will never face deadlock.

→ State is safe if ∃ atleast one safe sequence.

Ex: Total tapes = 12

(max. no. of resrces a process may or may'nt need in future simultaneously)

	Current usage	could ask for
P_0	5	5
P_1	2	2
P_2	2	7

Available: 12 - 5 - 2 - 2 ⇒ 3

Safe. Seq: P_1 P_0 P_2

↳ If we follow this seq, then def. there will be No deadlock.

→ In other requests we again have to run this algo to know if we grant req, then we are safe or not.

⇒ Algorithm to know if we are in safe state or not:

n: no. of processes.

work = available resource qty

finish [1 ... n] = false.

allocated [1 ... n] = current usage.

need [1 ... n] = could ask for.

PAGE NO. :
DATE : / /

```
// find a process that can complete its work now
while (find i so that finish[i] = false && need[i] <= work)
{
    work = work + allocated[i];
    finish[i] = True;
}
if (finish[i] = True for all i)
    return "Safe";
return "Not safe";
```

→ Instead of allocated & need, they can give max and allocated. Any two out of 3 can be given and are related as:

need[i] = max[i] - allocated[i].

→ If we want to write algo. for multiple resources, we can easily do so by using matrix instead of vectors and vector for work instead of var.

★ **Banker's Algorithm**: Given state is safe, and then we have request from some process, then we can:

→ ~~Check~~ Pretend request is Granted (update table)
→ Check if new state is safe.
  ↳ if safe, Grant
  ↳ else, Deny the request.

→ It was given by Dijkstra in 1965 on analogy with Banks & loans.

PAGE NO. :
DATE : / /

Gate CS 96

Que: Consider following state of system:

	Max. Need			Alloc.			Could ask for		
	R_0	R_1	R_2	R_0	R_1	R_2	R_0	R_1	R_2
P_0	4	1	2	1	0	2	3	1	0
P_1	1	5	1	0	3	1	1	2	0
P_2	1	2	3	1	0	2	0	2	1

Available R_0: 2, R_1: 2, R_2: 0

	2	2	0
P_1	0	3	1
	2	5	1
P_2	1	0	2
	3	5	3

a) Is this in safe state?

⇒ Safe seq: P_1 P_2 P_0

⇒ Yes, as safe sequence exists.

b) What will system do on a request by process P_0 for 1 unit of R_1?

⇒ Updated Table

	Alloc.			need		
	R_0	R_1	R_2	R_0	R_1	R_2
P_0	1	1	2	3	0	0
P_1	0	3	1	1	2	0
P_2	1	0	2	0	2	1

Available: R_0: 2, R_1: 1, R_2: 0

No safe seq. exist as if every one asks for their max resource, can't fulfil anyone & will go in deadlock.

So, goes in unsafe state.

Request Denied.

PAGE NO. :
DATE : / /

NOTE ↔ If safe seq. is $[P_2, P_1, P_0, P_1 \dots P_n]$, then any request by P_2 will be granted w/o even checking obviously, because that is what we want. Request from anyone else will be checked.

NOTE ↔ If available < request then anyway request will be denied w/o even checking.

MSQ Imp
Qu: Let $\langle P_1, P_2 \dots P_n \rangle$ be the only safe state of a system. Then which of the following are/is Always TRUE?

a) If P_2 is the first process to request some rsrc, then P_2 always have to wait.
⇒ No, we will Run & check the algorithm.

b) P_1's req. will always be satisfied w/o even checking.
⇒ Yes. (if req < available).

c) we can never satisfy any request coming from P_n w/o satisfying request from P_2.
⇒ No, nothing like that.

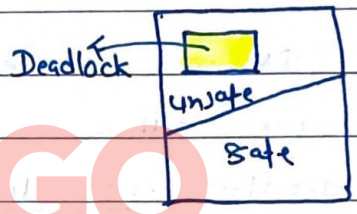
d) Processes' requests will always be satisfied in same order as safe state seq is we can't satisfy any out of order process request.
⇒ False.

PAGE NO. :
DATE : / /

System in Safe state → Deadlock Not Possible
System in unsafe state → Deadlock Possible.

In Deadlock avoidance we make sure system never goes in unsafe state.

No deadlock in unsafe state bcz process may never ask for its maximum needs and that too if asks then maybe not all process simultaneously.



Gate CSE

Qu: Which of the following is NOT true?

a) In deadlock prevention, the request for rsrces is always granted if resulting state is safe.
⇒ false.

b) Above stmt for deadlock avoidance ⇒ True

c) Deadlock avoidance is less restrictive than prevention
⇒ True

d) Deadlock avoidance require knowledge of rsrc req. apriori. ⇒ True.

PAGE NO. :
DATE : / /

→ Deadlock Prevention reduces efficiency, Avoidance is more efficient than prevention, but still it's inefficient, bcz. may be system can go to unsafe state to come back to safe state. So, rejecting all unsafe states reduces efficiency.

It's a Tradeoff.

D) **Deadlock Detection & Recovery:** It is 4th and final method to deal with deadlock. It is more flexible than avoidance.

Here we allow every request and keep checking for deadlock periodically. If it is deadlock, then will somehow recover from it.

- By killing process (crudest)
- Restarting
- Preempting resource
- Rollback.

For detection, we check the current state of system. We have matrix of allocated resources and requested resources (in avoidance we had needed resources).

Ex: Detect if system in Deadlock or Not?

⇒

	Allocation			Request			Available		
	A	B	C	A	B	C	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	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

⇒ As progress can be made in order $[P_0, P_2, P_1, P_4]$ (Many other orders)

so, not in Deadlock as of now.

PAGE NO. :
DATE : / /

File Systems.

→ Hardware related part of file systems is already covered in DBMS (platters, tracks, etc.). → only Magnetic Disk for Gate.

→ In this portion, we'll see the implementation part

- **File:** Data in some format. Directory/Folder is also considered as file with some special flag to distinguish.

- **File Systems:** Manging files on the hardware. It provide us the way to store file & it's metadata. It helps in retrieval of file.

→ There are many file operations like Create, Rename, Open, Read, write, etc.

→ Sector or block is the minimum unit of h/w. i.e. data is read or stored block wise. This is unit of atomicity of file.

→ Metadata of file/directory is all the info. of data. Like name, size, actual disk blocks address, protections, dates & times, etc.

→ Meta data of file is stored in datastructure known as inode. (index node).

→ Now, we'll see some filesystems & then how modern OS implement these filesystem.

PAGE NO. :
DATE : / /

→ Each file or directory has exactly one inode & each inode is given an inode number.

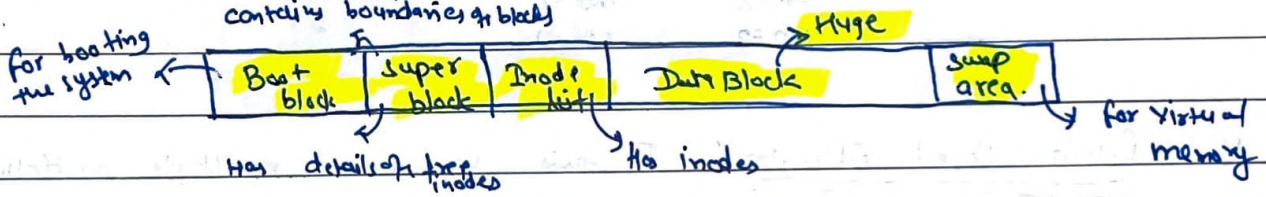
File | Directory = inode + actual data blocks.
↳ Metadata

→ Inode is basically a D.S. that describes the metadata of files.

→ Inode of a file contains data block address & inode of a directory contains inode table which contains inode no. of all files within it.

→ Basic job of FS includes converting the name of file to the actual data block address of Disk.

→ Disks is generally divided into various parts. The most common division of disk is:



→ Directories can be implemented as:

- Single directory for entire system.
- Single directory for each user.
- Hierarchical namespace (forms a tree or graph, ej links allowed)
  ↳ used in our PCs.

PAGE NO. :
DATE : / /

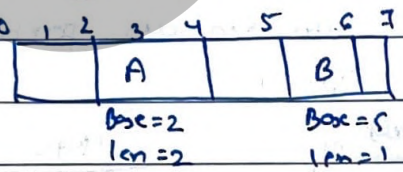
★ **File implementation**: Deals with allocation of disk spaces to files.

a) Contiguous
c) Linked.
e) Indexed.
b) Extent Based (Variation to contiguous)
d) FAT (variation to linked)
f) Multilevel (Variation to Indexed)

a) **Contiguous Allocation**: FS allocates space all at once. Inode contains first Disk block address & size of file.

Pros:
- Easy to implement
- Fast sequential & random access
- Not any storage overhead in inode ★

Cons:
- Can't grow file always
- External fragmentation.



b) **Extent Based Allocation**: In this, we have multiple continuous regions per file like segmentation. Each region is called extent. Inode contains array for base & length of each extent (like seg. table in OS).

Pros:
- Easy to implement
- Files can grow overtime
- Random address can be easily generated
- Low storage overhead
- Fast seq. access

Cons:
- External fragmentation is not fully solved.

Random address = like array, tell the addr of 8th line.

PAGE NO.:
DATE: / /

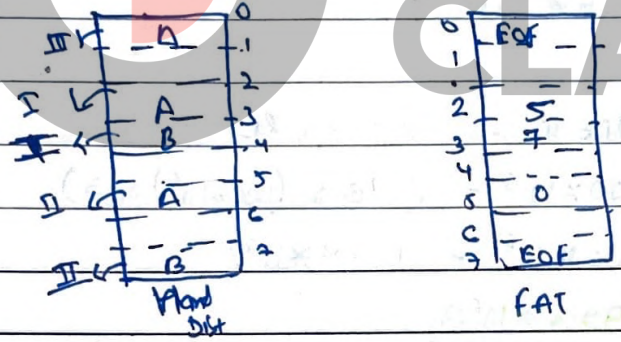
c) **Linked Allocation:** Here each block of a file has a link to next block. All blocks form a link list. Inode has address/link of first node only.

Pros:
- No External frag.
- Files can be grown easily.
- Easy to implement

Cons:
- Large storage overhead
- Slow sequential acces
- Hard to compute random address.

d) **FAT Table:** file allocation table. One for one hard disk. It contains address of next file pointer at current pointer & EOF at last pointer. Inode has 1 pointer address.

Ex:



Inode of A has 2 &
Inode of B has 3.

It also does pointer chasing, but it need not to bring all blocks in memory to go to next block, can check from FAT only.

This table has one entry for each block. Linked list of entries. Used in MSDOS & windows as. → Now NTFS

Disk has a section that contains FAT only. It needs to be cached for good efficiency. Using info in FAT, we can get random address.

PAGE NO. :
DATE : / /

Pros: • Fast random access

Cons: • Storage overhead of FAT Table. • Slow seq. access.

Que: If entry size = 16 bit & block size is 512 B, what's the max. size of FS?

⇒ 16 bit means 2^{16} blocks can be there atmost.
So $2^{16} * 512$ B. is FS size = 32 MB.

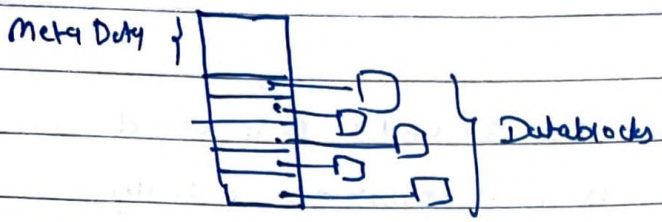
Gate 14
Que: Entry size = 4B, FS = FAT based, Disk size = 100×10^6. Block size = 10^3 B. Max. file size is ?

⇒ Max. file size = Disk size - FAT table
$= 100 \times 10^6 - (4B * (100 * 10^6)/10^3)$
$= 100 * 10^6 - (.4 * 10^6)$
⇒ $99.6 * 10^6$ B.
Ans.

c) Indexed Allocation: Linked allocation solved issue of external fragmentation & size dec. problem of contiguous allocation. But it do not support random access efficiently. Indexed allocation solves this problem by bringing all pointers in one block called the Index Block.

PAGE NO. :
DATE : / /

Inode directly contains pointer to datablocks.



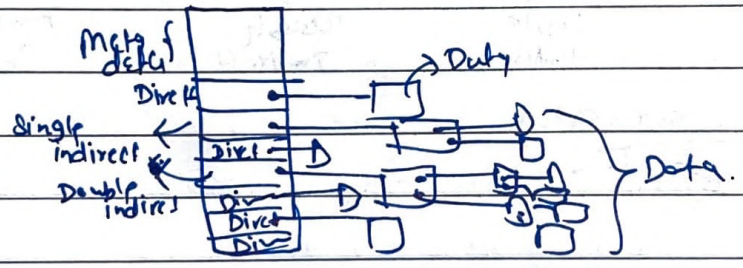
Inode.

Pros:
- Both random & sequential access is easy.

Cons:
- Inode becomes larger.

ii) Multilevel Index: It also contains pointers to data, but it can have a pointer to block which further contain data pointer (called single indirect) & double indirect & soon.

This solves file size's ~~det~~ problem with inode size.



PAGE NO. :
DATE : / /

Practice of File Systems:

Qu: Consider a file system with 2048 B blocked & 32 bit disk & file block pointers. File has 12 direct pointers. a singly indirect pointer, a doubly indirect pointer & a triply indirect pointer.

a) How large of a disk can this file system support?

⇒ 2^{32} (Max. Blks) * 2048 B (Block size) → $2^{11} * 2^{32}$ B ⇒ 8 TB.

b) What's the maximum file size?

⇒ Pointers per block = $\frac{2048}{32/8}$ → 512 pointers

FS = (12 * 2048) + (512 * 2048) + (512 * 512 * 2048) + (512 * 512 * 512 * 2048) B

(Direct, Single Indirect, Double Indirect, Triple Indirect)

= 24KB + 513 MB + 256 GB

Que: Consider a FS with 512 B blocks. Inode of a file holds N direct data blocks & a pointer to single indirect block. Single Indirect block (Pointers to) can have M data blocks. What is max. file size?

⇒ N * 512 + M * 512 = (M+N) * 512 B

PAGE NO.:
DATE: / /

Que: We have seen diff filesystems that support fairly large files. Now let's see how large a file various types of filesystems can support.

Assume, for all of the questions in this part, block size is 4KB.

a) Consider a really simple fs. "directfs", where each inode only has 10 direct pointers.

⇒ $10 \times 4KB = \underline{40KB}$.

b) Consider another fs, "extentfs". Extents have a pointer & a length (in blocks). Assume length field is 8bits. Assuming that an inode has exactly one extent, tell max. file size.

⇒ 1 extent can have max value of length = 11111111 = 255 Blocks

FS = $255 \times 4KB = \underline{1020KB}$.

c) Consider a fs, that uses direct pointers, single indirect & double indirect pointers. We call this fs, "indirect fs". Each type of pointer are exactly 1. Pointer size is 4B. What is max. file size?

⇒ Pointers per block: $\frac{4 \times 2^{10}}{4} = 2^{10}$ Pointer

~~fs = $(1 \times 512) + (128 \times 512) + (128 \times 128 \times 512)$ B~~

~~= $2^9 + 2^{16} + 2^{23}$ B.~~

~~FS = (1 ×) + k~~

FS = $(1 \times 4KB) + (2^{10} \times 4KB) + (2^{10} \times 2^{10} \times 4KB)$

A

PAGE NO. :
DATE : / /

Gate C'02

Que: In the Index allocation scheme of blocks to a file, the max. possible size of file depends on:

⇒ Size of block, No. of blocks used, size of block pointer.

Gate C'08:

Note: Unix file system uses multi level indexing.

Gate C'17

Note: External frag. is NOT present in linked & Indexed allocation scheme.

Gate C'21

Que:

Birthday/ ~~Algo in C++~~

Que: Suppose a file system can have three disk allocation strategies, continuous linked & index. We have just read the info. for a file from its parent directory. For continuous & linked allocation this gives address of 1 block & for indexed allocation, this gives address of index block. Now, we want to read 10th data block into the memory. How many disk blocks do we have to read for each strategy. (in worst case)

⇒ a) Continuous → 1.
b) Linked → 10.
c) Indexed → 2.

I/O means 1 op^n for read & 1 for write

PAGE NO.:
DATE: / /

Ques: Consider a file currently consisting of 100 blocks. Assume inode is already in memory. Calculate how many disk I/O operation are req. for contiguous, linked & single level indexed allocation strategies, if for one block, the following conditions hold. Assume data to add in block is stored in memory.

a) block is added in beginning. 51st block middle end.

→ Continuous: 201=(100×2+1) (2 shift) | 51=(2×50+1) | 1 (subject to free space available)

linked = 1 | 52 | 102

Indexed: 1 | 1 | 1

b). block is removed from beginning middle end.

→ Continuous = 198=(2×99) | 98=(2×49) | 0 (reduce length var.)

linked = 1 | 52 | 100 (99+1)

indexed = 0 | 0 | 0

NOTE: Using Virtual file system, one OS can support multiple file systems.

☆ How to keep track of free space, such that we can find free space quickly & low storage overhead. (in DBMS too)

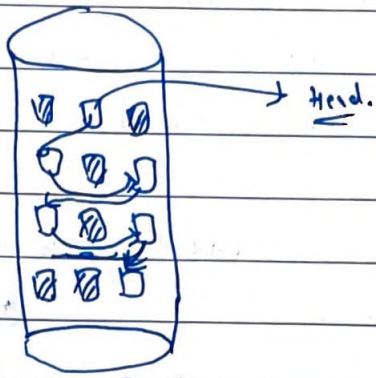
a) **Bit Vector:** Vector [No. of blocks]

Vector[i] = 0 if free, 1 if occupied.

Disadvantage is space to store vector.

PAGE NO.:
DATE: / /

b) **Free space linked list:** Have head of first free block & then ~~next~~ this free block points to next free block & that to next & so on.



Pro:- (No extra storage overhead)

We can optimise this as:

b1) **Grouping:** Store n free blocks in first free block. Last entry points to next block of free blocks.

b2) **Counting:** Specify start block and no. of contiguous free blocks.

★ **Some FS** (No need to memorise):

Windows ⇒ NTFS (NewTech), FAT32.
MAC ⇒ HFS (Hierarchical)
Linux ⇒ ext2, ext3, FAT 32
Unix ⇒ ext2, ext3, UFS, ZFS.

These notes are the property of GOClasses. Please do not share them without their consent.

For any questions or issues regarding my handwritten notes, feel free to reach out via email at karan757527@gmail.com or DM me on telegram at @karan757527 (or by clicking here).

I would love to know if you liked my notes. Your feedback and appreciation are invaluable, so don't hesitate to share your thoughts. Click here to connect with me on LinkedIn or you can find me as @agrawalkaran.

Wishing you success on your journey ahead!

- Karan Agrawal