Introduction

- The abstract view of system consists of three components:
(1) CPU
(2) Memory
(3) I/O
With these three components we can run all the possible programs and perform all the calculations, such as calculating prime numbers, serving to n/w, etc, but it would be on us to manage how these programs run:
(1) Who'd use CPU next?
(2) Which part of m/m should my program reside in?
(3) Which process should access a m/m loc. & when?
(4) Store & retreive files from disk.
...and a whole lot more.

- Operating System is a software which allows us to run our programs without worrying about all this. It takes care of managing various resources, such as CPU, m/m, I/O, etc.

It's not that OS is created with H/W in mind only, even H/W is designed keeping OS in mind, such as MMU in CPU, interrupts, etc.

* Dual Mode of Operation

- The OS performs critical as well as non-critical operations. It is undesirable for non-critical operations such as running user programs, reading & writing to program m/m etc to affect the execution of critical operations such as process scheduling, m/m access & protection, etc. That's why the OS is split into

two modes of execution which is controlled using Mode bit:

User Space: The CPU in this mode allows for running of user programs such as reading/writing to user memory.

Kernal Space: The CPU in this mode allows for running of Kernel code such as process scheduling, I/O handling, etc, along with user program instn.

- The CPU is in User or Kernal mode. Whenever an instn is executed by CPU, it has to check if it is in User Mode or Kernel Mode. This check can be done in H/W, and is very fast.

So the CPU is designed in mind and can have two or three or even one mode of execution, based on OSes it wants to support.

We still need a mechanism to switch to Kernel Mode (priviledged Mode). This happens using System Calls.

* System Calls

The Kernel acts as a big software and System Calls is the set of APIs provided by the Kernel to the user to interact with the Kernel space.

Whenever a User Space program calls a System Call it results in CPU executing a S/W interrupt instn, for example TRAP 80, etc. This results in a trapor S/W

interrupt which causes the CPU to change Mode bit to change to 0 (Kernel mode). The ~~Kernel~~ CPU now runs in Kernel mode and can run priveledged instns (Kernel instns).

Once the ~~System~~ ISR for System call is run, a prive ledged instn switches the Mode bit to 1 (User mode), and user space program continues to run.

- In Kernel mode, all instns: priveledged or non-priveledged, can be executed.

- Further exceptions & other s/w interrupts can also cause the CPU to switch to Kernel mode.

- Types / Categories of System Calls
  (1) Process Control : load, execute, fork, etc.
  (2) File Management : read, write, seek, etc.
  (3) Communication : pipes, fifos, etc.
  (4) Device Management : read, write, etc.
  (5) Information Management

* Cases when PC value changes :
(1) Context Switch : When we start a new program, and stop previous. This is critical.

(2) When you call a function. This is non-critical.
(3) When an ISR runs. This is critical.
(4) Normal Execution: PC+1

* System Boot

Once OS is generated it needs to be run on Hardware, i.e. how would on system start the H/W know that it has to load the Kernel code addr in PC & run it.

When the system is started a predefined instⁿ is loaded into the Instruction Register which is responsible to load the Kernel. These set of instⁿs is stored in a ROM since the state of RAM on bootup is unreliable.

This set of instⁿs is called Bootstrap Program. It is stored in a ROM m/m. The bootstrap program has a variety of tasks, such as it can diagnose the state of the m/c. It can be the case that the OS code is very small and the entire OS is stored in the ROM. Generally for large OSes, they are stored in the disk and need to be loaded into RAM.

It generally is the case that the ROM bootstrap program can read a fixed location on disk (say 0th block), which is called the boot block. The code in boot block might load the entire OS, but more generally it has addr. of the block with more sophisticated bootstrap program (compared to ROM bootstrap code), which then loads the entire OS into m/m and sets the Program Counter to Kernel code. Ex. of such bootstrap program is GRUB.

Processes

* Process Execution

- User has compiled and stored some program on the disk.
- User requests the OS to execute the code through some system call.
- The OS receives the system calls and starts running "loader" code.
  - Loader copies the bytecode from disk to some m/m location.
- Operating system schedules the program for execution.

* Process

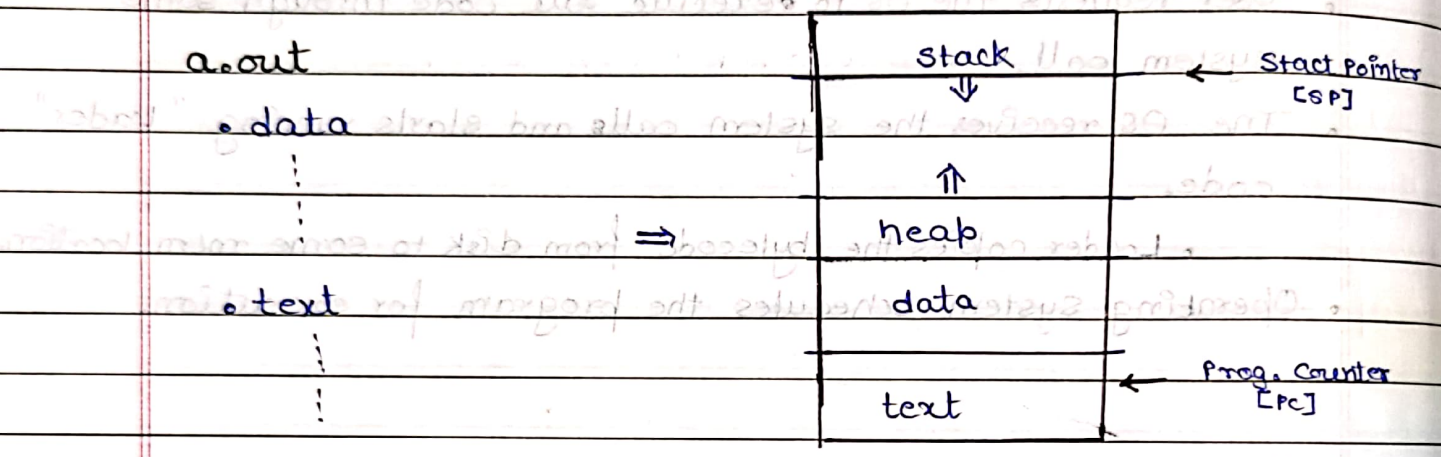
A program in execution is Process. (This is a broad defn.)
A process contests for resources, i.e. CPU, m/m, and I/O.
Executing can mean different things : It might mean that CPU is executing bytecode of a particular process; or the process is in memory.

- Context of a Process : when a process is executing on the CPU, the CPU registers are loaded with the processes's data. The MMU has info. regarding its location in m/m, the cache has data & instn for that process. The process might have some files open, and might be waiting for I/O to get some data. All this and more make up a processes context.

- Process Id: Every process is uniquely identified using a process identifier.

* Process Structure

When the loader copies a process from disk to m/m, it copies it in a specific structure



When OS copies a process into m/m it allocates a little more more m/m than needed at the time, so a process doesn't need to call malloc each time more m/m is needed.

* Process Control Block

The OS has the job of keeping track of process context and it does so by using a structure called Process Control Block for each process.

The PCB for a process consists of following info:

(1) Process Id.
(2) Program Counter
(3) Open Files
(4) Register Values
(5) Process State
(6) M/M limits & more.

- The OS stores one PCB per process somewhere in the RAM, and generally OS maintains a ~~list~~ linked list of processes.

* Fork System Call & Exec Syscall

- When the fork system call is executed by a user-space program, the call creates a new "child" process which is a replica of the original program. The fork call returns 0 to the child process, and returns either a -ve value if the creation of child process led to some error, or a +ve value which is the PID of the child process. Both the parent and child process execute from the point of return.

- To execute a program, the loader process does two things:
  (1) Create a new process. (fork syscall)
  (2) Load the program into the process. (exec syscall)

Ex.
```
int main() {
L1:     int c_id = fork();
L2:     printf("%d", c_id);
}
```

	Parent	Child
L1:	fork called	
	replica created.	
	c_id = 2537	cid = 0
L2:	2537	0

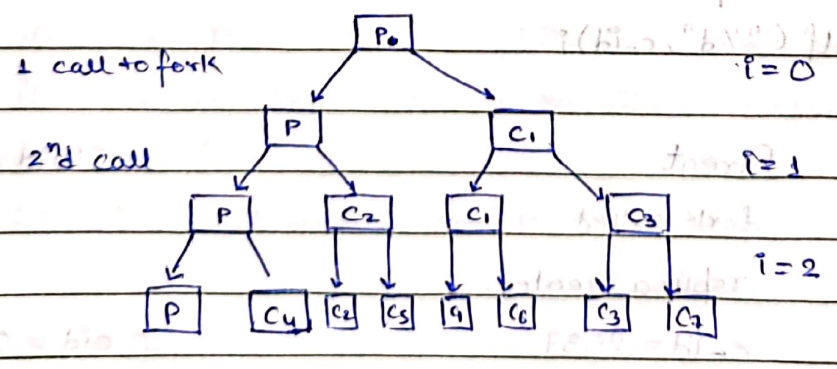
- Creation of "exact replica" means that whatever was run in parent process upto this point has also been executed in child process. From this point on all the instn would also be same for them, there execution may differ.

- Fork generally works on "copy on write" mechanism. Although a new PCB is created by the fork system call, ~~as long~~ the process in m/m isn't replicated elsewhere initially. As long as both the process only perform read on process m/m it isn't copied. The process m/m is copied only when either parent or child writes to the process memory.

- There's no specific order in which the parent & child process run.

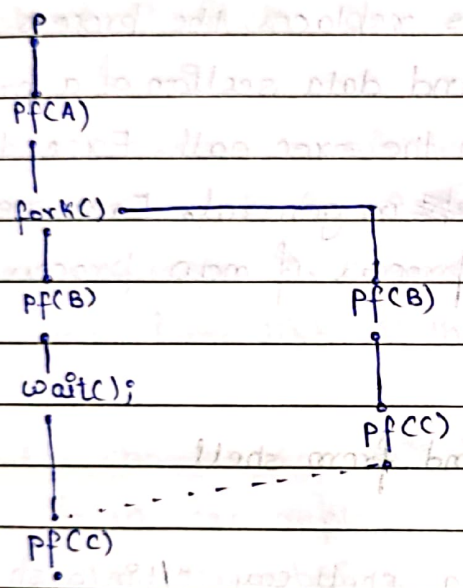
Ex. How many times "HW" is printed?

```
for (i=0; i<3; ++i)
    forK();
pf ("HW\n");
```



∴ 8 times.

Ex.
```
int main(){
    pf("A");
    int ch= forK();
    pf("B");
    if (ch) wait(NULL);
    pf("C");
}
```



So the output depends on order of execution. Which process is scheduled when, etc.

Possible Outputs:

ABBCC.

ABCBC.

- **Wait:** Wait system call blocks the execution of a process until one of its child process exits. It returns the pid of exited process. If a process has no child, calling wait on it has no effect.

- **Zombie Process:** When a process creates a child, parent keeps executing and the child terminates. When child terminates all resources allocated to it are freed except in the process table where the PCB for the child still exists. The PCB exists until the parent calls wait ~~on the~~ syscall. If the wait call isn't made then the child process becomes a Zombie.

- **Orphan Process:** A child process becomes orphan if its parent terminates while the child continues running.

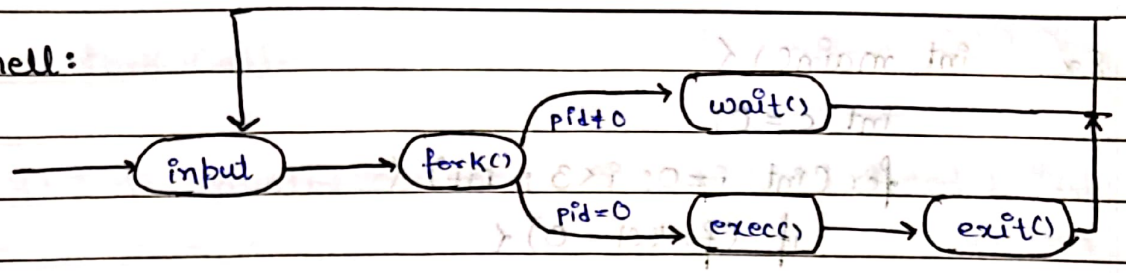
- **Exec:** We know that fork syscall replicates a process to a child process with a new PCB in Kernel space and ~~a new~~ copies the process m/m in User space.
Exec systemcall on a process replaces the process m/m of current process with text and data section of a program, path to which is provided in the exec call. Exec doesn't modify the PCB of the process, ~~exec~~ in general. Exec never goes back to executing original process if new process is loaded successfully.

Ex. Execution of ls command from shell.

When you type ls & ↵ on a shell say, "/bin/bash" the following happens:

(1) The shell process calls fork, after which in parent it calls wait.
(2) In child process it calls exec with path to ls command, say "/bin/ls", along with parameters to it.
(3) The exec command replaces the child's process m/m with ls command's text and data section with empty stack & heap. Since PCB isn't modified the open files don't change, so STDin/out/err files remain same.
(4) ls executes & outputs to STDout, and child terminates.
(5) The parent unblocks & asks for input again.

Shell:



* Context Switch

- Suppose we have a single CPU, but multiple processes in the memory. Say $P_1, P_2, \ldots P_n$.
  Suppose currently P_2 is running on CPU & it has to wait for some file to load from disk. At this time P_2's next instn can't run on CPU.
  If CPU chooses to wait for file to load, we are wasting CPU time. So we can replace P_2 running on CPU to say P_{10} running on CPU for the time being.

- The process of replacing currently executing processes context, such as register values, cache entries, TLB entries, open files etc by saving it in its PCB and loading another process's state from its PCB is known as Context Switch.

- Context Switch is a slow process.

- Context switch can happen in various cases: Current process is asleep (blocked), it is terminated, etc.

Ex.

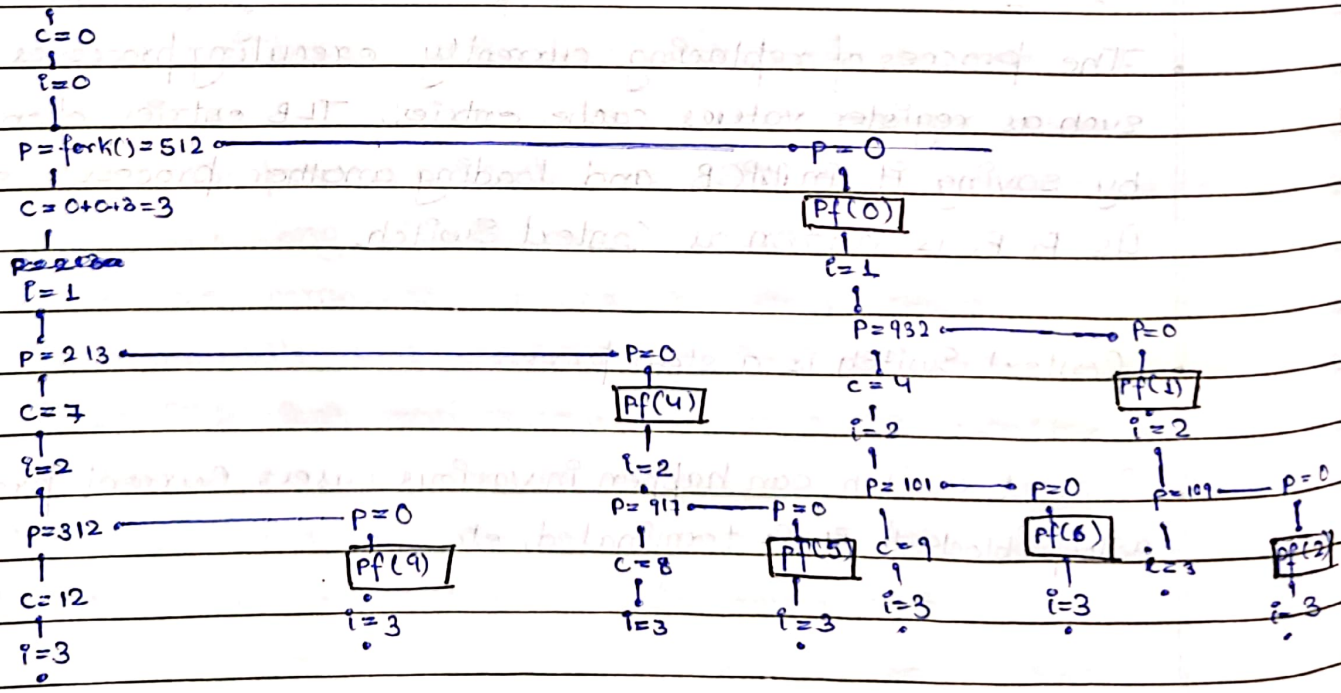
```
int main() {
    int c = 0;
    for (int i=0; i<3; i++) {
        if (fork() = 0) {
            pf("%d ", c+i);
            fflush(stdout);
        } else {
            c = c+i+3;
        }
    }
}
```

(a) How many times fork is called?

(b) Which of the following is <u>not</u> a valid (possible) output sequence?

(1) 0 1 4 6 5 9 2 (3) 9 4 0 5 1 2 6

(2) 4 0 1 6 9 2 5 (4) 0 1 5 4 9 6 2



∴ 7 fork Calls.

(4)th is invalid sequence. 5 can't be printed before 4.

Ex.

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

(A) 0 1 0 1

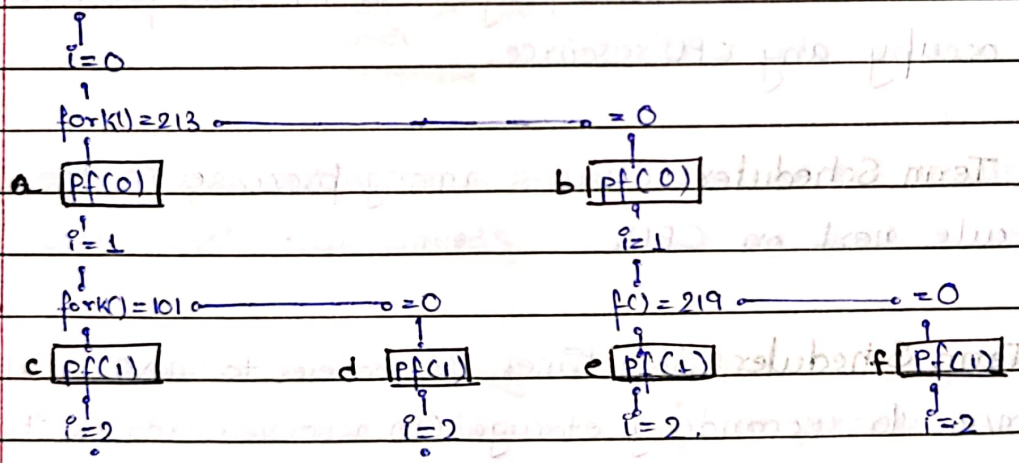
(B) 0 0 1 1 1 1

(C) 0 1 0 1 1 1

(D) 0 1 1 0 1 1

(E) 0 1 1 1 0 1

Which of the above seq. are possible in case of successful execution?



A is not possible in case of successful execution.

B is possible: a b c d e f

C is possible: a c b e d f, a d b f c e, etc.

D is possible: a c d b e f, b e f a c d.

E is not possible: 0 1 1 1 0 1

3rd 1 can't be before 2nd 0.

* CPU Scheduling

CPU execution time is a resource and a valuable one. It needs to be distributed among processes according to various parameters. CPU scheduling algorithms answer the question of how.

- When a user submits a task (i.e. holds info about the program to run) to OS to be executed on CPU, its expected runtime is also submitted.

(1) Long Term Scheduler: decides which of the submitted tasks should be in m/m. While a program is in task form it doesn't occupy any CPU resource.

(2) Short Term Scheduler: Chooses among processes in the m/m to execute next on CPU.

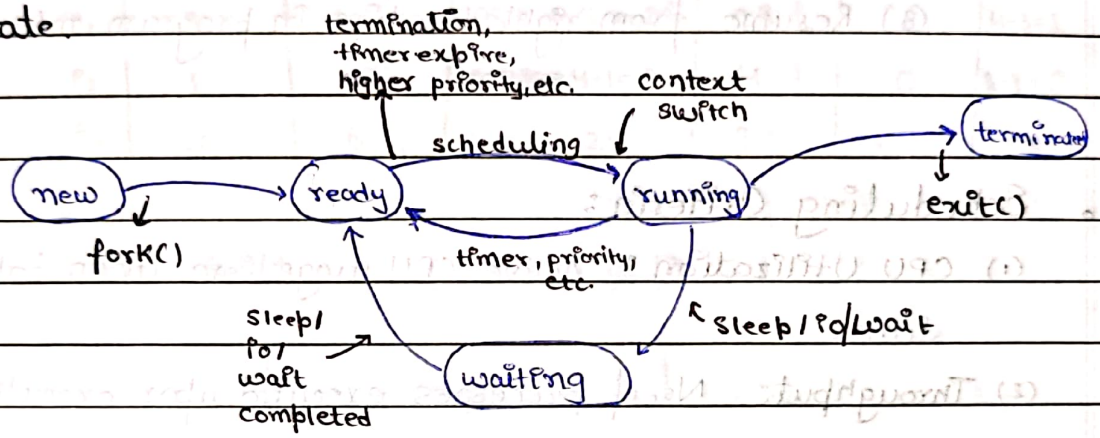
(3) Mid Term Scheduler: Sometimes OS chooses to move a process in m/m to secondary storage (in reserved area) & it can choose to bring these processes back in m/m as per requirement.

- Degree of Multiprogramming: No. of processes there can exist in the m/m. Longterm scheduler tries to optimize this.

- When a call to fork call is made it means that a PCB is created and added to a queue of processes which are ready to execute. (Ready State). Fork here is an event.

- When a process's code is executed by CPU, it is in Running State. The short term scheduler picks a process from ready queue on some event (ex. current process terminates) and performs context switching by which it sets Program Counter to processes inst^n.

- A process can wait for IO events, or sleep, and be in Waiting State.



new is just initial state where no process exists. It only exists when fork() call succeeds

- The OS maintains the state of the process in its PCB.
- The OS maintains queues for ready state: ready queue, waiting state: waiting queue and an additional queue called Job Queue to maintain list of submitted tasks.

- CPU scheduling decisions may take place when:
(1) Some process goes from running to waiting state.
(2) Some process goes from running to ready state.
(3) Some process goes from waiting to ready state.
(4) Some process terminates.

- All events, such as fork(), exit(), timer expire, etc are either syscall or H/W interrupt which switch Mode bit to 0, and OS code (scheduler) runs.

- **Dispatcher:** Performs three things:
  (1) Context Switch
  (2) Switch to User Mode
  (3) Resume from right location in program m/m.

- **Scheduling Criteria:**
  (1) **CPU Utilization:** Active CPU usage time upon total time.
  (2) **Throughput:** No. of processes executed upon execution time.
  (3) **Turnaround Time:** For a process the time b/w it is submitted to the time its terminated.
  (4) **Waiting Time:** A process might not always be in running state throughout its lifecycle. It can be in running queue. WT is the time a process spends in ready queue.
  (5) **Response Time:** When a process comes from new to ready state, the first time it gets to running state.

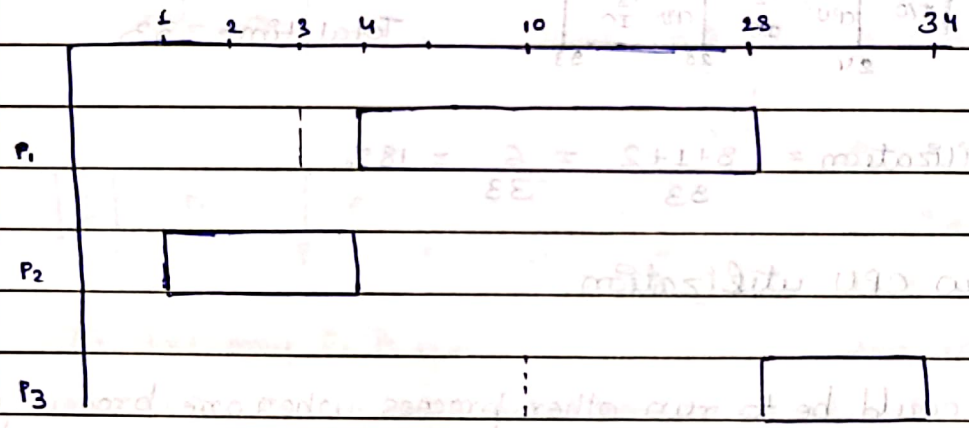
- **CPU Burst Time:** The time for which a process requires to execute on CPU.

* First Come First Serve

Assign CPU time to processes in order of their arrival time in ready queue.

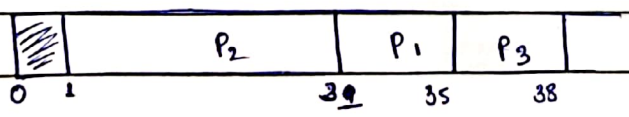
Once a process is assigned CPU time it is run until termination.

Ex.	AT	CPU BT	TAT = Term. - AT	Compl. Time	WT =	Resp. Time
P_1	3	24	28 - 3 = 25	28	1	4-3=1
P_2	1	3	4-1=3	4	0	1-1=0
P_3	10	6	34-10=24	34	18	28-10=18



- **Convoy Effect:** When a very long process is scheduled, a lot of short processes keep getting added to the ready queue and ready queue keep increasing. Results in lower CPU utilization.

Ex.	AT	CPU BT	CT	TAT = CT - AT	WT = TAT - BT	RT
P_1	3	4	35	35-3 = 32	32-4 = 28	31-3 = 28
P_2	1	30	31	31-1 = 30	30-30 = 0	31-1 = 0
P_3	5	3	38	38-5 = 33	33-3 = 30	35-5 = 30



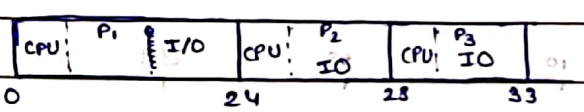
Avg RT = (28+0+30)/3
=> 19.33

Note: Till now we haven't considered the case where process does I/O, along with CPU execution.

Burst Time = CPUBT + IOBT

Ex. Assume FCFS doesn't take into account IO BT.

	AT	CPU BT	IO BT	BT	RT
P_1	0	3	21	24	
P_2	1	1	3	4	
P_3	2	2	3	5	



Total time = 33

CPU Utilization = $\frac{3+1+2}{33} = \frac{6}{33} = 18\%$

i.e. low CPU utilization.

One sol^n could be to run other process when one process goes to IO waiting.

* Shortest Job First

(Assumes that when a job is submitted we know its Burst Time)

From all the process currently in the ready queue select the one with least Burst Time.

Ex.

	AT	BT	CT	TAT	WT	RT
P_1	1	8	9	9-1=8	0	1-1=0
P_2	2	6	20	20-2=18	18-6=12	14-2=12
P_3	4	2	11	11-4=7	7-2=5	9-4=5
P_4	10	3	14	14-10=4	4-3=1	11-10=1

	P_1	P_3	P_4	P_2
0 1	9	11	14	20

Total time = 20.

At t=1 only P_1 is there.

By t=9 P_2 & P_3 are competiting.

By t=11 P_2 & P_4 are competing

By t=14 only P_2 is there.

Avg WT = $\frac{0+12+5+1}{4} = 4.5$

- Shortest Job First gives the lowest Waiting Time (avg) because it always moves short processes before long processes. (For Non-preemptive)

- One major issue with SJF is that we don't know the execution time (burst time) of next process. We can try to predict future burst time based on history of BT of the process.

- Predicting length of next CPU burst time:

(1) Simple Average: Given historical record of execution, we can take simple avg. of all BTs.

S_K : Avg. $(t_0 + t_1 + \dots t_{K-1})/K$

t_K: Actual K^{th} Burst Time.

$$S_K = \frac{t_0 + t_1 + \dots t_{K-1}}{K} \Rightarrow K S_K = \sum_{i=0}^{K-1} t_i$$

$$S_{K+1} = \frac{t_0 + t_1 + \dots t_{K-1} + t_K}{(K+1)}$$

$$\Rightarrow S_{K+1} = \frac{K S_K + t_K}{(K+1)}$$

$$\therefore S_{K+1} = \frac{K}{K+1} S_K + \frac{1}{K+1} t_K$$

Using this formula, we don't have to keep track of all $t_0, t_1 \dots$ only the last predicted burst time S_K & actual BT t_K are needed.

(2) Exponential Average: Recent history can be more accurate than earlier history. Given as:

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

where α is a parameter: $\alpha \in [0, 1]$

$\alpha = 0$: $S_{n+1} = S_n$

$\alpha = 1$: $S_{n+1} = t_n$

So by controlling α we modify weightage given to recent history vs older history.

Ex. We earlier had jobs with actual execution time 4, 7, 8, 16. We now have a job, with $S_0 = 10$. $\alpha = 0.5$. Find exponential & simple avg, for this job.

$$\text{Avg}_{\text{simple}} = \frac{4+7+8+16}{4} = 8.75$$

$t_0 = 4$. $S_0 = 10$ $S_1 = \frac{1}{2} \times 4 + 10 \times \frac{1}{2} = 7$.

$t_1 = 7$ $S_1 = 7$ $S_2 = \frac{1}{2}(7+7) = 7$.

$t_2 = 8$ $S_2 = 8$ $S_3 = \frac{1}{2}(8+7) = 7.5$

$t_3 = 16$ $S_3 = 7.5$ $S_4 = \frac{1}{2}(16+7.5) = 11.75$

* Shortest Remaining Time First

- Till now we have seen that once a process gets CPU time, it executes for its complete burst time & CPU doesn't stop it.

- Preemption: Forceful stoping of a process from running and moving it to ready state. OS can perform preemption based on various strategies, such as higher priority (by syscall), or time slice (by hardware interrupt), etc.

- SRTF says that at any point in time run the process with the shortest remaining Burst Time.

Ex.

	AT	BT	CT	TAT	WT	RT
P_1	1	8	19	18	10	0
P_2	2	5	9	7	2	0
P_3	4	2	6	2	0	0
P_4	10	3	13	3	0	0

	P_1	P_2	P_3	P_2	P_1	P_4	P_1

0 1 2 4 6 9 10 13 19

At $t=1$: only P_1 with RBT = 8 So, P_1

At $t=2$: P_1(RBT) = 7, P_2(RBT) = 5 So, P_2

At $t=4$: $RBT_{P_1} = 7$, $RBT_{P_2} = 3$, $RBT_{P_3} = 2$ So, P_3

At $t=6$: ——— " ———, ~~RBT(P4) = 13~~ So, P_2

⋮

context switches = 6. (only considering $P_i \rightarrow P_j$)

- Prone to Starvation of Longer BT processes.
- Overall has the least average waiting time provably.

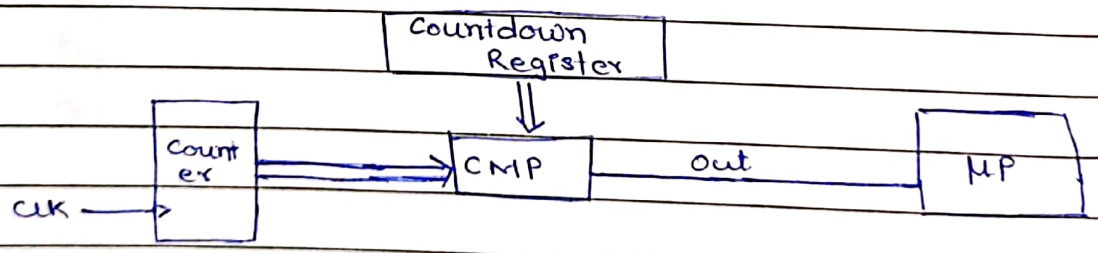
* Round Robin

- Generally we don't want to just run one process at a time to entirety. That's why we need preemption. Also, we have a lot of interactive programs such as browser, video player, etc.

Ex. Suppose we have two processes P_1 & P_2. For some time we run P_1, say S units, then we run P_2 for S units and so on.
For this to happen, we have to know that process ran for S units yet or not.

(1) One way is to poll after every unit of time to check, but it would be inefficient and slow.

(2) We can have a h/w timer interrupt to the CPU, which triggers Interrupt Service which is OS code.



So, during context switch, just before CPU starts new process's instn, OS sets Time Quantum to S.

- So, process runs for S time & Context Switch runs for γ time. Since CS is not useful work (overhead):

$$\% \text{ efficiency} = \frac{S}{(S+\gamma)} \times 100\%$$

Say • $Y \simeq S$, $\%eff = \frac{1}{2} = 50\%$

- RR Scheduling
  (1) Pick the process at front of ready queue & set its context.
  (2) Set the time quanta for h/w interrupt
  (3) Dispatch the process.
  (4) When timer expires and OS code is run again, move the process from running to ready state & at last of ready queue.
  (5) Goto (1).

  * If process terminates, goto (1).

Ex.

	AT	BT	CT	TAT	WT	RT
P_1	1	8	15	14	6	1
P_2	2	5	16	14	9	3
P_3	4	2	11	7	5	5
P_4	10	3	19	9	6	6

$S = 4$

	P_1	P_2	P_3	P_1	P_2	P_4

0 1 5 9 11 15 16 19

Avg Waiting Time = 1.58

~~P_1 P_2 P_3 P_1 P_2 P_4~~

context switches = 5 # preemptions = 2

Note: If there is preemption then there is Context Switch

Ex. Lets say we have some processes with the following

BT: P_1 P_2 P_3 P_4 P_5 . All arrive at $t=0$.

$\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$

200 2 2 2 2

& $S = 100$

So,

P_1	P_2	P_3	P_4	P_5	P_1

0 100 102 104 106 108 208

∴ #context switch = 5

preemptions = 4

WT: 8 100 102 104 106

AWT = 84.

RT: 0 100 102 104 106

ART: 82.4

Ex. Let's take same processes but with $S = 2$.

P_1	P_2	P_3	P_4	P_5	

0 2 4 6 8 10 208

CS = 5 # preemptions = 1

WT: 8 2 4 6 8

AWT: 5.6

RT: 0 2 4 6 8

ART: 4.

- For RR, in general low time quanta causes less response time.
- If Time Quanta is greater than largest BT, RR behaves like FCFS.

- If remaining BT of a process becomes zero as time quanta expires, we can't say which event would be responded to. Can't say if preemption would happen or not. Context switch happens, whichever event reaches CPU first.

- RR is preferable for interactive applications (i.e. applications requiring quick response time.

* Longest Job First

- LFS is a non-preemptive algo. which chooses from the current processes in Ready Queue to run next, whichever process has longest burst time.

- Could use Max-Heap data structure.
- Could also be preemptive (LRTF)

Ex. Use preemptive LJF Scheduling.

	AT	BT	CT	TAT	WT	RT
P_1	0	2	2	2	0	0
P_2	1	1	11	10	9	9
P_3	2	3	10	8	5	0
P_4	3	6	12	9	3	0

P_1	P_3	P_4	P_3	P_2	P_4

0 2 3 8 10 11 12

\# CS = 5 \# preemptions = 2.

* Priority Scheduling

- At each instance choose the process with highest priority.
- In preemptive version, if a new process with higher priority comes in while a process is running, then preemption happens.

- A process gets priority:
  (1) User Specified (Explicit)
  (2) OS provided (Implicit)
  Ex. in SJF, user provides BT for process. Based on that OS may assign priority.

- Could be implemented using Multi-Level queue.

- Starvation: When a low priority process is passed on for long time for higher priority processes.

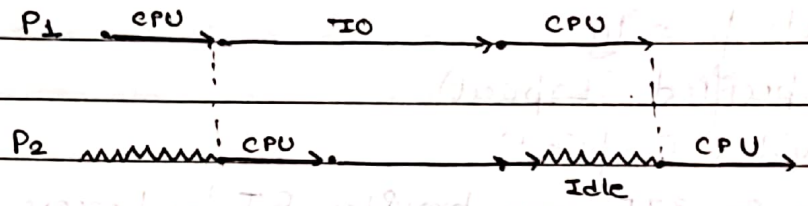
- Aging: The priority of low priority processes increases after sometime so that it doesn't starve for long.

- Generally, lower the priority value, higher the priority.

classmate
Date
Page

* We have seen that if there is only one CPU then at a time only one process can execute on the CPU, but its not the case with I/O. Since there are various I/O devices, and different processes may require I/O service from different devices, the I/O burst can overlap for the processes.

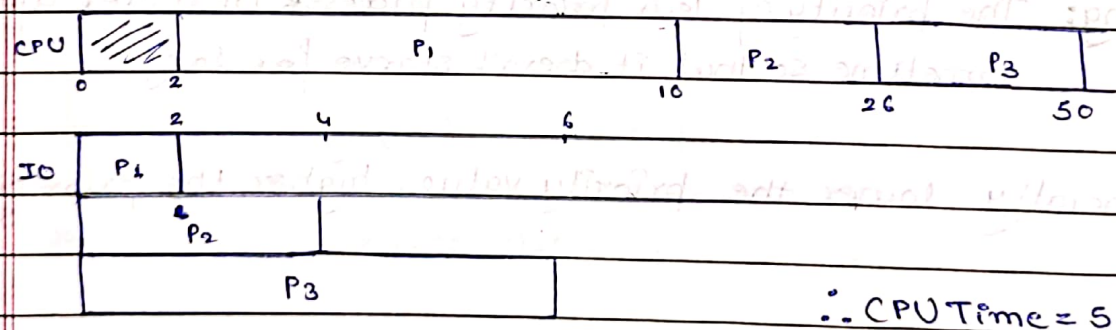
Ex.



Ex.

	AT	IO BT	CPU BT	CT	TAT	WT	RT
P_1	0	2	8	10	10	10-8=2	0
P_2	0	4	16	26	26	26-16=10	10-4=6
P_3	0	6	24	50	50	50-24=26	26-6=20

FIFO Algo.



∴ CPU Time = 50

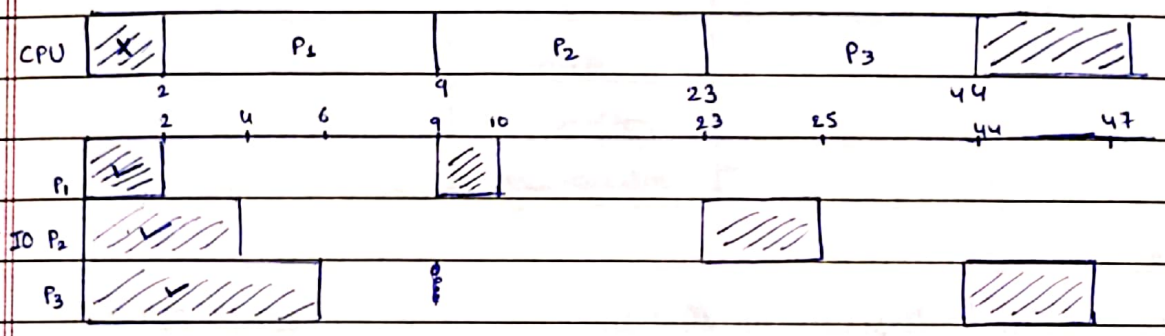
% utilization = $\frac{48}{50}$ = 96%

Ex.	AT	I/O BT1	CPU BT	I/O BT2
P_1	0	2	7	1
P_2	0	4	14	2
P_3	0	6	21	3

SRTF Algo.

All I/O operations can overlap.

Find % idle CPU time



∴ CPU time = 47

Idle time = 2 + 3 = 5 % idle = $\frac{5}{47} \times 100\% = 10.6\%$

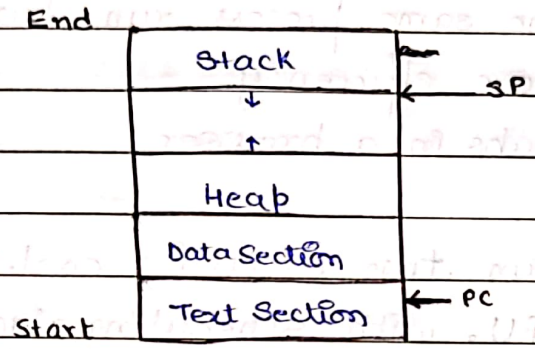
* Multilevel Queue Scheduling

- We can break the ready queue based on some criteria into multiple queues.
  During scheduling OS chooses one of the queues (maybe based on priority) and schedules one of the process from the queue based on that queue's scheduling algorithm.

- One of the well known criteria is:
  1. Foreground Processes
  2. Background Processes
  3. Interactive Process

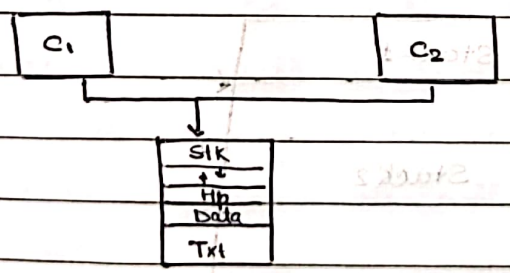
* Threads

- We saw the structure of a process in the memory:



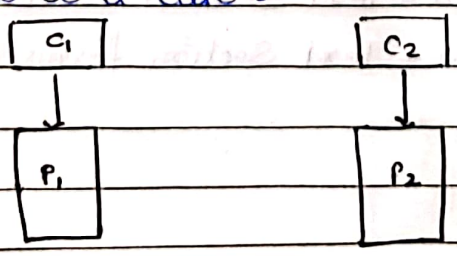
- Address Space: AS of a process is the set of all the addresses from start to end address in m/m that a process can access.

- Let's suppose we have two CPUs. Both have access to the same address space for a process.



Both CPUs also have access to different context for the same process (such as register content).

- There can also be a case:



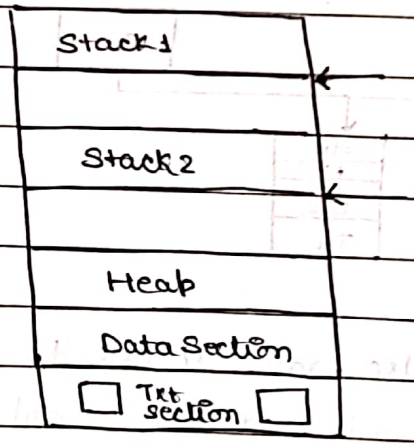
Both CPUs have different context as well as Address Space.

- We want for the same process run two different execution context to improve efficiency.
  Ex: Multiple tabs in a browser.

- We can also run two execution contexts for a process on a single CPU, with scheduling algorithms like Round Robin.

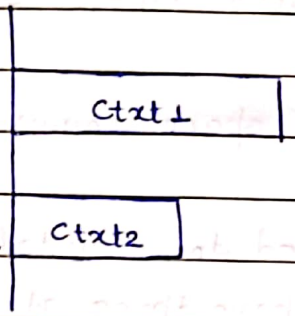
So number of execution contexts doesn't depend on the number of CPUs.

One such model of execution is to have two different stacks in same process's memory. We recognize such a setup as threads.



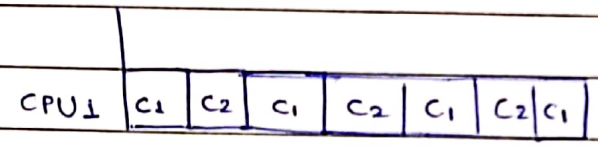
So, Stack 1, Heap, Data & Text Section forms one thread.
Stack 2, Heap, Data & Text Section forms another thread.

- **Parallelism:** When we have more than one CPU, two contexts can run in parallel, i.e. overlap in execution



This is concurrent & parallel execution.

- **Concurreny:** When we execute more than one context, on a single CPU, the two context are active at the same time, but their execution doesn't overlap.



This is concurent.

- Threads are all about concurrent and parallel execution.
- **Example Usecase:** For a HTTP webserver everybody wants to send a request, have server process the req. and respond. So, the code run for each req. is same but the context varies from request to request.

- Processes are heavy, i.e., each process has its own address space and doesn't share OS resources.

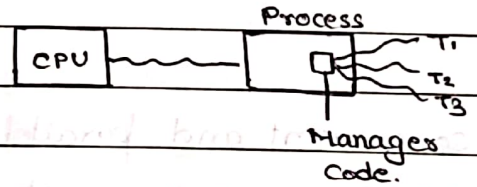
Single Thread Multi-thread

- Two processes can't access each other's address space. Communication b/w two processes is expensive, requires OS assist through mechanisms like Pipes, FIFO queues, sockets etc.

- Since threads are in same addr. space communication is cheap.

- To implement threads, we need to maintain multiple execution context within a process. We have three strategies for this:
  (1) Let User Program manage it.
  (2) As OS, handle this
  (3) A mix of both.

- User level Threads:



From OS stand point, it only sees the process and it schedules the process on CPU. Some code within the Process takes care of thread scheduling.
Hence managing the threads doesn't require any OS intervention. So, no system calls requires.

- Kernel Level Threads:

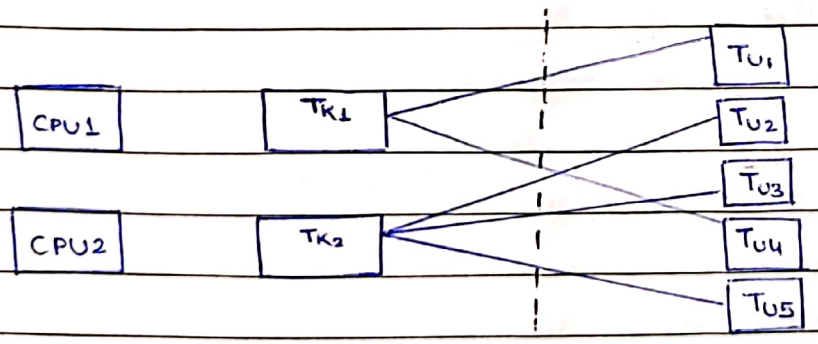
Kernel manages threads. User process needs syscall for threading operation. OS in response will create threads

How Kernel manages threads is a design problem.
One possible solution is to hold one PCB per thread with the mem pointer pointing to same address space, and have separate context.

The scheduling of threads is handled by the OS itself.
One of the reason to use Kernel level threads is to use multiple CPUs, since CPUs are resources owned by Kernel.

- When one of the threads in User Level threading requests IO or make sleep syscall, etc, then the whole process is made to go in wait state since OS sees just the process as entity.
  (Blocking call)

- The above points make an appealing case for multilevel threading, i.e threading both at user level & kernel level.



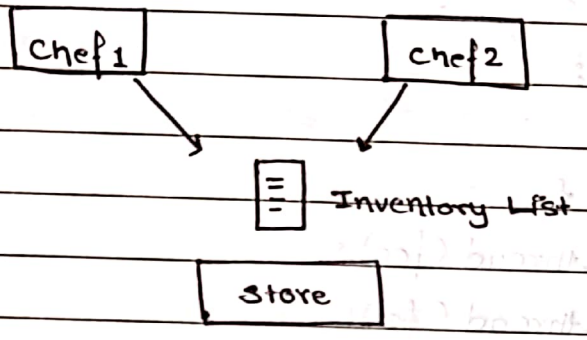
This allows for faster context switching b/w user level threads mapped to same K-Thread. Also allows for multi CPU execution by K-Threads. When one K-level thread gets blocked, the other can still run even in single CPU environment.

at user level, not OS context switch.

Synchronization

- When we have multiple active context ~~as~~ then we require control over how the shared resources, such as data section or heap are accessed, and hence we require Synchronization b/w different contexts.

Ex.



When a chef requires ingredients from store they first check IList, then collect from store, then update the list. (3 steps).

Say C1 checks IList for Butter, and 1 unit is available. It goes in store. C2 comes and checks IList while C1 is in store. Checks for butter and finds 1U available. He goes inside the store. Meanwhile C1 comes out and updates the IList. C2 is now looking for butter that doesn't exist.

If there was some Synchronization mechanism b/w the two this situation wont arise.

What if we make a rule that only one chef can check IL, collect, update IL. This solves the problem somewhat.

- **Synchronization** thus is implementing of protocols while concurrent execution contexts ~~try to~~ share resources so as to ~~ensure~~ ensure valid / correct execution of each context.

Ex.

```
int count = 0;
void foo() {
    count++;
}
int main() {
    T1 = create_thread (foo);
    T2 = create_thread (foo);
    wait (T1, T2);
    pf (count);
}
```

Both T_1 & T_2 run at the same time (concurrently). Lets suppose thread scheduling is done using Round Robin with quanta S & in S ~~ms~~ units. CPU can run one CPU instn.

Let's say count++ compiles to:

I_1 : LOAD R_1, 2000 {2000 is addr of count}

I_2 : ~~STORE~~ INCR R_1

I_3 : STORE 2000, R_1

CPU	T_1	T_2	T_1	T_2	T_1	T_2

This gives following execution sequence:

$T_1 I_1 \rightarrow T_2 I_1 \rightarrow T_1 I_2 \rightarrow T_2 I_2 \rightarrow T_1 I_3 \rightarrow I_2 T_3$.

which results in count = 1.

Similarly we can have different execution sequence based on scheduling algo, its parameter and various other parameters.

This makes the value of count uncertain.

① $T_1 I_1 \rightarrow T_1 I_2 \rightarrow T_1 I_3 \rightarrow T_2 I_1 \rightarrow T_2 I_2 \rightarrow T_2 I_3$: count = 2.

② $T_1 I_1 \rightarrow T_2 I_1 \rightarrow T_2 I_2 \rightarrow T_2 I_3 \rightarrow T_1 I_2 \rightarrow T_1 I_3$: count = 1.

⋮

There are $\frac{6!}{3!\,3!} = 20$ sequences possible.

We don't want this uncertainty. We want an "expected" result. We want certainity in execution irrespective of factors such as scheduling algorithms, parameters, blocking calls, etc.

* Race Condition: When concurrent execution contexts such as threads run, and access ~~the~~ some shared value, then the value of the shared variable depends on the execution sequence.

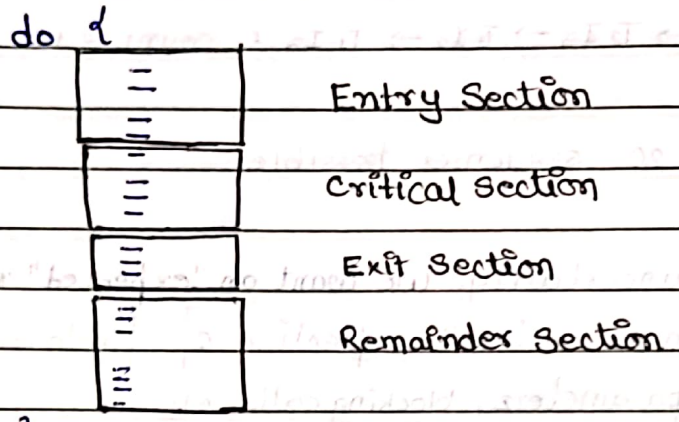
- If the data is readonly for all contexts then the order of access doesn't really matter.

- If even one of them writes to the variable then execution sequence matters.

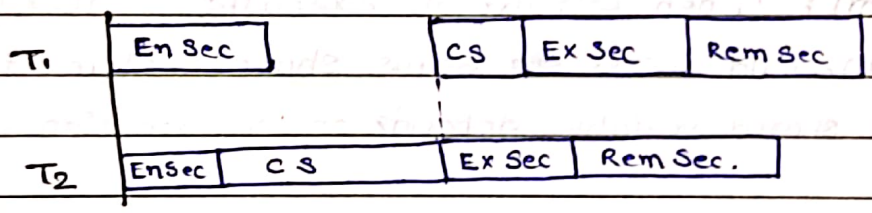
- Critical Section: A process / thread has a lot of instns. A bunch of instns in a thread which manipulates shared data such that the value of shared data depends on the execution order is called "Critical Section".

- Critical Section Problem: When a ~~process~~ context is executing in its critical section no other context should be allowed in their CS, manipulating same data. This is a way to enforce some kind of order.

- Based on this we have the following structure of a context:



Thread i



So, execution of Critical section shouldn't overlap.

* A Basic CS Problem Solution

- Say we have two threads:

```
int count = 0;
int flag = 0;
void foo() {
    while (flag == 1);      ] En Sec
    flag = 1;               ]
    count++;                ] CS
    flag = 0;               ] Ex Sec.
}
```

```
int main() {
    T1 = create-thread (foo);
    T2 = create-thread (foo);
    wait (T1, T2);
}
```

- Analysis 1

(1) T_1 comes and checks flag. It's 0. It comes out of loop.
(2) T_1 updates flag = 1.
(3) T_2 comes and checks flag. It's 1. It gets stuck in loop.
(4) T_1 executes CS and updates count to 1.
(5) T_1 updates flag = 0 in ExSec.
(6) T_2 comes out of loop.
(7) T_2 updates flag = 1.
(8) T_2 updates count = 2 in CS.
(9) T_2 updates flag = 0 in Ex Sec.

Hurray!! we get the desired output.
But this is only one possible execution seq. Our solution should work for all possible sequences.

- A solution to CS problem should ensure that atmost one thread should execute its Critical Section code, for all possible execution sequences.
So we only need to show only one counterexample execution sequence to show that solution doesn't work.

- Analysis 2

(1) T_1 checks flag. It's 0. It comes out of loop.
(2) T_2 checks flag. It's 0. It comes out of loop.
(3) T_1 updates flag = 1.
(4) T_1 enters CS. Updates count = 1.
(5) T_2 updates flag = 1
(6) T_1 updates flag = 0.
(7) T_2 updates count = 1 in CS.
(8) T_2 updates flag = 0.

We get wrong output.

- So this sol^n doesn't really work. One improvement to the sol^n can be that the Entry Section were to be executed in single Fetch-Decode - Execute cycle. Such a sol^n would require H/w support, since only CPU can allow for inst^n in Inst^n Set while do operations in single FDE cycle.

This requires an Atomic Inst^n, i.e. inst^n executes as whole or not at all.

* Test & Set Solution

- Suppose CPU does provide such an atomic inst^n:

TSTST Addr of flag

which functions as:

```
int test_set (int *flg_addr) {
    int vl = *flg_addr;
    *flg_addr = 1;
    return vl;
}
```

- Now we can modify previous sol^n as:

```
int flag = 0, count = 0;
void foo () {
    while (test_set (&flag)); ] EnSec
    count ++;                 ] CS
    flag = 0;                 ] ExtSec
}

int main () {
    T1 = create (foo);
    T2 = create (foo);
    wait (T1, T2);
}
```

- Analysis

(1) T_1 atomically gets value 0 from T_S & exits loop while setting flag 1.

(2) T_2 gets value 1 atomically from TS & gets stuck in loop.

(3) T_1 executes CS code & count = 1.

(4) T_1 updates flag = 0.

(5) T_2 gets val. 0 atomically from T_S & exits loop setting flag = 1.

(6) T_2 executes CS code & count = 2.

(7) T_2 updates flag = 0.

∴ We get desired result.

- This is spinlock pattern:

```
int lk = 0;
void lock (int* addr) {
        while (Test_Set(addr));
}
void unlock (int * addr) {
        *addr = 0;
}
```

```
int main() {
    lock(&flag);      Ent Sec
    CS                C.S.
    unlock(&flag);    Ext Sec.
}
```

This is known as Spinlock since the thread keeps ~~looping~~ spinning in while loop, waiting for lk to be 0.

Spinlock guarentees Mutual Exclusion.

Is Mutual Exclusion enough for C.S. Problem Sol^n?

No. Because of some other properties of interest.

- Now, let's suppose that two threads T_1 & T_2 execute the following functions.

```
                int flag = 0;
void foo() {                          void bar() {
    while (true) {                        while (true) {
        lock (&flag);                         lock (&flag);
        [ CS1 ];                              [ CS2 ];
        unlock(&flag);                        unlock (&flag);
    }                                     }
}                                     }
        T1                                    T2
```

Analysis

There can be a sequence where T_1 acquires the lock & hence T_2 is stuck in loop, T_1 executes CS, unlocks & before T_2 acquires the lock, T_1 again acquires the lock, and this may happen over and over and T_2 never gets to acquire the lock.

For any value of K, we can have an execution sequence where CS1 executes K times and CS2 executes 0 times. So, K isn't bounded.
But CS2 may be some very crucial code, we may not want it to wait unboundedly.

So, Bounded Waiting is a desired property of CS soln, i.e. a process is guarenteed to execute it's CS within some bounded units of time, it may be 1sec, 1month, 10years, but should be bounded.

* A sol^n to the Critical Section problem should satisfy the following three requirements:

- Mutual Exclusion: At a time at most one ~~process~~ execution context is allowed to execute it's critical section code which manipulates shared data.

- Progress: If no thread is executing in its Critical Section and some thread wishes to execute its CS code, then only those threads which aren't executing in their remainder section can participate in deciding which will enter it's critical section next, and this selection can't be postponed indefinitely.

- Bounded Waiting: Each process is guarenteed to execute in its Critical Section within a bounded amount of time, i.e. there is a upper bound to the number of times other processes are allowed to enter their CS, when there is some other process trying to enter CS.

* Two - Context Solution

Suppose there are only two process/thread executing, with shared data.

- Algo1: Strict Alteration

Put turn = 0 or 1;

T_0

```
loop { while (turn ≠ 0);
       [ CS0 ]
       turn = 1
     }
```

T_1

```
loop { while (turn ≠ 1);
       [ CS1 ]
       turn = 0;
     }
```

Analysis:

Supposing that value of turn is atomically read, the sol^n satisfies Mutual Exclusion.

Since at a time the value of turn can only be either 0 or 1, one of the two processes is eventually bound to enter it's CS. Before going to remainder section, in ExSec, the process switches the value of turn to other process's turn value, and in RSec, it has no say. If the other process wants to enter it is allowed to.
Suppose that turn is initialized to 0, but T_0 never wants to enter CS, but T_1 does, T_1 is stuck indefinitely in EnSec. So, progress isn't guarenteed.

We know that when both processes do execute (i.e. want to enter CS one or more times) they alternate value of turn. So once P_i alternates value of turn, P_{1-i} can only enter CS next, so the no. of turns before a process can enter its CS is 1.

- Algo 2:

```
int want[2]={0,0};

T0                              T1
while(1){                       while (1) {
  want[0]=1;                      want[1]=1
  while(want[1]);                 while (want[0]=1);
  CS1                             CS2
  want[0]=0;                      want[1]=0;
}                               }
```

Analysis

- Since the value of want[0] & want[1] are read atomically atmost ~~only~~ one of which can be 0 according to the algo, both threads can't enter CS together. ME satisfied.

- Suppose T_0 sets want[0]=1, then T_1 sets want[1]=1, both get stuck in while loop. So progress isn't satisfied since there are two processes, both of which want to enter CS, ~~and~~ but the decision to allow one of them can not be made finitely.

Say one of the two processes never wishes to enter CS, say T_i, so it never sets wants[i] to 1. The other process can easily enter CS unlimited no. of times.

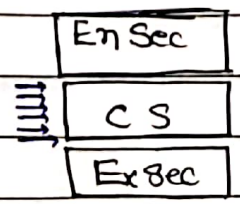
- Suppose the processes don't enter deadlock, and both are interested to enter CS, then again we have alternating execution so atmost 1 time a process can enter befor T_i is allowed to execute CS section.

Note: [Guidelines for verifying CS Sol^n requirements]

(1) While checking for progress, check all the corner cases: P_i for all i is the only process which wants to enter CS; Deadlock possibilities.

(2) For progress ensure that some T_i in it's remainder section doesn't block others.

(3) Ensuring ME means that ~~only~~ atmost one process can be in this region at a time:



(4) While checking for Bounded Waiting, assume that progress is guarenteed and at least one other process is contending to enter CS besides P_i, otherwise why would P_i be waiting.
Bounded Waiting cond^n means that if everything goes right what is the most no. of turns a process P_i has to wait before it can run in its CS.
Also assume each P_i is in loop if not already.

Ex. Which of the three properties is/are satisfied by the following sol^n:

T_1	T_2
while (x==1);	while (x==0);
x = 1;	x = 0;
CS	CS

Doesn't satisfy ME. Say x = 0. T_1 gets out of loop. Sets x = 1. T_2 gets out of loop & updates x = 0. Both are in CS now.

Say, x = 1, and T_2 doesn't want to execute CS. T_1 can never enter in CS. So, progress not satisfied.

Bounded waiting by def^n is 1. Atmost 1 process enters CS before P_i gets to enter CS in all situations.

Ex. T_1 bool S_1, S_2, T_2

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

```
while (S1 != S2);
    CS
S2 = !S1;
```

Say, initially $S_1 = S_2$, and T_2 doesn't want to execute CS, only T_1. T_1 wouldn't be able to. So, decision pending indefinitely. So, no progress.

Atmost one time, some T_j $(j \neq i)$ can enter CS before B. T_i is allowed. $\therefore$ BW = 1.

Under all situations, ME is ensured, since $S_1 == S_2$ & $S_1 != S_2$ are checked atomically.

- Algo 3: Peterson's Sol^n

```
int want[2] = {0,0};
int turn = 0;
```

T_0	T_1
want[0] = 1;	want[1] = 1;
turn = 1;	turn = 0;
while (want[1] && turn == 1);	while (want[0] && turn == 0);
CS	CS
want[0] = 0;	want[1] = 0;

Analysis:

- In all situations, only one process can enter CS.
- Since want[2] is initialized to {0,0} even if say P_j $(j \neq P)$

doesn't want to enter CS, P_i can enter since want[j] would be 0.

If both want to enter, turn as an atomic value would break the while loop for at least one of the process.

- Similarly after executing once P_i sets turn to $(1-i)$ before while loop. Since P_{1-i} wants to enter, it can enter.

  $\therefore$ BW = 1.

* There is one other H/W soln, to disable interrupts, so no preemption. But this can be problematic, say during Segmentation Fault.

Also, this doesn't work in multi-CPU environment since each has it's own Interrupt lines.

- H/W instn to ensure Mutual Exclusion is a workable soln. It ensures ME & Progress, not Bounded Waiting. Deadlock & starvation are possible. Busy waiting happens in while loop, wasting CPU m/m.

Ex. Assume lock = 0. P_1 acquires it and enters CS. P_2 a higher priority process comes & P_1 is preempted. P_2 tries to acquire lock but can't.

Hence Deadlock.

- The following algorithm makes use of Test_Set (int *lock) :

```
bool waiting[n];
bool lock;

Ti:
    do {
        waiting[i] = true;
        Key = true;
        while (waiting[i] && Key)
            Key = test_and_set(&lock);

        [ Critical section ]

        j = (i+1)%n;
        while ((j != i) && waiting[i] != true)
            j = (j+1)%n;
        if (j == i)
            lock = false;
        else
            waiting[j] = false

        [ Remainder section ]
    }
```

Analysis:

- In all situations in EnSec all processes are blocked until either lock is false or waiting[i] is false, which can happen for only one process at a time.
- Progress is ensured as in ExSec Pi unlocks Pj by either lock or waiting[j] false. If only single process, then lock is false anyway.
- BW = (n-1)

* Mutex Locks

- For a process trying to acquire some spinlock which is already expired for the time it executes while waiting it executes:

```
while (test_and_set (&lock));
```

This is just wastage of CPU since it is occupying CPU while waiting.
If there was a way to know that Pi is just waiting, then we could just take it out of competition to acquire CPU & move it to Wait Queue. Also there would require a signal which tells Kernel that lock is available & Pi can contend for it again & hence put it back in ready queue.

- Such support can obviously be provided by OS for Kernel level threads. User-level libraries & environments can also provide such support for user level threads.

- Sol^n 1:

```
int l = 0;
acquire (int *lock) {
    while (test_and_set (&lock)) {
        move to waiting queue
    }
}
release (int *lock) {
    *lock = 0;
    move 1 or more thread from waiting queue to ready queue;
}
```

When a process Pi with lock calls release, it can move one or more process from Waiting to Ready queue. Again only one process comes out of while loop and rest again go to waiting Queue.

Blocking: A process is in blocking state simply means it is in Waiting State and doesn't hold up CPU unnecessarily. It might block for I/O, some lock, or other resources.

This Sol^n, still ensures ~~Mutu~~ Mutual Exclusion & Progress with no busy waiting, but Bounded Waiting isn't satisfied.

* Semaphore

- Semaphore is a mechanism which performs atomic increament & decrement to a variable.

```
                    int S=0;
signal (S){                          wait (s) {
    S++;  }                              while (S≤ 0);
}                  Atomic  →  {  S--;
                              }
```

A thread can get stuck in wait if $S \leq 0$.
We can implement wait and signal using locks:

```
                    int S=0, l=0;
signal (S,l){                        wait (S,l) {
  acquire(&l);                         acquire (&l)
    S++                                while (S≤0) {
  release (&lock);                       release (&l);
}                                        acquire (&l);
                                       }
                                       S--;
                                       release (&l);
                                     }
```

- Wait ≡ Down ≡ P.
  Signal ≡ Up ≡ V.

- Generally Semaphore means blocking semaphore impl. not busy waiting implementation.

- We can implement Mutex Lock using Semaphore by limiting its value to 0 & 1. These types of semaphores are called Binary Semaphore.

- The unrestricted version is called "Counting Semaphore".

```
signal (&lock) {
    *lock = 1;
    if (some blocked process)
        wakeup();
}
```

```
wait (&lock) {
    if (*lock == 0)
        sleep();
    *lock = 0;
}
```

* Deadlock

- [For two Threads] When thread T_1 is waiting for T_2 & T_2 is waiting for T_1.

Ex.

T_1	T_2
wait (S_1);	wait (S_2);
wait (S_2);	wait (S_1);
$\boxed{CS}$	$\boxed{CS}$
release (S_2);	release (S_1);
release (S_1);	release (S_2);

Where S_1 & S_2 are binary semaphores.
Suppose the following seq:

T_1: wait (S_1)

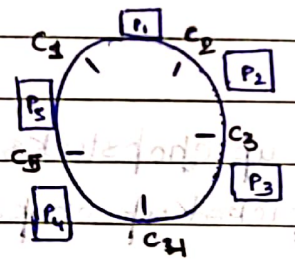
T_2: wait (S_2)

T_1: wait (S_2) x
T_2: wait (S_1) x } Blocked.

* Classic Synchronization Problem

1. Dining Philosopher.

There are five philosophers around a table, each eat for sometime, and think for sometime. There are five chopstick & two are needed to eat. Before eating 2 chopsticks must be acquired such that no chopstick is held by more than one phil. & no phil. starves either.



- Protocol 1
  - If a phil. P_i has to eat
    - First pick up chopstick C_i (on right)
    - Then pick up chopstick $C_{(i+1)\%5}$ (on left)

This doesn't work. Say all phil. P_i pick C_i. Now all will wait to acquire $C_{(i+1)\%5}$. So all starve.

- Protocol 2
  - Odd indexed phil. pick left chopstick first then right.
  - Even indexed phil. pick right chopstick first then left.

chopstick[i];

Pi (i odd)	Pi (i even)
wait(C[i_left]);	wait(C[i_R]);
wait(C[i_R]);	wait(C[i_L]);
eat	eat
signal(C[i_R]);	signal(C[i_L]);
signal(C[i_L]);	signal(C[i_R]);
think	think

This ensures no deadlock & no starvation.

- Protocol 3

Allow a Pi to pick up chopsticks only if both are available else relenquish the picked up chopstick so that some other Pi can pick it up.

Ex. Consider Dining Philosopher Problem with five chopsticks and four phil. Consider following statements:

(A) Deadlock isn't possible

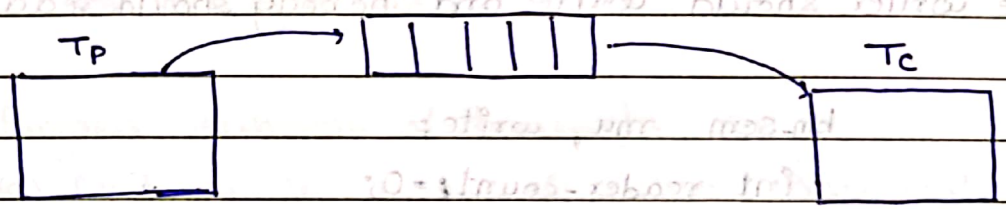
(B) Starvation isn't possible.

A is true. [Pegionhole Principle]

B is false. Starvation depends on the algorithm. May or maynot happen.

2. Producer Consumer

There is a producer thread and a consumer thread running concurrently. They have a shared buffer. A producer is only allowed to put on buffer if its empty. A consumer is only allowed to pull from buffer only if it has some value.




```
int buff [N]
int count = -1, in = -1,
int empty = N,        { Counting
        full = 0;       Semaphore }
int mu = 1;           { Binary Semaphore }
```

Producer

```
do { wait(empty);
     wait (mu);
     in=(in+1)%N;
     buff [in]= item();
     signal (mu);
     signal (full);
  } while (true);
```

Consumer

```
do { wait (full);
     wait (mu);
     out = (out+1)%N;
     item = buff [out];
     signal (mu);
     signal (empty);
  } while (true);
```

3. Reader - Writer

In many cases, only a few threads write data, while the rest of them read data. When multiple reader threads read data they don't affect the value so concurrently read can be performed. When a writer writes only a single writer should write and nobody should read.

```
bin_sem mu, write;
int reader_count = 0;
```

Reader

```
do {  wait(mu);
      reader_count += 1;
      if (r_c == 1)
            wait(write);
      signal(mu);
      /* perform read */
      wait(mu);
      reader_count --;
      if (r_c == 0)
            signal(write);
      signal(mu);
} while (true);
```

at least 1 reader is reading, so don't write

No reader is reading, so writing is allowed.

Writer

```
do { wait(write);
     /* perform write */
     signal(write);
} while (true);
```

* While checking for the properties of CS sol^n, check:

(1) Mutual Exclusion:

(a) One process is in CS, another process tries to enter: Show that second process blocks at entry.

(b) Two (or more) processes are in entry code: Show that atmost one will enter CS.

(2) Progress (= absence of deadlock):

(a) No process in CS, some P_i arrives: Show that P_i can enter.

(b) Two (or more) processes in EnSec: Show that at least one will enter CS.

(3) Bounded Waiting (= fairness):

One process in CS, another is waiting to enter: Show that if first process exits the CS & attempts to reenter, the waiting process gets in.

Bounded waiting is satisfied when you can give a constant bound K, which would mean that atmost K times other process can bypass a process P, b/w the time P requests to enter CS & request is granted.

Notice that the request may never be granted as well but that doesn't matter as long as K isn't exceeded.

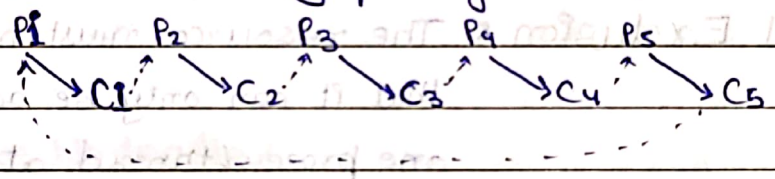
Ex. In Deadlocked situation, no progress, but BW is satisfied w/ K=0.

Deadlocks

- We saw in Dining Philosopher problem, that at a time a chopstick C_i can be held by atmost one philosopher exclusively. So, "Mutual Exclusion" is property of the chopstick.

In our first Protocol for the problem we said that all P_i could first pick left chopstick then right. If all P_i picked C_i to their left, then it's not possible for them to pick right C_i since no C_i remain. The P_is don't relenquish the left chopstick even though they can't get right chopstick, in our protocol. This is called "Hold & Wait".

Also the P_is wait circularly for right chopstick.



So, P_1 holds C_5, and req. C_1, but C_1 is held by P_2 which needs C_2 which is held by P_4 ... and P_5 req. C_5 which is held by P_1. This is called "Circular Wait".

Also, its not the case that P_1 can just snatch away C_1 from P_2 and use it as right chopstick. P_2 can't snatch away C_2 and so only. Only P_2 can give up C_1 willingly & it can't be preempted (taken forcefully). This is known as "No Preemption".

We see that in case of Protocol 1, our Dining Phil. was stuck in a standoff from where no process could proceed further.

Phil. here were process/thread, chopsticks are ~~thread~~ resources. And the situation they were in is called "Deadlock".

Ex. Some examples of resources requiring Mutually Exclusive access:

(1) CPU

(2) Critical Sections

(3) Maybe printers

(4) Keyboard (sometimes)

(5) Certain OS Data Structures.

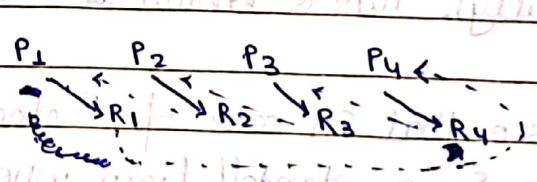
* There are four conditions necessary for deadlock :

1. Mutual Exclusion : The resource must have the property that it can only be acquired by atmost one process/thread at a time.

2. Hold and Wait : The state in which a process is holding some resources and waiting to acquire some others.

3. No Preemption : The resources acquired by a process can't be forcefully relenquished from them.

4. Circular Wait :



- Deadlock in system $\longrightarrow$ (ME $\wedge$ HW $\wedge$ NP $\wedge$ CW)

$\Rightarrow$ ($\neg$ME or $\neg$HW or $\neg$NP or $\neg$CW) $\longrightarrow$ $\neg$Deadlock.

* Deadlock ~~Prevention~~ Prevention

If at any point in time in system, if any one of the four necessary conditions : ME, HW, NP & CW; is made to be false then deadlock can be avoided.

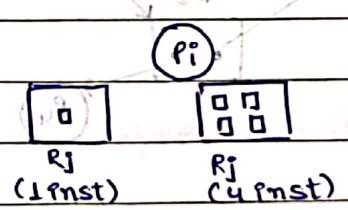
Ex. Deadlock can't occur in a system where every process requires at most one resource to execute. T/F?

Let's say Deadlock then ME, HW, NP & CW.
But H&W can't be true, since if a process holds it has nothing to wait for. If it waits it isn't holding anything.
∴ No H & W
∴ Contradiction
Hence, No deadlock.
True.

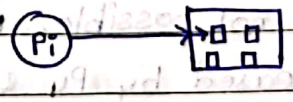
* Resource Allocation Graph.

Process

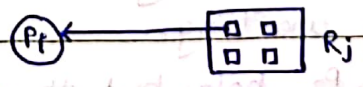
Resource



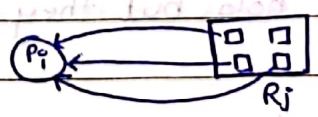
Pi requests instance of Rj



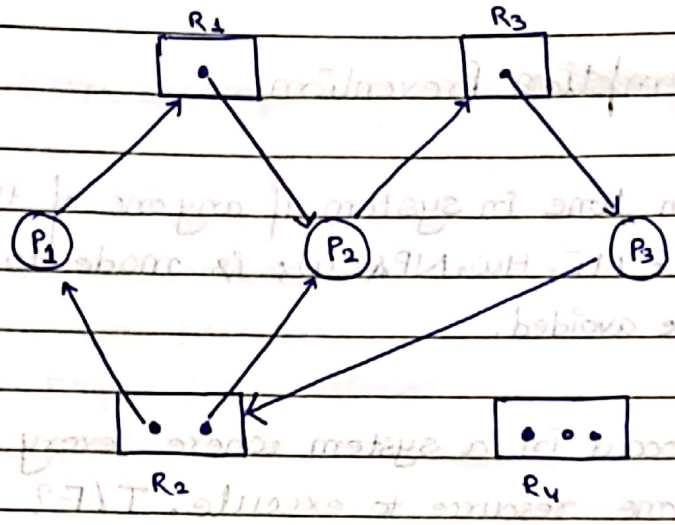
Pi holds an instance of Rj



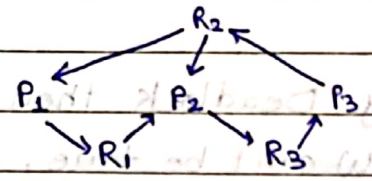
Pi holds 3 inst. of Rj



Ex.



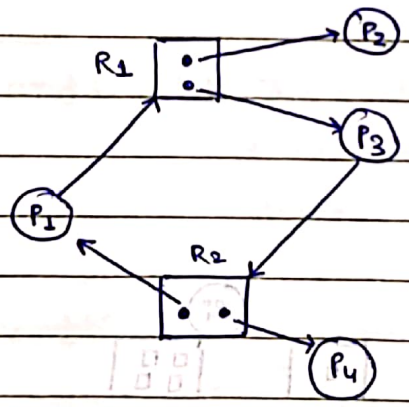
Here there is deadlock possible since all four cond^n are there.



CW

So, there is a possibility that there may be deadlock.

Ex.



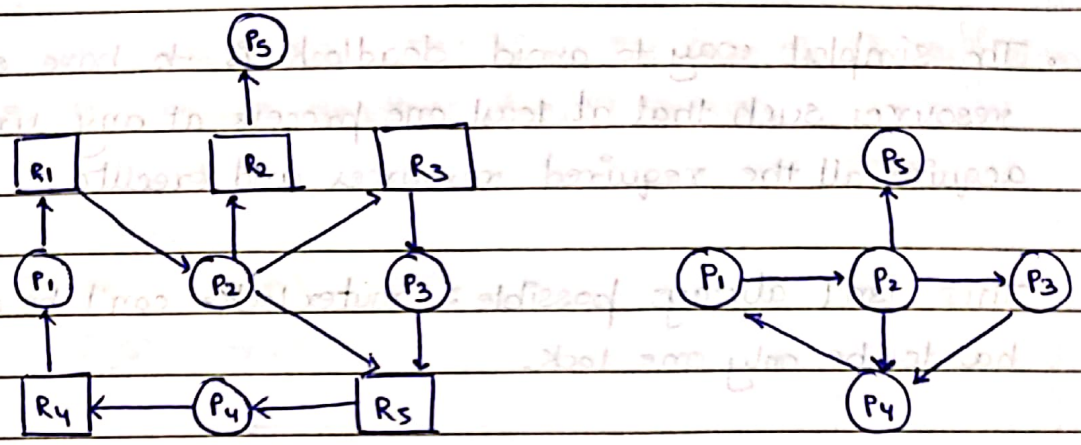
Deadlock is not possible. No circular wait here, since R_2 can be released by P_4 & R_1 can be released by P_2, they aren't circularly waiting.

Also P_4 & P_2 hold but they don't have to wait, they can execute.

So, at this instance we can guarentee no deadlock.

- Wait For Graph: Tells which process is waiting for which process.

Ex.



Wait For Graph.

$P_2 \rightarrow P_3$ since $P_2 \rightarrow R_3 \rightarrow P_3$.

$P_2 \rightarrow P_4$ since $P_2 \rightarrow R_5 \rightarrow P_4$, not $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_5 \rightarrow P_4$.

- If in a Resource Alloc. graph, each resource has single instance then Circular Wait → Deadlock.

* - The way resources are allocated to processes at current instance can help avoid possible deadlocks in future. This is "Deadlock Avoidance".

- Deadlock Recovery: If deadlock happens OS detects it and recovers from it.

- Deadlock Ignorance: If deadlock happens, ~~avoid~~ ignore it.

- Modern OSes use deadlock ignorance, since deadlock is rare in systems.

* Deadlock Avoidance

- The simplest way to avoid deadlock is to have enough resources such that at least one process at any time can acquire all the required resources and execute.

This isn't always possible: mutex locks can't be added. There has to be only one lock.

This isn't always feasible: can't add arbitrarily large no. of h/w resources.

Say there are n processes $P_1, P_2, \ldots P_n$. The ith process requires m_i ~~resources~~. So if the system has a min. of: instances of some resource R

$$\left(\sum_{i=1}^{n} (m_i - 1)\right) + 1$$

~~resources~~ instances of R, then at least one process can execute successfully.

So, $r = |R|$,

$$r \geq \left[\sum_{i=1}^{n} (m_i - 1)\right] + 1$$

$$r_{min} = \left[\sum_{i=1}^{n} (m_i - 1)\right] + 1$$

Also, suppose we have K inst. of R in system with each process requiring x $(x < K)$ no. of resources we can find the max. no. of process that can run in system, deadlock free guarenteed:

$$K \geq P_{max} \times (x-1) + 1$$

Ex. Suppose n processes, $P_1 \ldots P_n$ exists in system, with i^{th} process req. S_i resources at max. There are m resources in system. Which of the following is sufficient cond. to ensure deadlock doesn't occur:

(A) $\forall i,\ S_i < m$.

(B) $\forall i,\ S_i < n$.

(C) $\sum_{i=1}^{n} S_i < (m+n)$

(D) $\sum_{i=1}^{n} S_i < m \times n$.

w.k.t

$$\sum_{i=1}^{n} (S_i - 1) + 1 \leq m$$

$$\Rightarrow \sum_{i=1}^{n} S_i - n + 1 \leq m$$

$$\Rightarrow \sum_{i=1}^{n} S_i \leq m + n - 1$$

$$\therefore \sum_{i=1}^{n} S_i \leq m+n.$$

Ex. A system has 6 inst. of R & N process competing for it. Each process requires at most 2 Rs. Which of the following value of N could lead to deadlock?

(A) 1

(B) 2

(C) 3

(D) 4.

With N = 1, 2, 3 we violate Hold & Wait, so we can guarentee No deadlock.

For N=4, we can have hold & wait $P_1 \Rightarrow 2$, $P_2 \Rightarrow 2$, $P_3 \Rightarrow 1$, $P_4 \Rightarrow 1$. Deadlock depends on other three factors. So, this "could" lead to deadlock.

* Deadlock Prevention strategies

- Violate Mutual Exclusion for resource : If we allow more than one process to hold a resource, then ME is violated. This isn't always possible, hardware resources, locks, etc.

- Violate Hold & Wait : One possible way is to make a process, holding some resources & waiting for some to check if the resources it is waiting for can be acquired. If not then release all the held resources.

- Violate No Preemption : Allow kernel to preempt resources. This isn't viable. Say locks, shouldn't be preempted as it defeats the purpose.

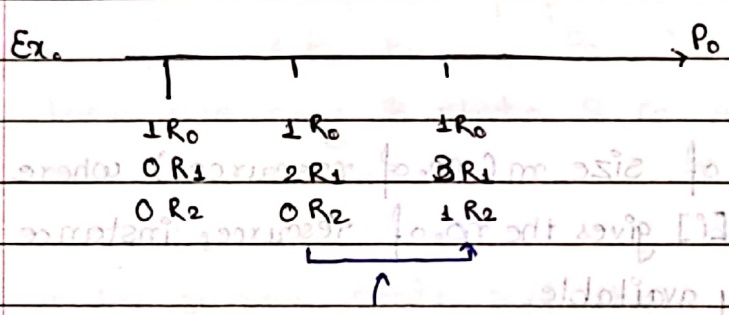
- Violate Circular Wait : One strategy is to restrict a process P_i, holding R_j, to access only R_k if $k > j$. This breaks circular wait.

- Violating Circular Wait is the most feasible & viable option to prevent deadlock.

* Banker's Algorithm
(Deadlock Avoidance Strategy)

- Requires that system has some a priori information available: Each process should declare the max. no. of resources it requires of each type to execute.

Ex.



Depicts the requirement of various resources by P_0 over time.

If req. for $1R_1$ & $1R_2$ can be satisfied, allocation is done. or P_0 has to wait.

This decision is taken by Banker's algo. ensuring deadlock can be avoided.

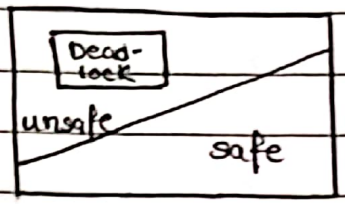
- At any point in time the system can be in one of two states (in context of deadlock)

(1) Safe State: The system is in safe state if there exists a sequence of processes for current allocation state, $\langle P_1, P_2 \ldots P_n \rangle$ such that any resource request P_i can still make can be satisfied by currently available resources along with resources held by P_j $(j<i)$. This sequence is called Safe sequence.

If state is safe then no deadlock.

(2) Unsafe State: The system is unsafe if no safe sequence exists for current allocation state.

If deadlock then unsafe state.



- State of the system at any point of time can be defined as follows:

~~(1) Availability~~

(1) Available : A vector of size m (no. of resources) where Available[i] gives the no. of resource; instance currently available.

(2) There are n processes, each of which requires some instn of each of m types of resources.

Max[i][j] or M[i][j] gives the max. no. of ~~[illegible]~~ resource R_j instances required by process i over its entire execution.

(3) Allocated : Allocated [i][j] the no. of R_j instances currently assigned to P_i.

(4) Need: Need [i] [j] the no. of R_j instances currently still required by P_i.

State ≡ (Available, Max, Allocation, Need).

So in the current state (S):

① We can allocate (some / all) resources available to some process such that it can complete work.

② After the process completes all the resources it held are relenquished

③ Goto ①.

& We get a sequence of process completions

$\langle P_i, P_k, P_m, P_s \ldots \rangle$ with all running process at t_8.

then we say state S is safe & the seq. is known as safe sequence.

So for current state S if there exists at least one safe sequence, then S is safe state & S can't be in deadlock, else S is unsafe & may lead to deadlock.

Ex.

	Alloc A	Max Req. A
P_0	5	10
P_1	2	4
P_2	2	9

Total A = 12

So, Available A in current state = 3.

Allocate 2 to P_1. P_1 executes & finishes. $|A| = 4 + 1 = 5$

Allocate 5 to P_0. P_0 executes & finishes. $|A| = 10$

Allocate 9 to P_2. P_2 ——— " ———. $|A| = 12$

∴ Safe seq: $P_1 \rightarrow P_0 \rightarrow P_2$.

∴ Current state is safe.

- If current state S is safe, and some req. by P_i comes for x, y, z no. of R_a, R_b & R_c, then we can allow it. Else P_i will have to wait.

- The state of the system can change when :

(1) A process releases resources.

(1.1) If state was safe before release then new state is also safe.

(2) A process request resources.

(2.1) If granted, then new state may/maynot be safe.

So, first we check if granting the resources would lead to unsafe state. If yes, we don't grant the resources as of now.

else grant & go to new state (which we checked was safe).

Ex.

	Allocated A	Max Req. A	Need A	
P_0	5	10	5	Total A = 12
P_1	2	4	2	
P_2	2	9	7	

Now suppose P_2 requests for 1 A at current state check if the req. can be granted.

Curr. Remain. A = 12 - 5 - 2 - 2 = 3

Say P_2 is granted 1 A ⇒ #A remaining = 2

We don't have any safe seq. So new state would be unsafe. So request not granted.

Ex. Determine if following state is safe or not.

	Alloc.				Max Req				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	3	0	1	4	5	1	1	7	2	1	0	3
P_1	2	2	1	0	3	2	1	1	1	0	0	1
P_2	3	1	2	1	3	3	2	1	0	2	0	0
P_3	0	5	1	0	4	6	1	2	4	1	0	2
P_4	4	2	1	2	6	3	2	5	2	1	1	3.

Avail = (1, 0, 0, 2)

Seq 1 : P_1 : Avail = (3, 2, 1, 2)

P_2 : Avail = (6, 3, 3, 3)

P_3 : Avail = (6, 8, 4, 3) {or P_0 or P_4}

P_0 : Avail = (9, 8, 5, 7)

P_4 : Avail = (13, 10, 6, 9).

∴ Seq 1 is safe.

Hence current state is safe.

* Deadlock Detection & Recovery :

- Assume that there is a mechanism to detect deadlock which runs either periodically or on some event, then the methods for recovery are :

(1) Recovery through preemption : This isn't same as deadlock prevention as it's happening after deadlock. The idea is to take resource from some other process.

(2) Recovery through rollback : Checkpoint a process periodically. Restart the process using checkpoints if it is found deadlocked.

- One of the detection algorithm can be the first part of Banker's algorithm.

- If all resources have single instance, then a cycle in wait for graph means deadlock.

- Detection Algorithm

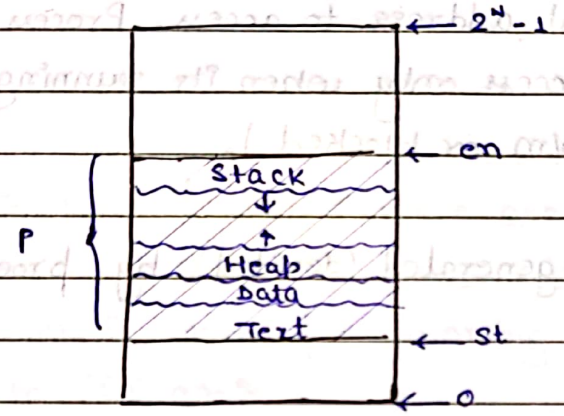
Just like Banker's algo. we'll have : Alloc, Available, Max matrix, along with Ri matrix.
Ri [x][y] : At instn i, the no. of Ry requested by Px.

We run the Banker's algo. replacing Need with Request matrix.
If no way to satisfy current Requests, then deadlock.

Memory Management

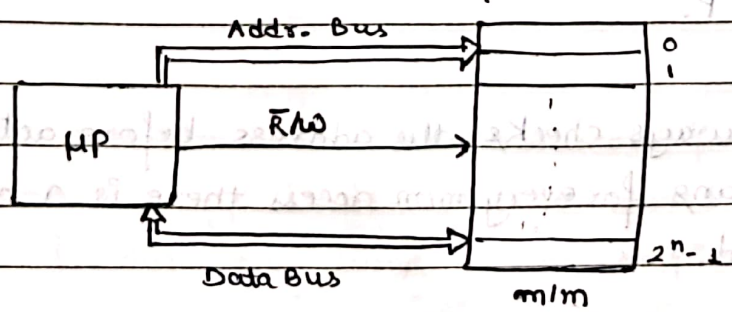
* Address Space of the Process

- It is the range of addresses that a process is allowed to access.



So address space of P is [st, en].

- We want the concept of address space and restrict the access that the process gets to m/m because during execution on CPU, CPU performs Fetch-Decod-Execute cycles, and we don't want CPU to say be redirected to some insl^n not part of the process, or get access to some loc^n where it shouldn't. So, basically security concerns.



- Lets assume process P is executing on the CPU with address space : [st, en] (so size = en-st+1).

Assume that P wants to access some m/m loc. A (maybe for next inst(PC), or some CALL or LOAD inst^n, etc.). In any case, the process decides what address to access. Process P can decide the location to access only when its running on CPU (not while its just in m/m or blocked).

Whatever addr. is generated (decided) by process P is put on the address line.

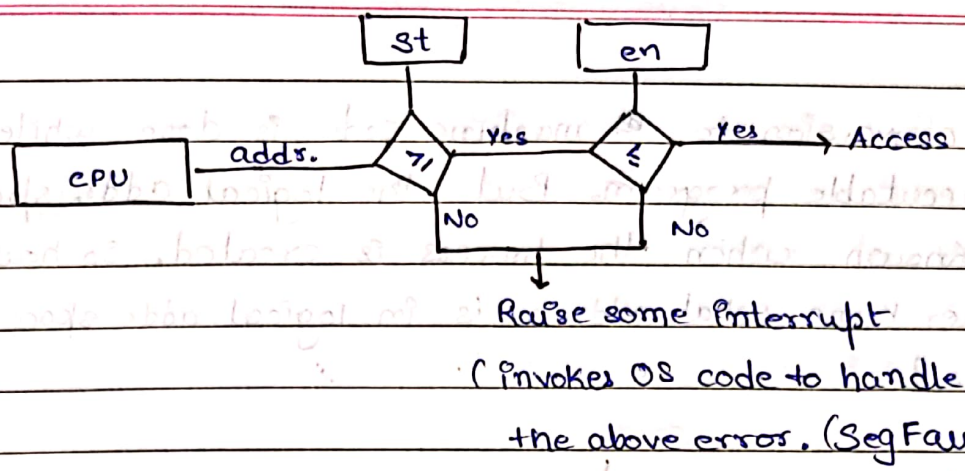
- Logical Addresses : The addresses generated by the CPU (when a process P is running on it) are called Logical Addresses.

- Logical Address Space : The range of all addresses that the CPU can generate for a process P is called its logical address space.

- Since CPU generates address as told by the process P, and this means CPU might generates some address outside range allowed for P.

Sol^n: CPU always checks the address before actual access. This means for every m/m access there is a checking overhead.

Performing such check in s/w would be too slow.
So, CPU has a Memory Management Unit (MMU) component.



- The MMU is part of CPU & hence are set for each process right before it is resumed, i.e., during Context Switch. When a process is moved from running state, these values are copied to PCB again.

- There is a more common design for the checking op^n:
  base = st limit = en - st + 1.

- Based on this design the starting or base address of the process is decided by the OS at time of creating the process.

- For OS code the base addr is set to 0 & limit to N where N is size of m/m.

*. Since CPU generates addresses based on process P's inst^{n}s, P should know it's logical address space starts at "base" & has "limit" no. of addresses locations during compilation.

Ex. Say this line's in our C code: { int x = 0; } {local scoped}
This needs to be converted to H/w inst^n: STORE a, 0;
where a is addr. of variable x.

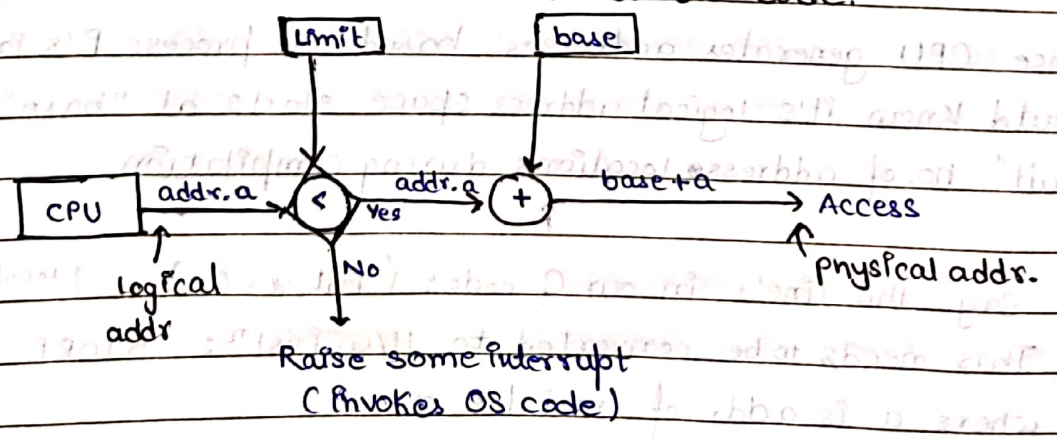
This conversion to machine code is done while compiling to executable program. But the logical addr. space is only known when the process is created. So how will compiler know what addr. a is in logical addr space of the process?

- One possible sol^n is for us as user to tell to the compiler, but for us to know [st, en] we'll have to first create the process so OS allocates some slot. This doesn't really make sense.

- Another sol^n is to let the compiler use a fixed logical addr. space [0, limit - 1], for some limit. So the actual addr in m/m for x would be $a' = a + base$
value of base is given by OS.

So this is like saying: x exists a loc. away from start of my prog., during compilation.
While accessing in m/m we get to know the start as "base", So the actual addr becomes $a' = base + a$
offset from start of prog.

This scheme is called Relocatable Code.



- So we can now differentiate b/w addresses:

(1) Logical Address: The address generated by the CPU.

(2) Physical Address: The address m/m receives for access.

Ex. In previous example while accessing variable x:

Logical Addr space: [0, limit - 1]

Physical Addr. Space: [base, base + limit - 1]

So a is logical addr & base + a is physical addr.

- This entire checking & translation happens in H/W for speed & efficiency. In MMU.

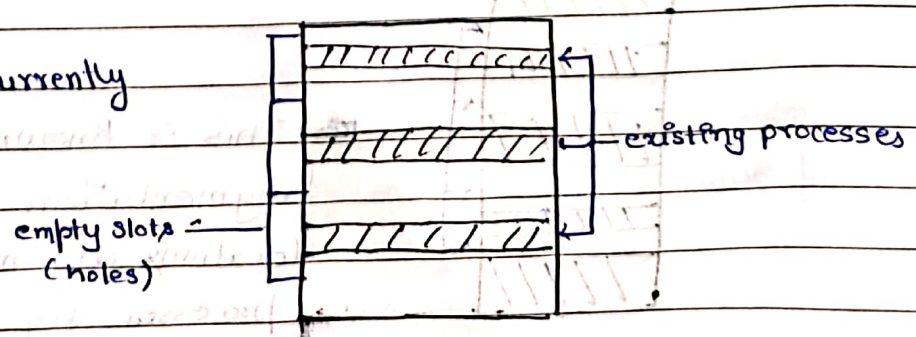
* Memory Allocation

- In multiprogramming OS, there can be multiple processes that can exist in m/m. If a new process is submitted where should it be put in memory?

(Assume user provides process size while submitting it. {by fork})

- Suppose that there isn't enough space in m/m for the process. Fork would return some neg. value.

- Suppose our m/m currently looks like this:



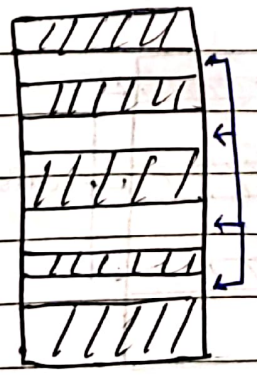
- First Fit: One of the solⁿ for allocating space for new process in m/m, is First Fit algorithm. OS keeps track of empty ranges (holes) in a list. when a new process is created it starts checking the list from the head & allocates the first hole of size large enough to accomodate the process.

- Best Fit: When a new process comes, traverse entire list of hole records & select the smallest out of all holes which are large enough to hold the process.

- Worst Fit: When a process comes, traverse entire list of hole records and select the largest of all the holes which are large enough to hold the process.

- Contiguous Memory Allocation: The process is allocated in the RAM in contiguous fashion. From base to base + limit all loc. are in addr. space of the process.
All three above are allocation algo. for contiguous m/m allocation.

- In contiguous m/m allocation, we may end up with holes so small that they can't accomodate any single process. If multiple of such wasted holes are combined, then it might be able to accomodate some process.



This is known as External fragmentation, since these m/m locations are not part of any processes log. Addr. space.

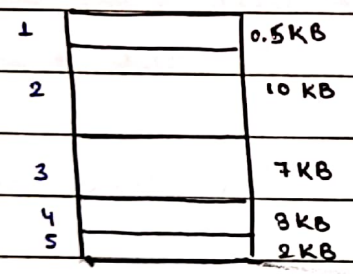
- One sol^n to External fragmentation is Compaction, i.e. move all the processes in one direction (say ↑ towards higher addr) and all holes towards another direction (say ↓ towards lower addr). All the holes would combine to form larger hole.

Since we are using separate Logical Addr Space & Physical Addr Space (relocatable code) we just update the base register for the process.

Compaction is an extremely expensive sol^n.

- Another sol^n is to divide the m/m into some fixed number of chunks (could be of variable sizes), such that each chunk is large enough to run some program.

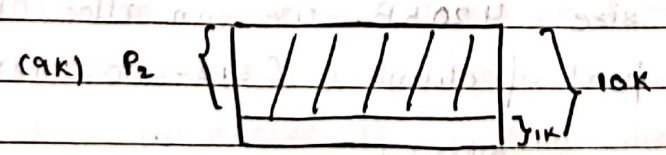
Ex.



Say a process comes of size 10K, assign it to chunk 2.

Say a process of size 9K comes then. Combine chunk 3 & 4 and assign it to this process.

This also has a problem. For P_2 of size 9K, we assign 2 chunks, combined size of which is 10K.



So this time m/m hole which is part of process P2's m/m is wasted, since this chunk is already allocated to P_2, even if some process comes of size 1K it can't be accomodated here.

This is called Internal fragmentation since the hole is internal to some process's address space.

* Paging

- We saw that allocating m/m contiguously wherever we find large enough hole had the disadvantage of "External Fragmentation".

- In paging, we first divide the m/m into equal size "chunks". We then perform allocation at chunk level now. Say our m/m is of size N & we have chunksize as K, then no. of chunks = $\frac{N}{K}$.

Ex.

0		256 KB
1		256 KB
2		256 KB
3		256 KB
4		256 KB

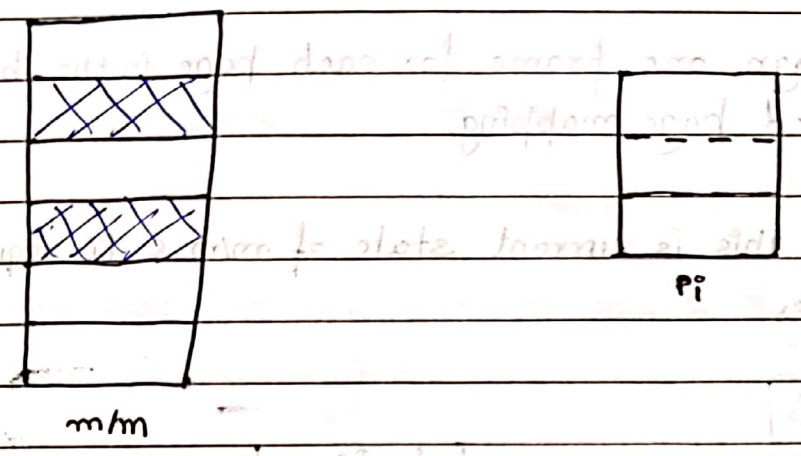
Now say P_1 comes of size 512 KB. We can allocate Chunk 0 & 1.

Now P_2 comes of size 420 KB. We can alloc. Chunk 2 & 3. But some part of chunk 3 (512 - 420 = 92 KB) isn't used.

Although these 92 KB is ~~part~~ allocated of P_2 as Chunk 3, it is unused.

This is "Internal fragmentation".

Ex. Say this is current state of m/m & process Pi requests 3 chunks to accomodate itself.



At the current state we don't have three contiguous ~~that~~ chunks to allocate to Pi, but we do have three free chunks overall.
If we had a way to divide ~~three~~ the process Pi into three chunks & accomodate in three non-contiguous m/m chunks.

This is what Paging exactly does!!

- In paging we divide the physical m/m into equal size partitions called "Frames".
The size of a frame is called "Frame Size".

When a process comes, it is divided into pages with each page having size equal to frame size (except last one. It may be less than frame size). Hence frame size is also called as page size. So,

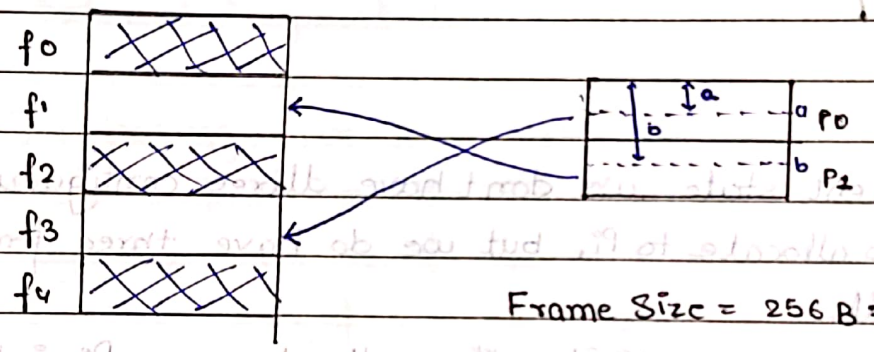
$$\text{Page Size} = \text{Frame Size} = K.$$

Say m/m is of size N, then, no. of m/m frames = $\frac{N}{K}$

Now say a process of size L comes. The no. of frames required would be = $\lceil \frac{L}{K} \rceil$

So, we assign one frame for each page in the process. This is called page mapping.

Ex. Suppose this is current state of m/m & P_i require two frames.



Frame Size = 256 B = K

Page No.	Frame no.
0	3
1	1

Now let's say we want to access some var. x which is in Page 0. Say x is also a locations away from start of process.

So, for this we'd have to know the start addr. of f_3.
So then physical addr. of $x = St_{f_3} + a$.

Say we want to access y, which is in Page 1. Say it's b locations away from start of process, then its (b-K) locs. away from start of P_1.
Then if we know the start addr. of F_1, say, St_{f_1} then physical addr. of $y = St_{f_1} + (b-K)$.

We can generalize this: Suppose we want loc. a of the process residing in page$_i$ & page$_i$ corresponds to frame$_j$ with starting addr or base addr St_{f_j}, then

$$\text{Physical Addr. of } a = \underbrace{(a - i \times K)}_{\text{offset from frame start}} + St_{f_j}$$

We can also find page no. of loc. a of the process, as

$$\left\lfloor \frac{a}{K} \right\rfloor$$

So,

$$PA(a) = \left(a - \left\lfloor \frac{a}{K} \right\rfloor \times K\right) + St_{f_j}$$

(K is page/frame size)

We can further shorten this:

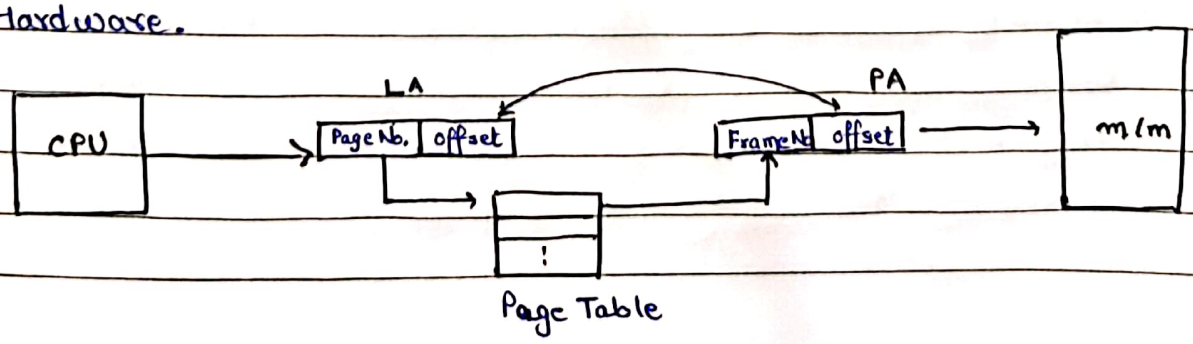
$$PA(a) = (a \% K) + St_{f_j}$$

$$= \underbrace{(a \% K)}_{\text{offset within frame}} + \underbrace{j \times K}_{j \text{ we get from Page Table}}$$

This is translation of Logical Addr. to Physical Addr. and is done by MMU.

The Page Table is also stored in PCB of the process.

The above formula works well in base-10, but digital systems work in binary & this same operation can be optimized for Hardware.



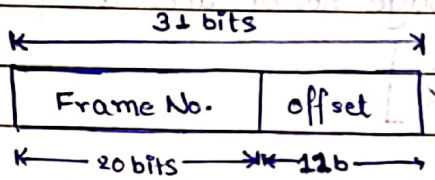
Ex. Say m/m is 4 GB. Page size is 4 KB. M/m is word addressable with 2B word size.

M/m size = 2^{32} B = 2^{31} words

Page size = 2^{12} B = 2^{11} words.

$$\#\text{ m/m frames} = \frac{2^{31}}{2^{11}} = 2^{20}$$

M/m addr: (Physical Addr.)

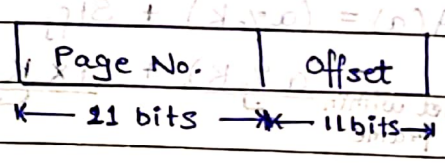


Now say a process comes of size 8 MB.

Process size = 2^{23} B = 2^{22} words

$$\#\text{ pages} = \frac{2^{22}}{2^{11}} = 2^{11}$$

Logical Addr:



Page Table Size:

Page table entry: (Frame No.,)

entry size: 20 bits.

entries = # pages = 2^{11}

PT size = $2^{11} \times 20$ b = 40 Kb

- For any m/m access, the CPU generates the logical addr., the MMU then has to translate the given page no. to frame no. For this, the page table needs to be looked up. The page table is stored in the PCB of the process in the main m/m. So first the page table needs to be read, and then the desired location in the m/m can be accessed.

So, for each access to m/m by the process, the processor has to make two m/m accesses. So, efficiency = 50%. So we require faster page table lookup.

One of the possible sol^n to improve this efficiency is to copy the page table into some cache. CPU has a special cache to store page table: Translation Lookaside Buffer. Generally TLB size is smaller than PT size, so only a part of PT lives in the TLB at any moment.

Since each process has its own logical address space, each process has its own page table. When a process is running, part of its page table is copied to TLB. When there is a Context Switch, the TLB is flushed (discard current entries) since at a time TLB only store "one" process's PTEntries. When new process is loaded, its entries fill up the TLB.

We can have a design where PTEs of multiple processes can be stored in TLB at a time. The entries then would have an additional value in TLB: Addr. Space Id (ASID) which is basically the PID.

- In the Page Tabele, we currently know that the entry contains frame number. Entries may additionally consist of Valid/Invalid bit. The purpose of this bit is to indicate if that page of the process is ever accesed. This is done, so that only valid pages are allocated frames & efficiency improves.

- There can be a situation where multiple processes currently in the m/m use some library code which is same. In our current design each process will have some dedicated pages in the m/m for this library code. (say K pages). Suppose there are n such process, then it results in nK frames being occupied instead of K.

This library code can actually be shared b/w the different processes, through a "method" called "Dynamic Linking".

Multilevel Paging

- The H/w assumes that the page table for a process is stored in contiguous space in m/m. When there is a TLB miss, the H/w fetches the PTE from main m/m. but from where in main m/m?

If Page Table is stored in contiguous m/m, then the H/w can easily compute the required addr. of page table given that starting addr. of page table is known to H/w (stored in some register). But there can be situations where the Page Table is so large that it doesn't fit in a single contiguous m/m area. Also it is not necessary that enough m/m is free contiguously

to store the Page Table contiguously. We are back at our original problem of "not having enough contiguous space in m/m to store a process".

So, what is the sol^n this time : "Paging"!!

- Multilevel Paging
  - When, for a process, its page table is too large to fit within a single frame, we perform paging on the PT. Say for process P, we page & we get PT_1. This PT_1 is too large for a single frame, so we page PT_1 & we get PT_2 for it. If PT_2 is also too large for a single frame, we repeat the process, until we reach a level l, such that PT_l can fit in a single frame.

    This is "Multilevel Paging".

Ex. Consider a system w/ Physical Add. Space of 256MB & Virtual Addr. Space of 8GB. Page Size = 4KB in the system.

PA Space = 256MB = 2^{28} B ∴ Physical Addr = 28 b.

VA Space = 8 GB = 2^{33} B ∴ Virtual Addr = 33 b.

Page Size = 4KB = 2^{12} B ∴ #Offset bits = 12 b.

∴ #frames = 2^{16} # pages = 2^{21}

So, PTE Size = 16 bits

∴ PT Size = $2^{21} \times 16b$ = 32 Mb = 4MB.

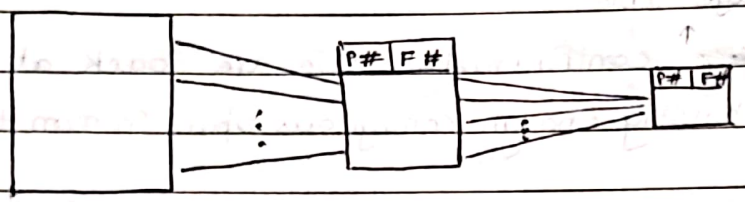
But this Page Table doesn't fit into single frame (4KB).

So, we perform paging on this.

PT_0 size = 4 MB = 2^{22} B Page Size = 2^{12} B $\frac{2^{12}}{2^{5}} = 2^{11}$

∴ # pages req. = $2^{22}B / 2^{12}B$ = 2^{10} #PTE's per page =

∴ PTE_1 Size = $2^{10} \times 16b$ = 16 Kb = 2 KB (This fits in one frame, so no more paging required).



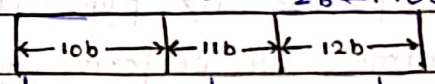
Process → PT_0 → PT_1

pages = 2^{21} | # pages = 2^{10} | # pages = 1 , # PTE in 1 page = 2^{10} entries

PT Size = 4MB | PT Size = 2KB

PT entries/page = $\frac{2^{12}B \leftarrow \text{Frame size}}{2B \leftarrow \text{PTE size}} = 2^{11}$

So, Virtual Addr:

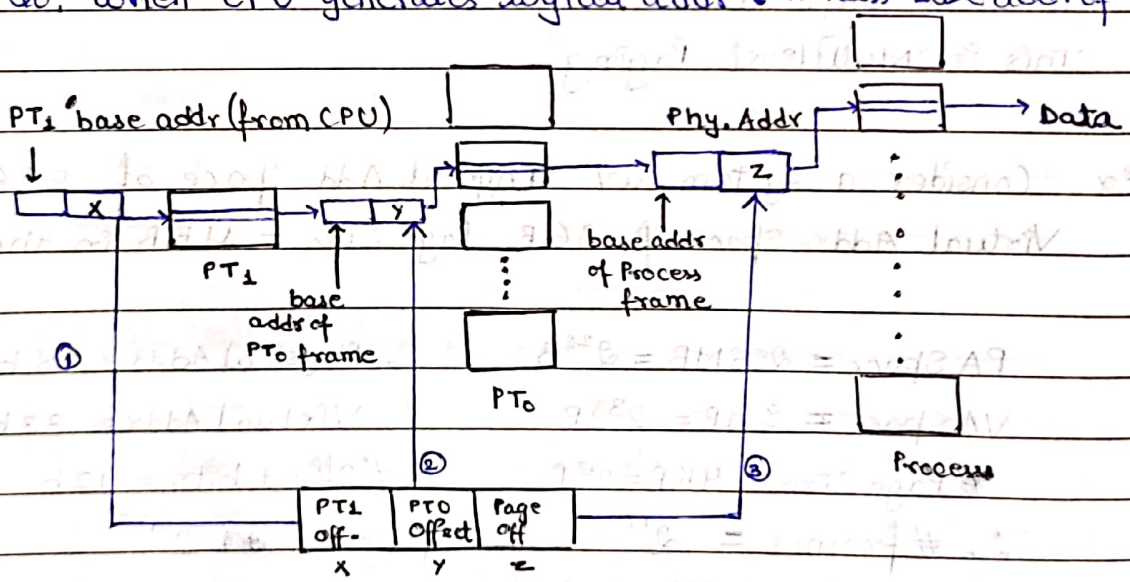


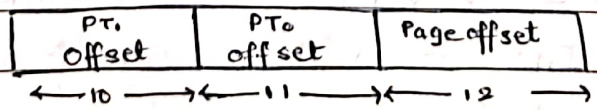
Offset in PT_1

Offset in PT_0

Offset in desired page

So, when CPU generates logical addr (it has base addr of top level PT):



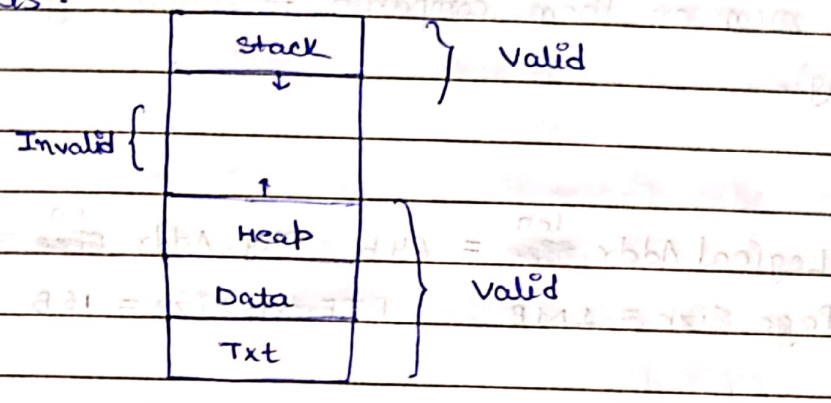


entries in PT_1 (single page) = 2^{10}

entries in each PT_0 frame = 2^{11}

words in each process page = 2^{12}

- Another motivation of Multi-level Paging: When a program is copied into m/m, and process is created, it looks something like this:

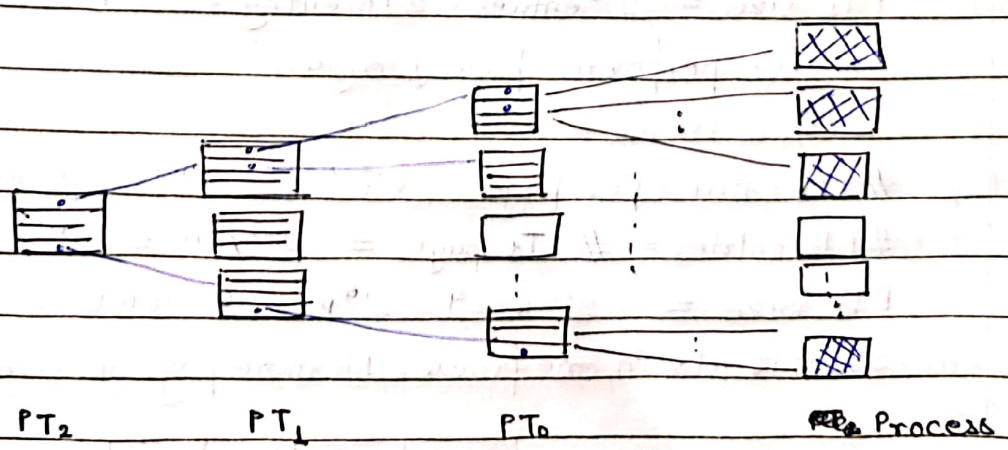


logical addr space allowed

Although the ~~process~~ may be of size: 2^{10} pages, but it might be the case that process uses only 2^8 pages ~~are valid~~ (i.e. accessed). So, storing 2^{10} entries in the page table is inefficient. So, multilevel paging helps improve m/m efficiency.

How?

Since not all entries in lower level Page Table (say PT_0) are valid, ~~we~~ and we've divided them (PT_0) into multiple pages, we can store the PT_0 frames with at least a single valid entry into m/m & PT_0 frames with no valid entry will not be allocated frame. And since it isn't allocated frame (say $PT_{of\ 10}$), the entry for it in PT_1 will be made invalid. This can then be repeated for PT_1.



At each level of PT, we can see frames with no valid entry.

These don't need to be stored in the m/m frames. So we save m/m on them, compared to flat paging (single level paging).

Ex. Logical Addr ~~Size~~ len = 64 b Phy. Addr. ~~Size~~ len = 64 b

Page Size = 1 MB PTEntry Size = 16 B

Page Size = 2^{20} B ∴ Page offset bits = 20

PTE size = 2^4 B

pages ~~in~~ of process = $\frac{2^{64}\ B}{2^{20}\ B} = 2^{44}$

∴ PT_0 size = 2^{44} entries × 2^4 B/entry = 2^{48} B > 2^{20} B (Page size)

(There might not be contiguous space available for PT_0, so its better to perform paging on it as well)

~~on~~ PT_0 entries per frame (at max) = $\frac{2^{20}\ B\ (\text{page size})}{2^4\ B\ (\text{entry size})} = 2^{16}$ entries

PT_1 entries = # PT_0 pages = $\frac{2^{44}\ \text{entries}}{2^{16}\ \text{entry/page}} = 2^{28}$ entries.

PT_1 size = 2^{28} entries × 2^4 B/entry = 2^{32} B > 2^{20} B.
(so we perform paging again.)

PT_1 entries per page = $2^{20}_B / 2^4_B = 2^{16}$ (at max)
PT_2 entries = # PT_1 pages = $2^{28} / 2^{16} = 2^{12}$ entries.
PT_2 size = $2^{12} \times 2^4 = 2^{16}$ B = 64 KB < Page size.
So, this fits in one frame. No more paging required.

Virtual (Logical) Addr:

12 b	16 b	16 b	20 b
PT_2 off.	PT_1 off.	PT_0 offset	Process page offset

Note: Here we have filled PT_i's page with max. no. of PT entries possible in a single page, i.e. 2^{16} entries. We may have chosen to fill it with variable no. of entries at any level.

Say at level 1, (i.e. PT_1) we only fill each PT_1 with 2^{10} entries.

Then # PT_2 entries = # PT_1 pages = $\frac{2^{28} \text{ entries}}{2^{10} \text{ entry/page}}$ = 2^{18} entries.

PT_2 size = $2^{18} \times 2^4 = 2^{22}$ B > 2^{20} B (Page size).

(So, we need to perform paging again).

PT_2 entries per frame (at max) = $2^{20}/2^4 = 2^{16}$ entries.

Suppose we only choose to fill each page with 2^{12} entries.

So, # PT_3 entries = # PT_2 pages = $\underbrace{2^{18}}_{\#PT_2 \text{ entries}} / \underbrace{2^{12}}_{\#\text{entries per page}}$ = 2^6

PT_3 size = $2^6 \times 2^4$ B = 2^{10} B = 1KB < Page Size.

∴ No more paging needed.

So, VA:

6B	12B	10B	16B	20B
PT_3 off.	PT_2 off.	PT_1 off.	PT_0 off	

Ex. Phy. Add Space: 36 bit Virtual Address Space: 32 bit PTE Size: 4B

Page Size: 4KB.

VA:

L_1	L_2	L_3	Page off
3	8	9	12

The system uses three level paging. What is the no. of bits used to store the next level Page Table (Page frame) address in each PTE of L_1, L_2 & L_3 resp.

At level 3: PTE stores address of Page frame/m/m frame. wkt m/m frame size = 4KB = 2^{12} B, So # m/m frames = $2^{36}/2^{12} = 2^{24}$

∴ In PTE of L_3 Page Table each entry uses 24 bits to store addr. of m/m frame.

At L_2:

In the PTE of L_2 Page Table, the address stored is of next level page table. Unlike Page Frame, we don't have a name for PT page. So, let's call it PTFrame & try to draw analogy.

Page Frame	PT Frame
• A process is divided into pages, each of size "Page Size". So the m/m is logically also divided into frames (slots) of size "Page Size".	• A Page Table is divided into pages with each page storing 2^K entries generally, each entry of size PTESize. So, we can divide the m/m into frames (slots) of size $2^K \times$ PTESize.

So, from VA format:

3	8	9	12
L_1	L_2	L_3	POff

wkt each L_3 page table frame stores 2^9 PTE & size of PTE is 2^2 B

$\therefore$ Size of L_3 PTFrame $= 2^9 \times 2^2$ B $= 2^{11}$ B $= 2$ KB.

$\therefore$ We can divided our 2^{36} B Phy. m/m into logical partitions of size 2KB, for L_3 Page Table.

So, # frames at $L_3 = \dfrac{2^{36}\text{ B}}{2^{11}\text{ B}} = 2^{25}$.

So, At L_2, each PTE stores 25 bit address for L_3 PTFrame addr.

At L_1:

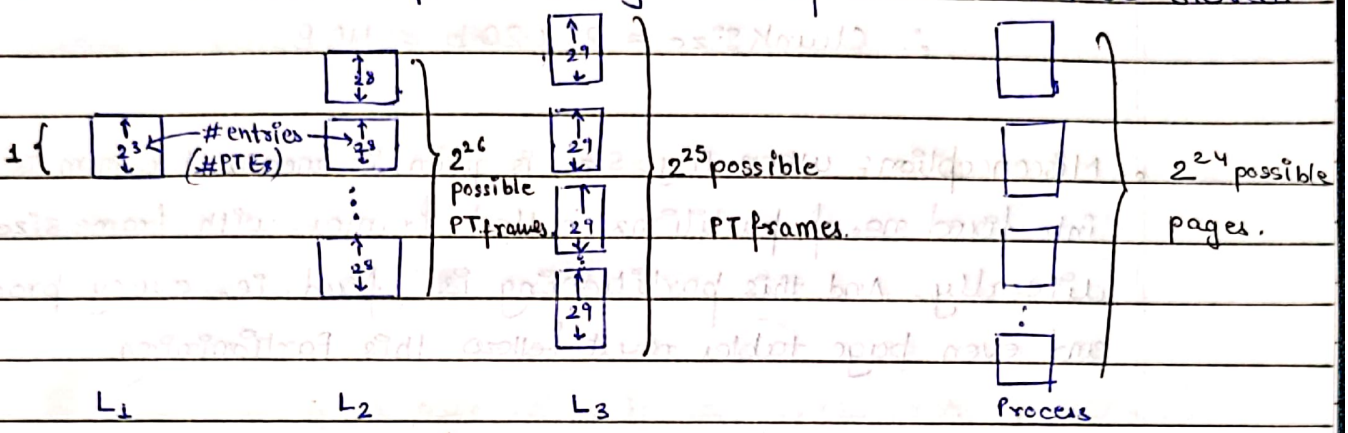
Size of L_2 PTFrame $= 2^9 \times 2^2$ B $= 2^{10}$ B.

frames at $L_2 = \dfrac{2^{36}\text{ B}}{2^{10}\text{ B}} = 2^{26}$

$\therefore$ At L_1, each PTE uses/stores 26 bit address for L_2 PTframe addr.

In Base Register:

We know at L_1, there is only one contiguous Page Table, ~~still we~~ so in MMU, the complete starting addr. of this PT can be stored.



- So, we can arbitrarily choose the no. of entries we want to store at any level Page Table

Ex. PAS: 32 bits VAS: 32 bits Page Size: 2^{16} B.

VA:	4	10	8	16
	L_1	L_2	L_3	Proc.

3 level paging. Find the chunk size (PT frame / Page Frame size) at each level. (Assume at each level, PTE only consist of address of nxt lvl frame/chunk)

At process level: ChunkSize = Frame Size = Page Size

$\Rightarrow 2^{16}$ B.

$\therefore$ #chunks = # frames = $\frac{2^{32}\text{B}}{2^{16}\text{B}} = 2^{16}$

At L_3: Since each PTE only consists of addr. so PTESize = 16b

entries per chunk = 2^8

$\therefore$ Chunk Size = $2^8 \times 16b = 2^9$ B.

#chunks = # L_3 PT frames = $\frac{2^{32}\text{B}}{2^9\text{B}} = 2^{23}$B

At L_2: Since each PTE only contains addr. of next level frame, so PTESize = 21b

#entries per chunk at L_2 = 2^{10}

So, chunk size = $2^{10} \times 21b$ = $2^7 \times 23$ B

#chunks = # L_2 PT frames = $\frac{2^{32}\text{B}}{2^7 \times 23\text{B}} \simeq 2^{20}$

At L_1: Each PTE contains only addr of L_2 frame. So,

PTE Size = 20 b.

PTEs at L_1 per chunk = 2^4

$\therefore$ Chunk Size = $2^4 \times 20$ b = 40 B.

- Misconception: when Page Size is given, it means the m/m is divided into fixed no. of partitions called frames with frame size = Page size literally. And this partitioning is fixed, ie. every process, and even page tables must follow this Partionining.

Correct Concept: Memory is logically partitioned into multiple chunks with a given chunk size. A process calls this chunk as page, and hence has page size. So m/m is logically divided for the process into frames of size = page size, so that at a time m/m can store a full page.

Now say we want to store a Page Table. We can divide the page table into small chunks as well (in lack of contiguous space). Now from perspective of this PT, the m/m is logically divided into frames of size = chunk size that it has used.

So, m/m is contiguous & is "logically" divided into different size chunks for different usecases. Process divides into Page size chunks. Page Table can divide into Page Size or a different chunk size.

Analogy: In C, for char m/m is divided into 1B chunk, for int into 4B chunk, for long 8B, etc.

Ex. In a 32-bit m/c. we subdivide the virtual addr into:

10-b	8-b	6-b	8-b

using 3-level paging scheme.

(1) What is the page size in system?

Page offset len = 8b

$\therefore$ Page Size = 2^8 words

(2) What's the size of a page table for a process that has 256K of m/m starting from addr. 0. Assume PTE size at each level = 2B.

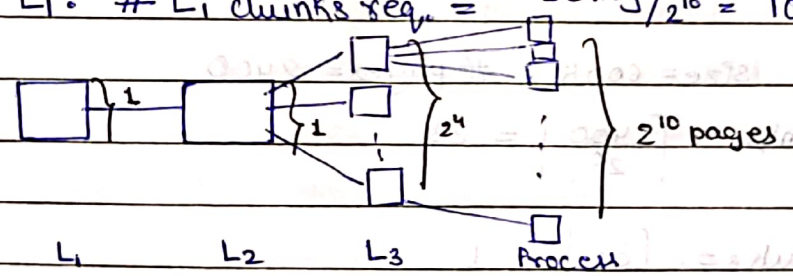
Process size = 256K words

$$\# \text{pages} = \frac{2^{18} \text{ words}}{2^8 \text{ words}} = 2^{10} \text{ pages.}$$

At L_3: # L3 chunks req. = $\frac{\text{Total entries}}{\text{entries per chunk}} = 2^{10}/2^6 = 2^4$ chunks.

At L_2: # L_2 chunks req. = $\frac{2^4 \text{ entries}}{2^8} = \lceil 0.. \rceil = 1$

At L_1: # L_1 chunks req. = $1 \text{ entry}/2^{10} = \lceil 0.. \rceil = 1$



$\therefore$ # PTEs stored (or reserved) = $1 \times 2^{10} + 1 \times 2^8 + 2^4 \times 2^6$

$\Rightarrow 2304$

$\therefore$ PT Size = # entries x entry size = $2304 \times 2B$

$\Rightarrow 4608$ B.

(3) What is the size of PT for a process that has code segment of 48K starting at 0x01000000, a data segment of 600K starting at 0x80000000 & stack seg. of 64K starting at 0xf0000000 & growing upward. (Assume all PTE at each level has size 2B).

Code Seg:

Size = 48K

$\#\text{pages} = \frac{48 \times 2^{10}}{2^8} = 48 \times 2^2 = 192$ ← page size.

$\#L_3$ chunks $= \frac{192}{2^6} = 3$

$\#L_2$ chunks $= \left\lceil \frac{3}{2^8} \right\rceil = 1$

$\#L_1$ chunks $= \left\lceil \frac{1}{2^{10}} \right\rceil = 1$

Total Page Table entries $= 3 \times 2^6 + 1 \times 2^8 + 1 \times 2^{10}$

$= 1472.$

Data Seg: size = 600K #pages = 2400

$\#L_3$ chunks $= \left\lceil \frac{2400}{2^6} \right\rceil = 38$

$\#L_2$ chunks $= \left\lceil \frac{38}{2^8} \right\rceil = 1$

$\#L_1$ chunks $= \left\lceil \frac{1}{2^{10}} \right\rceil = 1.$

Total PTEs $= 38 \times 2^6 + 2^8 \times 1 + 2^{10} \times 1$

$\Rightarrow 3712$

Stack Seg: size = 64K #pages $= 2^8 = 256$

$\#L_3$ chunks $= \frac{256}{2^6} = 2^2$

$\#L_2$ chunks $= \lceil 2^2 / 2^8 \rceil = 1$

$\#L_1$ chunks $= 1$

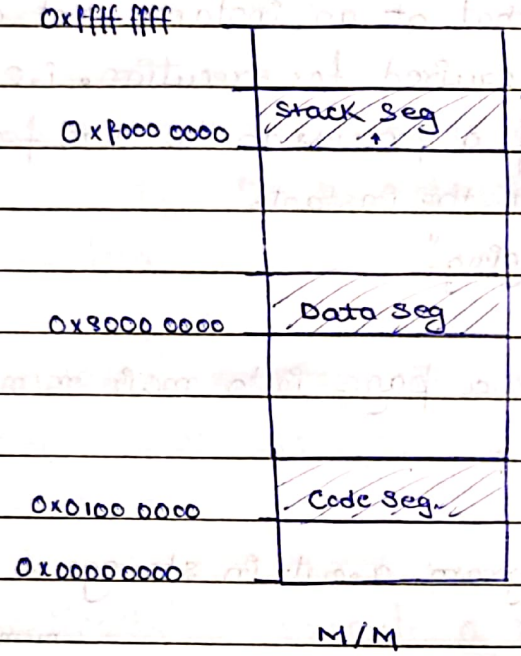
~~$\therefore$ PTEs $= 2^2 \times 2^6 + 2^8 + 2^{10} = 1536$~~

~~$\therefore$ Overall Total PTEs $= 6720$.~~

~~So, Page Table Size $= 6720 \times 2B = 13440\,B$~~

~~$\Rightarrow 13.125\,KB$.~~

The system has VA space of 32 bits. [0x0000 0000, 0xffff ffff]



#pages

$\frac{64K}{2^8} = 256$

$\frac{600K}{2^8} = 2400$

$\frac{48K}{2^8} = 192$

This is how our process resides in m/m

Page size $= 2^8 = \underline{10000\ 0000} = 0x100$

So, all segments start at page start.

$\therefore$ Useful L_3 entries $= 256 + 2400 + 192$

no. of entries per L_3 chunk $= 2^6$

So, no. of useful L_3 chunks $= 4 + 38 + 3$

Useful L_2 entries $= 4 + 38 + 3$

no. of entries per L_2 chunk $= 2^8$

So, no. of useful L_2 chunks $= 1 + 1 + 1$

Useful L_1 entries $= 1 + 1 + 1$

no. of entries per L_1 chunk $= 2^{10}$

So, no. of useful L_1 chunk $= 1$

So, no. of PTEs

$\Rightarrow 45 \times 2^6 + 3 \times 2^8 + 1 \times 2^{10}$

$\Rightarrow 4672$.

So, size of PT $= 4672 \times 2B$

$\Rightarrow 9344\,B$.

* Virtual Memory

- Locality of Reference: In general, while a process runs on the CPU, & accesses some m/m location x, then in near future it access m/m close to location x.

- It is emperically observed that at an instance not all the pages of a process are required for execution, i.e., it isn't necessary for all pages of a process to be m/m for the process to run in the CPU at the instant.
  This gives rise to "Demand Paging".

- Demand Paging: Only "load" a page into main m/m when required.

Ex. Suppose you have a program a.out in storage & it has 10 pages.
In Paging (general) the OS would've allocated 10 m/m frames to the process on execution.

In Demand Paging OS only allocates a few (say 2) m/m frames to the process & the loader loads only initial two pages of the process for execution.
Say, after some time, process needs to access some page which is on disk, then it can also be loaded by OS allocating some other frame for it.

So, only load a page in main m/m when "demanded".

Ex. Now consider a scenario where the logical address space is of 16 pages & physical m/m has 4 frames in system. Consider a scenario when all four frames are allocated to the process pages. Now if the process tries to access another page, then the OS can swap out a page by moving it from m/m frame to secondary storage & swap in the accessed page in the frame.

So, we can run a process with N pages with M CPU frames where $N \geq M$, by replacing pages & only allocating a frame to a page when its accessed.

- The idea of CPU accessing an address space larger than actual addressable physical m/m in system is called Virtual memory.

- "Practically", to execute a process you require two pages of a process at the least (one for text/code section, another for other).

- In Page Table, the PTE contains Valid/Invalid bit.
  If the bit is Valid, it means the page is valid (used by process) and in some m/m frame.
  If the bit is Invalid, it means, either the page is not in m/m or it is invalid (not ever used by process).
  When we were sure (before) that invalid bit meant illegal access, (OS) we raised H/W interrupt. (Say VASpace is 64bit, so we can have 2^{64}B m/m, but a process uses only 2GB i.e VASpace from 0 to 2^{31}B, then, if the process tries to access VA 2^{54}, then it is illegal.)

But now it is possible that access is invalid or page is not in the m/m.

- If page is valid, and it is not in m/m. When it is accessed, and not found in m/m, a "Page Fault" occurs. The OS then either assigns a free frame or replaces a assigned frame & puts the required page in it. OS then updates the PTE & restarts the last instruction.

Accessing secondary storage is slow & hence Page Faults slow down the system significantly.

Ex. Suppose a program has 10 pages. Some for code, some for data. Now it is executed & 2 frames are allocated to it.

Say it is using one frame for Page 1 (txt section) & one for stack section (Page 5). Now say Page 6 is required. There can be many reasons for it; ex:

(1) Some data is required, which can come from executable on the disk.

(2) Stack grows.

(3) Malloc.

Also suppose page 5 is written to (dirty page). So to bring in Page 6 & move page 5 out (to be accessed later) there is a special region on the secondary storage, called "Swap Area". The PT is updated & in PTE it is indicated that the Page 5 is in swap area.

- Since due to demand paging not all pages of a process need to be in m/m, the degree of multiprogramming also increases.

- While discussing relocatable code we said logically the process can access [0, limit] range of addresses for some limit. In case of Paging, if VASpace is K-bits limit becomes $2^K - 1$. VASpace of K-bit means CPU can generate K-bit addresses.
  This K can be much higher than PASpace of l bits because of demand paging & swapping, i.e. Virtual m/m.

- In case of Page Fault there can be multiple overheads:
  - Page Fault Overhead
    - Swap Out time
    - Swap In time
    - Restart overhead.

- When a page fault occurs, the OS:
  - Pauses the running process (by moving it to wait state).
  - Find the PTE for which page fault has occured.
  - Find the location on harddrive where page is stored.
  - Request HDD for the disk (IO operation).
  - When the HDD op^n is complete (page is copied to frame) OS updates the PT.
  - OS moves the process to ready / running state.

(If there's no free frame, then OS chooses a frame {based on some criteria} which may contain some (dirty) page, which needs to be moved to HDD {& PTE updated} before moving required page in its frame.)

- OS is also in m/m. There are frames occupied by OS. (They contain critical information). For proper working of the system OS pages should never be swapped out.

Ex. suppose a process is initiating HDD read, and the HDD completes the IO & raises interrupt. But say the OS page with ISR is swapped out to HDD. So, that page will have to be fetched to service the ISR, which again raises interrupt, and the cycle goes on.

This is just one of the scenario where swapping out OS pages can cause problem. So, OS pages are never swapped out.

Pages which are never swapped out of m/m are called Pinned pages

- Page Replacement Algorithms: When a page fault happens, and there's no free frame available, some frame's page either has to be overwritten or replaced. This page is called as "victim page". To choose this frame to be replaced OS uses Page Replacement Algorithms which make use of various selection criterias.

Some assumptions which are theoretical (may/maynot be followed practically):

- A process is allocated a fixed no. of frames to run.
- If page fault occurs for the process only one of these frames will be replaced, if full. (Local Replacement).

- **First In First Out:** Replaced the page which came into m/m (swapped in) the earliest out of the current pages.

Ex. Suppose Process is of 32 B & page size is 8 B. Two frames are allocated & the following is sequence of access of location:

5, 13, 1, 24, 12, 20, 25, 7

Find page fault rate.

31	
24	P_3
16	P_2
8	P_1
0	P_0

Accessed loc.:	5	13	1	24	12	20	12	7
Accessed page:	0	1	0	3	1	2	1	0

F_0	0	0	0	3	3	3	1	1
F_1		1	1	1	1	2	2	0
	0	1	0	3	1	2	1	0
	x	x		x		x	x	x

$\therefore$ # faults = 6

pf rate $= \frac{6}{8} = \frac{3}{4} = 0.75$

- **Optimal Algorithm:** Replace the page which won't be used for longest time in future. This isn't practical since we don't actually know which pages will be accessed in future.

Ex. Above example.

F_0	0	0	0	3	3	2	2	0
F_1		1	1	1	1	1	1	1
	0	1	0	3	1	2	1	0
	x	x		x		x		x

$\therefore$ # faults = 5

pf rate $= \frac{5}{8}$

- **Belady's Anomaly:** Increasing the no. of allocated frames results in increased page fault. FIFO suffers from this phenomenon.

- **Least Recently Used**: Replace the page which hasn't been used for the longest time in past. It is an estimation of Optimal algorithm, which says, "since a page hasn't been for a long time in past, it would probably not be used in near future".

Ex. 3 frames. Page Refrences: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

f_0	1	1	1	4	4	4	5	5	5	3	3	3
f_1		2	2	2	1	1	1	1	1	1	4	4
f_2			3	3	3	2	2	2	2	2	2	5
	1	2	3	4	1	2	5	1	2	3	4	5
	x	x	x	x	x	x	x			x	x	x

$\therefore$ # faults = 10

- **LRU Implementation**: Suppose that the process is allocated N frames, and we have the Page Table for the process as well. To know which page to replace, we'll have to keep track of the timestamp at which the page was last use. So, we keep & update the information about the timestamp in the PTE for the pages. So, for every page access, we'll have the overhead of updating the timestamp in PT. Also, this must be done in H/W, since PT is accessed in H/W. It can also happen that timestamp field in PTE overflows.

- Alternately we can maintain a stack which contains the info about "which page is used". So whenever a page M is accessed, if we have it in stack, we move it to the top, else if not in stack we push to top. The last entry in stack is the LRU page. This sol^n also costs overhead

per access.

- So LRU has implementation related overheads.
- Hence we approximate LRU as well (which itself is appx of Optimal algo.)
- Approximation 1: In the PT, for each PTE we maintain a few extra bits. When a page is accessed, we set the LSb. Periodically the OS shifts this field called "Reference bits" left, by one place. When a page fault occurs, and all frames are full, the reference bits are checked from right to left. Since the kth bit in a PTE is set (when page is accessed) more recently than $(k+1)^{th}$ bit, by checking 0th, then 1st, then 2nd.. reference bits, we can try to estimate which are the most recently used pages, and hence approximate the LRU page. The LRU page is replaced, its Ref bits are set to 0.

- Approximation 2: For every PTE we've one bit called "Recently Used". When a page is accessed:
  (1) If its already in m/m Set the RU bit if not already set.
  (2) If not in m/m then Page Fault!! & we are looking for a victim page. We have a pointer "ClockHand" pointing currently to some entry in PT. If it is pointing to a page w/ RU bit set, we flip it to False & move ClockHand to next PTE. We repeat the process until we find a page in m/m page's whose RU bit is already False. We replace this page, set the RU for the PTE of new page & increment "ClockHand".
  (This is called Clock / Second Chance Algorithm)

Ex. 3 frames. Second Chance algo. Page Accessed: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.

7_1	$7_1 \rightarrow$	7_1	2_1	$2_1 \rightarrow$	2_1	2_1	4_1	4_1	4_1	$4_0 \rightarrow$	3_1	3_1	$3_0 \rightarrow$	3_0	0_1	0_1	0_0	0_1	$0_1 \rightarrow$
	0_1	$0_1 \rightarrow$	$0_0 \rightarrow$	0_1	0_0	$0_1 \rightarrow$	0_0	2_1	2_1	$2_0 \rightarrow$	$2_0 \rightarrow$	$2_1 \rightarrow$	1_1	$1_1 \rightarrow$	$1_1 \rightarrow$	$1_1 \rightarrow$	7_1	7_1	7_1
		1_1	1_0	1_0	3_1	3_1	$3_0 \rightarrow$	$3_0 \rightarrow$	3_1	0_1	0_1	$0_1 \rightarrow$	0_0	2_1	2_1	$2_1 \rightarrow$	$2_0 \rightarrow$	2_0	1_1
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
X	X	X	X		X		X	X		X	X		X	X	X		X		X

(Subscript represents RU bit) (→ represents Clock Hand)

- There is also an "Enhanced Second Chance" which takes "Dirty Bit" in consideration too.

- There are few other replacement algorithms used, such as:
  (1) Least Frequently Used: Replaces page with smallest count of usage.
  (2) Most Frequently Used: Based on the argument that page with smallest "usage count" was just brought in & has yet to be used.

- Next, the question is, how many frames should a process be allocated.

So far we know that process is executed by CPU in F-D-E Cycle. In general the code & Data (Stack, Heap, Global) are in different pages. So we at least need one page for code & one for data. But then we'd have too many page faults.

Demand Paging doesn't actually put any restriction on min or max no. of frames a process can use.

Actually, the answer depends on both OS & system architecture.

- For a process with high priority, if it demands more m/m (frames) it should be given that. If there are no free frames, some low priority process's frame would be taken. This slows the system down since we've to swap out the page of low priority process. Also, if we keep on snatching frames

from low priority process, then it may happen that the frames fall very low. This will lead to that process spending significant time in Page Fault Service.

- This is leading to Thrashing, i.e., too many page faults.
- A simple sol^n to Thrashing is, if the no. of allocated frames for a process falls below some "min. value" swap out the process from m/m until enough frames are available for it.
- Consider a case where the OS scheduler which decides to run jobs based on CPU Utilization. If CPU Utiliz. is low it accepts & runs more jobs, So CPU Utiliz. increases. Again, to run more processes frames are taken away from already running processes & allocated to the new processes. This may also lead to some process ending up with no. of frames below some threshold, which would lead to increased page faults, i.e. Thrashing. The performance of system goes down & CPU Utiliz. is low. The OS will again accept more jobs & run them, & this again increases thrashing & so on.

- The question now is, what should this "number of frames" threshold be.
  - One answer to this is based on Locality of Reference. We keep track of the pages referenced for a process. At any instance, we check the last K (constant) references made, and count the number of unique references. This set of unique page references is called Working Set of the process. The assumption we make is: since the pages in WS were recently used they'll soon be used in future. So, the no. of frames allocated to a process at any instance should be $\geq |WS|$.

    So in a system with F frames & sum of WS for n processes D, if $D >> F$, then page faults ↑. So, move out pages for some pages of some processes (lower priority).

* Libraries.

- We discussed that Compiler should know (or only requires to know) the logical m/m of a process, for creating the bytecode. the OS would then understand what the compiler has done & create process from the bytecode.

- Generally what happens is that the compiler merges different object files, including library code into executable. So, the executable has the code of the libraries along with our code. This is called Static Linking.

- This is fine for small libraries, but for larger libraries, including everything into one executable is not feasible. Also these are "libraries" which are used by many codebases. If all (many) such executables w/ same library are run, then the same code would have multiple copies in the m/m.

- We can easy improve on this by sharing the pages w/ library code, by updating the Page Table of processes using the library to point to a unique frame w/ unique copy in m/m for that library.

- This requires both Compiler & OS support. These library files are stored in some location added to system path. The compiler is informed to compile without them & expect them to be available in m/m during runtime. Compiler puts stub code which communicates w/ OS to get the library pages' info. The Dynamic Linker is the OS module responsible for providing this info. & loading the required code in m/m.

- This is called Dynamic Linking & the libraries are called Dynamic Libraries.

- In Physical m/m, the initial few frames are allocated to OS. In the PT of every process, the PTE of initial few pages 0,1,2.. point to OS frames. The OS frames are protected though (i.e. only accessible in Priveledged/Kernel mode).
So, when a process makes Syscall, the mode bit is set to Kernel mode. Now for CPU to access OS code it needs to access OS Page Table. But the CPU right now has access to process's PT, and it can only generate logical addr, whatever addr. it generates will be translated using process PT. Since process PT has record for OS frames in initial few PTEs, the access is possible.

* Inverted Paging

- In the basic form of paging we had the issue that the PT for a process (flat) had to keep track of all possible pages in VASpace even if the process size was low. This lead to some PT being larger than the process itself.

- One possible soln is to use Inverted Page Table, with an entry for every frame, storing the info: <PID, Page No>, letting us know that m/m frame i stores the jth page of Process x.

- So, when CPU generates logical addr: (pg no, offset), we combine (pid, pg no) & lookup in Inverted Page Table. So, this saves space, but due to searching, this takes longer, compared to simple Page Table where we are just indexing by pg.no., so in $\Theta(1)$.

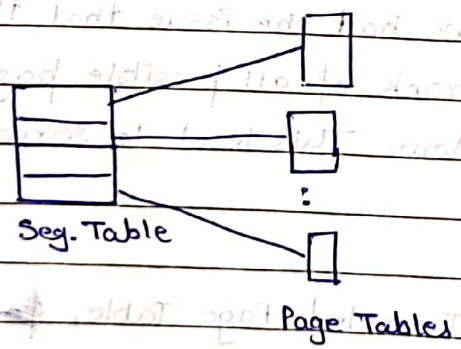
* Segmentation

- We know that logical m/m space of a process has multiple segments : Txt, Data, Stack & Heap. Process can generate address in any segment.

- It is easier if we do it in segments, by the logical addr. saying : "Give me x^{th} byte of ith segment".
  So, logical addr = (base addr of segment + offset)

- This is also helpful for compiler.
- This is called Segmentation, where we have segment table with each entry ST[i] gives the frame no. (translation info) for i^{th} segment of the process.

- One famous design is Segmented Paging.
  Logical Addr = (Segment No. + Page No. + Page Offset)



Each segment is paged, so each segment has its own page table.

File System

- Files are stored on secondary storage. The OS module responsible for creating, reading, writing, management of various files is called File System.

- File: A structured set of bits which is interpreted for different purposes.

- Some common operations on file: Create, read, write, update, etc.

- Some Key features of file system:
  (1) The files should be stored in well organized fashion.
  (2) The metadata related to files must be stored efficiently.
  (3) The access from disk should be efficient.

- The smallest division of a Magnetic Disk is Sector. So, the File System will read or write data to disk in unit of sectors. Generally it uses one or two sector as "block" size for unit.
- Directory: In Unix, a directory is a file with some special bits set to indicate that its a directory.
  (In Unix, everything is a file)

- File Metadata: Some common file metadata are:
  Name, Location, Size, Protection bit, Time, date & User info.
  Identifier: Unique tag identifying the file (inode no. in Unix).

In Unix, the metadata for a file is stored in a data structure called "Inode".

* Inode

- Inode in Unix is data structure that describes the metadata of a file. One Inode represent one file or directory. So, every file in Unix is Inode + File Data. An Inode is identified by Inode number.

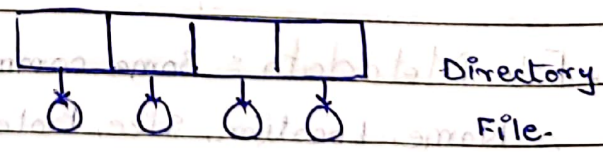
- Inode for a file contains metadata such as file name, date, permissions, etc, along with a pointer for the location on disk where the file is stored, called Data Block Pointer.

- For a directory, the Inode stores pointer to the directory's Inode Table. Inode table is a data structure which stores mapping from file name of files in the directory & their inode number. <filename, inode no.>

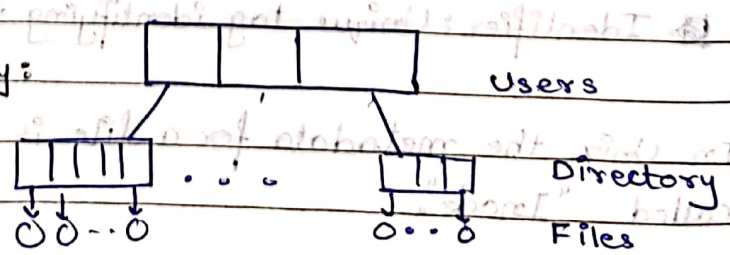
- File system views the disk as a sequence of blocks. The job of file system is: given the file name & offset, find the block, on the disk.

* Directory Structure

- Single Level Directory:



- Two level Directory:



- Multilevel / Tree Structure (Used nowadays) (DAG): Multiple levels of hierarchy, where directory entry can be for file or subdirectory.

* Disk

- In old Unix Systems, the disk was divided into five section:
  (1) Boot block : usually contains bootstrap program.
  (2) Superblock : size, #files, freelist, etc.
  (3) Inode Area
  (4) File Content Area
  (5) Swap Area.

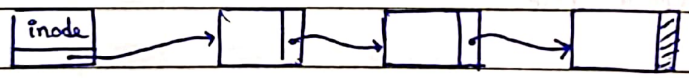
* File Implementation
(Allocation of disk space to files)

- Some key metrics used to evaluate / compare disk allocation strategies: Fragmentation, Appending to file (size increase), access time, implementation complexity, storage overhead, etc.

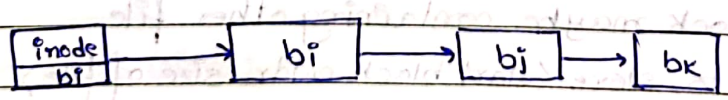
- Contiguous Allocation :
  - User specifies the length upfront & file system allocates all the space at once, i.e., later the file can't grow (increase in size) since the next block maybe containing other file.
  - The inode has to store <Start block addr, size of file>.
  - Sequential & random access is very fast & efficient.
  - Large External Frag., some internal fragmentation.

- Extent Based Allocation: Multiple contiguous regions allocated to file. Metadata would contain small array of multiple <start blockaddr, size> tuples. Each contiguous region is called Extent. File can grow over time. Small internal fragmentation, reduced external frag. compared to contiguous allocation. Fast sequential access. Simple to calculate random access.

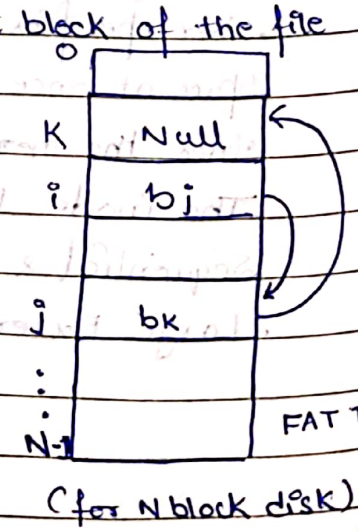
- Linked Allocation: The file is divided into block size (a little smaller than block size) chunks. Each chunk is stored in whichever block is available. Along with the chunk, in the block, a pointer to the block of next chunk is also stored. So the metadata only stores the pointer to the first block. File size can easily increase over time. No external frag. Large storage overhead, per chunk a pointer needs to be stored. Slower sequential access. Extremely difficult random access.



- File Allocation Table based Allocation: Store a File Allocation Table, which contains the block pointers of the linked list. For every block on the disk FAT table would contain an entry: FAT [i] = bj, means that the next block of the file acquiring bi is bj.



So, to get addr. of next block, ~~you~~ while you currently ~~a~~ are at bi, you go to FAT table & check entry FAT [i].



The benefit of this design over Linked Allocation: Suppose you want to access 100th file block. In Linked alloc. you'd have to read block 0, 1, 2... i.e. 99 blocks before getting addr. of 10th block. In FAT based alloc. you just read the FAT table (2-3 blocks maybe) in m/m & check it 100 times. 100 m/m opⁿ is much much faster than 100 IO opⁿs.

Ex. Suppose FAT contains block no. in entries.

(1) If the entry is 16 bits in size, what is the max. no. of blocks?

(2) If each block is 512 B what is the size of FS?

(3) What's the size of FAT?

(1) 2^{16} blocks

(2) $2^{16} \times 512\,B = 2^{25}\,B = 32\,MB$

(3) No. of entries x size of entry

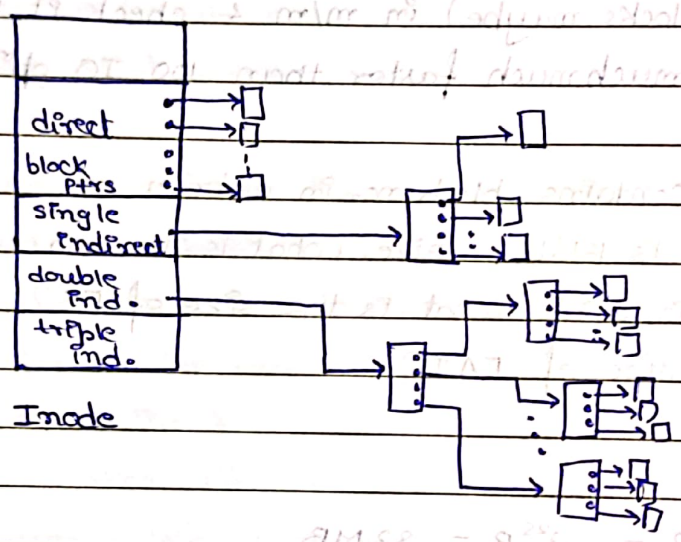
$\Rightarrow 2^{16} \times 16b = 2^{17}\,B = 127\,KB$.

The random access time improves compared to Linked Allocation.

- Indexed Allocation: Although Linked Alloc. solved external fragmentation, it still can't support direct access.

In Indexed Allocation, we store all block pointers for a file in the Inode. Hence, there is a limitation that you must specify the max. size of file upfront. Since Index Table can grow large, it requires large contiguous space. Results in fast sequential access, from Inode, you can get address for any block of the file. It has no external frag. Fast seq. & random access. Index Table overhead increases Inode size. File can grow overtime as long as it doesn't exceed maximum size.

• Multilevel Indexed Allocation: The inode would contain some pointers to data block, some pointer to index table block, some pointers to two-level index table blocks & few pointers to 3-level index blocks.



So, the inode size can remain small, and file size can grow very large. But for access we may have to go through multiple levels to get a block.

Ex. Consider a file system with 2048 B blocks, & 32 bit disk & file block pointers. Each file has 12 direct pointers, a single, double & triple indirect ptr.

(a) How large of a disk can this system support.

(b) What is max. file size?

(32 bit disk means 2^{32} blocks)

(a) 2^{32} blocks $\times$ 2048 B/block $= 2^{43}$ B $=$ 8 TB.

(b) Block size = 2048 B

file block ptrs / block = 2048 B / 4 B = 512

direct data blocks = 12

single level dblocks = 1×512

double level dblocks = $\underset{\uparrow \text{ \# ptrs}}{1} \times 512 \times 512$

triple level = 1×512^3

So, max file size = $(12 + 512 + 512^2 + 512^3) \times 2048$ B.
$\simeq 256.5$ GB.

* Free Space Management

- The OS needs to keep track of free blocks available on the disk to allocate to files.

- The data struct. used to keep track of free blocks:
(1) Must quickly find the blocks
(2) Not occupy much space on disk itself.

- Bit Vector: Maintain a list of one bit per block on disk. If BIT[i] = 1, block i is free, else occupied.
For N blocks it requires N bits & searching for free block takes O(N) & may require multiple disk block read.

(Free list)
- Linked List: Store a single pointer to a free block. That block should itself store pointer to next free block. When a block is required, go to first free block, allocate it & update the head pointer. When a new block is freed, make it head.
Space req. (additional): $\log_2 N$ {N no. of blocks}
Time to find free block: O(1).

- There are two enhancements to Linked list method
(1) Grouping: Store more than one ptr to free blocks within one free block.
(2) Counting: specify start block & no. of free blocks.