9/7/25

Operating System

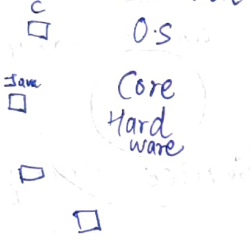
(I) It gives a platform for other applications.

(II) It is a manager, it manages all the tasks like Memory allocations, processing tasks, allocate different I/o devices.

(III) It interact with the Hardware of the machine (system) and the user.

(IV) Os is resource allocator.

Processor cycle



Core Hardware

Resources of the System → Brain of system

RAM, CPU, I/o device, Software files.

- Primary Memory → RAM, ROM
  Secondary Memory → Harddisk

- Single User → laptop
- Multi-level user → Vs code, word etc.

{ Size O·S ↓ 10gb, 12gb

Bootstrap loader / bootstrap program

Location of bootstrap loader is ROM [Read only Memory]



} The process is Booting

O·S ↓ Software

* Throughput – No. of tasks executed per unit time.

* O·S is a System Software which works as an Interface between user and hardware → CPU, I/o devices, RAM

→ Windows (Majority used os)
- Primary goal → convenience (95%) → (82%)
- Throughput (e.g Linux)

market → (Multi) Users

Functionality of OS-

1. Resource Management
2. Process " (cpu scheduling)
3. storage Mgnt (HD) (file system)
4. Memory Mgnt (RAM)
5. Security

15/07/2025

- Book
  Operating System Concepts
  Silberschatz and Galvin, Addision-wesley inc.
- Alternate
  Operating System, Detail & Detail

(1) What is an Operating System?

→ "An Operating System is an integrated set of programs when major function is to Manage Resources (like CPU, Memory, disk etc), control input and output, Schedule jobs, handle errors, provide security acts an an interface b/w the user and machine."

◇ An OS is application neutral and service specific.

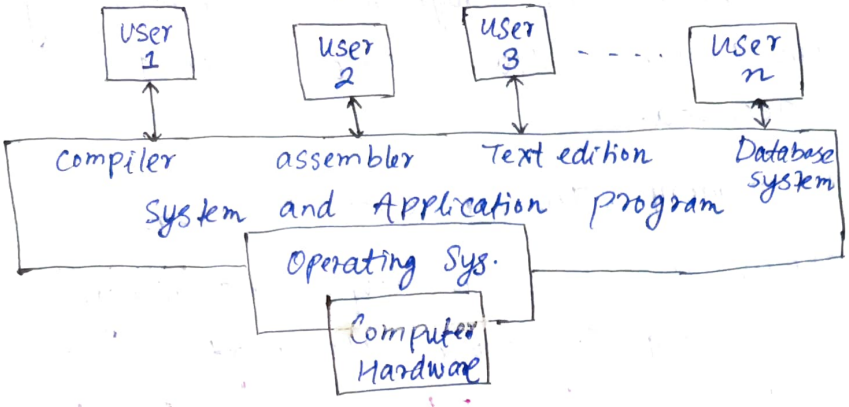
Needs of an Operating Systems

- Need to understand interaction between the Hardware and applications.
  - New applications, new Hardware.
- Need to understand basic principles in the design of computer Systems.
  - Efficient resources management, security, flexibility.
- Increasing need for specialized operating system.
  - e.g embedded O.S for devices - cell phones, sce sensors and controllers.
  - real-time O.S - aircraft control, multimedia Services
- OS offers generic services (i.e Input/output) to support all the I/O operations.

Computer System Components

- Hardware
- O.S
- Application programs
- Users

Abstract view of system



Types of O.S
- Batch
- Multiprogramed
- Multi tasking
- Real time OS
- Distributed
- clustered
- Embedded

Operating System Views

- Resource allocator
  → to allocate, resources (software and hardware) of the computer system and manage them efficiently
- Control program
  → Controls execution of user programs and operation of I/o devices.
- Kernel
  → The program that executes at all times (everything else is an application with respect to the kernel.

Batch System Issues

Punch Cards

- Spooling
- Solutions to Speed up I/o.
- Offline processing
  → load jobs into Memory, from tapes, card reading
- Spooling
  → use disk (random access device) as large storage for reading as many input files as possible and storing output files until output devices are ready to accept them.
  → Allows overlap I/o devices of one job with computation of another
  → Introduces motion of a job pool that allows OS chooses next job to run so as to increase cpu utilization.

location of O.S
↳ Hard disk/ Secondary Memory

Multiprogramming Systems

→ complete the while process then go to others.

- Use interrupts to run multiple programs simultaneously
  → when a program performs I/O execute another program till interrupt is received.
- Requires secure memory, I/O for each program
- Requires intervention if program loops indifinitely.
- Requires CPU scheduling to choose the next job to run
- ICs are used.

* Multi processing - Multiple processors works together to execute multiple Processes Simultaneously

~~→ Multiprocessor~~

interrupt → Software interrupt, Hardware interrupt

Time Sharing / Multitasking

Responsiveness

- It is an logical extension of multiprogramming
- CPU executes multiple jobs by switching among them.
- Program queued for execution in FIFO order.
- like multiprogramming, but timer device interrupts after a quantum (time slice)
  → Interrupted program is returned to end of FIFO.
  → Next program is taken from head of FIFO.
  → execute the process acc. to specified time to all.
- Interactive (action/response)
  → when OS finishes execution of one command

personal Computing System

Parallel Systems

- Multiprocessor system with more than one CPU in close communication.
- Improved Throughput, economical, increased reliability.
- Kinds:
  - → vector and pipelined
  - → Symmetric & asymmetric multiprocessing ~~multithreading~~
  - → DIStributed memory vs shared memory.
- programming Models.
  - → Tightly coupled vs loosely coupled, message based vs shared variable

ubiquitous → All time Everywhere

examples

Throughput → No. of jobs Completed within 1 unit of time

Distributed Systems [LATER]

Real time Systems

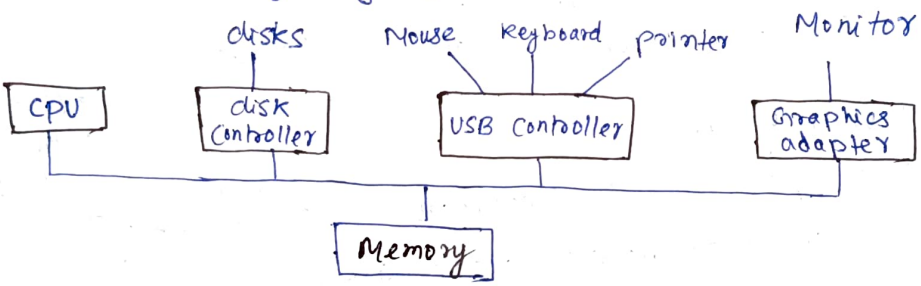
21-07-25

- Correct System function depends on timeliness.
- Sensors and actuators
- Hard real time systems
  - → Failure if response time too long.
  - → Used for dedicated application.
- Soft real time systems
  - → less accurate if response time is too long.
  - → useful in applications such as multimedia, virtual reality.

well-defined fixed time constraints

Modern Computer System

- Computer System Operation
  - → one or more cpu's device controllers connect through common bus providing access to shared memory.
  - → Concurrent execution of cpu's and devices competing for memory cycles.



unit of CPU cycle → Hertz

Computer Startup

- bootstrap program is loaded at power-up
  - → Typically stored in ROM or EPROM, generally known as firmware.
  - → Initializes all aspects of system, from CPU registers to device controller, to memory contents.
  - → loads Operating system kernel and starts execution
  - → OS starts executing first process, such as 'init' and waits for some event or occur.

Multitasking
multi ~~single~~ user
↓
windows, linux

Interrupts

- Occurence of event is usually signaled by an interrupt.
- Interrupt transfers control to the interrupt service routine.
  - → Interrupt Service routine Segments of code that determine action to be taken for each type of interrupt.
  - → Interrupt vector contains the address of service routines.
- OS preserves the state of the CPU.
  - → stores registers and the program counter (address of interrupted instruction)
- Trap
  - → software generated interrupt caused either by an error or a user request.

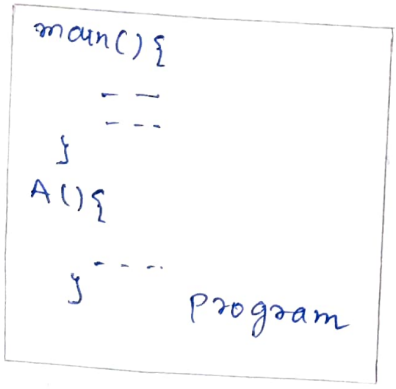
Direct Memory Access (LATER)

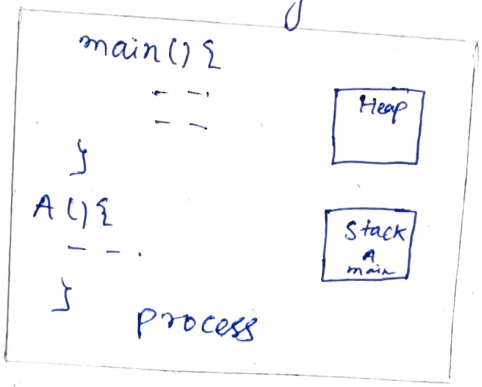
Concept of process & Scheduling Algorithm :-

Something is in execution mode is called process

* process = ? ~~process~~ program

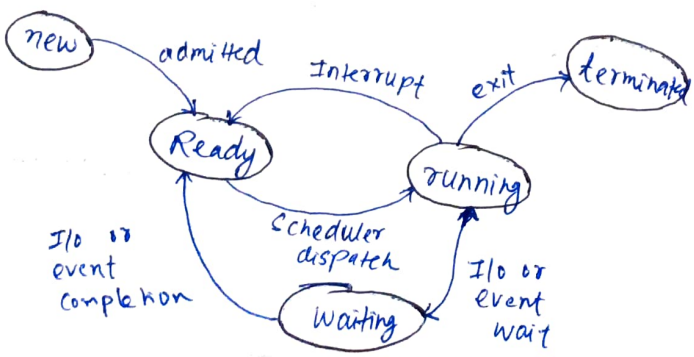
- A process is a program in execution [assesive ~~active~~ entity]
- A program is an inanimate / passive entity
  - text section
  - data section
  - stack





Process State

- A process changes state as it executes.



lifecycle of a process
→ when it is created & terminated

Process Concept

22/7/25

- An Operating System executes a variety of programs
  - → batch System - Jobs
  - → time shared system - user program or tasks
  - → Job and program used interchangebly.
- Operating System processes executing system code and user Processes executing user code
- Process is an entity

→

* Process are resided in different state during its lifetime this is called life cycle of a process.

(i) New → It is a state when it is just started. (1) OR → the process is being created.
(ii) Ready → waiting to be assigned to a processor. (4)
(iii) Running → Instruction are being executed (2)
(iv) waiting → waiting for some event to occur (3)
(v) Terminated → process has finished execution (5)

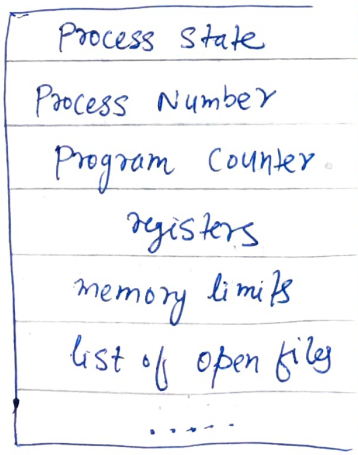
Process Control Block (PCB)

* DMA, Address Generation, (memory mapping)

- Contains information associated with each process.

→ ~~Redeque~~ Ready queue reside inside the main Memory.

- Process state- eg new, ready, running etc.
- Program number- process ID
- Program Counter- adress of next instruction to be executed.
- cpu registers - general purpose registers stack, pointer etc.
- cpu scheduling Information - process, priority, pointer.
- Memory Management Information - base/Limit information
- Accounting Information - time limits, process no
- I/o status Information - List of I/o devices allocated



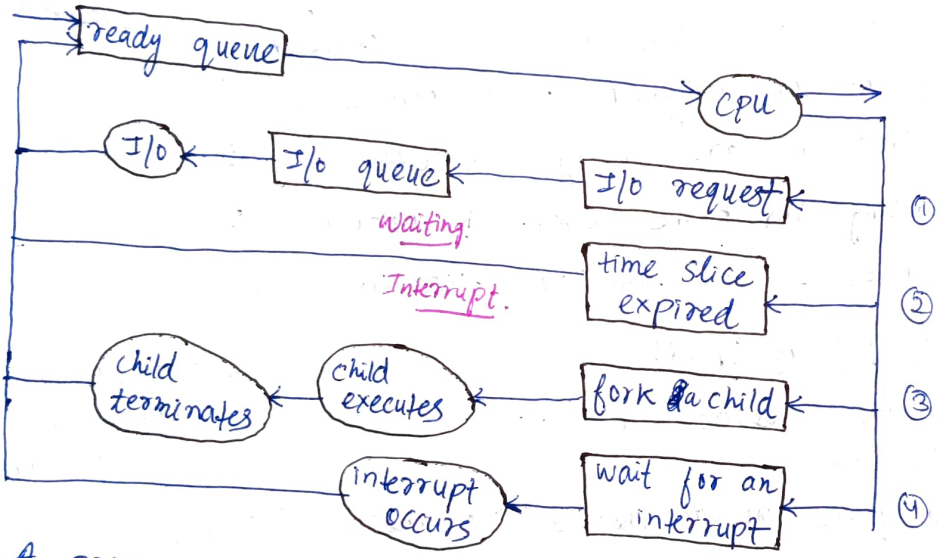
Process Control Block

Process Scheduling Queues

- Job Queue - set of all ~~in~~ processes in the system (Outside RAM)
- Ready Queue - set of all process residing in main memory, ready & waiting to execute.
- Device Queue - set of processes waiting for an I/o device
- Process migration between the various queues.
- Queue structures - typically linked list, circular list etc.

Queuing Diagram of process scheduling

Process (PCB) moves from queue to queue



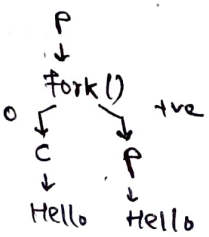
- A process is a combination of cpu operation + i/o operation
- A fixed time slice is used here. It is for time sharing system. Every process will use fixed time for execution. It has fixed time.
- Fork function will create child process. **Fork()**
- Different types of interrupts are there. Interrupts solve the problem, otherwise process is abort.

2^n → Print
2^{n-1} → child

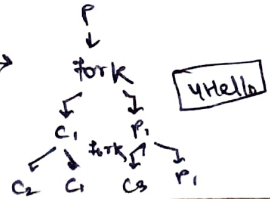
No. of fork 2^n

fork() → 0 → child
→ +1 → +ve → parent
→ -1 → child ×

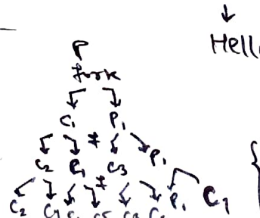
```
main(){
  fork();
  Printf("hello");
}
```




```
main(){
  fork();
  fork();
  Printf("Hello");
}
```




```
fork()
fork()
fork()
P#("H")
```



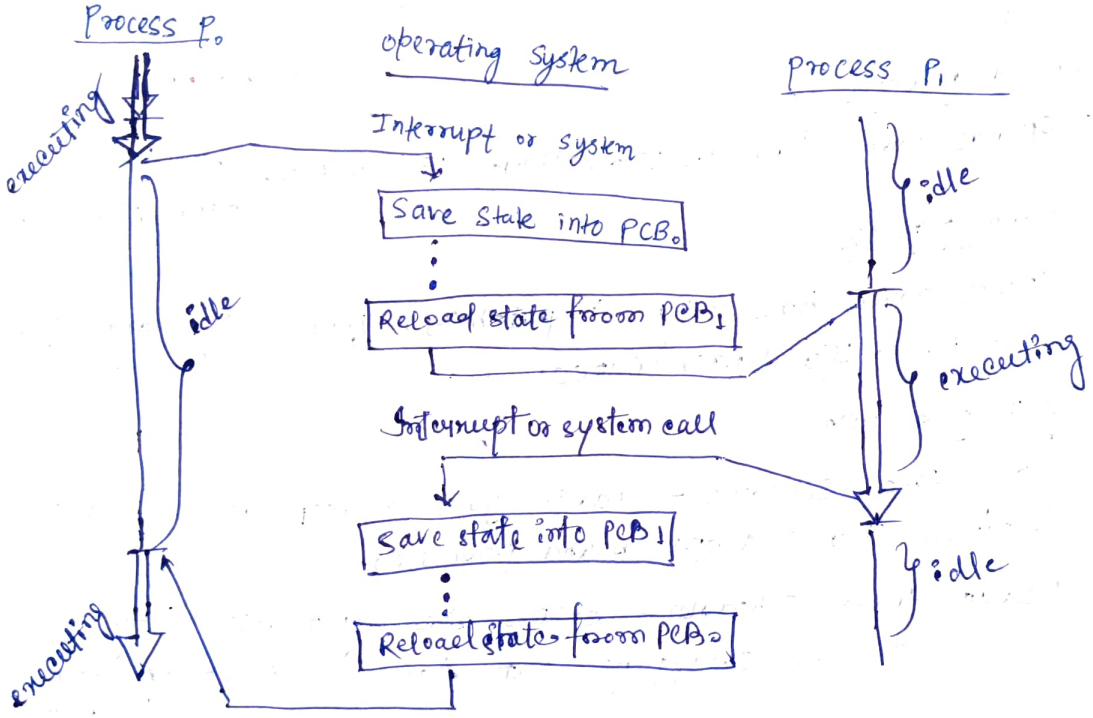
7 child
1 Parent

8 Hello

Enabling Concurrency: Context Switch

- Task that switches cpu from one process to another process.
  - → The cpu must save the PCB state of the old process and load the saved PCB state of the new process.
- context - switch time is overhead
  - → system does no useful work while switching.
- Time for context switch is dependent on hardware support (1 - 1000 microseconds)

Cpu switch From process to process



Schedulers

28/7/25

- Long-term Scheduler (for job Scheduler)
  - → selects which processes should be brought into the ready queue.
  - → invoked very infrequently (second, minutes), may be slow
  - → controls the degree of multiprogramming
  - → If the degree of multiprogramming is stable, then the avg rate of process leaving the system.
- Short time scheduler (cpu scheduler)
  - → selects which process should execute next & allocate cpu
  - → invoked very frequently (milliseconds) must be very fast.
  - → once every 100ms
- Medium term scheduler
  - → swap in swap out process temporarily
  - → balance load for better throughput.

Medium Term (Time-Sharing) Scheduler

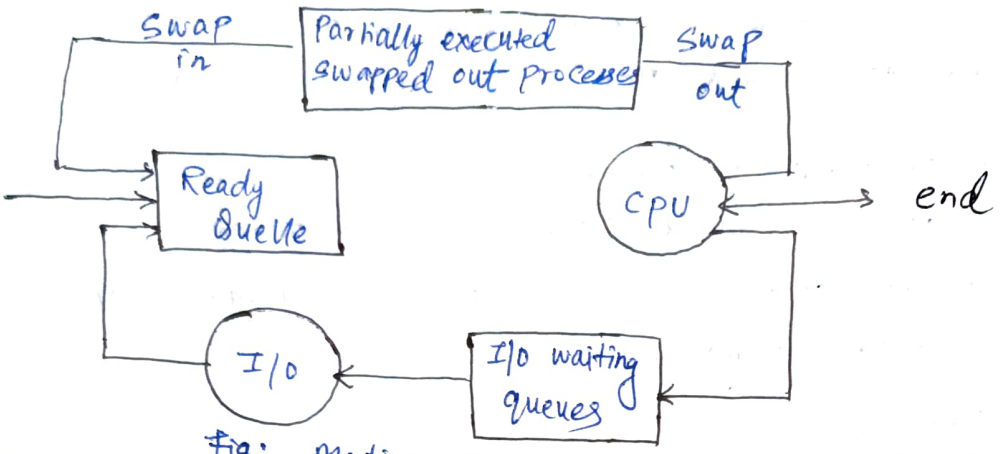


fig: Medium Term Time sharing Scheduler

Process Profiles

- I/O bound process
  - → spends more time in I/O, short CPU bursts, CPU underutilized.
- CPU bound process
  - → Spends more time doing computations, few very long CPU bursts, I/O Underutilized.
- The right job mix
  - → long time Scheduler - admits jobs to keep load balanced between I/O and cpu bound processes.
  - → Medium term Scheduler - ensures the right mix (by sometimes swapping out jobs & resuming them later)

execution time

Process Scheduling -

→ Schedulers manges the running process of the System.

CPU scheduling decisions may take place when a process

1. Switches from running to waiting state. (normal)
2. Switches from running to ready state (abnormal)
3. " " waiting to ready (abnormal)
4. Terminates (normal)

→ Scheduling Under 1 and 4 is non-preemptive [Normal situation]
All others are preemptive [abnormal situation]

Scheduling → preemptive
Scheduling → Non-Preemptive

Dispatcher

→ Dispatcher module gives control of the cpu to the process selected by the short-term scheduler this involves:
  → context switching
  → jumping to the proper location in the user program to restart that program.
  → Dispatch latency - time it takes for the dispatcher to stop one process & start another running.

Scheduling Criteria

→ cpu utilization - keep the cpu as busy as possible
→ Throughput # of processes that complete their execution per time unit.
→ Turnaround time :- amount of time to execute a particular process (Total time taken)
→ waiting time :- amount of time a process has been waiting in the ready queue.
→ Response time :- amount of time it takes from when a request was submitted until the first response is produced, not output (for time sharing environment).

29/7/25

Scheduling Algorithms

Preemptive Scheduling

- Scheduler suspends a running process
- Allows other processes to run without each process having to complete
- Must be careful with real time.
- Must be careful to avoid race condition.
- Multiuser friendly

Non-preemptive Scheduling

- Run each process of completion
- Not efficient for I/o bound application
- Easy to understand & implement.
- No race condition issues.
- Must prevent starvation.
- No way to guarantee turn around.

CPU Scheduling Algorithm

First Come First Serve (FCFS)

Example Data:

Process	Arrival time (millsec)	Service time / Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

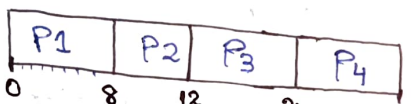
Logic

* The process which has will come first in the system will get the CPU first, that process

Arrival time
↳ Where a process enters into the system in ready queue.

Find out the average waiting time and average turn around time.

Gnatt chart



P4 P3 P2 P1
ready queue

* FCFS is a non pre-emptive type Scheduling

* n time of gnatt chart & total of Burst time must be same. if not than the Gnatt chart is incorrect. [26 = 8+4+9+5]

* Waiting time;

  P_1 = 0ms [Doesn't wait in the ready queue]

  P_2 = 8 − 1 = 7ms

  P_3 = 12 − 2 = 10 ms

  P_4 = 21 − 3 = 18 ms

* Average w.t = $\frac{P_1 + P_2 + P_3 + P_4}{4} = \frac{0+7+10+18}{4} = 8.75$ ms

* Turn around time (TAT)

  P_1 = 8 ms

  P_2 = 12 − 1 = 11 ms

  P_3 = 21 − 2 = 19 ms

  P_4 = 26 − 3 = 23 ms

Average TAT = $\frac{8+11+19+23}{4}$

= $\frac{61}{4}$ = 15.25 ms

(ii) Second Algorithm – Shortage of ~~first~~ first

SJF → Sortest Job first

* The process which has the low burst time will get the cpu first.

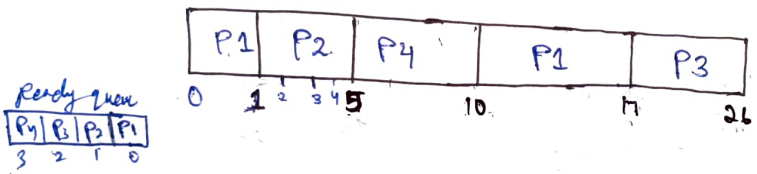
SJF are of two type
Preemptive & non – preemptive

PYQ- 2024, 2022
Preemptive

* Pre-emptive SJF

Example DATA:

Process	Arrival time (ms)	Service time (ms)
P1	0	8, 7
P2	1	4
P3	2	9
P4	3	5



Waiting time –

P1 = (10 - 1) = 9 ms [1 → (execution time + Arrival time]

P2 = 1 - 1 = 0 ms

P3 = 17 - 2 = 15 ms

P4 = 8 - 3 = 2 ms

* Dhakhalji

Av. waiting time = $\frac{9+0+15+2}{4}$ = 6.5 ms

Turn around time –

P_1 = 17 ms

P_2 = (5 - 1) = 4 ms

P_3 = (26 - 2) = 24 ms

P_4 = (10 - 3) = 7 ms

Av. TAT = $\frac{17+4+24+7}{4}$ = 13 ms

Example:- Question

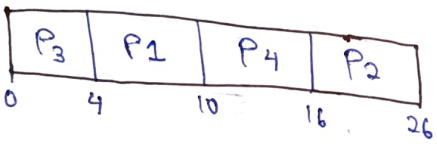
4/8/25

Suppose there are 4 set of processes that has arrived at the same time in the system in the order P_1, P_2, P_3, P_4. The CPU busst time in ms of each process given in the following table.

Find out the Average waiting time & Average Turn around time using non-premeptive SJF

Process	CPU Burst time (ms)
P_1	6
P_2	10
P_3	4
P_4	6

Ans.-



← Non-premtive SJF

Waiting time

$P_1 = 4$ ms , $P_3 = 0$ ms

$P_2 = 16$ ms $P_4 = 10$ ms

Average w.t $= \frac{4+16+0+10}{4} = \frac{30}{4} = 7.5$ ms

Turn around time

TAT of P1 = 10 ms

P2 = 26 ms

P3 = 4 ms

P4 = 16 ms

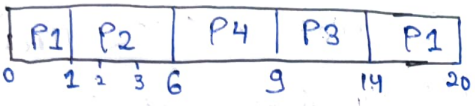
Average TAT $= \frac{10+26+4+16}{4} = \frac{56}{4} = 14$ ms

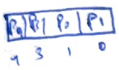
[Dhakhaldaji karna mana hai X]

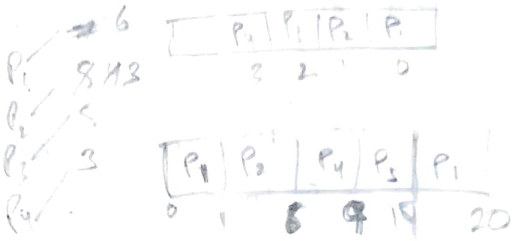
Question.

Process	Arrival time (ms)	CPU Burst time (ms)
P1	0	7 6
P2	1	5 3 2 1 0
P3	3	5
P4	4	3

* pre-emptive SJF







waiting time -

P1 = (14-1) = 13 ms [waiting time - execution time]

P2 = 0 ms

P3 = 9-3 = 6 ms

P4 = 6-4 = 2 ms

Average W.T = $\frac{13+0+6+2}{4} = \frac{21}{4} = 5.25$ ms

Turn around Time -

P1 = 20 ms

P2 = 5 ms

P3 = 14-3 = 11 ms

P4 = 9-4 = 5 ms

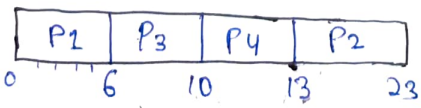
Avg Turnaround time = $\frac{20+5+11+5}{4} = \frac{41}{4} = 10.25$ ms

Ex-3

Process	Arrival time	CPU Burst time
P1	0	6 2
P2	4	10
P3	4	4
P4	8	3

Ans

* Non pre emptive SJF



* waiting time

P1 = 0 ms

P2 = 13-4 = 9 ms

P3 = 6-4 = 2 ms

P4 = 10-8 = 2 ms

A.V = $\frac{0+9+2+2}{4} = \frac{13}{4} = 3.25$ ms

= 3.75 ms

* Turn around time

P1 = 6 ms

P2 = 23-4 = 19 ms

P3 = 10-4 = 6 ms

P4 = 13-8 = 5 ms

A.V = $\frac{6+19+6+5}{4} = \frac{36}{4} = 9$ ms

Que- write Advantages & Disadvantage of FCFS, SJF [preemptive and Non-premtive]

Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process ~~what~~ with the highest priority. Equal priority process are Scheduled in FCFS order.

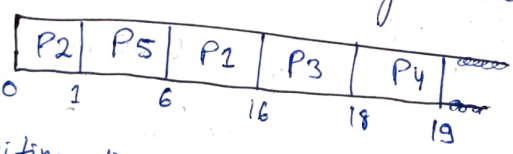
~~Priority~~ priorities are generally some fixed range of numbers such as 0 to 7. In this text we use low numbers to represent high priorities. (0 means highest) (But not fixed)

A SJF is simply a priority algorithm where the priority is the inverse of the predicted next CPU bust. larger the CPU bust lower the priority and vice versa.

Process	Burst time (ms)	priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Here priority is → 1>2>3>4>5

* Assume all the process has same arrival time
* Non-premtive priority scheduling



* waiting time
  - P_1 = 6 ms
  - P_2 = 0 ms
  - P_3 = 16 ms
  - P_4 = 18 ms
  - P_5 = 1 ms

$$Av = \frac{6+0+16+18+1}{5}$$

$$= 8.20 \text{ ms}$$

* TAT
  - P_1 = 16 ms
  - P_2 = 1 ms
  - P_3 = 18 ms
  - P_4 = 19 ms
  - P_5 = 6 ms

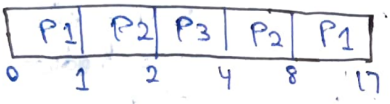
$$Av = \frac{16+1+18+19+6}{5}$$

$$= 12 \text{ ms}$$

Pre-Emptive priority

Process	Burst time (ms)	Priority	Arrival time (ms)
P1	10 9	3	0
P2	8 4	2	1
P3	2	1	2

Priority – 1>2>3



PYQ-2021

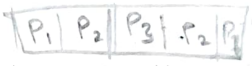
Preemptive Priority

Waiting time → $P_1 = \frac{(8-1)}{7}$ ms, $P_2 = \frac{(4-1-1)}{2}$ ms, $P_3 = 0$ ms

Av time → $\frac{7+2+0}{3} = \frac{9}{3} = 3$ ms

Turn around → $P_1 = 17$ ms, $P_2 = 7$ ms, $P_3 = 2$ ms

Av → $\frac{17+7+2}{3} = \frac{26}{3} = 8.66$ ms



1. FCFS (first come first serve)

Non-premitive Scheduling Algorithm

Advantages:
- Easy to understand & implement
- works well for tasks that arrive in Order & need similar time.

Disadvantages:
- Long tasks at the front delay short tasks (called convoy effect)
- Not good for short & important tasks.
- Average waiting time can be high.

2. SJF Non-preemptive (shortest job first)

Advantages:
- Gives minimum average waiting time
- Good for reducing the time all tasks wait.

Disadvantages:
- Hard to know how long a task will take (burst time)
- If a short job comes later, it still has to wait.
- Can cause starvation (long jobs may never get a chances if short jobs keeps coming)

3. SJF Preemptive

Advantages:
- Better for real time & quick response systems.
- Reduces waiting time for short tasks.
- More fair then non-Preemptive in many cases.

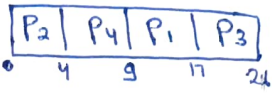
Disadvantages:
- complex to implement
- Frequent switching (context switching) may slow down Performance.
- Still suffers from Starvation for long tasks.

5/8/25

Process	Burst time (ms)
P_1	8
P_2	4
P_3	9
P_4	5

Arrival time = 0

$P_2 \rightarrow P_4 \rightarrow P_1 \rightarrow P_3$



* If the low Burst time process execute first then waiting time also reduced that's called convey effect.

* waiting time

P_1 : 9ms, P_2 : 0 ms P_3 : 17 ms P_4 : 4 ms

Av : $\frac{9+0+17+4}{4}$: $\frac{30}{4}$: 7.5

Starvation

If a low priority process is waiting for longer ~~time~~ or indefinitely blocking time for CPU is starvation. It occurs in Priority Scheduling.

Solution of Starvation is ~~edging~~ Aging

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

For e.g

We could increase the priority of a waiting process by 1 in every 15 minute. Suppose a process has initial priority of 127 would have the highest priority in the system and wanted to be executed, then it would take no more than 32 hours for a priority-127 process to Age to a priority-0 Process.

Round Robin Scheduling (RR)

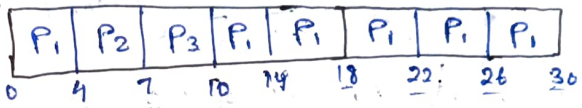
RR scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS but premption is added to enable the system to switch between processes.

A small unit of time, called a time quantum or time slice is defined.

A time quantum is generally from 10 to 100 ms in length. The Readyqueue is treated as a circular ~~cube~~ queue. The cpu scheduler goes around the Readyqueue and allocating the CPU to each process for a time interval of upto 1 time quantum.

To implement the RR scheduling, we keep the ready queue as a FIFO queue of the process. The new processes are added to the tail of the ready queue.

Process	Burst time (ms)	Arrival time = 0
P_1	24 ~~20 16 12 8 4~~ 0	Time quantum = 4 ms
P_2	3 0	
P_3	3	



Waiting time

P_1 = 10 − 4 = 6 ms

P_2 = 4 ms

P_3 = 7 ms

Av waiting time = $\frac{6+4+7}{3} = \frac{17}{3} = 5.66$ ms

Turn around time

P_1 = 30 ms

P_2 = 7 ms

P_3 = 10 ms

Av TAT = $\frac{30+7+10}{3} = \frac{47}{3} = 15.66$ ms

~~In~~ ~~softw~~ we need to consider the effect to context switching on the performance of RR scheduling. Assume that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. → It occur in context switching

If the quantum is 6 time units, However the process require 2 quanta resulting in a context switching. If the time quantum is 1 time unit, then 9 context switches will occur slowing the execution of the process accordingly.

Thus we need the time quantum to be large with respect to the context switch time

FCFS (Non-preemptive)

SJF (preemptive, Non preemptive)

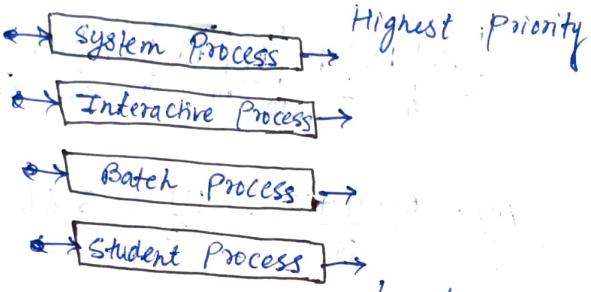
Priority (preemptive, Non preemptive)

RR

* Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm partitions the ready queue into several seperate queues.
- Each queue has its own scheduling algorithm.
- Seperate queue for foreground and background processes.
- Foreground queue is scheduled by RR algorithm and background queue is scheduled by FCFS algorithm.

* Contd.



* Multilevel Feedback queues

- Multilevel Feedback queues
  - → Arriving processes enter the highest level queue and execute with higher priority than processes in lower queues.
- long processes repeatedly descend into lower levels.
  - → Gives short processes and I/O bound processes higher priority.
  - → long processes will run when short and I/O bound processes terminate.
- Process entering a higher level queue preempt running processes in lower queue.

* Highest - Response - Ratio - Next (HRRN) Scheduling

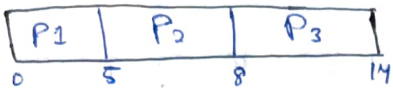
- Improves upon SPF scheduling
- It calculates dynamic ~~prop~~ priority according to the formula

$$\text{Priority} = \frac{\text{waiting-time} + \text{Service-time}}{\text{Service-time}}$$

- Considers how long process has been waiting
- Prevents indefinite postponement
- Still non-preemptive.

Example

Process	Arrival time	Burst time / service time
P_1	0	5
P_2	2	3
P_3	4	6



Priority of P_2 = $\frac{\text{waiting + service}}{\text{service}}$: $\frac{3+3}{3}$: $\frac{6}{3}$ = 2 ms

* waiting of P_2 : 5 - 2 = 3 ms
* waiting of P_3 : 5 - 4 = 1 ms

Priority of P_3 : $\frac{1+6}{6}$: $\frac{7}{6}$: 1.16 ms

Priority of -

$P_2 > P_3$ so, P_2 will get the chance first.

Waiting time

P_1 : 0 ms, P_2 : 3 ms, P_3 : 4 ms,

Av : $\frac{0+3+4}{3}$: $\frac{7}{3}$: 2.33 ms

TAT.

P_1 : 5 ms, P_2 : 6 ms P_3 : 10 ms

Av : $\frac{21}{3}$ = 7 ms

completion Time,

P_1 : 5 ms

P_2 : 8 ms

P_3 : 14 ms

* **Example: Selfish RR scheduling**

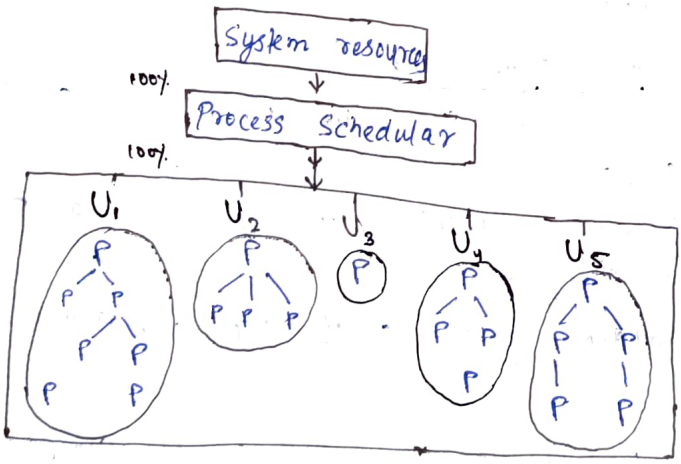
- A variant of round robin scheduling is called selfish round robin scheduling.
- After that maximum is reached, newly entering process are placed on a holding queue.
- Process in the holding queue don't get any time slice of the CPU.
- when a process in the round-robin queue completes and leaves the system, the oldest process in the holding queue is allowed to enter the round robin queue.
- In selfish RR, there is a max limit on the number of Process that can be placed in the RR queue including the process being execute by the CPU.

* Fair Share Scheduling

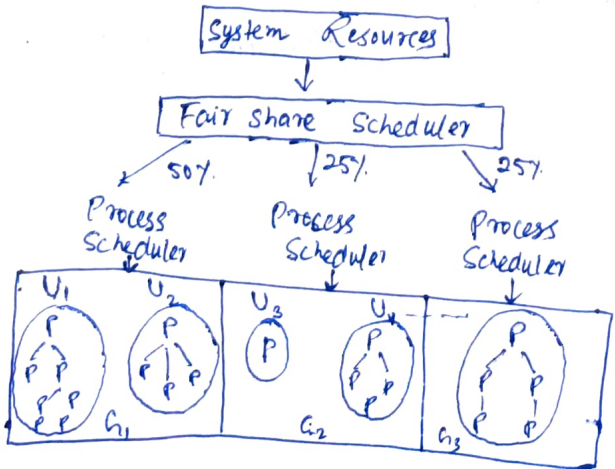
- FSS enables a system to ensure fairness across groups of processes by restricting each group to a certain subset of the system resources.
- Some user groups more important than others.
- Ensures that less important groups cannot monopolize resources
- Unused resources distributed according to the proportion of resources each group has been allocated.
- Groups not meeting resources utilization goals get higher priority.

* Fair Share Scheduling

→ **Standard UNIX process scheduler-** The scheduler grants the processor to users, each of whom may have many processes.



→ **Fair share scheduler-** The Fair Share scheduler divides system resource capacity into portions which are then allocated by Process Schedulers assigned to various



Process Synchronization

11/8/25

Producer-Consumer Problem- Suppose that we wanted to provide a solution to producer-consumer problem that fills all the buffers we can do so by having an integer count that keeps track of the number of full buffers initially count is set to 0. It is incremented by the producer. It is decremented by Consumer.

Independent
→ Cooperative process
share → variable
memory
Code
Resources
↳ CPU
↳ Printer
↳ Scanner

Producer

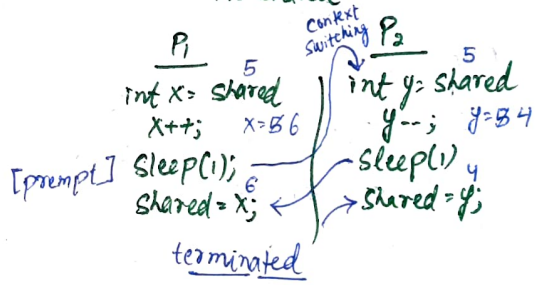
```
while(true){
/* produce an item and put in nextProduced */
while(count == BUFFER_SIZE)
; // do nothing
buffer[in] = nextProduced;
in = (in+1) % BUFFER_SIZE;
count++;
}
```

Consumer

```
while(true){
while(count==0)
; // do nothing
nextConsumed = buffer[out];
out = (out+1) % BUFFER_SIZE;
count--;
/* consume the item in nextConsumed */
```

Race Condition

int shared = 5 6 4



∴ Wrong becoz in 5, 1 is added & 1 is substracted but answer is 4 or 6

∴ Not synchronized

Data inconsistency

Race Condition

Count++ could be implemented as

I1: register 1 = count
I2: register1 = register2 + 1
I3: count = register1

count-- could be implemented as

I4: register2 = count
I5: register2 = register2 - 1
I6: count = register2

Conclusion- When a operating point shared variable & a common data then no other process can shared that.

Race Condition Contd.

Consider this execution interleaving with "count = 5" initially.

I1: producer execute register1 = count (register1 = 5)
I2: " " register1 = register1 + 1 (register1 = 6)
I3: consumer " register2 = count (register2 = 5)
I4: " " register2 = register2 - 1 (register2 = 4)
I5: producer " count = register1 (count = 6)
I6: consumer " count = register2 (count = 4)

* Race Condition

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

→ Critical Section problem :-

Each process has a segment of code, called critical section, in which the process may be changing their common variables updating a table, writing a file, and soon.

Solution to Critical Section Problem :-

1. Mutual Exclusive - If process P_i is executing in its critical section, then no other processes can be executing in the critical sections.
2. Progress :- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinetely.
3. Bounded waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

10:30 Bounded Buffer Producer Consumer Problem (P-C)

- Assume that each process executes at a non-zero speed
- No Assumption Concerning relative speed of the N processes.

* Bounded Buffer problem

- Imagine a fixed-size buffer (like a box that can hold only N items)
- A producer puts items into the buffer.
- A consumer takes items out.
- The buffer is bounded (limited capacity), so:
  - → Producer must wait if the buffer is full.
  - → Consumer must wait if the buffer is empty.
- The challenge is synchronization so that producer & consumer don't interfere with each other.

B.B.P and the P.C.P generally refer to same core synchronization problem just described in slightly different words

* Producer - Consumer Problem

- This is the general name for the situation where,
  - → One or more producers generates data.
  - → One or more Consumers processes that data.
- The data is stored in a shared resource (like a buffer, queue, fil
- The bounded buffer problem is actually ~~the~~ classic example of the producer - Consumer problem.

In short :-

- Producer-Consumer → the general problem of coordinating producers and consumers.
- Bounded Buffer → A specific form of producer-Consumer where the storage has a fixed capacity.

```
void consumer(void)
{
  int item c;
  while (true)
  {
    while (count==0);
    item c = Buffer(out);
    out = (out+1) mod n;
    count = count - 1;
    process_item(itemc);
  }
};
```

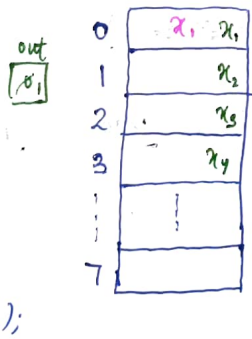
out [0]

buffer empty

like a memory or space in RAM

n=8

Buffer [0 ... n-1]




```
int count = 0;
void Producer(void)
{
  int item p;
  while (true)
  {
    Produce_item(item p);
    while (count == n);
    Buffer[in] = item p;
    in = (in+1) mod n;
    Count = Count + 1;
  }
}
```

local variable of Producer

function which will P.I

Infinite buffer full

1. load Rc, m[count]
2. DECR Rc
3. Store m[count], Rc

Rc = 3 2 pre-empt again

Case 1: x_1 ✓

(0+1) mod 8 → in

1 mod 8 = 1

(0+1) mod 8 → out

= 1

Normal / Best case

1. load Rp, m[count]
2. INCR Rp; ✓
3. Store m[count] Rp

Rp [8 4] pre-empt → Interrupt

Terminated

* and what consumer do is, it consumes the item and to consume it will execute this whole code & then finally will bring out the item from the buffer. & then it can process anything it wants

Case 2: x_1, x_2, x_3 already

x_4 ← item

(0+1) mod 8 → out

= 1

↳ Producer I_1, I_2
Consumer I_1, I_2
Producer I_3
Consumer I_3

flow

* when producers code is runs then producer produces an item & Places it in a buffer

* Buffer and Count are shared variable

* Two local variables
  - Consumer → out
  - Producer → in

Two Process Solutions

Algorithm 1: I finished with it, Now you've it.

- This applies to a set of two processes (P0, P1)
- Both are cooperating through a shared integer variable "turn".
- Its value will be either 0 or 1 at a time
- Turn == 0, P0 executes in ~~cs~~ critical section.
- Turn == 1, P1 executes in critical section

$P_0 \rightarrow CS$ turn = 1
$P_1 \rightarrow CS$ turn = 0

```
int turn = 0; / initial value of turn
    can be set to 0 or 1 /
P0: do
  { while (turn == 1); / keep looping
    as long as turn equals 1 /
    <critical section>
    turn = 1; / enables P1 to enter CS /
    <remainder section>
  }
```

Busy waiting / spin

```
P1: do
  { while (turn == 0); / keep looping as
      turn equals 0 /
    <critical section>
    turn = 0; / enables P0 to enter CS /
    <remainder section>
  }
```

* **General structure of a typical process P_i**

```
do {
    Entry section
        <critical section>
    Exit section
        <Remainder section>
} while (true);
```

turn = 1
$P_1\ P_0\ P_1\ P_0$ —
turn = 0
$P_0\ P_1\ P_0\ P_1$ —

* Each ~~request~~ process must request permission to enter its critical section. The section of code implementing this request is the entry section. ← Entry Section

* The critical section is followed by an exit section. The remaining code in the remainder section. ← Exit Section

Explanation —

- The processes (P0, P1) wishing to execute its Critical section, then it first examines the content of turn. If the value of turn is equal to the no. of process (0 or 1), ~~equal~~ then the process may proceed to its Critical section.

otherwise it is forced to wait (looping). The waiting process repeatedly check/reads the value of turn until it is allowed to enter its critical section. This procedure is known as busy or spin waiting.

After a process has gain access to its CS and after it has completed that section, it must update the value of turn to that of the other process.

This solution guarented the mutual exclusion property but has 2 drawbacks.

(i) Processes must strictly alternate in their use of their CS. Therefore, the pace of execution is ~~det~~ dectated by the slower of the two processes.

P0 → 1 hr
P1 has to wait

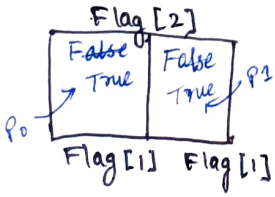
(ii) If one process fails, the other process is Permanently blocked. This is true for whether a process fails in its CS or outside of it.

Algorithm 2:-

- Algorithm-2:- let me have it, After you finish with it.
- Boolean flag[2]; //initially both set of to false

```
P0: do { flag[0] = true; // intend to enter cs
     { while (flag[1]); // keep looping as long as flag[1] is true
       <critical section>
       flag[0] = false; // enables P1 to enter cs//
       <remainder section>
       } while(1);

P1: do { flag[1] = true; //intend to enter cs
     { while (flag[0]); // keep looping as long as flag[0] is true
     <critical section>
       flag[1] = false; //enables P0 to enter cs//
         <Remainder section>
       } while(1);
```



* ~~How~~ this 2 Algorithm are known as Dekker's Algorithm

Algorithm 3:- Peterson's Algorithm

- I need it, but first you have, if you also need.

```
int turn; // Initial value does not matter
Boolean flag[2]; // Initially both set to false.
P0: do { flag[0] = true; // intend to enter CS
     turn = 1;
   { while (flag[1] && turn == 1); // keeping looping as long as flag[1] is true
   <critical section>
   flag[0] = false;
   <remainder section>
   } while (1);

P1: do {flag[1] = true; // intent to enter CS
   turn = 0;
   {while (flag[0] && turn == 0);
   // keep looping as long as flag[0] is true
   <critical section>
   flag[1] = false;
   <remainder section>
   } while (1);
```

concept of Process Scheduling
→ Intro
→ Concept of CPU scheduling

How mutual exclusion Program, bound.w happens for P.A Explanation/ Justify

→

* Concurrency — It ~~con~~ encompasses a host of design issue, including communication among processes, sharing of and competing for resources, synchronization of activities of multiple processes and allocation of processor time to processes.

Distributed system/ processing —

Multiple processes runs on multiple / distributed processor.

Peterson's Algorithm satisfies all three requirements :-

1. Mutual Exclusive - The condition in the while loop ensures that both processes can't be inside the critical section at the same time. If both want to enter, the turn variable forces one to wait.
2. progress - If the critical section is empty, the process that is not blocked by the turn variable and flag can immediately enter. The choice of who enters depends only on the processes currently interested in entering, not on any other inactive process.
3. Bounded waiting - when a process sets turn to the other process before waiting, it guarantees that the other process will get at most one turn a head, so no process waits forever.

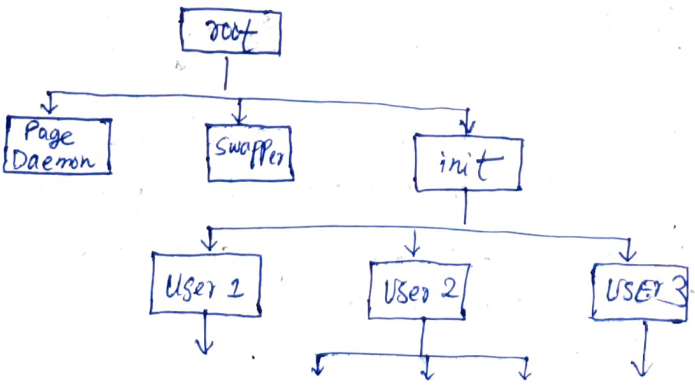
13/8/25

Process Creation

- Processes are created and deleted dynamically
- Process which creates another process is called a parent process; the created process; the created is called a child process.
- Result is a tree of processes.
  → e.g UNIX processes have dependencies and form a hierarchy.
- Resources required when creating process
  → cpu times, files, memory, I/o devices etc.

* Most of the operating systems identify processes according to a unique process identifier (PID) which is typically an integer number

UNIX process Hierarchy



Process Creation

- Resource sharing
  - → Parent and children share all resources.
  - → Children share subset of parent's resources.
- Execution
  - → Parent and child execute concurrently.
  - → Parent waits until child has terminated.
- Address space
  - → (It has the same program and data as the parent) child process is duplicate of parent process.
  - → child process has a program loaded into it.
- New process is created by the fork() system call
- The New process consist of a copy of the address space of the original process. This mechanism allows the Parent process to communicate easily with the child process.
- Both the processes continue execution at the instruction after the fork(), with one difference; the return code for fork() is 0 for the child process, whereas non-zero PID (Process Identifier) of the child process is returned to the parent.

Process Termination

- Process executes last statement and asks the operating system to delete it (exit) by using the exit; function system call.
  - → Output data from child to parent (via wait) (system call)
  - → Process's resources are de-allocated by operating system.
- Parent may terminate execution of child processes.
  - → Child has exceeded allocated resources.
  - → Task assigned to child is no longer required
  - → Parent is exiting
    - OS does not allow child to continue if parent terminates
    - cascading termination. (cheese wise)

Cooperating Processes

- Concurrent process can be
  - → Independent processes
    - * cannot affect or be affected by the execution of another process.
  - → Cooperative process / Dependent
    - * can affect or be affected by the execution of another process.

Advantages of process Cooperation:

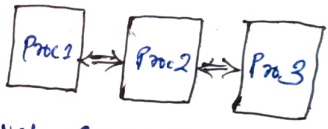
- Information Sharing
- ~~Comp~~ Computation speed up
- Modularity
- Convenience (e.g editing printing compiling)

Concurrent execution requires

- Process Communication and process Synchronization.

Interprocess Communication (IPC)

2/8/25



- Seperate address space isolates processes
  - → High creation / Memory Overhead: (Relatively) High Context Switch Overhead
- Mechanism for processes to communicate and synchronize acting.
  - → via Shared memory - Accomplished by mapping addresses to common DRAM
    - Read and write through memory.
  - → via messaging system - processes communicate without resorting to shared variables
    - send and receive messages
    - can be used over the network.

Shared memory

Interprocess IPC using shared memory requires communicating processes to stablish a region of shared memory
A shared memory region resides in the address space of the process creating the shared memory segment.
Other processes that wish to communicate using the shared memory segment must attach it to their address space.

There their are two operation that are read & write used in shared memory technique.

Message Passing technique —

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and this is particularly used in the distributed environment.

Here the two functions that is send & receive are used For example.

A chat program used on www could be designed so that chat participants communicate with one and other by exchanging messages.

* If Process P and Q wish to communicate, they need to:

→ establish a communication link between them
→ exchange messages via send / receive.

Fixed Vs Variable size message

Fixed message size - Straight forward physical implementation, programming task is difficult due to fragmentation.

Variable message size - Simpler programming more complex physical implementation.

Direct Communication

- Sender and Receiver Processes must name each other explicitly
  - send (P, message) - send a message to process P
  - receive (Q, message) - receive a message from process Q
- properties of communication link
  - links are established automatically
  - A link is associated with exactly one pair of communicating processes.
  - Exactly one link between each pair
  - link may be unidirectional usually bidirectional.

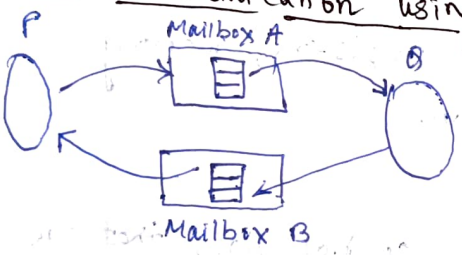
Indirect communication

- Messages are directed to and receive from mailboxes (also called Ports).
  - → Unique ID for every mailbox
  - → Processes can communicate only if they share a mailbox.

```
Send ( A, message) /* send messages to mailbox A */
Receive ( A, message) /* receive message from mailbox A */
```

- Properties of communication link
  - → link establish only if Processes share a common mailbox
  - → link can be associated with many processes.
  - → Pair of Processes may share several communication links.
  - → links may be unidirectional or bi directional.

Indirect Communication using mailboxes





Chp-1

Privileg Instructions

- Machines have two kinds of instructions
- 1. "normal" instructions, e.g, add, sub etc
- 2. "privileged" instruction e.g →
  - initiate I/O
  - switch stak vector or contexts.
  - load/save from protected memory etc.
- Operating system hide privileged instructions and provide virtual instructions to access and manipulate virtual resources e.g I/o to and from disc.. files.
- virtual instructions are system calls.
- operating systems interpret with instructions.

Processor Modes

Machine level typically has 2 ~~types~~ modes. e.g user mode and kernel mode.

User mode

Processor executes normal instruction in the user program upon encountering a privileged instruction, Processor switches to kernel mode, and the operating system performs a service.

Kernel mode

Processor executes both normal & privileged instructions

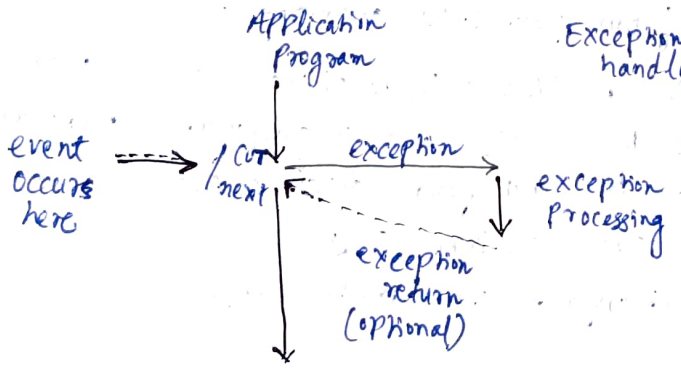
System calls

- Virtual instructions are often presented as a set of system calls
- Typical implementations (in order of prevalence)
  - → single privileged instruction with parameters.
  - → interpret to other privileged instructions
  - → Jump to fixed locations.
- system calls return result in registers, memory etc., and an error indication.

Exceptions

Exceptions are a form of exceptional control flow that are implemented partly by the hardware & partly by the operating system. Because they are partly implemented in hardware, the details vary from system to system

An exception is an abrupt change in the control flow in response some change in the processor's state



Anatomy of a exception - A change in the processor's state (event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing the handler returns control to the interrupted program or aborts.

- The change in state is known as an event. The event might be directly related to the execution of the current instruction
- For example, a virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero.
- On the other hand, the event might be unrelated to the execution of the current instruction. For example a system timer goes off or an I/o request completes.

8/9/25

* DEADLOCK

↳ Stuck in any situation.

In multiprogramming environment several processes are competing with each other & every process needs recources to complete its task. But it sometimes ~~haven~~ may happen that resources are not available & processes may enter into waiting condition. The waiting processes never again change their state. and they are waiting for some resources because they are waiting for the resources which are held by some other process & that process are also waiting for some other resources. If such type of scenario is occured then we can say that it is in deadlock state.

System Model

- System consists of a collection of m resources types & n processes.
  - → Resources types $R_1, R_2 \cdots R_m$
    CPU cycles, memory spaces, I/o devices.
- Each resource type R_i has W_i instances.
- Each process utilizes a resources by executing a sequence of three actions.
  - → request it
  - → use it
  - → release it

Deadlock characterization

Deadlock can arise it four conditions hold simultaneously.

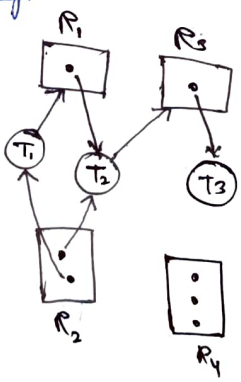
- Mutual Exclusion : only one process at a time can use a resource. ↳ Non-sharable mode
- Hold & wait : a process holding at least one resources is waiting to acquire additional resources held by other processes.
- No preemption : a resource can released only voluntarily by the process holding it. after that process has completed its task.
- circular wait : there exist a set $\{P_0, P_1, \ldots P_n\}$ of waiting process such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held $P_n \ldots P_{n-1}$ is waiting for a resource that is hold by P_n and P_n is waiting for a resources that is held by P_0.

Introduction
Process syn x
Thread x
CPU scheduling

Resource Allocation Graph

circle → process
□ → resources
• → instance of the resources

Fig 1



Deadlock Problem

A set of blocked processes each holding a resource & waiting to acquire a resource held by another process in the set.

Example-

→ System has 2 disk drivers
→ P_1 & P_2 each hold one disk drive and each needs another one.

Example

→ Semaphores A and B is initiated to 1.

P_0	P_1
wait(A);	wait(B)
wait(B)	wait(A)

Resource Allocation Graph

Process



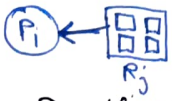
Resource Type with 4 instances



P_i requests instance of R_j

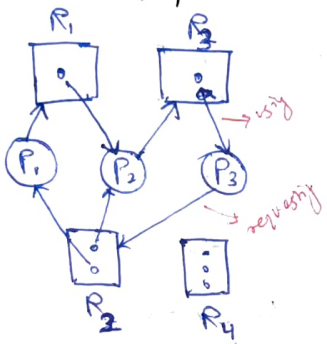


P_i is holding an instance of R_j



Resource allocation Graph with a Deadlock

Fig 2



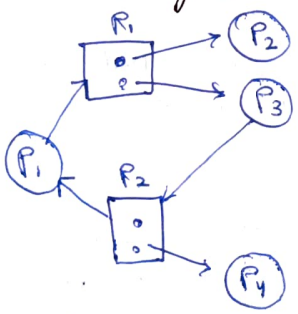
Now, suppose that P_3 process request and instance of resource type R_2 in the previous diagram Fig 1 we see that there is no request from process P_3 to resource of R_2

At this point to minimal cycle exist in the system.

(i) $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

(ii) $P_2 \rightarrow P_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Graph with a cycle But no deadlock



In this RAG there is no deadlock still we have a cycle.

(i) $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

Basic facts—

If Graph contains no cycles ⇒ no deadlock

If Graph contains a cycle ⇒

→ if only one instance per resource type, then deadlock.

→ If several instances per resource type → Possibility of deadlock.

Methods for handling deadlocks (Prevention & Avoidance)

- Ensure that system will never enter a deadlock state.
- Allow the system to enter a deadlock state & then recover.
- Ignore the problem & pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention:

- Mutual Exclusion - not required for sharable resources; must hold for non-sharable resources.
- Hold and wait - must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest & most useful model requires that each process declare the maximum no of resources of each type that it may need.
- The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular wait condition
- Resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
- A system is in a safe state if there exists a sequence ^{of} ^{all} the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j < i

- That is
  - → If P_i resources needs are not immediately available then P_i can wait until all P_j have finished.
  - → When P_j is finished. P_i can obtain needed resources, execute, return allocated resources and terminate.
  - → When P_i terminates P_{i+1} can obtain its needed Resources and so on.

Basic Facts

→ If a system is in safe state ⇒ no deadlocks

→ If a system is in unsafe state ⇒ possibility of deadlock.

→ If Avoidance ⇒ ensure that a system will never enter an unsafe state.



Avoidance Algorithm

→ Single instance of a resource type. Use a resource allocation graph.

→ Multiple instance → Banker's Algorithms

Bankers Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time.

Data structures for the Bankers Algorithm

let n = number of processes, and m = number of resources types.

→ Available: vector of length m. If available [j] = k, there are k instances of resource type R_j available.

→ Max: n x m matrix. If Max [i,j] = k then process P_i may request at most k instances of resource type R_j.

→ Allocation: n x m matrix. If Allocation [i,j] = k then P_i is currently allocated k instances of R_j.

→ Need: n x m matrix. If Need [i,j] = k, then P_i may need k more instances of R_j to complete its task.

Need [i,j] = Max [i,j] − Allocation [i,j].

Bankers Algo
→ Safety algo
→ resource allo algo respectively

Safety Algorithm

1. let Work and Finish be vectors of length m and n
   Initialize:
   Work = Available
   Finish[i] = False for i = 0, 1, ... n-1

2. Find an i such that both:
   (a) Finish[i] = False
   (b) Need$_i$ ≤ Work / Available
   If no such exists, go to step 4

   i → process
   j → resource

3. Work = Work + Allocation$_i$
   Finish[i] = true
   go to step 2.

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

* Resource - Request Algorithm for Process P_i: 15/9/25

Examples of Banker's Algorithm.

→ 5 processes P_0 through P_4
3 resources types
A (10 instances) B (5 instances) C (7 instances)

Snapshot at Time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	(Total system Av available)
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example

The content of the multiply need is defined to be Max - allocation

	Need A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

• The system is in a safe state since the sequence P_0, P_1, P_2, P_3, P_4

Example: P_1 Request (1,0,2)

- check that Request ≤ Available (that is (1,0,2) ≤ (3,3,2) ⇒ True

Working → ① In resource request algorithm we have to check Request ≤ need$_i$. In this case it is true i.e, (1,0,2) ≤ (1,2,2)

② Now we have to check Request$_i$ ≤ Available$_i$ i.e, (1,0,2) ≤ (3,3,2) ⇒ True

③ Now available = available − Request

∴ (3,3,2) − (1,0,2) = (2,3,0)

④ → Allocation$_i$ = Allocation$_i$ + Request$_i$

⑤ → need = Need$_i$ − Request$_i$

(2,0,0) + (1,0,2) = (3,0,2)

(1,2,2) − (1,0,2) = (0,2,0)

	Allocation ABC	Need ABC	Available ABC	New Matrix
P_0	010	743	230	
	302	020		
	301	600		
	211	011		
	002	431		

Que- Can the request (3,3,0) by P_4 be granted??

Shell scripting Lab

3/11/25

(i) Shell program is a collection of a series of command for a UNIX shell such as ~~bourne~~ Bourne shell and C shell.

(ii) No seperate compiler is required to execute these shell script as the shell itself interprets and executes them.

(iii) The shell scripts includes both UNIX command as well as built in functions in the shell

→ factor 15
15: 3 5

→ x=3
y=5
z=`expr $x + $y`
echo $z

→ let x= 3+7
echo $x

{ echo "welcome"
welcome

gedit t2.sh
./t2.sh
ls -l → Permissions
chmod 755 t2.sh
./t2.sh

Memory Management

Base Register
Limit Register } function

How RAM communicated with the processor
What are the different types of addresses → logical, → Physical
Address Binding —

Base and Limit Register

Def^n - Base and limit Registers are used in memory management to protect processes and control their access to memory.

Function

1. Base Register:
   - Holds the starting (lowest) address of the process in memory.
   - It tells where the process's memory begins.
2. Limit Register:
   - Holds the size (range or length) of the process's memory.
   - It tells how much memory the process can use.

Purpose

- To prevent a process from accessing memory that doesn't belong to it
- Provides memory protection and isolation between different processes.

How it works

when a program tries to access memory
- The cpu adds the address to the Base Register value.
- Then it check if the address is less than the limit register value
- If it is within range → Access allowed
- If not → Trap(error) → "Memory protection fault"

Eg	Register	value	Meaning
	Base Register	3000	process starts at address 3000
	Limit Register	1200	Process Size = 1200 bytes

So, valid addresses = 3000 to (3000 + 1200 - 1) = 4199

- If a process tries to access address 4100 → Allowed
- If it tries address between 4300 → Error! (outside limit)

RAM communicated with processor

RAM = Random Access Memory

It's the temporary storage where data and programs are kept while the CPU is using them.

When Power goes off → data in RAM is lost.

1 CPU Communication b/w RAM and CPU [Example you open a game → cpu needs the game's data to run it]

1. CPU needs data or instructions [cpu say's: Hey RAM give me data stored at this memory address]
2. CPU sends a request to RAM
3. RAM sends the data to CPU [RAM quickly finds the data & sends it back to the CPU through the system bus]
4. CPU uses the data [cpu performs calculations, run programs or display output on the screen]
5. If new data is produced [CPU may send it back to RAM to store temporarily until needed again]

3 Buses help in Communication

Address Bus → CPU tells RAM where to find or store data.

Data Bus → Actual data travels CPU ↕ RAM

Control Bus → Sends control signals (like 'read' or 'write')

Types of Adresses

When a program runs, the CPU and memory use two types of addresses to locate data:

→ Logical Address (Virtual Address)

- The address generated by the CPU while a program is running
- It is also called a virtual address.
- The program thinks it is using these addresses.

e.g You write a program & store a variable at address 100. That 100 is a logical address (used by the program)

* logical address doesnot show the Real location in RAM - It's like temporary or imaginary address used by the CPU.

→ physical address (Actual)

- The actual address of the data in the Main Memory (RAM).
- It is where the data is really stored in hardware.

e.g That same variable (logical variable 100) may actually be stored at physical address 5000 in RAM.

* To Convert logical → physical the system uses a special unit called the MMU (Memory Management Unit)

Address Binding

It means connecting (or mapping) a program's logical address to physical addresses in Memory.

OR

It is the process of deciding where in Memory a program's instruction & data will go.

eg when you write a program, you don't know where it will be stored in RAM

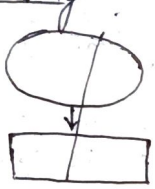
So, the OS decides & binds (links) you program's logical address → actual physical address

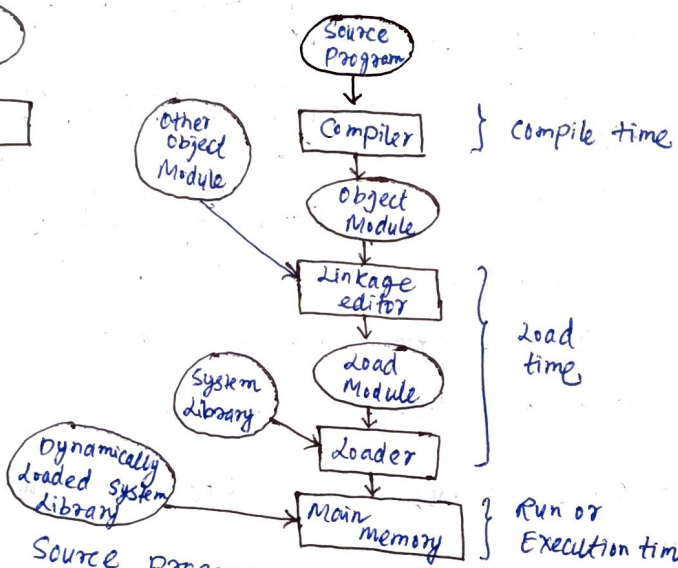
Types

1. Compile time binding – when the program is compiled
2. load time binding – when the program is loaded into memory
3. Execution time binding – while the program is running.

Address Binding

4/11/25





Addresses in the source program are generally symbolic. A compiler will bind these address to rellocatable addresses. The linkage editor or loader will bind these addresses to absolute addresses. Each binding is a mapping from one address space to another.

(i) Compile time Binding –

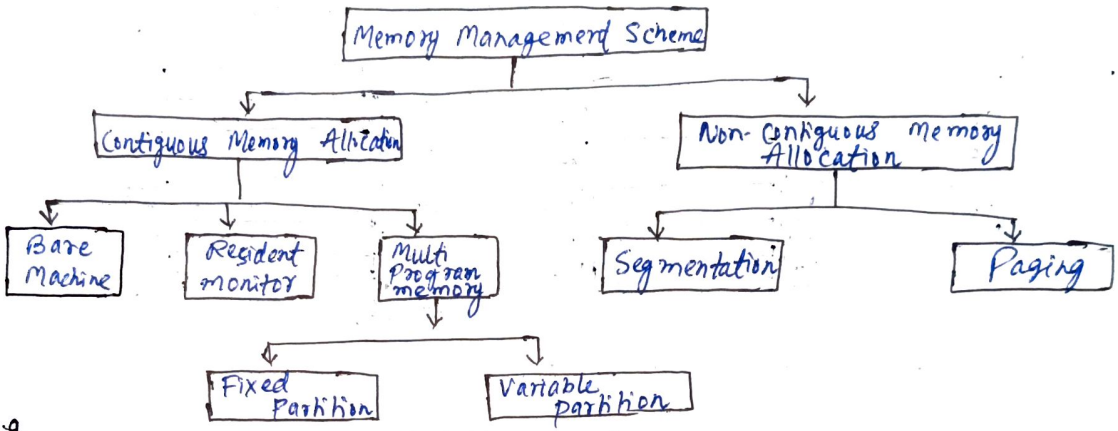
(ii) Binding on compile time generates absolute addresses where a prior knowledge is required that where a process resides in the memory. After some time if the starting location of a process in memory is changed, then the entire process must be re compiled to generate the absolute address again.

(iii) Load time Binding –

At compile time if it is not known that where a process will recide in memory then the compiler must generate rellocatable address. In this case final binding is delayed until load time. If the starting address is changed then we need only to reload then user code to incorporate changed

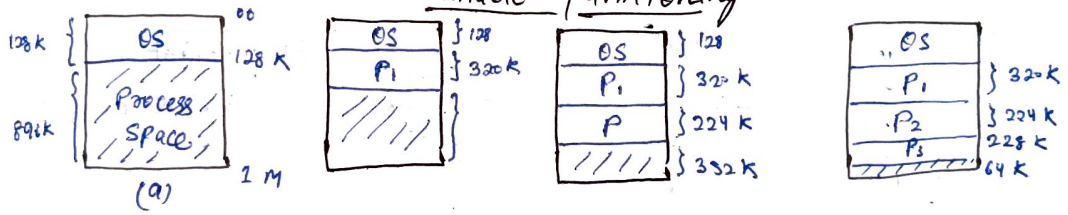
(iv) Execution time—
This method permits moving a process from one memory segment to another during run time. In this case final binding is delayed until run time. Special hardware is needed for this scheme or method to work. most general purpose system OS is used.

Memory Management Scheme



e.g

Variable Partitioning



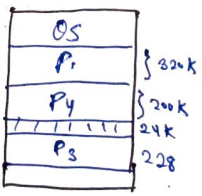
1024-128 = 896

P1 = 320k

created in Fixed partition

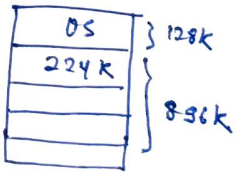
Internal Fragmentation
External Fragmentation
Difference

created in varible partitioning



external fragmentation → compacting

* Fixed partition



* There are three techniques (algorithm) that are most used in variable Partioning to allocate memory efficiently.

(i) First fit :- In first fit we allocate the first hole that is big enough. Searching can be started either at the begning of the set of wholes or where the previous first fit search ended. we stop searching as we obtain a free hole that is large enough to accomodate the comming process.

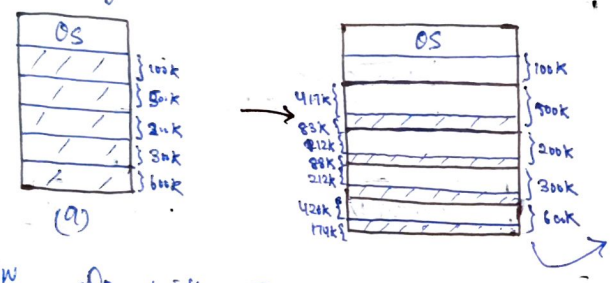
(ii) Best fit :- In best fit we allocate the smallest hole that is large enough. we must search for the entire list until it is sorted by Size. This strategy produces list size of unused memory space.

unused memory Space → fragmentation

(iii) **Worst fit** :- In worst fit we allocate the largest hole again we must search for the entire list unless it is sorted by size this strategy produces unused memory space.

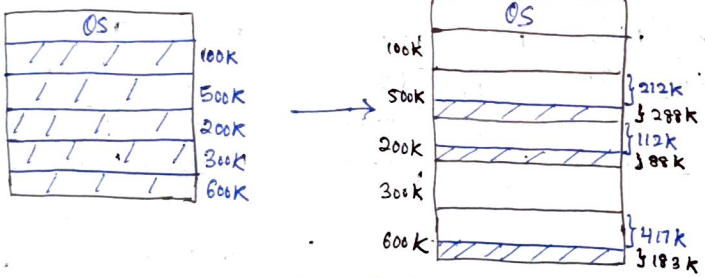
Example

For the partition of 100k, 500k, 200k, 300k and 600k in Order, Place the process of size 212k, 417k, 112k and 426k in Order according to the Best fit Algorithm & show the unused memory spaces.



H/W Do with First fit & worst fit

First fit

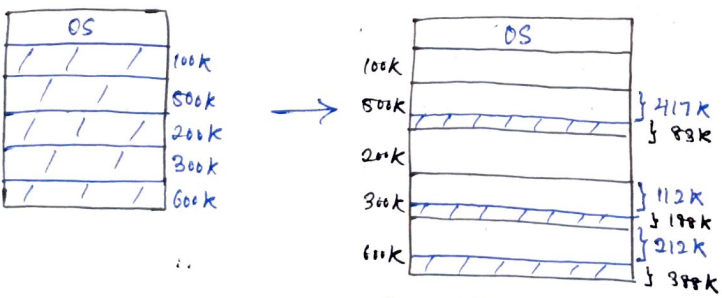


∴ 426k will wait becoz the memory doesnt have that much of space

fragmentation = 288k, 88k, 183k

PYQ-2024 Best, worst

worst fit



∴ 426k has to wait becoz the memory does not have that much of space

fragmentation = 83k, 188k, 388k

Non- Contiguous Memory Allocation

10/11/25

Paging - Paging is a memory management scheme that removes the requirement of contiguous allocation of memory.

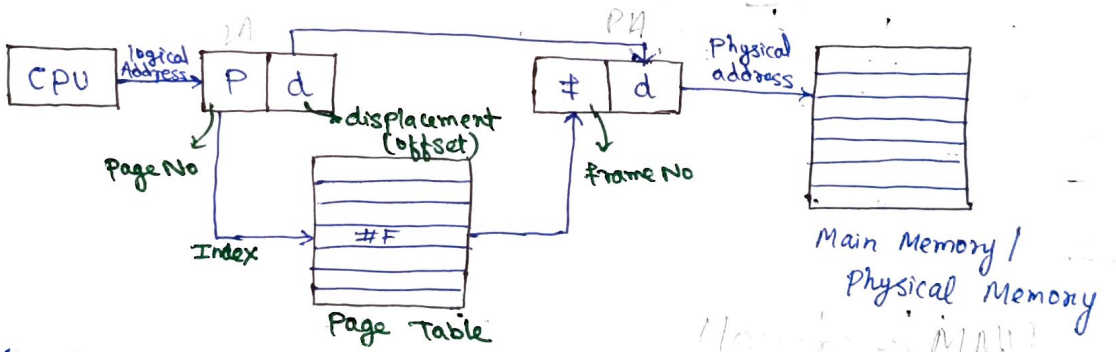
This scheme permits physical address space of a process to be non- contiguous.

On paging technique the physical memory is conceptually divided into a number of fixed- size blocks called **Frames** and logical address space (or virtual address space) is also splitted into fixed size blocks called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

The backing store is also divided into fixed size block that are of the same size of the frames. i.e the size of the frame is same as the size of a page for a particular hardware.

Principle of Paging



(i) Initially the logical address is divided into 2 parts
(a) Page no (p) (b) Displacement or offset (d) within that page.

In the page table to find the corresponding frame No. If the page is available in the Page Table then using that frame number with the offset (d). to find out the actual physical address of that page in the memory by the help of the Memory management Unit (MMU).

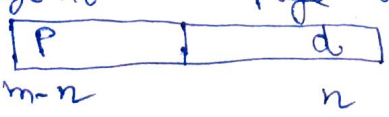
Basic Background Concept

(i) If the capacity of physical memory is m and the size of each page is p then the number of frames in the physical memory will be $f = \frac{m}{p}$.

Both m and p are usually an integer power of 2.

This resulting in f being also an integer.

(ii) The logical address is defined as



If the size of logical address space is 2^m and page size is 2^n units then the ~~higher~~ high order $m-n$ units of logical address designates the page no, and the n low order units designate the page offset.

Que- Why are the Page sizes always Power of 2?

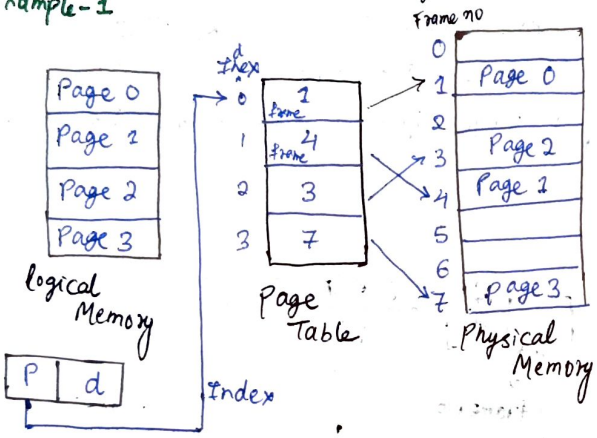
Ans Pages are of Power of 2 size because memory address are binary. Power of 2 sizes make it easy to divide the address into Page number & offset using bits. It's faster for the CPU. Since no extra calculation is needed.

Continuation---

11/11/25

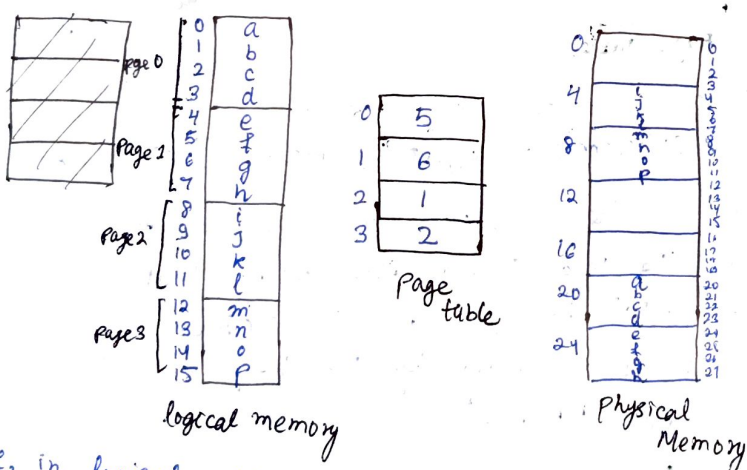
Paging Model of logical & Physical Memory

Example-1



#. Page Size = 4 bytes
Physical Memory = 32 bytes
Page No = $\frac{32}{4} = 8$

Example 2-



Here, in logical address $n=2$ & $n=4$ & Page size = 4 bytes & Physical memory = 32 bytes. for logical address page no = 0 & offset = 0. Indexing into the page table we find that Page 0 is in frame 5. Thus logical address 0 maps to Physical address is 20 (frame no = 5 × Page size = 4 + offset 0)

~~logical~~ page 0, logical address 3 [00011] maps to Physical address is 23 ($5 \times 4 + 3$).

Page 1, logical address 4 [100] maps to physical address is 24

~~5~~ $(6 \times 4 + 0) = 24$.

page 2, logical address is 8, 1000 → offset 0, physical address is 4 (1×4+0)=4.

page 3, logical address is 12, offset → 1100, physical address is 8 (2×4+0)=8.

Hardware Support -

Hardware implementation of page table can be done in several ways the simplest way, the page table is implemented as a set of dedicated registers. These registers should be built with very high speed logic to make the paging address translation efficient.

This approach is consumed very large time to access a user memory location. If we want to access location i, we must first index into the page table & this task requires memory access. It provides us with the frame no which is combined with the page address to produce actual / physical address. we can access the desire place in memory.

The standard solution to this problem is to use a special, small, fast lookup hardware cache, called a translation look-aside buffer (TLB). The TLB is associative with high speed memory Each entry in TLB consist of two parts

(i) key (ii) value.

* Paging with TLB

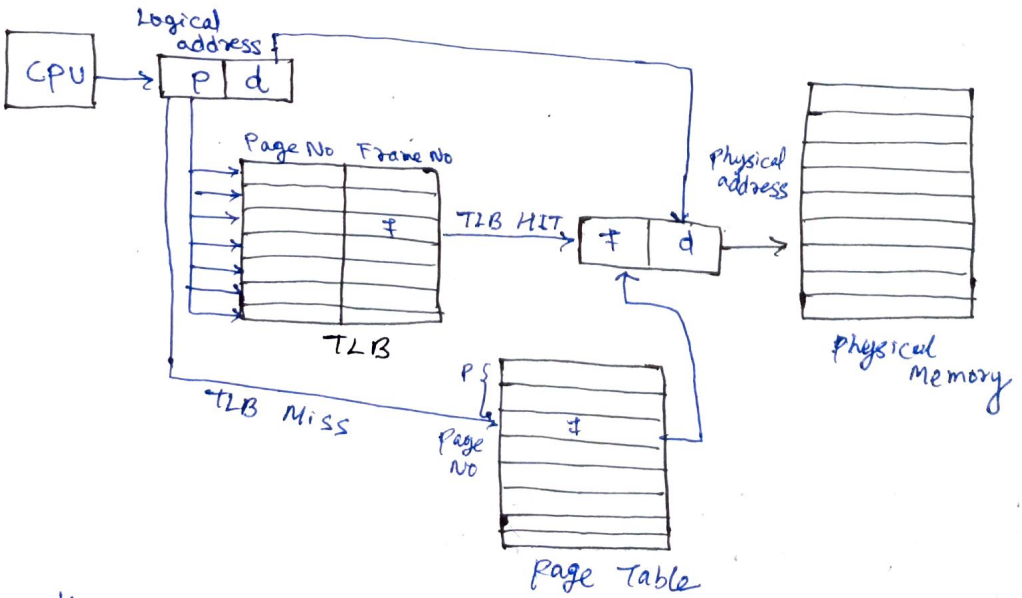


Fig # paging with TLB

Que 1. Consider a logical address space of 8 pages of 1024 units each mapped to a physical memory of 32 frames then

(i) How many bits are in the logical address?

(ii) " " " " " " physical " ?

Let no. of bits in physical memory is m

Therefore size of physical memory is 2^m

Given. No. of pages = $8 = 2^3$

No. of frames = $32 = 2^5$

∴ frame size = page size

$= 1024 = 2^{10}$.

$$f = \frac{m}{p}$$

$$2^5 = \frac{2^m}{2^{10}}$$

or, $2^m = 2^5 \times 2^{10}$

or, $2^m = 2^{15}$

$m = 15$

So, no. of bits in physical address is 15.

Let no. of bits in logical address is n

Therefore size of logical memory = 2^n

Given. No. of pages = $8 = 2^3$

$$f = \frac{m}{p}$$

$$2^3 = \frac{2^n}{2^{10}}$$

$2^n = 2^3 \times 2^{10}$

$2^n = 2^{13}$ ∴ So, no. of bits in physical address = 13.

$n = 13$

Que 2. On a simple paging system with 2^{24} bytes of physical memory have, 256 pages of logical address space and a page size of 2^{10} bytes. How many enteries are in the page table? How many bits in the phy. add specify the frame?

Que 3. How many pages will be required to convert a logical address of 18 bit into physical address during mapping if the size of each page is 1K. ($1024 = 2^{10}$)

Paging with TLB

If a computer system takes t_1 time to access TLB and t_2 time to access memory then mapped memory access will take (t_1+t_2) time when page no is present in TLB.

and when the page no is not present in TLB then we require t_1 time to access TLB and t_1' time to access the page no in page table & then T_2 time to access memory. So, we require total $t_1 + t_1' + t_2$ time when the page is not present in TLB.

Therefore the effective address time :-

T_{eff} = hit ratio * $(t_1 + t_2)$ + miss ratio * $(t_1 + t_1' + t_2)$

[$t_1' = t_2$ for a particular computer hardware]

Prob 2. Consider a Paging System with the page table in memory—

(i) If a memory takes 200 ns, how long does a ~~mapped~~ paged memory reference take.

(ii) If we need TLB's and 75% of all page table refrences are found in TLB's what is the effective refrence time? [Assume that finding a page table entry takes 0 time if the entry is there]

Ans (ii) hit ratio = $\frac{75}{100} = 0.75$

miss ratio = 0.25

$t_1 = 0$, $t_1' = t_2 = 200$ ns

T_{eff} = hit ratio * $(t_1 + t_2)$ + miss ratio * $(t_1 + t_1' + t_2)$

= 0.75 * (0 + 200) + 0.25 * (0 + 200 + 200)

= 0.75 * 200 + 0.25 * 400

150 + 100 = 250 ns

(i) Given, $t_{mm} = 200$ ns

T_{eff} without TLB = $2 t_{mm}$

= 2 * 200

= 400

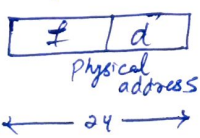
Que 2. Size of physical memory = 2^{24} bytes, ∴ physical address bit = 24

Pages = 256

Page size = 2^{10} byte So, $d = 10$ (offset) $d = 10$

(i) No. of enteries = no. of pages = 256

(ii)



f → frame no. bits

f = physical address bit - offset

= 24 - 10 = 14

Que 3. Logical address = 18, ∴ logical address space = 2^{18}

Page size = 1 KB = 2^{10} bytes

No. of pages = $\frac{\text{L.A.S}}{\text{Page size}} = \frac{2^{18}}{2^{10}} = 2^8 = 256$

17/11/25

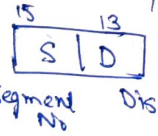
Page memory reference will take the amount of time to search a page number in the page table which is same as the amount of time of memory access bcz page table is present in memory. So the page memory will take 200ns.

Segmentation-

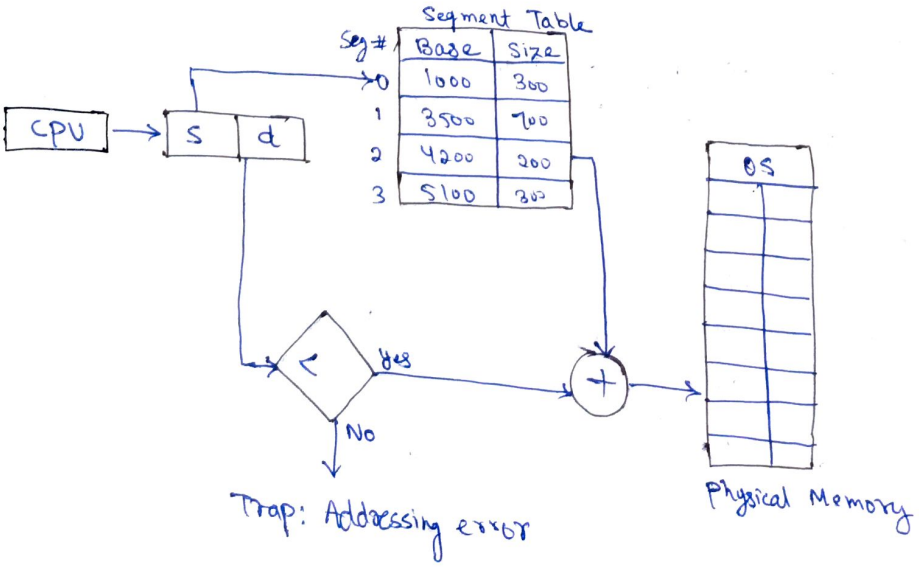
Segmentation is another technique for non-contiguous memory location. It is different from paging as pages are physical in nature & hence are fixed size, whereas segments are logical division of program & hence are of variable size.

In segmentation we divide the logical address space into different segments. The general division can be main program, set of subroutines, procedures, functions & set of data structure (stack array etc) each segment has a name & length which is loaded into physical memory as it is—

The logical address consists of two types:—
<segment-name offset> Hence address is considered as two dimensional each segment consist of linear sequence (0,1,2,3—) of address starting from 0 to maximum value depending upon the size of segment in general, segment name is replaced with segment no. Hence the logical addresses is divided into two part segment no & displacement.



Hardware Support in Segmentation—



Virtual Memory

Virtual Memory can be implemented as
- Demand paging
- Demand Segmentation

What is virtual Memory?

Advantages & Disadvantage of virtual Memory

In demand Paging, it is not required to bring an entered process into memory at load time and bring the process into memory only when it is needed. Therefore less I/o operation is needed, no unnecessary I/o is required, less memory is needed, faster response & more users can be participated.

Valid-Invalid Bit

A valid-Invalid Bit is associated with each page table entry.
(V in memory - memory resident; I = not in Memory)
Initially, valid-Invalid bit is set to i in all enteries.

Page fault

The reference (address) to that Invalid page will trap (interrupt) to OS and causes a page fault.

Procedure for handling a page fault —

(i) OS checks an internal table to see if refrence is valid or invalid memory access.
(ii) If
Invalid reference ⇒ abort the process.
Just not in memory ⇒ page in the referred Page from the disk
(iii) Find a Free Frame
(iv) ~~Reload the~~ Read the referred page into this allocated frame via Scheduled disk operation
(v) Update both internal table & page-table by setting validation bit = V.
(vi) Restart the instruction that caused the page fault & resume process execution.

Page Replacement

18/11/25

Page Replacement takes the following approach —
(i) If no frame is free, find one that is not currently being used by other process and free it.

Find the free frame

(i) find the location of the desired page on this → (victim page)
(ii) find the free frame
  (a) If free frame, use it
  (b) ~~use~~ If no free frame, use page replacement algorithm to select a victim frame
  (c) write the victim page to the disk, change the page & frame table accordingly

(iii) Read the new desired page in the newly free frame change the page & frame table

(iv) Restart the process.

Page Replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, and enerm us virtual memory can be provided for programmers on a smaller physical memory.

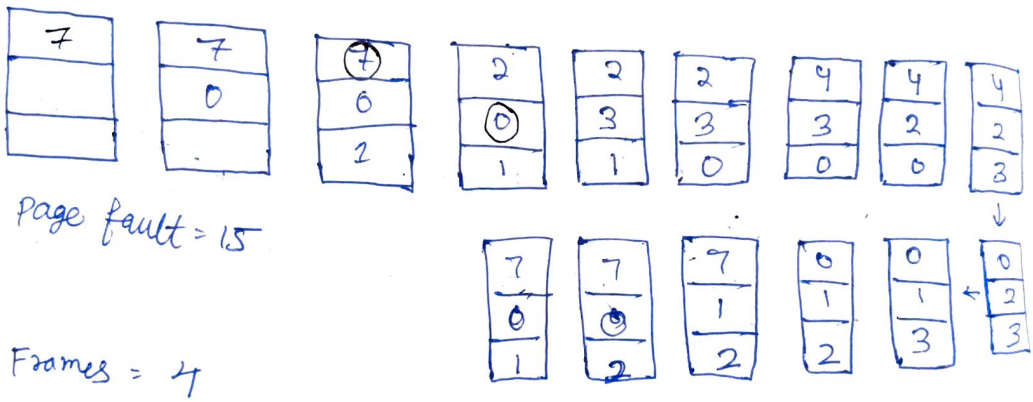
Page Replacement Algorithm

(i) FIFO Page Replacement

Reference string is nothing but page number.

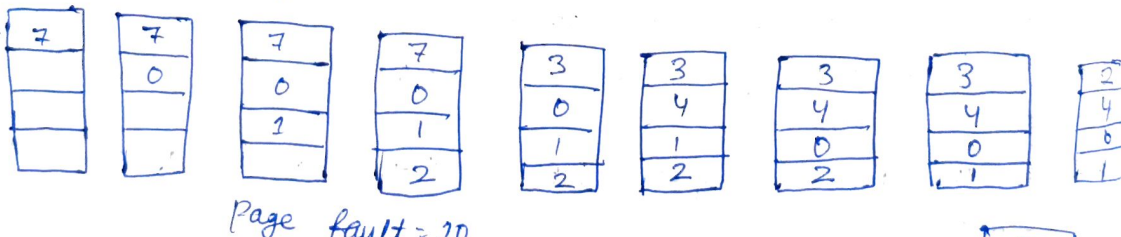
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

No. of Frames = 3



* Page fault = 15

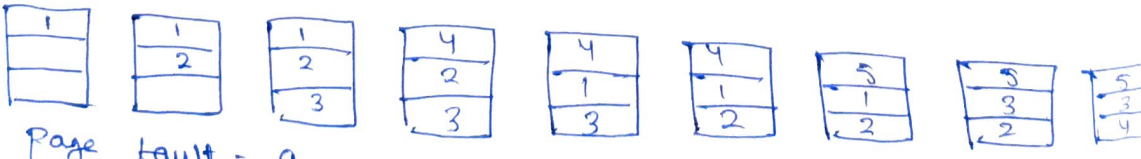
No. of Frames = 4



Page fault = 10

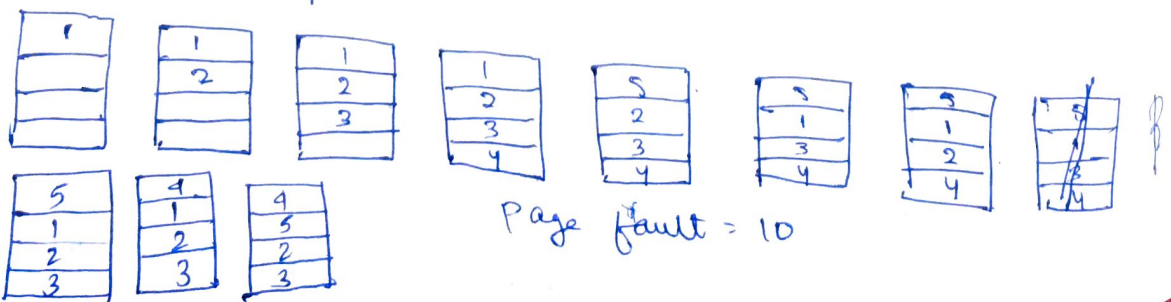
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

No. of frames = 3



Page fault = 9

No. of frames = 4



Page fault = 10

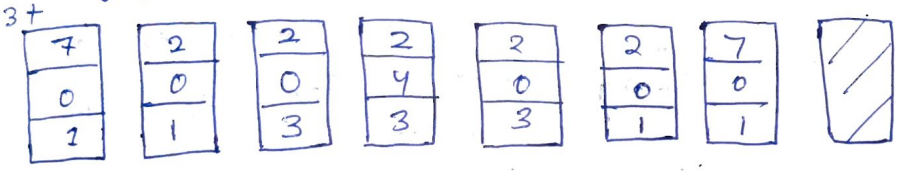
* The situation like this Even if increase the frame no

(ii) Optimal Page Replacement Algorithm

Replace the page that will not be used for the longest period of time.

Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

No. of frames = 3



Page fault = 9

Pages that are not available in future

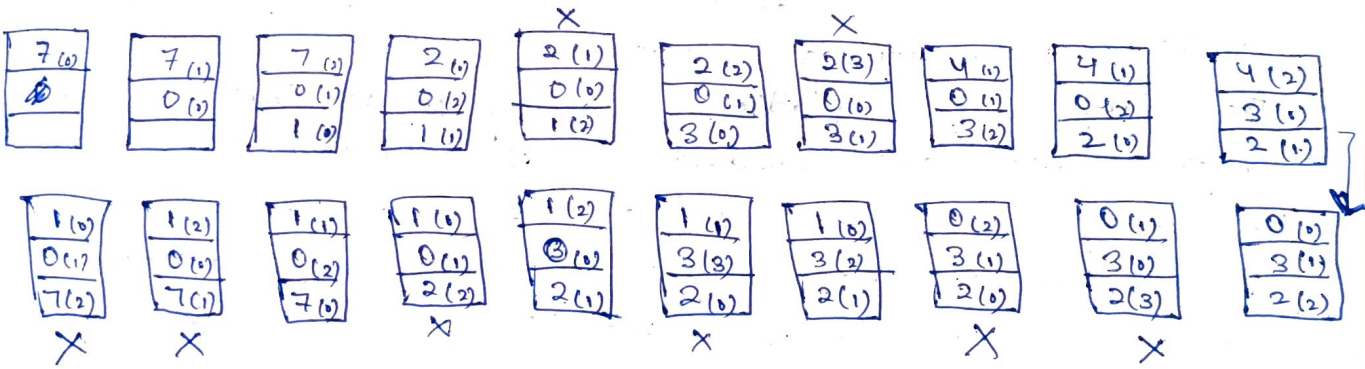
(iii) List Recently Used (LRU) Page Replacement Algorithm —

LFU
MFU

In this algorithm we place the page that has not been used for longest period of time.

* Same as above

* There are two methods for implementing the LRU algorithm using counter & the stack



Page fault = 12

Que. Application of Page Replacement Algorithm

Thrashing

If the no of frames allocated to a low priority process falls below the minimum no required by the computer Architecture, we must suspend that process execution. we should then page out its remaining pages, freeing all its allocated frame.

If the process does not have the no. of frame it need to support pages in active use, it will quickly page fault. Consequently it must replace some pages again & again and Pages fault occurs again & again.

This high paging activity is called Thrashing.

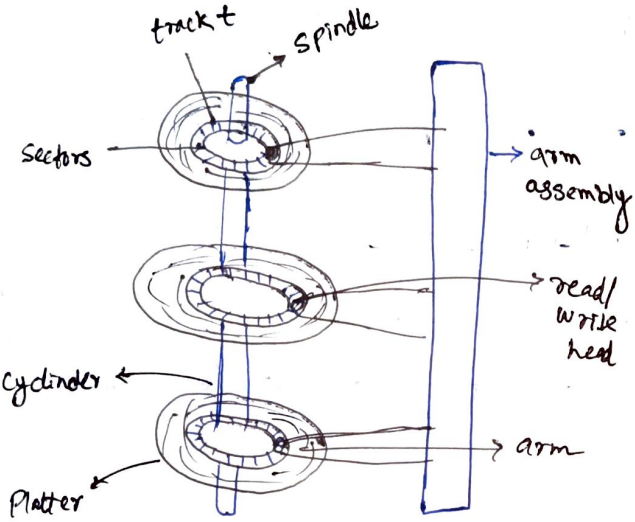
A process thrashing if it is spending more time paging then executing

Que. What is locality of reference?

Causes of Thrashing —

Secondary Storage: —

Magnetic Disks



Magnetic disk provide the bulk of secondary storage for modern computer system. Each disk ~~has~~ platter has flat circular shape like a CD. Two surfaces of a platter has covered with a magnetic material we store information by recording it magnetically on the platter.

A read/write head flies just above each surface of every platter. The Heads are attacted to a disk ~~r~~ that moves all heads ~~other unit~~ as a unit.

The Surface of a platter is logically divided into circular tracks which are subdivided into sectors.

The set of tracks that are at one arm position makes up a cylinder. There may be thousand of concentric cylinder in a disk drive and each ~~trak~~ track may contain hundreds of sectors.

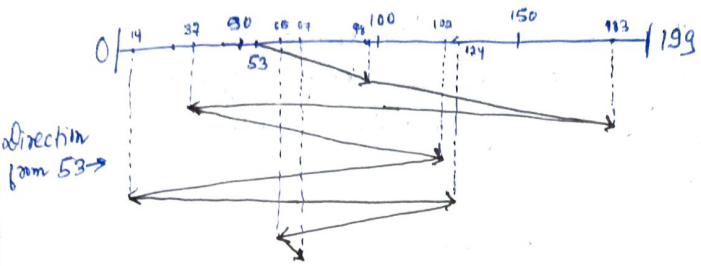
24/11/25

Disk scheduling Algorithm

1) FCFS

Disk queue with requests for I/O on cylinder does
98, 183, 37, 122, 14, 124, 65, 67
head starts at 53.

Max = 200



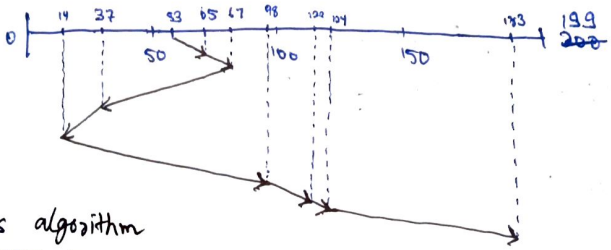
Calculate head moment
= (98-53) + (183-98) + (183-37)
+ (122-37) + (122-14) +
(124-14) + (124-65) +
(67-65)
= 640 cylinders

2) SSTF (Shortest seek time first)

Disk queue with requests for I/O on cylinders does —
98, 183, 37, 122, 14, 124, 65, 67

53
65-53 = 12 ← Shortest
53-37 = 16

98-67 = 31
67-37 = 30 ←



Calculate head moment
= (65-53) + (67-65) + (67-37) +
(37-14) + (98-14) + (122-98)
+ (124-122) + (183-124)
= 236 cylinders.

This algorithm
* suffer from starvation

53→65→67
53→37→14 = 208

Scan Algorithm

3) In the Scan scheduling Algorithm, the disk arm starts at one end and moves ~~forward~~ toward other end. Servicing requests as it reaches each cylinder until it gets to other end of the disk.

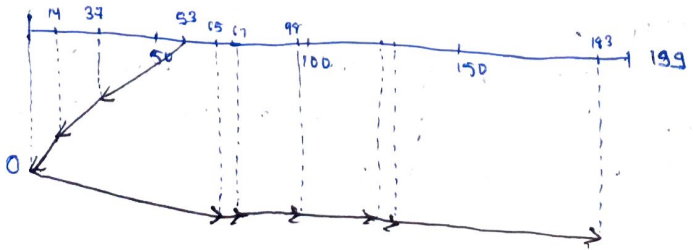
At the other end the direction of head movement is ~~reserved~~ reversed and servicing continues.

(ii) The head continuously scan back & forth across the disk. That's why the scan algo is sometime called elevator algorithm.

Scan

Requests → 98, 183, 37, 122, 14, 124, 65, 67
Head direction towards → 0th cylinder.

Max = 200



Head movement
= (53-0) + (183-0)
= 53 + 183
= 236

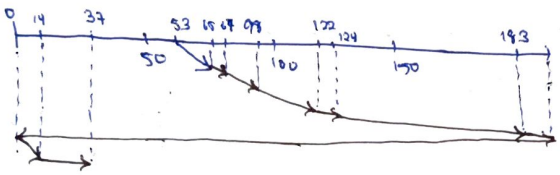
* C- Scan Scheduling —

(i) Circular scan scheduling is a variant of Scan ~~ed~~ scheduling and it provides a more uniform wait time (ii) like scan, C-scan moves the head from one end from the disk to other servicing request along the way.

(iii) when the head reaches the other end, however, it immediately returns to the begening of the disk without servicing any request on the return trip. treats

(iv) The C-scan scheduling algo essentially ~~trip~~ cylinder as a circular list that wraps around from the final cylinder to first cylinder

Request → 98, 183, 37, 122, 14, 124, 65, 67
Head direction ~~towards~~ → ~~top~~ th cylinder Max = 200



Total cylinder

Head Movement = (199 - 53) + 200 + 37
= 146 + 200 + 37 = 383

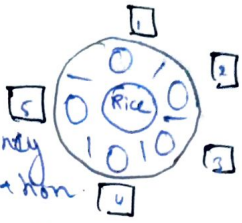
Real time system } Short Note
(Galvin ch-19)

Dining Philosophers Problem

Problem :

Suppose there are 5 philisophers who spend there live thinking & eating. ~~Each~~ the philosophers share a circular table surrounded by 5 chair. Each belonging to 1 Philosophers. In the centre of the table there is a bowl of rice and the table is laid ~~by~~ with 5 single chopsticks. when a philosophers think, ~~he or~~ she doesn't interact with their collegues. From time to time each Philosophers get hungry & tries to pickup 2 chopsticks that are closest to ~~them~~ her. A philosopher may pickup only one chopsticks at a time Obiously, she cannot pickup a chopstick that is already in the hand of a neighbour. when a hungry philosopher has both her chopsticks at the same time she eats without relasing ~~e~~ her chopstick.

when she is finished eating she puts down both of her chopsticks and start thinking again.



It is a example of a large class of concurrency control problem. It is a simple representation of the need to allocate several resources among several processes in a deadlock free & starvation free manner.

∴ For solving these problem, we use a synchronization tool called semaphores.

Semaphores

type → variable name

A semaphore 's' is an integer variable, that, a part from a initialization that is accessed only through two standard atomic operation : wait() & signal() the wait operation ~~is~~ was origanally termed as P and signal was originally called V the defination of wait function is as follows —

```
wait(s)
{
  while s<=0
    ; // no operation
    s--;
}
```

the defination of signal

```
signal(s)
  {
    s++;
  }
```

All modification to the integer value of the semaphores in the wait & signal operation must be executed indivisibly. That means when the process modifies the Semaphores values to other process can simultaneously that modify that same semaphore value.

In addition, the modification of s-- must be executed without interruption.

→ Binary Semaphore
→ Counting Semaphore