Operating System

Date: / / Page no:

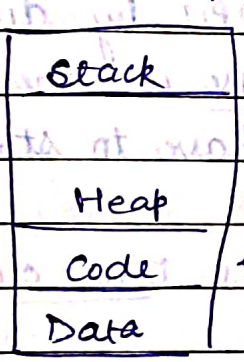
* **Program** → file resides on Hard Disk

```
int main()
{
    cout<<
}
```

compiler → Binary Program

It is first stored in RAM & then is executed.

* **Process**: that is loaded Program in RAM is the process (when called by processor)



← Program inside Memory.

* **Process States**

① Single Tasking System (MS-DOS)

New → In-Memory → Finished

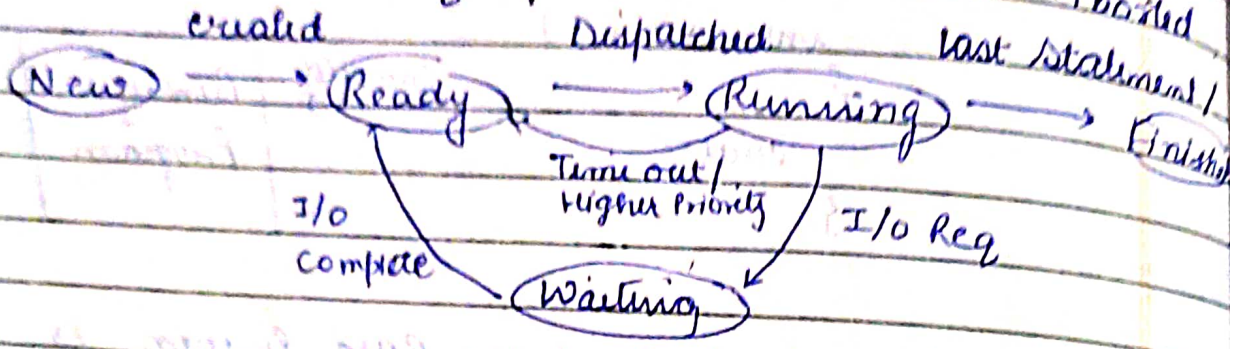
1) allows single process.
2) 1 prog in RAM at a time

Eg. CPU task will not performed if I/O is performing.

Date / /

(2) Multiprogramming System (5-State Model)

→ Max use of processor.



New → Program just Begin (Not Loaded on RAM)
HD

Ready → In RAM

Running → assigned to CPU by dispatcher.

i) Higher Priority Case if then currently running state have to stop & CPU does not pick OS assigned task to CPU

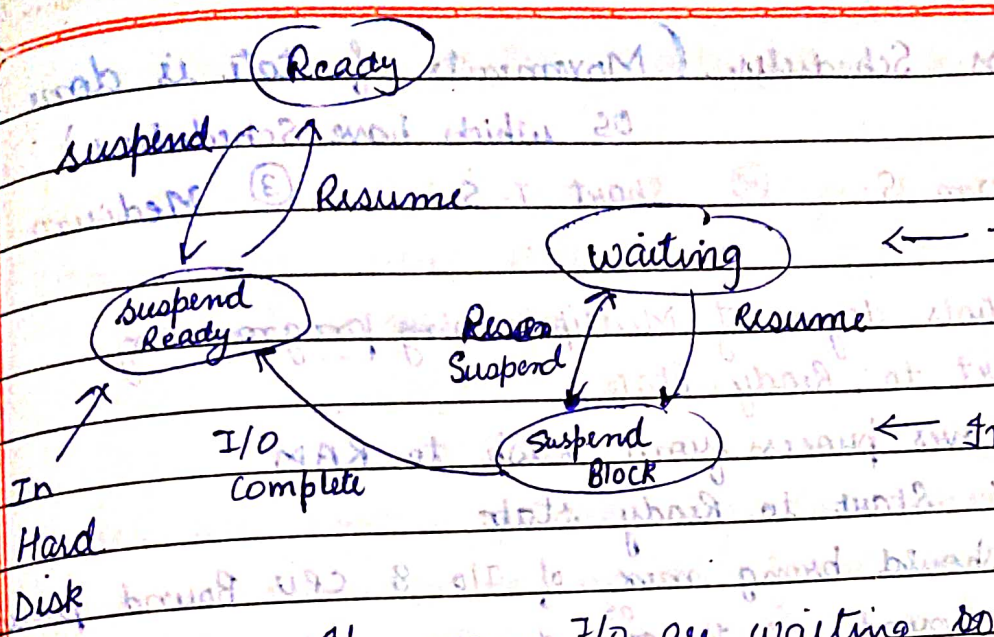
If current state's time out then also Ready state.

ii) Eg. scanf (I/O Requests) then user have to give data, processor can do some other work till then

Finished → Error / last statement comes

(7 states Model)

Date: / / Page no:



If many I/O are waiting so we move some of them from RAM to Hard Disk.

If a for priority task space is required then some tasks are transfered to Hard Disk.

* Process Control Block (PCB) → To store inf. about process
  - Most essent. Comp.

1) Process ID
2) Process State
3) CPU Registers
4) Account Inf.
5) I/O ~~Acc~~ Inf.
6) CPU Scheduling Inf.
7) Memory Information

Date: 1 1 Page no.

Process Scheduler. (Movements of state is done by OS which have Schedulers)

① Long Term S. ② Short T. S. ③ Medium T. S.

Decisions happens again after a long time.
- ↳ controls degree of Multiprocessing programming
- ↳ Start to Ready State.
- ↳ moves process from disk to RAM
  - ∴ Start to Ready state
- ↳ should bring mix of I/O & CPU Bound process
  - I/O Bound → If OS do more I/O process
  - CPU Bound → CPU

② Short T.S. (Dispatcher)
- ↳ Ready to Running.
- ↳ It allocates picked process to processor.
- ↳ It should always be active & take decision in short interval.

③ Medium T.S
- → waiting to suspended Block.
  - ∴ RAM to HD
- → Resume from suspended Ready to Ready
  - HD to RAM.

→ Job Queue
→ I/O Queue

Date: / / Page no:

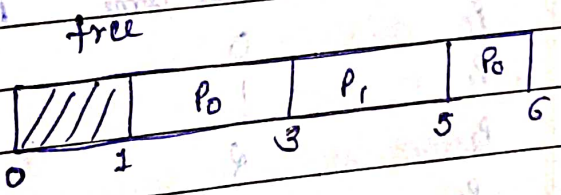
Scheduling Algorithm

* Short Term Scheduler & Dispatchers (Context switch)
→ These alg. are only for S.T.S. or for jobs in ready queue

* When is a process picked by the scheduler.
a) when other process moves from running to waiting. (Time completes / High Pro)
b) new to ready (High Priority) running to ready
c) when process terminates

(b) & (c) → preemptive

* Gant Chart



① Arrival Time : P0 (1)

② Completion Time : P0 (6)

③ Burst Time : (3-1) + (6-5) (P0)
2 + 1 = 3

④ Turn Around Time : C.T - A.T (P0)
= 6 - 1 = 5

⑤ Waiting Time : Time process spend in Ready Que.
2 ⇐ (3-5) (P0)

TAT - BT

Date: __/__/__ Page no:

⑥ Response Time

Arrival Time - 1st time it get CPU

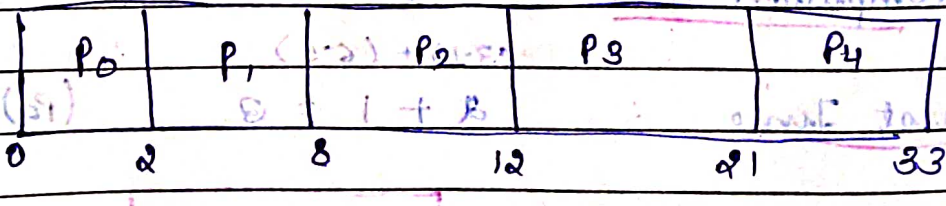
* Goals of Scheduling Alg.

① Max CPU utilization
② Max Throughput (No. of jobs per unit time)
③ Min Turnaround Time
④ Min Waiting Time
⑤ Min Response Time
⑥ Fair CPU Allocation (No Starvation)

* FCFS (Non-Preemptive) → 1st assign CPU to process / processor can't be taken.

Process	Arrival Time	Burst Time	Completion Time
P_0	0	2	
P_1	1	6	
P_2	2	4	
P_3	3	9	
P_4	4	12	

⇒



(C.T - A.T) Date: / / (TAT = BT)

Completion Time	T.A.T	Waiting Time
2	2	0
8	7	1
12	10	6
21	18	9
33	29	17
	66/5	33/5

Avg. Time a process spent in your System / Running State.

↓

Avg. Time a Process Spend in Waiting Queue.

→ Simple & Easy to Implement
→ Non-Preemptive
→ Convoy Effective.
→ Avg. waiting Time.

Not Optimal (CPU B.P. comes first)

, CPU Bound P.
Multiple I/O B.P.

→ if CPU takes 1st CPU B.P. which takes more time to process with CPU compared to I/O B.P. with CPU
→ & even after completing there will be a long Queue for I/O & all the I/O B.P. ~~Com~~ come together.

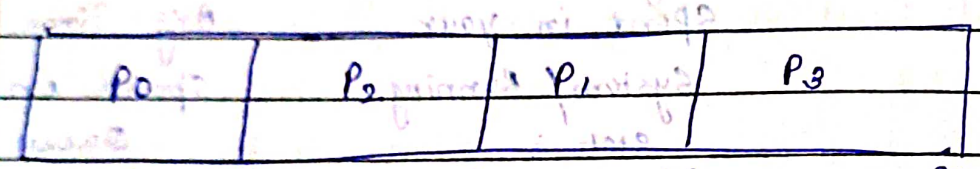
Round Robin is better than FCFS in terms of response time.

Date: 1/1 Page no:

Shortest Job First (Non-Preemtive)

given

Process	A.T	B.T	C.T	T.A.T	W.T
P_0	0	5	5	5	0
P_1	1	4	12	11	7
P_2	2	3	8	6	3
P_3	3	8	20	17	9
				39/4	19/4

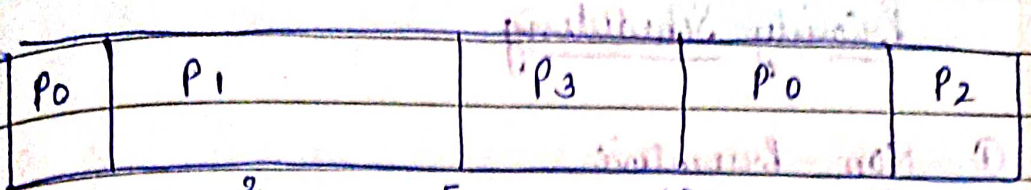


At 0 only P_0 was there.
Since Non prem. ∴ runs till 5
Now all process are present.
∴ process comes acc. to smallest B.T. first.

* SJF with Preemtion / Shortest Remaining Time First

Process	AT	BT	CT	T.AT	WT
P_0	0	8	17	17	9
P_1	1	4	5	4	0
P_2	2	9	26	24	15
P_3	3	5	10	7	2
				52/4	26/4

Date: __/__/____ Page no: _____



At 0 only P_0 was present ∴ P_1

1 → $P_0 = 7$, $P_1 = 4$

2 → $P_1 = 3$, $P_2 = 9$ ∴ P_1

3 → $P_1 = 2$, $P_3 = 5$ ∴ P_1

No new jobs ∴ normal SJF

→ Min. avg. waiting time among all Scheduling alg.

→ Impractical (CPU B.T. is diff. to find)

→ May cause high waiting & response time for CPU Bound jobs.

* Pre-emptive scheduling may cause starvation
* Shortest Remaining time first schedule may cause starvation.

Date: / / Page no.

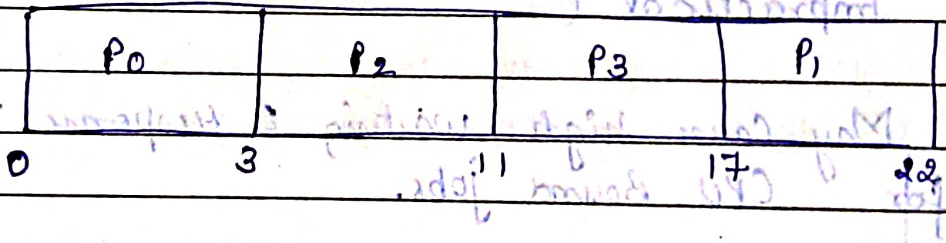
* Priority Scheduling

① Non - Preemtive

(same Priority then FCFS)

Process	A.T.	Priority	B.T.	CT	T.A.T	W.T
P_0	0	5	3	3	3	0
P_1	1	3	5	22	21	16
P_2	2	15	8	11	9	1
P_3	3	12	6	17	14	8

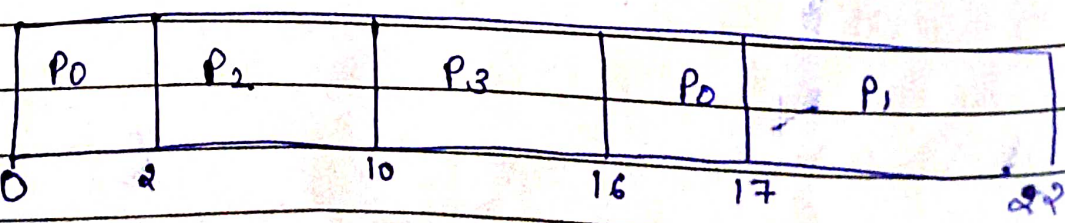
P_2 has highest priority.



② Preemtive

Resp. Time

	Process	A.T.	Priority	BT	C.T	T.A.T	W.T
0-0 = 0	P_0	0	5	3	17	17	14
17-1 = 16	P_1	1	3	5	22	21	16
2-2 = 0	P_2	2	15	8	10	8	0
10-3 = 7	P_3	3	12	6	16	13	7



Date: __/__/____ Page no: ____

* Waiting & Resp. Time depends on Priority.

⇒ low Priority can Lead to Starvation.

Soln Aging ↘ If process is waiting for a long time then its priority inc. & thus no starvation.

⇒ Priority can be decided

i) Statically by Long Schedulers
   Eg: deadlines (less deadline more priority)

ii) Dynamically by Running state.
   ↳ Eg Aging.

All 3 are Similar:

1. FCFS (Arival Time as Priority)
2. SJF (Burst Time as Priority)
3. Priority S. (Priority ———)

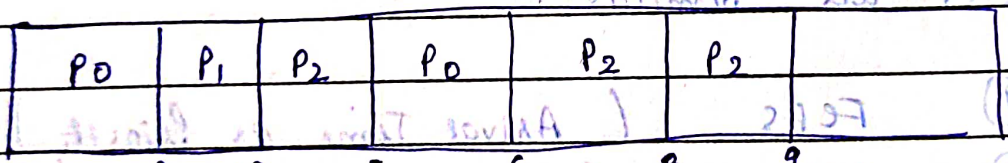
Date: / / Page no:

Round Robin Scheduling (Preemptive)

→ Time Quantum.
→ Circular Queue.
→ Avg. waiting time can be High, But good Response time
→ Sensitive to time Quantum
  ↳ Smaller : context switch overload.
  ↳ Large : Become FCFS

Process	A.T.	B.T	T.T	W.T	R.T
P_0	0	3	6	3	0
P_1	1	1	2	1	1
P_2	1	5	8	3	2

Time Quantum = 2



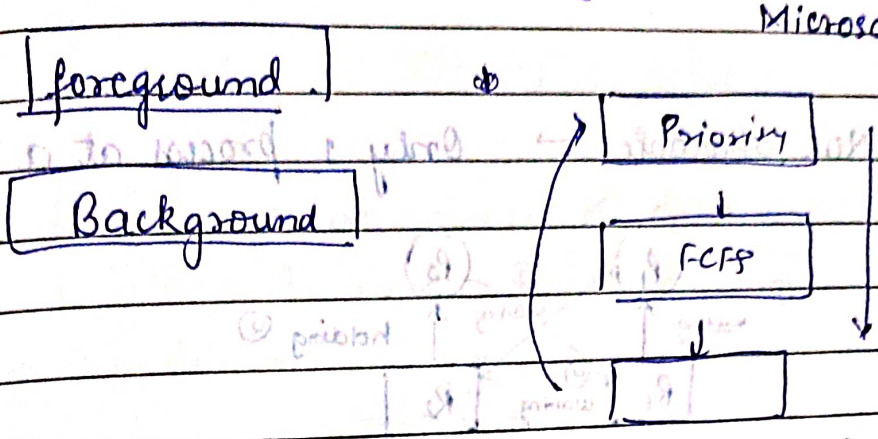
$P_0 P_1 P_2 P_0 P_2$

(use more advanced)

Date: __/__/____ Page no: ____

Multilevel Queue Scheduling. (Most used)

Microsoft ✓ Unix ✓ Solaris ✓



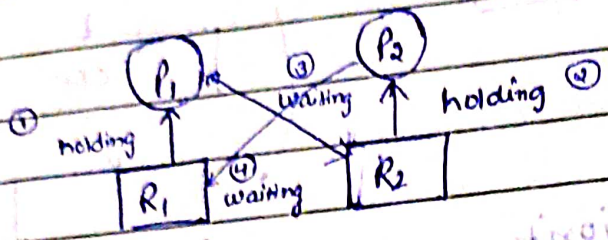
① Based on priority ✗ all are independent.

② Round Robin b/w Queues !

Date: / / Page no:

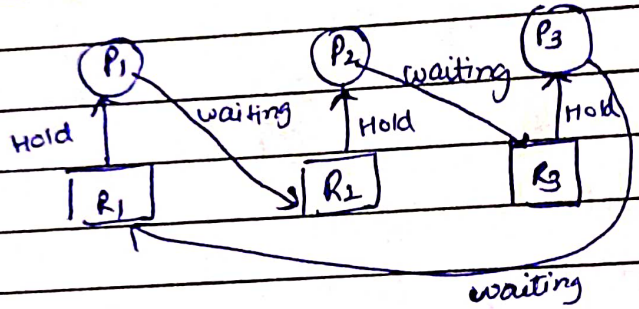
Deadlocks (No Progress)

* Not Sharable → Only 1 process at a time



Both processes are holding Non Sharable Resource & waiting for them only.

* Resource Allocation Graph



Cycle
∴ Deadlock

Necessary Cond.

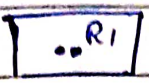
1. Mutual Exclusion : (Non sharable Resource)
2. Hold & Wait
3. No Preemtion (Resource Preemption)
   Once OS assigns a process it a Resource it can not be taken back.
4. Circular Wait
   Process are waiting for resources in circular manner.


Resource

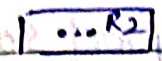

Process


Hold

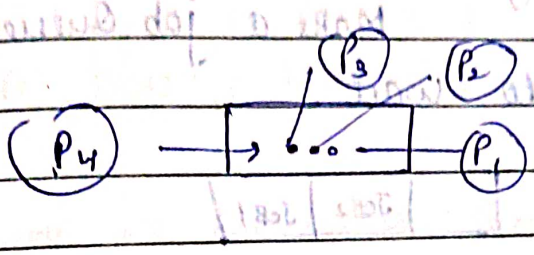

wait



2 instances of R_1



3 inst —— R_2.



Deadlock Handling Method

① Deadlock Prevention : One of all the 4 necessary cond. To avoid ~~all~~, * To Recieve a request that can cause Deadlock prevent.

② Deadlock Avoidance :
Any Req. can be sent to the OS but it runs some algorithm which decides to grant the req. or not.
(Banker's Algo.)

③ Detect & Recovery
Continuously check for deadlock & if it occurs then do something to Recover it.

④ Ignore the deadlock.
it is very infreq. so why to do extra work.

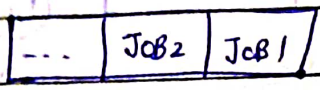
Date: / / Page no.

(1) Deadlock Prevention

4 ways (Eliminate any one of all)

(1) Mutual Exclusion

i) Spooling: (Not a full proof Sol^n)
Make a job Queue so no need to wait.



(2) Hold & wait

i) To declare all the Resources which the process will use in future (Impractical)

ii) A process can not make a new Req. untill unless it releases all the holding Resources.
(But wastage of Resources)

(3) No Preemption

OS can take processes. But it is ~~always~~ also impract because may be process is in middle & it is taken away.

(4) Circular Wait:

We No. Resource & make a rule that Resources can be take in inc. No. only.
Process having
Eg. Resource 2 can not take Resource 1.

Date: / / Page no:

② Deadlock Avoidance (Banker's Alg.)

	Allocated		Max		Need (MAX - Alloc.)	
	R_0	R_1	R_0	R_1	R_0	R_1
P_0	0	1	7	5	7	4
P_1	2	0	3	2	1	2
P_2	3	0	9	0	6	0
P_3	2	1	2	2	0	1
P_4	0	0	4	3	4	3

Req. from P_3 for $<1,0>$ should OS grant this req. or not ?

Total = $<10, 5>$
R_0 R_1

⇒ 1st check.
if Allocated + Req < Max

$<1,1> + <1,0>$
$(<2,1>) < (<2,2>)$ ✓

⇒ 2 Available 2nd check

$<10-6, 5-2>$
$<4, 3> \rightarrow <3, 3>$
Req < Available.
$(<1,0>) < (<4,3>)$ ✓

Initial avail.
$<3,3>$
P_0 Need < avail
P_0 ✗
P_1 $<1,2> < <3,3>$
$\therefore P_1$ ✓ $<3,0>$ +
$\therefore$ avail = $<5,3>$
P_2 ✗
P_3 ✓
avail = <5

$\therefore P_1 P_3 P_4 P_0 P_2$ ✓

⇒ Consider that req. is granted
Then check if it results in safe state or not
if yes then grant else not be granted.

Date: / / Page no:

If safe seq. is generated → safe state
else Not a safe state.

Safe seq → If you have serial order of processes which can execute with all resources

Safe seq = { 3

while (all Processes are added)
{
a) Find Pi such that
needi <= available.

b) If (no such i exists)
return false;

else (we found an i)
{
available += allocatei
Add Pi to Safe seq.
}

return true;

Date: __/__/____ Page no: _____

Limitations

① It assumes process knows its max need.
② It assumes before requesting new Res. all the prev. holding res. are freed.

* Deadlock Detection & Recovery

1) Detection

↳ If Resource Allocation graph have a cycle & Resources have single instances.

↳ For Multiple instances use Banker's Algorithm (Modified)

2) Recovery

Kill Process

① Pick a process & kill it
② Kill all Resources

Preempt Res. (take Res.)

→ It may lead to starvation

Date: __/__/__ Page no.

Process Synchronization

Process
- Independent
- Cooperative (comm. with other processes)

→ Interconnection of processes can be within computer or even with other computer

→ Synchronization takes place for both multiprocessors & single processors & with concurrent execution

↓ interleaving of processes

| P_1 | P_2 | P_1 | P_0 | P_2 | P_0 |

→ Processes communicate using shared Memory (global Variables)

//

```
int SIZE = 10;
char buffer[SIZE];        ← global variable
int in = 0, out = 0;
int count = 0
```

Date: __/__/____ Page no: ____

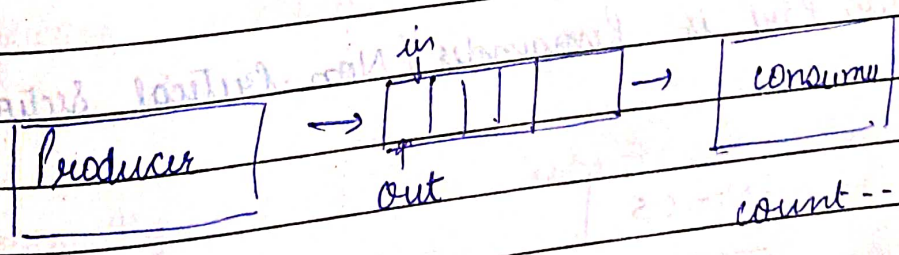
```
Void producer()
{
  while (true)
  {  while (count == SIZE);
     buffer[in] = produceItem();
     in = (in+1) % SIZE;
     count++;
  }
}
```

CPU register

reg = count
reg = reg + 1
count = reg

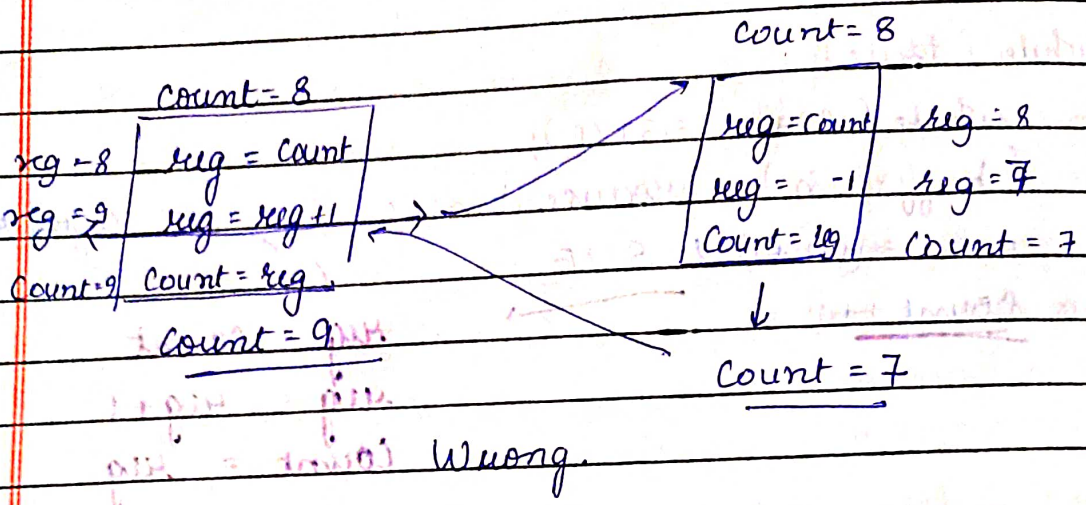
```
Void consumer()
{
  while (true)
  {  while (count == 0);
     consumeItem(buffer[out]);
     out = (out+1) % SIZE;
     count--;
  }
}
```

reg = count
reg = reg - 1
count = reg



Date: / / Page no:

Race Condition



Wuong.

when output depends upon sequence of execution it is K/a as Race Condition.

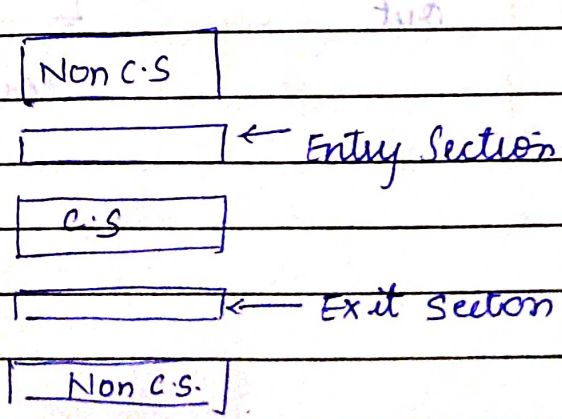
Critical Section

Part of code where one, shareable Resources are accessing known as Critical section.

Count++	Count--

Other Part is Remainder / Non-Critical Section.

Soln



Read Only → No need of critical section

It is only when multiple processes are writing on shared Resources

* Goals of Synchronization

① Mutual Exclusion (only 1 Operation in critical Section)
② Progress
③ Bounded Waiting (Fair Chance to enter critical section)
④ Performance (Lock Mech. should be Fast)

↳ Process who do not want to use shared Resource (critical section) should not enter critical section

① Perf: (Locks / Mutex)

Software (Slow) | Hardware (Faster & so it is preferred over software)

↓ Test & Lock

Peterson Alg.
Bakery Alg.

- Lock / Mutex is basic building block
- Process wait who wants to access critical section can only have to wait & other processes can run

Date: / / Page no:

Semaphore: It is a high level mech.

wait & signal

wait → if a process want to access critical section

signal → once process is done so as to inform allow other processes to access C.S.

It These 2 must be atomic

For this we use locks

Monitors

- Thread Synch. (used for)
- In Java
- JVM ~~work~~ manages monitors (Software Sol^n)

→ we put critical section in a class & all processes which want to access it are objects.

Application of Schronization

① Process Synchronization (usually we deal with it)

② Thread Synch. (Implemented by Kernel)

Date: __/__/____ Page no: ____

① **Locks**

Test & Lock.

```
int amount = 100;
bool lock = false;
```

```
void deposit (int x)
{
while (test_and_set(lock)){;}
amount = amount + x;
lock = false;
}
```

```
Void withdraw (int x)
{
while (test & set (lock)){;}
amount = amount - x;
lock = false;
}
```

Atomic // `bool test_and_set (bool * ptr)`

```
{
bool old = *ptr;
*ptr = true;
return old;
}
```

(using Hardware)

* Semp. are Variables operated by Kernel to maintain atomacity of wait & signal by using Locks.

Date: _/_ _/_ Page no:

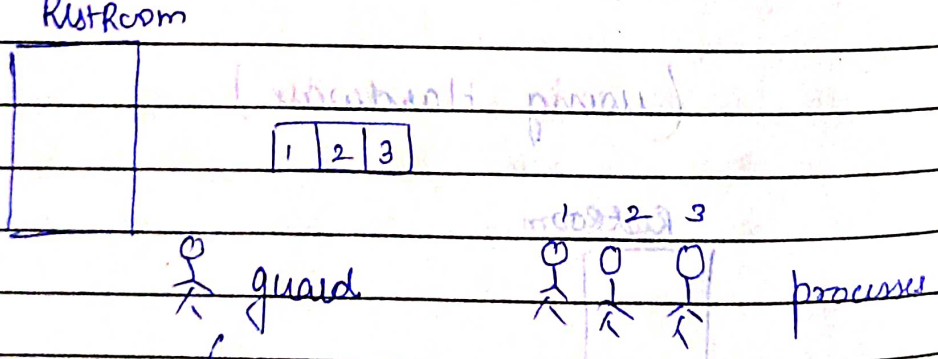
② Semaphore ① Counting S.

Problems to Lock :-

① Busy Waiting : Many processes are waiting for 1 process to leave C.S.
② No Bounded Waiting : NO Sequence or queue.
③ Multiple Non-Shareable Resources (with lock Complicated)

Semaphore maintains count.
wait → increments count
signal → decrements count

* Wait says don't wait (Run in loop) go Sleep & ~~Sem~~ with Signal Semaphore will wake it up.



if Restroom is occupied guard (semap.) tell processes go out I will call you when Restroom is available.

```
Struct Sem
{
    int count; // No. of (RestRoom) Resources
    Queue q;
};

Void wait()
{
    s.count--;
    if (s.count < 0)
    {
        ① Add caller process
           to q;
        ② sleep(p);
    }
}

Void signal()
{
    s.count++;
    if (s.count <= 0)
    {
        ① Remove process P
           from q;
        ② wakeup(P);
    }
}
```

* Original Implementation by Dijkstra

```
S = 3

P() {                          (wait in Switch)
    while (S == 0);
    S = S-1;
}

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

Problem → ① Busy Waiting
② If not going to use Res. still it is in Ready Queue (No need)

Date: __/__/__ Page no:

② Binary Semaphores.

- → Only True / False
- → Used as Lock also (Mutex) with extra functionality

```
Struct BinSem
{   bool val;
    Queue q;
};
```

```
BinSem S(true, empty)
          ↓        ↘ Queue
         val
```

```
Void wait()
{   if(s.val == 1)
        S.val = 0;
    else{
        ① Put this process P in q.
        ② sleep(p);
    }
}
```

```
Void signal
{   if( q is empty)
        s.val = 1;
    else {
        ① Pick process P from q
        ② wakeup (P)
    }
}
```

Date: __/__/____ Page no: ____

* Monitors.

→ Also built up on locks.
→ Usually By Multithreading Systems like JAVA.

class + Synch. functions

```
Class AccountUpdate {
    private int bal;
    void synchronized deposit (int x)
    {
        bal = bal + x;
    }
    void withdraw (int x)
    {   bal = bal - x;
    }
};
```

→ If even 1 func. ~~have~~ is sync. then that class is Monitor
→ One thread can access monitor at a time.

Priority Inversion

Low Priority (L)
Medium (M) > sharing one shared
High (H) Memory

if initially L is accessing than M will take & not H.

∴ To solve this problem Priority Inheritance takes place.

Priority of H is inherited by L so that M does not stop it to execute.

Date: __/__/____ Page no: ____

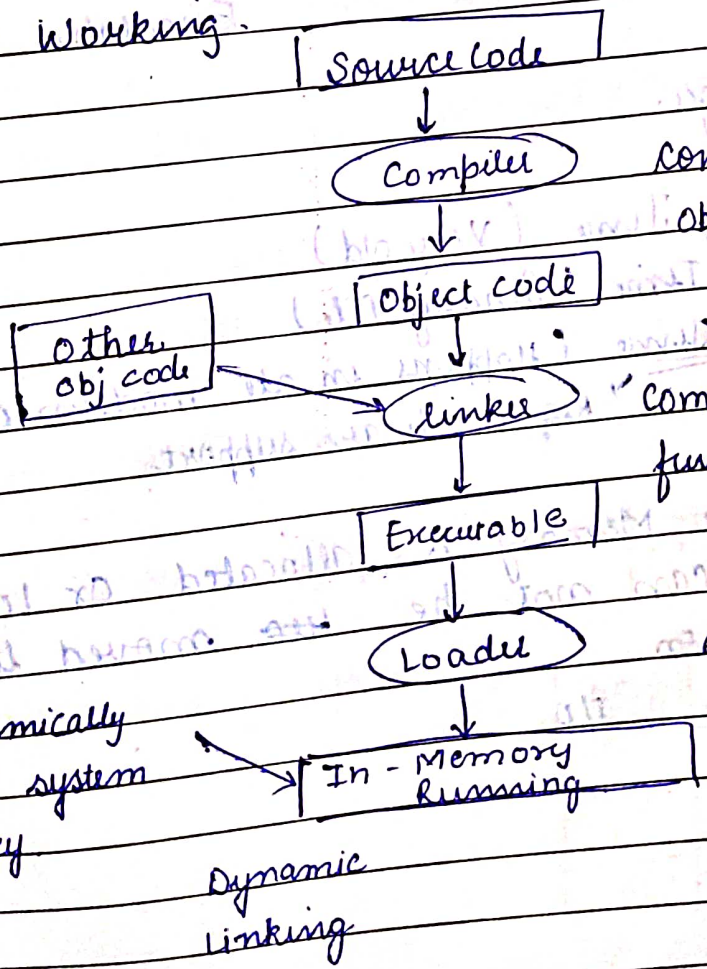
Memory Management

* Memory Hierarchy

CPU → Cache → Main Memory → Sec. Mem.

⇒ Memory should have
i) Low Access Time (Cache, M.M., S.M)
ii) Low cost
iii) High Capacity
(not other 2)

* Prog. Working

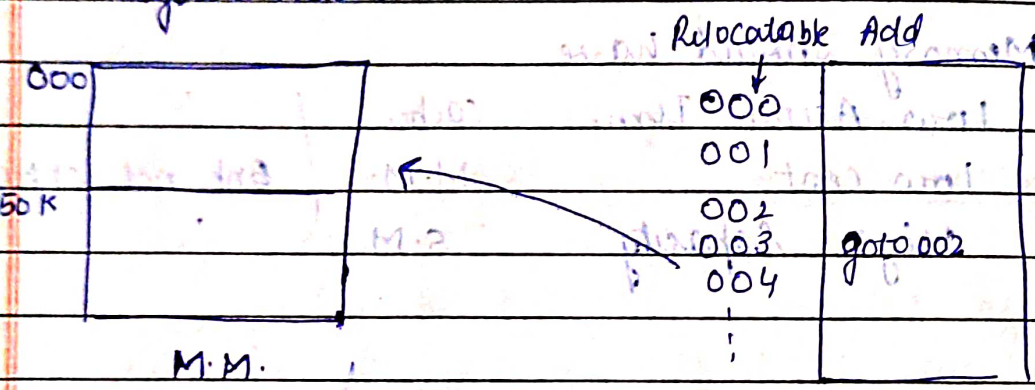


Date: / / Page no.

Address Binding

Mapping of Relocatable Addresses to Physical Address.

↓ generated when Decimal is converted to Binary



* Diff - stages.

1) Compile Time (Very old)
2) Load Time (Binary File)
3) Run Time (Happens in all modern OSs)

↳ Req. Hardware support.

Problem:

One Memory is allocated or loaded then it can not be ~~the~~ moved to diff. Location.

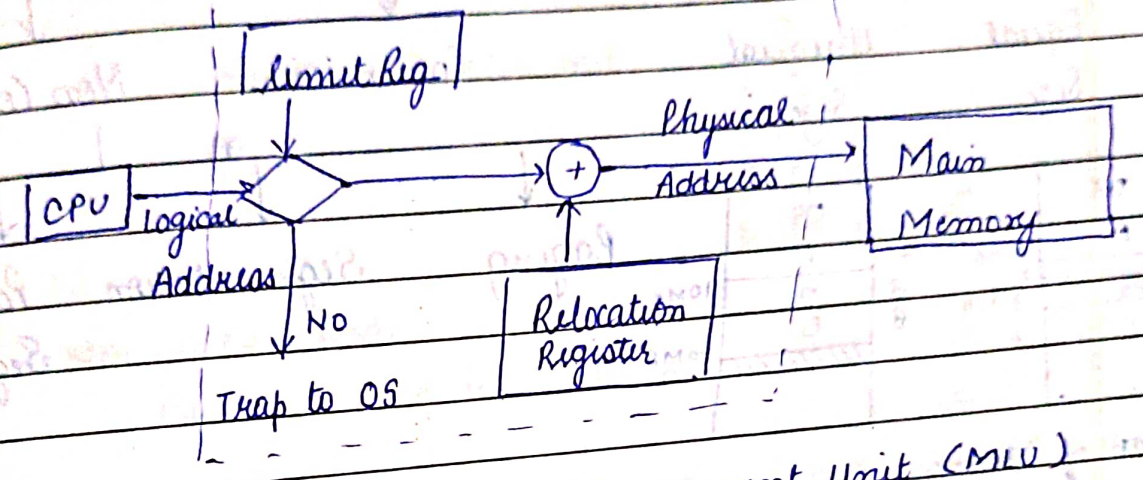
Eg. I/O

Date: __/__/____ Page no: ____

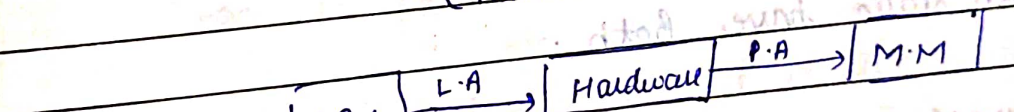
Run Time

CPU.

To move process to diff. location we convert logical address (generated by CPU) to physical address.



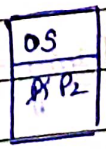
Memory Management Unit (MMU)
(Hardware is required)



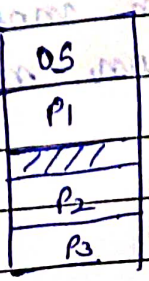
* If we use software then it will be very slow & so we use hardwares.

⇒ Evolution of Memory Management

1) Single Tasking System



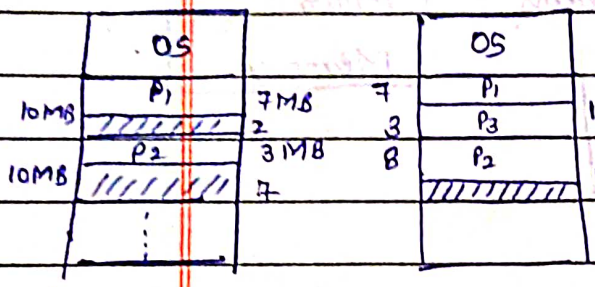
2) Multiple Tasking System



Date: / / Page no:

M.M. in Multi Tasking

- Static
  - Equal Size
  - Unequal Size
- Dynamic
  - Continuous
    - Paging
    - Segmentation
    - Paging & Segm.
  - Non Conti.



∘) 9MB of Memory is wasted

Both static have Both:

① External Fragmentation:

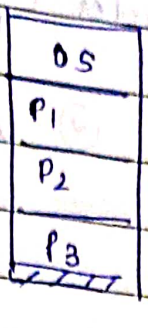
When a process want to add a size of memory which is available but in seperate junks so it can not be used

② Internal Fragment:

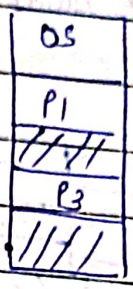
when more memory is needed allocated than needed.

Continuous

P1 1MB
P2 8MB
P3 3MB



Now P2 goes out

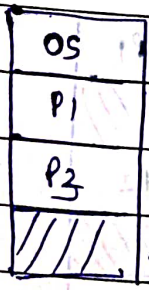


Now P4 4MB

But we can not allocate it.

∴ No Internal Fragmentation but still External Frag. is present.

We can do contraction but it is very costly



Dynamic Partitioning

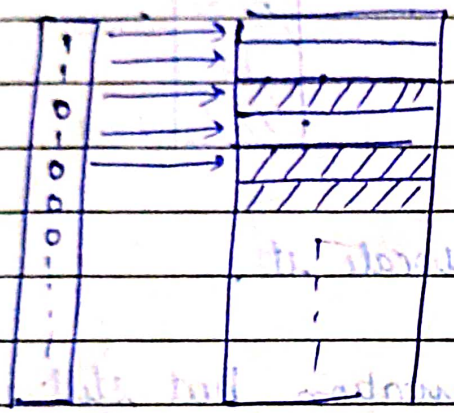
Date: / / Page no:

* way to handle holes (free spaces)

① Bitmap ② Linked List

↓

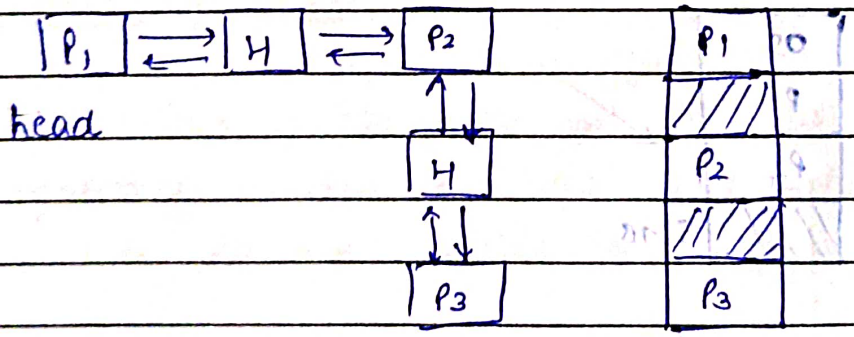
Req. large amt of memory to store it.



1 denotes allocated mem
0 denotes free space

∴ But it is not used much.

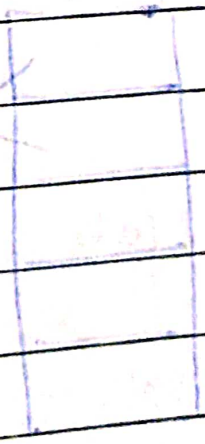
Linked List



If P2 is removed

Best

① First Fit
② Best
③ Next
④ Worst

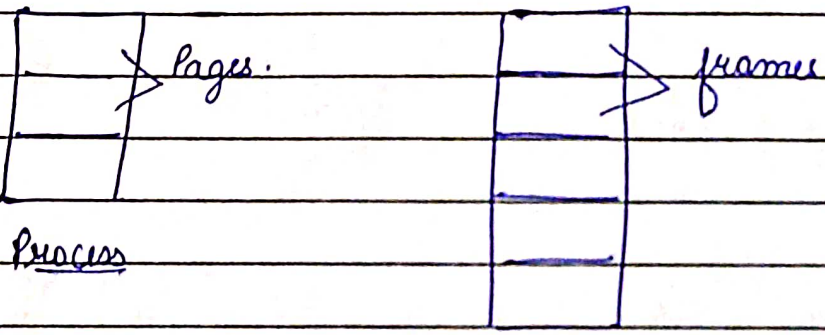


Date: _/_/_ Page no:

* **Paging :**

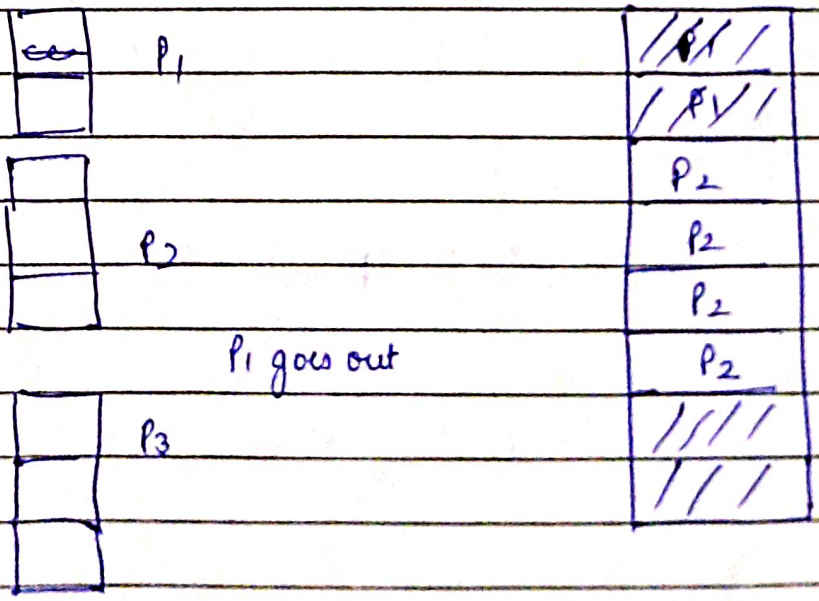
Programmers see it as continuous memory allocation & but internally it is stored in non-contiguous manner.

* Page size always equal to frame size



Process are divided into equal sized blocks K/a Pages
Main Memory frames

Eg:



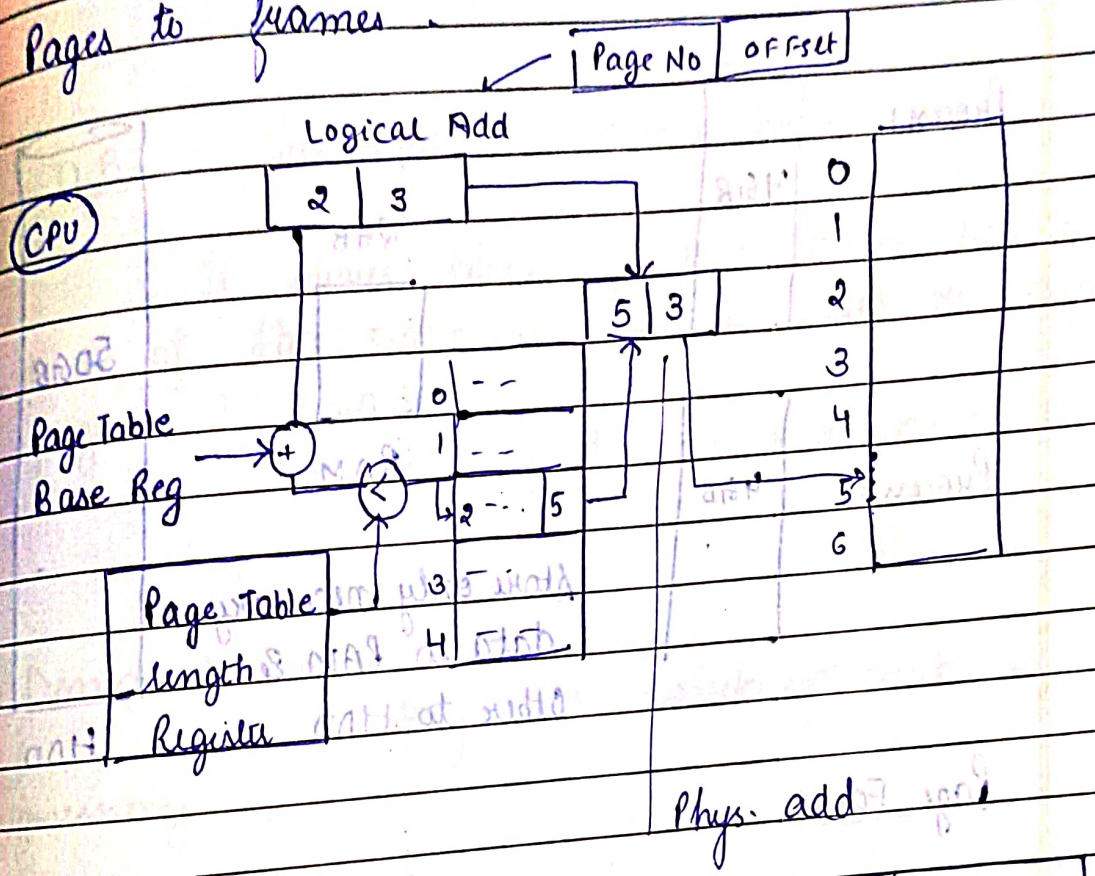
Pages goes to Frame either continuously or not.

Date: __/__/____ Page no: ____

we need run time Binding.

* Page Tables.

They stores mapping of Logical address Pages to frames.

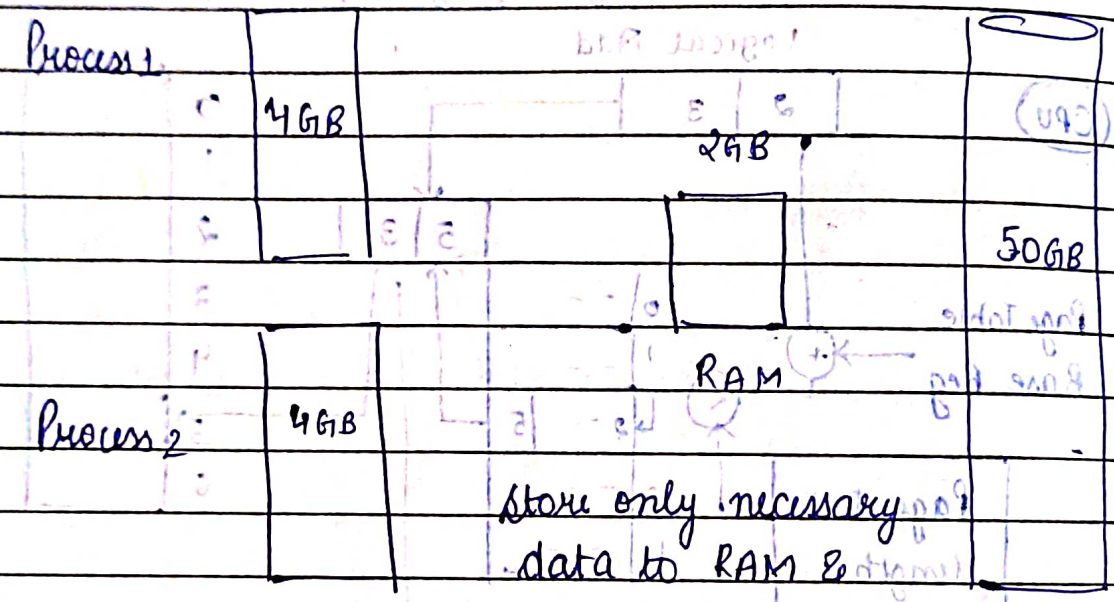


Paging is nothing but dividing memory into frames to store process non-continuously & to be maintain page to frame mapping.

Date: / / Page no:

Virtual Memory

It gives illusion of having full Memory available but in reality it is not so



Page Fault

If req. is not present in RAM & we OS have to look for it in HDD, it is Page Fault

Very costly operation.

Avg. Access Time = 0.99 x 10 ns + 0.01 x 5000000 ns
= 50 ms

0.99 → Hit Ratio Prob.
0.01 → Page Fault Prob

* TLB Translational Lookaside Buffer.

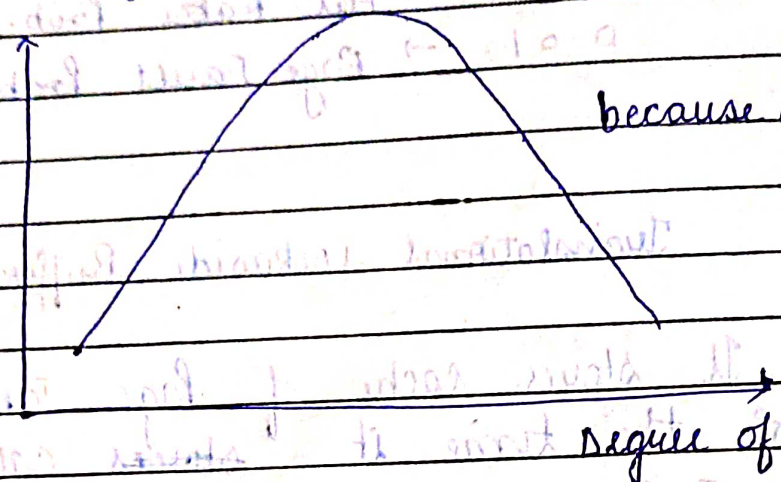
It stores cache of Page Table because most of the time it calls for same page No.
Hit Ratio of TLB is very high.

* Demand Paging:
Only Keep working part in memory.

* Pure Demand Paging:
Keep minimal working part
⇒ Load few req. things to RAM in order to optimise
Page Faults may occur.
CPU utilization will go down.

Date: / / Page no:

Thrashing



Because of ↑ Page Faults due to ↑ degree of Multiprocess CPU utiliz. ↓ this is K/a Thrashing

* Page Replacement Algorithm:

1. First In First Out (Suffers from Belady's Anomaly) ↓ If No. of Frame ↑ Page Fault also ↑
2. Optimal
3. Least Recently Used

When Page Fault occurs, OS need to bring page from HDD to RAM & RAM do not have space so we need to replace a page. For this we need Page Rep. Alg.

Data: __/__/____ Page no: ____

P.R.A. can be Local & Global

→ Replace this processed page → popular

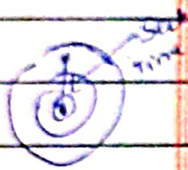
→ Replace other process's page

Disk
String
Graph
Rest
Memo.
Flag.
DP

To implement Sec. Memory. (I/O)
Not part of co

Disk Scheduling

- Each Disc has flat circular shape like CD or DVD.
- Generally Diameter ranges from 1.8 to 5.25 inc.
- Both surface are covered by magnetic film mat.
- Disc is divided into tracks
  ↓ further divided in
  Sectors.



Seek time → time required to move R/W head on desired track.

Disk Scheduling → In case of multiple I/O request, disk Sch. algo. must decide which req. must be executed first.

① FCFS

- Simplest
- Req. are entertained in order they arrive

Adv.

- Easy to implement
- No starvation (May suffer from convoy effect)
- Can be used with less load

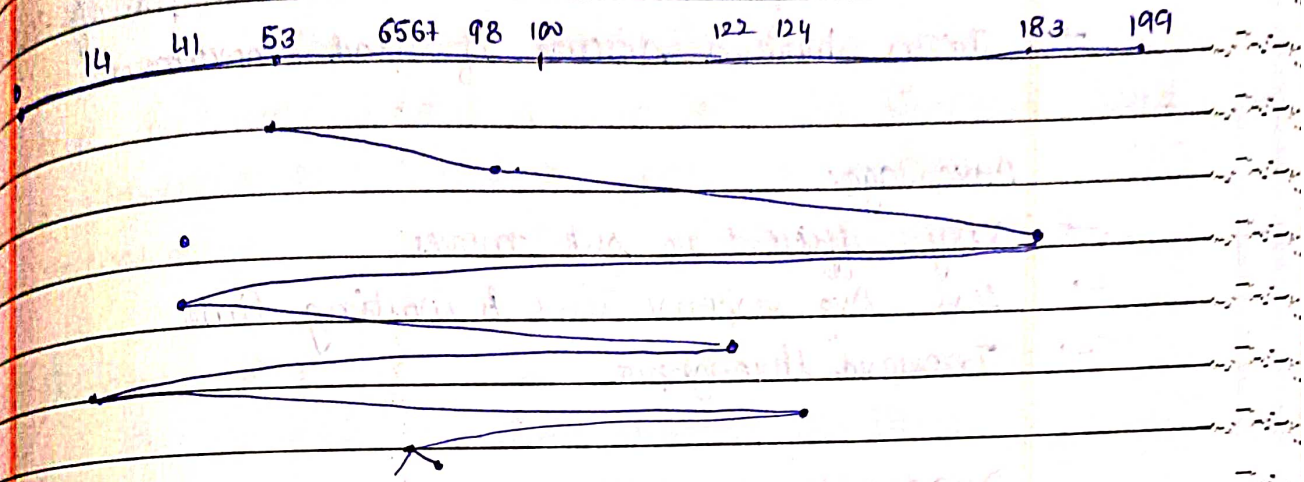
Disadvantage.

- Require seek time
- Inefficient
- More waiting & response time.

Date: / / Page no:

98, 183, 41, 122, 14, 124, 65, 67

Head = 53



(98 - 53) = 45
+ (183 - 98) = 85
+ (183 - 41) = 142
81
108
110
59
2
632

total move (Seek / track)

→ Avg. disk access time = seek time + Rotational latency

Capacity of disk = surfaces * track per surf * Sector per track * bytes.

To No. of Sector = Surf * tracks * Sector = 2^x

bits = x.

Date: / / Page no:

④ SSTF (Shortest Seek Time First)

- Services req. which is closest current pos. of R/W head.
- It is broken direction of head movement.

Advantage:
- Very efficient in seek moves.
- Less Avg response Time & waiting time.
- Increased throughput

Disadvantage:
- overhead to find out closest req.
- Request which are far from head will starve
- High variance in response time & waiting time.

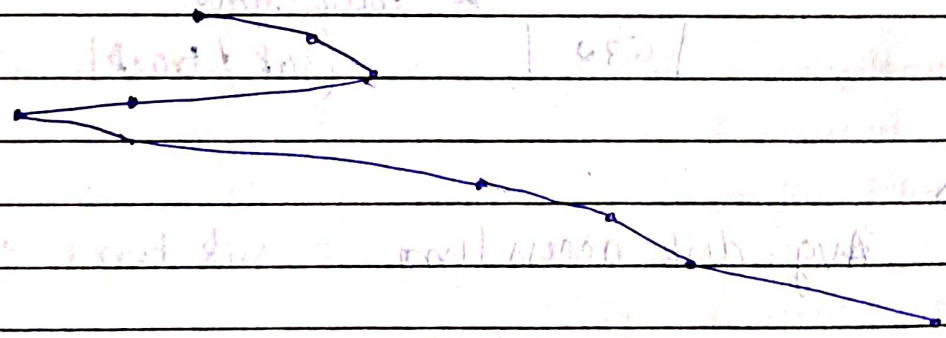
2 diff. ←

41 ✓ 199 ✗ 98 183 41 122 14 124 65 67 53 ↑ up

Sort ∴ Not variance.

<12> <12> But 65 because ↑

0 14 41 53 65 67 98 122 124 183 199



65-53
67-65
67-41
41-14
98-14
122-98
124-122
183-124
= 236

or 67-53 = 12
+ 67-14 = 53
+ 183-14 = 169
236

199 - 53 = 146
199 - 183 = 16

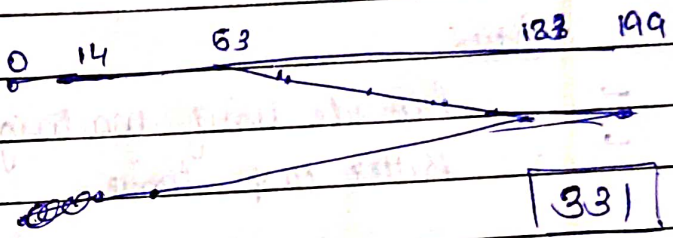
Date: / Page no:

(14)

③ SCAN disk Scheduling

Head starts at one end of disk & moves towards other end, servicing requests in its way & reach other end & then direction of head is reversed & processed. Continue & head continuously scan backs & forth across disk

Elevator Alg



331

199 - 53
+ 199 - 14

Adv:

① Simple
② No Starvation (Bounded Waiting)
③ Low variance & avg. waiting time

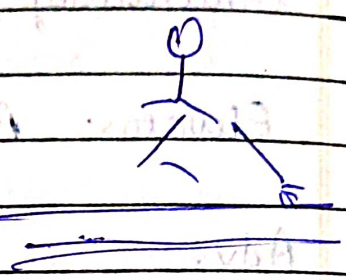
Disadv:

① Long waiting time for starting pt. just visited by head
② Unnecessary mobile till end of disk even if there is no request.

Date: / / Page no:

C-SCAN (Circular - SCAN)

Head start at one end of disc & moves towards other end servicing requests in b/w & reach other end & then direction of head is reversed & head reaches first end without satisfying any request.



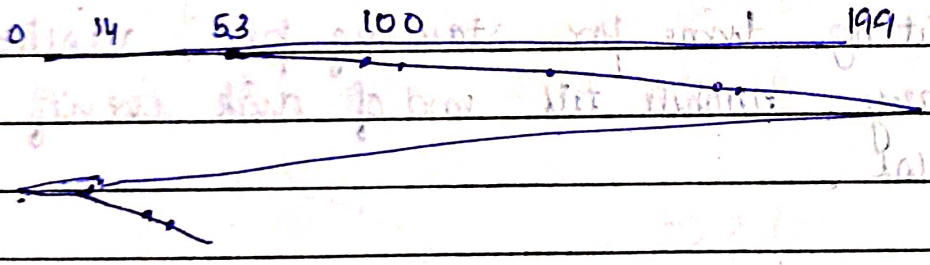
Adv:

→ Provide uniform waiting time.
→ Better resp. time

Disadv:

seek

More & mov. as comp. to SCAN



199 - 53 = 146
199 - 0 = 199
41 - 0 = 41
386

386

Date: __/__/____ Page no: ____

LOOK

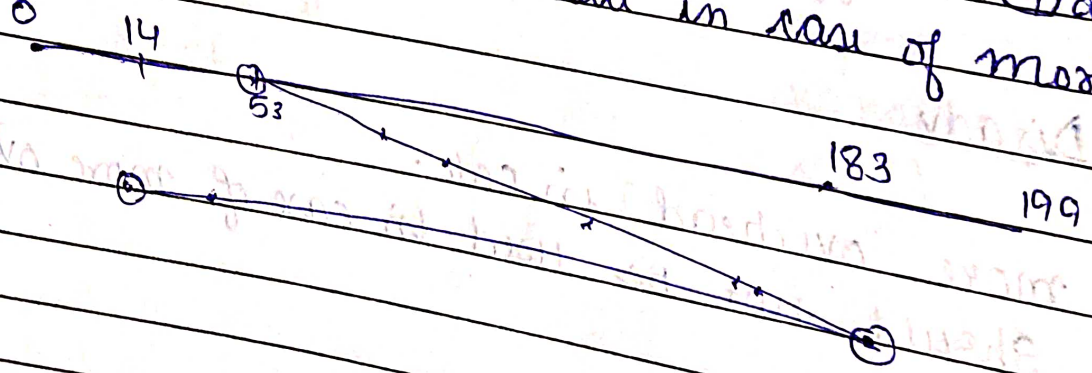
It is same as SCAN alg. but instead of going till last track we go till last req & change dir.

Adv.

- Better perf (SCAN)
- less load → (auto)

Disadv

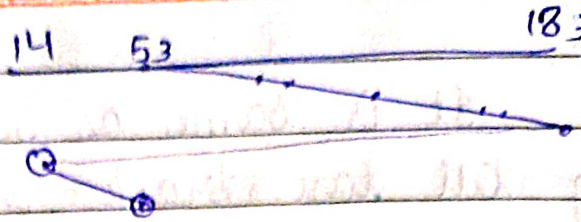
- Overhead to find last req. (brain)
- should not be used in case of more load.



183 - 53 = 130
183 - 14 = 169
41 - 14 = 27

Date: / / Page no: 326

C-LOOK



Adv. of both C-SCAN & LOOK algo.

→ will satisfy request only in one dir.
→ will go till last req. & return back not till last track.

Adv.

→ More unif. waiting time compared to look C-SCAN.
→ More eff. compared to C-SCAN.

Disadv

→ more overhead in calc.
→ Should not be used in case of more overload.

Date: __/__/____ Page no: ____

Classic Problem

Producer - Consumer

```
Void producer()
{
while(1)
{
produce()
wait(E)
wait(S)
append()
Signal(S)
Signal(F)
}
}
```

```
Void consumer()
{
while(1)
{
wait(F)
wait(S)
take()
signal(S)
signal(E)
use()
}
}
```

Date: __/__/____ Page no: ____

(2) Reader - Writer

```
writer()
{
    wait(write)
        write operation
    signal(write)
}
```

```
Reader()
{
    wait(mutex)
    readcount++
    if(readcount == 1)
        wait(write)
    signal(mutex)
        read operation
    wait(mutex)
    readcount--
    if(readcount == 0)
        signal(write)
    signal(mutex)
}
```

Date: __/__/____ Page no: ____

⑥ Dining Philosophers.

```
Void Philosopher (void)
{
    while (T) {
        Thinking ()
        wait (s);
        wait (chopstick [i]);
        wait (chopstick [(i+1)%5]);
        Eat ()
        signal (s);
        signal ( chopstick [i]);
        signal ( chopstick [(i+1)%5]);
    }
}
```

* To prevent deadlocks:

① Allow at most 4 philosophers to be sitting simultaneously at the table.

② Allow 6 chopsticks to be used simultaneously at the table.

③ Allow philosophers to pick up her chopsticks only if both chopsticks are available.

↳ altering problem statement.

Date: __/__/____ Page no: ____

ans

x One philosopher picks up her right chopstick first & then left chopstick i.e, reverse sequence of any philosopher.

* Odd philosopher picks up first her left chopstick & then her right chopstick, whereas even philosopher picks up her right chopstick & then her left chopstick.

Date: / / Page no.

(1) Peterson Solution

P0

```
while (1)
{
    flag[0] = T
    turn = 1
* ⇒ while ( turn == 1 && flag[1] == T);
    critical section
    flag[0] = F
}
```

P1

```
while (1)
{
    flag[1] = T
    turn = 0
    while ( turn == 0 && flag[0]
    critical section
    flag[1] = F
}
```

Mutual Exclusion ✓
Progress ✓
BW ✓

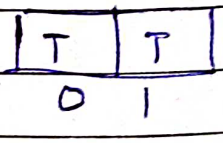
(2)

```
while (1)
{
    flag[0] = T
    while ( flag[1] == T);
}
```

✓ switch →

```
while (1)
{
    flag[1] = T
    while ( flag[0] == T);
}
```

Mutual Exclusion ✗
Deadlock ✓



Date: / / Page no:

```
while (1)
{  turn = 1
   while (turn == 1);
      critical section
}
```

```
while (1)
{
   turn = 0;
   while (turn == 0);
      Critical section;
```

Mutual exclusion ✓

Progress X

$2^{20} = 1M$

$2^{30} = 1G$

$2^{40} = 1T$

Date: / / Page no:

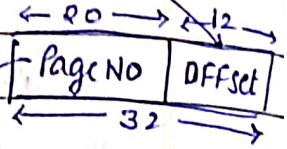
Numerical (Memory is byte addressable)

Given = Logical Add. space = 4GB

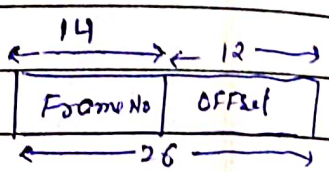
Physical Add space = 64 MB

Page size = 4 KB = $2^2 \times 2^{10} = 2^{12}$

⇒ ① Logical address = $2^2 \times 2^{30} = 2^{32}$ Byte



② No. of Pages = 2^{20}



③ No. of Phy. add = $2^6 \times 2^{20} = 2^{26}$

④ No of Frames = 2^{14} * 16KB

⑤ No. of Page Table Entry = No. of Pages in Process
= 2^{20} = 1 Million

Valid / Invalid

⑥ Size of Page Table = $2^{20} \times 14$ (14 + 2 = 16)
(No. of Page) (Frame No bit)

16 bit ⇒ 2 byte

2^{20} 1 Page = 16 Frames (14 bit)

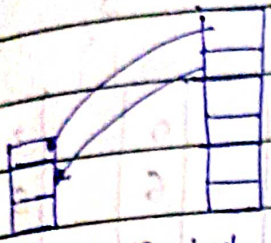
$\therefore 2^{20}$ Page = $2^{20} \times 14$

⇒ $2^{20} * 2$

= 2 MB

Locality of Reference

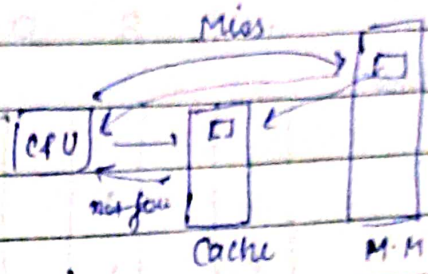
① Spatial Locality



Compiler Block is moved

Because if ||| is used then there is high chances that next instruction is close to this.

② Temporal Locality



In case of miss CPU goes to MM that particular word is copied to cache

There is high chances that Next instruction will be the same

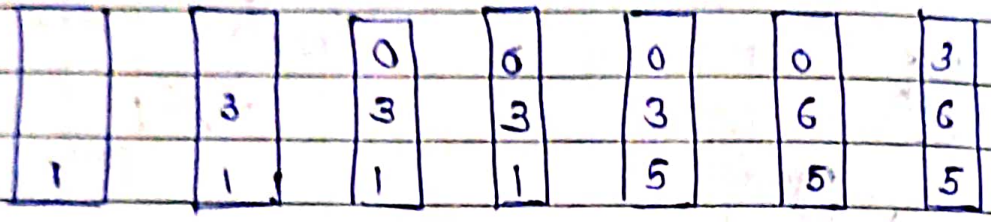
Date: / / Page no.

Page Replacement Algo.

① FIFO

1 3 0 3 5 6 3

3 Page Frame



Miss M M. Hit Miss M. Miss

6 Page Faults. * Belady's anomaly

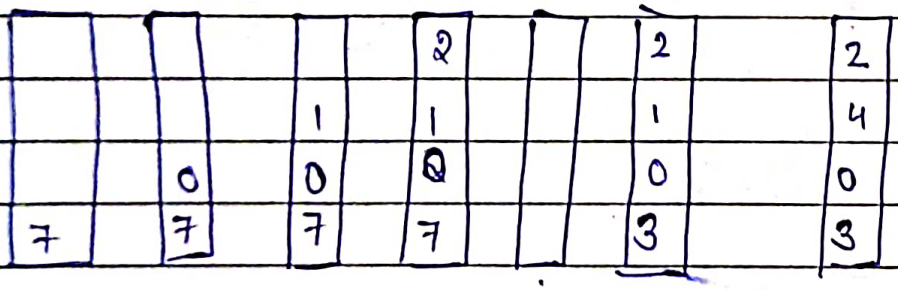
No. of page Faults ↑ with ↑ in page frame

② Optimal Page Replacement.

No. of Page Frame = 4

because 7 is Not used for longest duration

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



Hit Hit H H H H H H

∴ 6 Page Faults

It is perfect but not possible in practice as OS Cannot Know future req.

It acts as benchmark so that other replacement algorithm can be analyzed

Date: __/__/____ Page no: ____

(3) Least Recently used.

Same for this.

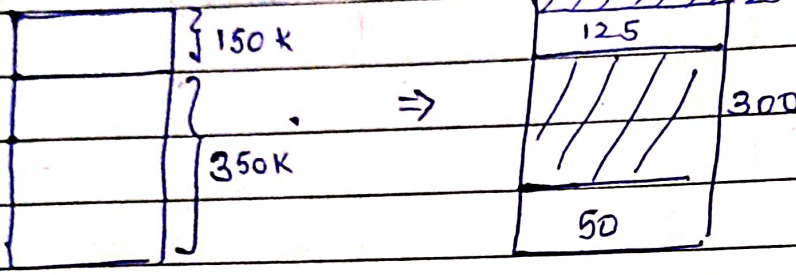
(because 7 is least recently used)

Date: __/__/____ Page no:

Partition Allocation Scheme. (used to avoid internal fragmentation)

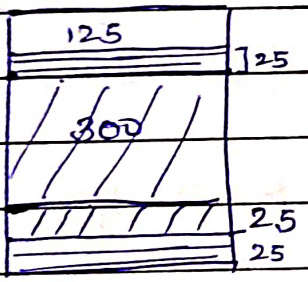
Q 300K 25K 125K 50K.

① First Fit → allocate to largest first hole large enough.



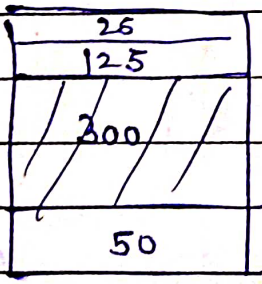
0 left

② Best Fit → smallest hole whose size is greater than or equal size of process.



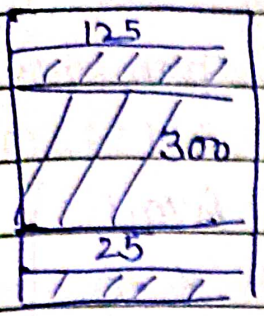
∴ 50K is wasted

③ Worst fit Opp Best fit (largest hole)



0 ✓

Next fit → search for first sufficient partition from last allocation point



50 wasted.

→ yang scheduling

For parallel systems that schedules related threads or processes to run simultaneously on diff processors.

Rate Monotonic Scheduling

used in real time OS with static priority scheduling class. It is assigned on basis of cycle duration of job
(Shorter the cycle of the job, higher the priority)

Fair Share scheduling

It is a scheduling strategy in which CPU usage is equally distributed among system users or group as opposed to equal distribution among processes.
It is K/a guaranteed Scheduling.

Date: / / Page no:

Bankers Alg. (Deadlock Avoidance) + Deadlock Detection

Q Total A = 10 B = 5 C = 7 — pre mentioned

(10-7) (5-2) (7-5)

Process	Allocation			Max Need			Avail			Need (Req.)		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2	~~7~~	~~2~~ 4	3
P_2	2	0	0	3	2	2	5	3	2	1	2	2
P_3	3	0	2	9	0	2	7	4	3	6	0	0
P_4	2	1	1	4	2	2	7	4	5	2	1	1
P_5	0	0	2	5	3	3	7	5	5	5	3	1
	7	2	5				10 ✓	5	7	✓		

Avail ~~Alloc~~ = Need

⇒ Req. Need = Max Need - Alloc.

P_1 Re.Need x

P_2

1 < 3

2 < 3

2 <= 2 ✓

if Need <= Avail ✓
else x

i) P_2 → one complete then it release other resources

P_3 X

P_4 ✓

P_5 ✓

ii) P_4 ✓

iii) P_5

iv) P_1

v) P_3

$P_2 - P_4 - P_5 - P_1 - P_3$

9

			Avail	Req[1]
P_1	~~3~~	9	~~4~~	6
P_2	✓	6	~~5~~	5
P_3	~~3~~	5	0	2
P_4	0	10	9	10
	7			

P_3 ✓
P_2 ✓
P_1 ✓
P_4 ✗ ∴ Deadlocked.

3 ~~2~~ 2

				Need	
P_0	0 0 1	8 4 3	~~8~~ 4 2		3 2 2
P_1	3 2 0	6 2 0	3 0 0		6 4 2
P_2	2 1 1	3 3 3	1 2 2		8 5 3
	5 3				
	3	P_1 ✓			
		P_2 ✓			
		P_0 ✓			

P_0	1 0 1	4 3 1	3 3 0	3 3 0 ✓
P_1	1 1 2	2 1 4	4 3 1	1 0 2 ✓
P_2	1 0 3	1 3 3	5 4 3	0 3 0 ✓
P_3	2 0 0	5 4 1	6 4 6	3 4 1 ✓
			3 4 6	

Date: __/__/__ Page no. ____

Resource Req. Alg

i) if req ≤ need go to step2.
Else Error.

ii) if request ≤ available, go to step3
Else wait.

iii) Available = Available - Request.

Allocation = Allocation + Request

Need = Need - Request.

iv) Check new state if safe or not.

Safety Alg.

if need ≤ available then Execute Process.

new available = Available + Allocation

Else

do not execute go Forward.

	Allocation	Max	Avail	Need
P_0	0 0 1	8 4 3	3 2 2	8 4 2
P_1	3 2 0	6 2 0		3 0 0
P_2	2 1 1	3 3 3		1 2 2

Req 1 P_0 (0, 0, 2)

$P_0 < Need$ ✓

$P_0 < Avail$ ✓

Avail = 3, 2, 0

Alloc = 0, 0, 3

Need = 8, 4, 0

⇒ 0 0 3 8 4 3 3 2 0 8 4 0

6 4 0

P_1 ✓

P_2 ✗

P_0 ✗ ∴ Not Safe.

Req 2 $P_1 = (2, 0, 0)$ $P_1 < Need$

$P_1 < Avail$

P_0 0 0 1	8 4 3	1 2 2	8 4 2
P_1 5 2 0	6 2 0	3 3 3	1 0 0
P_2 2 1 1	3 3 3	8 5 3	1 2 2
		8 5 4	

P_2 ✓

P_1 ✓ ∴ Safe.

P_0 ✓

Date: __/__/____ Page no: ____

Kernel

→ core of OS.
→ backbone of OS
→ 99 % of time when we say OS is working it is actually Kernel
→ CPU scheduling, Memory Management & Device Manag. are most imp things controlled by Kernel
→ Body = CPU
Brain = Kernel.

Types

① Micro Kernel :
→ Basic functionality.
→ add. installation of prog. to operate extra ~~Support~~ features.

② Monolithic Kernel :
→ advance funct.
→ It will support add. device for which micro was demanding extra driv

Eg.

Mi Embedded OS
Smart Watches
Apple initially

↓
Linux OS.
Android (Modified Linux OS)
firefox OS
google chrome OS

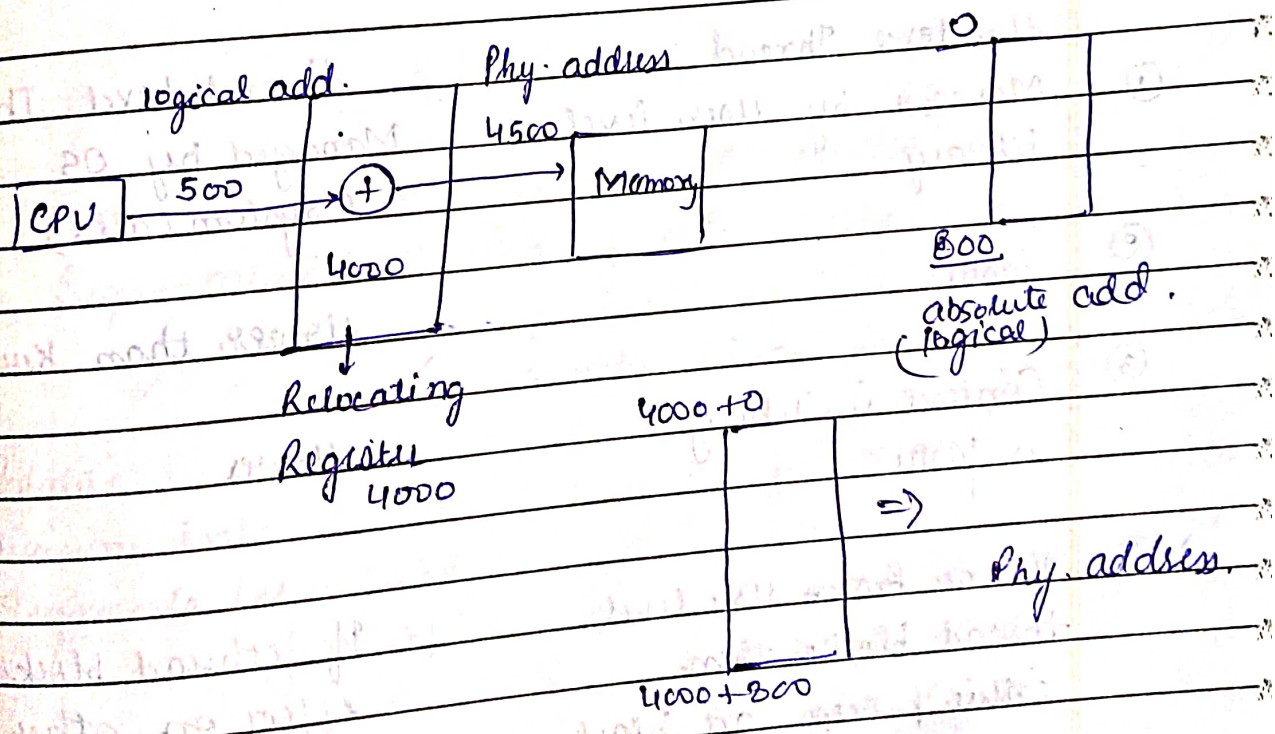
Date: / / Page no:

Windows (Hybrid Kernel)

* System Calls

Address Binding

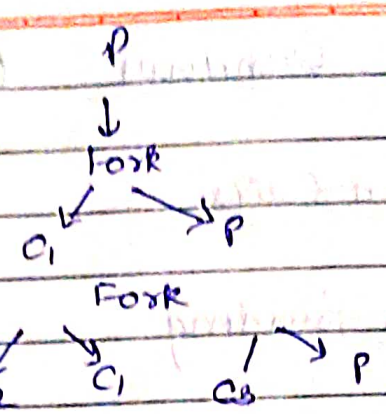
→ Processes can reside at in any part of phy. memory.
→ Addresses in source prog. are generally symbolic.
→ Compiler binds these symbolic addresses to relocatable address (14 byte)
→ Linkage editor/ loader binds these relocatable addresses to absolute addresses



Date: __/__/____ Page no: ____

* **Fork()**

0 child
1 parent
-1 no

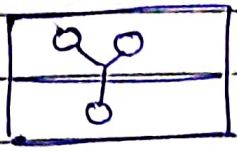


3 child.

$\therefore \frac{\text{child}}{\text{Parent}} = \frac{2^n - 1}{1}$

Thread (always share code & data)

	User level Thread	Kernel level Thread
①	Managed by user level library	Managed by OS (system calls)
②	faster	slower than Kernel level
③	Context Switching is faster	slower
④	If one user level thread blocks then entire process get block.	If thread block, NO effect on other.



hybrid Model-level.

Date / / Page no:

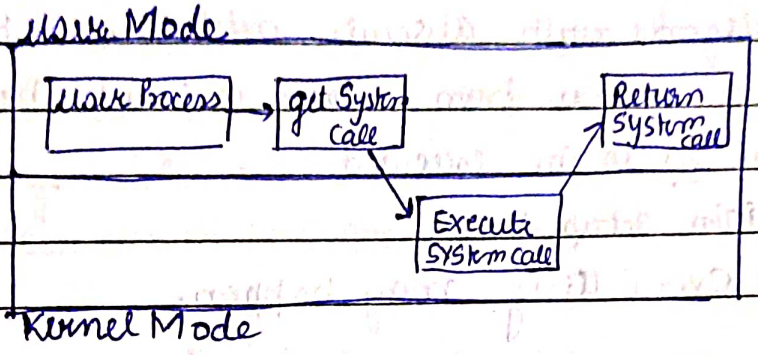
User Mode V/S Kernel Mode

↓ ↓

Initially all imp. drivers works here.

(user works here)

Processor (CPU) switches between the two



Eg. Bank!

Segmentation & Paging ← CPU don't Know about it.

↓ ↓

Relative simply divide into pages

division into

Segments (user point of view)

Memory Management.

Date: / / Page no:

Address Binding

Fetching data from Har Sec. disk to Main Memory

① Compile Time

Train ticket

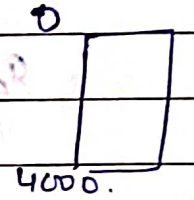
During Compile ~~works with~~ absolute address will be embedded in main code

It knows ~~where~~ from where it should be loaded & where it is to be executed.

Adv: Min setup time

Disadv: Overwritting may happen.

② Load Time genuate must, Rellocatable address code if memory location is not known at compile time.



Flight ticket (only on Board's Now Seat No)

③ Execution Time (exchanging seats in flight)

Rellocation of process from one memory ~~Space~~ Block to other during execution. (Binding delayed)

New address is allocated to process

→ Need hardware support for address Maps.

④ Dynamic Loading:

just load main module & other when they are asked out in dynamic way

⑤ Dynamic Linking: Stub (whose func. body is in which lib.)

Date: / / Page no:

Only when fun is called then only library is loaded.

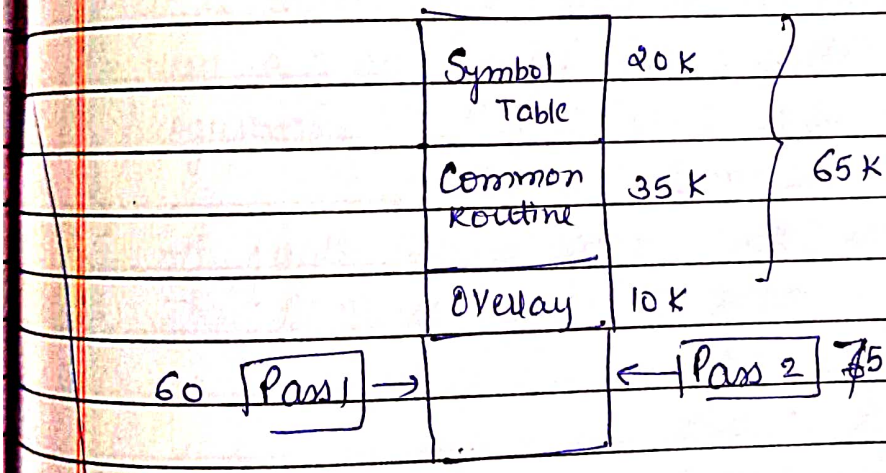
* Overlay.

Assembler translates in 2 Pass.

Eg. Pass 1 60K
Pass 2 75K
Symbol Table 20K
Common Rout. 35K
} 190K

Pass 1 & Pass 2 are mutually Exclusive

But if I have 150.



Overlay helps to execute it.

Compile Time → ① If Memory loc is known
② Recompile if add. changes

~~Execution Time~~ → Need

Date: __/__/____ Page no:

If n Resource 3 Process

	A	B	C
Demand	3	4	6

No. of Resources = (3-1) + (4-1) + (6-1) + 1

= 2 + 3 + 5 + 1 = 11

Date: / / Page no:

Booting

It is process of loading & initializing of OS

Steps

① Memory Initialization
   i) Reset CPU is active
   ii) All devices get power & initialize
   iii) Memory is power up

CPU is activated. (Works only if motherb., ROM & CPU are working)

Now it needs instructions to execute
∴ Need a non-volatile memory chip. This contains a programs K/a BIOS (Part of motherboard)
(Basic I/O System)
→ first program that runs on a computer.
→ It includes i) to configure hardware on comp.
ii) select storage device from OS when OS is to be loaded

→ It runs diagnostics & build inventory of devices K/a POST (Power on Self Test)

→ It provides API for use of other prog (Based on Priority)

POST first check BIOS then Test CMOS RAM, if no prob. then check CPU, hardware devices.
If some errors are found then error messages are displayed on screen or No. of beeps are heard. These beeps are K/a POST beep codes.

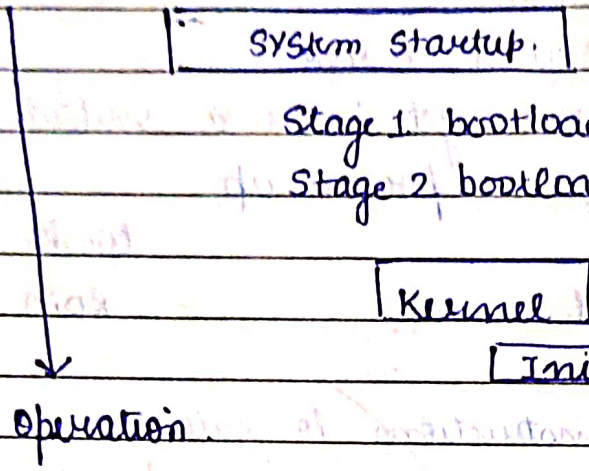
3 Files Boot ⇒ Command.com
MS-DOS.Sys
IO.Sys (Input-Output)

Loads it into RAM (windows) OS

Date: / / Page no:

① BIOS perform System Inventory
② It loads OS's Master Boot Record (MBR)
③ It runs OS Boot loader

→ Power up / Restart



2nd Master Boot Record.

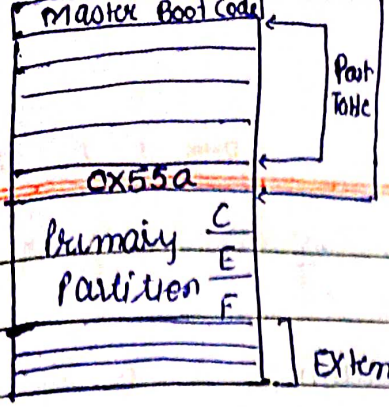
It is a small program that starts when computer is booting, in order to find OS.

It starts with POST & ends when Bios searches for MBR on Hard drive, which is generally located in first sector, first head, first cylinder.

(Cylinder 0, head 0, sector 1)

Bootstrap loader is stored in ROM or non-volatile memory.

x If POST is successful then ~~boot~~ it will load OS for computer into memory.



Date: / / Page no:

① init(c)

→ last step of kernel boot sequence.
→ used to determine initial run-level of the system.
↓
to decide initial state of OS.

Level

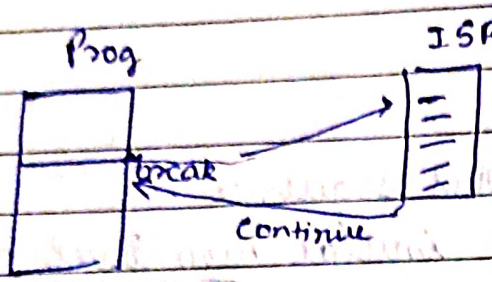
0 System Haull
1 single User Mode.
2 Full multiuser mode with network manager.
3 ' with X display.
4 5 '
6 Reboot

Next step of init is to start up various daemons that support networking & other services.
It manages display, keyboard & mouse.

Date: __/__/____ Page no: ____

Interrupt

to break sequence of operation.



controll goes to ISR.

Signals by Hard./Software when process needs immediate attention.

* ISR (Interrupt Service Routine)

In I/O devices one of the bus control lines is dedicated for this purpose & is called ISR.

⇒ when device raise interrupt

① Processor first complete execution of instruction i.
② It loads PC with address of 1st inst of ISR
③ Before loading, address of interrupted instruction is moved to temporary location. (So as to ~~start~~ continue with process (i+1)

④ while processor is handling interrupt, it must inform device that its req. has been recognized so ~~or to~~ that it stops sending interrupt req. signal.

Date: __/__/____ Page no: ____

Interrupt latency:

Time an interrupt is recevied & the start of execution of ISR.

1. device controller issues interrupt
2. Processor finishes current inst. execution
3. Processor signal ack. for interrupt
4. Proc. pushes process status onto control stack.
5. It loads new PC based on interrupt
6. save remainder of process state inf.
7. Processor executes ISR.
8. Restore process state inf.
9. Processor pops process status from control stack.

Simple Interrupt Processing.

Date: __/__/____ Page no: ____

	Paging	Segmentation
①	divided into fixed size pages.	into variable size segments
②	OS is accountable.	Compiler is accountable.
③	Page size is determined by hardware.	by user.
④	Faster	slow.
⑤	internal fragment.	external fragmentation.
⑥	OS maintains free frame list	OS maintains lists of hole in main memory.
⑦	Invisible to user.	Invisible to user.
absolute address	Page No + offset	Segment No + offset

Modern OS have 2 memory abstractions where segmentation is used for "handling the programs"
Paging for "Managing physical Memory"

Date: / / Page no:

→ 64 bit processor can have 64 or 32 bit version of OS.

However 32 bit OS, 64 bit processor would not run at its full capability.

Adv. of 64 bit over 32

- can do a lot of multi-tasking
- gamers can play high graphical games.

However upgrading video card instead of going 64 bit processor would be more beneficial

⇒ 32 bit system = 2^{32} memory addrss
4 GB of RAM or physical Memory

64 2^{64}
18 Quintillion bytes

* Major diff is No. of calculations per second they can perform, which affects speed at which they can complete task.

64 bit can be dual core, quad core, six core.

CNT

Date: __/__/____ Page no: ____

	MAC address	IP address
①	Media Access Control Address.	Internet Protocol Address.
②	ensure physical address of computer is unique.	Logical address uniquely locate computer connected via network.
③	6 byte hexadecimal address 128 bits	4 byte or 16 bytes 32 bits
④	Retrieved using ARP protocol	RARP protocol.
⑤	Chip maker manufacture provides MAC address.	ISP, Internet Service provider provides IP addr

Logical Address	Physical Address
① generated by CPU	generated by MMU. It is location in MMU Memory unit
② User can view logical add.	Never view phys. add.
③ Logical address can be used to access phy. add.	User can indirectly access phy. address but Not directly
④ Do not exists physically in memory (Virtual address)	exists.
⑤ generated at Compile-time & Load time address binding.	generate at run time address binding.

Logical Address Space : (define size of process)
It is ~~used for~~ set of all logical addresses generated by program's perspective.

Physical Address Space : (define size of Main Memory)
set of all physical addresses mapped to corresponding logical address.

Date: __/__/____ Page no: ____

Inverted Page Table

→ alternate to multi-level paging schemes

→ No. 1 Page Table Entry for every frame of main memory.

∴ No. of Page Table Entries = No. of frames in Phy. Memory

Single page Table is used to represent paging inf. of all processes.

→ overhead of storing individual page table for every process get eliminated & only fixed partition of memory is req. to store paging inf. of all processes together.

→ Indexing is done wrt. frame No. instead of logical page No.

Page No	Process id	Control bit	Chained Pointer

* Reduce Memory Space
* Longer Lookup time
* difficult shared memory implementation

Date: / / Page no:

what happens when a command is executed in shell ?

Command ⇒ shell ⇒ convert to ⇒ Kernels ⇒ H/w.
Eg (BASH) binary

shell converts command to binary.

GCC → GNU Compiler Collection

It is integrated distribution of compilers for several programming languages.

High L. lang → Machine Language.

Steps

① Preprocessing :

Removal of comments,
Expansion of macros, included files.

< > preprocessor directory.

② Compilation

take preprocessed file as input, compile it & produce intermediate compiled output
output is Assembly code which is machine dependent

③ Assembly :

Computer understands only binary. So Assembler converts assembly code to binary code

* Object & Executable file come in several formats such as ELF (executable & linking Format) & COFF (Common Object File format)
↑ windows
↑ linux

(4) Linking :

Linking of func. calls with their def. is done.

Assembler left address with actual of functions to be called & linker does final process of filling in addresses with actual definitions.

(1) gcc -E Hello.c -o HelloOutput

(2) gcc -S Hello.i -o Hello.s

(3) gcc -c Hello.c -o Hello.o

(4) gcc. -o Output Hello.c

Phases of Compiler

① Lexical Analysis : (symbol table)

① Identify lexical unit in source code.
② classify lexical unit
③ Ignore Comments

X = Y + 10	
X	Identifier
=	Operator
Y	Identifier
+	Operator
10	Number

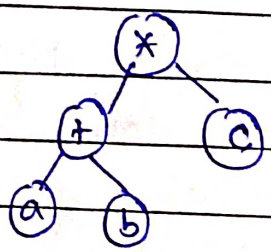
② Syntax Analysis

It is about discovering structure in ~~node~~ code.

It make sure that source code was written by programmer is correct or not.

① Obtain tokens from lexical analyzer.
② check if expression is syntactically correct or not
③ Report syntax error.
④ Construct parse tree.

(a+b) * c



Date: / / Page no:

③ Semantic Analysis:

check consistency of the code.

- allow type checking.
- if types mismatch → semantic error.

```
float x = 20.2
float y = x * 30;
```

Semantic analyzer will typecast int 30 to float 30.0.

④ Intermediate code generation.

It is b/w High level & Machine level language.
It needs to be generated in such a manner that makes it easy to translate it into target Machine code.

```
total = count + rate * 5
```

⇒ t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3

⇒ t1 := rate * 5.0
⇒ total := count + t1

⑤ Code optimization.

Removes unnecessary code & arrange seq. of statements to speed up execution of prog. without wasting resources.

Date: __/__/__ Page no: ____

⑥ Code generation.

Produce page & object code as a result.

* Symbol Table contains a record for each identifier with fields for attributes of identifier.

* Error Handling.

Lexical An. : Wrongly spelled token.
Syntax An. : Missing Parenthesis
Intermed. C. g. Mismatched Operand for operator
Code opt. When system is not reachable.
Code gen. Unreachable statements.
Symbol Table Error of multiple declared identifier.