Operating Systems (Code Snippets)

1) Operating system → It is an interface between user and computer hardware.

Types of operating system →

1) Batch OS →
A set of similar jobs are stored in main memory for execution. A job gets assigned to CPU, only when execution of previous job completes.

2) Multiprogramming OS →
The main memory consists of jobs waiting for CPU time. The OS selects one of the process & assign it to CPU. Whenever executing process needs to wait for any other operation (like I/O), the OS selects another process from job queue and assigns it to CPU. This way, the CPU is never kept idle and user gets flavour of getting multiple tasks done at once. (Non-preemptive)

3) Multitasking OS →
Multitasking OS combines the benefits of Multiprogramming OS and CPU scheduling to perform quick switches between jobs. The switch is so quick that user can interact with each program as it runs. (preemptive)

Same thing

4) Time sharing OS →
Time sharing systems require interaction with user to instruct the OS to perform various tasks. The OS responds with output. The instructions are usually given through input device like keyboard

Real Time OS :
are usually built for dedicated systems to accomplish a specific set of tasks within deadlines.

Threads

A thread is a lightweight process & forms a basic unit of CPU utilization. A process can perform more than one task at same time by including multiple threads.

1) A thread has its own program counter, register set & stack
2) A thread shares resources with other threads of same process the code section, the data signal, files & signals

2 types of thread

1) User thread

→ implemented by user

→ context switch time is less

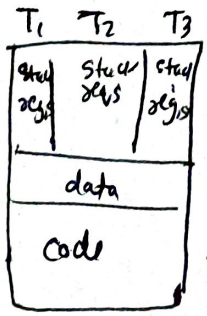
→ If one user level thread is blocked the entire process will be blocked

2) Kernel thread

→ implemented by OS

→ context switch time is more

→ if one kernel thread performs blocking operation then another thread can continue execution



→ threads share data, code etc but to perform multitasking use different stack, registers

} Shared

→ context switch → switching both T_1 & T_2 etc

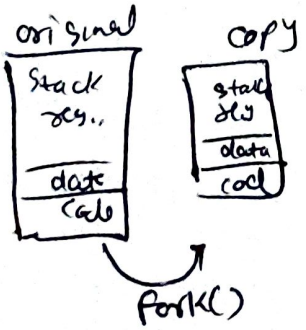
→ if like T_1 wants input before working so it perform blocking operation, thus if T_1 blocked all process stop in

Process

A process is program under execution. The value of program counter (PC) indicates the address of next instruction of process being executed. Each process is represented by a Process control Block (PCB)

Process

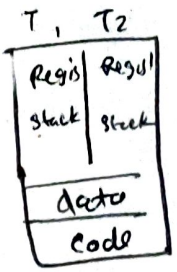
- → system calls involved in process (system call decreases speed
- → OS treats different process differently
- → Different process have different copies of Data files, code
- → context switching is slower
- → Independent



Threads

(User level) here

- → no system call involved
- → All user level threads treated as single task for OS.
- → thread share same copy of code and data
- → context switching is faster
- → Interdependent



A process with n fork() system calls generate $2^n - 1$ child process.

→ basically thread is segment of process.

Process Scheduling

Important times with respect to process

1) Arrival time → Times at which process arrives in ready queue
2) Completion time → Time at which process completes its execution.
3) Burst time → Time required by a process for CPU execution
4) Turn Around time → Time difference between completion time and arrival time.

Turn around time = Completion Time − Arrival Time

5) Waiting Time (WT) = Time difference between turn around time and burst time.

Waiting Time = Turn around time − Burst time

→ Why do we need scheduling?

A typical process involves both I/O time & CPU time. In a uniprogramming system like MS-DOS time spent waiting for I/O is wasted and CPU is free during this time. In multiprogramming systems, one process can use CPU while another process is waiting for I/O. This is possible only with process scheduling.

OBJECTIVE of Process Scheduling Algorithm

1) MAX CPU utilization
2) Fair allocation of CPU
3) MAX throughput (no. of process that complete their execution per time unit)
4) Min turnaround time (time taken by process to finish execution)
5) Min waiting time (time for which process waits in ready queue)
6) min response time (time when process produces first response)

Different scheduling Algorithms

(1) (FCFS) First come first serve :

Simplest scheduling algorithm that schedules according to arrival times of process.

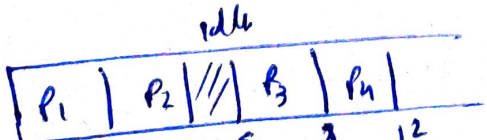
is non-preemptive (scheduling is not interrupted in middle of execution) means it switch to other process or terminate process only after completion of older process)

eg

Process	Arrival time	Burst time	Completion time ← by Gantt chart	Turnaround	WT
P_1	0	2	2	2 (2-0)	0
P_2	1	2	4	3 (4-1)	1
P_3	5	3	8	3 (8-5)	0
P_4	6	4	12	6 (12-6)	2

* waiting time (WT) = Turnaround − Burst time

Gantt chart



ready queue is empty

only P_2 in ready queue

2) Shortest Job First (SJF)

Process which have shortest burst time are scheduled.

Mode → non-preemptive (cannot be stopped in betn)

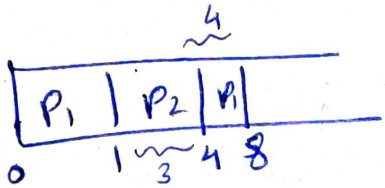
- from ready queue select one with lowest burst time

3) Shortest remaining Time First (SRTF)

It is preemptive (can be interupted) mode of SJF algorithm in which jobs are scheduled according to shortest remaining time.

eg

	Arrival	Burst time
P_1	0	~~5~~ 4
P_2	1	3



P_1 P_2 so start P_2

4 3

Burst time

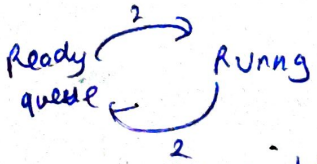
4) Round Robin (RR) scheduling

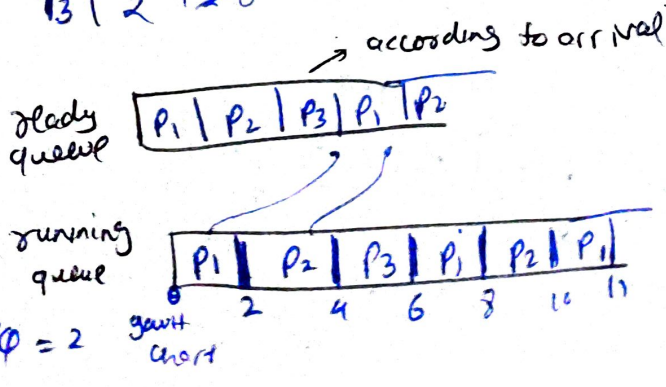
Each process is assigned a fix time in cyclic way.

Criteria = Time quantum Mode → Preemptive

eg Let TQ = 2

	Arrival	Burst
P_1	0	~~5~~ → 3 → 1
P_2	1	~~4~~ → 2
P_3	2	2 → 0





5) Priority Based Scheduling
(Non Preemptive)

In this scheduling, processes are scheduled according to priorities ie highest priority process is scheduled first. If priorities of two process match, then scheduling is according to arrival time.

6) Highest Response Ratio Next (HRRN) Non Preemptive

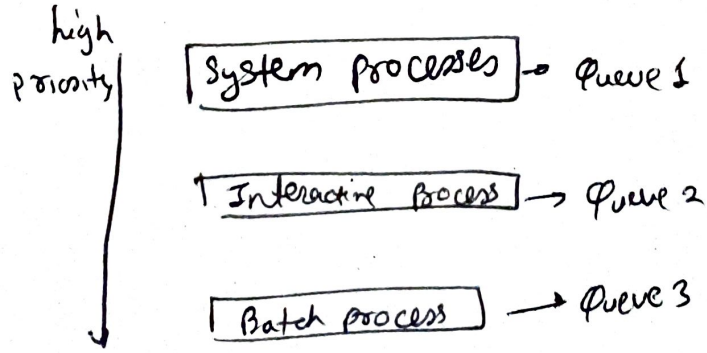
Process with highest response ratio is scheduled. This algorithm avoids starvation

$$\text{Response Ratio} = \frac{\text{Waiting Time} + \text{Burst time}}{\text{Burst time}}$$

7) Multilevel Queue Scheduling (MLQ)

According to priority of process, processes are placed in different queue, Generally high priority processes are placed in top level queue. Only after completion of processes from top level queue, lower level queued process are scheduled

eg



All three different type of processes have their own queue. Each queue have its own scheduling algorithm.

eg queue 1 & queue 2 → Round Robin
queue 3 FCFS

8) Multi level Feedback Queue (MLFQ) scheduling

It allows process to move in between queues. The idea is to seperate processes according to characteristics of CPU bursts. If process uses too much time, It is moved to lower priority queue

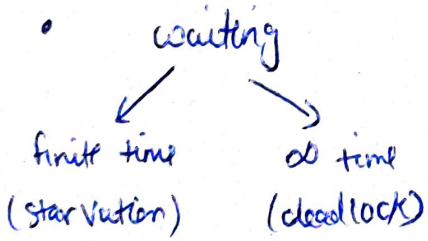
important points

→ FCFS can cause long waiting time, especially when first job takes too much CPU time.

→ Both SJF and shortest remaining time first algorithms may cause Starvation

→ If time quantum for Round Robin scheduling is very large then it behaves same as FCFS scheduling.

→ SJF is optimal in terms of average waiting time for given set of process. SJF gives min avg waiting time but problem is here to predict time of next job

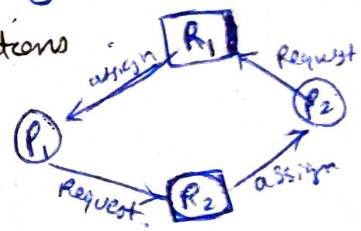


Deadlock

a situation where set of process are blocked because each process is holding a resource & waiting for another resource acquired by some other process. Deadlock can arise if following conditions hold simultaneously.

1. Mutual exclusion → One or more than one resource are non-shareable
2. Hold & wait → A process is holding at least one resource and waiting for resources.
3. No preemption → A resource can not be taken from a process unless process releases the resource.
4. Circular wait → Set of processes are waiting for each other in circular form.

↑ 4 necessary conditions for deadlock.



Method of handling deadlock

1) Deadlock prevention & avoidance → the idea is to not let system into deadlock state.

2) Deadlock detection & recovery → Let deadlock occur, then do preemption to handle it once occur. or kill the process.

3) Ignore the problem all together → If deadlock is very rare, then let it happen & reboot the system. This is approach that both windows & UNIX take.

eg of Deadlock → Process 1 is holding resource 1 & waiting for resource 2 which is acquired by process 2 & process 2 is waiting for resource 1.

Bankers algorithm

is resource allocation & deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources.

(deadlock avoidance) & (deadlock detection)

eg → A = 10, B = 5, C = 7 ← (resources) Say it is safe or unsafe.
(deadlock occur)

Process	Already Allocated			Max need			Available			Remaining need (max-allocated)			
	A	B	C	A	B	C	A (10→7)	B (5→2)	C (7→5)	A	B	C	
P_1	0	1	0	7	5	3	3	3	2	7	4	3	P_1 ✓4
P_2	2	0	0	3	2	2	5	3	2	1	2	2	P_2 ✓1
P_3	3	0	2	9	0	2	7	4	3	6	0	0	P_3 ✓5
P_4	2	1	1	4	2	2	7	4	5	2	1	1	P_4 ✓2
P_5	0	0	2	5	3	3	7	5	5	5	3	1	P_5 ✓3
	7	2	5				10	5	7	→ initial = final			

(no deadlock)

Safe sequence → order in which (process) sequence occurs.

- Eg available 332 & one process with less than this is P_2
  So first P_2 is executed after execution it leaves its resources (200)
  so now available becomes (3+2) (3+0) (2+0) → 5 32
  with 532 P_4 can occur & release 211 so new available → 743
  (Remaining need < Available).

Safe sequence → P_2, P_4, P_5, P_1, P_3 — (safe).

↑ works in line after P_4 can do all P_1 P_3 P_5 but after P_4 P_5 comes so write it.

But all sequence are allowed.

Q A system is having 3 process each require 2 units of resources 'R'. The minimum no. of units of 'R' such that no deadlock will occur. (13)

a) 3 b) 5 c) 6 d) 4. ✓

eg 3 process P_1 P_2 P_3

Let if 2 units of R. 1 1

So deadlock. P_1, P_2 need 1 more & P_3 need 3.

- if R = 2

P_2 P_3

→ this we can control —— not this

runs then deallocate but here at one case P_2 P_3 → deadlock

1 1

(we should have all situations deadlock free).

- if R = 3.

P_2 P_3

1 1

runs & free resource →

P_3 1

but for case

P_1 P_2 P_3

1 1 1

deadlock

- if R = 4.

P_2 P_3

1 1

P_2 P_3

1 1

both run

(all case deadlock free)

So no deadlock.

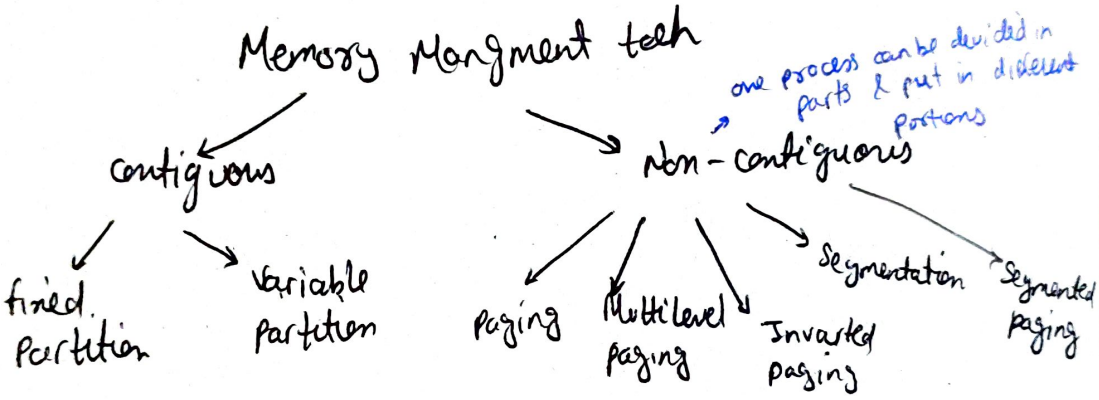
Memory Managment Techniques :-

here memory = primary memory

Intro :-

These Techniques allow memory to be shared among multiple processes.

- overlays :- The memory (RAM here) should contain only those instructions and that data which is required at given time.
- Swapping :- In multiprogramming, the instructions that have used the slice are swapped out of memory.



⇒ Single partition Allocation scheme

The memory is devided into 2 parts, One is kept to be used by OS and other is kept to be used by users.

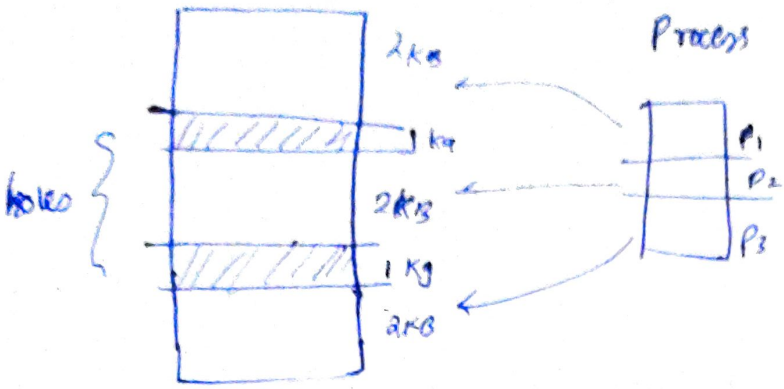
⇒ Multiple Partition Scheme

1) Fixed Partition → Memory is devided into fixed size partition
2) Variable Partition → Memory is devided into variable size partation

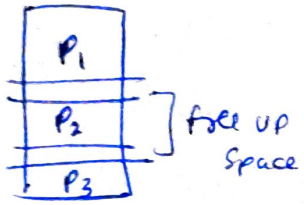
Variable partition allocation schemes :-

1. First fit → arriving process is alloted first hole of memory in which it fits completly.
2. Best fit → arriving process is alloted the hole of memory in which it fits the best by leaving minium memory empty.
3. Worst fit → The arriving process is alloted the hole of memory in which it leaves the maximum gap.

- In non contiguous process is devided into different portions while allocation

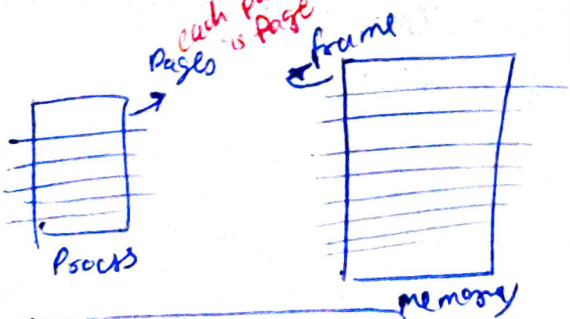


But now holes are dynamically alloted & after again & again let holes becomes empty now



to solve this external fragmentation we do Paging

here we devided process in equal size and also to memory (RAM) in equal size before allocation



Page Size = frame size

new process coming also devided into pages of same size

here occurs external fragmentation

→ The unused spaces formed between non-contiguous memory fragments are too small to serve a new process request. This is called external fragmentation.

→ It refers to unused memory blocks that are too small to handle a request.

Internal fragmentations

- The difference betn memory allocated & the required memory is called internal fragmentation
- It occurs when main memory is devided into fixed-size blocks regardless of size of process.
- It refers to unused space in partition which resides within an allocated region, hence the name

Need of Paging →

The cause of external fragmentation is the condition in Fixed partitioning and variable partitioning saying the entire process should be allocated in contiguous memory location. Therefore paging is used to remove external fragmentation.

1. **Paging →**

Physical memory is devided into equal sized frames. The main memory is devided into fixed size pages. The size of physical memory frame is equal to size of virtual memory frame.

2. **Segmentation →**

Segmentation is implemented to give users view of memory. The logical address space is a collection of segments. Segmentation is implemented with or without the use of paging.

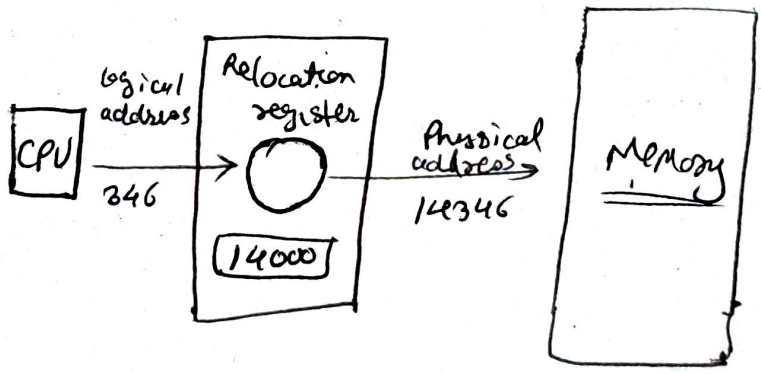
Non Contiguous Allocation

Paging & segmentation are 2 ways which allow a process's Physical address space to be non-contiguous. It has advantage of reducing memory wastage but it increase overheads due to address translation. It slows the execution of memory because time is consumed in address translation.

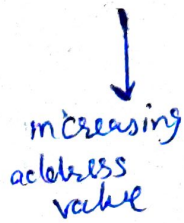
Logical & Physical address

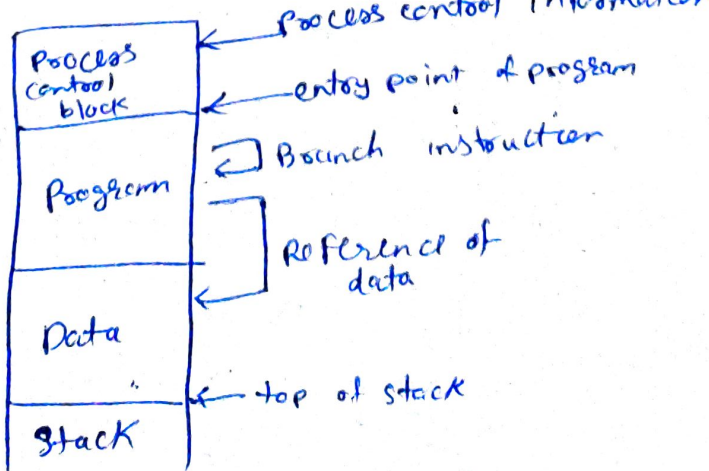
1) Logical /virtual address → Logical address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically. This address is used as reference to access the physical memory location by CPU. User can only view logical address of program. The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

2) Physical Address → identifies physical location of required data in memory. User never directly deals with physical address but can access by its corresponding logical address. Logical address space must be mapped to physical address by MMU before they are used.



Process look like





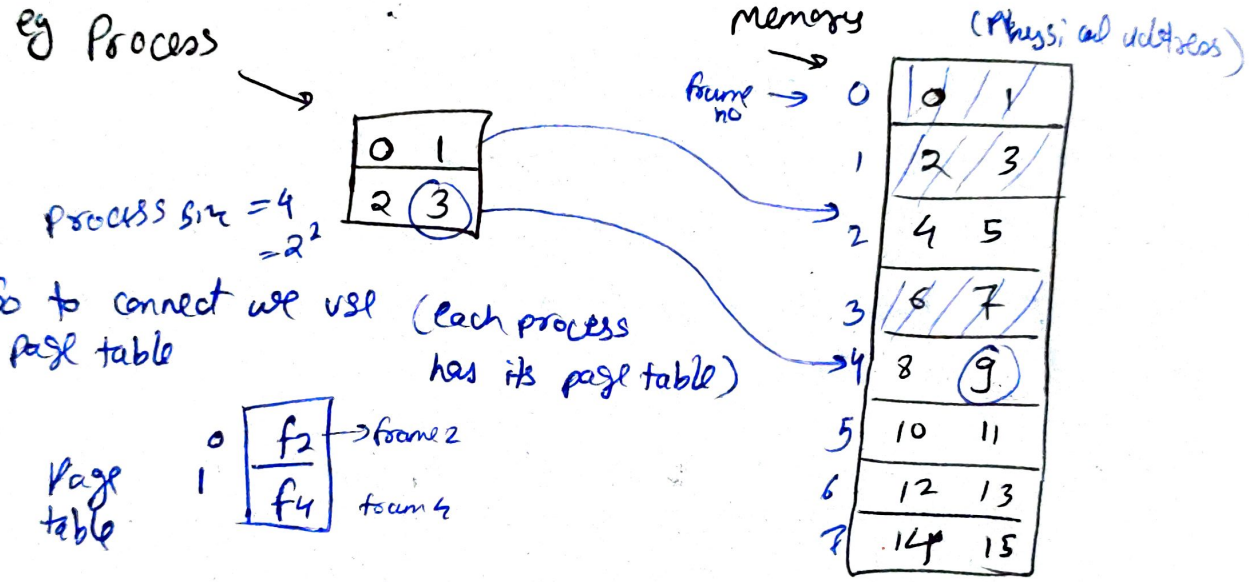
How paging Works?

Paging is memory managment scheme that eliminates need for contiguous allocation of physical memory. This scheme permits physical address space of process to be non-contiguous.

→ The physical address space is devided into number of fixed size blocks called frames.

→ Logical address space is also splitted into fixed-size blocks called pages.

Page-size = Frame-size



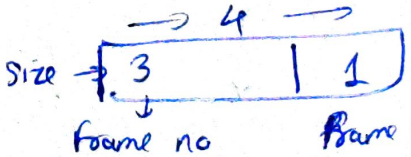
Page address (4 bits 2^2, 2 bits)

Main memory size = 16B (2^4)

Let frame size = 2 = Block size

No of frames = $\frac{16}{2}$ = 8 (2^3)

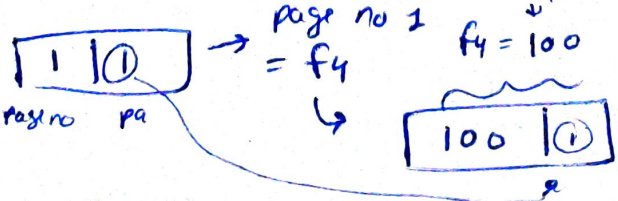
→ Physical address (16 bit



now

Let want location of 3

3 = 11



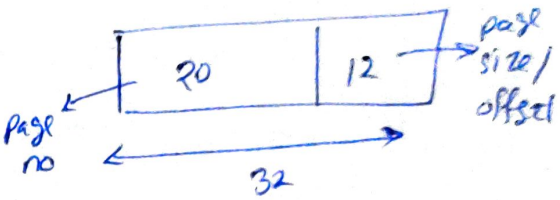
Q. Given Logical address space = 4GB, Physical address space = 64MB
Page size = 4KB, No. of pages = ? = 2^{20}
No. of frames = ? = 2^{14}
No. of entries in page ? 2^{20}
Size of page table = ?

Note
1K = 2^{10}
1M = 2^{20}
1G = 2^{30}
1T = 2^{40}

Note :- Memory is byte addressable
So convert space to byte not bit

⊙ Logical address space (relative to page) ⟶ logical address
4GB ⟶ $2^2 \times 2^{30}$ Byte = 2^{32}

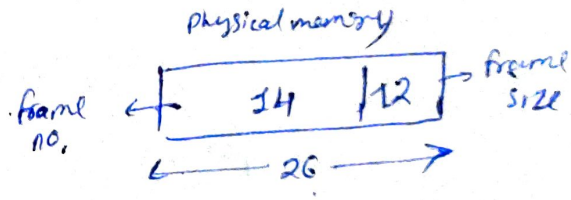
Page size = 4KB = $2^2 \times 2^{10}$ Byte
= 2^{12} Byte



⊙ So no. of pages = 2^{20} (no. of pages)
= 2^{20}

⊙ Physical address space 64MB = $2^6 \times 2^{20}$ Byte = 2^{26} Byte

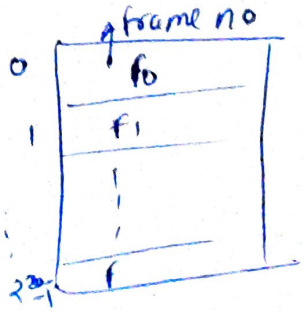
frame size = page size = 12



⊙ No. of frame = 2^{14}

⊙ No. of enteries in page = no. of pages in process
= 2^{20}

⊙ Size of page table = No. of pages × Size of frame
as in page table
= $2^{20} \times 14$ bits



Segmentation vs Paging

Paging

- In paging, program is devided into fixed size pages.
- For this OS is accountable
- Page size is determined by hardware
- Is faster
- Paging could result in internal fragmentation
- logical address is split into page number & page offset.
- Paging is invisible to user

Segmentation

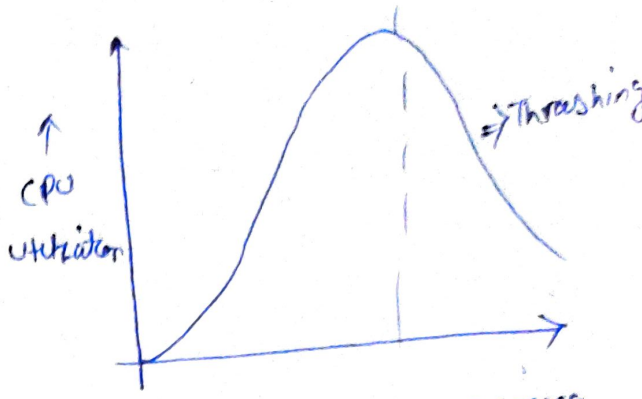
- In this, program is devided into variable size sections.
- For this compiler is accountable
- Section size is given by user.
- Segmentation is slow
- Segmentation could result in external fragmentation
- logical address is split into Section number & section offset.
- Segmentation is visible to user.

Segmentation

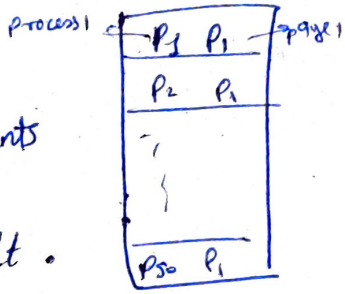
Segmentation is another non-contiguous memory allocation scheme like paging. Like paging In segmentation, process isn't devided into fixed size pages. It is variable size partitioning theme. Like paging, in segmentation, secondary & main memory are not devided into partitions of equal size. The partition of secondary memory area unit known as segments.

THRASHING

- Page fault → page of a process is not in RAM
- Degree of multi programing :-
  no. of process in RAM.
- full process is not placed in RAM but a part of it called page is put in RAM.



for max degree of multi programing we place Page 1 (P_1) of all 50 process



if Page fault occurs, page fault servic takes time.

→ but let our CPU wants page 2 of P_1 so it generate page fault.

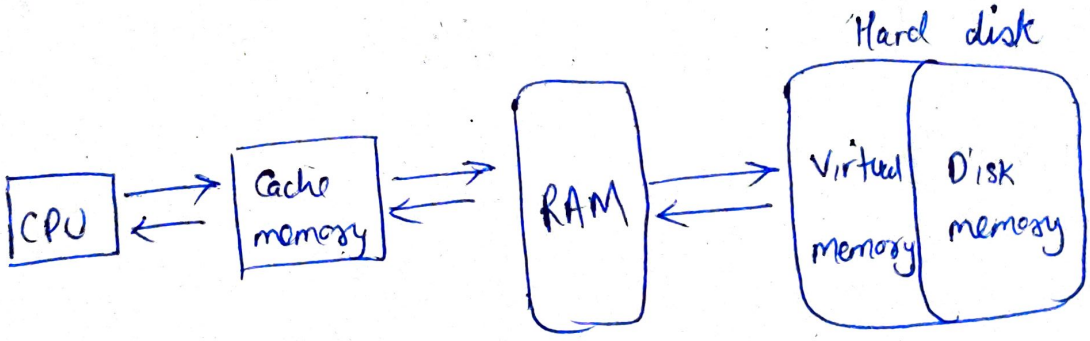
If similarly other pages of the process are called, so many page fault occurs & to solve them CPU utilization decreases. This is called thrashing.

Virtual Memory

Virtual memory is a technique that allows the execution of process that may not be completely in memory. In this only a portion of virtual address space of resident process may actually be loaded into physical memory. As a consequence, sum of virtual address spaces of active processes in a virtual-memory system can exceed the capacity of physical memory provided that the physical memory is large enough to hold a minimum amount of address space of each active process.

for ex. 1 M program can run on a 256 K machine by choosing which 256 K to keep in memory at each instance, with pieces of program being swapped between disk & memory as needed.

- Virtual memory is just a extension of main memory (hard disk)



Virtual memory solves problem of insufficient memory by converting a part of disk memory into virtual address there by creating a large size of RAM to accomodate the increased demand for memory requirement.

Virtual memory can be done with paging or with segmentation

Demand paging

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped all at once. Rather they are swapped in only when process needs them (on demand) This is termed as lazy swapper.

- effective access time.

Let normal memory access requires 200 nanoseconds, & serving a page fault takes 8 milli seconds (8,000,000 nanoseconds). with page fault rate of p (0 to 1 value) effective access time (EAT) is

$$EAT = (1-p)^* (200) + p^* 8000000$$
$$= 200 + 7999800^* p.$$

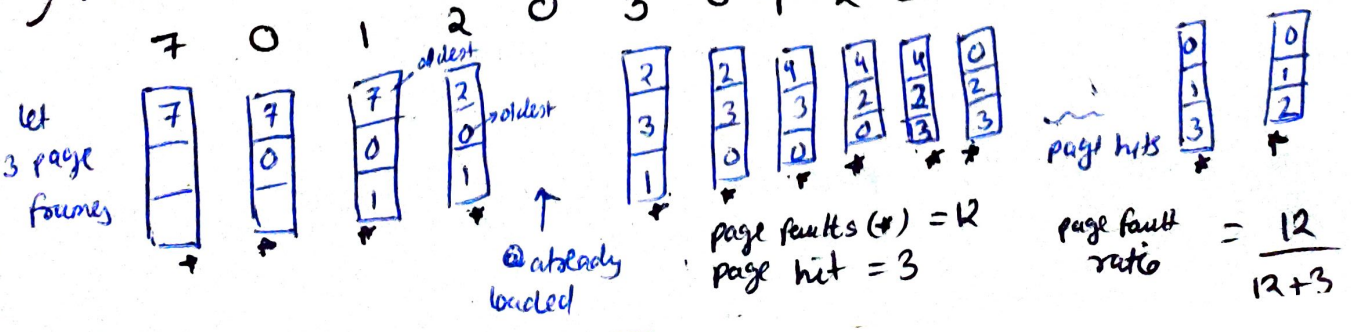
COPY ON WRITE (COW)

idea behind copy on write fork is that pages for a parent process do not have to be actually copied for child until one or other of the processes changes the page. They can be simply shared betn the two process in meantime.

Page replacement

In order to make most use of virtual memory, we load several processes into memory at same time. Since we only load pages that are actually needed by process at any given time, there is room to load many more processes than if we had load in entire process

1) FIFO page replacement.

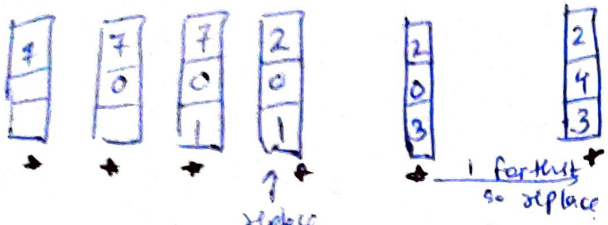


- An interesting effect that can occur in FIFO is **Belady's anomaly** in which increasing number of frames available can actually increase the number of page fault that occur.

2) Optimal Page Replacement

- Replace the page that will not be used for longest time in future
→ see in future so not practical, but has lowest page fault ratio.

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

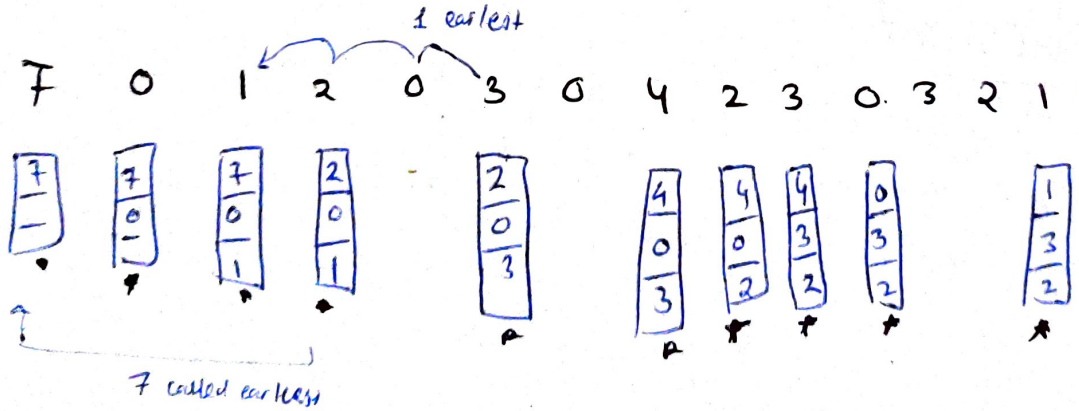


8 page fault
7 page hit

(pf. ratio = 8/15)

3) LRU page Replacement

- Least recently used
- See in page, replace one which is called earliest. (replace oldest seen one)

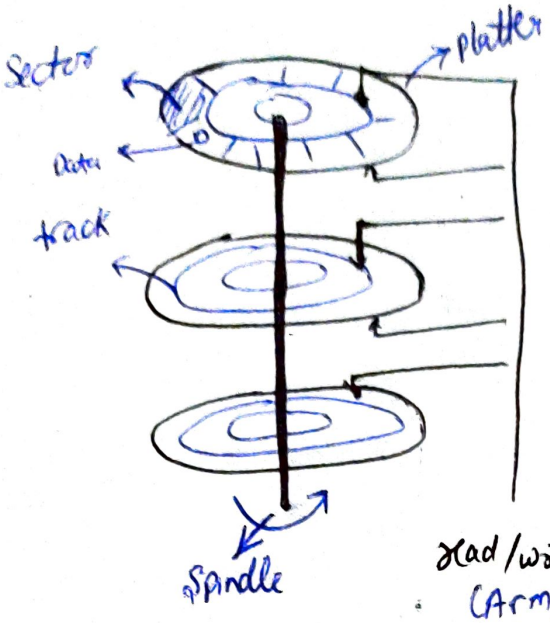


10 page fault
5 page hit

page fault ratio = 10/15

Best → Optimal page replacement > LRU > FIFO

Disk Architecture



Head/write arm
(Arm)
move forward, backward.
is present on both side of platter
surfaces.

Platter → surface → Track → sector → Data

(2 surfaces in platter)

$1K = 2^{10}$
$1M = 2^{20}$
$1G = 2^{30}$
$1T = 2^{40}$

Disk Size = Platter × Surface × Track × sector × Data.
(2) (Tracks in a surface)

• Disk Access time

1) Seek time : Time taken by R/w head to reach desired track. (Avg value)
2) Rotation time : Time taken for one full rotation (360)
3) Rotational latency : Time taken to reach to desired sector (half of rotational time)
4) Transfer time : $\frac{\text{Data to be transferred}}{\text{Transfer rate}}$.

Transfer rate : (Data rate) = {No. of Heads × Capacity of one track × No. of rotation in 1 sec}

• Disk Time = seek time + Rotation time + Transfer time + CT (Controller time if given) + QT (queue time wait)

CT → optional
QT → optional

Disk Scheduling algorithm

Goal : To minimize seek time

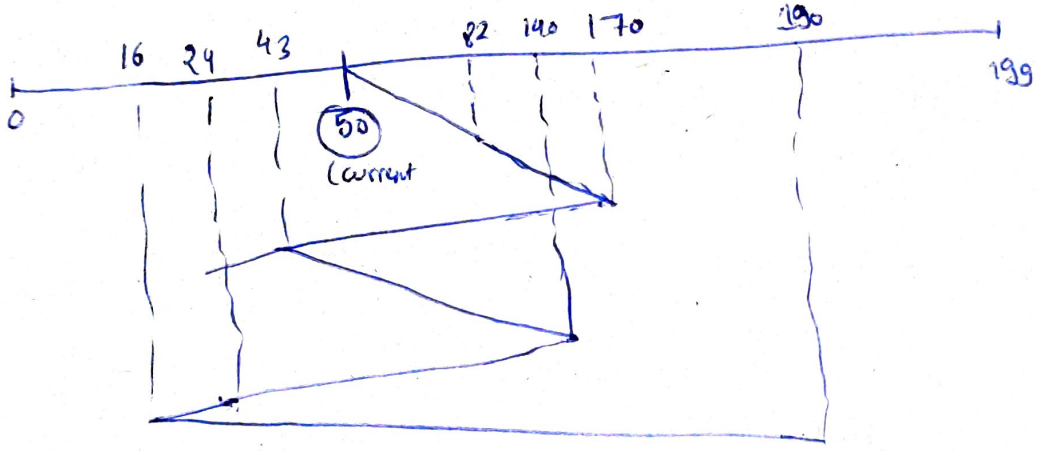
1) FCFS 2) SSTF (Shortest seek time first) 3) SCAN
4) LOOK 5) CSCAN (circular Scan) 6) CLOOK (circular look)

1) FCFS

Q A disk contains 200 tracks (0-199). Request queue contains track no. 82, 170, 43, 140, 24, 16, 190 respectively. Current position of R/w head = 50. Calculate total no of tracks movement by R/w head.

↑ ≈ seek time

Let visualize circular track as



movement = (82-50) + (170-82) + (170-43) + (140-43) + (140-24) + (24-16) + (190-16)

= 642

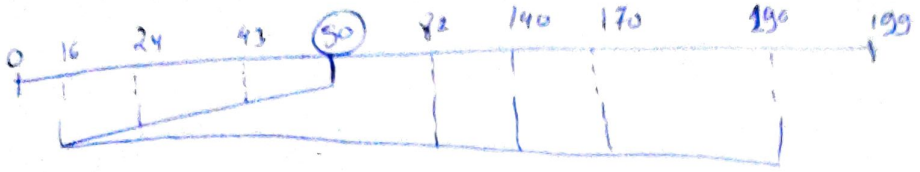
(here there is no starvation)

but performance is less.

2) SSTF

Same question with Shortest seek time first

also → if R/W head takes 1 ns to move from one track to another the total time taken — ?



Time taken = (50 − 16) + (190 − 16)

= 208

- generate overhead (to know all requests)

- Response time is less
- have starvation

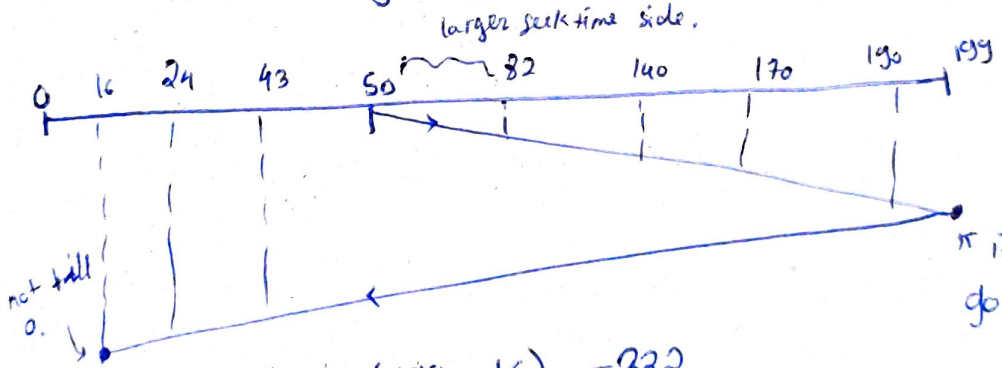
eg 40, (61), 30, 20, 10 (starvation)

(50) → has to wait more

3) SCAN

Same question with SCAN.

- from current move to one with larger gap, then cover all at that direction. but it goes till end of that side. eg not till 190 but 199

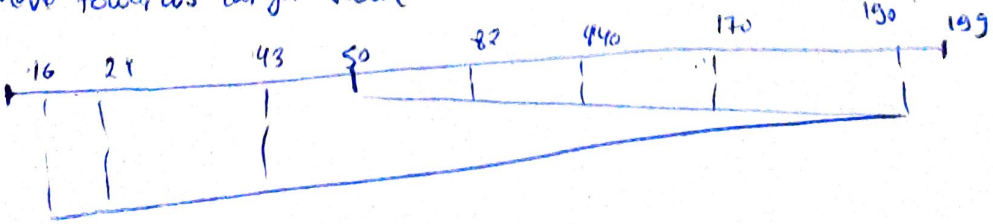


time = (199 − 50) + (199 − 16) = 332

4) LOOK

Like SCAN but not till end

1) move towards larger value

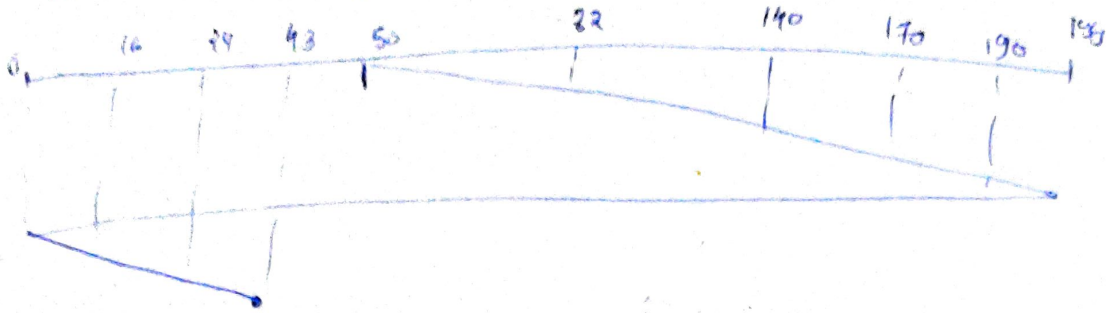


time = (190 − 50) + (190 − 16)

= 314.

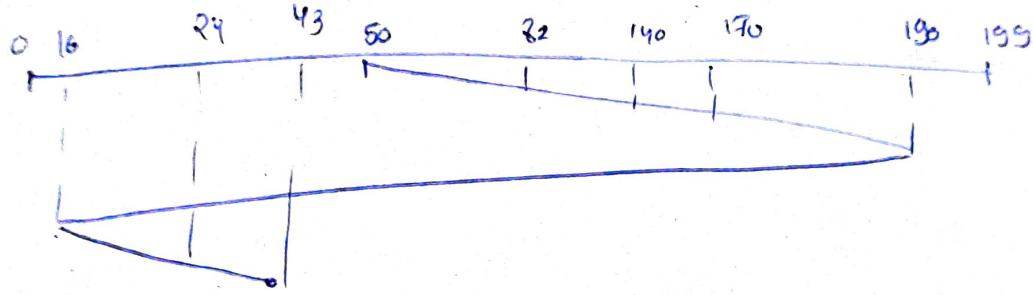
5) C-SCAN

1) move towards larger value



time $(199-50) + (199-0) + (43-0) = 391$

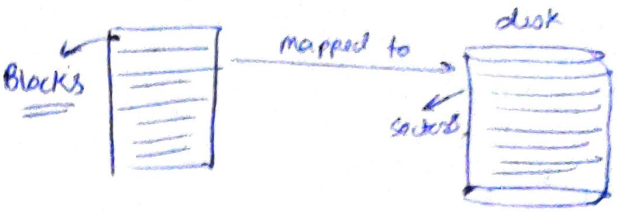
6) C-LOOK



$(190-50) + (190-16) + (43-16) = 341$

FILE SYSTEM

- User sees → file in → folder / directory → managed by → Filesystem



- Operations on files

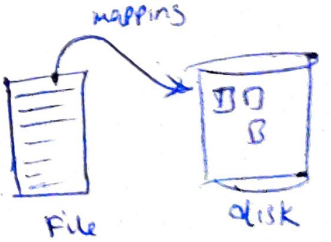
1) Creating 2) Reading 3) Writing 4) Deleting
5) Truncating 6) Repositioning

- File attributes

1) Name 2) Extension (Type) 3) Identifier (special number given to file)
4) Location 5) Size 6) Modified date, created date
7) Protection / Permission 8) Encryption, compression.

- File Allocation Methods

→ Contiguous
→ Non Contiguous
  - → Linked list allocation
  - → Indexed allocation

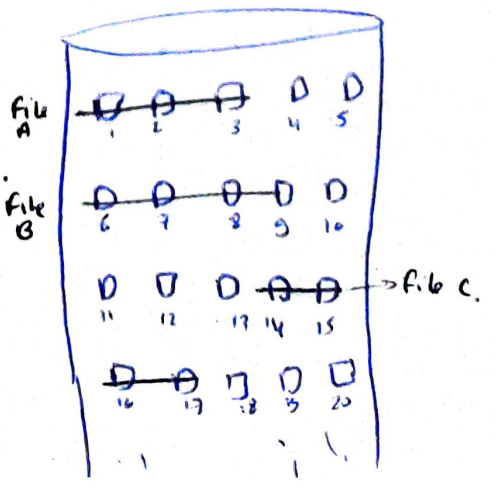


1) Contiguous Allocation

Directory

file	start	length
A	0	3
B	6	5
C	14	4

length 3 = 3 blocks



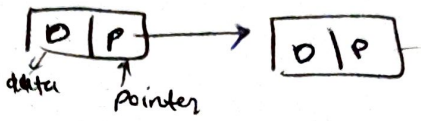
2) Non Contiguous Allocation

• Linked list allocation

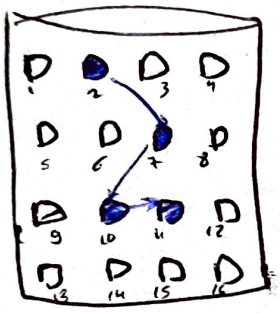
here in directory

file	start
A	2.

& in block it is saved as



So



so file is like

⊙ 2 → 7 → 10 → 11

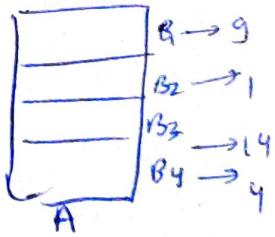
file A

- no external fragmentation
- file size can increase

Problem → large seek time
→ overhead of pointers

• Indexed file Allocation

here a block is used as Index block, which contains index of all block - eg

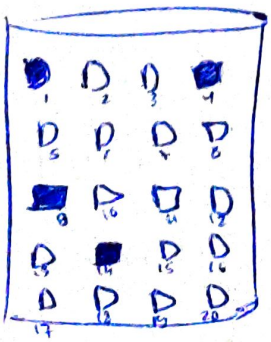


Directory

file	Index Block
A	6

data in 6

B1 - 9
B2 - 1
B3 - 14
B4 - 4



Disadvantage
→ pointer overhead
→ multilevel index

Advantage
→ Support direct access (eg direct can see B3 without B1)
→ No external fragmentation.

Process Synchronization

Process

- cooperative process
  - share variable, memory, code, resources.
  - (execution of one affects other process)
- independent process

• Printer - spooler problem

Printer is slow. So whatever to be printed is first stored in spooler directory.

Let see problem.

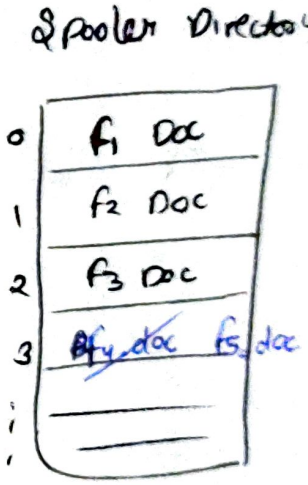
code →

I_1. Load Ri, m[in]

I_2. Store SO[Ri], "filename"

I_3. Increment Ri

I_4. store m[in], Ri



In [3]

Will fill f3 Doc

In [3] → place 3 is next.

now let 2 parallel process comes

P1 → f4.doc

P2 → f5.doc

now let P1 runs first P1 I1 I2 I3 | P2 I1 I2 I3 I4

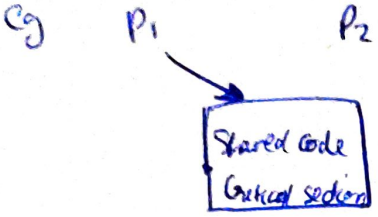
prempt.

- since for P1 I4 not runs so In[3] & P2 overwrites in that and hence we loose f4.doc

Critical section → it is part of the program where shared resources are accessed by various processes.
↑ cooperative

4 condition for process synchronization

1) Mutual exclusion :- Primary rule. (must be followed)

eg 

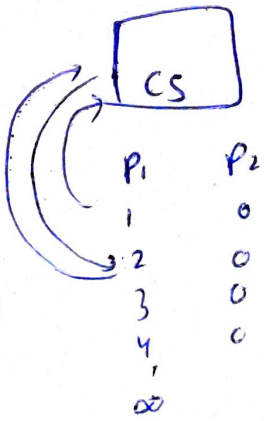
if P_1 accessing shared code then P_2 is not allowed.

2) Progress :- eg P_1 not running & also stoping P_2 to run. so here no progress happens. This is not allowed if you want process synchronization

3) Bounded wait :- eg

P_1 comes out & again enters. this happens again & again so P_2 starves.

so for synchronization wait should be less for all process.



4) No assumption related to hardware speed.

eg condition like it work in 32 bit not 64 bit, this is not allowed.

Soln must be universal

• Semaphores

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achive Synchronization

can be
1) counting ($-\infty$ to $+\infty$)
2) Binary (0, 1)

→ P(), Down, wait → synonyms. (Entry code)

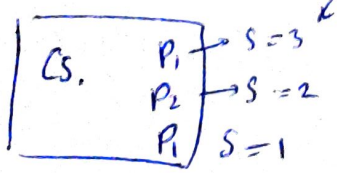
→ V(), up, signal, post, Release (Exit code)

```
Down (semaphore S)
{
 S.value = S.value - 1;
 if (S.value < 0)
 {
 put Process (PCB) in
 suspended list, sleep();
 }
 else
   return;
}
```

```
up (semaphore S)
{
S. value = S.value+1;
if (S. value ≤ 0)
{
  select a process from
   suspended list
    wake up();
}
}
```

eg processes P_1 P_2 P_3
S initial value = 3.

So



now S = 0 & no process can enter critical section

- if S = -1
  So Process P_4 goes in suspended list(). (Blocks)

- now to run process from suspend list, use up code.
  here it goes to ready queue.

Binary Semaphore → more used

0 ↘ 1

```
Down (semaphors)
{
  if (S.value == 1)  → successfully in critical section
  {
    S.value = 0;
  }
else
  {
    Block this process
    & place in suspend
    list, sleep()
  }
}
```

```
UP (semaphore S)
{
 if(suspend list is empty)
{
 S.value = 1
}
 else
{ select a process from
  suspend list & wake up()
 }
}.
```

Q Each process P_i {1 to 9} ↙ so are cooperative execute following code.
how many maximum no. of process that may present in CS at any point of time

```
repeat                    ↙ P(1 to 9)
entry → P(mutex) → decrease mutex.
        [ CS ]
exit  → V(mutex)
        ↳ value up.
```

mutex

[1]

∅ → P_1

1 → P_{10}

∅ → P_2

1 → P_{10} leave

∅ → P_3 enter

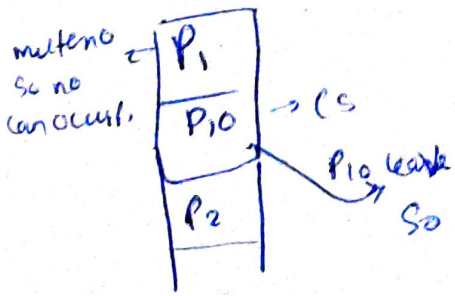
1 → P_{10} enter

:0 → P_4 enter

& P_{10} execute this

```
repeat
V mutex
  [CS]
V(mutex)
```

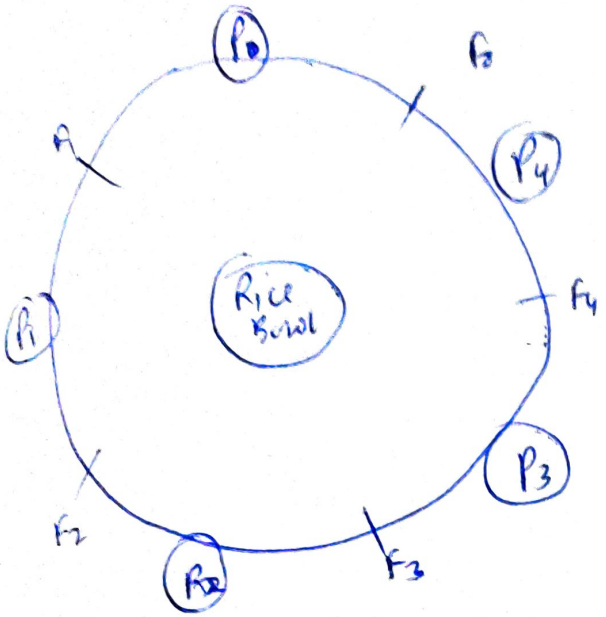
→ this occur when P_{10} leaves CS.



all can enter like this

So max = 10

Dining Philosophers Problem



's philosphers'

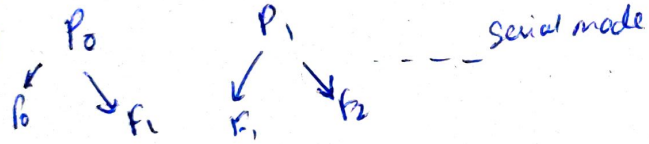
's fork'.

Philosopher sometimes eat & sometimes think

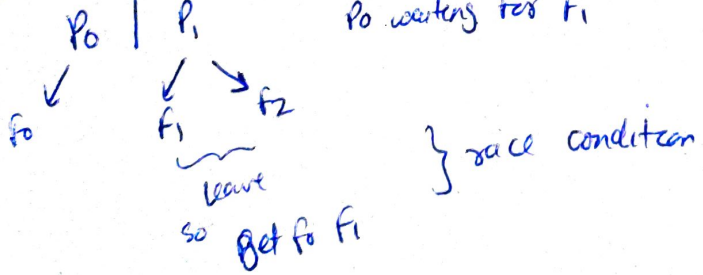
```
Void Philosher (void)
{
  while (true)
  {
   thinking ()
   table_fork (left)=i
   table_fork((i+1)%N)
   EAT()
    Put fork(i)
    Put fork((i+1)%N)
  }
}
```

Case I

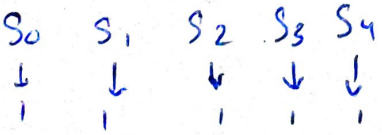
Eg



Case II



to Solve this we use binary semaphore.



P_0	S_0	S_1
P_1	S_1	S_2
P_2	S_2	S_3
P_3	S_3	S_4
P_4	S_4	S_0

and code changes as

```
Void philosphar (void)
{
while (true)
{
thinking();
wait ( take_fork (si))
wait ( take_fork ( S(i+1)%.N)
EAT ();
UP ( put_fork (i));
UP ( put fork ((i+1)%.N)) ;
}
}
```

here if $P_0 \rightarrow S_0$
$P_1 \rightarrow S_1$
$P_2 \rightarrow S_2$
$P_3 \rightarrow S_3$
$P_4 \rightarrow S_4$

together

so deadlock happened