Operating System

3 pillers of OS: Virtualization Concurrency Storage

- Booting - Bootstrap Program - Loading OS - OS Architectures
- Process execution - System Calls - Privilages - Process States - interrupt - context switches - Interprocess Communication
- Threads
- System Call APIs
- Time Sharing ⇒ Multiprogramming, multitasking
  Scheduling Algorithms

Synchronization:

- Peterson's Solution
- Hardware Based (Test & set lock)
- Software Based (Semaphore / mutex)

- Deadlocks — Banker's Algorithm.

Memory Management (RAM)

- Contiguous / Non Contiguous (Paging, segmentation)
  - virtualization

Disk Management (Hard Disk)

- Algorithm FIFO, SSTF For decrease Seek Time.

File System

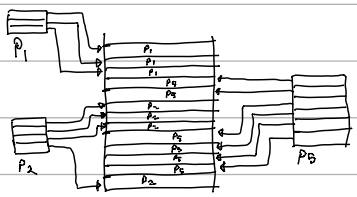
- Contiguous / Non Contiguous Memory Allocation.
  - Unix Inode Structure.

Necessary Condition for Deadlock:

1. Mutual exclusion
2. Non Preamptive (No termination before finishing)
3. Hold & Weight — i.e. Don't give up control
4. Circular Dependency.

Static Memory Allocation,
Dynamic Memory Allocation.

Paging

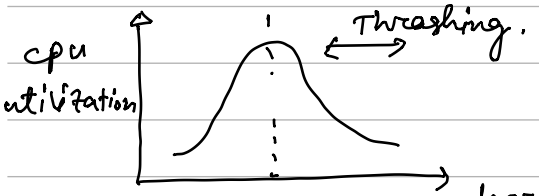


Actual memory.

process sees as contigious block of memory.

Thrashing : When CPU is doing mostly page swaps and not executing the processes.

↳ occurs when degree of multiprogramming is high.

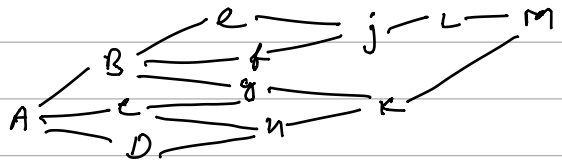


Multiprogramming : multiple process in RAM so that when 1 process goes I/O CPU can execute another process

Multitasking : multiple process share common resource CPU, RAM & using Round Robin to execute many processes at the same time.

Multiprogramming:

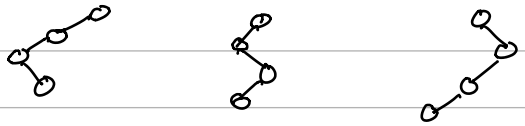
Travelling Salesman problem



Solution: Back Tracking exponential Sol^{n}.

⇒ Trick in multiprogramming:

machines: M1 M2 M3 ...



checking all possible possibility and updating a global variable Global Min

if a machine Mi finds

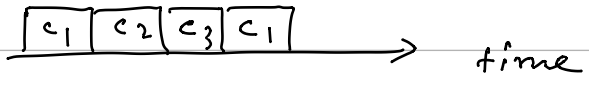
current Total + MST of Remaining > Global Min (current) (Terminate) because better path has already achieved.

- we need Isolation between Processes.
  Some features like Copy/Paste have to be Process sharing
  ↳ OS provides necessary features.

- OS has to provide these features
  1. Virtualization
  2. Concurrency
  3. Storage.

Virtualization: Process P_1, P_2, P_3 believes they have their own CPU C_1, C_2, C_3

this can be achieved through Time sharing.

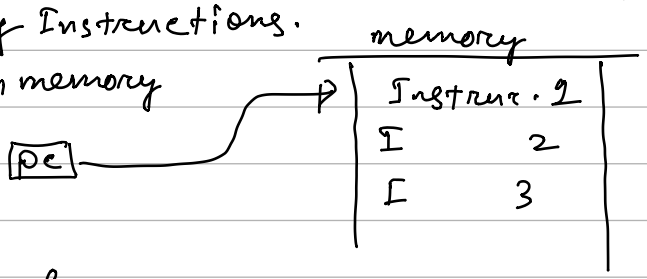


m = no. of Processes when m >> n
n = no. of Cores virtualization is must.

Q. How a process Run ?

program code → Compile → executable → Run it → Process

// executable is sequence of Assembly Instructions. Stored in memory



Instruction cycle { Fetch → Decode → Execute → Store → PC+1

Stack : functions along with their local variables.

Heap : Malloc , so that it's globaly accissible.

P_1 & P_2

when P_1 goes I/O

or, P_2 comes with high Priority

Process P_1 is thrown out & P_2 runs (Context Switching)

Mechanism & Policy:

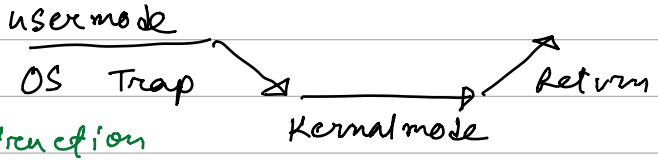
How → Scheduling (which program to run next)

○ Restricted Operations ;

mechanism → 2 privilage Modes → User - mode, Kernal - mode.

e.g. Open (file) # open is a System Call

OS provided interface / API



The instruction to go from User mode to Kernal mode called OS Trap.

1) Context Switching

Save state of Running Programme.

↳ { pc, registers, SP, memory ... }

SP = Stack Pointer

2) What happens when open(filename, mode) is called.

user code

open —— { System Call }

→ Implemented as a library Routine Runs few instructions in sequence.

↓

* Part of the OS but runs in user mode.

① Format Arguements.

open(filename, mode)

to the OS defined function/ instruction Argument format.

[then pushes the Arguements to User mode Stack]

② Trap ↳ Kernel Mode

Handler → copies Arguements from User Stack to Kernel Mode Stack.

at this stage you can kill a process.

* → validate.

→ Return file Descriptor.

→ RTU (come to User mode)

RTU:

↳ Lower Privilage level.

↳ Restore State of the calling Process

↳ PC - Resume Running.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call **trap** into OS
	save regs (to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to PC after trap	
		...
		return from main **trap** (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

locations of these **trap handlers**, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.

System Call APIs :

Creation of Process with Fork :

1) Create and initializes the process Control Block (PCB) in the Kernel.

2) Create a new Address Space

3) Initialize the address space with a copy of entire contents of the address space of the parent.

4) Inherit the execution context of parent (open file, privileges)

5) Inform the scheduler, the new process is ready to run.

UNIX Process Management APIs

- fork
- exec - sys call to change the program being run by the current process.
- wait - wait for a process to finish.
- signal - send notification to another process.

NESO

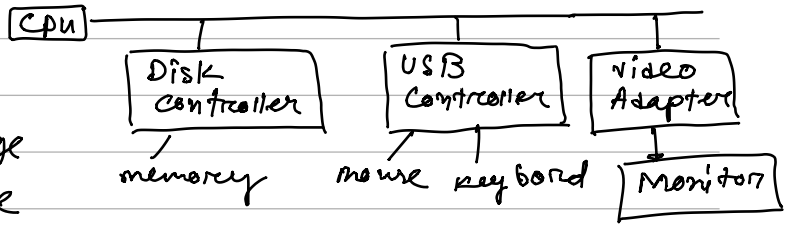
Bootstrap Program : 1) Runs when Computer is powered up.
2) Stored in ROM.
3) Know how to load the OS and start executing it.

Interrupt : can be done by hardware or software.

System calls.

I/O structure in Os

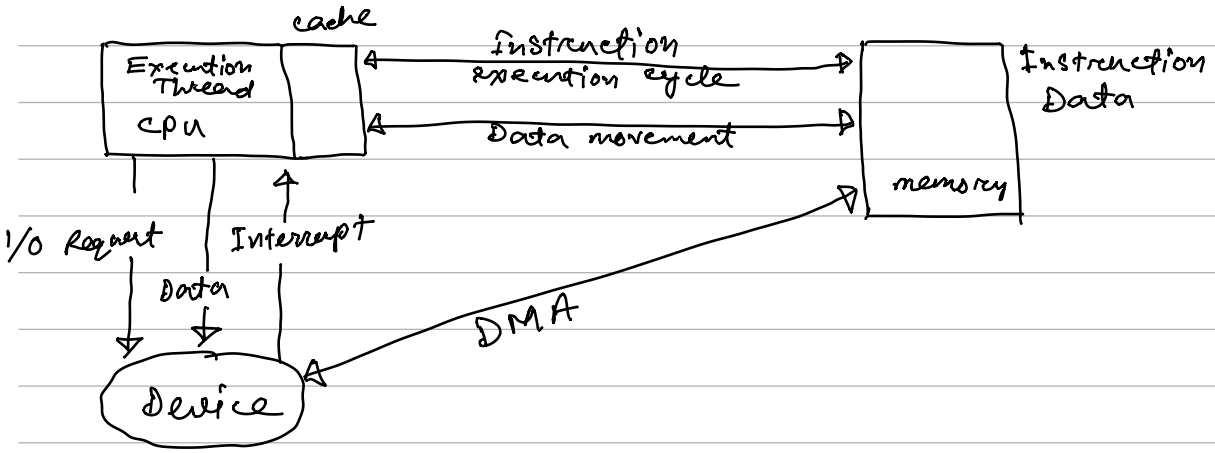
mouse connects to USB socket, that connects to device controller. Some Dev Controller has DMA, some Dont.



Device controller maintains ① Local Buffer Storage
② Set of special purpose registers.

* Os has device drivers for each device controller.
↳ To create interface between device & OS.

Working of I/O :



① To start an I/O devic driver loades the appropriate register in device Controller
② Device controller examines the registers & determins what action to take.
③ Controller starts transfer the data from device to its Local Buffer.
④ Once the transfer of data is complete device driver get an interrup by the device controller that data transfer is complete.
⑤ Device driver returns control to the OS.

old way

* Using DMA device controller can transfer blocks of dat to & from memory with no interruption in CPU.
↳ Using DMA only 1 interrup to the Driver is generated per block.
↳ CPU is Free
(without DMA interrupt is done per byte)

- **Single Processor**
  - only 1 processor

Multi Processor System

- parallel system
- Processors share Computer Bus, Clock, Peripheral Devices, Memory.
- Increase throughput ↑

Types of MultiProcessing

all cpu are involved doing all processes.

Clustered System

- 2 or more individual System coupled.
- Provide high availability.
- Can be symmetric & asymmetric.
  - [one machine in hot-standby mode giving application to other machines to run]

- All OS must do ① Multiprogramming
  ② Time Sharing.

- Interfaces : CLI & GUI
  - e.g. Bourn Shell, Bourn-again shell (Bash), c shell, Korn Shell

command line Interpreter can

- ① execute code which are in command line
  ② execute code in different program when called.

- **System Call** : sys call provides an interface to the services made available by an OS.

User mode : NO direct Access to Memory Hardware.
Kernel mode : direct Access

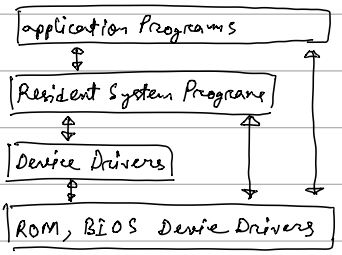
* sys calls are programatic way a program request service of OS kernel
* Sys calls are written in c/c++.

Types of System Calls:

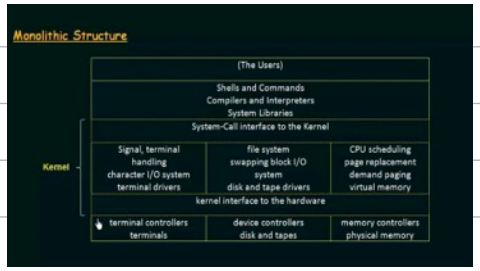
1. Process Control: (load, execute, malloc)
2. File manipulation (open, close ...)
3. Device Management (request device, release, read, write ...)
4. Information Maintenance
5. Communication. (send / receive messages)

Structures of OS

1. Simple Structure (used in Ms DOS)

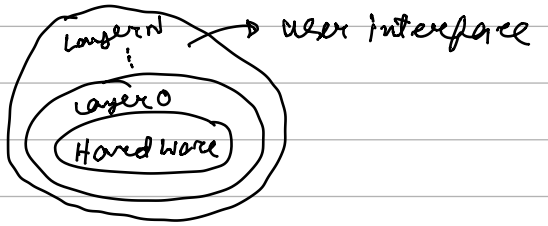


2. Monolithic Structure



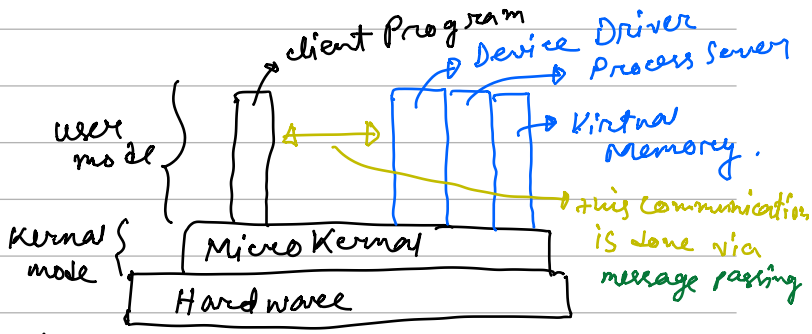
3. Layered Structure:

* very inefficient
UI has to pass through many layers to reach hardware.



4. Micro Kernal Structure:

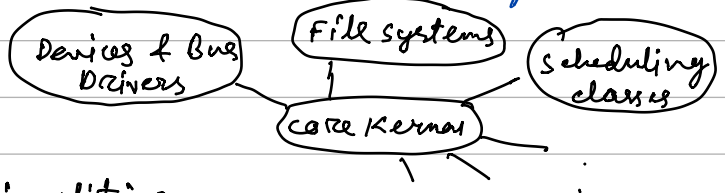
* when client program request system call micro kernal makes connection between client Prog. & OS services which are implemented as system programs. via message passing.



* Disadvantage: System overhead can cause decreased performance.

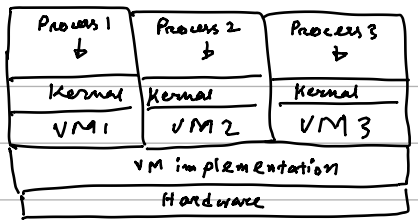
5. Modular Kernal: (Best)

* implemented using OOP.
\# Core Kernal have core functionalities other Kernals Load at boot time or Run time.



* No system overhead, coz no message passing b/w Application & service. Service Loads into core Kernal & Application directly access it.

Virtual Machines :



VM software runs in Kernal Mode
VM itself " " User mode.
each VM has own virtual User mode
virtual Kernal mode.

Operating System Generation & System Boot :

- OS made For a particular hardware
- OS can be used in a class of hardware
  - SYSGEN (generating OS for specific compatible Hardware)

Booting : Starting Computer by loading Kernal.
Bootstrap program / BS Loader (in Rom) does the job.
the OS resides in the Disk

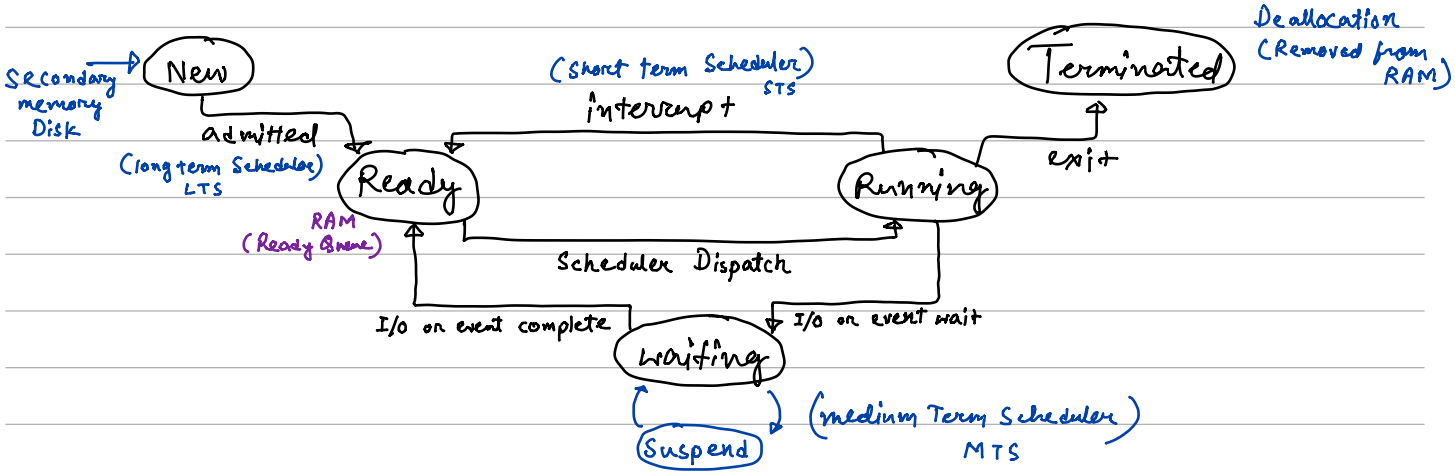
Processes :

process
programm in execution

thread
a unit of execution within a process, can be one or many.

Process States :

New	process is being created
Running	Instructions are being executed
waiting	Process is waiting for some event to occure (I/O)
Ready	Processore is waiting to be assigned to a processor
Terminated	Process has finished execution.



PCB →

Process State
Process number
PC
Registers
List of open files
⋮

Process Control Block : ① P_id

② Process State (New, Running, waiting ...)

③ PC (address of next instruction) ④ CPU Registers

⑤ CPU scheduling information.

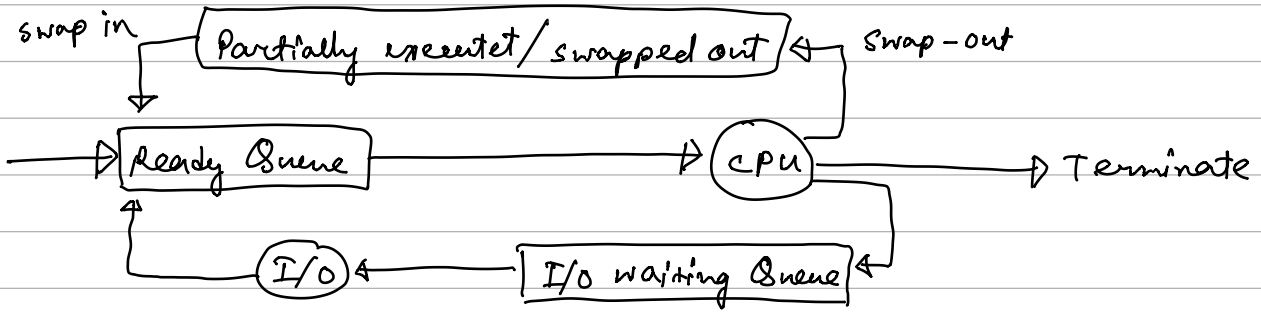
⑥ Memory management information.

⑦ I/O status information.

Process Scheduling : scheduler decides which process to release.

* Job Queue : As soon as a process enters system it's put in JQueue. contains all processes in the System.

* Ready Queue : The processes that are in main memory (RAM) process that are ready & waiting to be executed.



Context Switch : save state & restore state

When interrupt occurs context is saved.

Context is represented in the PCB of the Process.

* Context switching time is pure Overhead
  ↳ System does no useful work.

Operations on Processes

① Process Creation : ① user can create / OS can create

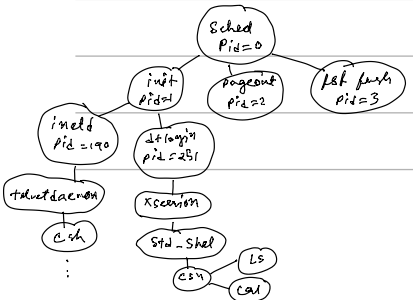
② Process itself can create.

* child process may need all resources of Parent process.

* Address space of child process:

1) child may be duplicate of parent
2) child may load completely new program

Tree of Processes on a Solaris System



② Process Termination:

- exit() system call. Process requesting OS to delete the process
- child process can send to it's parent a status of termination the parent can receive via wait() system call.
- OS deallocates all virtual memory, I/O buffers.

Another process can kill a process by system call.

✓ generally such system call can only be invoked by the parent.

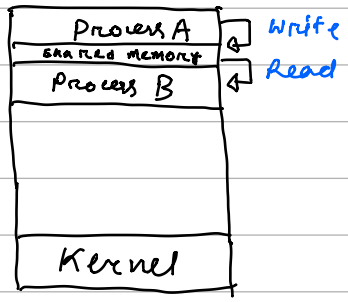
this is required if child over uses resources, child process is no longer required.

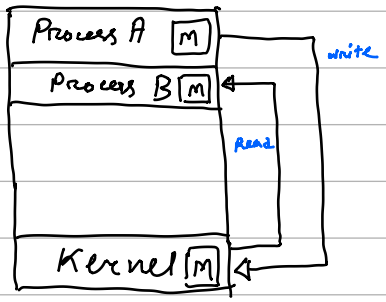
Interprocess Communication

there are 2 types of Processes → Independent - can't or can't be affected
→ Cooperating - can & can get affected.
e.g. any process sharing Data.

why process Cooperation?
① Information Sharing
② Computational Speedup
③ Modularity
④ Convenience

o Fundamental model of Interprocess Communication:
① Shared Memory (multiple process read & write on a common memory location)
② Message Passing. (message exchange b/w 2 process)





* To Shared Memory system to occur must remove the restriction of one process's memory can't be accessed by other process.

Shared memory can be understood by producer consumer problem (client server model)

2 types of Buffers
① unbounded buffers
② Bounded buffers

use a buffer to fill by producer & consumed by consumer. → Solution
↓
like a shared memory.

Message passing System : → 2 operations send message, Receive message.
↳ msg. can be fixed size or variable size

① Direct Communication :

→ Send (P, message) send msg. to P
Receive (Q, message) receive msg. from Q. } Symmetric in Addressing.

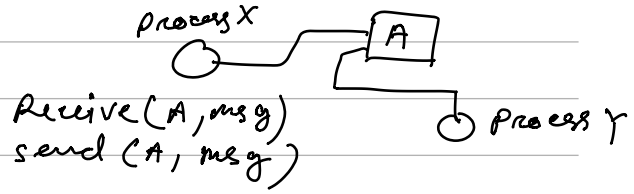
↳ A link must be connected between P to Q

→ send (P, msg)
recive (id, msg) id of receiving process } Asymmetric in Addressing.

② Indirect Communication :

* analogy to mailbox

⇒ for 2 process have connection they must share common mailbox



* if there's more than 2 process connected. then many things are possible like round robin to determine next receiving process.

* Mailbox can be owned by process or the OS.

---

① Synchronous & Non Synchronous :

Blocking Send : Sending process is blocked until msg received by receiving process or mailbox.
Non blocking : Sending process sends the msg & resumes operation.
Blocking receive : Receiver is blocked until a msg is available
Non blocking " : Receiver retrieves either a valid msg or null.

① Socket : when processes needs to communicate via a channel.
(same as socket used in computer Network)
socket is identified using Port Number.

* Interprocess communication mechanism ⇒ different process communicate in same Device.

* Remote Procedure Calls (RPC) ⇒ processes residing in different Systemp wants to communicate over a Network.

Remote Procedure Calls

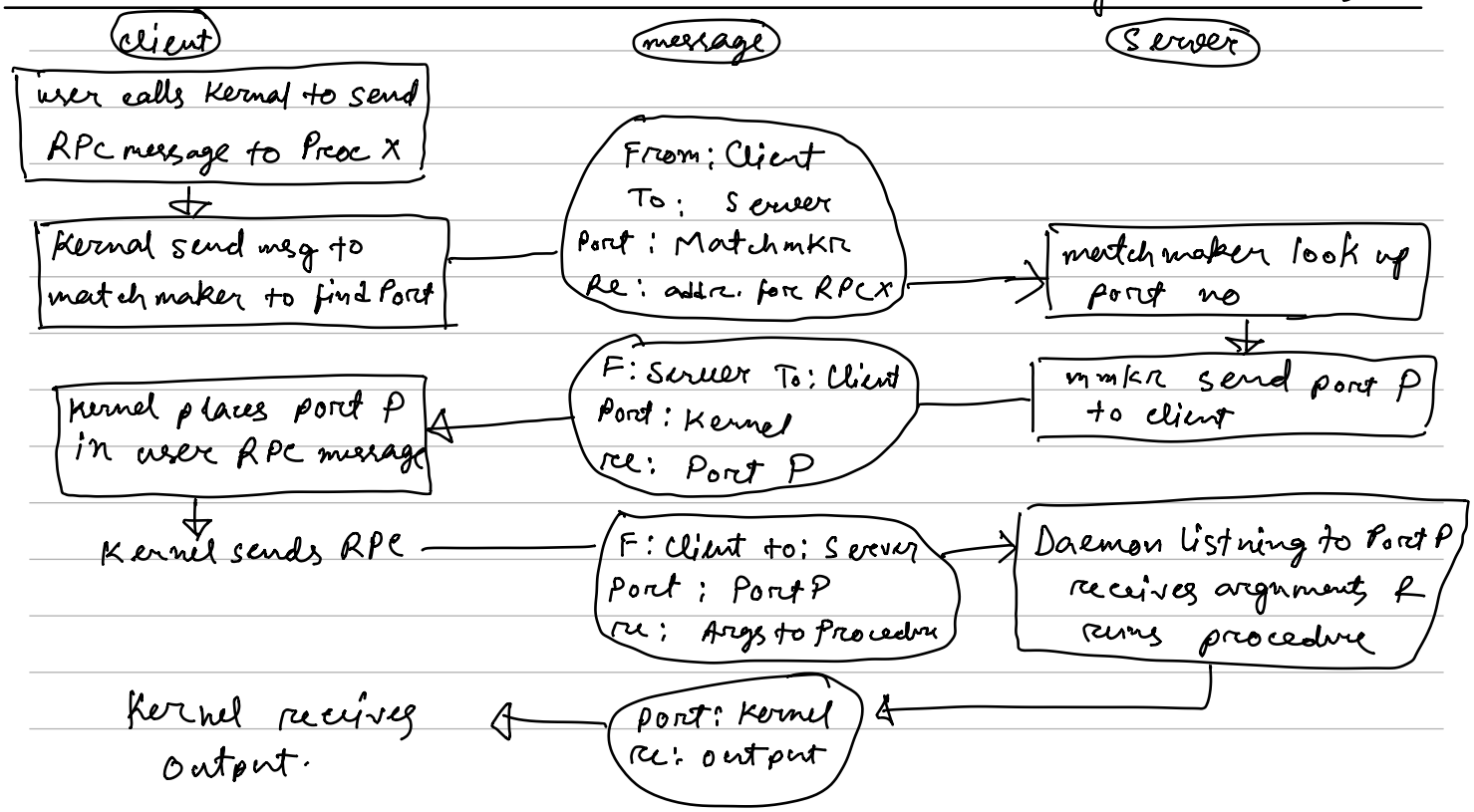
Inter process communication

RPC is similar to IPC but here only Message Passing Scheme works.

* Remote device running a RPC daemon always listning to a port. Message contains parameters to pass to that function.
# RPC hides the detail of communication by providing a stub, so system runs procedure as if it's done locally.
* function parameter marshalling & unmarshalling.
(converting to XDR) (reverting to device representation)

Issues in RPC

Issues	How it's resolved
① different data representation in different devices. i.e. For 32 bit integer some are little some are big-endian	RPC system takes machine independent data e.g. XDR (external data Representation) Before sending or after receiving data it's converted to machine's own representation
② Procedure call can fail, re-execute multiple time.	OS ensures message is executed "exactly once" & not "at most once". sending an ACK to tell procedure is executed once.
③ How client knows which port is for which procedure Name?	① By assigning fixed unchangable Port no to procedure at compile time ② By Rendezvous mechanism / (match making mechanism).

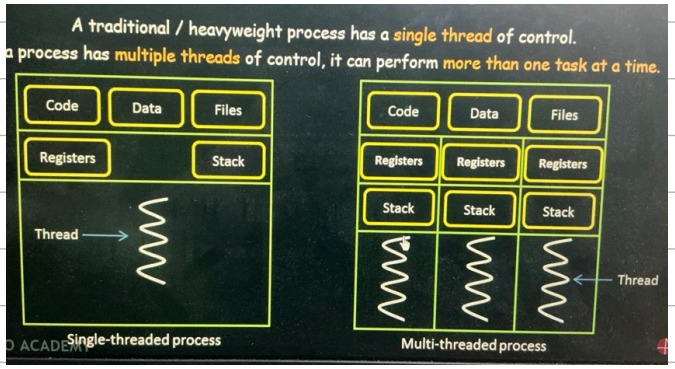


Thread

Thread is a basic unit of CPU utilization.

✓ Comprises of Thread ID, Program Counter, Register set, A stack.

✓ Code, data, OS resources like open file, signals can be shared between threads of same process.



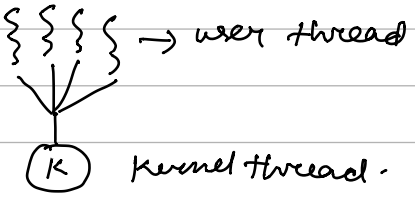
Types of Threads

User Threads
above kernel level without Kernel Support

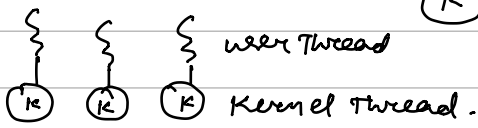
Kernel Threads
supported and managed by OS

Relation between User and Kernel Thread :

(1) Many to One model :

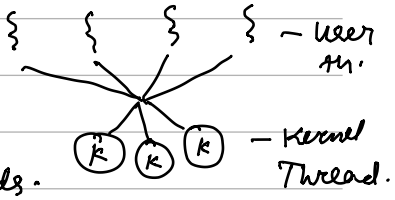


(2) One to One model :



(3) Many to many model :

↳ Multiplexes many user threads to a smaller or equal no. of kernel threads.



Simultaneous Multithreading (SMT) / Hyperthreading : (→ Name by Intel.)

SMT systems allows their processors to become multiple logical processors for performance.

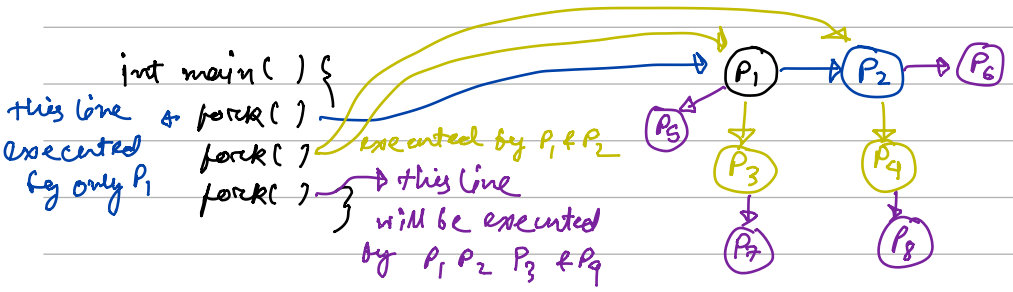
0

Fork & Exec :

fork : duplicates the process.

exec : the program specified in the parameter to exec() will replace the entire process including all threads.

↳ Process ID remains same.




```
// ex1.c
{ print("in ex1", getpid())
  execv("./ex2", args)
  print("back to ex1")
  return 0 }
```

```
ex2
{ print("in ex2", getPid( ))
  return 0 }
```

↳ o/p:

in ex1 2561 → Both these have same PId.

in ex2 2561 ↵

* UNIX has 2 types of Forks. → ① which duplicate all threads
② " doesn't " " "

- if exec( ) is followed immediately after fork ② is better option.

Thread Cancellation :

① Asynchronous Cancellation : Another th immediately terminate the target thread.

Quick ↑

② Deferred Cancellation ; Thread periodically checks if it can terminate itself & terminates in a orderly manner.

↓ Safer but not quick

CPU Scheduling

process execution consists of cycle of CPU execution and I/O wait (cpu & I/O Burst)

Preempt means "Take action in order to prevent from happening"

preemptive = Hatao

- CPU Scheduler:
  whenever the CPU becomes idle short-term Scheduler (CPU scheduler) selects any one among the ready processes.

- Dispatcher: Dispatcher is the module that gives control of the cpu to the process selected by the scheduler.
  (Time taken by dispatcher to stop one process & start another called Dispatch latency).

Circumstances when CPU Scheduling may Occur:

Non Preem (1) Process switches from Running to waiting State.

Preemp (2) " " " Running " Ready " (interrupt)
(Round Robin / OS interrupt)

preemp (3) " " " Waiting " Ready " (I/O complete)

Non Preem (4) " Terminates.

(1) & (4) choice is easy schedule another process. → Non Preemptive (Cooperative)

(2) & (3) have choice to schedule which process → preemptive (Non Cooperative)

* Preemptive: required in scenario like higher priority job comes.
↳ Failure: If one process writing on memory then another process arrives & disturbes.

Scheduling Criteria :

① CPU utilization : Ranges from 0% to 100%

② Throughput : # Processes Completed / unit Time

③ Turnaround Time : Time of Submission of a process to time of completion (waiting to get CPU + execution + I/O + ... until Termination.)

④ Waiting Time : Time spent in Ready Queue.

⑤ Response Time : Time from submission of request until first response is produced.
↳ * After starting executing process may keep generating multiple responses.

☐ First Come First Serve : FIFO is maintained.

- Avg. waiting time varies heavily. (if process's CPU burst time vary heavily)
- FCFS is non Preemptive (Cooperative)
- Not good in time Sharing System.
- Convoy Effect : Smaller processes waiting behind Large one.

Turnaround Time = Completion Time − Arrival time

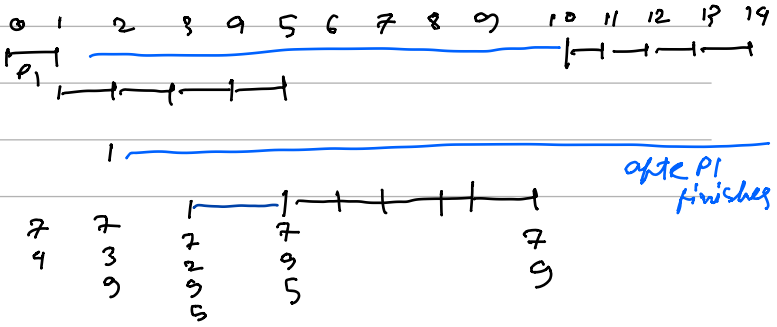
Waiting Time = Turnaround Time − Burst Time

☐ Shortest Job First ; (shortest Next CPU Burst) If 2 process in Ready Queue having same smallest next CPU Burst then FCFS is used to break Tie.

- Can be preemptive or non preemptive

e.g:

	arrival	Burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



For Preemptive Scheduling

Waiting Time = Total Waiting Time − No. of Time Process executed − Arrival Time

Preemptive SJF == Shortest Remaining Time First Scheduling.

* Implementing SJF is difficult.
  ↳ How will scheduler know next CPU burst of Process?
  * Assumption: process's next CPU Burst will be similare length to its previous ones. by approximating the next CPU Burst.

Preemptive Scheduling: | Turnaround Time = Completion Time − Arrival Time |

☐ Priority Scheduling:

SJF also priority scheduling where Priority $\propto \frac{1}{\text{next CPU Burst Time}}$.

Tie breaking with FCFS algorithm.
* Can be preemptive or non-preemptive.

- A low priority process may be indefinitely Blocked by incoming higher priority processes. (Starvation)
  ↳ solution to Starvation — Aging
  (gradually increase the priority that wait in system for long.

☐ Round Robin Scheduling:

- Designed especially for Time Shared System.
- similer to FCFS but preemptive. (Switch between processes)

FIFO Ready Queue P_1 P_2 P_3 P_4

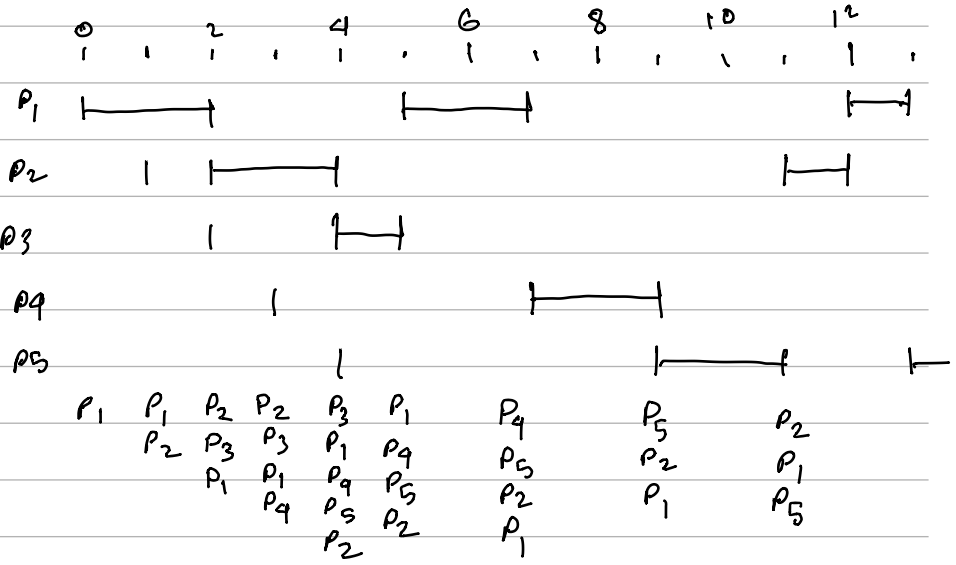
For RR

Turnaround Time = Completion Time − Arrival time

Waiting Time = Turnaround Time − Burst Time

= Last Start Time − Arrival Time
− Preemption × Time Quantum
(no. of time it got CPU)

	Arrival	Burst
e.g. P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3



☐ Multilevel Queue Scheduling:

* class of scheduling Algorithms
① Fore ground Processes (Interactive) (RR is used)
② Background Process (Batch) (FCFS may be used)

○ Processes are permanently assigned to one Queue.

e.g. ↑ System Process Queue
UI Processes
⋮

☐ Multilevel Feedback Queue:

○ difference b/w MLQ & MLFQ is here Process can goto another q.
i.e. ✓ if process use too much CPU it should be moved to low-Priority q.

Process Synchronization:

cooperative Processes → share memory
→ Pass message

☐ Critical section problem ⇒ To have mutual exclusion, progress & bounded waiting.

○ Peterson's Solution: Let the other process enter critical section.

turn = i or j ◇ 2 processes.
flag[i] = True flag[j] = True

code in process i:

```
flag[i] = True
turn = j
while( turn==j && flag[j]);
   (don't do anything)
// critical section
flag[i] = false
// exit CS
```

○ Hardware based Solution: Test & Set Lock:

```
bool testSet( bool *target)
    bool rv = *target;
    *target = true;
    return rv
```

```
while( TestSet(&Lock)) ;
    (nothing)
// then critical section.
Lock = False ;
// exit CS
```

satisfies mutual exclution
* If there are multiple Processes, Starvation may occure.
(bounded waiting is not satisfied)

○ Semaphore: (proposed by Dijkstra)
↳ denotes a single integer value.

wait ( ) or P( ) "Test"
signal( ) or V( ) "increment"

```
P(semaphore s){
    while( s <= 0);
    s--; }
```

```
V(semaphore s){
    s++; }
```

Types of Semaphores:

① Binary Semaphores ; S value 0 or 1. also called mutex locks.
② Counting Semaphores : unrestricted. usually equal to no. of processes.

* Semaphores wastes CPU cycle called Spin lock.
↳ Solution ; Make use of system call to put the process from running (Ready) queue to waiting q (change state of Process)
obviously it's to be implemented via hardware.

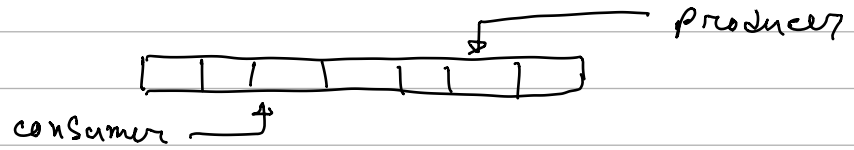
○ Deadlocks :

P_0	P_1
wait (A) → got A	wait (B) → got B
wait (B) ~ LOOP	wait (A) ~ LOOP
:	
signal (A)	signal (B)
signal (B)	signal (A)

waiting → finite (Starvation)
→ Infinite (Deadlock)

○ Using Semaphore to Solve Classic Synchronisation Problems:

① Bounded Buffer Problem / Producer - Consumer Problem.



Sol^n : semaphore sem, empty, full ; 3 integers.

if empty = 0 Producer waits if full = 0 customer waits.
both checks sem before reading / writing.

② Readers - Writers Problem :

in a database reader = read writer = read & write.
on same data if

①	②	
Read	Read	✓
Write	Read	✗ } Problematic.
write	write	✗ }

solution :

```
Semaphore  sem, wrt ;
int        readerCount ;
```

* idea if (wrt) is free writer can write
if readerCount == 0 then free the (wrt)
↳ (sem) is used for updating this variable.

```
wait(wrt)                    wait(sem)
// write                        readercount++;
signal(wrt)                     if(readercount>=1) wait(wrt)
                             signal(sem)
                             // READ
                             wait(sem)
                                 readercount--;
                                 if(readercount==0) signal(wrt)
                             signal(sem)
```

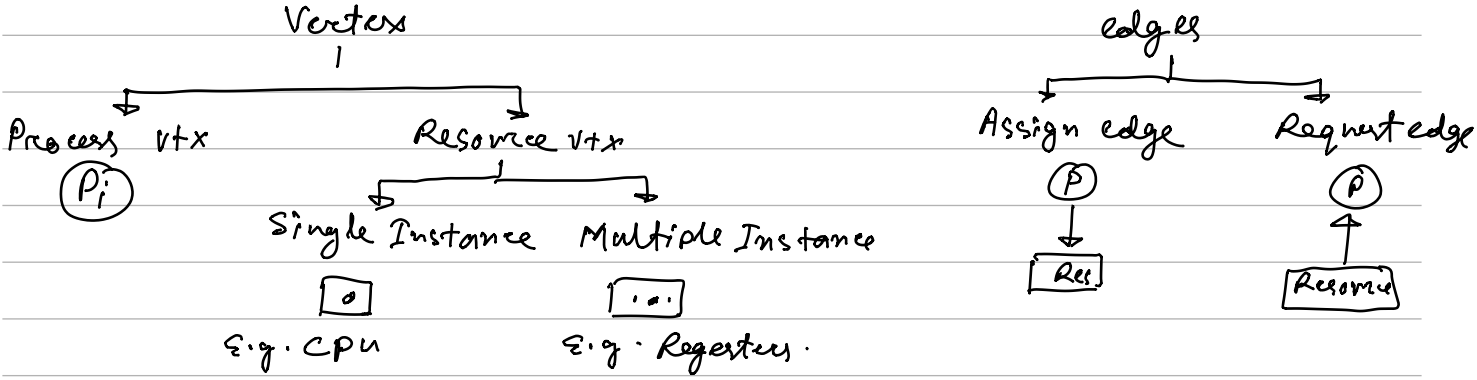
(3) Dining Philosopher's Problem:

Soln: Semaphore sems[5]
ith philosopher will hold i, i+1th sem & then signal both.

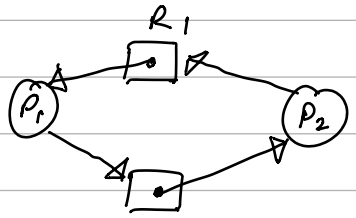
↳ Solution for deadlock: Odd philosopher first pick his left
even " " " Right.

Understanding Deadlocks

Resource Allocation Graph (RAG):

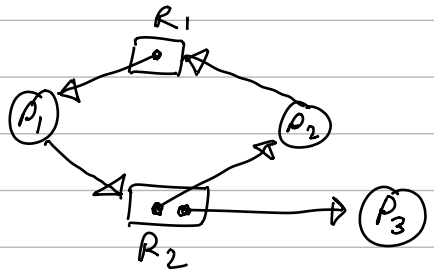


E.g:



It's a deadlock

E.g



Not a deadlock

* Cycle is not a determining factor of Deadlock if multi Instance resource is there.

* after P_3 Terminates P_1 gets one resource of R_2.

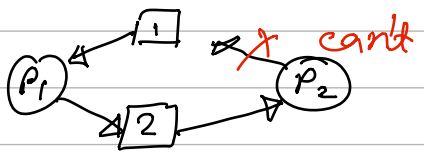
Handling Deadlock:

1. Ignore (Ostrich method) * most of the OS, windows, linux ignores. for performance.
2. Prevention → eleminate any one or all of the conditions — ME, NP, H+W, Cwait.
3. Avoidance (Banker's Algo)
4. Detection & Recovery ⇒ if there's deadlock kill any one process.

Prevention by not having circular wait

* process cannot ask for Low-Priority Resource

P_1 → 1) Printer
P_1 → 2) Register
P_2 → 3) Scanner
P_2



Banker's Algorithm for Deadlock Avoidance, Deadlock Detection

Total resources, A = 10, B = 5, C = 7

Processes	Allocated already			max need			current available			remaining need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2 ①	~~7~~	~~4~~	~~3~~ ④ ✓
P_2	2	0	0	3	2	2	5	3	2 ②	~~1~~	~~2~~	~~2~~ ① ✓ can take the request
P_3	3	0	2	9	0	2	7	4	3 ③	~~6~~	~~0~~	~~0~~ ⑤ ✓
P_4	2	1	1	4	2	2	7	4	5 ④	~~2~~	~~1~~	~~1~~ ② ✓ can take
P_5	0	0	2	5	3	3	7	5	5 ⑤	~~5~~	~~3~~	~~1~~ ③ ✓ can take
	7	2	5				10	5	10			

already allocated ↑

$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

it's not a unique seq ∴ (Safe Sequence)

Memory Management

↳ management of primary memory.
↳ To increase degree of multiprogramming / as many no. of processes in the RAM.

Q// 4 GB Ram
1 4GB Process
> Process does 70% time I/O

cpu Utilization = $1 - \frac{7}{10} = \frac{3}{10} = 30\%$

8GB Ram
2 Processes of 4GB
> 70% I/O.

cpu utilization = $1 - \frac{7}{10} \times \frac{7}{10}$
$= \frac{51}{100}$
$= 51\%$

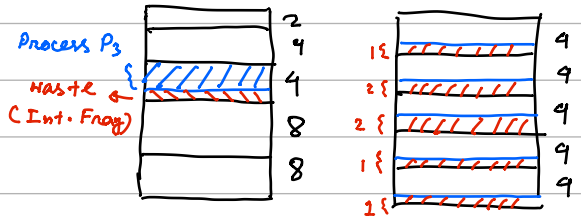
Memory Management Techniques

- Contiguous
  - Fixed Partitioning (Static)
  - Variable Partitioning (Dynamic)
- Non Contiguous
  - Paging
  - Multilevel Paging
  - Inverted Paging
  - Segmentation
  - Segmented Paging

① Fixed Partitioning

- Internal Fragmentation
- Limitation on number of Process.
- external Fragmentation.

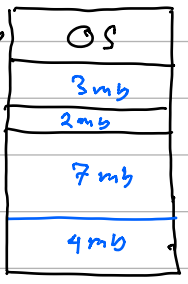
(whenever Internal Frag then External Frag also occurs)

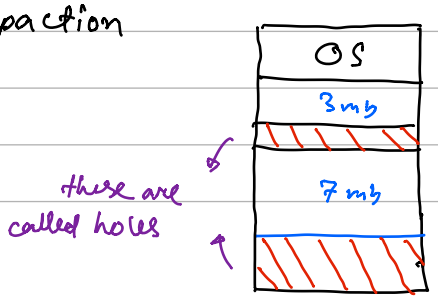


② Variable Partitioning:

incoming process 3mb, 2mb, 7mb, 9mb

- No Internal Frag.
- No max limit of process size or no. of processes
- External Fragmentation still occur.
  - ↳ Solution: Compaction





☐ methods of assigning incoming process to available hole:

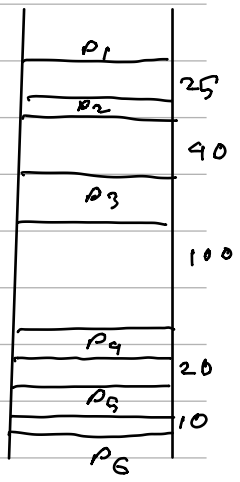
generally ↙ ↖ fastest

① First - Fit : First Big enough hole search from start.

② Next - Fit : Have a pointer to continue from onwards the last assigned hole.

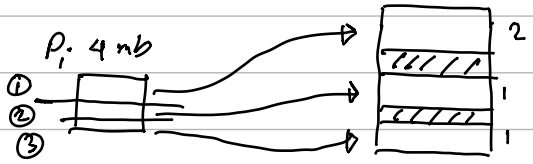
③ Best - Fit : The hole where internal frag. is min.

④ Worst - Fit : The hole where max memory left over.



Non Contiguous Memory Allocation :

idea : any incoming Process's data to be devided in the holes.



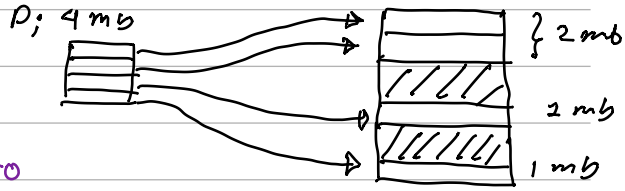
But, This is process is time Consuming.

The solution is,

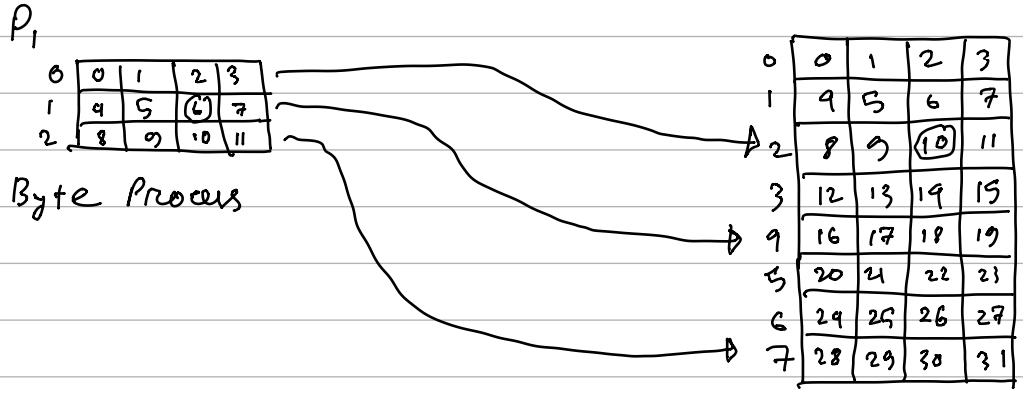
✓ Devide the Process's data in Parts before hand (called pages)

✓ Already Partition the Main memory into Parts (called Frames)

size of Page = size of Frame



- CPU asks MMU (memory management Unit) for Main memory Location of i^{th} Page of a Process, using help of page table.
- Each process has its own page table
  - ↳ Mapping of i^{th} page to j^{th} frame.



⇒ CPU wants what is at 6th Byte of Process P_1

Logical Address = 6 = | 01 | 10 |

page no | offset

Since there are 3 Page 2 Bit Page no.

CPU query → MMU → 2

frame[1] i.e. 001

Memory location = | 001 | 10 |

Same as Logical Address Offset.

Since there are 8 Frames in Memory 3 Bit Representation

⇒ Logical Address 9 == Memory address 29

| 10 | 01 |

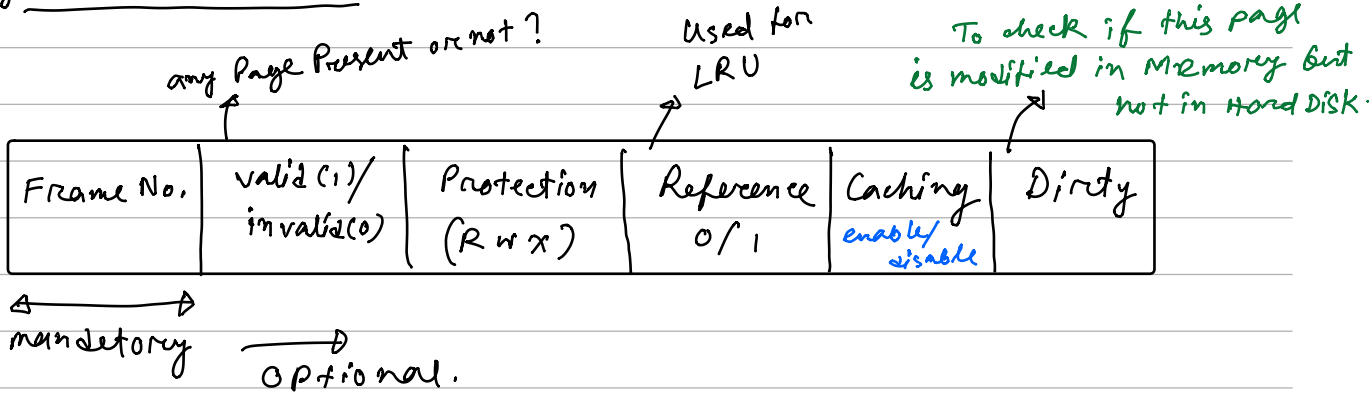
Page (2) offset = 1

| 111 | 01 |

frame (7) offset = 1

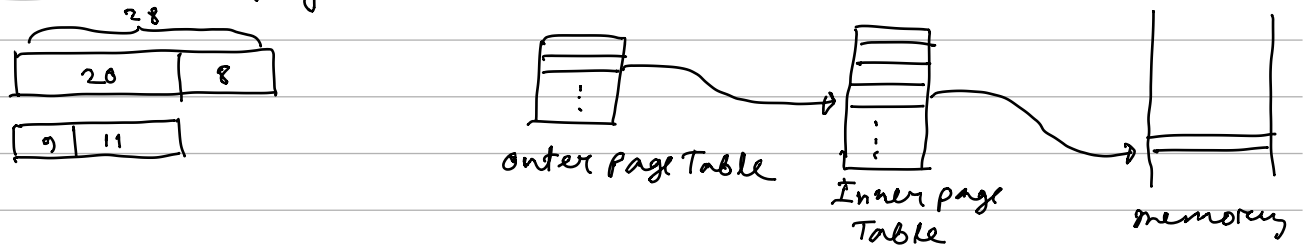
This mapping is Done by MMU using Page Table.

☐ Page Table entries :



○ Multi Level Paging :

(When page table Address number exceeds too much) more that what can fit in memory



Inverted Page Table:

global page table

frame no.	Page no.	PId
0	P_0	P_1
1	P_1	P_1
2	P_0	P_2

cpu → CPU linear Searches on whole page Table (extremely inficient)

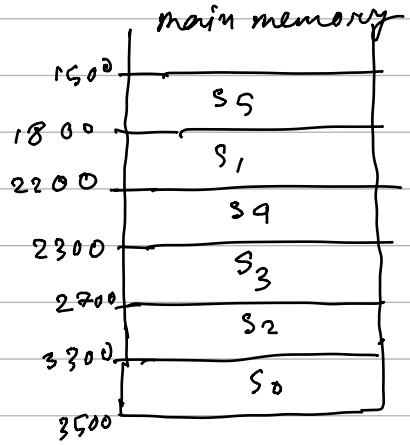
* entry == * frame in Main mem.

Segmentation :-

Idea: Paging Devides the program without context.
Inefficient if a function is devided 1/2 , 1/2 coz cpu has to fetch twice the data.

Segmentation — Devide the process according to content.

Segment no.	Base Addr	Size
0	3300	200
1	1800	400
2	2700	600
3	2300	400
4	2200	100
5	1500	300



CPU asking logical addr | s | d |

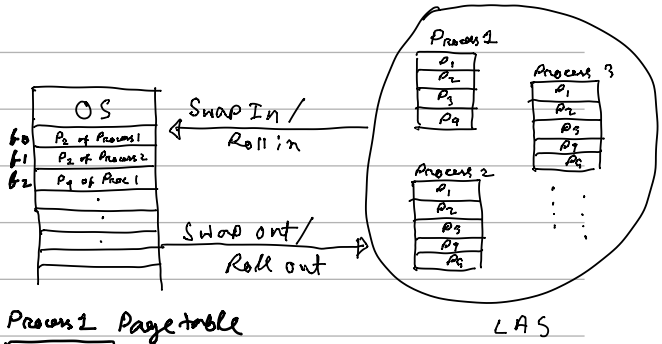
segment no. size

if s = 2 d = 600 , go to 2700 in Main mem & fetch 2700 to 3299 Data.

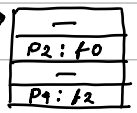
Overlay: Method of fitting Larger process in memory.
(process size > memory available).

☐ Virtual Memory:

Idea: CPU thinks all processes are present in its whole in the main memory.



*When CPU asks for P1 of Process 1 (Page Fault)



↳ Resolving Page Fault:

1. OS Trap, control from user mode to OS.
2. OS checks if P_1 is valid in Hard Disk.
3. Bring data from HD to MM & update P_1 in Page Table.
4. Control back to user mode.

○ Effective Memory Access Time:

$$EMAT = P \times \text{page fault service Time} + (1-P) \times \text{main memory access time.}$$

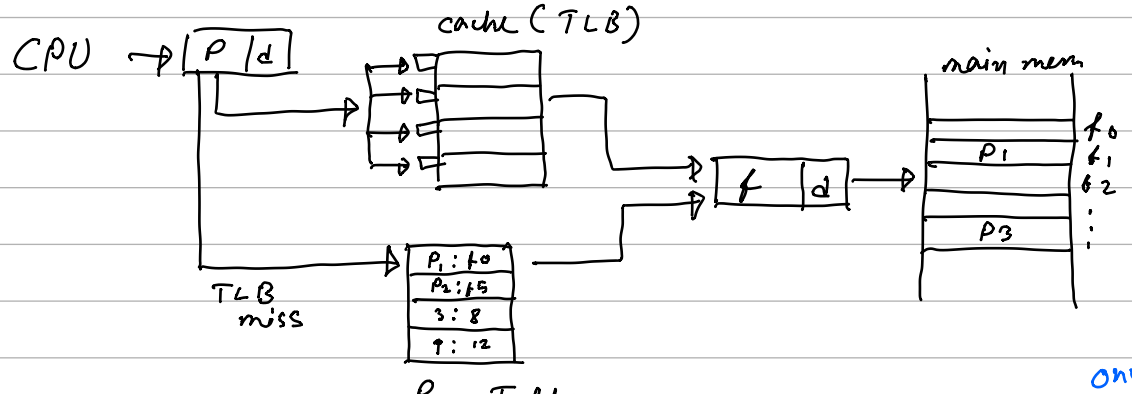
$P \times (PFST + \text{mm access time})$

But this is negligible compared to PFST.

☐ Translation Lookaside Buffer (TLB):

↳ commonly known as Cache.

Idea: Process page table resides in main memory, Instead store page table in Cache.



$$EMAT = \%Hit\left(TLB + \text{mm Access Time}\right) + \%miss\left(TLB + \text{mm Access Time} \times 2\right)$$

once for finding in page Table one for actual frame.

Page Replacement Algorithm :

1. FIFO
2. Optimal Page Replacement
3. LRU
4. MRU (most Recently used)

cache

* this to be replaced by 4

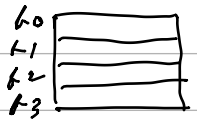
2, 1, 0, 3, 4, 2, 0, 3, 2, 1, 2, 7

Belade's Anomaly for FIFO Algo :

Page query: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

case 1

case 2

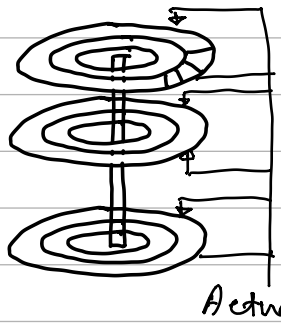


3 Frame cache — Hit 3, miss 9

4 frame Cache — Hit 2, miss 10

* occures only for FIFO algorithm.

Disk Architecture:

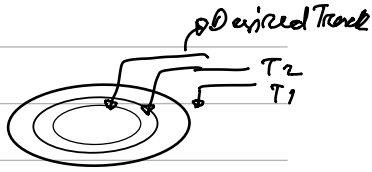


platters → surface → Tracks → Sectors → Data

no. = 8 no. = 2 no. = 256 no. = 512 each Sector 512 KB Data

Total Data = 8 × 2 × 256 × 512 × 512 KB

Read write Actuator Both Surface of Disk

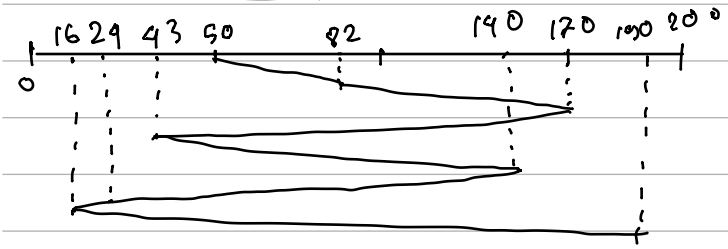


- Seek Time: Time Taken to reach Desired Track
- Disk Scheduling Algorithm to minimize Seek Time:

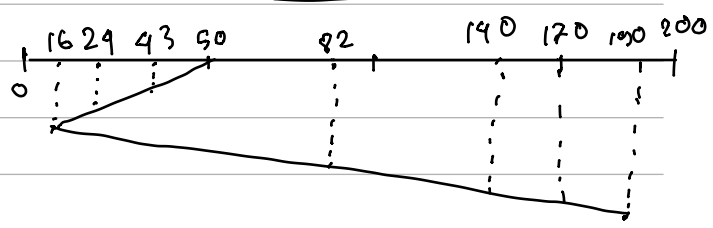
1. FCFS
2. SSTF (Shortest seek Time First)
3. SCAN
4. LOOK
5. CSCAN (circular SCAN)
6. CLOOK (circular look)

Eg: already received Track Request 82, 170, 43, 140, 24, 16, 190 & Track Head currently at Track 50.

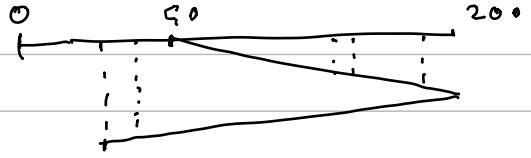
FCFS



SSTF

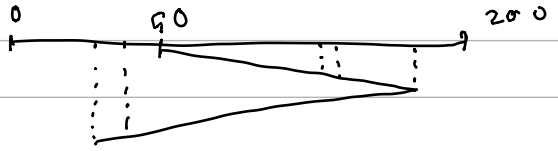


SCAN Algo:



move towards the last end (defined beforehand which end)

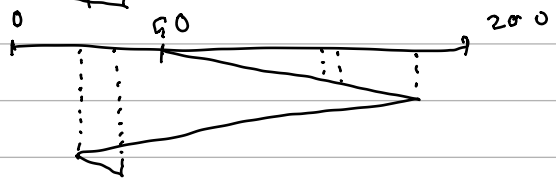
LOOK Algo:



same as SCAN but don't go to the extreme end.

C-SCAN

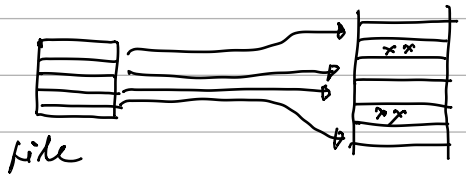
C-LOOK



File System in OS :

↳ How Data is Stored & Fetched.

Basically when a file is saved it's devided into Blocks & stored in the Disk



Allocation Methods

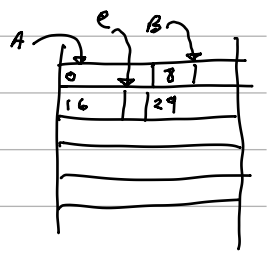
- Contiguous Allocation
- Linked list Allocation
- Indexed Allocation

Contiguous Allocation :

- Internal & External Frag.
- Difficult to grow file.

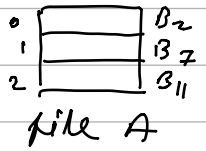
Dictionary

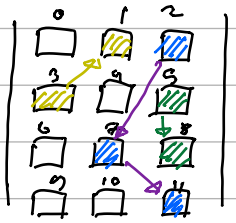
File	Start	Size
A	0	8
B	10	12
C	22	12



Linked List Allocation :

- No external Frag
- File size can increase
- Large seek Time.
- Random / Direct access Difficult
- Overhead of pointers.



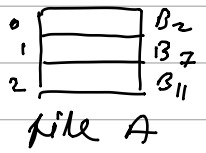


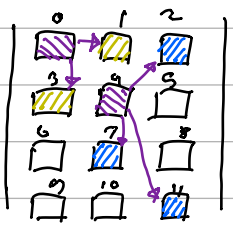
Dictionary

File	Start
A	2
B	3 → 1
C	5 → 8 → 10

Indexed Allocation :

- No external Frag.
- Unlike Linked list, support Direct Access
- Pointer Overhead
- Multilevel Indexing.

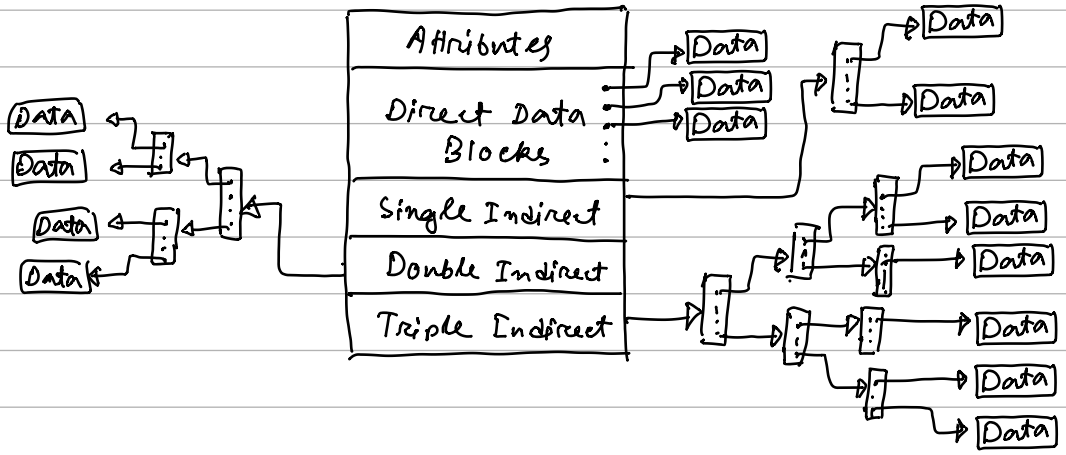




Dictionary

File	Idx Block
A	4
B	0
C	10 → 5 → 8

UNIX Inode Structure:



e.g. size of Disk Block = 128 Byte, Address space of Disk is 8 Byte
in 1 block max 16 address space can fit.

* if the Inode has 1 single, double & triple indirect & 7 direct Block
max possible file size = $(7 + 16 + 16\times16 + 16\times16\times16) \times 128$

$(7 + 16 + 16\times16 + 16\times16\times16)$: no. of Addresses

128: Block size of 2 address.