INDEX

NAME: Himanshu Mundhra STD.: 16 SEC.: CS ROLL NO.: 10057 SUB.: Operating Systems

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		• Silverschatz, Galvin		
		— OS Concepts 9th Ed		
		• Maurice Bach		
		— Design of Unix OS		
		http www.facweb.iitkgp.ac.in/	~isg/	OS
		144.16.192.60 /	~isg/	OS
		• Design of operating systems		
		— Hansen Per Brinch		

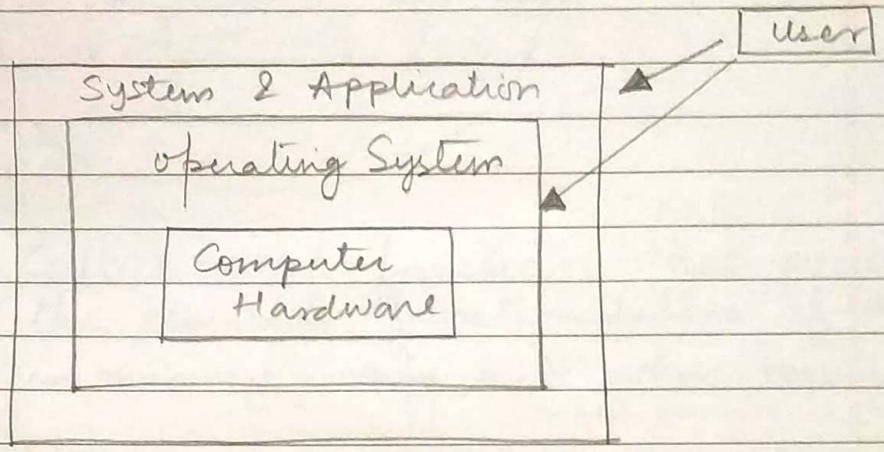
SPA0017047 A

Scanned by CamScanner

What is an Operating System ?

- starts with bootstrap program's loading
- never terminates without command.

* Program that acts as an INTERFACE b/w Computer Hardware and Users.



Computer Hardware ~~ڈ~~ direct access is Not allowed.

PRIMARY GOALS

- Make the Computer System Easy to use
- Ensures efficient utilisation of Resources
  - CPU time
  - Memory
  - I/O Devices
  - Files

Goals are Conflicting

time ↓

- Minicomputers & Mainframes (late 1960s – mid 1980s)
  - ~ Best utilisation of resource often at cost of user experience
- Workstations and Terminals (1980s – present)
  - ~ Fast response time to users often at expense of Resource Utilisation
    Context Switching

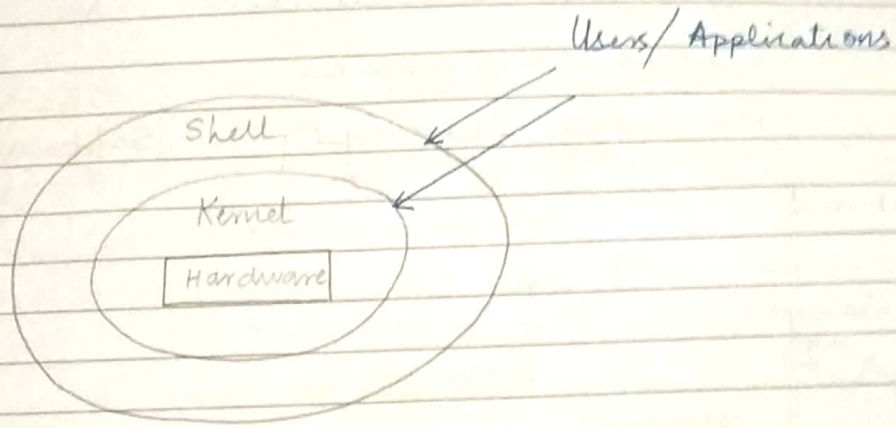
Scanned by CamScanner

- Personal Computers and Laptops
  (1985 - present)
  - flexible user interfare
  - Single-user systems

- Mobile Systems
  - conservation of energy
  - User experience
  - Resource Constraints

Scanned by CamScanner

Typical Structure of a Unix-like OS.

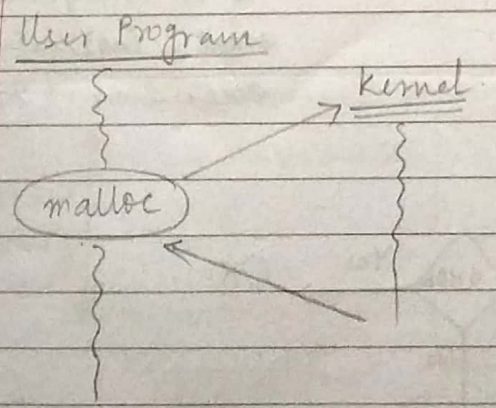


Kernel: Collection of functions that provide the essential functionalities of an OS
- idle in memory, only run when required.
when service is requested through syscalls

Shell: Interact with Kernel, whenever they require some service from the OS.

OS (i.e kernel routines) ARE INTERRUPT-DRIVEN.

Interrupt: Event that causes normal sequence of program execution to be interrupted



When CPU is interrupted;
- execution halts,
- transfer control to interrupt handler routine
- Handle the Interrupt
- Return to complete interrupted execution

Table of pointers to interrupt routines (interrupt vector) indexed by UNIQUE DEVICE NUMBER (interrupting device)

Scanned by CamScanner

Types of Interrupt -

event that leads to change in sequence of program execution.

send signal to CPU by System Bus

Hardware Interrupt

- **External**
  (I/O device interrupts)
  ^C , ^D
- **Internal** — can happen any time
  (Timer Interrupt, Memory Protection Violation *)

Software Interrupt — caused by execution of some INSTRUCTION

- **TRAP**
  It is an instruction
  (syscall (MIPS), int n (X 86))
  function call to a Kernel SubRoutine : perform some task
- **EXCEPTION** — ILLEGAL PROGRAM Inst EXEC
  - divide by 0
  - modulo by 0
  - Invalid Op Code
  - occur any time due to an unexpected event; instruction execution → RTE

Timer

Clock → Initial Value = 0 → Interrupt

Down Counter

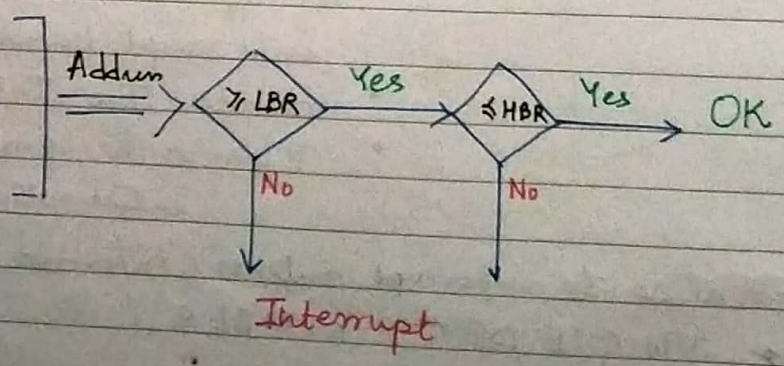
Timer Interrupt

* Memory Protection Violation

Fence Registers

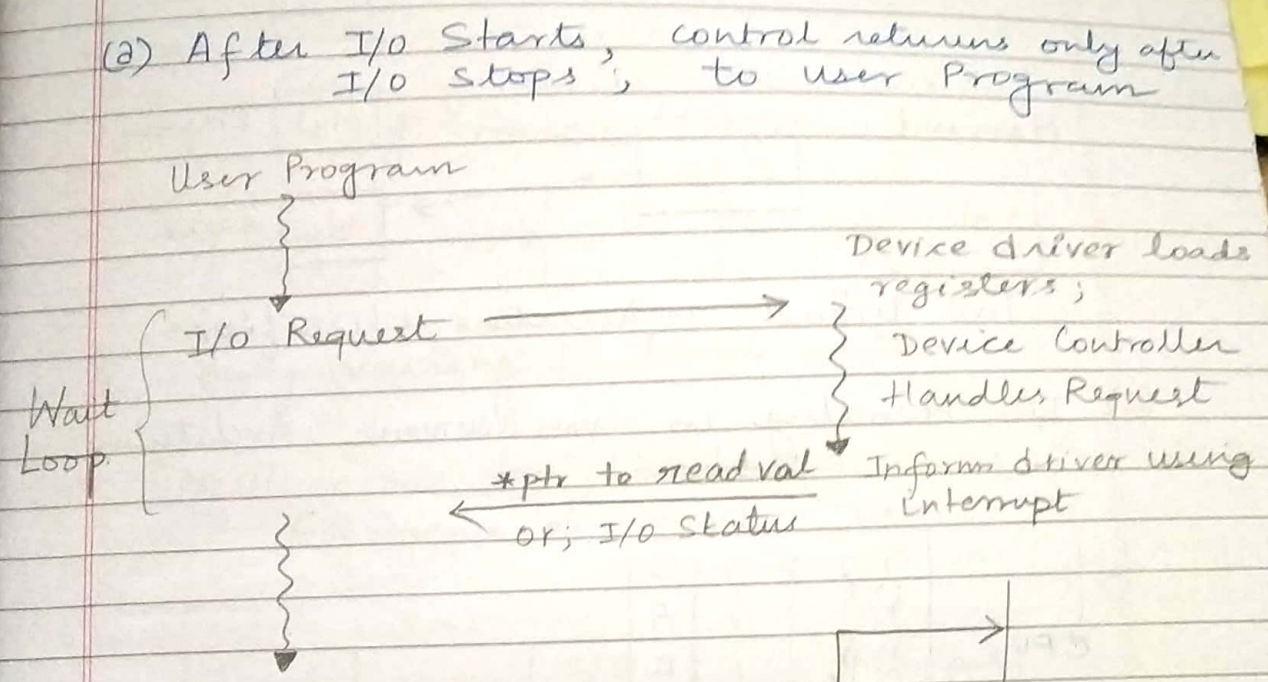
LBR

UBR

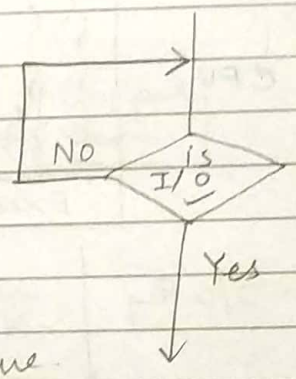


HANDLING I/O REQUESTS.

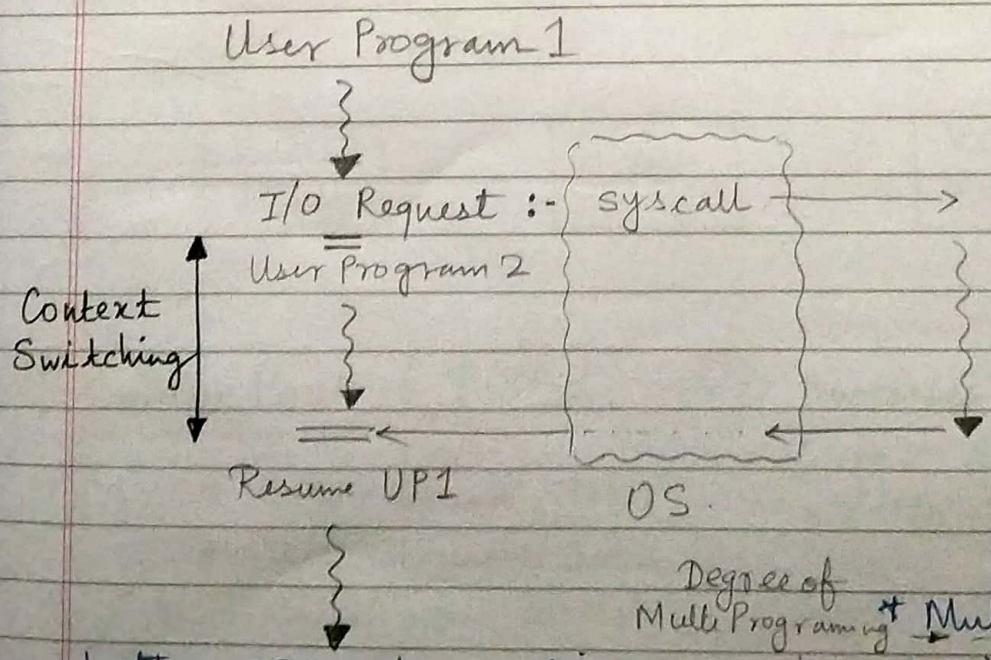
(a) After I/O Starts, control returns only after I/O stops; to user Program



- **BUSY WAITING**
- More than one I/O cannot be handled at a time I/O Queue



(b) After I/O Starts; control returns to user program; before completion of I/O

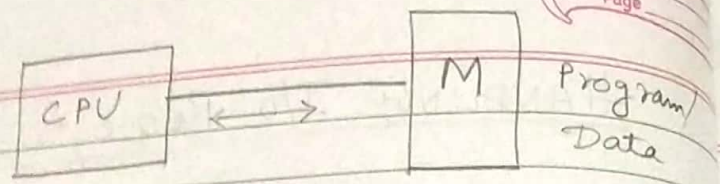


MULTI Programming

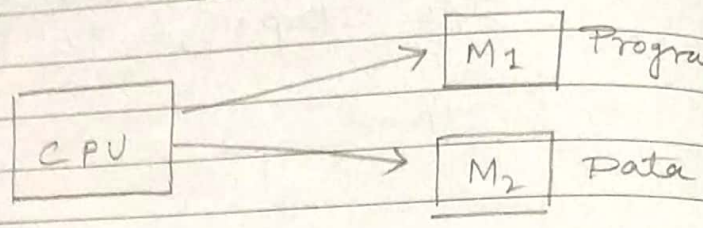
Degree of Multi Programming * Multiple programs in active state loaded in memory at same time.

* better CPU utilisation

Von Neumann

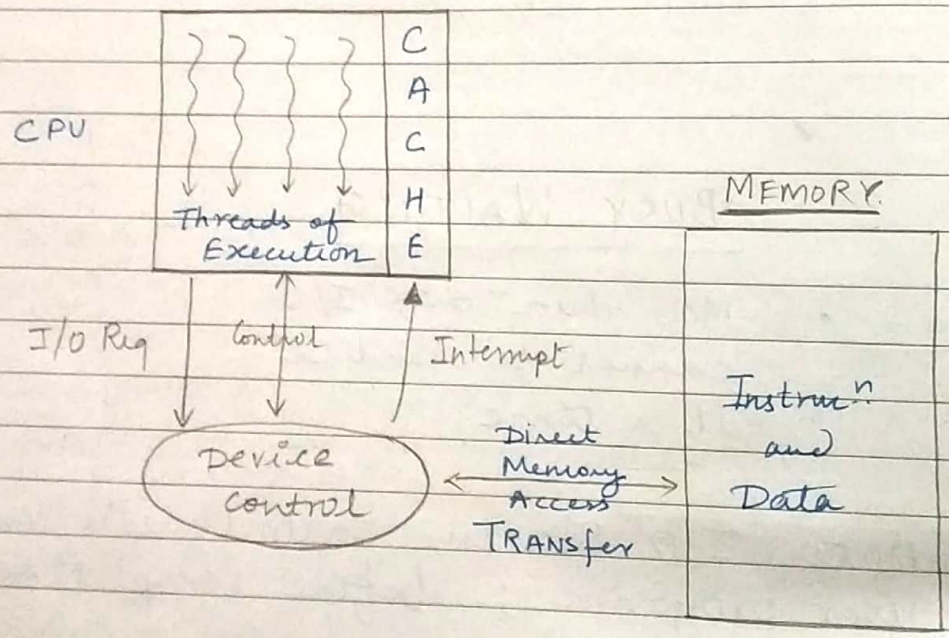


Harvard



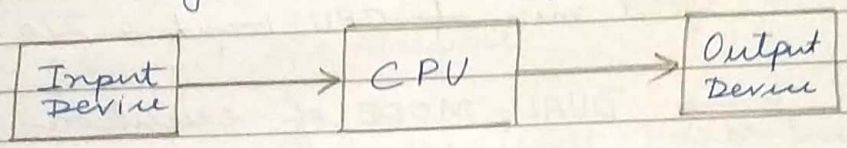
Two broad architecture classifiers

(c) I/O Handling in von Neumann Architecture



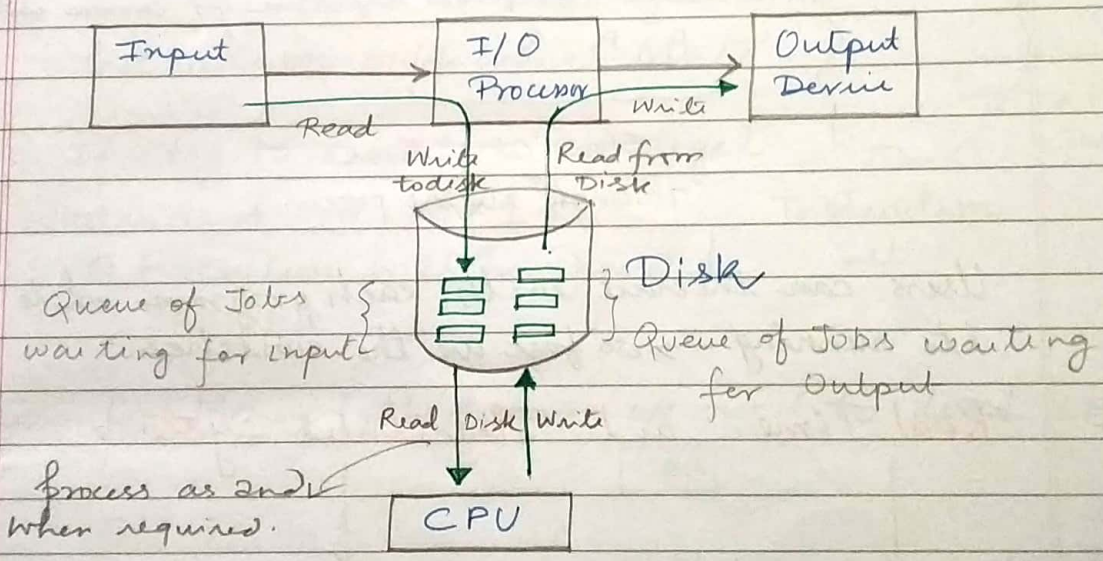
EVOLUTION of OS

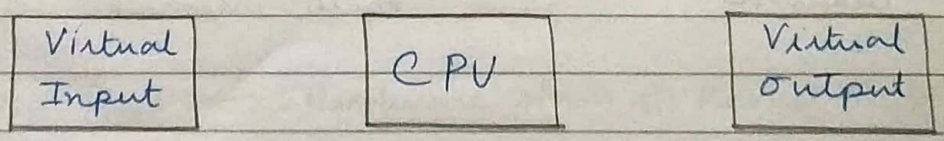
* Batch Processing OS
  Groups of similar Jobs, were put as batches
  (e.g. Fortran Programs)



Busy Waiting ~~ST~~ load in this.

* Simultaneous Peripheral operations On Line
  Emergence of Hard disks.





Job Pool (Disk) → processes WAITING for allocation to the MAIN MEMORY

* CPU - bound Jobs
* I/O bound Jobs

Scanned by CamScanner

* Multiprogramming

Give preference to first come processes

- multiple programs were "running" at the same time
  => MEMORY PROTECTION
- Good mix of CPU-bound & I/O bound jobs

need of → DUAL-MODE of execution

NO user interaction with Computer System

* Time Sharing or Multitasking

All equal priority

- Logical extension of multiprogramming
  Interactive computer system for common use

$P_1 \quad P_2 \quad P_3 \quad \cdots \quad P_n$

Share CPU time
Through all the process

Users can interact with each program while its running → so fast are the switches.

* Real-Time and Distributed Systems

Real-Time → Meet-some deadline

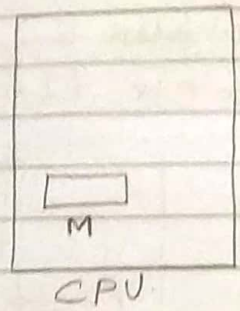
Distributed → Share Resources & Fault Tolerance.

Scanned by CamScanner

Dual-Mode of Execution

distinguish b/w Privileged Instr^n Exec / User Instr^n Exec

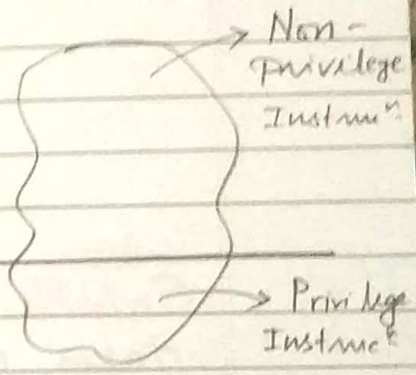
* A computer system can execute programs in Multiple Privilege Levels. ⇒ minimum of 2.



M = Mode Bit

0 → privileged, kernel, supervisory

1 → User



(*) CPU can execute a privileged instruction only when M=0

(*) If try to execute a privileged instruction @ M=1 :

- Exception will be generated TRAPPED

Say for Memory Protection — we need to load LBR & UBR

This has to be a privileged instruction

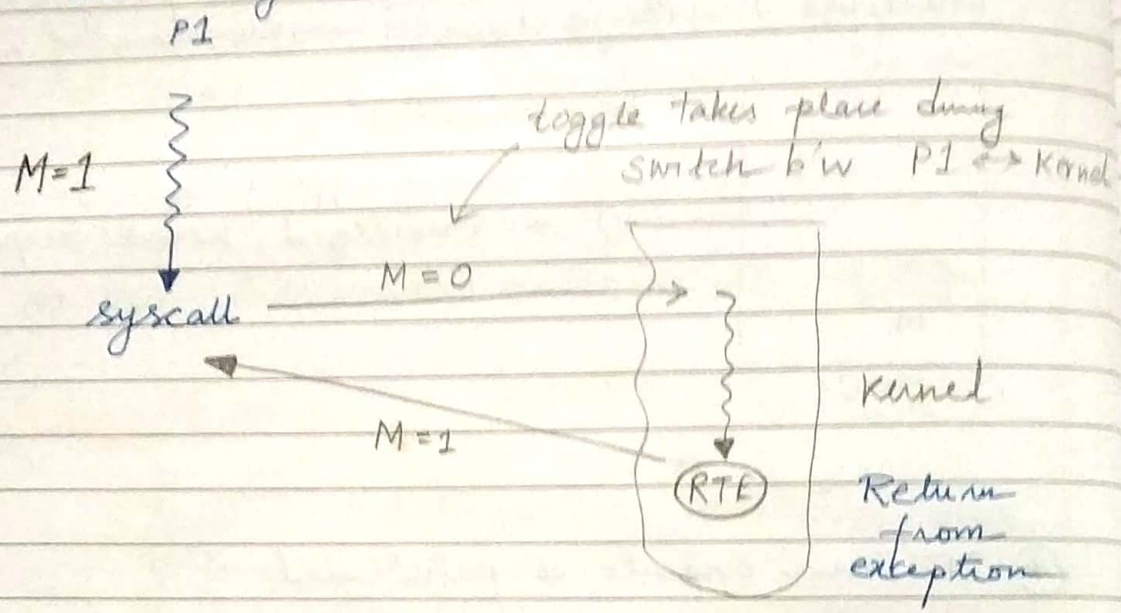
(Switching to privileged mode AT instruction ??)

System Boot ⇒ Hardware starts in Kernel Mode

op sys loaded → starts apps in USER MODE

Scanned by CamScanner

Changing Processor Mode



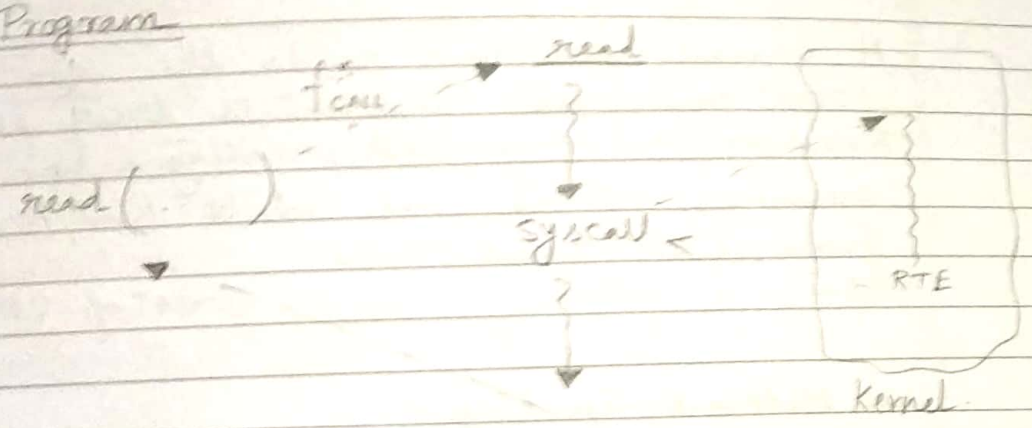
Switch from (USER → Privileged)
takes place during a
syscall instruction (non-priv itself)
interrupt

Switch from (Privileged → User)
takes place when we RTE → (non-priv itself)
Return from Exception

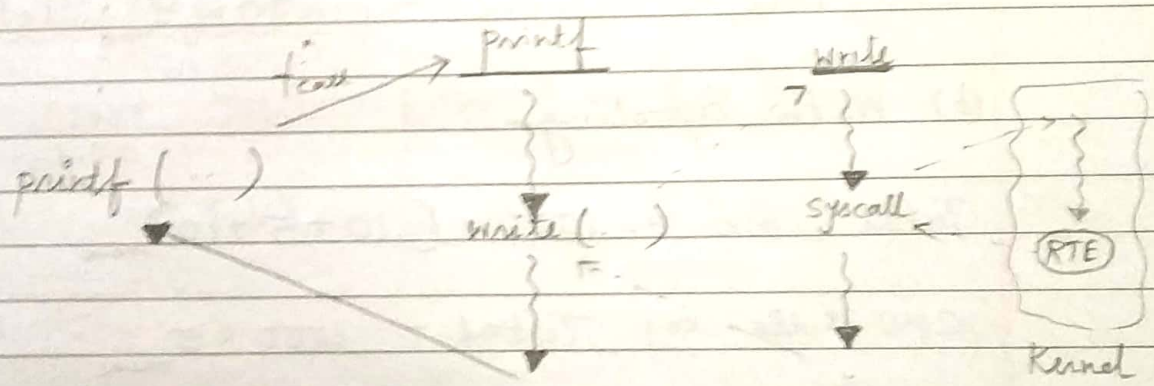
Scanned by CamScanner

'System Call' in a loose sense
- ask OS to perform tasks on user's behalf.

① Program



② C Program



These are not really system-calls; but we can call them that in a LOOSE-SENSE.

Types of System Calls

(a) Process Control : fork() , exit() , wait()

(b) File Manipulation : open(), read(), write()

(c) Inter-Process Comm : pipe(), shmget(), shmat()
shmdt(), shmct()
semget(), semop(), semctl()
signal(), kill()

Scanned by CamScanner

Problem 1: SPOOL

A payroll program reads the info about 1000 employees and prints their payroll. Reading a card and printing a line consumes 100 ms each while Read/Write from disk takes 10 ms. Salary processing requires 5 ms of CPU time

(a) Without Spooling

Total Time = $1000 \times (100 + 5 + 100)_{msec}$ = 205 s

CPU Idle = Total − $1000 \times 5_{msec}$ = 200 s

$\frac{40}{41}$ %

97.56 % IDLE

(b) With Spooling

Total Time = $1000 \times (10 + 5 + 10)_{msec}$ 25 s

CPU Idle = Total − $1000 \times 5_{msec}$ 20 s

75 % Idle

I/O device utilisation improves.

(a) ~ 5 card per second.

as 1 card ≡ Read => 100 ms
Process => 5 ms
Write => 100 ms
} 205 ms

(b) ~ 9 card per second.

as 1 card ≡ Read => 100 ms
Write to Disk => 10 ms
} 110 ms

Scanned by CamScanner

Problem 2: Multiprogramming

Two Jobs

Job 1 :

CPU	I/O
10	30

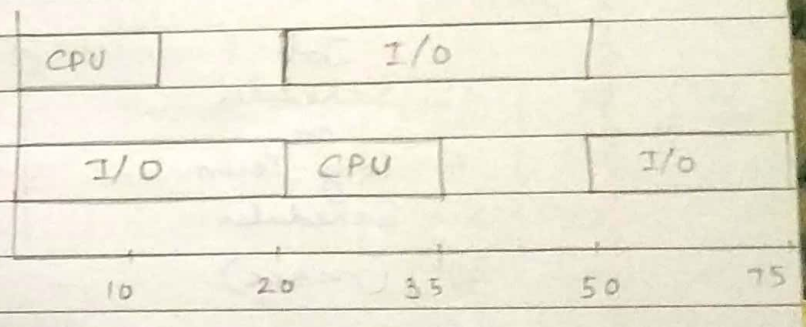
Job 2 :

I/O	CPU	I/O
20	15	25

Batch Processing : Time = (10+30) + (20+15+25)
= 100 s

Multi-programming :

75 s

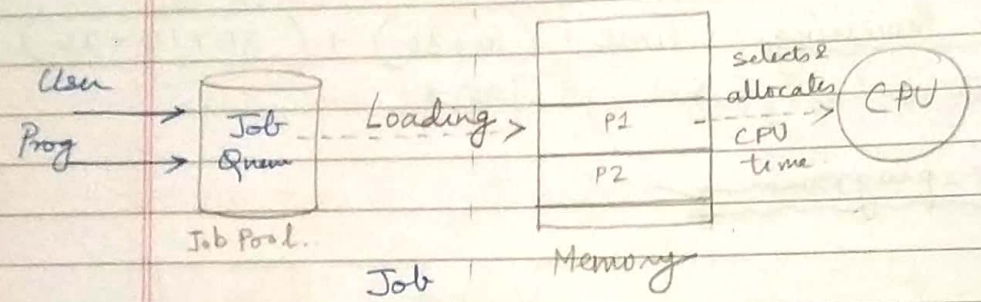


Scanned by CamScanner

PROCESS

* It is a (program) in execution — active entity
  (program → passive)
  - Different from a program
  - Changes its state over time

* CPU is switched b/w several processes
  - context switch



Job Scheduler or Long-Term Scheduler (~sec)

Process Scheduler or Short term schedule (~10ms).

Program ⇒ process ; when an exec file loaded into memory

Job Scheduler → Degree of Multiprogramming
→ no. of processes loaded in Memory

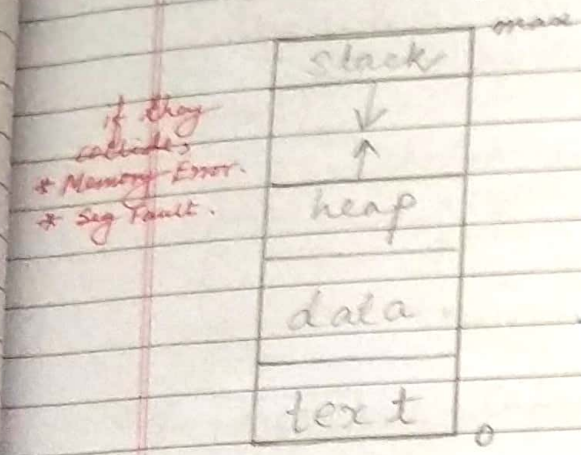
Medium ~ term Scheduler
Reduce degree of programming
Remove processes from Main Memory
SWAPPING
- Improve process mix
- memory reqts

Processes are Swapped Out; (status saved)
then Swapped In (status restored)

if they collide,
* Memory Error.
* Seg Fault.

Scanned by CamScanner

Structure of a Process.



if they collides,
* Memory Error.
* Seg Fault.

text : program code

data : global variables

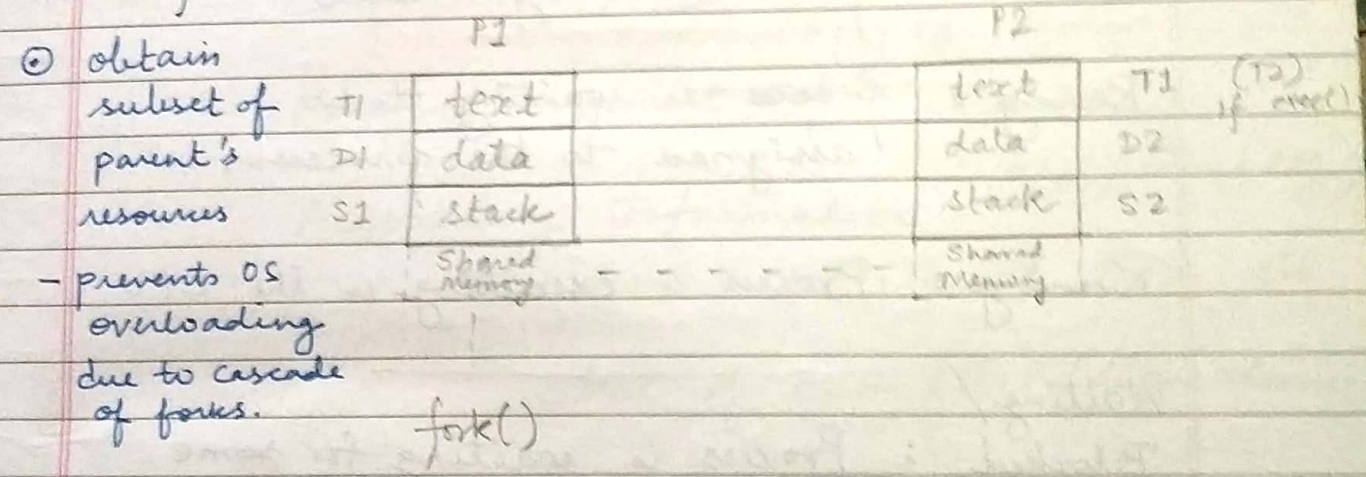
stack : store Return Address, Local Variables, Parameters
temporary data

heap : dynamically allocated data space.

fork() – creating children processes

⊙ obtain subset of parent's resources

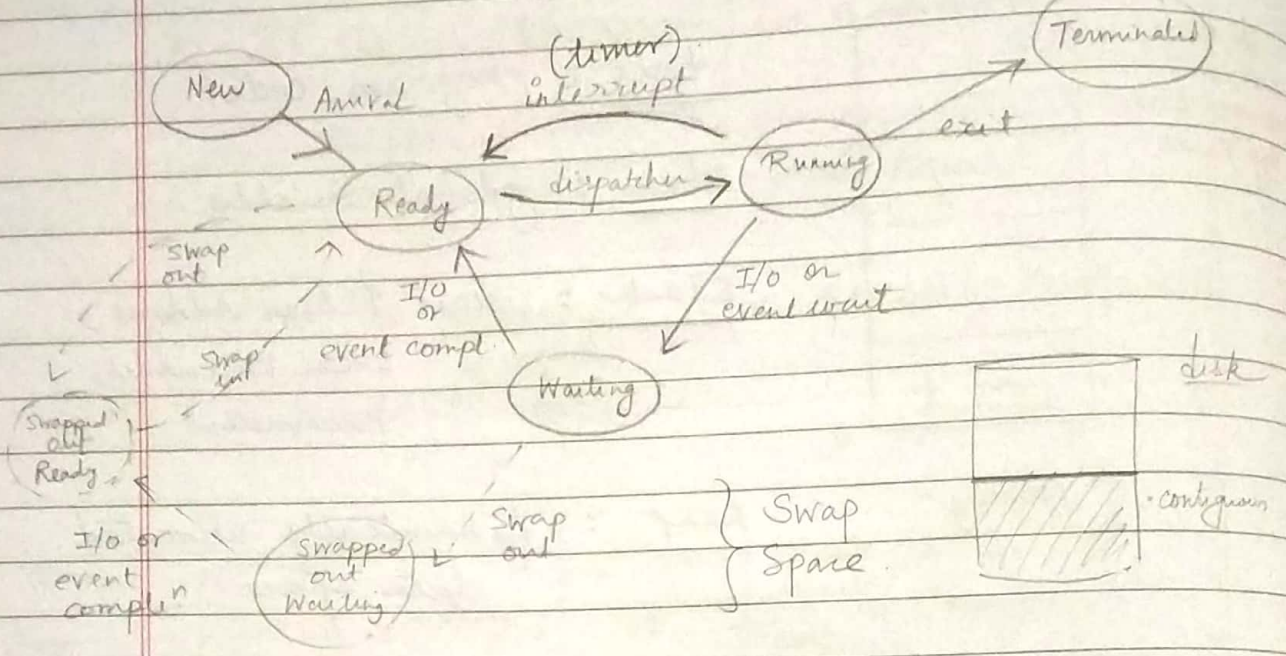
– prevents OS overloading due to cascade of forks.



(·) Here,
D1 copied into D2;

(·) In case an exec() is called in P2; the structure changes to that of the program (process) specified inside exec().

PROCESS STATES - defined by current activity



New : process is being created

Ready : process is waiting to be assigned to the processor

Running : Process is executing in the CPU

Waiting/ Blocked : Process is waiting for some event to occur
(I/O completion, read from pipe, signals)

Terminated : Finished execution
(•) Removed from all queues.
(•) PCB & Resources destroyed

Swapped Out Waiting -

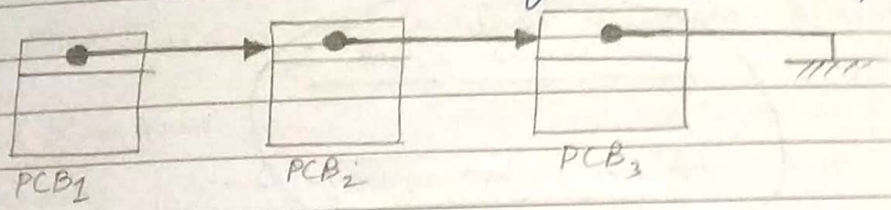
Swapped out Ready -

important for correct exec of pr

Scanned by CamScanner

PROCESS CONTROL BLOCK (PCB)

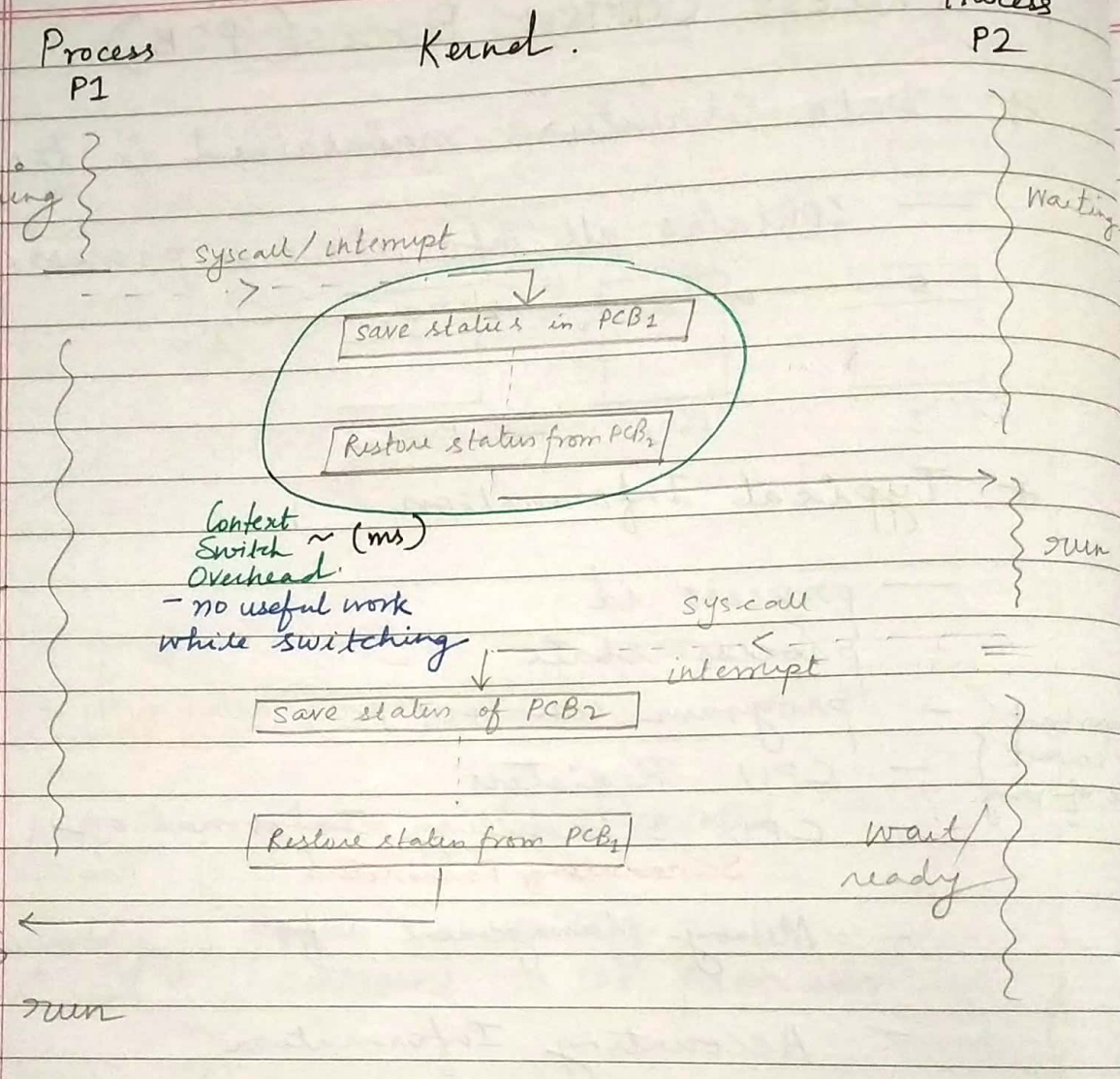
* Data Structure maintained in the kernel
  - contains all info about a process.



* Typical Information
  - process id
  - process state
  - program counter (PC) address of the next instr. to be executed in memory
  - CPU Registers
  - CPU Scheduling Information
    Scheduling Parameters (eg. Process Priority)
  - Memory Management info (page tables, base/limit reg)
  - Accounting Information
    CPU time used, total time etc
  - I/O info
    list of open files / input device

important for correct exec of prog.

Scanned by CamScanner



CONTEXT- SWITCH

What is stored? each time

- Program State
- Program Counter — Stack
- CPU Registers — data, heap
- Memory Manage Info

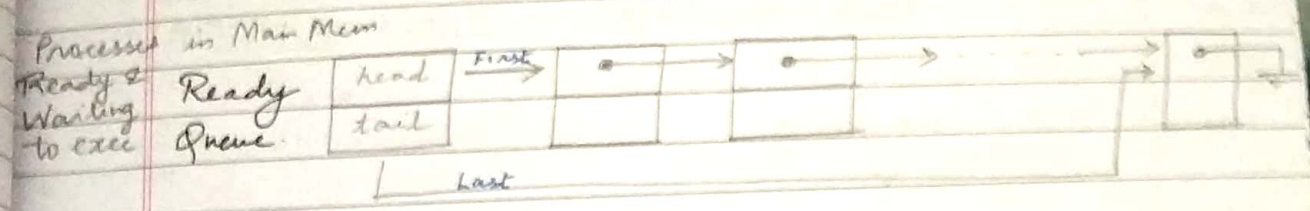
But I guess, load पूरा ही होता है।

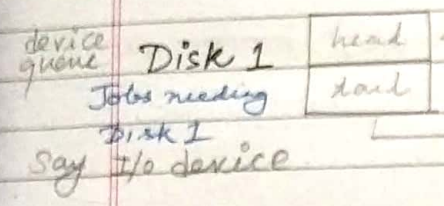
Scanned by CamScanner

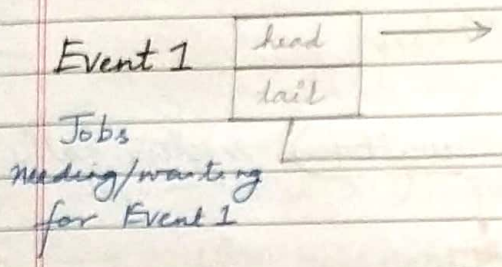
Process Scheduling

* Selects one of the processes in the Ready State to use the CPU next

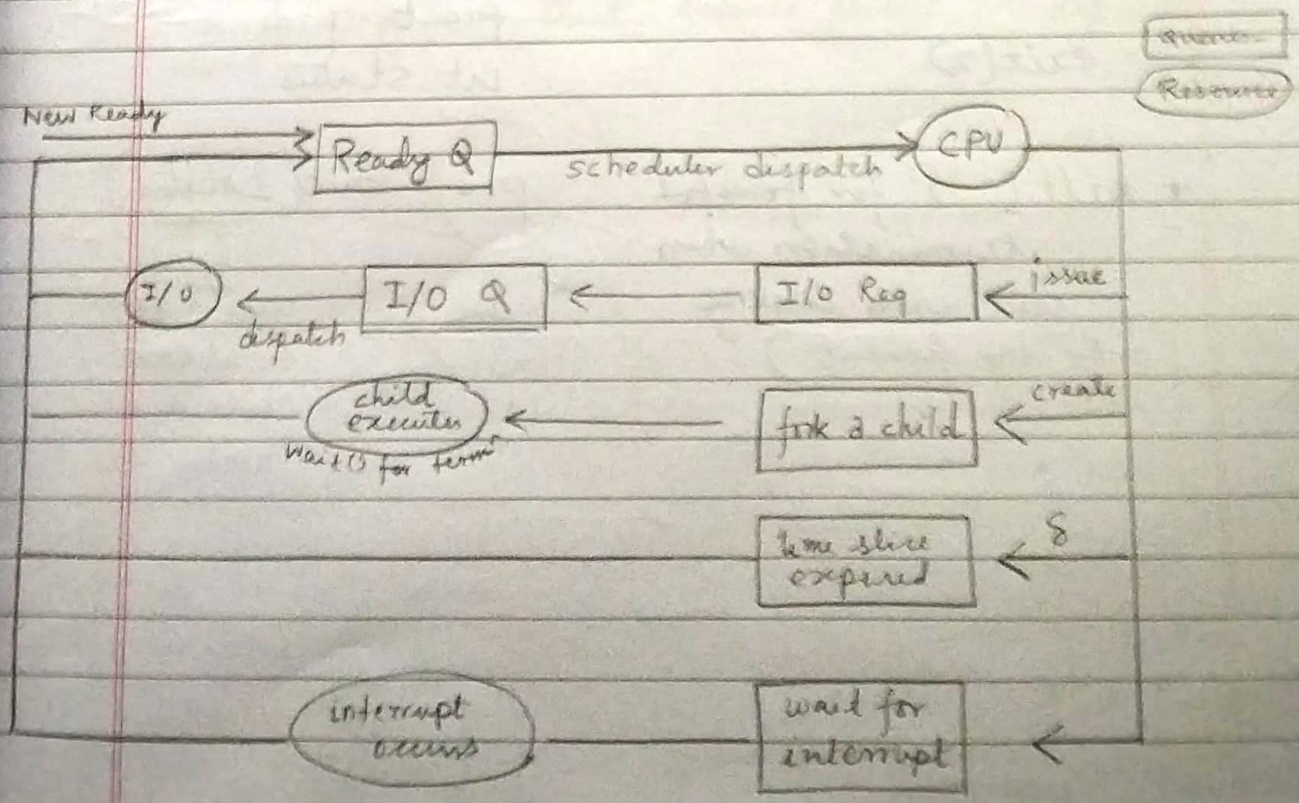
* Multiple Scheduling Queues Exist







At TERMINATION (o) removed from all the queues
(o) PCB, and Resources --> deallocated



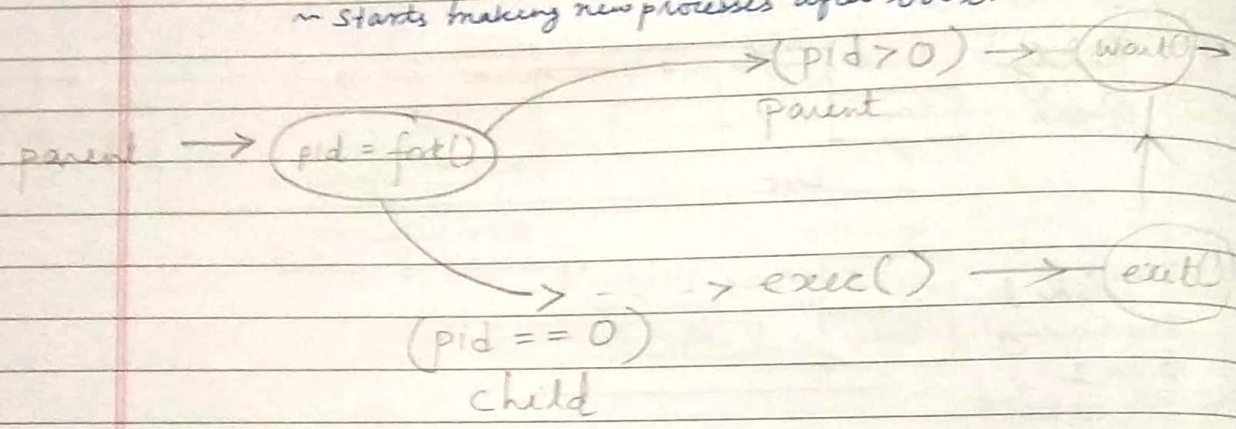
Scanned by CamScanner

Operations on Processes

* UNIX System

(a) Process Creation

- every new process is created using fork()
- the process to start is called 'init' pid=1
  ~ starts making new processes after boot.



(b) Process Termination

* We use exit() and wait() system calls

child	parent
⋮	⋮
exit(2)	pid_t pid int status

move away from Ready Q while child is executing

* kill( ) for forceful termination when sent with SIGKILL (only by parent).

pid = wait(& status)

pid: pid value of terminating child

& status: allows parent to obtain child's exit status too

Scanned by CamScanner

ZOMBIE PROCESS & ORPHAN PROCESS

kill( ) has no effect

* A process that has terminated, but the parent has not called wait( ) yet.
  - child pid still exists because parent will need it for wait( ) in File Descriptor table

* The parent does not call wait( ) and terminates, but child is already/still executing

Zombie → Orphan

↓

init will become the default parent of the orphan processes.

— periodically invokes wait( ).
~ enable any orphan to send its exit status & be releived from ~~the process table~~

How to kill( ) the Zombie Process?

→ kill( ) does not work, no effect
* end the parent → either it waits for Zombie
  or Zombie becomes Orphan
  if Orphan → init becomes parent by default

Scanned by CamScanner

Inter-process Communication (IPC)

- multiple processes using forks()
- may want to communicate with each other

* Processes Run <u>Concurrently</u> in the OS
  time sharing loop
  10 ms - 10 ms - 10 ms

Concurrent ≠ Parallel
@ instant only one process

2 CPU's; 2 processes in parallel at same time

→ may be <u>Independent</u> or <u>Co-operating</u> processes

If it cannot affect or be affected by other concurrent process.

may SHARE DATA

* How to implement co-operating processes?

(a) Using shared memory
(b) Using message passing
— pipe(), named pipe(), RPC()
Socket() → across machines

(A) SHARED MEMORY

Process A		Process B		Syscalls
T1	text	T2	text	shmget
D1	data	D2	data	
S1	stack	S2	stack	shmat tach
				shmdt ach
Shared Memory	<= SM =>	Shared Memory		shmctl

* attach shared memory segment to a process

Sample Code

```
char
int s

key =
shmid

if (
{
m
A
s
}
else
{
B
}
shm
```

Scanned by CamScanner

Sample Code Segment

```
char *myseg;
int shmid;

Key = 7329;
shmid = shmget(Key, 250, IPC_CREAT | 0666)

if( fork()==0 )
{
    myseg = shmat(shmid, NULL, 0);
A   : } make changes in *myseg
    : } store automatically in SHM
    shmdt(shmdt)
}
else
{
    myseg = shmat(shmid, NULL, 0);
B   shmdt (shmid)
}
:
shmctl(shmid, IPC_RMID, NULL)
```

an unique id

size in bytes

110 110 110
user group other

flag

MODES

OCTAL

Attach 'shmid' to address space

detach it

destroy shm

Shared Memory

- faster than passing messages → involves lot of syscalls each pass.
- Sys Calls required only to setup 'shm' region after which, routine access.

Message Passing

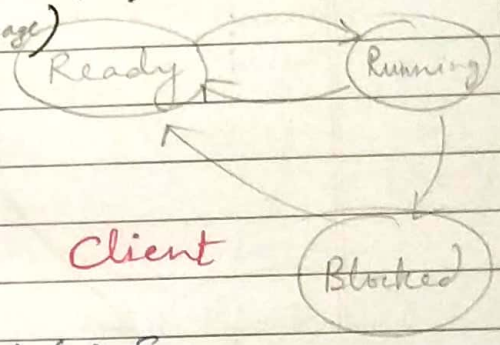
- useful for smaller amounts of data
- distributed systems' implem is simpler
- each message pass is basically a sys call.
  - needs lot of Kernel intervention
- Multi-Core System better performance
  shm → cache coherency iss

Inter Process Communication

* Shared Memory
* Message Passing

Two or more processes can exchange messages
- Following functions are required

send (Recipient, message)
receive (Sender, message)



Example :

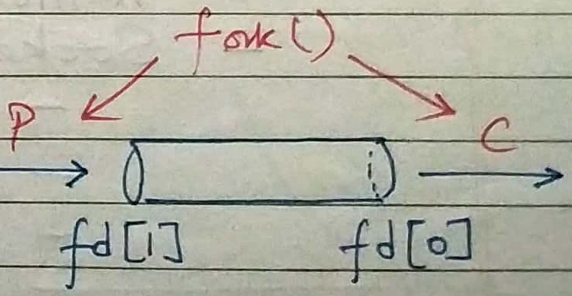
Producer	Client
while(1) {	while(1) {
produce an item	receive ( )
send( )	consume item

if both these calls are blocking, then no synch problem

* Primitives supported by Unix/Linux

(a) Unnamed Pipes

Communicating processes must share "fd" Related process
— Parent-Child
— Sibling



Controls the weight of the most recent bursts (α)

Scanned by CamScanner

CPU Scheduler (Process Scheduler)
Short - term Scheduler

* Decide which process is to run next, once the CPU becomes available
* Scheduling decisions are taken :

(a) Process switches from
RUNNING → WAITING / BLOCKED
- I/O Request
- parent invokes wait() syscall

(b) Process Switches from
RUNNING → READY
- Timer Interrupt

(c) Process Switches from
WAITING/ BLOCKED → READY.
- Completion of I/O
- Terminate of Child.

(d) Process Terminates
- exit() syscall

* CPU - Scheduling Algorithm

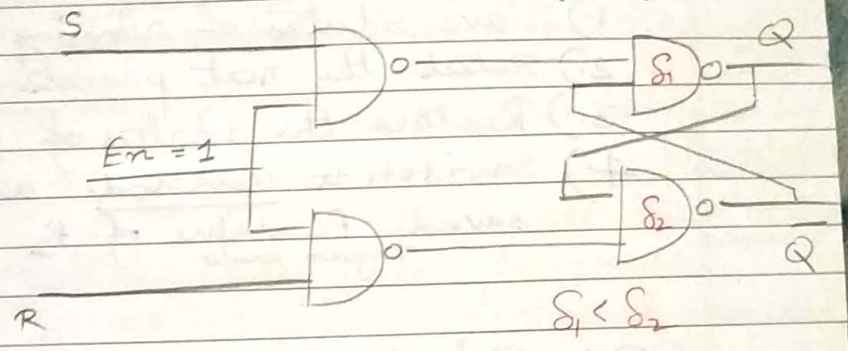
- Non- Pre Emptive - Never take CPU away from a Running Proc UNLESS, it voluntarily relinquishes it. (a), (
wait() or exit()

Pre Emptive - The CPU can be forcibly ta away from a running proc
— may lead to Race Condition (
access to shared data
CO-ORDINATION
SYNCHRONISATION
— OVERHEAD
Critical Section Business

What is race condition?

S-R flipflop

S	R	Q_n
0	0	Q_{n-1}
1	1	?



- Final Result of Computation depends on the RELATIVE ORDER of EXECUTION.

Example

B = 5000

Process 1		Process 2		
1.) LW	R1, B	(4) LW	R5, B	
2.) ADDI	R1, R1, #2000	(5) SUBI	R5, B, #1000	
3.) SW	R1, B	(6) SW	R5, B	

Order of Execution	Final B
1, 2, 3, 4, 5, 6	6000
4, 5, 6, 1, 2, 3	6000
1, 2, 4, 5, 6, 3	7000
4, 1, 2, 3, 5, 6	4000

We need to make {1,2,3} and {4,5,6} blocks ATOMIC.

Controls the weight of the most recent

$$\tau_{n+1} = \alpha t_n + \alpha(1-\alpha)t_{n-1} + \dots \alpha(1-\alpha)^j t_{n-j} + \dots (1-\alpha)$$

* Coefficients of successive terms get smaller and so
  → more importance to more recent hist

$\alpha = 0$ :: Recent History has no effect; $\tau_{n+1} = \tau$

$\alpha = 1$ :: only the most Recent has effect; τ_{n+1}

[c] Pre-Emptive SJF

Shortest Remaining Time Next (SRT)

SJF * Scheduling Process will take place ONLY when a process completes its CPU burst

SRT * Scheduling Process takes place EVERY time when a new process arrives, in addition to when a job completes its CPU Burst

[D] Priority Scheduling

* A priority value is associated with proc
  — CPU is allocated to process with highest priority
  — Process with same priority are run in FCFS mode.

* Statically assigned priority
  — may lead to indefinite waiting : STAGNATI
  If there is steady stream of High Priority Process

Aging - gradually increase priority of system process as the READY WAITING TIME increases

— DYNAMIC PRIORITY.

Scanned by CamScanner

Linux:

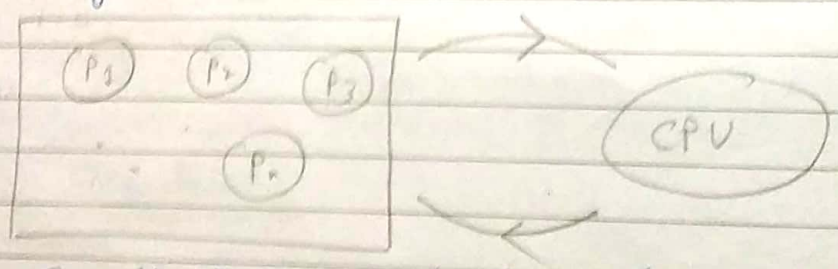
* All processes are assigned a static priority value
  - Range : -20 to 19
  - Smaller value ⇒ higher priority
  - Default value is 0

* How to change process priority?
  - Two commands: nice and renice
  ⇒ users can only decrease priority.
  ⇒ root can increase priority.

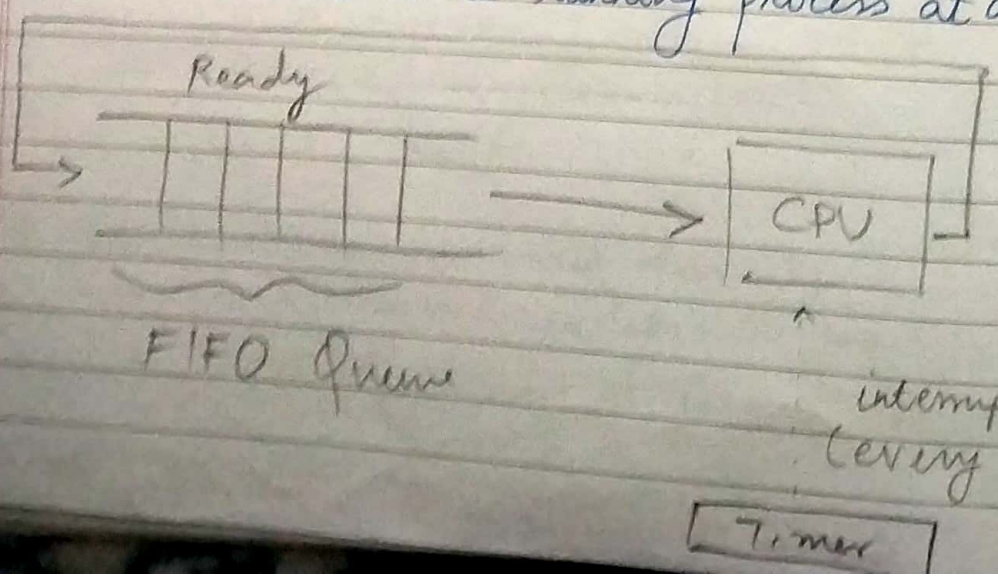
[e] ROUND - ROBIN SCHEDULING

* Pre-Emptive
* Designed for interactive (time sharing) systems

* Good Response Time is desired.



* Small Amount of time (~ms) allotted to a running process at a time



One a process gets the CPU

(a) process has a CPU burst less than δ
- releases CPU voluntarily

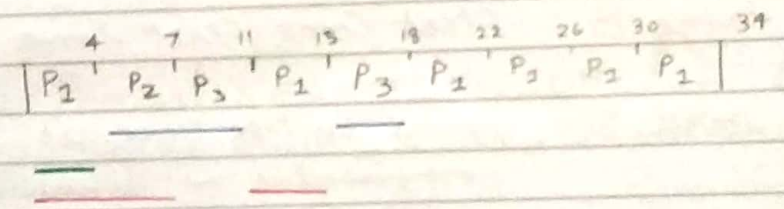
(b) process has a CPU burst greater than δ
- timer goes off to result in an INTERRUPT : CONTEXT SWITCH

- even if an UNTIMELY switch occurs, the TIMER is always RE-INITIALISED to ZERO.

Example

δ = 4ms

Process	P_1	P_2	P_3
ARRIVAL TIME	0	0	0
CPU_BURST	24	3	7



$$\text{Average Waiting Time (AWT)} = \frac{P_1.\text{wait} + P_2.\text{wait} + P_3.\text{wait}}{3}$$

$$= \frac{(3+4+3) + (4) + (3+4+4)}{3}$$

$\left(\frac{25}{3}\right)$

~ context switch overhead
→ Multiple Context Switches Occur
- Register Values Storing
- File Descriptors ; open, close

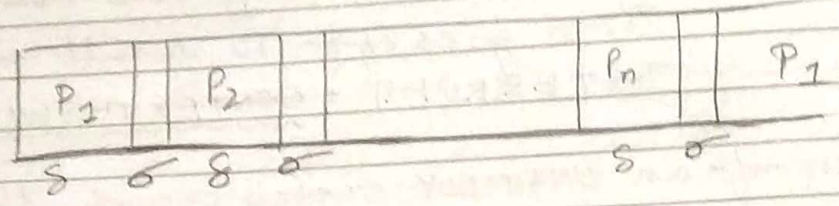
SJF $\left(\frac{\text{[illegible]}}{3}\right)$

$\left(\frac{10}{3}\right)$

n processes in the READY QUEUE, sufficiently large CPU Bursts

$\delta \rightarrow$ time quantum

$\sigma \rightarrow$ context switch overhead.



Max Waiting Time for getting CPU next execu^n cycle $= (n-1)\delta + n\sigma$

Very large δ : First Come First Serve

Very small δ : Context Switch becomes comparable ~ dominant

Typically; $\delta \gg \sigma$

$\delta \Downarrow \sim 10ms$ depends on scenario environment we work in

$\sigma \Downarrow \sim 10\mu s$

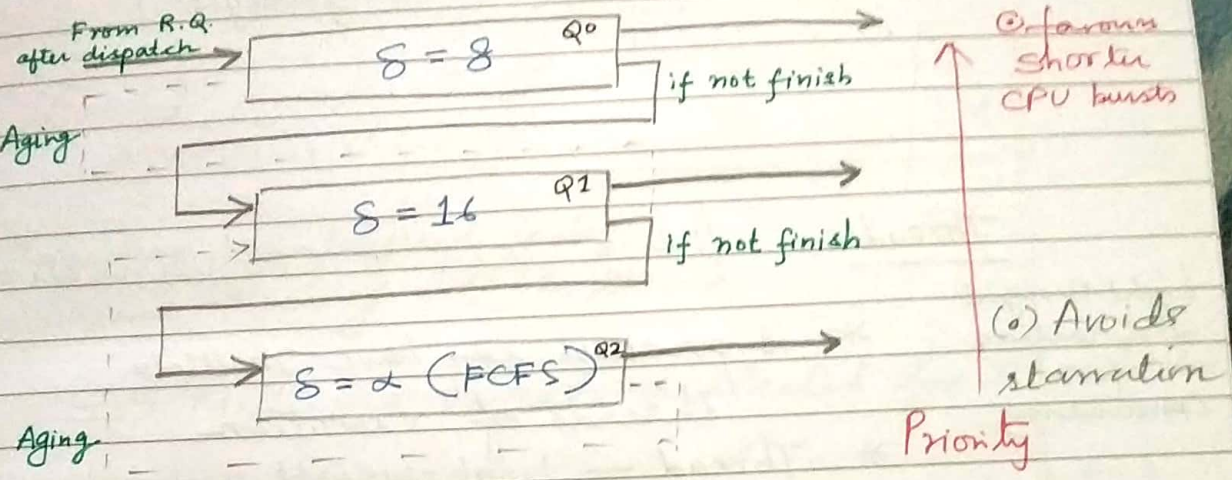
Timer : Re-initialisation => privileged mode

Scanned by CamScanner

Multi-Level Feedback Queue Scheduling

~ Variable CPU time allocated depending on Priority

(*). Multiple scheduling queues, different priorities



(*) Process RESIDES in ONE of the THREE Ready Queues → Q_1, Q_2, Q_0.

(*) Priority $Q_0 > Q_1 > Q_2$

- process from Q_i can execute iff $\forall j < i$; Q_j are empty.

if a new process arrives at Q_i, all $Q_{j>i}$ will get preempted

(*) Q_0, Q_1 : Round Robin,
Q_2 : FCFS

(*) If the $\boxed{\text{CPU Burst} < \delta}$ → process remain in the same queue

(*) If a Process waits too long, in a queue. $(i \neq 0)$
it is moved back to Q_{i-1}.

- this form of aging prevents starvation.

Scanned by CamScanner

Multi Threading :

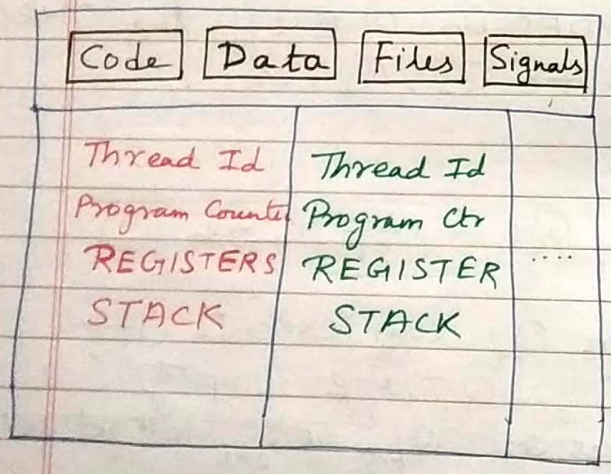
Process - a program in execution, with a single thread of control.

- Traditional (or heavyweight)

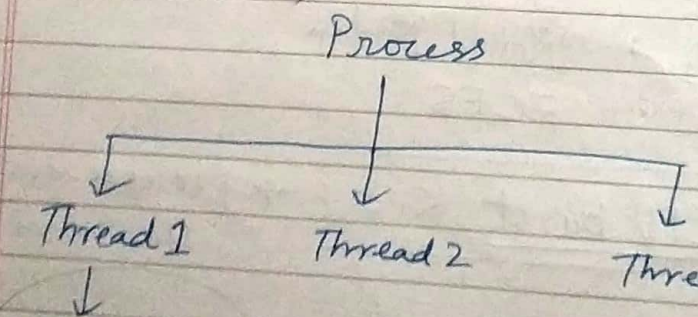
Threads - basic unit of CPU utilisation

Light weight abstraction for concurrency

* A process can have multiple threads of execution
* Thread — Light-weight process



Multi-Threading Process



Thread ID
PC
Registers
Stack

only this is stored for each THREAD, rest is common to all

Example

(a). WEB

Share

Thread

the

(b) W

ded

What's the problem in the fork() paradigm?

Scanned by CamScanner

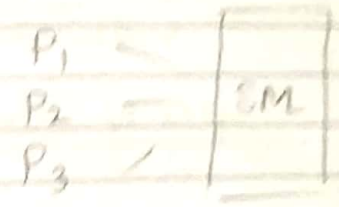
Examples

(a). WEB BROWSER

- Display
- User input
- Downloading

} appear to be go on, on the same time

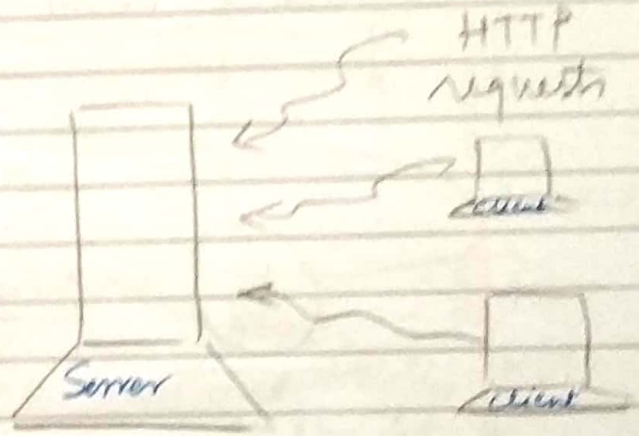
Shared Memory involves syscalls(), locks



Threading has Inherent feature that Memory will be shared

==> Multiple threads, dedicated for these tasks

(b) WEB SERVER



- see new process
  : fork() child
  dedicate child to it -

What's the problem in the fork() paradigm?

the number of processes may become VERY HIGH.
→ system memory overload, {copying of text, data & stack.}

Multithreading - threads are spawned on incoming requests

⊙ ONLY ONE MAIN PROCESS, rest are Threads dedicated to each connection

on arrival of new request, a new thread shall be created and assigned / dedicated for that request.

Scanned by CamScanner

Page 148

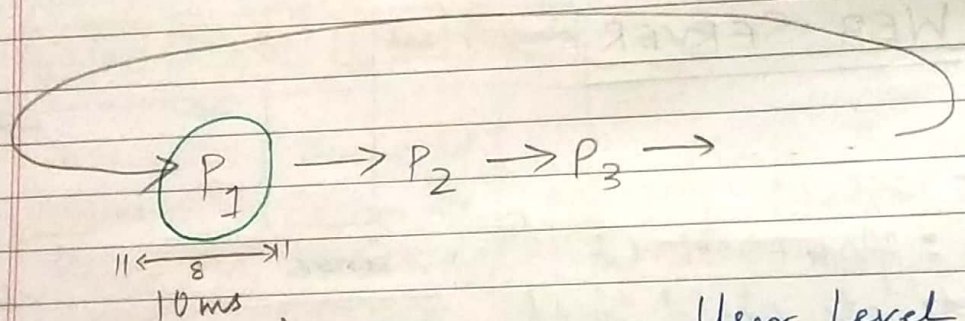
Benefits of Multi Threading

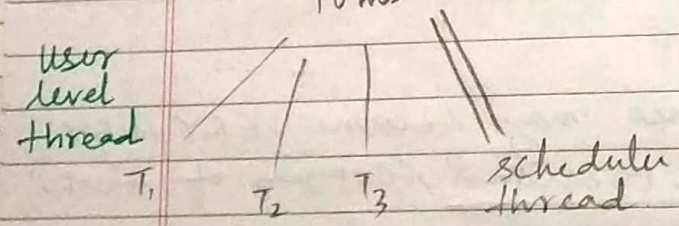
(a) Context Switch is much faster / cheaper
(b) Creating new threads takes much less time as compared to creating a new process
(c) Better resource sharing among peers
(d) Better scalability for multiprocess system
performance when work load on system increases

Types of Threads –

(a) User-Level Thread – no help from KERNEL required
Language Construct

(b) Kernel-Level Thread – support & manage direct by KERNEL (operating system)





User Level Threads are managed by Thread Libraries; they are invisible to the Kernel.
ULT need to be mapped to KLT; to run on a CPU.

Kernel
– for kernel, only P_1 is known
the switching of threads is taking place by the USER

language Hand Code for Multi Threading

Managed by the Thread library

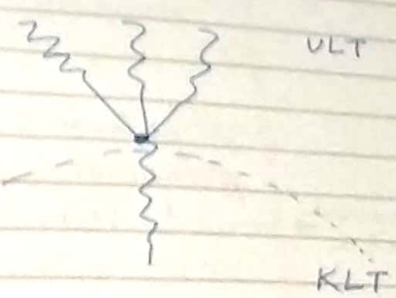
Restrict number of threads supported at an instant

ULT/
(a) Many
if (u
i.e
if a
syscal
also
No Para
(b)
• Re
(c)
(a)
(b)

Scanned by CamScanner

ULT / KLT Correspondence

(a) Many - to - one Model.



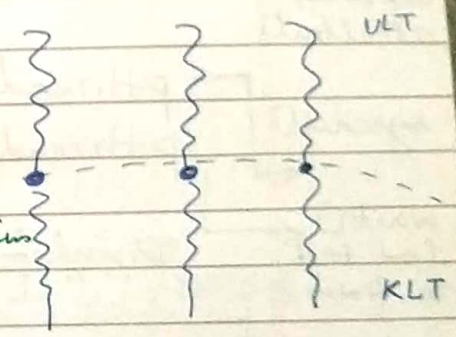
Managed by the Thread Library

if (ULT) cause sys call()
i.e (KLT). →

if a ULT makes a blocking syscall(); all other ULTs also get blocked.

No Parallelism or Concurrency Possible.
— Hardly used;

(b) One - to - One Model.



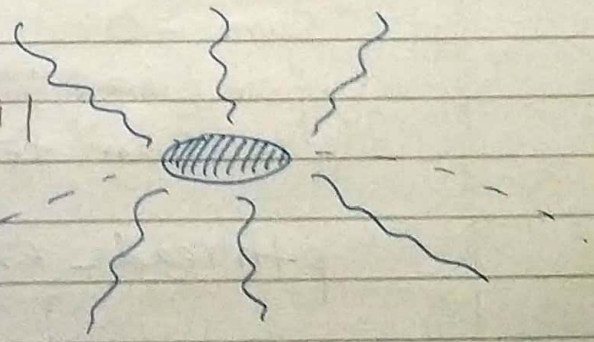
- Rectifies the previous problem — provides more concurrency / parallelism
- No. of KLT = No. of ULT
  Increases that overhead.

Restrict number of threads supported at an instant

— LINUX / WINDOWS implement

Multicore ds fast suitable; KLTs can be in different CPUs

(c) Many - to - Many



— Solaris 9 ds पहले।
Sun MicroSystems

(a) → As many threads as possible, but schedule only one AT A TIME

(b) → Limit on number of threads I can schedule

(c) → NO RESTRICTION

Scanned by CamScanner

POSIX pthread Library

Standard — thread library

```
#include <stdio.h>
#include <pthread.h>
int sum;
void *runner(void *param);

int main (int argc, char * argv [])
{
    pthread_t tid
    pthread_attr_t attr;
    pthread_attr_init ( &attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join ( tid, NULL);
    printf ("\n sum = %d", sum);
}

void * runner (void * param)
{
    int i, N = atoi(param);
    sum = 0;
    for (i=1; i<=N; i++)
        sum += i;

    pthread_exit (0);
}
```

Shared Data

Returns the default attribute

Thread ID

execute function

fun param

syscall()

wait() for tid to finish

only 1 thread computes $\left(\sum_{i=1}^{N} i\right)$. ./a.out 12 ✓

Scanned by CamScanner

Suppose we wish to create 10 threads

```
#define N_THR 10
.
.
.
pthread_t mythreads[N_THR]
.
.
.
for(i=0; i< N_THR; i++)
{
    pthread_create(mythread[i], & attr...)
}
.
.
.
for(j=0; j < N_THR; j++)
{
  pthread_join(mythreads[i], NULL)
}
```

Thread execution begins at "runner" functions

```
pthread_mutex_t
pthread_mutex_init(*, NULL)

pthread_mutex_lock( )
pthread_mutex_unlock( )
```

Scanned by CamScanner

Thread Scheduling using pthread

Pre-Emptive

(a) Process Contention Scope (PCS)

- Typically exist for MANY-to-ONE and MANY-to-MANY

Competition for CPU time amongst threads of same process

- The thread library schedules the threads to run within the time allocated for the process

(b) System-Contention Scope (SCS)

- one to one Model.
- Used for scheduling Kernel-level threads

Competition for CPU takes place in all threads

- Kernels share CPU time among all such threads

* Contention Scope Values

PTHREAD_SCOPE_PROCESS
PTHREAD_SCOPE_SYSTEM

* Two functions :

```
pthread_attr_setscope(
        pthread_attr_t *attr,
        int scope);
```

set whatever depends on my OS if it supports Kernel Level Threads

```
pthread_attr_getscope(
        pthread_attr_t *attr,
        int* scope);
```

Scanned by CamScanner

Thread Cancellation

pthread_join() → wait for thread to end

We need a MASTER STOP to kill threads in b/w

pthread_cancel(pthread_t tid);
- by default tid terminates immediately

if some FLAGS defined/set during making,
then it may wait for completion b4 term

Asynchronous Cancellation – Immediate termination of target thread.

I guess Flags ही help है

Deferred Cancellation – Target thread checks periodically, if it shd terminate → allows orderly termination.
These checks may be performed at points called **CANCELATION POINTS** → program part where its safe to terminate

Safe ??

- Thread might be in its Critical Sections unorderly termination may cause logical errors.

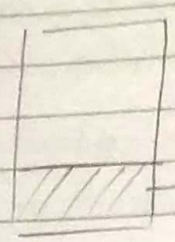
- Unorderly / Asynch termination AT TIMES leads to, not properly being able to Recover the Allocated Resources to that (Target) Thread.

Scanned by CamScanner

SIGNAL HANDLING

* Mechanism through which, the Kernel notifies the process/thread that some EVENT has occured.

Process at PCB



→ list of pending signals.

When Process (1) gets the CPU back, CPU check for pending signals stored in Process Control and takes the necessary actions.

* Signals defined under POSIX

SIGABRT → Abort
SIG BUS → Bus Error
SIG ILL → Illegal Instruction
SIG KILL → Kill Process
SIG QUIT → Terminal Quit
SIG SEGV → Invalid Memory Reference
SIG INT → Interrupt (^C)
SIG USR1, SIG USR2 } → User defined symbols.

* For most signals, default signal handlers exist – run by Kernel.

Synchronous Signals – those signals generated by same running process' events
%0, /0, Seg Fault

Asynchronous Signals – those signals generated by an event external to the running process.
^C

Signal

#incl

void

{

}

ma

{

}

Som

Scanned by CamScanner

Signal Handler in C.

```
#include < signal.h >

void sig_handler ( int signo )
{
    if ( signo == SIGINT )
    {
        printf ( " Received SigInt \n" );
    }
    .... signal ( ... )
}

main ()
{
    signal ( SIG INT , sig_handler );
    while(1)
    {
        sleep(1);
    }
}
```

user-defined OVERRIDES default kernel space stuff

=> in case discarded, it gets reinstalled.

installs the signal handler

Some system. → signal handler gets discarded,
→ signal handler stays

Scanned by CamScanner

How to send a Signal to a process or thread ?

$P_i \longrightarrow P_j$
from to

$t_i \longrightarrow t_j$

kill (pid-t pro-id , int signal) ;
⇒ send signal to pid.

pthread-kill (pthread-t tid, int signal)
⇒ send signal to tid.

When it comes to Multi threads; who should the signals be delivered to ? Which threads ?

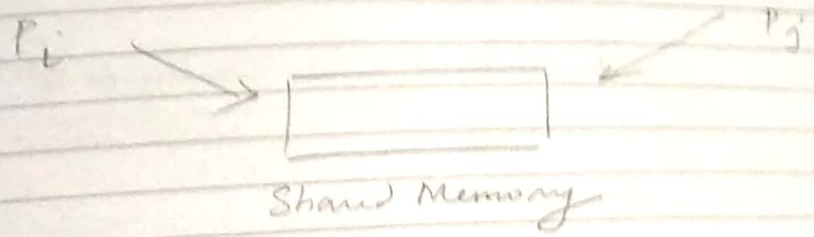
<u>Synchronous</u> → to the same process which caused it

<u>Asynchronous</u> → depends on type of signal.
^C should be sent to all threads

(•) Threads can specify which signals to receive
which signals to block
— Signals are sent only to those threads which do not block it.

Scanned by CamScanner

PROCESS SYNCHRONISATION



Race Condition → when two concurrent processes try to access shm' their order of ACCESS may lead to diff results.

Low level instruc^n may not be atomic → get executed in an interleaved fashion

* Processes execute either concurrent / parallel

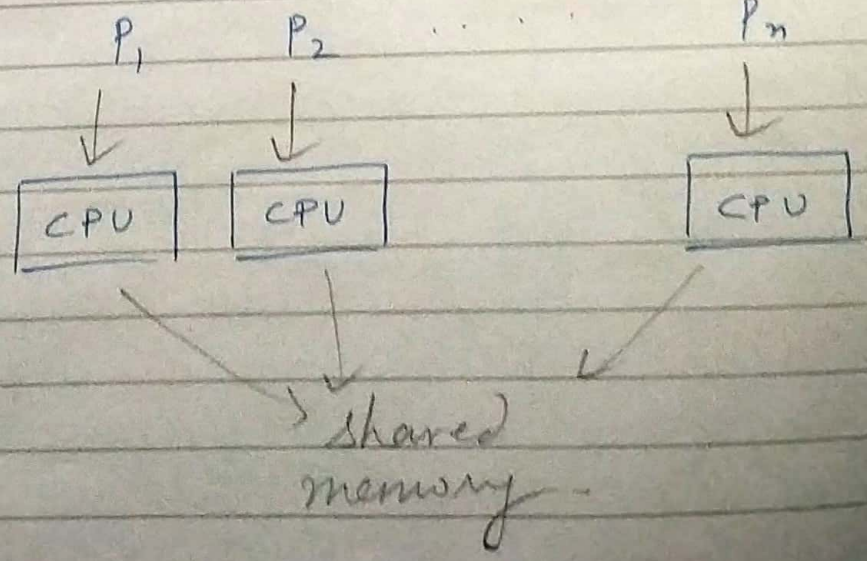
CONCURRENT

P_1 P_2 . . . . . P_n

[CPU] Time Sharing ⇒ Context Switches.

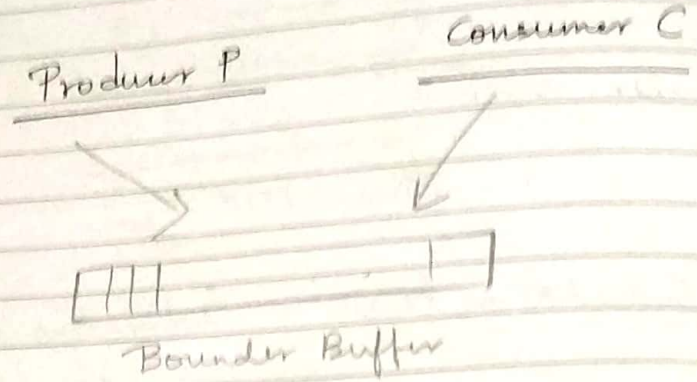
No idea about, when these context switches are happening.

PARALLEL



Scanned by CamScanner

Producer - Consumer Problem




```
#define BUFF_SIZE 50
typedef struct {
} item
item buffer [BUFF_SIZE]
int in = 0
int out = 0
int count = 0
```

→ present in shared memory

* Concurrent (or Parallel) execution of P & C may Result in RACE CONDITION

Scanned by CamScanner

P

```
while(1){
    next_prod = ProduceItem();
    while( count == BUFF_SIZE );
    buffer[in] = next_prod;
    in = (in+1) % BUFF_SIZE;
    count ++;
}
```

```
LOAD  R1, 0(Count)
ADDI  R1, R1, 1
STORE R1, 0(Count)
```

C

```
while(1){
    while( count == 0);
    next_cons = buffer[out];
    out = (out+1) % BUFF_SIZE
    count --;
    ConsumeItem( next_cons);
}
```

```
LOAD R2, 0(Count)
SUBI R2, R2, 1
STORE R2, 0(Count)
```

Here; 'count' is updated in both;

	R1	R2	count ↓10
LOAD R1, 0(Count);	10		(10) init
LOAD R2, 0(Count);		10	
SUBI R2, R2, 1;		9	
STORE R2, 0(Count);		9	9
ADDI R1, R1, 1;	11		9
STORE R1, 0(Count);	11		11

Sol^n – Restrict Context Switch when 'shm' updation is going on.

Scanned by CamScanner

Critical Section Problem

* n Concurrent Processes $P_0, P_1, \ldots, P_n$
* Each P_i has a segment of code, called CRITICAL SECTION, where it updates shared data, variables, files etc.
* REQUIREMENT: When a process is executing its CS, no other process should be allowed to execute

* Structure of P_i :

```
while (1)
{
    <entry section>        → Request Entry to Critical Section
    <Critical Section>
    <exit section>         → Process has come out of CS
    Remainder Section
}
```

* Correct Solution to the CS Problem must →

(a) Mutual Exclusion - If P_i executes in CS, no other $P_{j \neq i}$ executes in CS

(b) Progress - If no process is executing in their CS, and some process(es) wish to enter CS, the decision as to who enters next will be taken only by processes that are not executing in their Remainder Section and their decision can not be postponed indefinitely → deadlock check

Kernel Structures prone to Race Conditions
(•) open file ti
(•) memory alle
(•) process list
(•) interrupt handl

(c) Bounde
the
enter
made
reques

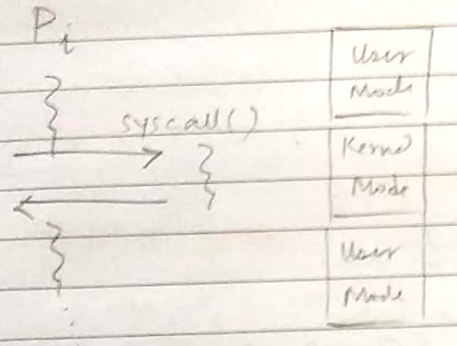
Critica
Kerne
P_i
(a) P
if

Scanned by CamScanner

(c) **Bounded Waiting** : There exists a ~~upper~~ bound (limit) on the number of times other processes can enter their CS, after a Process P_i has made a Request to enter CS and that request is granted. → starvation check

Critical Sections in the Kernels

Kernel's → routines — exec when some syscall() interrupt taken place.

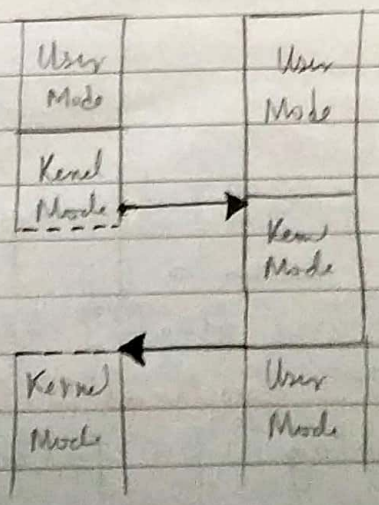


do we allow two processes to execute Kernel Mode Simultaneously ?

(a) **Pre Emptive Kernel** - Allows process to be pre-empted while it runs in Kernel Mode

Kernel Structures prone to Race Conditions ?

- (•) open file list
- (•) memory alloc
- (•) process lists
- (•) interrupt handling



(•) Designed carefully to ensure shared Kernel data are free from RACE CONDITIONS

Pre Emptive Kernels, despite of the need to tackle RACE CONDITIONS are preferred over Non

(•) [All benefits of Pre Emption]

(b) Non Pre-Emptive Kernel

Kernel-mode execution cannot be pre empted

⇒ No RACE Condition while executing Kernel routines as only one process active in Kernel at a time

Peterson's Solution

"लालाजी आंदोलन"

```
int turn = 0
boolean flag[2]
```

shared data

P_0

```
while(1)
{
entry { flag[0] = true;      -> it is ready to enter
        turn = 1;            -> if other process wants to enter, ...
        while ( flag[1] && turn == 1 );

        --- Critical Section ---

exit  { flag[0] = false;

        } Remainder Section
```

if other process is waiting, allow entry to CS

P_1

```
while(1) {
        flag[1] = true;
        turn = 0;
        while ( flag[0] && turn == 0 );
        --- Critical Section ---
        flag[1] = false
        Remainder Section
}
```

Scanned by CamScanner

flag[i] → P_i wants (ready to) to enter CS

turn → whose turn it is to enter CS

This can have only one value at a time } Mutual Exclusion } Happens through the value of this variable only

* Pure Software Solution
* Applicable to 2 processes only
  ⇒ difficult to generalise

* Busy waiting / Spin lock. → other process "SPINS" about the while(..) till it can enter CS.
  - waste CPU cycles which another process can use efficiently

Requirement Check - → See next pg

(a) → Critical Section only executes when process' ready
So flag[1] = True.
For both CS to be executed together, both the while(cond) should be false. Since the flags are true; it requires (turn == 0) and (turn == 1)
NOT POSSIBLE ⇒ MUTEX

below. (b) (c)

Problem - May fail in Modern Day Architecture (that use pipelines basically)

pipelining may → ⊙ Allow out-of-order execution for instructions that are independent of each other

while(flag[1] and turn == 1)
flag[1] → other process is READY
turn == 1 → other process allowed to execute

This "allowing other process to execute" is actually important, because this ensures that after one process executes, the other process (if ready) shall be allowed to execute; if its not (!ready) then only the execution of self CS takes place.

(loops over)

(a) Deals with Starvation this way. (atmost one wait ← bound)
(b) The way stuff goes, no Deadlocks possible either

* it is essential for flag[i] = True when process i sets turn = 1-i else MUTEX violation occurs.

Allow other process only when you are ready.

Eg

P_0	P_1
while(1)	while(1)
{	{
flag[0] = FALSE (init)	flag[1] = FALSE (init)
flag[0] = TRUE	flag[1] = TRUE
turn = 1;	turn = 0;
while(flag[1] && turn == 1);	while(flag[0] && turn == 0);
-- Critical Section --	-- Critical Section --
flag[0] = FALSE	flag[1] = FALSE
-- Remainder Section --	-- Remainder Section --
}	}

may inter change due to pipelining

entry

exit

Sequence

P_1 : turn = 0 ⟶ context switch

P_0 : turn = 1

P_0 : flag[0] = TRUE

P_0 : while(flag[1] && turn == 1);

turn = 1, but flag[1] = false so (P1) can't CS. So (P0) goes in exit sect.

⟶ enters CS (∵ flag[1] = false)

⟶ context switch

P_1 : flag[1] = TRUE

P_1 : while(flag[0] && turn == 0);

⟶ enters CS (∵ turn == 1)

pipeline ⇄ inter change

Here, due to the interchange the turn is not changed after CS happens

Mutual Exclusion gets violated.

Another way for the two while(..) to be true together is if say: One of the flags is false, and turn pointing to that process.

(flag[1] == false && turn == 1) ⇒ P0 enters CS.

Then Context Switch Happens; Only way for (P1) to enter CS is if (flag[0] → false) or (turn → 1) Now, the way our code is designed ↓ not possible already in CS → Ready

if (flag[1] is false), then to reach CS, turn will always need to be set to 0 in between. Hence this will prevent P1 → enter CS

Synchronisation Hardware

* A simple method

→ Consider a uniprocessor system

Solution

!! If system clock is updated by interrupts then no ↓

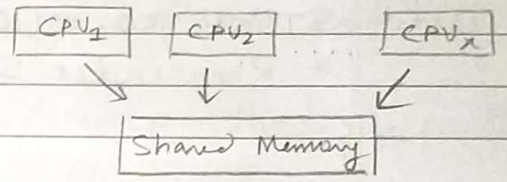
Disable Interrupt
* Critical Section *
Enabling Interrupt

} guarantees atomicity during CS exec

→ Difficult to extend for multiprocessor

time consuming to ~~be scal~~ to all processors

This because for Multiprocesor Systems need to disable interrupts FOR ALL PROCESSES.



x86 processors have a built in Disable Interrupt

* Bad Solution

- disturbing the scheduling algorithm for all the processors
- difficult and diminishes performance

* Special Instructions – Additions to the existing Instruction Set

(a) Test - and - set

- "Read Modify Write"
- Access Memory twice
- Longer Time

```
boolean TAS ( boolean * target)
{
    boolean rv = *target
    *target = TRUE
    return rv ;
}
```

return old value

By defaut Must be atomic, no mid interrupt b/w these instruc.

Egs: TAS R1 , 0(1000) : R1 = Memory[1000]
Memory[1000] = 1

Memory[1000] → 0 (lock)
→ 1 (unlock)

Acts as a Lock

Scanned by CamScanner

Initialy STORE R0, 0(1000)

P_1 P_2 . . . . . P_n

```
while(1)
{
L: TAS R1, 0(1000)
entry { if (R1 == 1)
          goto L
*Critical Section*
exit { STORE R0, 0(1000)
* Remainder Section*
}
```

⊛ TAS should be a atomic instruction
— Need Hardware Support for it

⊛ Can be extended to multiple processes

⊛ Busy waiting on Spin Lock
consumes CPU cycles, while waiting to enter

(b) Swap Instruction - Swap contents of register and memory location

$R_1 = 1$
Swap(R_1, 0(1000)) } TAS (R_1, 0(1000))

MUTEX LOCKS : POSIX Library → lock
count++
unlock

pthread_mutex_init()
pthread_mutex_lock(), pthread_mutex_unlock()
(CS)
pthread_mutex_destroy

Only related threads, can use ; Global variable is shared only by Related Threads

For others → Semaphores.

involves busy waiting

PROCESS SYNCHRONISATION

- Mutual Exclusion Problem
- Software Solution
- Hardware Solutions
- Mutex Locks.

SEMAPHORE – A general mechanism, using which, we synchronise processes

– It is an INTEGER VARIABLE that, apart from initialization; can be operated upon using two Atomic operations

(a) wait(s) → referred as P
(b) signal(s) → referred as V

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

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

Two types of Semaphores:

(a) BINARY SEMAPHORES – * 0 → locked
* 1 → unlocked.

~ MutEx Locks minus (~ no busy wt
~ no need for lock to be global or thread to be related ??)

(b) General / Counting Semaphore – any integer value

Scanned by CamScanner

Uses of Semaphores

(i) Mutual Exclusion

* Binary semaphore S
  semaphore S = 1

P_1	. . . . .	P_n
while(1)		while(1)
{		{
wait(S)		wait(S)
CS		CS
signal(S)		Signal(S)
RS		RS
}		}

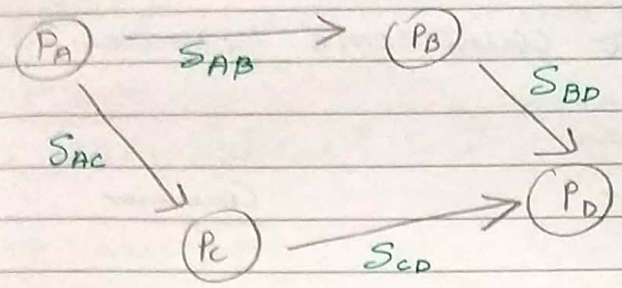
P_1 P_2 P_3

	S
(P_1) wants to enter CS	1
P_1 → wait(S)	1 → 0
P_1 enters CS	0
(P_2) → wait(S) ⇝ $\boxed{\text{BusyWait}}$	0
P_1 → signal(S)	0 → 1
P_2 enters CS.	1 → 0

if ot wants entry

(ii) Process Synchronisation

* Process precedence graph



A → B means B can't execute b4 A

⇒ For every precedence relation, define a binary semaphore → initially 0.

Semaphores $S_{AB} = S_{AC} = S_{CD} = S_{BD} = 0$

P_A	$P_{B/C}$	P_D
{	{ wait(S_{AB} / S_{AC})	{ wait(S_{BD})
⋮	signal(S_{BD} / S_{CD})	wait(S_{CD})
signal(S_{AB})	}	⋮
signal(S_{AC})		}
}		

Since, all 'Sems' are initialised to zero, unless one process signals at it, they will keep on waiting.

Scanned by CamScanner

(iii) Process Synchronisation

* General or Counting Semaphores
* Used to count some resource
  - buffer capa.

sembuff = 0

Producer	Consumer
⋮	⋮
signal(sembuff)	wait(sembuff)

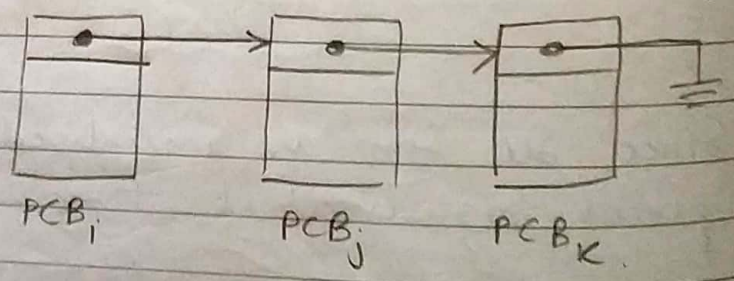
Semaphore Implementation

* Avoid Busy Waiting
* A process executing wait(),
  - finds the semaphore locked,
  - puts itself in the blocked state. instead of busy waiting

(i) state of process in PCB is WAITING only

* All processes waiting on a Semaphore in queue of that semaphore
  → Their PCBs will be chained together

wait() on S ⇒



Add link field in PCBs.

So, busy waiting's biggest problem is that the time that process is waiting could be given to some other process to carry out some meaningful task.

(I) Here, I think what they mean is wait() or signal() of CS. So, time between [lock & unlock = execution of CS] is small. the time for busy waiting is small.

(II) 3PR wait ki Naubat ∅, toh woh Running → Waiting hp by block and then CPU can schedule some other process.

= Number of Process waiting for it in the Queue.

Semaphore of Waiting Queue

```
typedif {
        int value;
        struct process* list;
} semaphore;
```

wait() adds a process to this list

signal() removes a process from this list

```
wait ( semaphore* s)
{
    (s -> value) -- ;
    if ( (s -> value) < 0 )
    {
        Add this process to s -> list ;
        block() ;
    }
}
```

→ Atomic

→ disable intern

Not the best solⁿ

→ special instrⁿ like TAS CAS

syscall() → suspends the invoking process.

```
signal (semaphore* s)
{
    (s -> value) ++ ;
    if ( (s -> value) <= 0 )
    {
        Remove process (p) from s -> list ;
        wakeup( p) ;
    }
}
```

→ Atomic

syscall() → ($P_{waiting}$ → P_{Ready}) → placed in Ready Queue

This, to be made Atomic, will also render some amount of busy waiting. BUT,
we here shift the BUSY WAITING from the User Space → Kernel Space (wait() & signal() of Critical Section)
Small codes for them;
almost never occupied → if occurs very short time.
Thus, we reduce busy waiting by fair amount of time.

Scanned by CamScanner

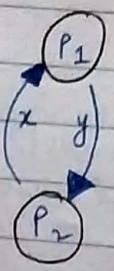
P_1	P_2	P_3
S_x $\boxed{x++}$	$\boxed{x--}$	$\boxed{x--}$
S_y $\boxed{y--}$	$\boxed{y++}$	

Semaphore $S_x = S_y = 1$

P_1	P_2	P_3
wait(S_x)	wait (S_y)	
$x++$	$y++$	
signal(S_x)	signal (S_y)	
wait(S_y)		
$y--$		
signal(S_y)		

wait (S_x) →	wait(S_y)
wait (S_y)	wait (S_x) →
⋮	⋮

P_1 after executing a wait(x) Context Switch to P_2
P_2 after executing a wait(y) Context Switch to P_1



Now; P1 (waits on y) for (signal(y)) to be generated by P2
and
P2 (waits on x) for (signal(x)) to be generated by P1

Waiting processes are waiting for a signal that can ONLY be generated from another waiting process.

Deadlock - None of the waiting processes can proceed.
Cyclic Dependency.

(*) List of processes waiting on a Semaphore

- FIFO order
  - → No process waits indefinitely long
- LIFO order
  - → Some processes may never get chance to continue

Starvation or Indefinite Blocking

Classical Process Synchronisation Problems -

⟨A⟩ Bounded-Buffer Producer - Consumer Problem

```
int n;
semaphore mutex = 1;   → Binary → Enter CS / Leave
semaphore empty = n;   } Buffer Status.
semaphore full  = 0;   }
```

space ← semaphore empty
count of elem ← semaphore full

Producer

```
while(1) {
    produce an item
    wait(empty)
    wait(mutex)
        add next item
        to buffer
    signal(mutex)
    signal(full)
}
```

Consumer

```
while(1) {
    wait(full)
    wait(mutex)
        Remove elem from
        buffer in next cons
    signal(mutex)
    signal(empty)
    Consume item in next cons
}
```

Scanned by CamScanner

(B) Readers - Writers Problem

* Data Set shared among a number concurrent processes.

- Reader : only read the data allowed
- Writer : read and write both allowed

* Requirements

- Any number of concurrent readers
- A writer MUST HAVE exclusive acc

* **First Readers - Writer problem :**

- A writer waiting for some readers to
- Some MORE readers arrive
- Give priority to the NEW READERS

Second A Yahan pe Writer ki priority hp

⊙ Writer may starve.

held by First Reader
Released by Last
All Readers

```
semaphore rw_mutex = 1;   mutual exclusion for writer
semaphore mutex = 1;      for update read_count
int read_count = 0;
```

if ok wait entry

Writer

```
while(1) {
    wait(rw_mutex)
    • Perform Writing
    signal(rw_mutex)
}
```

Reader

```
while(1) {
    wait(mutex)
    if (++read_count = 1)
        wait(rw_mutex)
    signal(mutex)
    • Perform Reading
    wait(mutex)
    if (--read_count = 0)
        signal(rw_mutex)
    signal(mutex)
}
```

if there is a reader, writer shall wait for it

if there is no read writer can now operate

mutex → Mutual Exclusion when Readers update the "read_count"

rw_mutex → Mutual Exclusion for writers.
Only one writer (and no readers) at that time

(*) After a 'Writer' finishes, if the (read_count == 0);
its upto the scheduler whether to
- Schedule a new writer thread
- Schedule a new reader thread.

no violate any rule since these threads are in WAITING STATE not waiting on a SEMAPHORE

Reader-Writer Locks

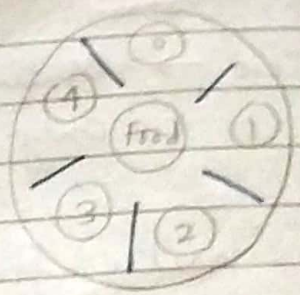
Dining Philosopher's Problem

Concurrency - Control

Philosophers

→ Think

→ Eat



Thinking : leave the table

Eat : ·) Occupy an empty chair

·) Pick up two chopsticks (left & right) one by one.

·) Start Eating

Shared Data

Semaphore Chopstick[5];

Int 1 to 1

Philosopher i

```
while(1)
{
    wait( chopstick [i] );
    wait( chopstick[(i+1)%5] );

  >EAT

    signal ( chopstick [i] );
    signal (chopstick [(i+1)%5]);

  >THINK
}.
```

PROBLEM

If all the philosophers arrive at the table, and pick up their LEFT CHOPSTICKS

⟹ Deadlock Situation

Solutions

(a) Pick up both chopsticks only if available, else WAIT()

(b) Atmost 4 philosophers can sit at a time.

bad sol' unequal weights for diff threads CS560 dphil

(c) ODD Philosophers (1, 3) => Left, Right...
EVEN Philosophers (0, 2, 4) => Right, Left..

— deadlock free problem; still no guarantee of NO STARVE check back pages

Semaphore Libraries.

(a) POSIX unnamed semaphores.

```
#include <semaphore.h>
sem_t sem;
sem_init(&sem, 0, 1)
sem_wait(&sem)
sem_post(&sem)
sem_destroy(&sem)
```

Initial Value

Shared by the threads belonging to this process.

(b) P_Thread Mutex Lock

```
pthread_mutex_init( . . . . )
```

- lock
- unlock
- destroy

(c) General Semaphores. # include <sem.h>

semget(...) → create a Semaphore Array

semop(...) → used to implement wait() & signal() on entire Array as an ATOMIC operations

semctl(...) → destroy

Scanned by CamScanner

DEADLOCK :

* A set of processes are waiting for each other INDEFINITELY.

* System Model

  - N processes, M resource types
    - anything process may need during execution
    - can have several instances
      - SEVERAL PRINTERS
      - SEV. Memory Blocks

  - A process requests the resource(s) as →

    (a) Request resource (system call(), may wait())

    (b) Uses Resource

    (c) Releases Resource (system call())

    flow.

~ Some process requests a resource which is being held by a process waiting for another resource

cause of deadlock

Scanned by CamScanner

Necessary Conditions for Deadlock.

** A deadlock will occur if all the following two conditions are Simultaneously True.

(a). Non - Sharable Resources :- A process needs exclusive access to a resource,

(b) Hold - and - Wait :- A process continues to hold some of the resources while WAITING() for some other.

(c) No - PreEmption :- Cannot take away a resource from a process and give to some other unless voluntary release.

(d) Circular Wait :- A chain of Hold-Wait conditions exist in the system

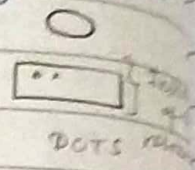
Scanned by CamScanner

Resource State Modeling

* Resource Request - Allocation Graph

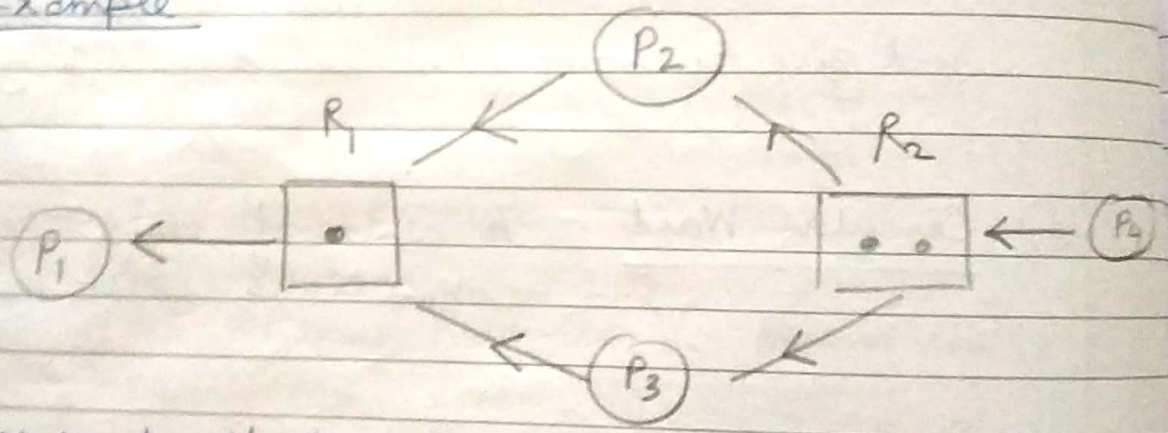
$G = \{ V, E \}$

- Bipartite Graph ; two types of Vertices
  (i) Process Nodes $P = \{p_1, p_2 \dots\}$
  (ii) Resource Nodes $R = \{R_1, R_2 \dots\}$



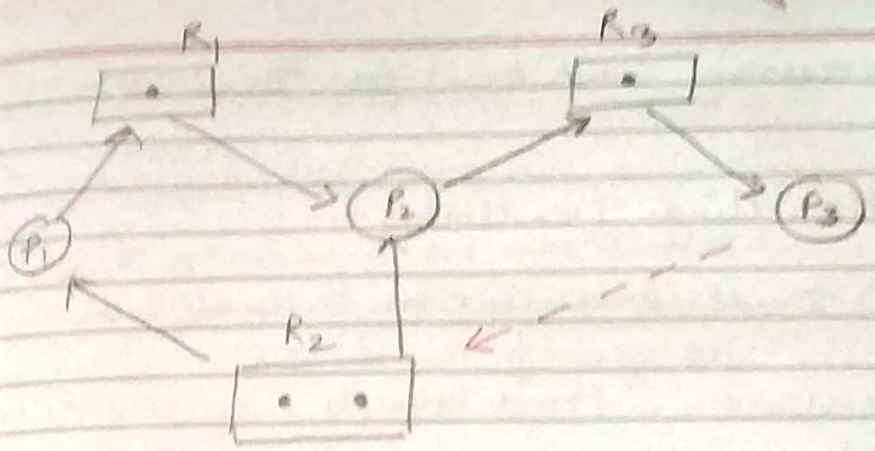
- two types of Edges as well.
  (i) Request Edge $P_i \rightarrow R_j$
  (ii) Allocation Edge $R_j \rightarrow P_k$

Example



Not deadlock situa^n.
Since P_1 can finish.

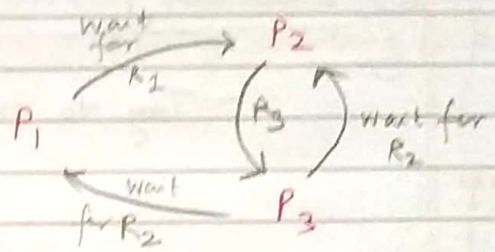
⟹ When a resource gets assigned, a Request Edge changes to Allocated Edge
⇝ Graph is continually changing



(i) Not deadlock

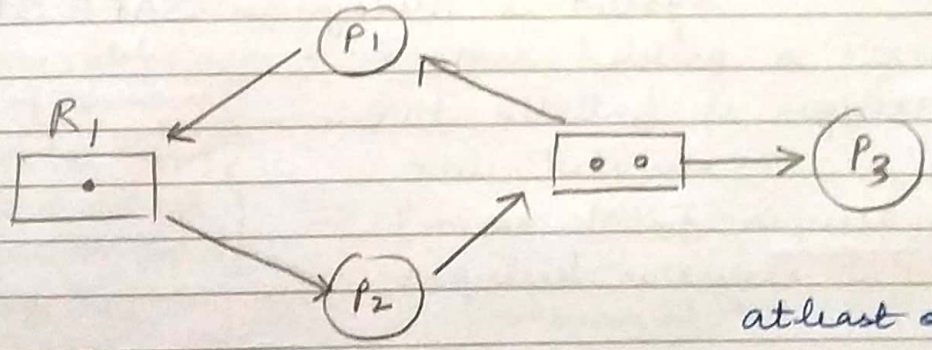
$P_3 \rightarrow P_2 \rightarrow P_1$

(ii) Deadlock



* Cycle is NOT ENOUGH

Cycle + All Resource Node involved in Cycle have one instance each



atleast one node with indegree > 0 and outdegree = 0

Scanned by CamScanner

Necessary Conditions for Deadlock - Recap

Handling Deadlocks -

Unix Windows follow these

(a) Deadlock Detection & Resolution

↓ Manual Detection

↓ Forcibly kill one or more processes

High Overheads

(b) Deadlock Prevention — take care that it never happens

- Try to make one of the four conditions fals

(c) Deadlock Avoidance — deadlock may happen

- ENSURE that (SAFE / UNSAFE state) system is always in SAFE state

Scanned by CamScanner

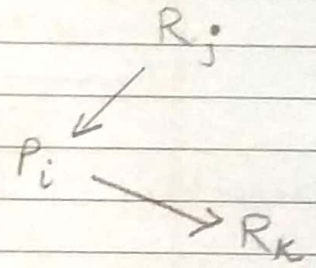
Deadlock Prevention -

(a) Non-Sharable Resources

* All resources are not sharable
* Some sharable resources are -
  CPU time, Memory Alloc,
* Keyboards, Printers, Card Readers
  these devices can NOT be shared prse
  → use SPOOLing : create virtual sharable devices
  Virtual Printer : actually printing on disk

logically make these resources sharreable so that deadlock may be resolved.

(b) Hold - and - wait



* Process

Rules

- Process holding a Resource is not alloted to request for new Resource.

=> Process MUST request all its required resources together whenever it does.

- poor utilis^n of resources
- possible starve too if all res are never avail together

(c) No PreEmption

- Cannot take Resource away from a process FORCIBLY.

* Not possible in General.
  PreEmption Might not always be possible.

if a request is made for a resource that can not be allocated to the process;
IT SHALL LOSE all of its HELD RESOURCES
Restart only when it can regain all its resources.

Scanned by CamScanner

(d) Circular Wait

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

Introduce Some Rules -

- Hierarchical Resource Allocation
  - Every Resource type is assigned a UNIQUE NUMBER

* When a running process requests for resources — it must do so in ascending order of Resource Numbers

Always wants on a Higher Resource
— POOR UTILISATION of RESOURCES
a Resource with Smaller Number might be needed later in the process' lifetime.
So if we are initially waiting for that resource to be allocated, which we do not even need ATM; pretty senseless allocation policy here

Deadlock Avoidance

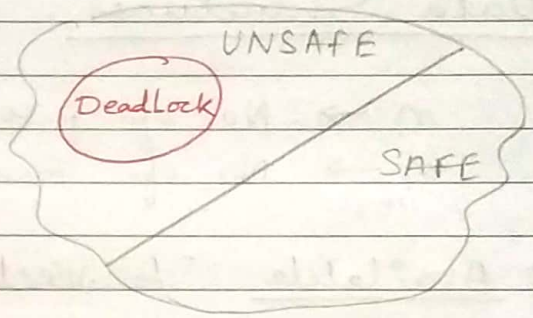
Modern Day OS → not feasible

* Every Running Process MUST declare its resource requirements in ADVANCE.

Used in DBMS Transaction.

Represent State as VENN DIAGRAM

⇝ Constantly Monitor the State



Ex: A Resource R_i with 12 instances

	Maximum Need	Current Allocation
P_0	10	5
P_1	5	2
P_2	9	2
		9.

Safe State - A state is said to be safe if the system can ALLOCATE resources to each process IN SOME ORDER. So that there is NO DEADLOCK.

Safe Sequence

For the above Example:

Safe Sequence → P_1, P_0, P_2

P_1, P_2, P_0

Other sequences MAY LEAD to deadlock, since they are UNSAFE.

We say MAY, because it may release some resources

BANKER'S ALGORITHM.

- Checks the safety of a state
- Checks if some resoume(s) is allocated to a process, can the state become unsafe ??

=> If so, process is <u>not</u> allocated the resoumes.

Data Structures

$n \rightarrow$ No. of processes
$m \rightarrow$ No. of Resource Types

(a) <u>Available</u> : A vector of length 'm'

No. of instances of R_i that are available

(b) <u>Max</u> : An $n \times m$ Matrix, that records maximum resource req of each proc

$Max[i][j] \Rightarrow P_i$ max request a maximum of so many instances of R_j

$Max_i \Rightarrow$ Vector of ith row of Matrix

Scanned by CamScanner

(c) <u>Allocation</u> - An $n \times m$ matrix

Allocation[i][j] $\Rightarrow$ No. of instances of R_j currently allocate to P_i

$Allocation_i \rightarrow$ vector of ith row of Matrix.

(d) <u>Need</u> - An $n \times m$ matrix

Need = Max - Allocation

Scanned by CamScanner

Banker's Algorithm

[A] Safety Check Algorithm

Step 1: Initialise Work = Available
Finish[i] = false $\forall i = 1$

Step 2: Find an index i such that
Finish[i] = false and
$Need_i \leq Work$
If no such i exists, go to Step

Step 3: Work = Work + $Allocation_i$
Finish[i] = true
go to Step 2

Step 4: If Finish[i] = true $\forall i$
then the State is Safe

Scanned by CamScanner

[B] Resource Request

$Request_i$ => resource request for P_i

Step 1 : if $Request_i \leq Need_i$, goto Step2
else Raise Error Condition

Step 2 : if $Request_i \leq Available$, goto Step3
else P_i must wait

Step 3 : Tentatively Allocate Resources :

$Available = Available - Request_i$
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i - Request_i$

If Resulting State is Safe
P_i is Allocated Resources
Else
Previous State Restored
P_i made to wait

Scanned by CamScanner

Recovery from Deadlock:

* Practical OS do not handle deadlocks

Try & Error Methods

(a) KILL one or more processes involved in Deadlock.

(b) Pre-Empt some of the resources held by the deadlocked process, and give to others → attempt to resolve deadlock

- **Process Termination**
  - abort all processes
  - abort one - check - continue

* very high overhead, run Deadlock Detection each kill to ensure!

Factors to keep in mind while choosing abort:

(i) Priority (ii) Extent (%) of completion

(iii) Resources held & needed (iv) How many more after this killed

Decision is Policy-Based → minimise cost

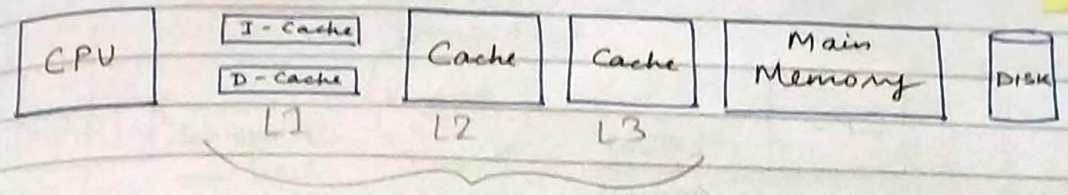
- **Resource Pre-Emption**
  - select victim; best choice → minimum cost
  - Roll back; to safe state for restart process later
    Generally TOTAL ROLLBACK
  - tackle Starvation keep picking same victim process
    → COST ↑ add no. of roll backs.

Variable Size Partition

Load
Load
— Pro
Ins

Scanned by CamScanner

MEMORY MANAGEMENT



Not visible to OS
- managed by Hardware / COA
-> just a Speed-Up
- CPU feels Main Memory Access happening

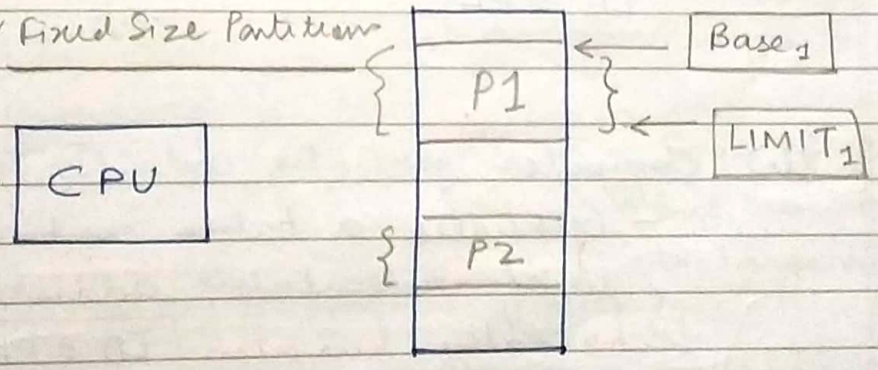
When we talk about OS → CPU ⟷ Main Memory (interaction)

* Program occupying contiguous locations in memory.

CONTIGUOUS ALLOCATION

Variable Size Partitions

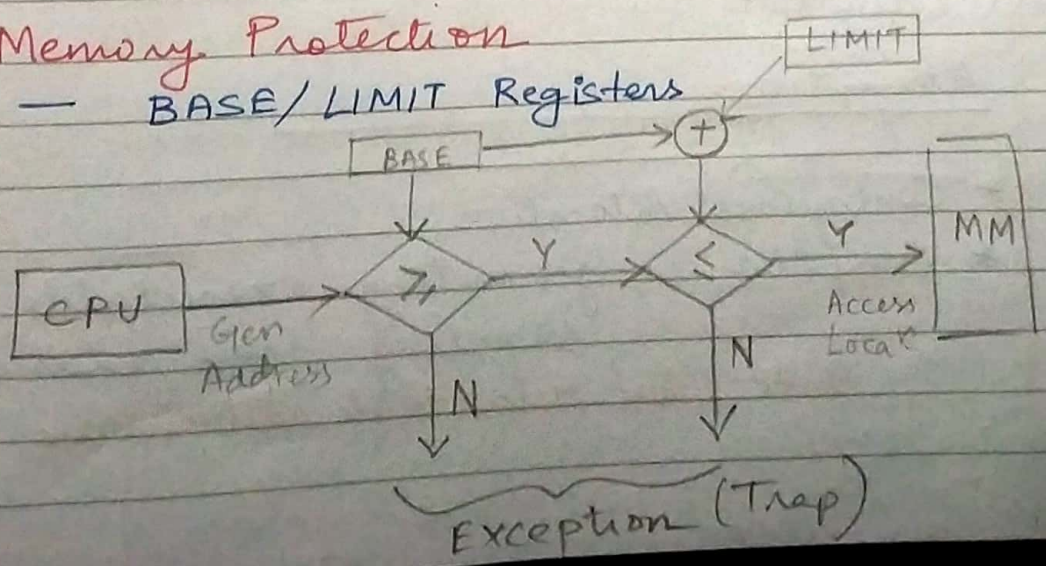
Fixed Size Partitions



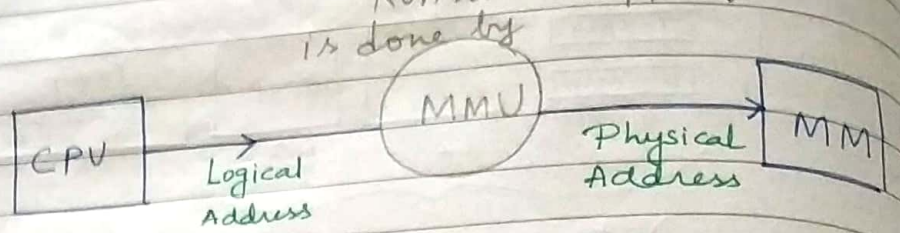
Multiple Ongoing Processes

* Memory Protection
  - BASE / LIMIT Registers

Load Base
Load Limit
- Privileged Instr.



Address Binding :



Address gen
by CPU
for Sec Memory
or wrt prog code
LA → PA

Address as
viewed by MM
Main Memory की
Address

* Three Ways of Address Binding

(a) Compile Time → old technique
(b) Load Time → relatively old
(c) Execution Time → followed

instead of relocatable
address
direct ABS Address
Bind करते हो।

(a) Compiler has enough information about the starting address
$\boxed{LA \equiv PA}$
– generates absolute code

(b) Compiler (i) generates code @ Start addr = 0
– (ii) Generates a table containing list of relocatable addresses.
technically, here also $\boxed{LA \equiv PA}$
since this change occurs before exec. starts
occurs during load time

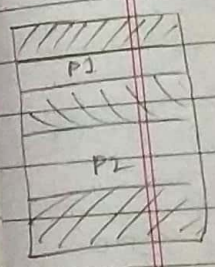
Assumptions

* Contiguous Allocation
* Compiler is generating executable code starting from address zero
thats what relocatable addresses are.

Logica

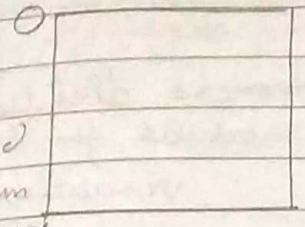
set of
all LA
generated
by a program
max

Contiguous
Memory
Allocation



Scanned by CamScanner

Logical Address Space



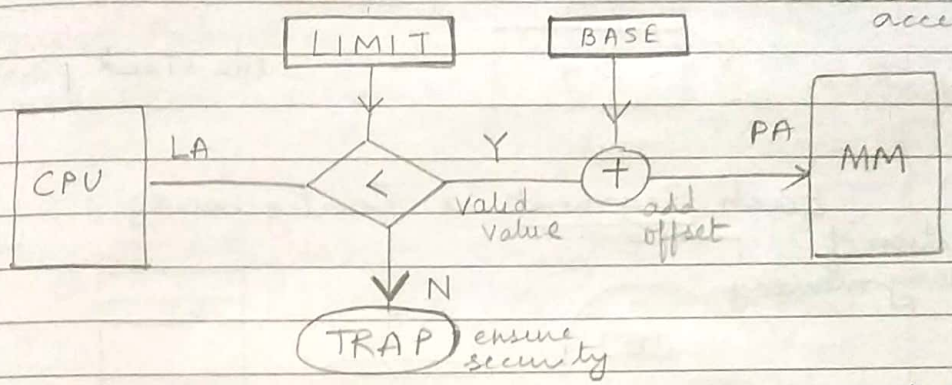
set of all LA generated by a program

for 32-bit Address Bus

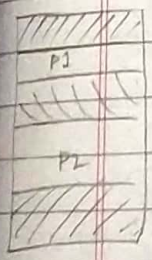
$$max = (2^{32} - 1)$$

→ ensures legality of memory address access.

Contiguous Memory Allocation



* When a new process arrives of size S bytes, a memory block of size ≥ S bytes is allocated.



Some kinds of Mem Alloc → chunks of 2^x.

Fragmentation – Wastage of Memory

(a) External fragmentation – Space of ≥ S avail but no single block is large enough

periodic compaction ⟸ can solve this problem (time take hectic).

(b) Internal fragmentation – Wastage of Memory Space inside the allocated block

fixed size memory alloc
है इसमें हो रहा है

memory blocks are of size (2^x)

→ not there in Var Sized Partition.

→ not a big deal as comp to Ext Fragment.

Segmentation -

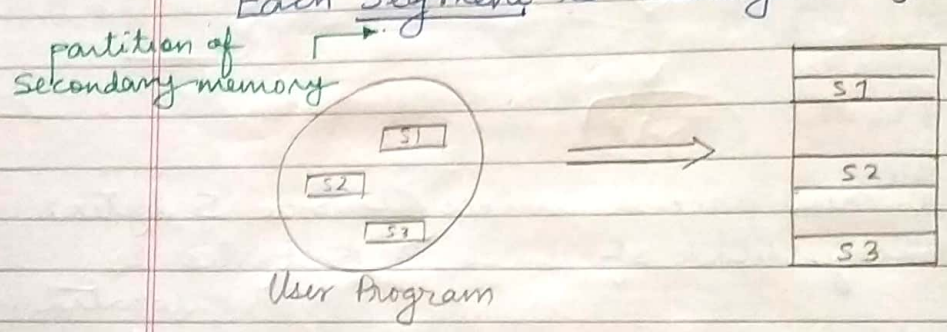
* Supports the user's view of memory
  visible to programmer

process divided into modules for better visualization

Sub Routines (R-only)	S1	
	S2	
	S3	} Stack R/W
R/W Data	S4	

• variable sized partitioning

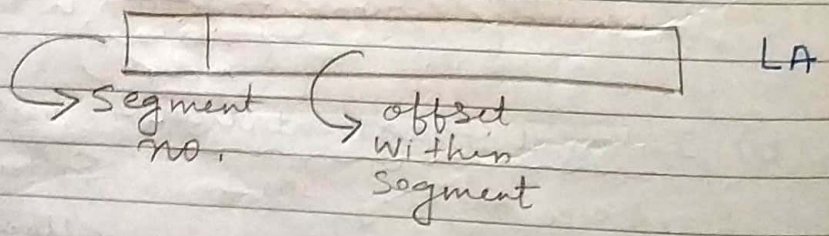
Each Segment is Contiguous;



Segmentation Architecture -

* Logical Address generated by the CPU

< segment no, offset >



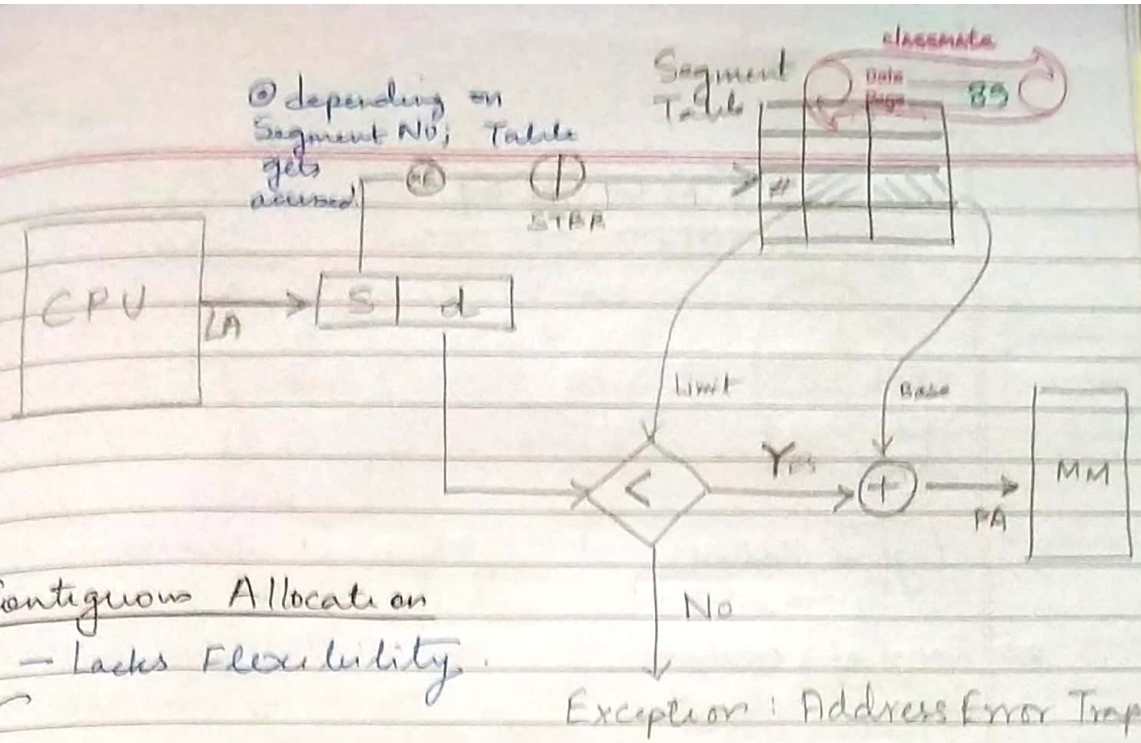
Segment Table Base Register (STBR) →

Segment Table Limit Register (STLR)

Segment Table

	BASE	LIMIT	valid	Access Permiss.n
S1	~	~	1	r
S2	~	~	1	rw
S3	~	~	1	rw
			0	
			0	

→ Stored as separate segment in the Main Memory



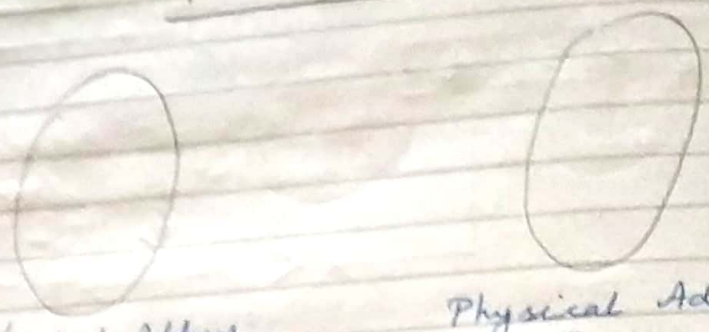
Contiguous Allocation
- Lacks Flexibility.

→ Programs/ Segment stored in NON-Contiguous Fashion

Variable Sized Non Contiguous Memory Allocation
↔ depending on the diff module sizes.

Since Variable Sized Partitioning Scheme →
(i) chances are of suffering EXTERNAL FRAGME^n
(ii) not suitable for swapping

PAGING



Logical Address Space
⇒ Range of Address Space visualised by CPU
SECONDARY MEMORY

Physical Address Space
⇒ Actual physical Main memory available

$S_{LA} > S_{PA}$ (typically)

* LA space divided into equal-size chunks called PAGES
  → 512 B, 2 KB, 16 KB, 16 MB

chunk sizes same (imp)

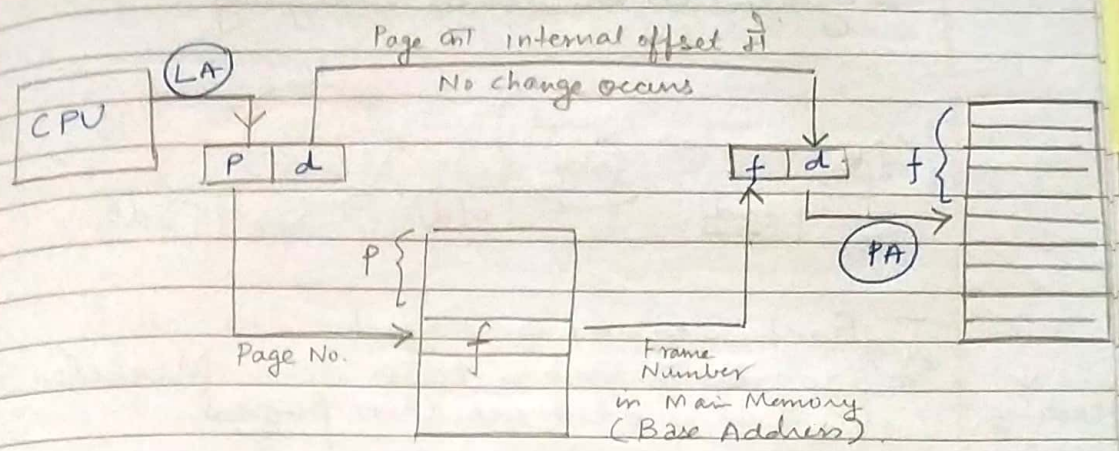
* PA space is also divided into chunks of $\boxed{\text{same size}}$ called Page frames / Blocks

one to one mapping of Page ↔ Frames

Page can Reside in any Available Block
- completely eliminates Ext Fragmentation

Every process has its own page table, of the size of the maximum number of pages it has in the

* Swapping
  makes it possible for the total PA space for all processes, to exceed the real physical memory
  → Increase the degree of Multiprogramming



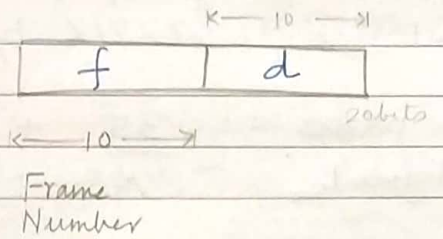
Page Table

* META-DATA * } also stored in Main Memory } Insta" Access Time doubled

PTBR accessor

MM

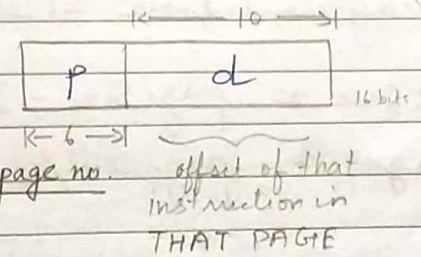
1MB



$2^{10} = 1024$ page frames

LA

16 bits



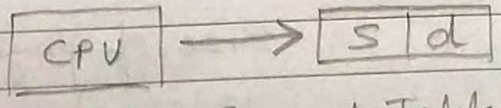
Page Size

$2^{10} = 1KB$

Max 64 pages

2^6 or $2^{SIZE(P)}$

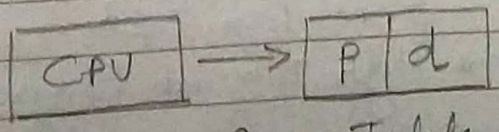
Segmentation



– Segment Table

Each segment is contiguous

Paging



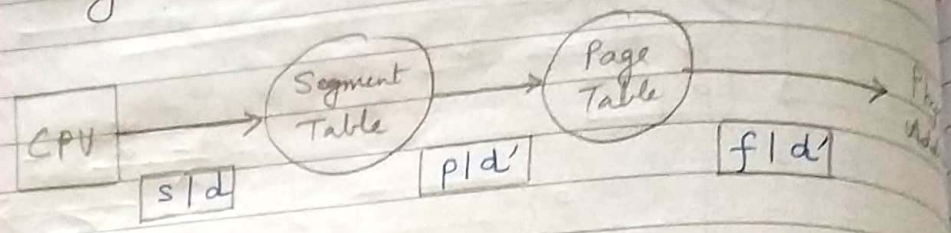
– Page Table.

Pages of the program can reside in any MM frames

Scanned by CamScanner

Segment Number | Page Number | Page offset

Segmentation with Paging :



- Each Segment is Paged.
So program is broken down into Modular Seg
which are then paged.

Each Seg has its own page table

<u>Example</u>:

* Page Size = 2 Kbytes = 2048 bytes
* Segment Size = 72766 bytes

* No of pages for a Segment = $\frac{35}{2048} + \frac{1}{1086}$

- Internal fragmentation = 2048 - 1086 = 9

  - Worst case ≈ page Size - 1
  - Average ≈ page Size / 2

* Optimum Page Size that minimises the <u>Memory Overhead</u> !

Size of page table
internal fragmentation

<u>Small P</u> • Large Page Table (more memory storage
• Less Internal Fragmentation

<u>Large P</u> • Small Page Table
• High Internal Fragmentation

Scanned by CamScanner

Calculation of Optimum Page Size P.

S : Average Size of a Segment
E : Size of each page table Entry
P : Page Size

Avg internal frag = $P/2$
Avg no. of pages per segment = $\lceil S/P \rceil$

TOTAL OVERHEAD

$$V = \alpha\underbrace{\left(\frac{P}{2}\right)}_{\text{Avg Int Frag}} + \beta\underbrace{\left(\frac{ES}{P}\right)}_{\text{Page Table Size}}$$

here we are giving them equal weights

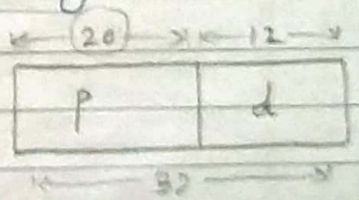
$$\frac{dV}{dP} = \alpha\frac{1}{2} - \beta\frac{ES}{P^2} = 0$$

$$P = \sqrt{2ES\cdot\frac{\beta}{\alpha}}$$

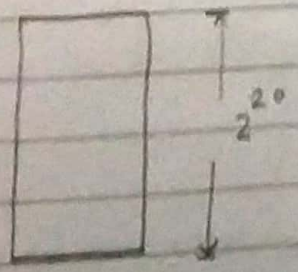
Implementation of Page Table.

* Typically stored in Memory

$P = 4KB$



$\Rightarrow 2^{20}$ entries in PT.

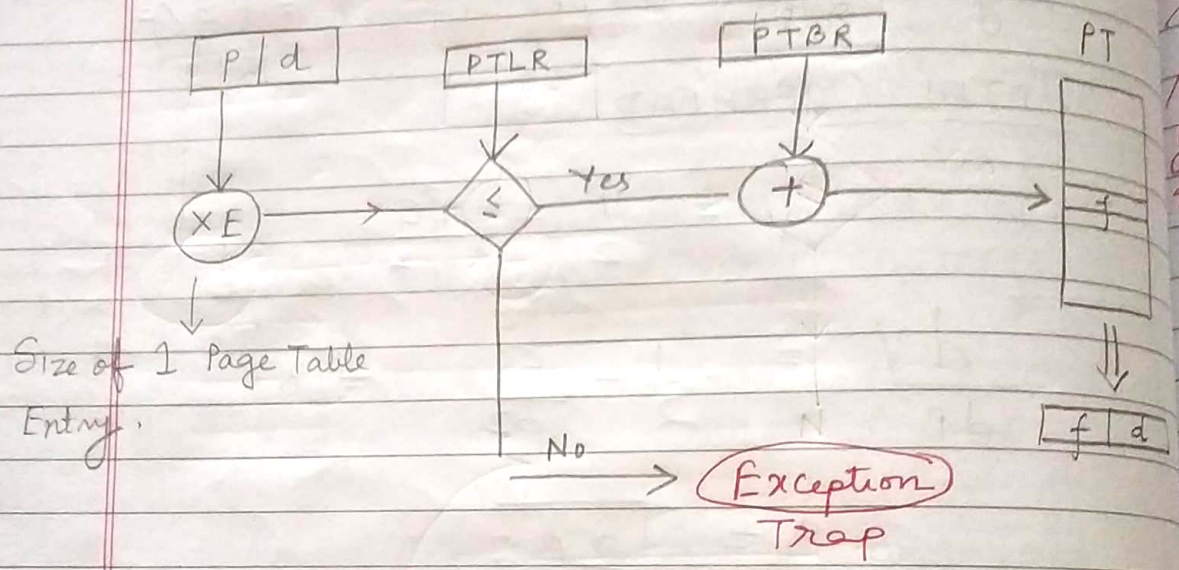


> Hardly दिखेगा full होगा

* TWO SPECIAL REGISTER :

PTBR - page table base register
PTLR - page table length register

⇒ Each process stores these values in the corresponding PCB



Two Memory Access for every instruction or data access

$\approx 2 \cdot t_{mm}$

Since Page Table is also stored in the Main Memory.

Speed is halved

Translation Lookaside Buffer (TLB)

* Is a high-speed (associative) memory where recent page access information are stored.
hardware cache

$$t_{TLB} \approx 10 - 20\% \text{ of } t_{MM}$$

* Implemented in Hardware
* unlike P. Table
* one pc.
* Flushed on Context Switch.*

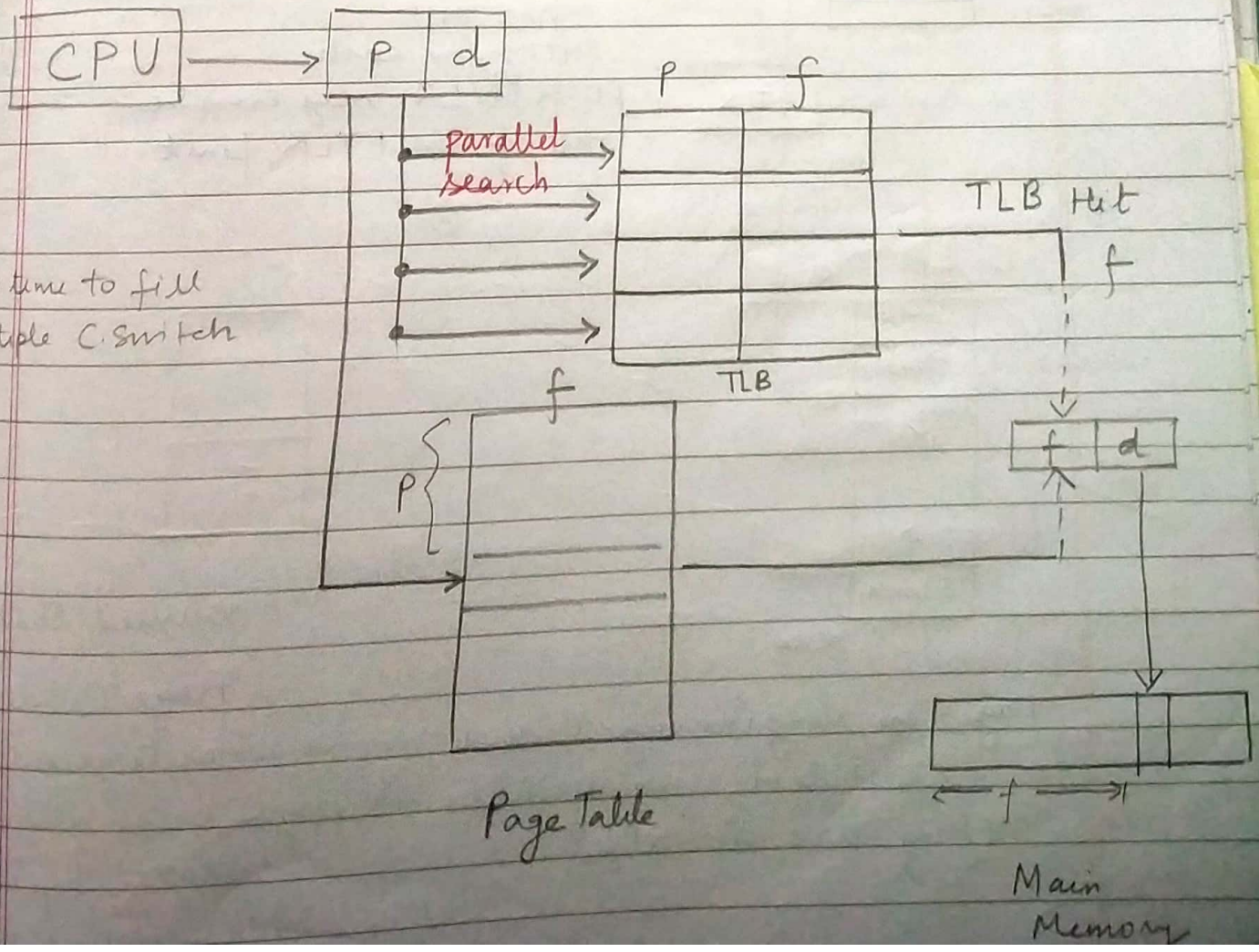
TLB

Page #	Frame #	Process_id
2	8	
4	5	

One we access a page, we store its frame number in a "TLB" so that it is referred for all subsequent queries.

Check: TLB is accessed in a Set Associative Manner
Hardware is such that all locations are searched for together/simultaneously

	PT
0	
1	
2	8
3	
4	5



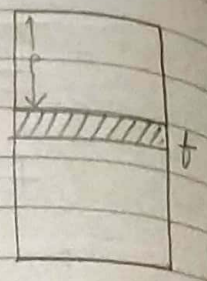
① Takes time to fill
② Multiple C. Switch

Scanned by CamScanner

Contents of the Page Table.

p -> f mapping

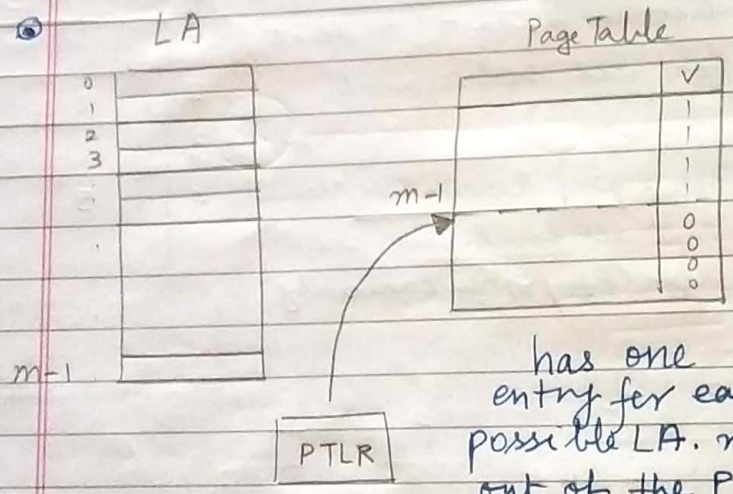
Frame No	Access Permissions	Valid/ Invalid



Page Table
=> contiguous

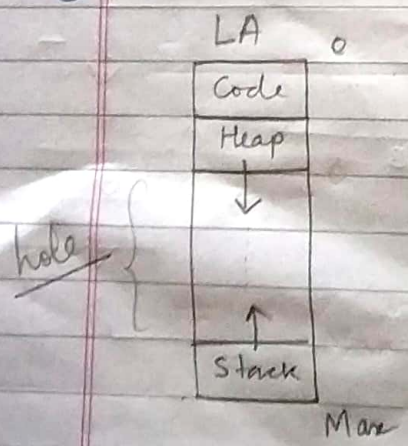
0 : Read Only
1 : Read/Write

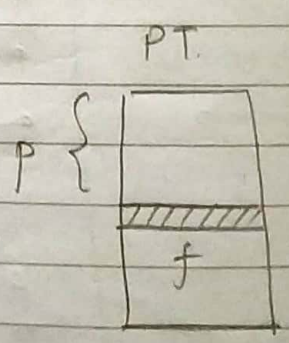
0 : Page Invalid (exi LA
1 : Page Valid (within LA sp



has one entry for each possible LA. may even be out of the PTLR limit.

Any page beyond PTLR is considered INVALID.





Pages in beginning & end are valid, in middle they are invalid.

Assumed that Page Tables contiguous, we calculate address of p directly using

Structure of a Page Table

Let (32b LA) and (Page Size = 4KB)

$32 - 12 = 20$ → (2^{20}) Page Numbers

$(2^{12}\ B) \Rightarrow 2^{12}$ locations

p	d
← 20 →	← 12 →

Size of Page Table

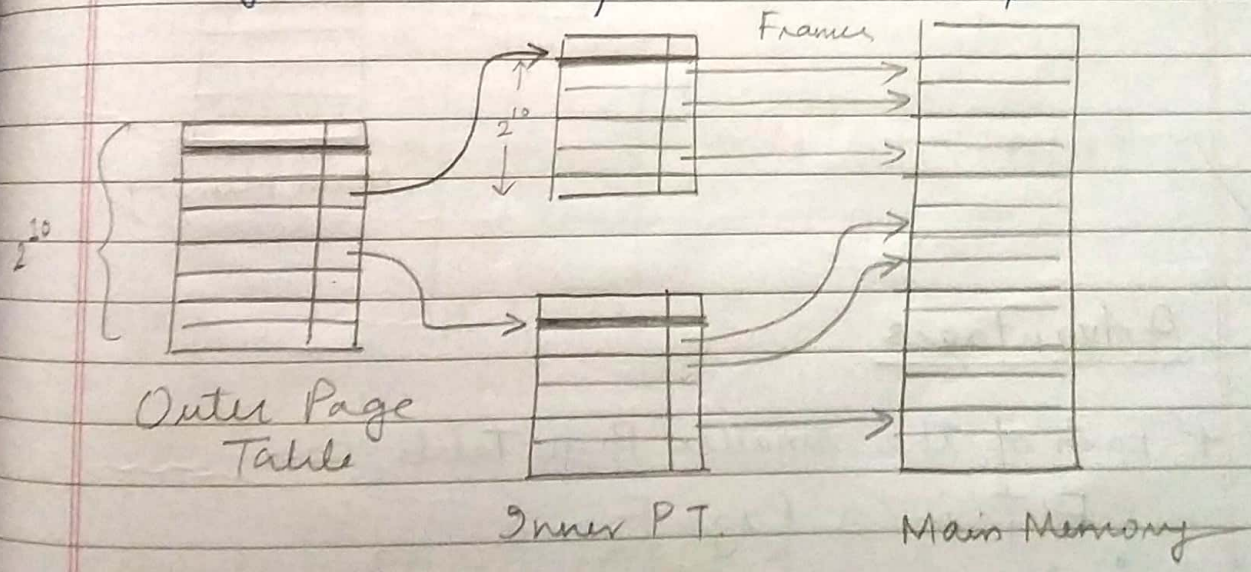
= Rows × Size per Row.

$(2^{20}) \times$

We shall need this much Contiguous Memory Space for PT.

HIERARCHICAL PAGE TABLES –

Page Tables – Split into multiple levels.



2 Level Paging

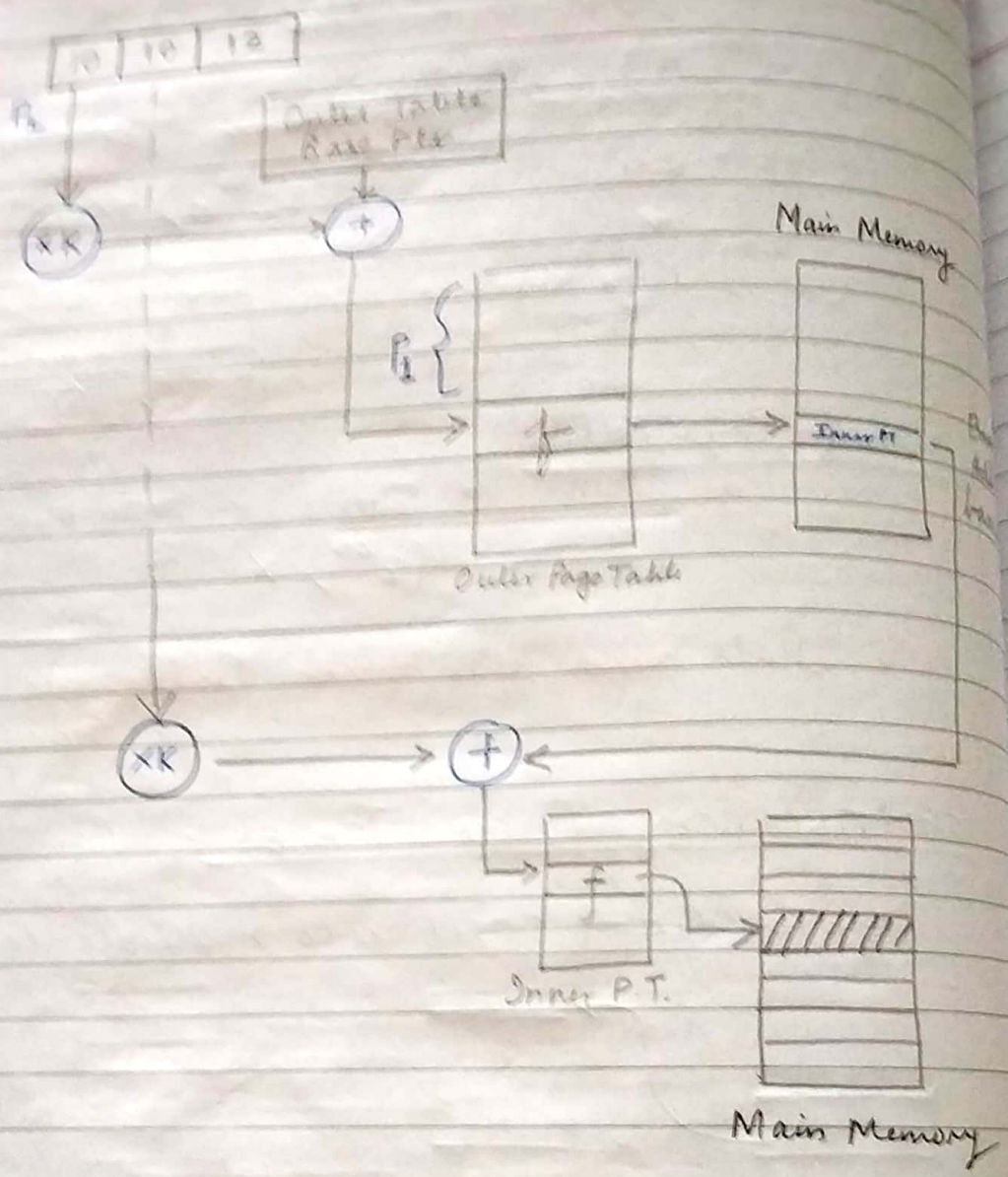
* 32 bit Logical Address
* 4KB page Size

P_1	P_2	d
← 10 →	← 10 →	← 12 →

$P_1 \rightarrow$ Outer PT

$P_2 \rightarrow$ Inner PT.

Scanned by CamScanner



Advantages

* Each of the smaller Page Table can FIT in a Page Frame

  ⇒ No Contiguous Space Needed.

* Memory Space is Saved.

  Very few inner page tables may need to actually be stored.

MULTI LEVEL PAGE TABLE

* Logical Address 64 bits

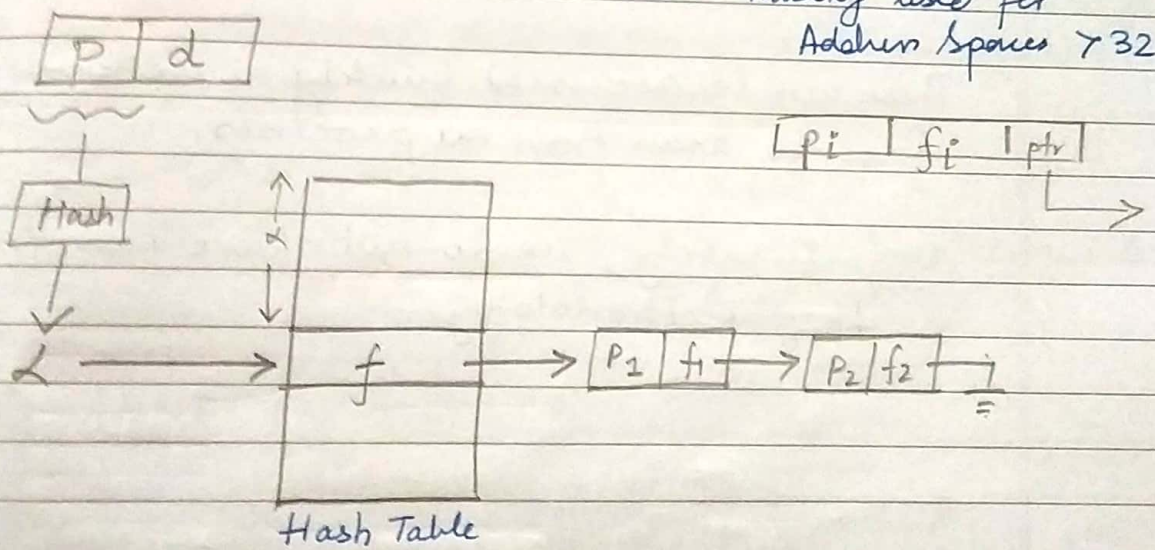
P_1	P_2	P_3	P_4	P_5	P_6	d
2	10	10	10	10	10	12

Worst Case → 7 memory Accesses

This is too much overhead for a Memory Lookup; hence these are hardly used.

HASHED PAGE TABLE

— mostly used for Address Spaces > 32b



Inverted PAGE TABLE

Page No	Process Info

* For Small Systems
* One Entry for each Main Memory Frame.
* LA generated as

P-id	Page	off

used to get corresp entry in the Page Table

Scanned by CamScanner

The thought behind this is that a lot of pages from different processes are mapped to same frame, and have that stored in their Respective Page Tables, with adequate valid/invalid bit etc.

instead, if we store which <process, page> is stored in the frame → SAVING HELLUV of Space.

Problem is SHARED MEMORY

for Shared Memory, many processes have pages pointing to same the Shared Frame.

These can be accessed quickly as we know which frame to access from all processes.

Unfortunately, we do not have this data in this methodology.

Scanned by CamScanner

VIRTUAL MEMORY

* **Conventional Paging :**

— Program's completely loaded into Main Mem before it starts execution.

— Size of Logical Address Space CANNOT exceed that of physical memory

* **Virtual Memory :**

— Complete seperation b/w logical & physical address space.

— For a program whose size exceeds size of Physical Memory

⇒ loading only a part of the program in memory (that is required)

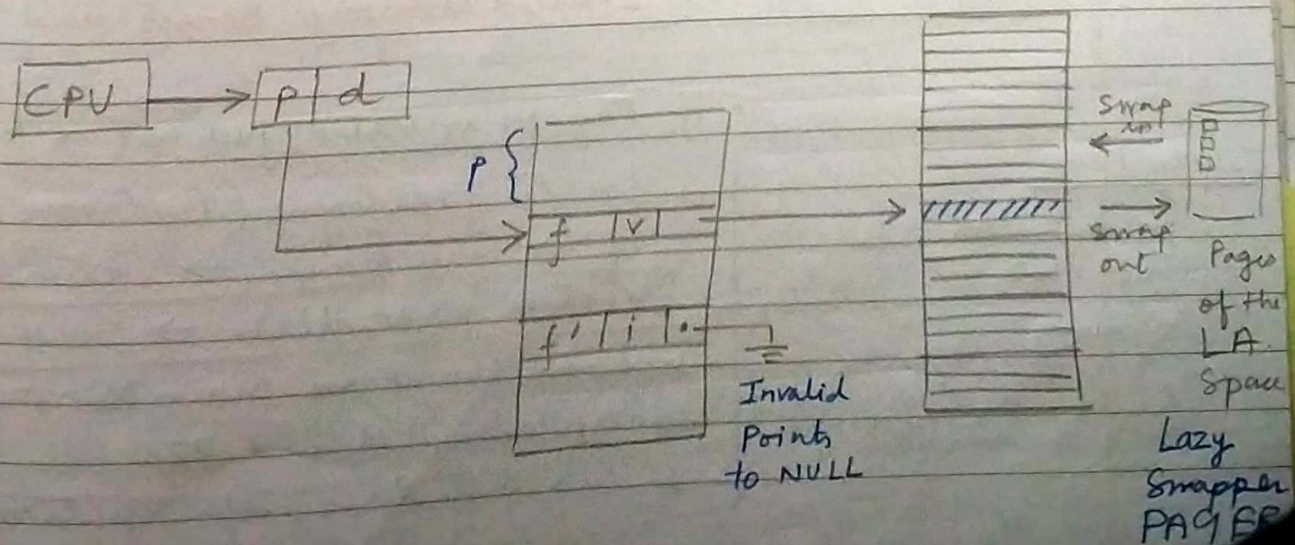
→ Better Memory Utilisation

→ More Active Programs can be Accomodated

Increases degree of Multi Programming

DEMAND PAGING -

* Page is loaded in Memory on Demand.
* Before execution starts, only first page is loaded. or start with no page in Memory.



Where in the disk are the Logical Pa
stored ?

* OLDER OS

— seperate partition in disk : SWAP
— data blocks stored in contiguous regions in swap area.
⇒ fast access to disk blocks

* MODERN OS

— load the pages corresponding to ca DIRECTLY from the file System.
— use SWAP AREA to store the dynamic data segments only.

* CPU generates < p, d >
* Search for p in the TLB
  — Found: get f and compute <f,d>
  — Else
    * Search for p in the Page Table
      — Found Valid Entry: get f → <f,
      — Invalid Frame: generate Page Fault
        • if Page No is invalid, Error Generat Invalid Memory Access and Re
        • if Page No is valid, just not in Mem
          → Locate free frame (Free Frame List
          if no free frame → victim pag
          Victim page dirty → Swap-o
          → Load Frame in Main Memory
          → Update TLB and PT.
          → Restart this instruction

Dirty Page
• page that was updated while in memory

Disk opr.

Scanned by CamScanner

Hardware Support

* For TLB and PT access.
* Restart fault inducing Instruction feature
* Mark page dirty for write operations
* Secondary Memory : Swap Space

Page Replacement

Page Fault

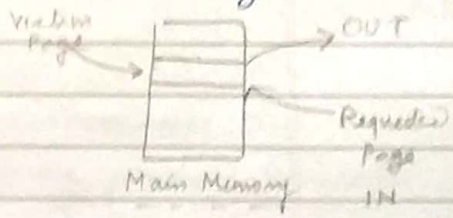
- Free Frames Available → 1 disk I/O
- Free Frame Not Available → Page Replacement → 2 disk I/O

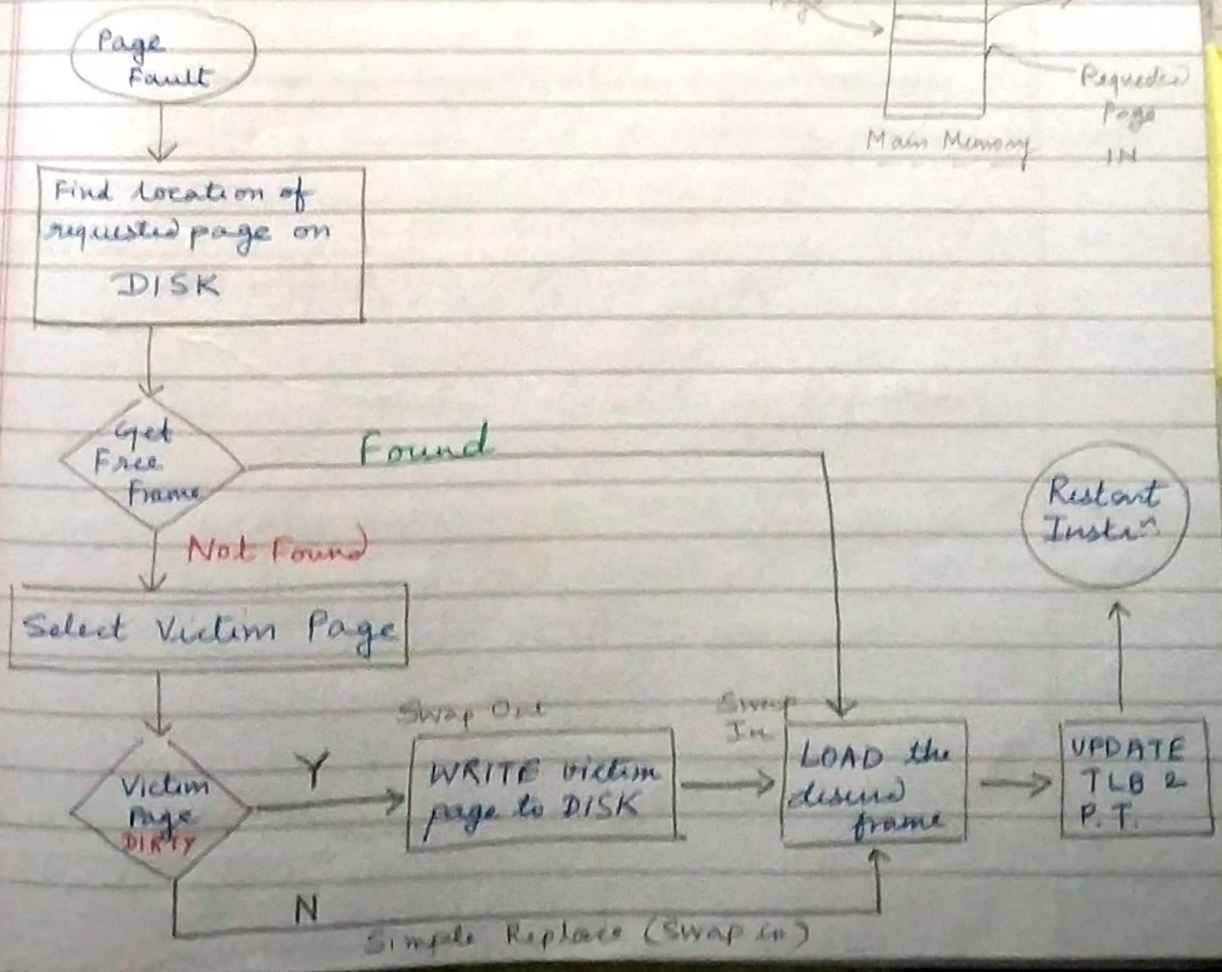
Page Table

	Dirty bit	Other
	0 clean 1 dirty	

Used to select VICTIM Page

* When a new page is loaded → clear
* when a write operation occurs → dirty





Let's Say

$P_f \rightarrow$ probability of Page Fault

$\delta \rightarrow$ probability that victim page is

Ignoring TLB; Single Level Page Table

$$EAT = (1-P_f)(2 \cdot t_{mm}) +$$

→ Once to Access Page Table
Once to Access the Frame

$$P_f \left[2t_{mm} + (1-\delta)t_{I/O} + (\delta)2t_{I/O} \right.$$

• Read page into a Frame
Only Snap In

• Also write dirty page to disk
+ Snap In

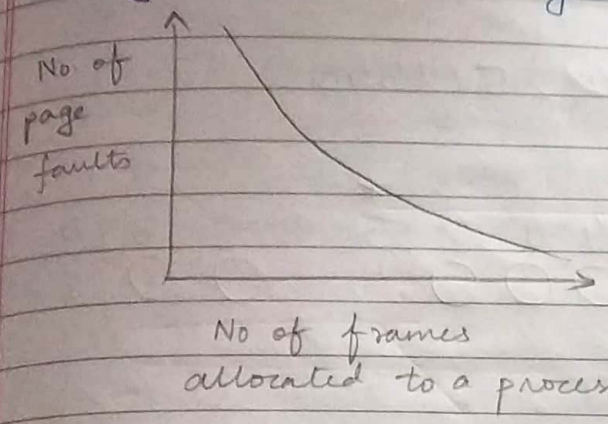
$mm \rightarrow$ Extend for

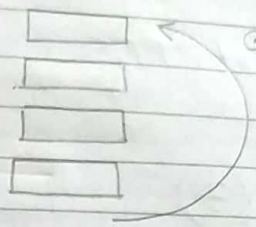
(i) TLB

(ii) 2-level page table

VICTIM PAGE SELECTION

- Page Replacement Algorithm





⊙ If alloc ≤ 5 frames Every time new Page Faults

Program Locaility of Reference — Set of pages that are frequently accessed.

Page Reference String –

Sequence of Page Numbers corresponding to memory access that a running process generates in time.

$$PRS = \{ P_1, P_2, P_3 \ldots P_n \}$$

→ Property : Locality of Reference

Basically, program is more likely to refer to a Page adjascent to it, rather than an absolutely random page.

Scanned by CamScanner

Page Replacement Algorithms

(a) FIFO Page Replacement

Let PRS : 1 2 3 4 1 2 5 1 2 3 4 5

							*					
f(3) :	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	2	3	4	1	1	1	2	5	5
		2	2	3	4	1	2	2	2	5	3	3
			3	4	1	2	5	5	5	3	4	4
	+	+	+	+	+	+	+			+	+	

9 Page Faults

							*					
f(4):	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	1	1	1	2	3	4	5	1	2
		2	2	2	2	2	3	4	5	1	2	3
			3	3	3	3	4	5	1	2	3	4
				4	4	4	5	1	2	3	4	5
	+	+	+	+			+	+	+	+	+	+

10 Page Fault.

Belady's Anamoly

As the number of frames increase, no. of page faults also increases

not all instances will suffer some May suffer.

Stack Algorithms

Set of pages in memory for n frames is ALWAYS a subset of set of pages which will be in memory for $n+1$ frames.

$$F_{n,i} \subseteq F_{n+1,i}$$

Stack Algorithms do NOT exhibit BELADY'S ANAMOLY

FIFO is not Stack → FIFO exhibits BA

(b) Optimal Page Replacement

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

⇒ OFFLINE ; can't be implemented ONLINE
⇒ Look into the future.

Minimum Page Faults : Gives Lower Bound to test other Algorithm.

Theoretical only

f(3) :

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	3	4	4
	2	2	2	2	2	2	2	2	2	2	2
		3	4	4	4	5	5	5	5	5	5
+	+	+	+		+			+	+		

7 Page Faults (Minimum for this PRS-f)

Analytic

P → Page Ref String
For optimal page replacement, number of page faults for P and Rev(P) is SAME.

(c) Least Recently Used (LRU)

- We use the recent past to predict the near future.

⇒ LOCALITY REFERENCE.

To predict which choice of page is optimal, we look into past to see which page is used least and predict it shall be used farthest in the future

	1	2	3	4	1	2	5	1	2	3	4	5
f(3):	1	1	1	4	4	4	5	5	5	3	3	3
		2	2	2	1	1	1	1	1	1	4	4
			3	3	3	2	2	2	2	2	2	5
	+	+	+	+	+	+	+			+	+	+

10 Page Fault

LRU is a Stack Algorithm, hence never exhibits Belady's Anamoly.

$P_{n,i}$ → n most Recent

$P_{n+1,i}$ → n+1 most recent

In practice,

LRU or some approximation is used as

Implementing LRU :

(i) Counter-Based Implementation

what we implemented in LAB ASSN

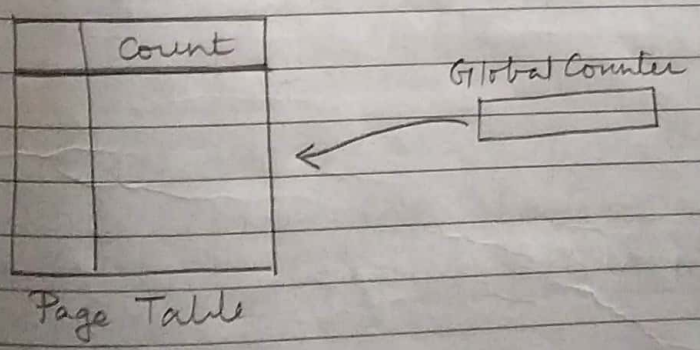
* OS maintains logical clock
  - → initialised to 0
  - → incremented at every memory reference

* Include a "counter" field in PageTable/TLB
  - Whenever a page P_i is accessed, the logical clock value is copied to the page table entry for P_i

* For selecting the victim page
  - Select the page with smallest count value

Software Solution is NOT possible, since 'count' needs to be incremented at each access along with necessary updates.

Hardware Required – quite a bit.

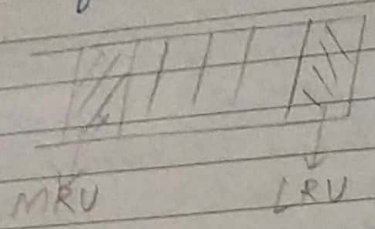


Clock Overflow also needs to be rectified.

(ii) Stack - Based.

what we did in ASSN for TLB.

* A stack containing recent page references are maintained.



Page Ref String: 4 7 0 7 1 0 1 2 1 2 7

f(3)

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

Tail always points at the LRU page.

⊙ Doubly Linked List

→ max 6 link updates

⊙ Hardware Stack Required.

LRU Approximation Algorithms

(i) Using Reference Bits :

- Associate a Reference Bit with every page in Memory
  - ⟹ Seperate Hardware provision Reqd in the PT

- When a page is first brought into memory, its ref-bit is set to 0
- → Every time a Page is Referenced, the corresponding ref-bit is set to 1
- → Periodically (≈ 100 msec) all the bits are reset to 0.

- Victim Page : Any Page where ref-bit is 0.

Maintain Pointer -> Search List Cyclically.
prevent continuous replacement of first few pages. (if we always search from Start)

if we always start from the start, we tend to replace the starting pages more often
→ Like giving higher priority to bottom pages

not STACK

(ii) Second Chance Replacement

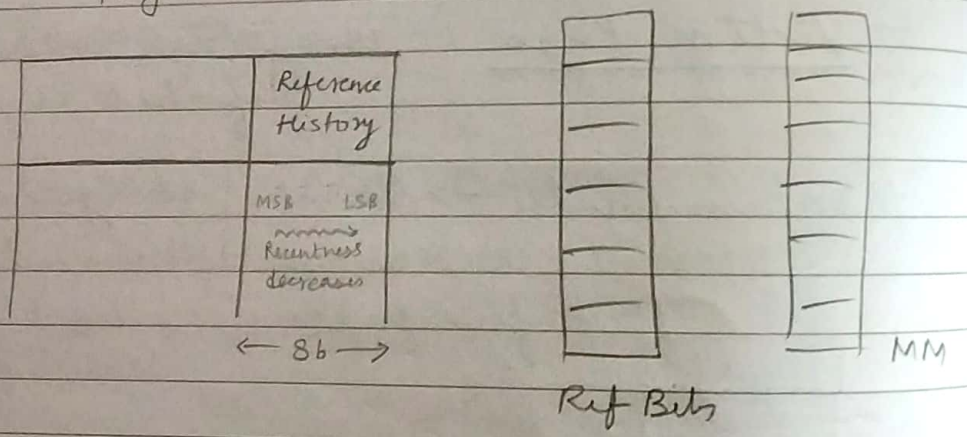
— Modification of Previous Approach

Instead of Periodic Resetting:
During Victim Selection –
while Searching the ref-bits,
- if a bit = 1; bit is toggled to 0
  move on to next entry

Cyclic Search here as well

(iii) Multiple Bits per Page

Page Table



- When a page is 1st brought, the Page Ref History is set to 0's.
- Ref Bits are updated as usual.
- Periodically (≈ 50 msec) there will be a timer interrupt and the OS will do
  a) Shift Ref History bits Right by
  b) Copy Ref Bits into MSBs.

Victim Selection

Identify the page with **Smallest** Ref History Value. (Treat it as integer)

Manageable Overheads

* While Replacing → take into Consideration DIRTY or CLEAN page
  - data loss

Enhanced Reference Bit Scheme

* Two Tuple : < ref-bit, modify bit >

< 0, 0 > :	Recently Used	Modified
< 0, 1 > :	Recently Used	Modified
< 1, 0 > :	Recently Used	Modified
< 1, 1 > :	Recently Used	Modified.

Replace a Page that belongs to the 1st Non-empty class among these categories (check in this order)
=> Multiple Scans through the list

Scanned by CamScanner

ALLOCATION of Frames to a PROCESS.

How many frames to allocate?

MINIMUM → depends on the ISA

| → how many memory accesses required per instruction

Instructions are WORD-ALIGNED

- instruction can start from an Ad multiple of 4.

MIPS : 32 bits

Memory Interleaving ←

Scanned by CamScanner

Allocation of Frames

Processes $P_1, P_2 \ldots P_n$
Total No. of Frames = S.

Minimum No. of Pages to P_i : ISA
How many may be needed to execute atleast one instruction.

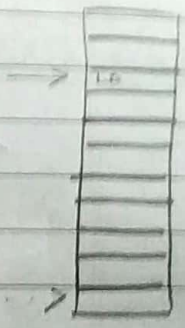
* In the worst case, how many pages can be refered by an instruction

MIPS:

$$\boxed{LD\ R1,\ 10(R5)}$$

Fetch

M[R5+10]



MIPS - Min 2 Pages.

CISC

Memory to Memory Move

$$\boxed{MOVE\ M1,\ M2}$$

Multi Word Instruction

Scanned by CamScanner

Global Allocation

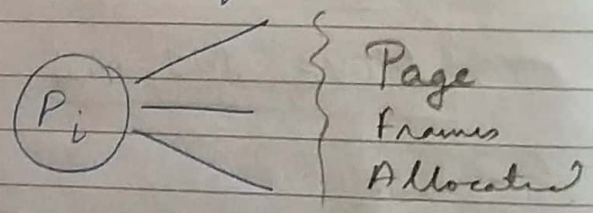
- When a process starts,
  some minimum no. of pages allocated
  → process size
  → priority

As more pages are requested, (page-faults)
→ additional pages are assigned from the global pool.

* → Process can take away pages from another process

Local Allocation

- When a process P_i generates a page fault
* → the victim page to be replaced will be one of the frames allocated to P_i



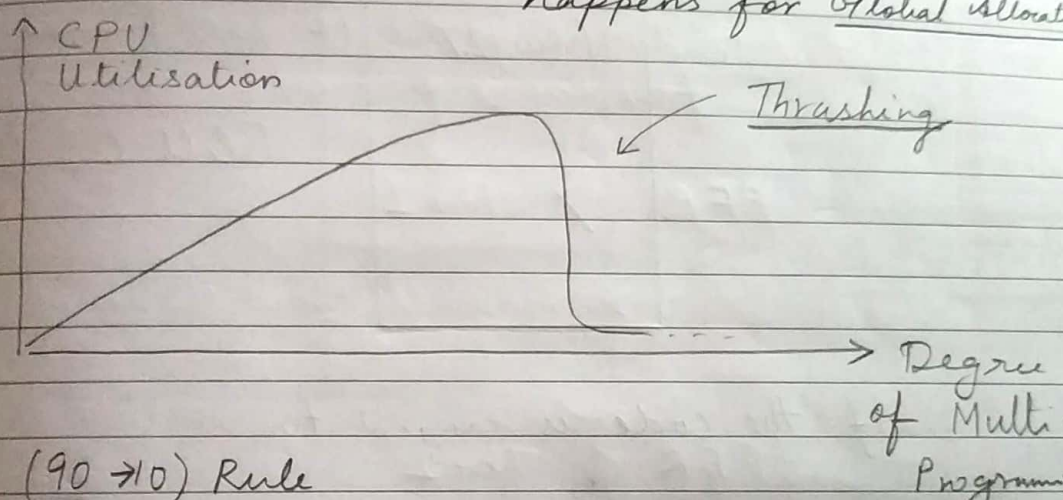
In Local Allocation, a process cannot effect the behaviour of other process
In Global Allocation, a process requesting too many pages can effect other processes by causing them Page Faults.
But, Memory Utilisation is better,

As memory utilisation is of highest priority, most modern OS use GLOBAL ALLOCATION

THRASHING

* Too many page faults.
* A process requests a page P_i
  - → P_i is loaded, replacing another P_j
  - → P_j required again, replace P_i

→ Too few pages to process.
- happens for Global Allocation



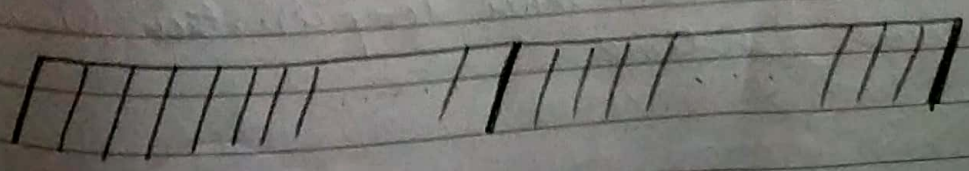
(90 → 10) Rule
90% of execution time spent
10% of code.

Example

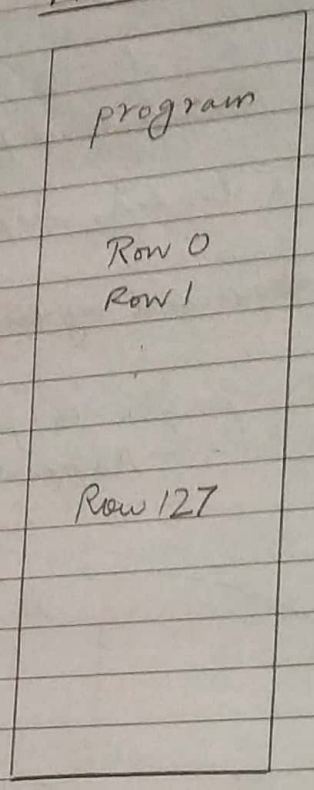
Page Size = 512 bytes
Int Var = 4 bytes

```
for i = 1 to 128
    for j = 1 to 128
        A[i][j]++
```

} Array is in ROW MAJOR

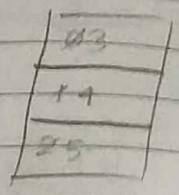


Main Memory



Page Faults = 128

if there are 3 Frames



Still 128 Page Faults

if the code is changed to

```
for j = 1 to 128
    for i = 1 to 128
        A[i][j]++
```

1 Access Column Wise –

128 × 128 Page Faults

This because to access column wise, we need a new row for each inner loop iteration, hence page fault occurs each time.

Working Set Model

— principle of locality

Δ : Working-Set Window Size

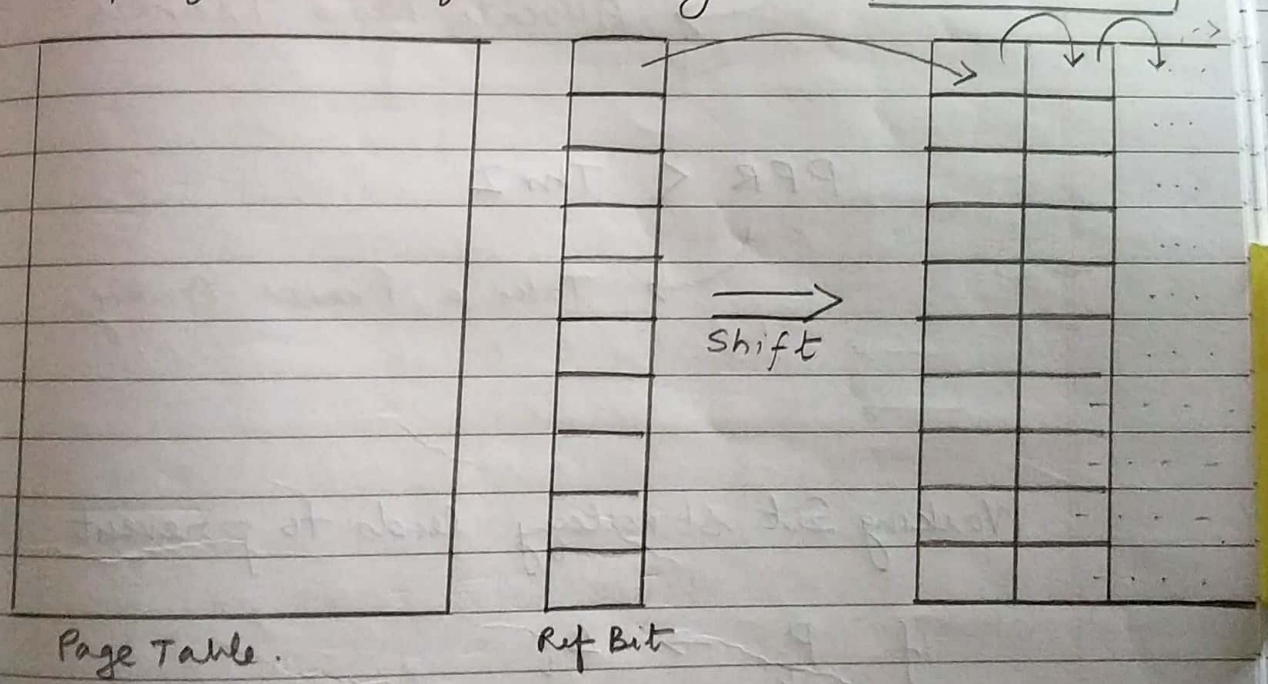
Working Set – The set of pages that are referenced by a process in the last Δ memory accesses.
→ optimal value

Page Ref String → 1 2 3 2 3 2 3 2 3 4 3 5 3 5 2 3 2 3 5 2 5 3 5 3

$\Delta = \{2, 3\}$

$\Delta = \{1, 2, 3\}$ $\Delta = \{2, 3, 5\}$

Keeping Track of Working Set $\boxed{\Delta = 15000}$

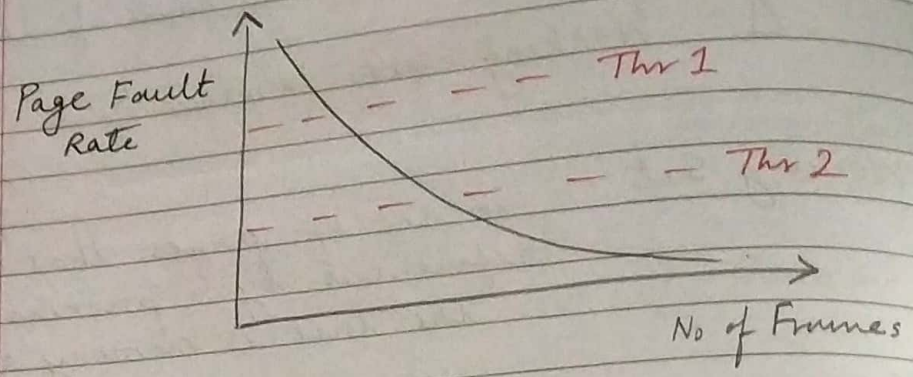


Timer Interrupt (for Shift) every 5000 mem access

Those pages which have either reference bit set or one of the History bits set, shall be considered part of the working set.

Store appropriate Process Δ length in TLB to boost throughput

Page Fault Frequency



* Monitor Page Fault rate for every process.

PFR > Thr 1

→ Allocate More Frames

PFR < Thr 2

→ Take a Frame Away.

Working Set strategy tends to prevent thrashing

if $P_i \rightarrow \Delta_i$

then if

$$\left(\sum \Delta_i\right) > m$$

then Thrashing is occuring

Suspend or SwapOut process while maintaining high degree of multiprogramming

Secondary Storage Systems

BROAD

Magnetic Disks

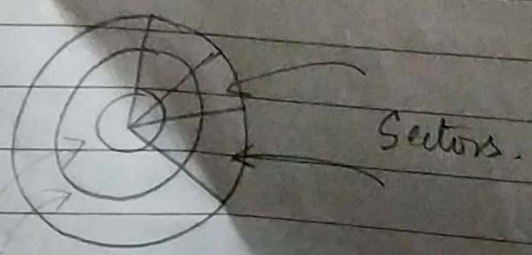
* Data Stored as tiny magnets on the surface of the medium

* Surface

  - Data Recorded in Concentric Tracks
    - → outermost track - Track 0
    - → Track Numbers Increase as we move towards the centre

* Each track is divided into sectors

Sector → minimum unit of data that can be read or written. (together)



Scanned by CamScanner

No. of bytes that a sector can store → constant.

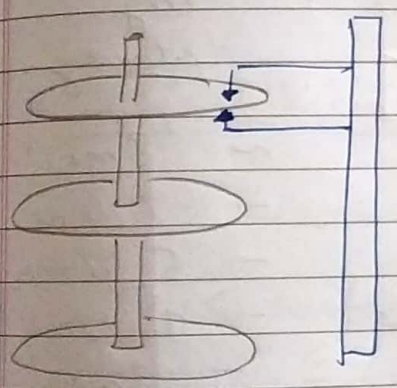
Low cost disks

* No. of sectors/ track is constant
* No of bits / unit length varies from Track to Track.

Sophisticated Disks

* Bit density of outer tracks is larger than that of inner tracks

Multi Platter Disk units



* platters
* surfaces
* tracks / surface
* sector / track
* Data capacity of sector.

Scanned by CamScanner

Disk Access Time - has three components

(a) Seek Time The time taken by the R/W head to reach the desired track

Radial Movement 3ms - 12ms

(b) Latency of Rotational Delay - The time taken for the desired sector to rotate under the R/W head.

2ms - 10ms

Max Delay - one full Rotational Delay

Avg Delay - Half the Rotational Delay

Rotation Speed (RPM)		Max Latency (time for 1 rot)		Avg Latency
3600	~	16 ms	~	8ms
4200	~	14.3 ms	—	7.14ms
5400	~	11.2 ms	—	5.56 ms
7200	~	8.34 ms	—	4.17ms
10,000	~	6 ms	—	3ms
15,000	~	4 ms	—	2ms

(c) Data Transfer Time - Time required to transfer sector data.

Example -

Sector Size = 1024 B
5000 tracks / surface
64 sectors / track
8 surface
10,000 RPM.

Total Disk Capacity = 8 × 5000 × 64 × 1 KB = 2.56 GB

Data Tran
Rate

* Fast

* Logi

OPERA

* OS

B loc

Scanned by CamScanner

Data Transfer Rate = 6 ms per Revolution

$$(64 \times 1 \times 8) = 512KB$$

Sectors ↓ Sector Size ↓ surfaces per Revolution

6 ms → 512KB

1 sec → $\frac{512 \times 10^3}{6}$

= 85 MBps

दूसरा track पे जाने को R/W head move करना होगा ना !

* Fast Rotation Speed -

→ R/W head floats on top of a layer of AIR

* Logical View of a Disk

→ Linear ordering of sectors

< surface no, track no, sector no >

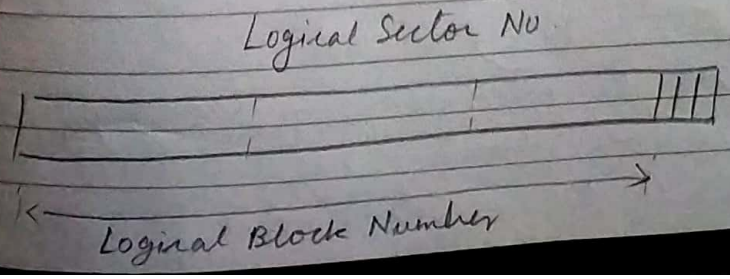
⊙ LOGICAL SECTOR NO. ⊙

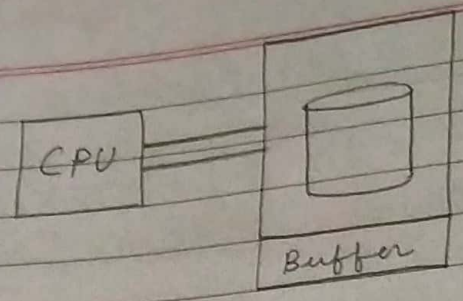
OPERATING SYSTEM VIEW :

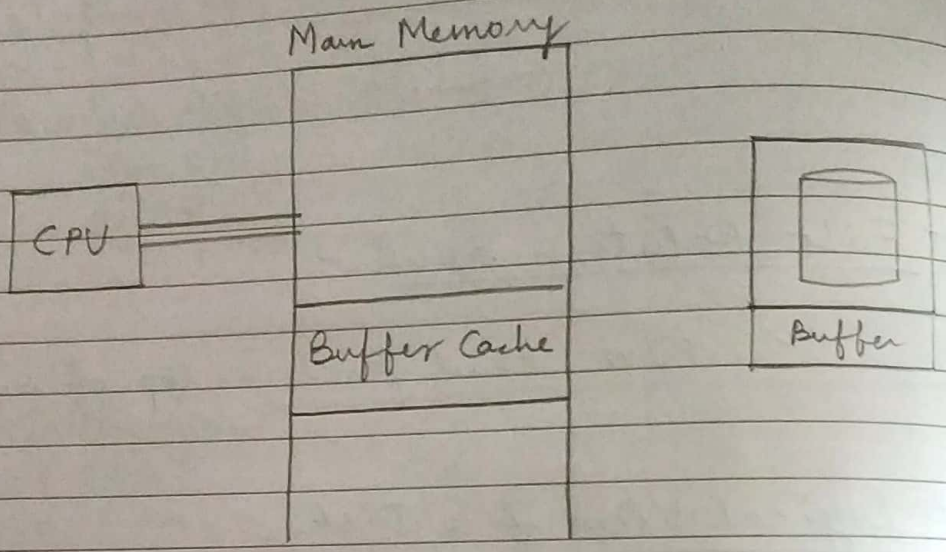
* OS Views the disk as a collection of blocks

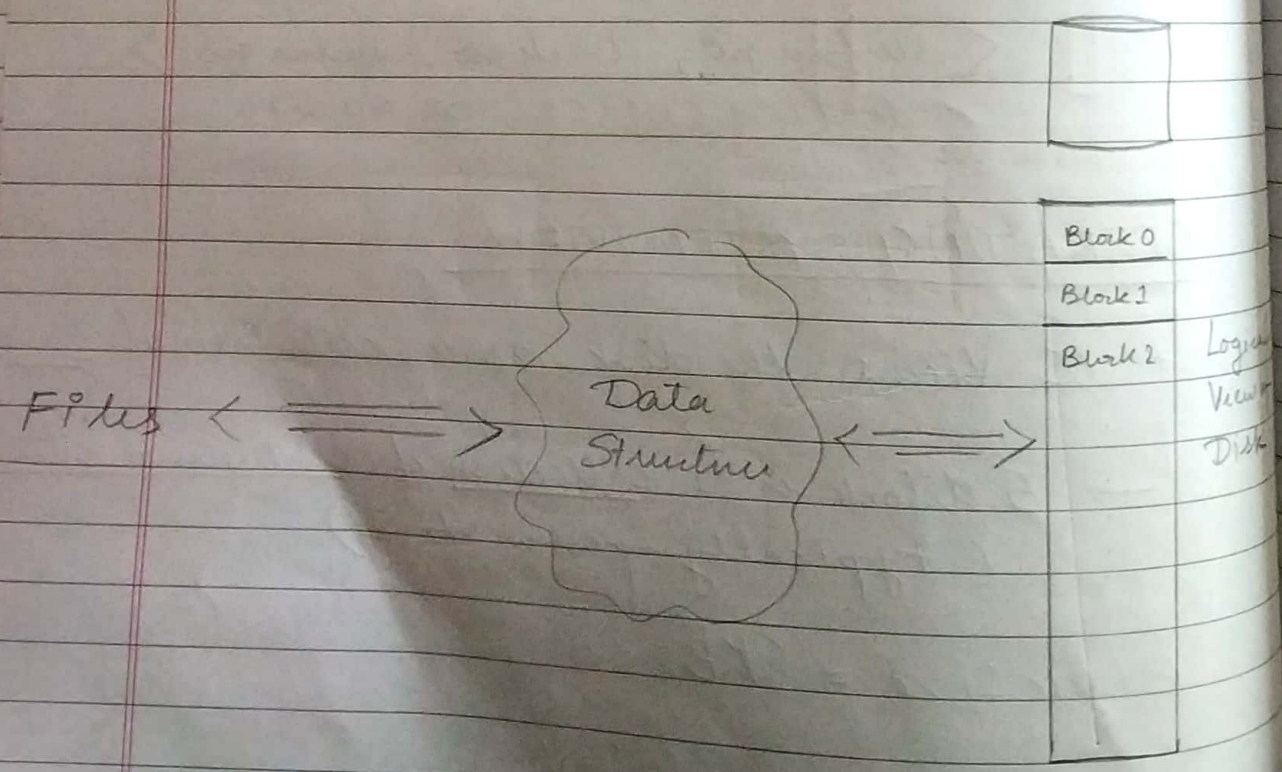
→ A block is a collection of contiguous sectors (typically power of 2)

Block Size = 8KB.





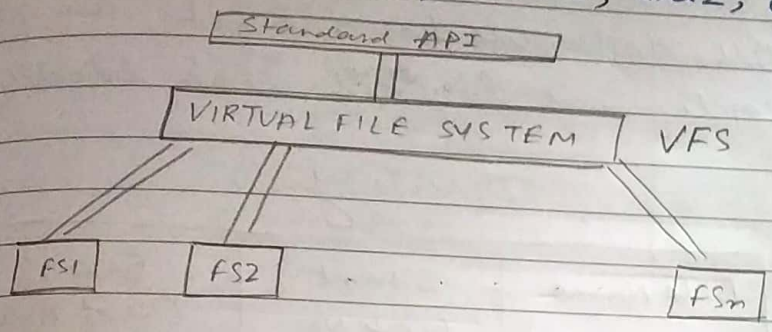




Scanned by CamScanner

FILE SYSTEM

* Data Structure that is maintained by the OS. (on disk) so that files can be stored and accessed.
  - FAT, FAT 32, NTFS, ext2, ext 3.



* Format a Disk
  - Creates a new (Empty) file system on the disk

Sector

H	Data	T

Header
Track No.
Sector No.
Good/Bad.

Trailer : CHECKSUM (CRC).

* Info Maintained by FS

(a) Metadata about the FS - File System Name
Size
No. of Files
No. of free disk blocks
Info about file data

(b) Directory - File name & other info
(c) Locate the data blocks for a file
(d) Maintaining info about free disk blocks

Scanned by CamScanner

ALLOCATION METHODS

* How the data blocks of a file are stored in disk ?

(1) Contiguous Allocation

* All data blocks of a file are allocated contiguous logical disk blocks.

Directory

File Name	Start Block No.	Size (No of Blocks)
abc.pdf	58	12
xyz.txt	10	3

Advantage

- Simple to implement
- Access is typically faster

Disadvantage

- Difficult for file to grow
  ⇒ OVERHEAD
- Lot of disk fragmentation

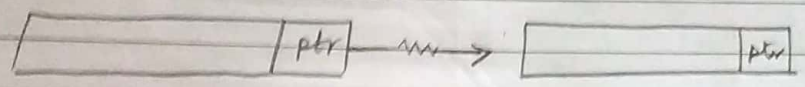
Scanned by CamScanner

(b) Linked Allocation

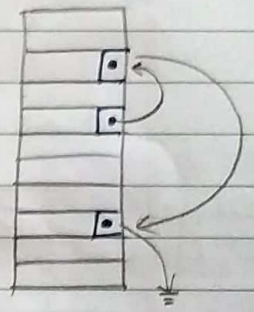
* Each file data is stored as a linked list, of data blocks

  - No External Fragmentation
  - Each block must store a pointer to the next block.
    ⇒ Data Capacity Decreases

    Block :



* Random Access to file data
  - Linear Dependency

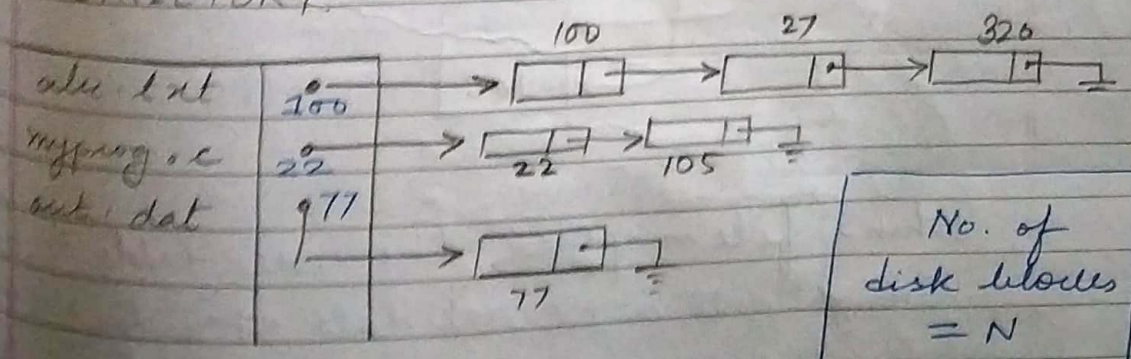


* If a data block gets corrupt, all subsequent links are LOST.

FILE ALLOCATION TABLE (FAT)

* We move all information about the links to a <u>SEPERATE REGION</u> → disk (multiple copies)
  → copy in Memory

DIRECTORY



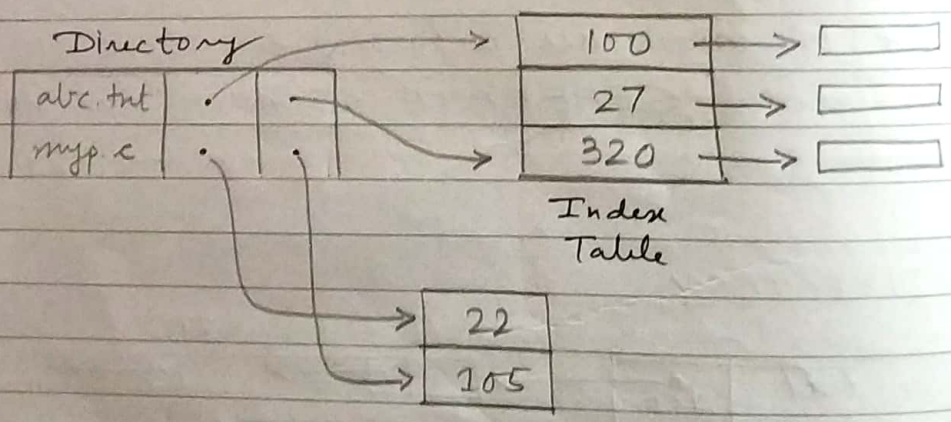
No. of disk blocks = N

Directory

FAT
- stores only next location
- small, can be stored in Main Memory for FAST ACCESS
- may even store Multiple Copies

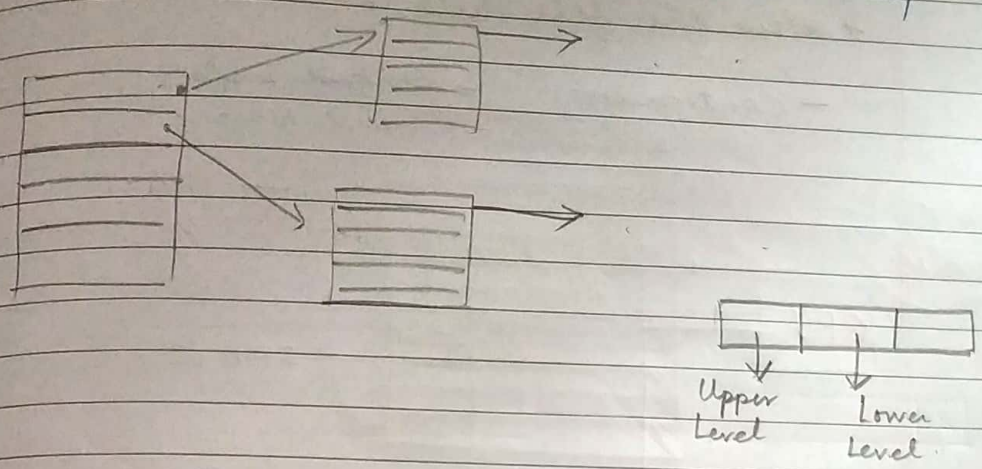
(c) Indexed Allocation

* Every File will have its own index table



When a file is opened, the index table is brought into a Kernal data structure in Main

* For large files, you may need
MULTI-LEVEL INDEXING



Example - Index Node in Linux/Unix

Directory

File Name	inode
abc.txt	5

inode

Size	File Info
access perm	
date mod	
# links	
⋮	
Direct 0	Index Table Info.
Direct 1	
⋮	
Direct 9	
Indirect	
Double Ind	
Triple Ind	

Example

Size Block = 4KB
Block = 32 bits
ptr/Blk = 1024

Max Size of File (in # blocks).

$$10 + 1024 + (1024)^2 + (1024)^3 \approx (1024)^3 = 1G$$

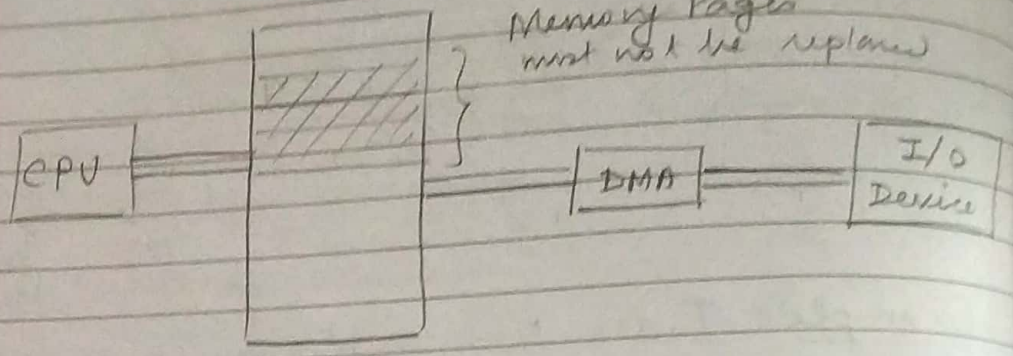
$\therefore$ Total Capacity: $1G \times 4KB = 4TB$.

Scanned by CamScanner

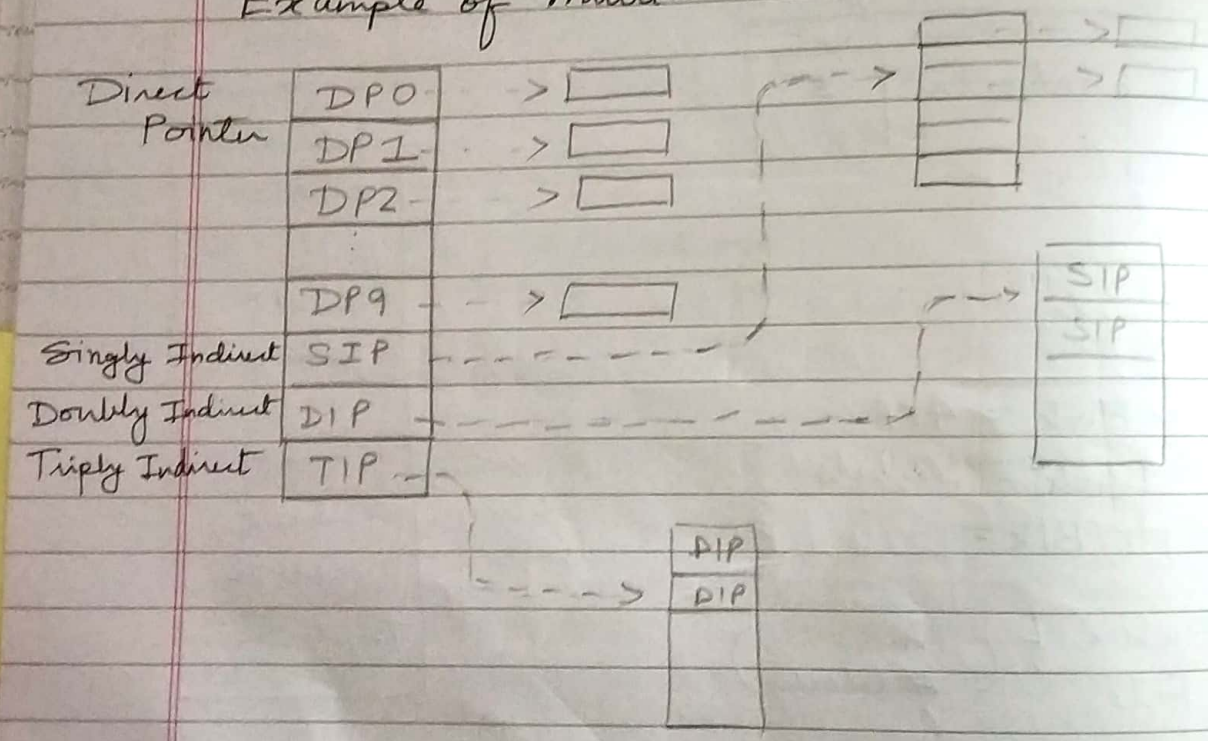
File Systems

* How the file data can be located ?

- Contiguous
- Linked Lists / Linked Blocks
- Index Table



Example of multilevel indexed allocation



Block Size = 8KBytes

Block Address = 32 bits

$\frac{8K}{4} = 2K$

DP = 10

SIP = 2048

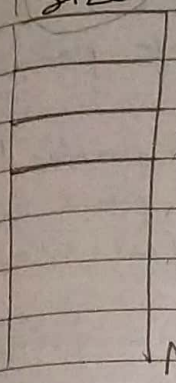
DIP = 2048 × 2048

TIP = 2048 × 2048^2

$\approx (2048)^3 \rightarrow 8G$

Max File Size = 8G × 8KB = 64TB

How many bits → (48 bit field) 64TB → 46 bits



offset (7500) < 8K.

↓

Stored in DPO

(2A3B6C Hexa)
offset

(0010 1010 0001 1011 0110 1100)

8KB / block

2^{13} / block

↓

0010101000 1

block no wrt file system

331 ≥ 10 → not DP, SIP.

Keeping Track of Free Blocks.

(a) Bit Map Encoding

Stored in say SUPERBLOCK

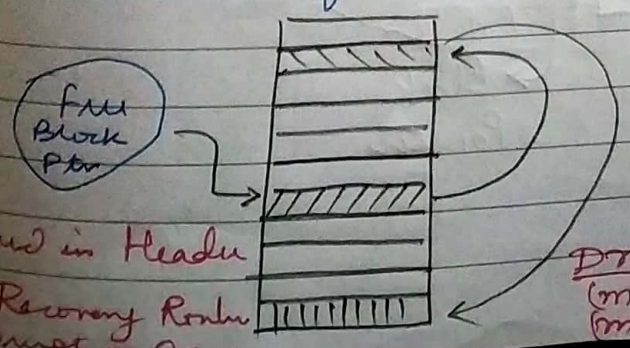
* Very Fast
* Suitable for Smaller Systems.

BM	0	1	2	. . .	N-1

N → Number of Blocks

BM[i] = 0; free block.
= 1; occupied block.

(b) Linked List of Free blocks



* Each free block stores pointer to next free block.

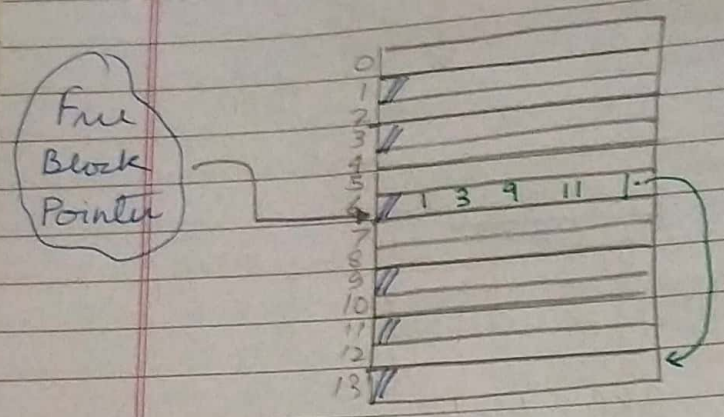
Stored in Header

Helps in Recovery Routine in case Corruption Occurs

Drawback → Request m blk
(m) → locate
(m) → use || double (X)

(C) Blocked Linked List of Free Blocks.

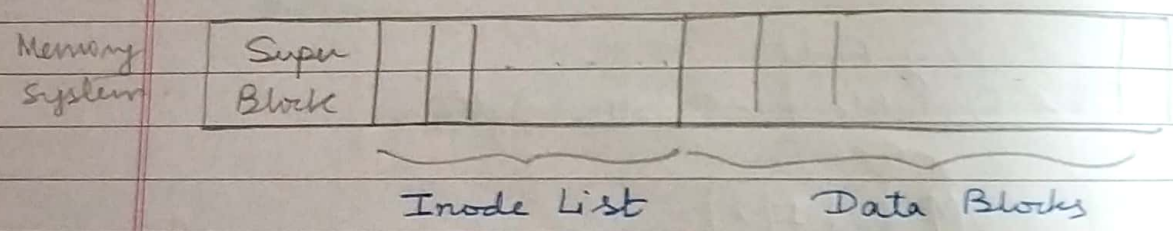
* A disk can store K disk pointers.

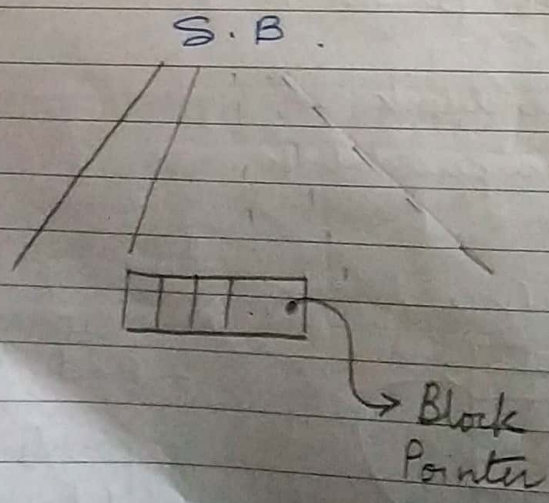


Each pointer block can point to (K-1) free blocks, with a pointer to the next pointer block.

Unix uses similar stuff -

UNIX - LIKE File System -





Scanned by CamScanner

File System Implementations

basic two needs
* keep track of free disk blocks
* keep track of data blocks for each file

Directory Structures

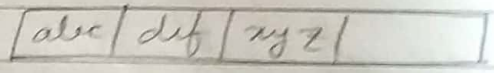
* **Directory**

Place where the name of a file and other information are stored

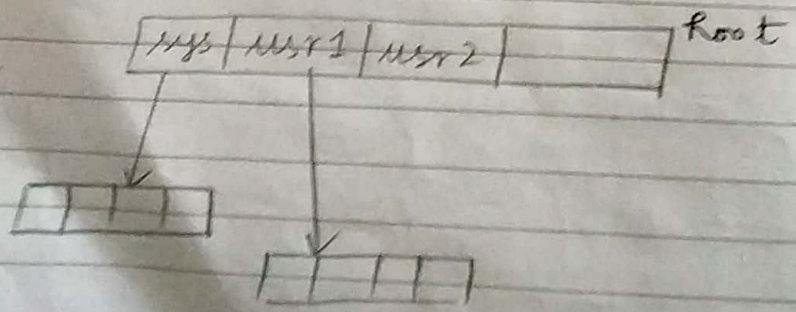
* Various directory structures possible

(a) SINGLE LEVEL DIRECTORY

A single system wide directory exists
→ file names must be unique



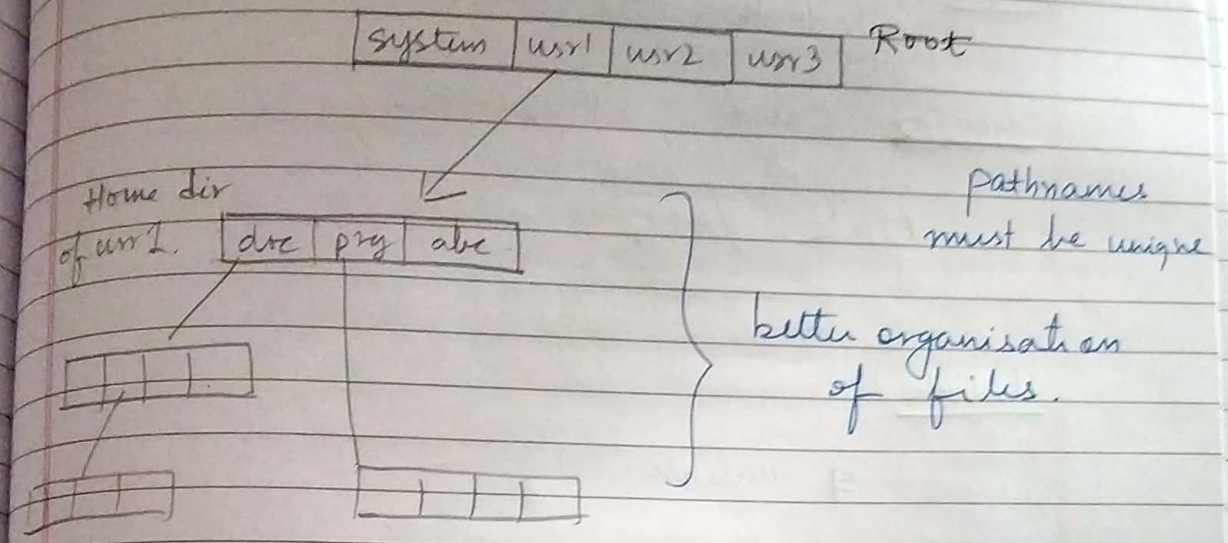
(b) TWO LEVEL DIRECTORY



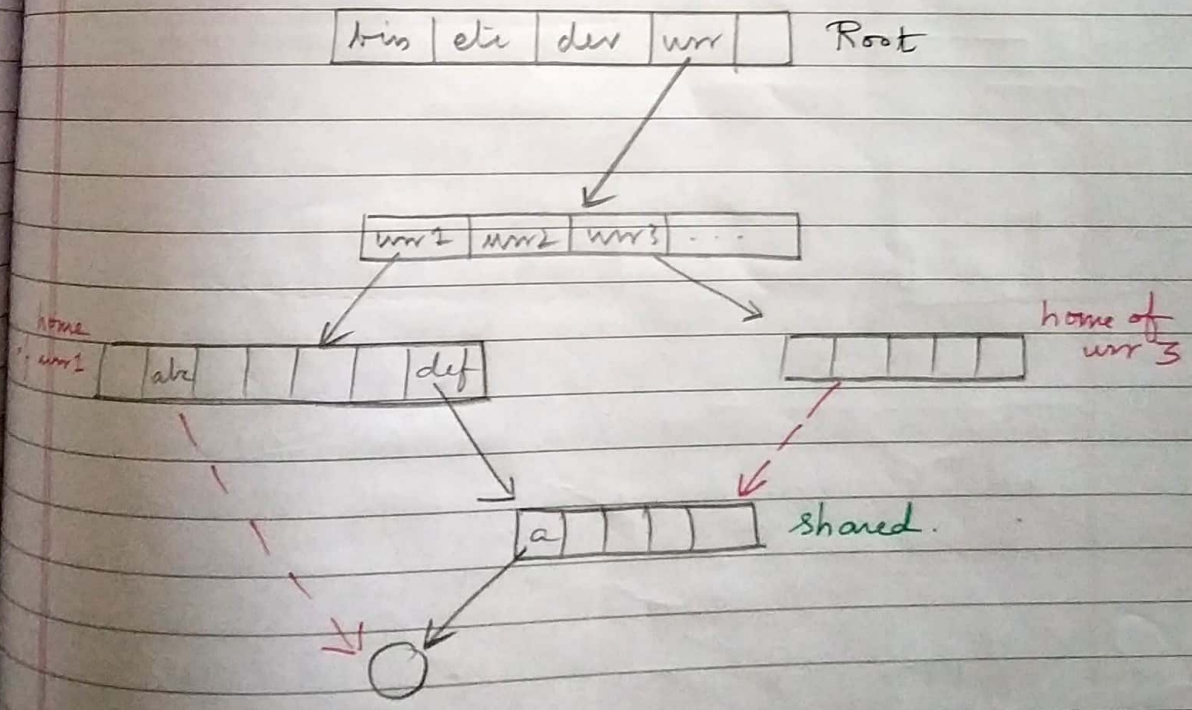
Path
/usr1/a
/usr2/a

(c) TREE STRUCTURED DIRECTORIES

Extend directory hierarchy to any number of levels.



(d) Acyclic Directory Structure. (DAG)



link /usr/usr1/def/a /usr/usr1/abc

* Commands to link and unlink files and directories.

Link :

$ link old-name new-name

Lets say cwd : /usr/usr1

link /usr/usr1/def/a . abc.

Unlink :

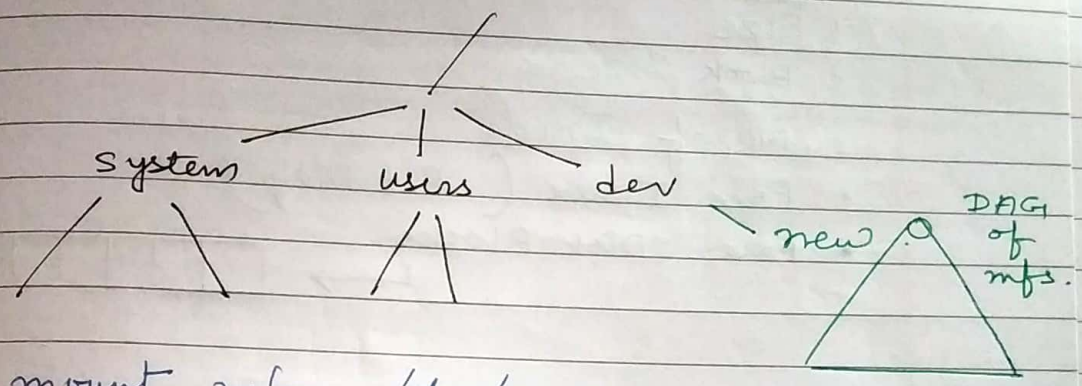
$ unlink

Mounting a File System

can be stored anywhere.

File System has a Unique root.
" " can be represented as a DAG.

Existing System



mount mfs /dev/new
myfilesystem

unmount too.

Index Node (i-Node) → corresponding to each file

Fields Stored.

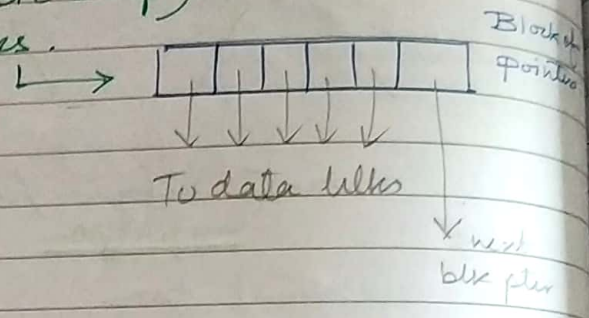
Free / Occupied.	
Type	Regular / Directory
File Size	
Various Dates	Create / Mod / Access
No. of links	1 by default
Access Permiss^n	
DP0	Point to disk blocks.
DP1	
⋮	
DP9	
SIP	
DIP	
TIP	

Case Study : UNIX-LIKE File System

- ∃ an node corresponding to every file and

SuperBlock - Stores info about the FS. MetaData

- Name
- Size
- Block Size
- No. of Inode
- Free Index (Bit Map)
- Free Disk Blocks.



inodes

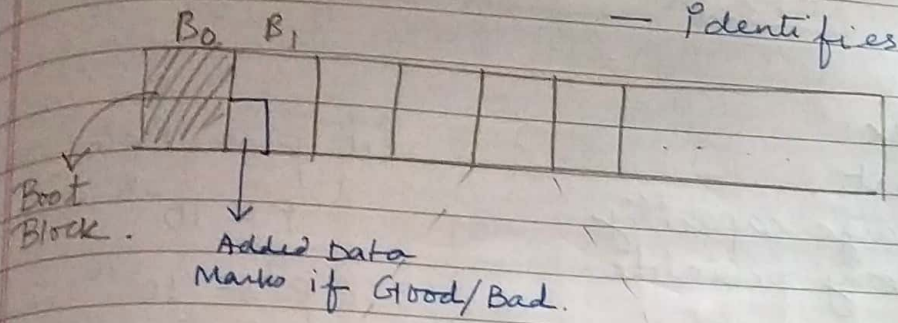
as stored in the disk

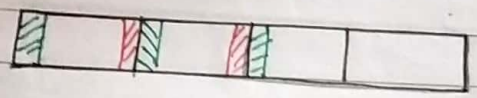
in-core inode
[Gets created wherever a file is opened]
Kernel Data Structure in Memory

Seek - pointer of the present r/w location
Open - how many processes are accessing the f

Formatting a Disk

* Cleaning all data, and creating a fresh empty file system.
  — identifies bad blocks or sectors





header
- track #
- sector #
- good/bad

trailer
- CRC check word.

To identify good/bad block.

Write 0	Read & Store in D.
Read & Verify	// Complement D.
Write 1	Write $\overline{D}$
Read & Verify	Read & Verify
Ans	Ans

→ Basically check if each bit can store both 0 & 1 values.

stuck at 0/1 rectification

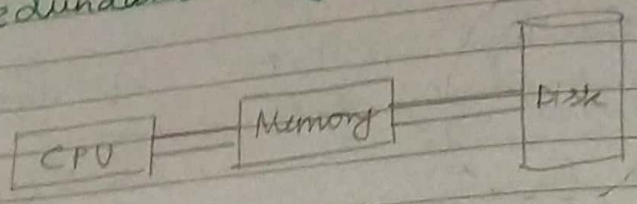
If the free inode list / free disk blocks bitmap get corrupted;
possible to recover them by scanning all the inode & data blocks corresponding to the entries.

Eg:- fchk. LINUX Command.

We also use

RAID → Redundant Array of Independent Disk.
↳ store imp data.

Redundant Array of Inexpensive Disks



Disk:
- fast
- large capacity
- very reliable.

let $p \rightarrow$ Prob (Disk = Fail)

$p \approx 10^{-5}$

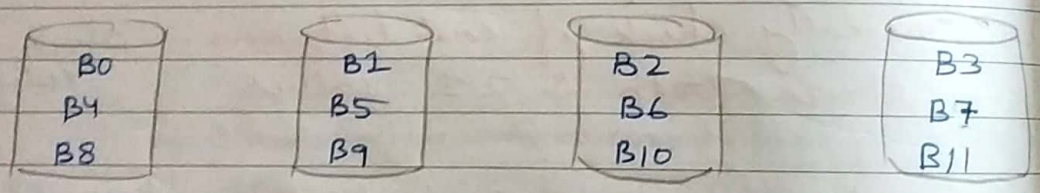
$p^2 = 10^{-10}$

prob (2 Disk = Fail)
very low

Raid Level 0 :

* Block stripping ⇒ Similar to MEMORY INTERLEAVING.

Eg: 4 DISKS



* CPU visualises RAID system as a UNIFIED DISK UNIT.
  → array of logical blocks

* A set of blocks in the same row across the disk units is called a STRIPE.
  ↳ Blocks in a STRIPE can be Accessed in Parallel.

* No Redundancy → No concern for Reliability
* NOT Tolerant against disk failures

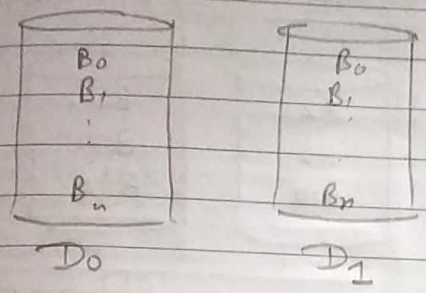
Scanned by CamScanner

Raid Level 1 :

* Mirroring – Fault Tolerance

Every logical block's stored in MORE than ONE disk units (~2).

Egs.



Tolerates upto ← 1 Disk Failure.

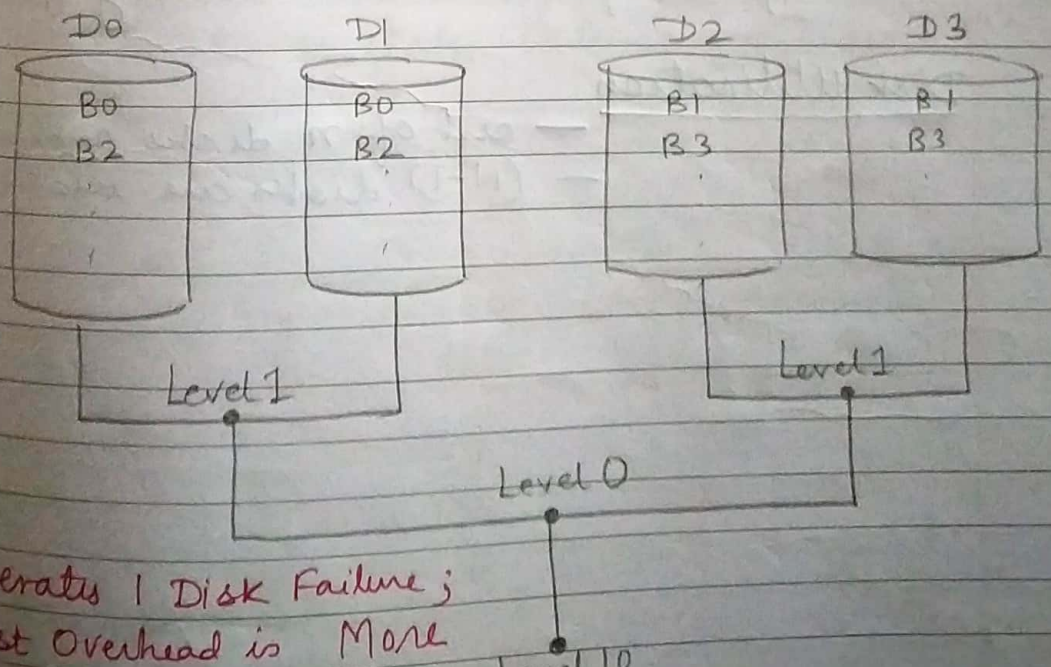
Read Operation

- Scope for Parallelism
- Both units can be Accessed Tog.

Write Operation

- Need to update both D_0 & D_1
- Again; scope for parallelism

Example : Combining L-0 and L-1

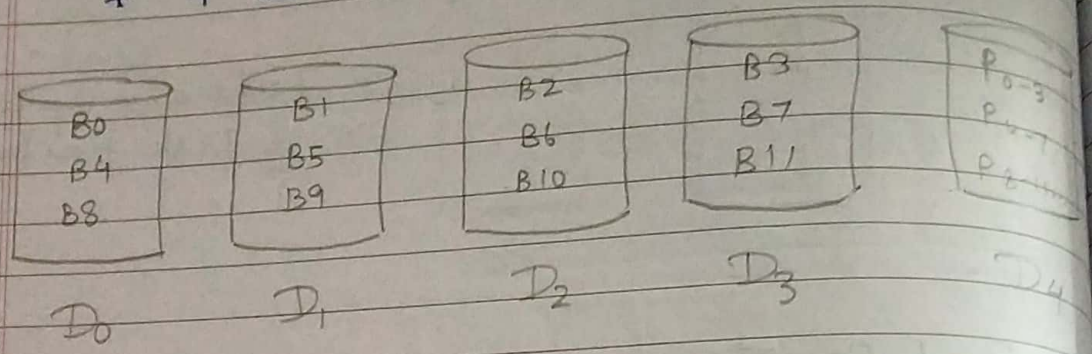


✓ Tolerates 1 Disk Failure;
✗ Cost Overhead is More (2x Actual Cost)

Raid Level 4

* Concept of "parity" disk block
  Save space without sacrifising on fault toleranc

* 4 data disks and 1 parity bit disk



* If D2 fails
  or D0
  D1, D3
  - Take D2 out
  - Insert New Disk

$$P_{0-3} = B0 \oplus B1 \oplus B2 \oplus B3$$

$$B2 = P_{0-3} \oplus B0 \oplus B1 \oplus B3$$

* If D4 fails — All redudancy gets lost

Disk Utilisation

- out of n disks available
- (N-1) disks are usable.

Scanned by CamScanner

Raid Level 5

* Rotating Priority

D0	D_1	D_2	D_3	D_4
B_0	B_1	B_2	B_3	P_{0-3}
B_4	B_5	B_6	P_{4-7}	B_7
B_8	B_9	P_{8-11}	B_{10}	B_{11}
B_{12}	P_{12-15}	B_{13}	B_{14}	B_{15}
P_{16-19}	B_{16}	B_{17}	B_{18}	B_{19}

Raid Level 6

* Extension of RAID-5

- Use of double parity blocks - better redundancy & fault tolerance
- upto 2 disk failures (not all 2)??

D0	D_1	D_2	D_3	D_4
A_1	A_2	A_3	AP_1	AP_2
B_1	B_2	BP_1	BP_2	B_3
C_1	CP_1	CP_2	C_2	C_3
DP_1	DP_2	D_1	D_2	D_3

Scanned by CamScanner

	Raid 0	Raid 1	Raid 4	Raid 5	Raid 6
Implemented as	Disk Striping	Disk Mirroring	Parity Disk	Rotating Parity Disk	Double Rotating Parity Disk
Capacity	NB	NB/2	(N-1) B	(N-1) B	(N-2) B
Reliability	0	1	1	1	2*
Throughput (Sequential R)	NS	$\frac{N}{2}S$	(N-1) S	(N-1) S	(N-2) S
Sequential W	NS	$\frac{N}{2}S$	(N-1) S	(N-1) S	(N-2) S
Read Latency	T	T	T	T	T
Write Latency			2T	2T	3T

Here

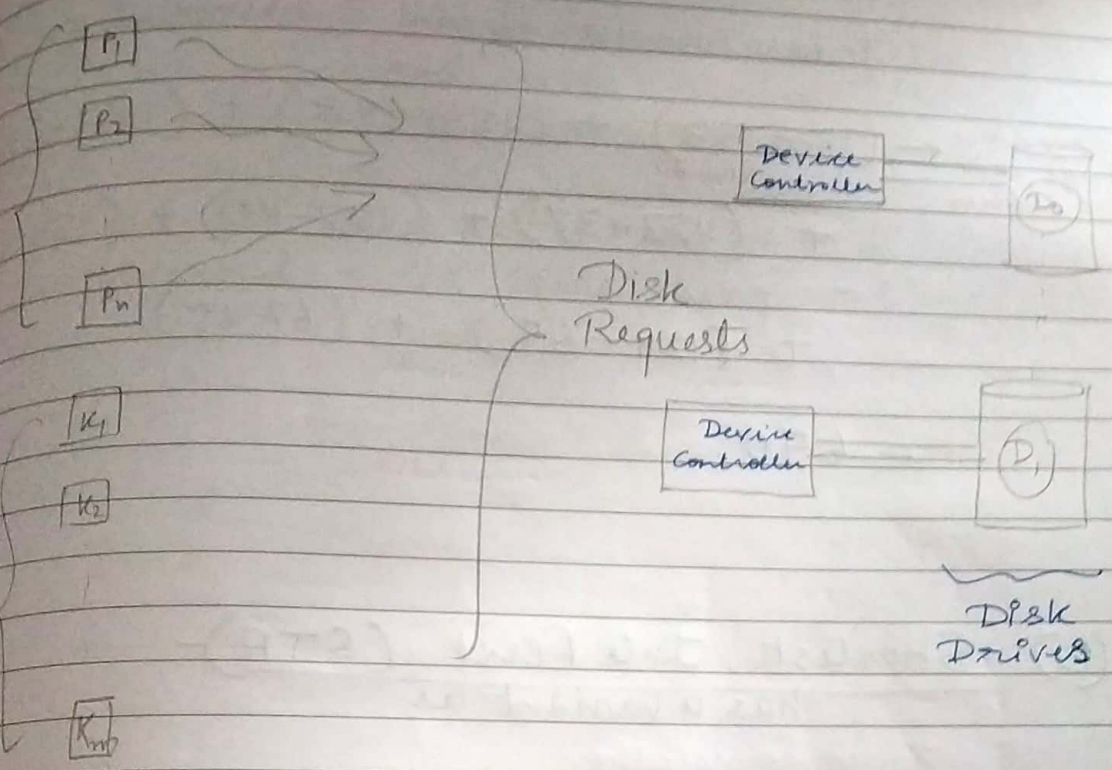
N → Number of Disk Units

B → No. of blocks in each disk unit

S → Disk Throughput under Sequential Workload.

Scanned by CamScanner

DISK SCHEDULING



Disk Scheduling

Eg: 200 tracks, 0 to 199

⊙ Present state of Request Queue :

cylinder or tracks numbers : → 98, 183, 37, 122, 14, 124, 65, 67.

⊙ Current Head Position → $\boxed{\text{Track 53}}$ ☆

What are the different ways in which scheduling can be done ??

Access Time → Seek + Rotational (low)

Bandwidth → $\left(\dfrac{\text{Total Bytes Transferred}}{T_{\text{finish Req}} - T_{\text{request placed}}}\right)$ (high)

Scanned by CamScanner

(A) First Come First Serve (FCFS)

Total Tracks to be traversed =

only thing that can be reduced. ← $(98 - \underset{(\text{init})}{53}) + (183-98) + (183-37)$

$+ (122-37) + (122-14) + (124-14)$

$+ (124-65) + (67-65).$

$= 640.$

(B) Shortest Job First (SJF)
has a variant as

Shortest SEEK TIME first (SSFT)

$(65-53) + (67-65) + (67-37)$

$+ (37-14) + (98-14) + (122-98)$

$+ (124-122) + (183-124)$

$= 236.$

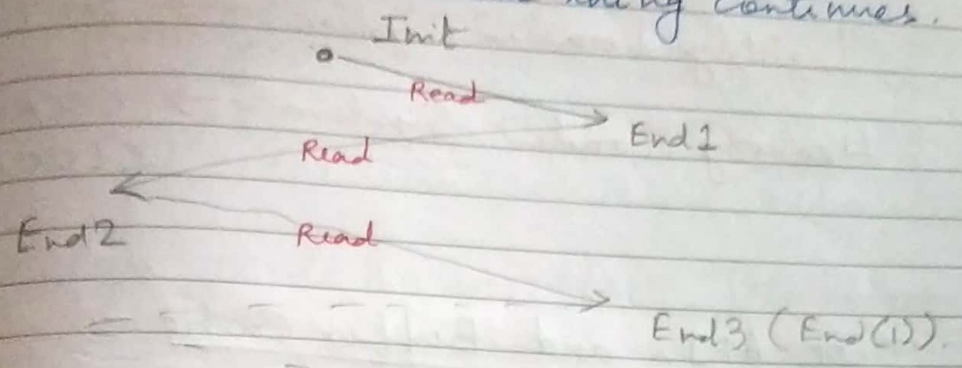
Remember, transfer time is constant for all.

Possibility of Starvation

SCAN or Elevator Algorithm

* Disk Arm, starts at one end of the disks and moves towards the other end SERVICING REQUESTS on the way

* Once the other end is reached – direction is reversed – the same thing continues.



⊛. If the head has just crossed;

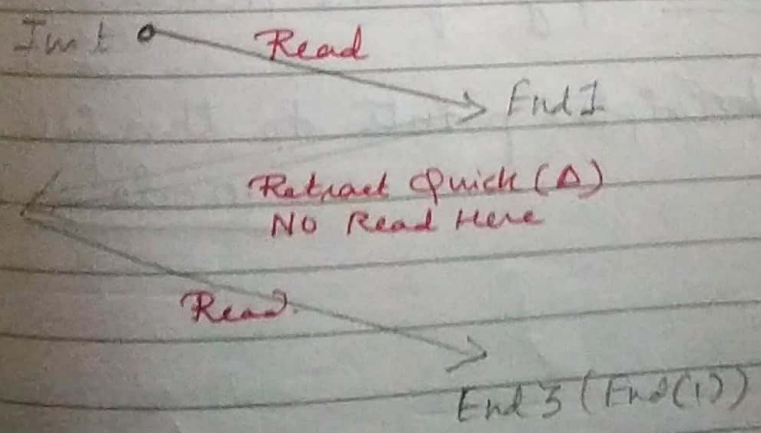
~ Requests towards middle ~ T delay

~ Requests towards edges ~ 2T delay

↓

The purpose for this being, when we have finished one side, higher chance more stuff at the other end. Hence, CIRCULAR LIST Implementation

5] C-Scan : Circular Scan



(F) Look and C-Look.

* Variation of SCAN and C-SCAN;
* Move head only till the last request, not till the disk boundaries.

Lightly Loaded System → SSTF

Heavily Loaded System → CSCAN or CLOOK.

Disk APIs and Buffer Cache –

(a) open System Call

* Given the file name, the inode is located

~/doc/exam.txt

/usr/name/doc/exam.txt

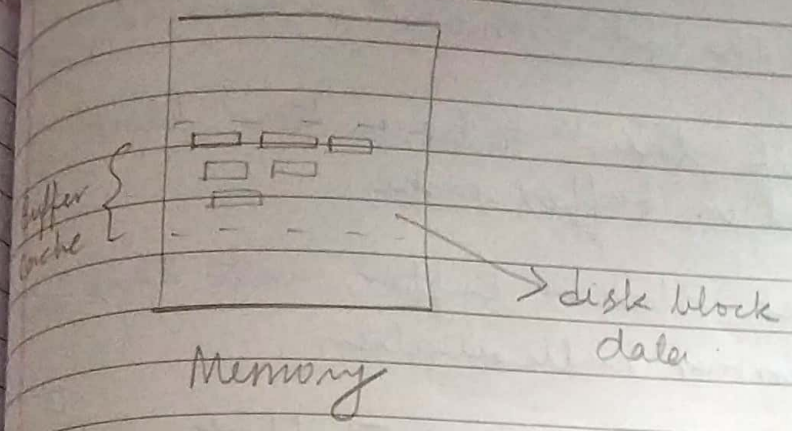
/usr/name → user home directory

exam.txt → inode.

- An in-core copy of the inode is created in kernel.
- The file descriptor points to this file's inode in kernel.

Scanned by CamScanner

(b) Read System Call



* From the 'fd', the in-core inode is located.
  ↳ in-core inode has an offset, file offset which is initialised to zero (seek).

* If the Requested block is not in Buffer Cache:
  - Read the disk block into Buffer Cache
  - Access the data.

Else data access is done directly from Buffer Cache.

Scanned by CamScanner

(C) Write System calls

* From the fd, "inode" is located.

* Check if the "to be written" block is present in buffer cache

No - Get Block in Buffer, write there and further handle situation.

YES - if not, get entry block in buffer and write there

* Periodically, all "dirty" buffer blocks are written back to disk.

Flush forcefully.

[D] close system calls

* Write back dirty blocks of the file from the buffer cache into disk

* Write back in-core inode fields, into disk inode.

Scanned by CamScanner

Search_mkdir ()

{

// Direct Pointers

SIP ->

Blocks 1 and 2

inode table

Ihdr

inode for new file/dir

Scanned by CamScanner

when create new file → 0 INE

→ 1 DIR

name <<

CWD → inode → Search in Data.

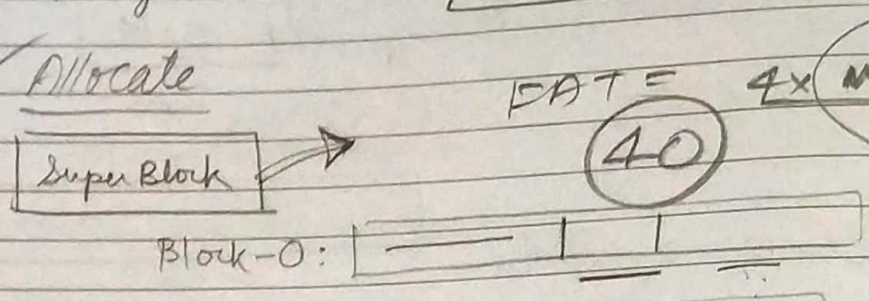
Sea

97
× 79
63 | 160 | 63
763

✓ File System Size & Block Size CLA

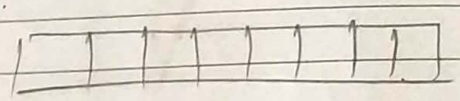
✓ Allocate

FAT = 4 × (No. of Blocks)



$64 \to 2^{26}$, $1M = 2^{10}$ } 2^{16}

Array of Block No.



(Block Size) → P/(F.S.S / Block No.)

✓

FAT_{size} = (Block No) × size int

$(Block)^2 < \frac{FSS}{4}$ 2^{32} 2^{26} 2^{22}

"r" "w"

$FSS < \frac{(BS)^2}{4}$

read(fd, ——)

$(FAT)_{size}$ = (No. of Blocks) × sizeof(int)

No of Blocks = $\left(\frac{FS\ Size}{Block\ Size}\right)$

$\frac{size(FS)}{size(Block)} \times sizeof(int) \leq size(Block)$

$size\ of(FS) \times size(int) \leq size^2(Block)$

directory

array of structure

(F) index ["name", Index to FAT (1st block)]

close [curr block, offset in block]

index ["name", start block, curr block, offset, end block, endoffset]

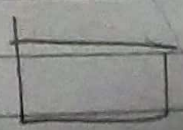
open → O_CREAT ← Bit vector
O_RDONLY
O_WRONLY | O_TRUNC (size == 0)

close → curr block/offset ;

read → (fd, *, len)

write → update FAT ; End Block, Eof, offset

my



Master Module

- first needs to create keys for the various shared memories.
  We shall use ftok() for that.

Create Msg Structure → long
char*

Create Page Table Structure
Create Free Frame List DS
Create Process Info DS

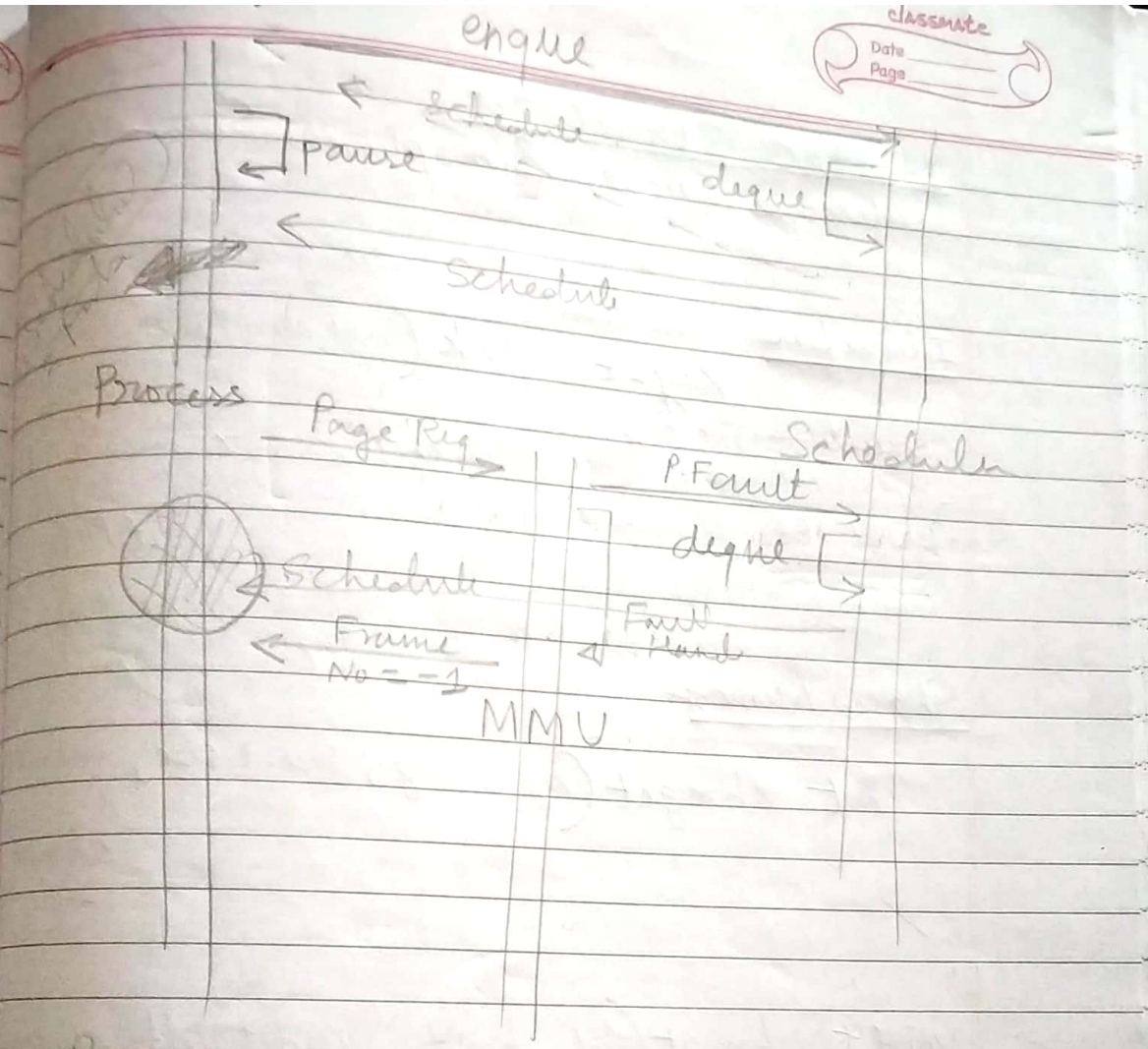
int main (argc, argv[argc, argv[5]])
{
key_q1 = "/tmp/
key_q2 =
key_q3 = ftok
key_sm1 =
key_sm2 =
key_sm3 =

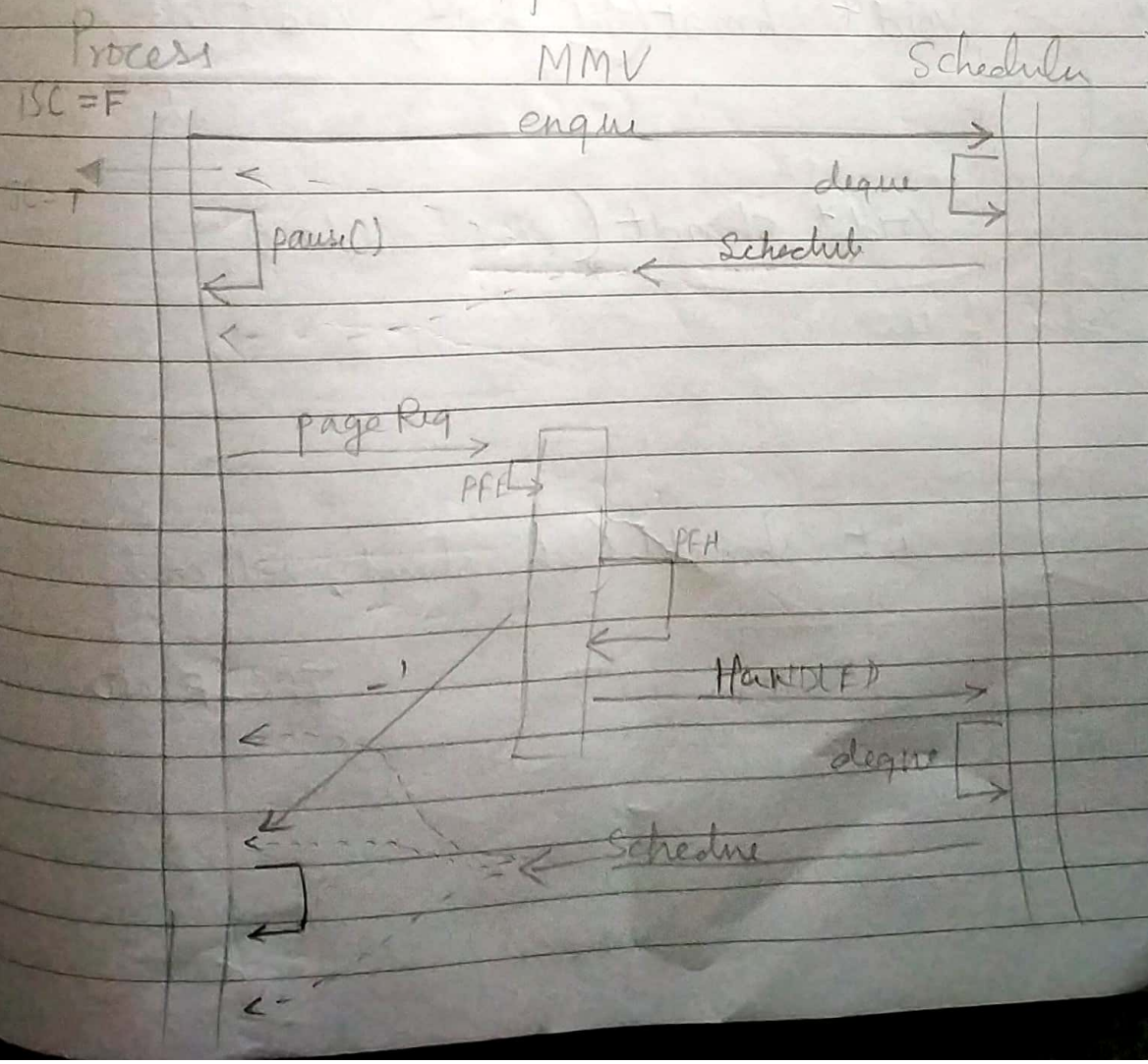
Get Shared Memory shmget()
Get Message Queues msgget()

Attach SM1, SM2, SM3.

Var0 Var1

Scanned by CamScanner


enque
Schedule
pause
deque
Schedule
Process
Page Req
Scheduler
P.Fault
deque
Schedule
Frame No = -1
Fault Hand
MMU


Process
MMU
Scheduler
ISC = F
enque
deque
pause()
Schedule
page Req
PFE
PFH
-1
HANDLED
deque
Schedule

<sys/ipc.h>

XSI IPC Structures

Message Queues, Semaphores, Shared Memory

Identifier

existing Accessible File — lower 8 bits

```
key_t ftok(const char *path, int id)
```

key_t Success
-1 Error

generate unique key
- since we take only lower 8 bits of id
⟹ MAYBE collisions

APUE 557

<sys/shm.h>

Shared Memory

```
int shmget(key_t key, size_t size, int flag)
```

shmid Success
-1 Error

```
void* shmat(int shmid, void* addr, flag)
```

(NULL)

pointer to shm
-1

```
void* shmdt(void* addr);
```

Return Value of this

0 Success
-1 Error

```
int shmctl(int shmid, int cmd, NULL)
```

↓

IPC_RMID

Scanned by CamScanner

Can only be used b/w processes with common Ancestor.

int pipe (int pipefd[2])

int pipe2 (int pipefd[2], int flags)

read	write

pipefd pipefd + 4

Zero Success
-1 Failure

See APUE 568 Pg

We need to close the write() end of a PIPE before Reading from it ELSE EOF won't get after all data read

<sys/stat.h>

int mkfifo (const char* path, mode_t Mode)

int mkfifoat (int dirfd, const char* path, mode_t mode)

path → Name of FIFO file

mode
↳ 0666
0777

Zero Success
-1 Error.

0 → Octal

		r	w	x
6	owner	1	1	0
6	public	1	1	0
6	others	1	1	0

Read Write Execute

Usage is Similar to any File (just this one is FIFO).

fork()

pipe(), mkfifo(),

shmget(), shmat(), shmctl()
shmdt()

dup() dup2() read() fopen() fscanf()
open() close() write() fgets() fprintf()

wait()
waitpid()

srand()
rand()

exec()
execvp() execvpe()
execlp() execlpe()

pthread_create()
pthread_join()
pthread_exit()

pthread_mutex_init()
_lock()
_unlock()

pthread_attr_init()

pthread_kill()

Signal Handler

Scanned by CamScanner

pid_t fork(void) sys/types.h unistd.h

As soon as fork() encountered, program splits into CHILD & PARENT.

In Child : Return Value of 0

In Parent : Return Value of Child Process Id.

Returns (-1) on Error to Fork new Process

pid_t getppid(void) → Parent की pid_t

pid_t getpid(void) → current process की pid_t

always successful

Page 262 APUE.

pid 1 → init()

sys/wait.h

pid_t wait(int *status) wait for one child to terminate

wait for process specified by pid_t terminate.

pid_t waitpid(pid_t pid, int* status, int options)

status → Stores termination status info

options → Constants

WIFEXITED(status) True for normal termination exit() or _exit()

WEXITSTATUS(status).

- returns Exit Status

exit( ) or _exit( )

Bench की Stuff

sys/types.h signal.h

int kill(pid_t pid, int signal)

Send 'signal' to pid.

0 → Atleast 1 Signal sent successfully

-1 → Error.

Scanned by CamScanner

K, m, f, s

MQ1, MQ2 ①

MASTER

SM1 Create Page Table (m size)

SM2 Create free frame list

Creates:
- Ready Queue (MQ1)
- Memory Queue
- Instruct Queues

K processes —
$u \sim rand(1, m)$
$m_i \simeq u$] allocate frames proportionately
$len(R_i) \sim rand(2m_i, 10m_i)$.

CHECK UP

K processes (250 ms spacing) || waits for Scheduler Notif.

SCHEDULER

- Waits() on MQ1 (Ready Q)
- Starts scheduling when process gets added
  - Selects 1st process from RQ
  - Remove from Queue MQ1
  - Send Signal for exec
  - Wait for Notif from MMU
- ↔ block till Message from MMU
  - → Type 1: Page Fault — enqueue process
  - → Type 2: Terminated
- ↔ schedule first process
- Notifies Master about Termin.

MQ2

SM1, SM2 ② IQ_i waits() on creation

MMU

MQ2

PROCESS

waits on creation

MQ1, IQ_i

R_i

Sends (-9) to mark end of R_i

- Sends Page Number to MMU from R_i
- waits for Response
  - ⊙ Valid Frame Number — Parse next no.
  - ⊙ Page Fault (-1) — save current elem of R_i; wait() it is already out of the Ready Queue. Scheduler enqueues it once Fault Resolved
  - ⊙ Invalid Page Reference (-2) terminates.

P_i (IQ_i)

P_{ii} (IQ_{ii})

P_{iii} (IQ_{iii})

Waits for Page Number on $(IQ)_s$.

- -9 sends Type II to Scheduler release frames
- Receive Page Number — increment TimeStamp
- Check TLB, Send Frame No.
- Fail: Check Page Table for Process. Send Frame No.
- (Page Fault) send (-1)

Invoke Page Fault Handler

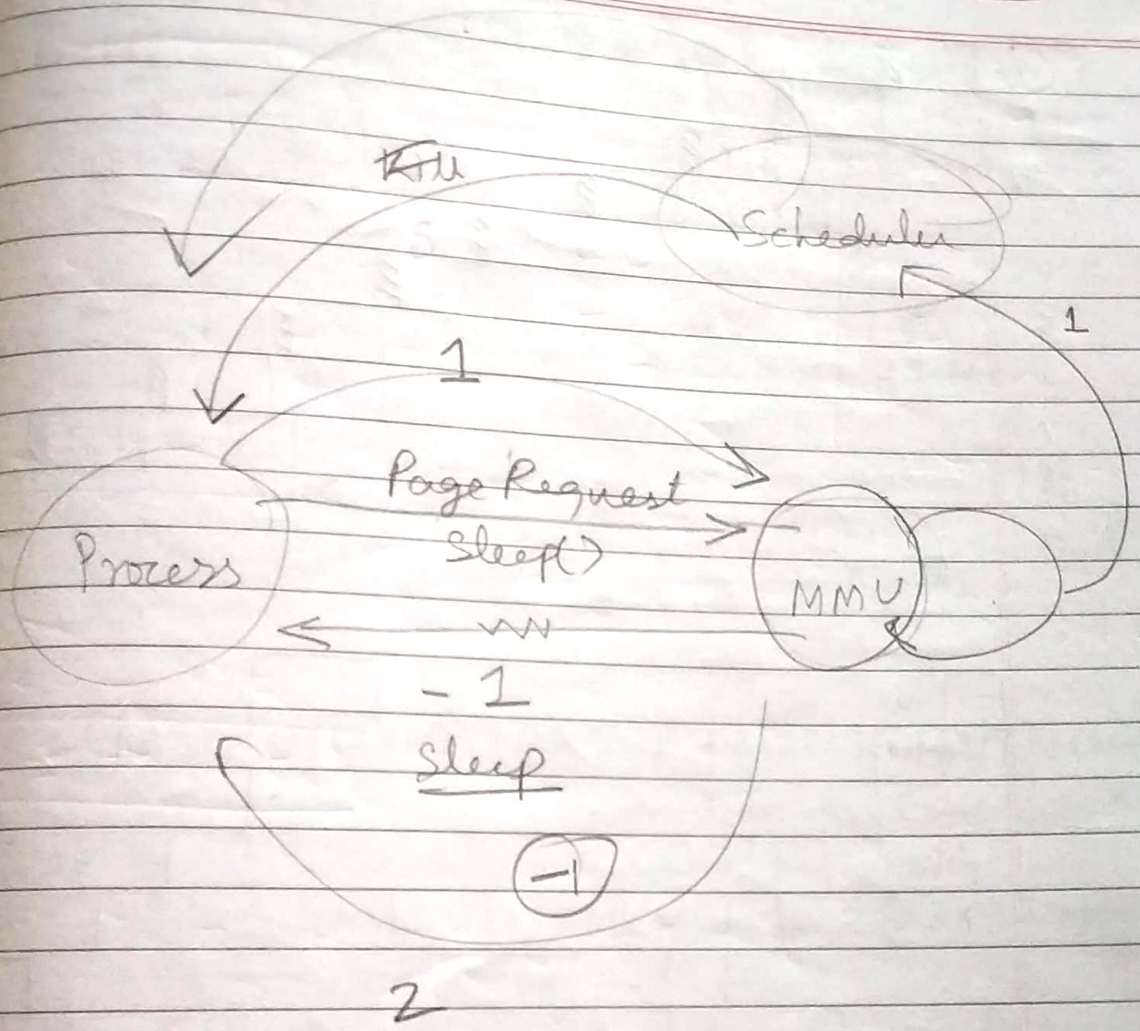
→ FreeFrame Available — update Page Table, update FreeFrame list

→ No FreeFrames Present — Local Page Replacement

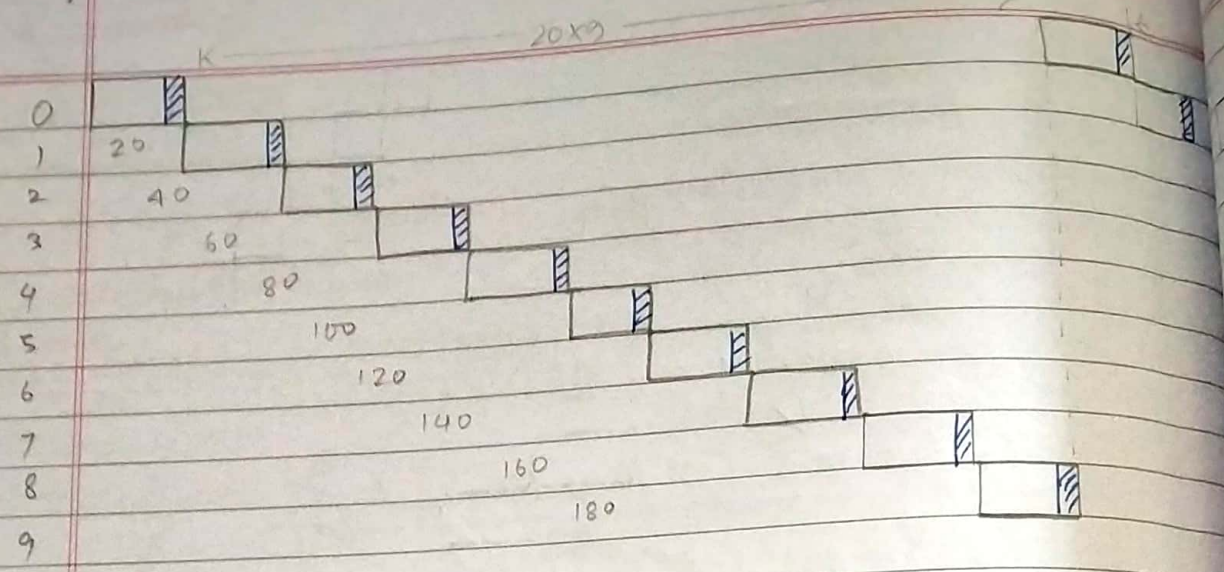
LRU ← Select Victim Page, Replace it, New Frame → update table

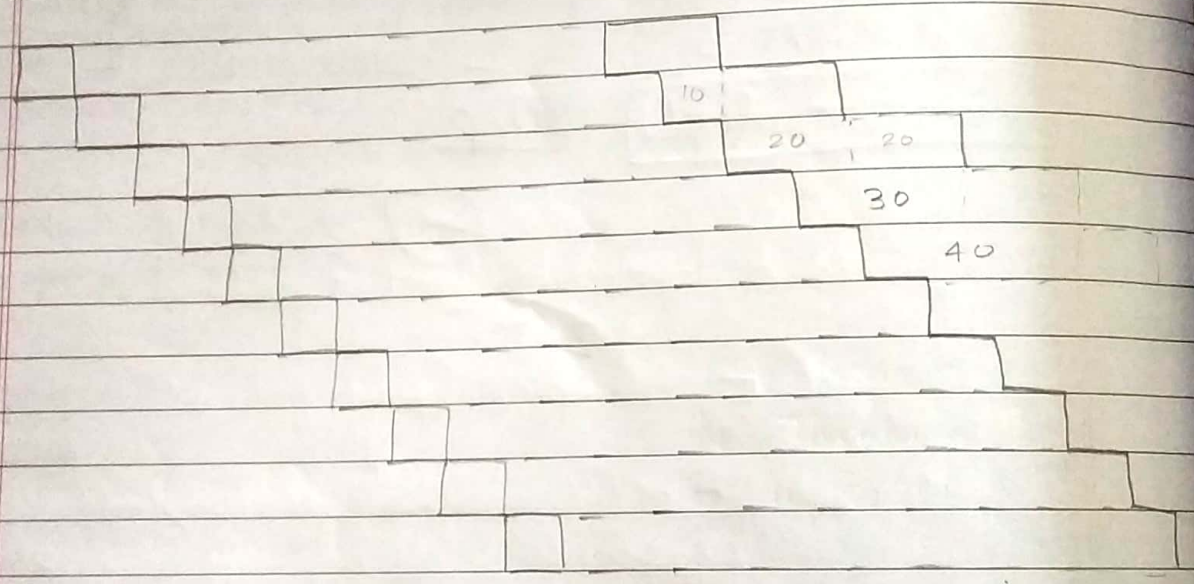
Send Type I message to Scheduler

Scanned by CamScanner



Scanned by CamScanner





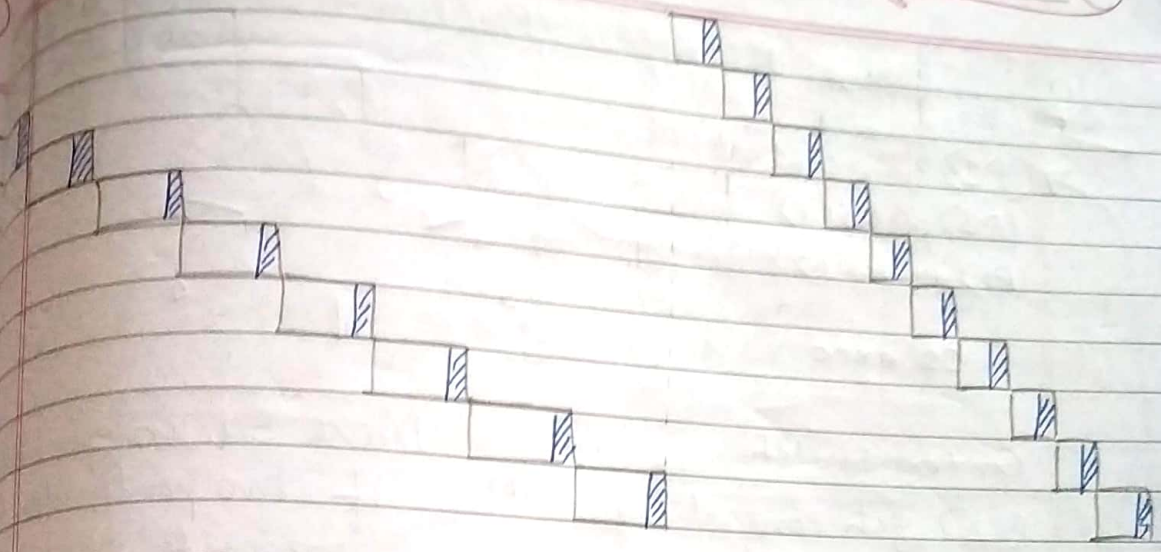
$(P_0) \rightarrow (0 + 20\times9 + 20\times9) + (0 + 20\times9 + 20\times9) \times 49$

$(P_1) \rightarrow (20 + 20\times9 + 20\times9) + (10 + 20\times9 + 20\times9) \times 49$

$(P_i) = (20\times i) + (20\times9) + (20\times9) + ((10\times i) + (20\times9) + (20\times9))\,49$

$$\frac{\left(\sum_{i=0}^{9} P_i\right)}{10} \quad \frac{20}{10}\sum_{0}^{9} i + 49\cdot\sum_{0}^{9} i + \underbrace{((20\times9)+(20\times9))\times50}$$

$$\underbrace{2\times45}_{90} + \underbrace{49\times45}_{47^2-4} \quad \underbrace{(20\times9)\times100}_{18000}$$



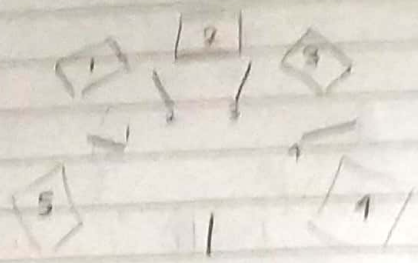
$$P_i = (23\times i) + (23\times 9) + (23\times 9) + \left[(13\times i) + (23\times 9) + (23\times 9)\right] \times 49$$

$$\frac{\sum_0^9 P_i}{10} = \frac{23\sum_0^9 i}{10} + \frac{13\times 49}{10}\sum_0^9 i + (23\times 9)\times 100$$

$$\text{AWT} = \frac{23\times 45}{10} + \frac{13\times 49\times 45}{10} + 20700$$

Scanned by CamScanner

Dining Philosopher Problem – Starvation Explore
Arbitrator Solution



(P1) gets control
(P1) picks (1)
(P1) picks (2)

CS exec

Context Switch
(P2) gets control
(2) not available
Blocked

Context Switch
(P3) gets control
(P3) picks up (3)
(P3) picks up (4)

CS exec

Context Switch
(P1) gets control
(P1) finishes
(P1) releases (1), (2)

Context Switch
(P2) in control
(3) not available
Blocked

Context Switch
(P3) in control
(P3) finishes
(P3) releases (3), (4)

Context Switch

More so, in general if two particular opposite philosophers are fast to get hungry, they will come back quickly to eat again thereby leaving the middle / between slow philosopher STARVED.

Starvation depends on the Thinking Time of the different philosophers

Prevention

(•) Scheduling Threads → manually in such a way so as to implement AGING

(•) Thread System → guarantee that threads are blocked and unblocked in the same order (FIFO) types

P0

(i) flag[0] = True

(ii) turn = 1

(iii) while (flag[1] && turn);

(iv) critical Section

(v) flag[0] = false

P1

flag[1] = True

turn = 0

while (flag[0] && turn);

critical Section

flag[1] = false

		turn
1.1		–
1.2		0
1.3	(flag[0] = false) ✓ enter CS	0
	Critical Section	
	context Switch	
0.1		
0.2	turn = 1	1
0.3	waiting	

[0 – 1)	∅
$(1^- \cdot 1^+)$	P4 (9)
$(2^- \cdot 2^+)$	P4(8) – P5 (2)
$(3^- - 3^+)$	P4(7) – P5(2) – P3 (7)
$(4^- - 4^+)$	P5(2) – P3(7) – P2(6) – P4(6)
$(5^- - 5^+)$	P5(1) – P3(7) – P2(6) – P4(6) – P1(5)
$(6^- - 6^+)$	P3 (7) – P2(6) – P4(6) – P1(5) – P6(3)
6 – 9	P2(6) – P4(6) – P1(5) – P6(3) – P3(4)
9 – 12	P4 (6) – P1(5) – P6(3) – P3(4) – P2(3)
12 – 15	P1(5) – P6(3) – P3(4) – P2(3) – P4(3)
15 – 18	P6(3) – P3(4) – P2(3) – P4(3) – P1(2)
18 – 21	P3(4) – P2(3) – P4(3) – P1(2)
21 – 24	P2(3) – P4(3) – P1(2) – P3(1)
24 – 27	P4(3) – P1(2) – P3(1)
27 – 30	P1(2) – P3(1)

Scanned by CamScanner

Shared Memory { Buffer, Status

CLA (M → PP, N → ??)

(P) ← Worker → (C)

Put 1050 p-1 (INT) into buffer
if FULL wait

- remove one element from buffer
if Empty wait

→ WORKER THREADS

- installed Signal Handlers → f(x).

sigusr 1 → put to sleep
sigusr 2 → wake from sleep

Running Status → shm ⇒ STATUS

→ Schedulu Thread

• send → sleep, → Waking } Workers

Round - Robin Algo → δ = 1

→ REPORTER THREAD

Scanned by CamScanner

Worker (. . . . .)

LD Signal Handler

Producer ( )

generate 1000

srand(i . . .)

for(i = 0 , i < 1000 , i++)

{

generate(1)

~~push in~~

while out + 1 - in

(out - in) + Buffsize.

while(full);

lock [Buff[end ++] = gend.

}

status change

Consumer ( )

while(empty());

Con id

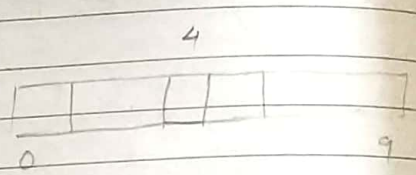
[el = buff[start ++]

$N = [10, 50, 100]$ $\qquad 2^8 = \underline{\underline{256}}$

for n in N

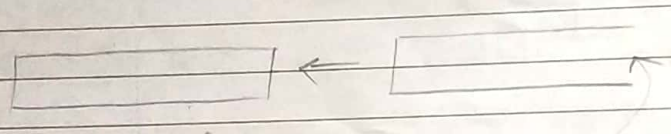
ROUND ROBIN

(I) Ready Queue
(II) Waiting Queue.

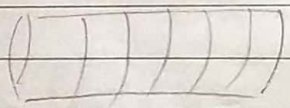


$((0-4)+10) \% 10$

i = 0 JobQueue [ ]
Cur Time = 0;

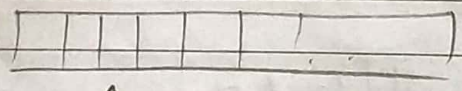


Last Job Index = -1
Cur_Time = 0
(DEQUE).

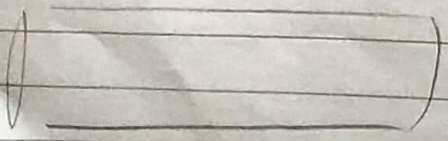


MJI array

LJI.
Cur_Time = 0.



i



```
if ( JQ[i].Arrival <= Cur Time )
{
    if ( RQ.empty() or
    i < RQ.top() → ind
```

```
while(1)
{
    for(        ; i<N; i++)
    {
        if( JQ[i].Arrival <= Cum_Time)
        {
            w = Cum_Time - JQ[i].Arrival
            b = JQ[i].Burst
            rr = ((w+b)/b)
            m[i] = rr;
            S.insert({rr,-i});

        }
}
```

$\longrightarrow$ S.erase(s.be)

```
Cum_Job_Index = -(S.begin() -> second);
Schedule[CJI] = Cum_Time

Cum End Time = Cum Time + JQ[CJI].Burst

for( i=0; i<N; i++)
{
    if( m[i] )
    {
        S.erase({m[i],-i});
        w =
        b
        rr
        m[i] = rr
        S.insert(
```

```
Last_Job_Index = -1
Cur_Time = 0;

while(1)
{
  for (i = LJI+1;  ; i++)
  {   if ( JQ[i].Arrival <= C_T && !Done[i] )
      {   Next_Job_Index = i
          break;
      }
  }   Schedule[NJI].pb(cur_time);

  if( δ < JQ[Next_Job].Burst)
  {   Cur_Time += δ
      JQ[i].Burst -= δ
  }
  else
  {   Cur_time += JQ[NJI].Burst
      Done(NJI) = 1;
  }

  cur_Last_JOI = cur_J
```

Scanned by CamScanner

```
i=0, t=0
while(1)
{
    while( JQ[i].Arrival <= t)
    {
        PQ.push( JQ[i].Burst, i);
    }

    if( PQ.empty() )
    {
        if (i==N) break
        t = JQ[i].Arrival
        continue;
    }

    CurJob = PQ.top();   Cur_Job_Index = Cur_Job.S.
    PQ.pop();
    Schedule[Cur_Job_Index].pb( t)
      t1 = t + Cur_Job.Burst
      t2 =     i==N? INT_MAX : JQ[i].Arrival.

    if( t1 <= t2)
    {
        TN += t1 - JQ[Cur_Job_Index].Arrival
        t = t1;
    }
    else
    {
        rem_Burst = t1 - t2
        PQ.push( rem_Burst, Cur_Job);

        t = t2;
    }
```

Scanned by CamScanner

```
while()  i=0, t=0;
{
        while( JQ[i].Arrival <= t )
        {
            PQ.push({ JQ[i].Burst, i });
            i++;
        }
        if( PQ.empty() )
        {
            if (i==N)
                break
            t = JQ[i].Arrival;
            continue;
        }
        Cur_Job = PQ.top();
        PQ.pop();
        Schedule[Cur_Job.second] = t
        t += Cur_Job.burst;
                first
}
```

%6.2lf

FCFS
NPF_SJF } JobQueue [1] or [2] use

Scanned by CamScanner

* Shell - Command interpreter
  ↳ csh, sh, ksh, bash

/etc/passwd

xyz : /bin/bash
default Shell

① Simple External Commands

gcc -o xyz xyz.c

./a.out

- ls -l

② Input Redirection

./a.out < infile.txt

③ Output Redirection

cat xyz.c > out.txt

④ Combo

⑤ Run a Command in the Background

./a.out < in > out &

⑥ Pipe Mode in → ⟶ → out

Scanned by CamScanner

<sys/types.h>

System Call fork() { <0 ; (-1) if create child unsuccessful
0 → newly created child process
>0 → process Id of child created

— process Id = pid-t (normally integer)

getpid() - process Id of current process.

Basically fork() is a System Call ;
program whenever encounter करता है
then a parallel child process is created
- जिसका Process Id = 0
- Separate Address Space
- Has same values for previous variables
  - in program before fork()
- changes to "COMMON" variables do not get reflected

Scanned by CamScanner