Operating System

INDEX

NAME. : Om Kaldate Std. : Div. :

Roll No. : School / College. :

Sr. No	TITLE	Date	Teacher's Sign / Remark

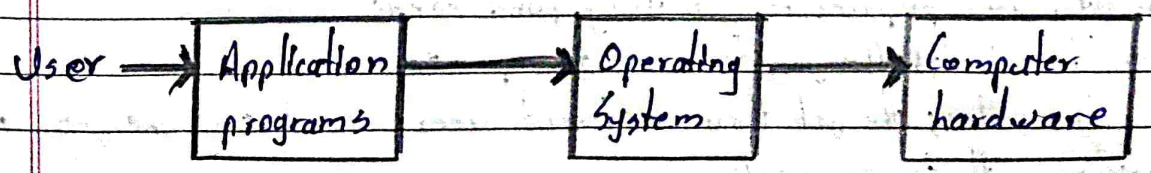
- Application software performs specific task for the user
- System software operates and controls the computer system and provides a platform to run application software.

- An operating system is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

□ what if there is no OS.
- bulky and complex app hardware interaction code must be in app's code base
- resource exploitation by 1 app
- no memory protection

□ What is an OS made up of
- collection of system software

□ An operating system function
- access to the computer hardware
- interface between the user and the computer hardware.
- resource management (aka arbitration) memory, device, file, security, process
- hides the underlying complexity of the hardware aka abstraction
- facilitates execution of application programs by providing isolation and protection.



the operating system provides the means for proper use of the resources in the operation of the computer system.

L2

Page No.
Date

Operating system goals

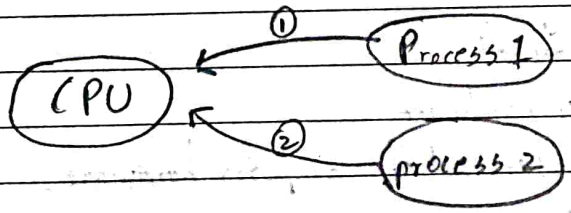
1. maximum CPU utilization
2. less process starvation
3. higher priority job execution.

Types of operating systems

1) single process operating system — MS DOS, 1981
2) batch-processing operating system — ATLAS, Manchester Univ, late 1950s
3) multiprogramming operating system — THE, Dijkstra, early 1960s
4) multitasking operating system — CTSS, MIT, early 1960s
5) multi-processing operating system — Windows NT
6) distributed system — LOCUS
7) real time operating system. — ATCS

1. Single process operating system.

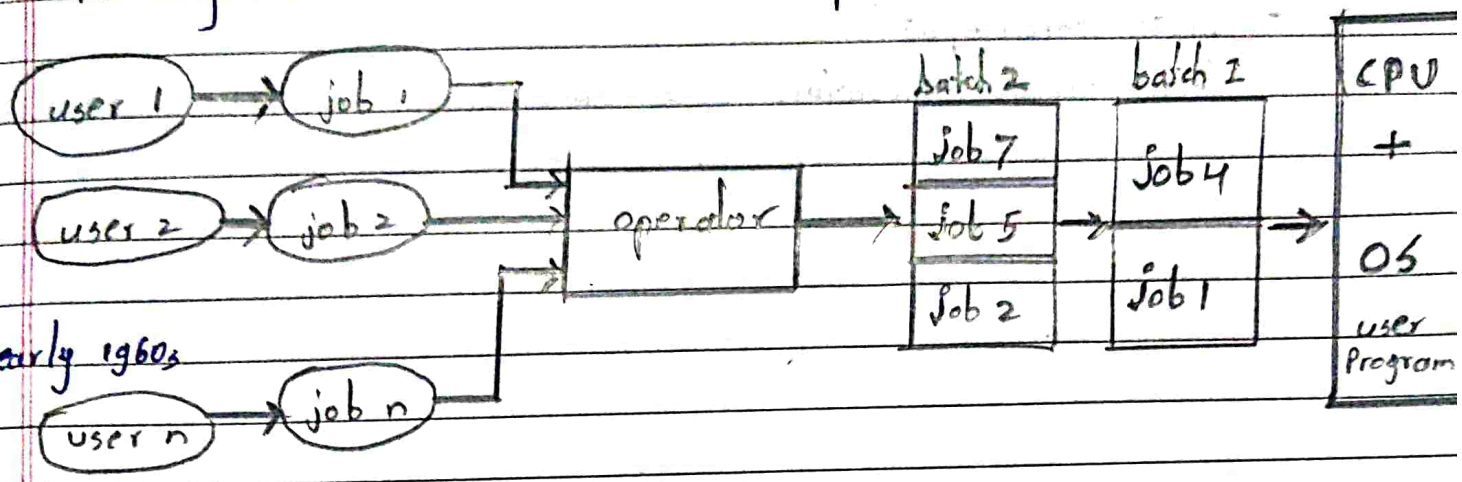
- only 1 process executes at a time from the ready queue (oldest).



2. batch-processing operating system

- Firstly, user prepares his job using punch cards
- then, he submits the job to computer operator
- operator collects the job from different users and sort the job into batches with similar needs
- then, operator submits the batches to the processor one by one
- all the job of one batch are exetted together.

- priority cannot be set, if a job comes with some higher priority
- may lead to starvation, batch take more time to complete
- CPU may become idle in case of I/O operations



1950s - early 1960s

- operator sort the jobs into batches

③ multiprogramming operating system
- increases CPU utilization by keeping multiple jobs (code & data) in the memory so the CPU always has one to execute in case some job gets busy with I/O
- single CPU
- context switching for processes
- switch happens when current process goes to wait state
- CPU idle time reduces

job 1	job 2	job 3	job 4

PCB - process control block

④ multitasking operating system
- it is a logical extension of multiprogramming
- single CPU
- able to run more than one task simultaneously
- context switching and time sharing used
- Increases responsiveness
- CPU idle time is further reduced

Time quantum : 1000 ms
- it gives CPU control to every job for only quantum time

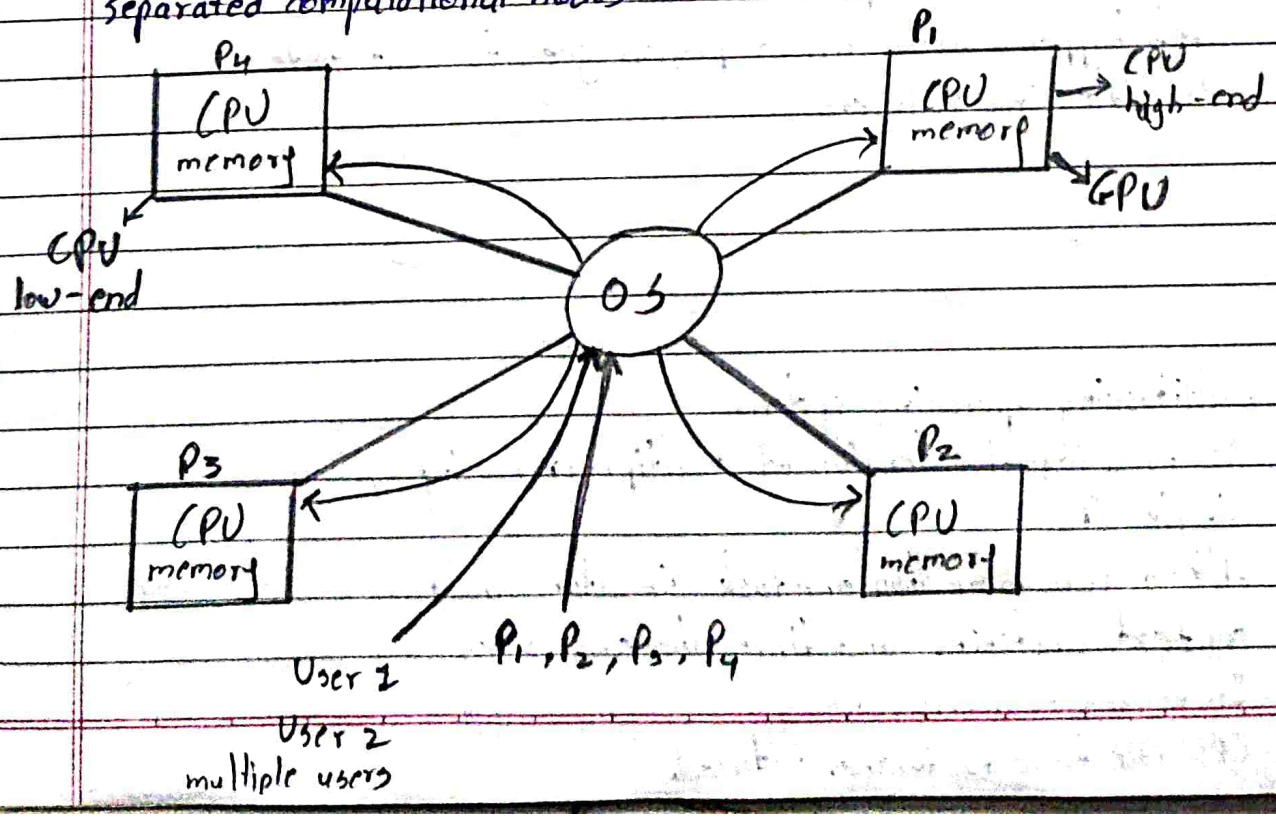
Page No.
Date

(5) multiprocessing operating system
- it has more than 1 CPU in a single computer
- increases reliability. 1 CPU fails, other can work
- better throughput
- lesser process starvation, if 1 CPU is working on some process, other can be executed on other CPU

context switching
Time sharing
CPU ≥ 1

(6) Distributed operating system
- OS manages many bunches of resources. >= 1 CPUs
  >= 1 memorys
  >= 1 GPUs

- loosely connected autonomous
- interconnected computer nodes
- collection of independent, networked, communicating, and physically separated computational nodes



Page No
Date

(7) Real time operating system

- real time error free computations, within tight-time boundaries
- air traffic control system, ROBOTS etc, industrial applications

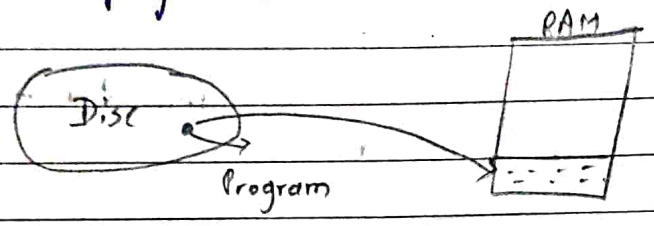
L
3

Page No.
Date

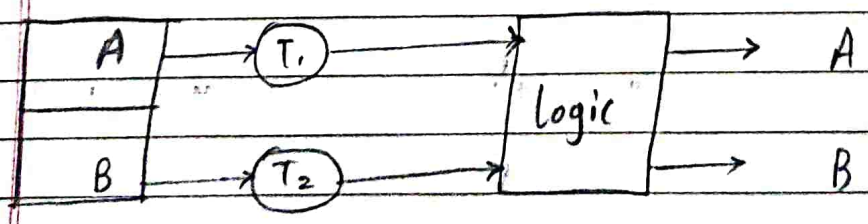
multi-tasking vs multi-threading

- Program - a program is an executable file which contains a certain set of instructions written to complete the specific job or operation on your computer.
  - it's a compiled code ready to be executed
  - sorted in disk

- Process - program under execution. resides in computer's primary memory RAM



- Thread
  - single sequence stream within a process
  - an independent path of execution in process
  - light-weight process
  - used to achive parallelism by dividing a process's takes which are independent path of execution
  - E.g. multiple tabs in browser, text editor (when you are typing in an editor, spell checking, formatting of text and saving the text are done concurrently by multiple threads)



multi-tasking	multi-threading
• the execution of more than one task simultaneously is called multitasking	a process is divided into several different sub-tasks called as threads. which has it's own path of execution. this concept is called as multithreading
• concept of more than 1 process being context switched	concept of more than 1 thread. threads are context switched
• 1. CPU	CPU >= 1 better to have more than 1 CPU
• isolation and memory protection exists. OS must allocate separate memory and resources to each program that CPU is executing	no isolation and memory protection. resources are shared among threads of that process. OS allocates memory to process. multiple threads of that process shared the same memory and resources allocated to the process

• Thread sheduling - threads are sheduled for execution based on their priority. even through threads are executing within the runtime. all threads are assigned processor time slices by the operating system.

Thread context switching	process context switching
• OS saves current state of threads & switches to another thread of same process	OS saves current state of process & switches to another process by restoring it's state
• doesn't includes switching of memory address space.	includes switching of memory address space.
• Program counter. registers & stack are included	

- fast switching - CPU's cache state is preserved	slow switching CPU's cache state is flushed

4. Components of OS

1) Kernel - a kernel is the part of operating system which interacts directly with the hardware and performs the most crucial tasks
- heart of OS core component
- very first part of OS to load on start-up

2) User space - where application software runs, app don't have privileged access to the underlying hardware. it interacts with kernal
- GUI
- CLI

a shell also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed

□ functions of kernel

① Process management
- scheduling processes and threads on the CPU's
- creating and deleting both user and system process
- suspending and resuming processes
- providing mechanisms for process synchronization or process communication
- process and threads schedeling
- process creation and termination.

Page No
Date

(2) memory management
- allocating and deallocating memory space as per need
- keeping track of which part of memory are currently being used and by which process

(3) file management
- creating and deleting files
- creating and deleting directories to organize files
- maping files into secondary storage
- backup support onto a stable storage media

(4) I/O management
- to manage and control I/O operations and I/O devices
- buffering (data copy between two devices), caching and spooling

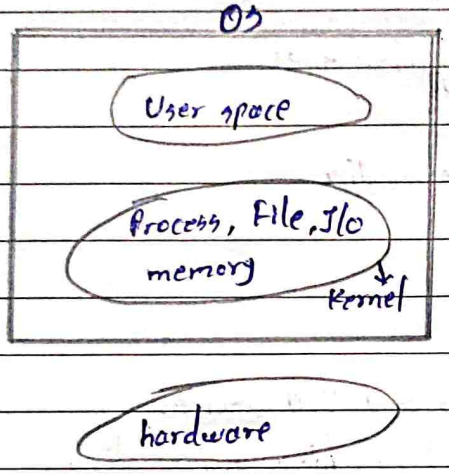
1) Spooling - within differing speed two jobs
Eg. print spooling and mail spooling

2) Buffering - within one job
Eg. youtube video buffer

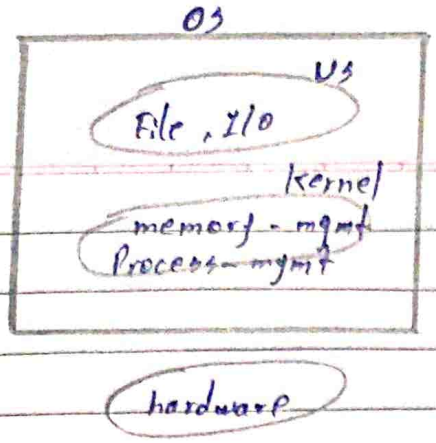
3) Caching - memory caching, web caching

Types of kernels

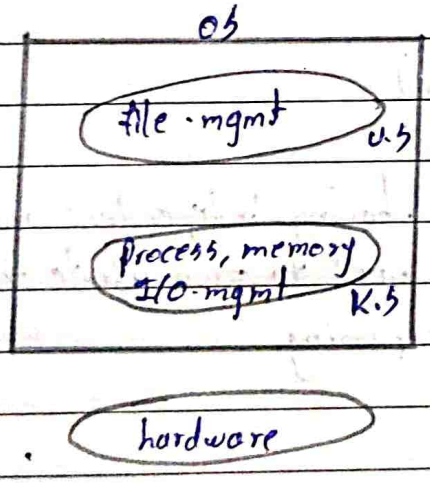
① Monolithic kernel
- all functions are in kernel itself
× - bulky in size
× - memory required to run is high
× - less reliable, one module crashes --> whole kernel is down
- high performance as communication is fast (less user mode, kernel mode overheads)
- Eg. Linux, Unix, MS-DOS



② Micro kernel
- only major functions are in kernel 1) memory management
2) process management
- file management and I/O management in user space
✓ - smaller in size
✓ - more reliable
✓ - more stable
× - performance is slow
× - overhead switching between user mode & kernel mode
- Eg. L4 Linux, Symbian OS, MINIX



③ Hybrid kernel
- advantages of both worlds
- combined approach
- speed and design of mono
- modularity and stability of micro
- Eg. Mac OS, Windows NT/7/10
- IPC also happens but lesser overheads



④ Exo · kernel

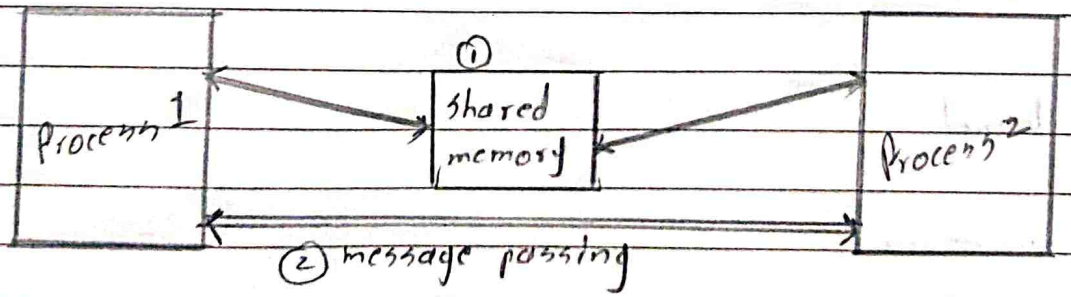
Page No.
Date

⑤ Nano kernel

Q. How will communication happens between user mode and kernel mode?

→ Inter process communication (IPC)

- two process executing independently, having independent memory space (memory protection), but some may need to communicate to work
- done by shared memory and message passing



System calls

- How do apps interact with kernel → using system calls

Eg. Mkdir laks
- Mkdir indirectly calls kernel and asked the file management module to create a new directory
- Mkdir is just a wrapper of actual system calls
- Mkdir interacts with kernel using system calls

Eg. Creating a process
- user executes a process. (user space)
- gets system call
- Exec system call to create a process (kernel space)
- return to user space

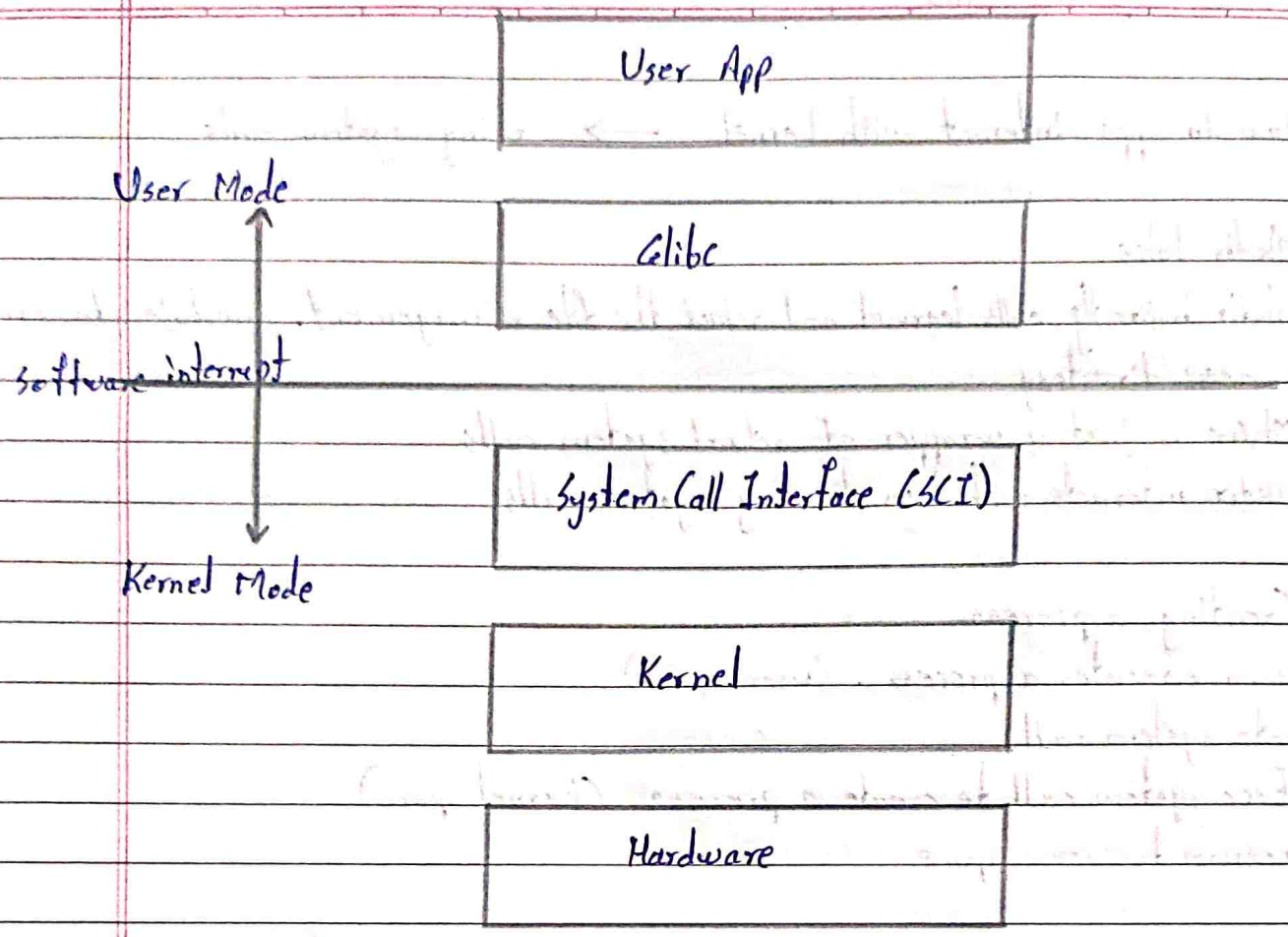
- Transitions from US to KS done by software interrupts.
- System calls are implemented in C.

- A system call is mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform user programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs.
- System calls are the only way through which a process can go into kernel mode from user mode.

* To creating new directory

- GUFI → it is symbol of new fill, which present of desktop

- CLI → it is command which will given in command promt Mkdir → Linux



- Types of system calls

1) process control
- end, abort
- load, execute
- create, process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

2) File management
- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes

Page No
Date

3) Device management
- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

4) Information maintenance
- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

5) Communication management
- create, delete communication connection
- send, recive messages
- transfer status information
- attach or detach remote devices

Examples of windows and unix system calls

Category	Windows	Unix
Process control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Management	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()

	SetFileSecurity ()	chmod ()
	InitializeSecurityDiscriptor()	umask ()
	SetSecurityDiscriptorGroup()	chown ()
• Device Management	SetConsoleMode ()	ioctl ()
	ReadConsole ()	read ()
	WriteConsole ()	write ()
• Information Management	GetCurrentProcessID ()	getpid ()
	SetTimer ()	alarm ()
	Sleep	sleep ()
• Communication	CreatePipe ()	pipe ()
	CreateFileMapping ()	shmget ()
	MapViewOfFile ()	mmap ()

16 What happens when you turn on your computer.

① Power on

On → power supply
- mother board
- Harddisk storage

② CPU loads BIOS or UEFI
- CPU init
- goes to a chip BIOS it's first default memory

③ BIOS or UEFI run tests & init hardware
- loads some settings from a memory area. Backed by CMOS battery
- program loads with settings. it tests itself that all hardware are working correctly if it not then show error or warning
- POST → power on self test

④ BIOS or UEFI hands off to boot device
- Boot devices → disk (HDD or SSD)
  CD
  USB device
- Boot loader program execute on actual OS

• BIOS → MBR master boot record
  - start from 0th index of disc
  used in old system

• UEFI → EFI extensible firmware interface
  partition in disk
  boot loader

Page No.
Date

⑤ Boot loader loads the full OS
program → actual OS initialize

windows → bootmgr.exe
mac → boot.efi
linux → CRUB

1) PC on

2) CPU initializes itself and looks for a firmware program (BIOS) stored in BIOS chip (basic input-output system chip is a ROM chip found on mother board that allows to access & setup computer system at most basic level)
- In modern PC's CPU loads UEFI (unified extensible firmware interface)

3) BIOS loads configuration settings

4) CPU runs the BIOS which tests and initializes system hardware. if something is not appropriate (like missing RAM) error is thrown and boot process is stopped
- This is called POST (power on self test) process
- UEFI can do a lot more than just initialize hardware. it's really a tiny operating system.
- Eg - Intel CPU's have the Intel Management Engine
this provides a variety of features, including powering Intel's Active Management Technology, which allows for remote management of business PC's

5) BIOS will handoff responsibility for booting your PC to your OS's bootloader
- BIOS looked at the MBR (master boot loader), a special boot sector at the begining of a disk. the MBR contains code that loads the rest of the operating system, known as a "Bootloader" The BIOS executes the

bootloader, which takes it from there and begins booting the actual operating system - windows or Linux for example

- in other words, the BIOS or UEFI examines a storage device on your system to look for a small problem, either in the MBR or on an EFI system partition, and runs it

6) the bootloader is a small program that has the large task of booting the rest of the operating system. windows uses a bootloader named windows boot manager (Bootmgr.exe), most Linux system use GRUB, and mac uses something called boot.efi

L7

32 bit vs 64 bit

32 bit =

0	0	0	0
1	2	3	4
FF	FF	FF	FF

2^{32}

64 bit =

0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8
FF	FF	FF	FF	FF	FF	FF	FF

2^{64}

64 >= 32

- 32 bit process 1 instruction cycle → 32 bit instruction 1 data process
- 64 bit process 1 instruction cycle → 64 bit instruction 1 data process

1) A 32 bit OS has 32-bit registers, and it can access 2^{32} unique memory address, i.e. 4GB of physical memory
2) A 64 bit OS has 64-bit registers, and it can access 2^{64} unique memory address, i.e. 17,179,869,184 GB of physical memory
3) 32-bit CPU architecture can process 32 bit of data and information
4) 64-bit CPU architecture can process 64 bit of data and information
5) Advantages of 64-bit over the 32-bit operating system

a) Addressable memory – 32-bit CPU 2^{32} memory addresses,
64-bit CPU 2^{64} memory addresses

b) Resource usage – installing more RAM on system with 32-bit OS doesn't impact performance, however, upgrade the system with excess RAM to the 64-bit version of windows, and you'll notice a difference

c) Performance – all calculations take place in the registers, when you'r performing math in your code, operands are loaded from memory into registers, so having larger registers allow you to perform larger calculation at the same time
32-bit processor can execute 4 byte of data in 1 instruction cycle while
64-bit processor can execute 8 byte of data in 1 instruction cycle

Page No
Date

- compatibility - 64-bit CPU can run both 32-bit and 64-bit OS while 32 bit CPU can only run 32-bit OS
- Better graphic performance - 8 bytes graphics calculations make graphic-intensive apps run faster.

Storage devices basics

- what are the different memory present in the computer system.

Register
Cach
Main memory
} primary memory

Electronic disk
Magnetic disk
Optical disk
Magnetic tapes
Hard disk
} secondary memory

1) register - smallest unit of storage, it is part of CPU itself. a register may hold an instruction, a storage address, or any data such as bit sequence or individual characters
   - register are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU
2) cache - additional memory system that temporarily stores frequently used instructions and data for quicker processing by the CPU
3) Main memory - RAM
4) Secondary memory - storage media, on which computer can store data and programs

- comparing primary memory with secondary memory

1) Cost
- primary storage are costly
- registers are most expensive due to expensive semiconductors and labour
- secondary storage are chepaper than primary

2) Access speed
- primary has higher access speed than secondary memory
- registers has highest access speed, than comes cache, then main memory

3) Storage Size
- secondary has more space

4) volatility
- primary memory is volatile
- secondary is non-volatile

nce

Introduction to process

- what is program? → compiled code. that is ready to execute.
- what is process? → program under execution
- how OS creates a process → converting program into a process

- Steps to create a process
1) load the program and static data into memory
2) allocate runtime stack
3) heap memory allocation
4) I/O tasks
5) OS handoffs control to main().

- Architure of process

stack	local variables, function arguments and return values
heap	dynamic allocated variables
data	global and static data
text	compiled code (loaded from disk)

- Attributes of process
- feature that allows identifying a process uniquely
- process table
  1. all processes are being tracked by OS using a table like data structure
  2. each entry in that table is process control block (PCB)
- PCB - stores info/attributes of a process
  1. data structure used for each process. that stores information of a process such as process id. program counter. process state, priority etc.

- PCB structure

process ID	unique identifier
Program counter (PC)	next instruction address of the program
process state	stores process state
priority	based on priority a process gets CPU time
registers	
list of open files	
list of open devices	

- Registers in the PCB. it is a data structure. when a process is running and it's time slice expires. the current value of process specific registers would be stored in the PCB and the process would be swapped out. when the process is scheduled to be run, the register value is read from the PCB and written to the CPU registers. this is the main pursose of the registers.

lure

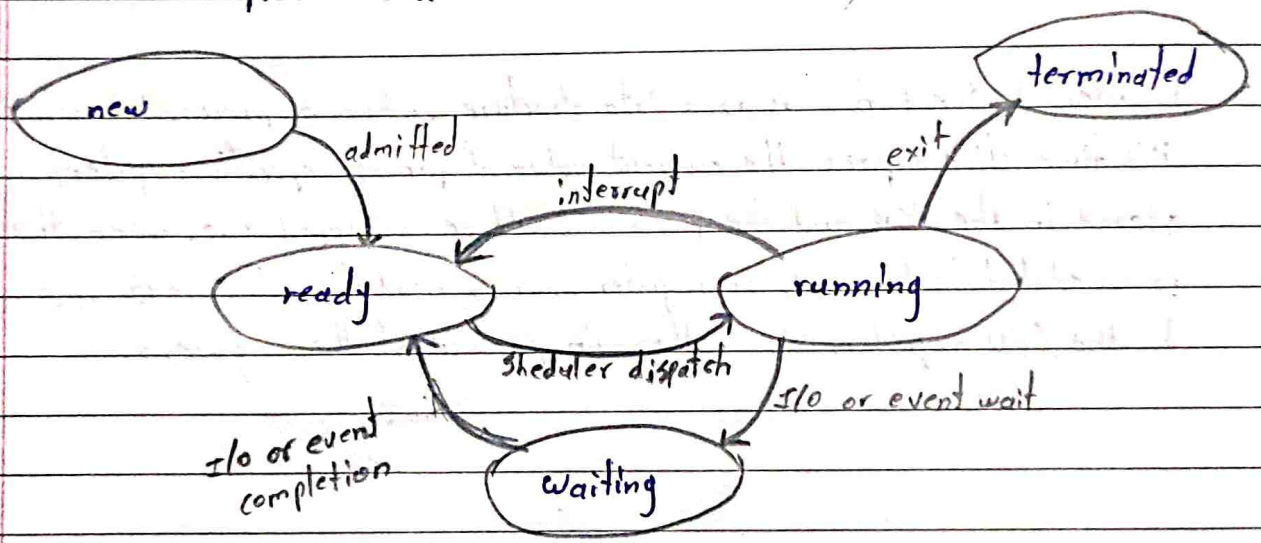
L10

Process states , Process queues

Page No.
Date

- Process states . as process executes , it changes state , each process may be in one of the following states

1) New - OS is about to pick program and convert it into processes , or the process is being created
2) Run - instructions are being executed . CPU is allocated
3) Waiting - waiting for I/O
4) Ready - the process is in memory . waiting to be assigned to a processor
5) Terminated - the process has finished execution . PCB entry removed from process table



- Process queues

1) Job queues
   i) processes in new state
   ii) present in secondary memory
   iii) job shedular LTS - long term schedular picks process from the pool and loads them into memory for execution

2) Redy queue
   i) process in ready state
   ii) present in main memory
   iii) CPU schedular STS - short term schedular picks process from ready queue and dispatch it to CPU

Page No.
Date

=> Waiting queue
:> process in wait state

- Degree of multi-programming - the number of processes in the memory
  - LTS controls degree of multi-programming

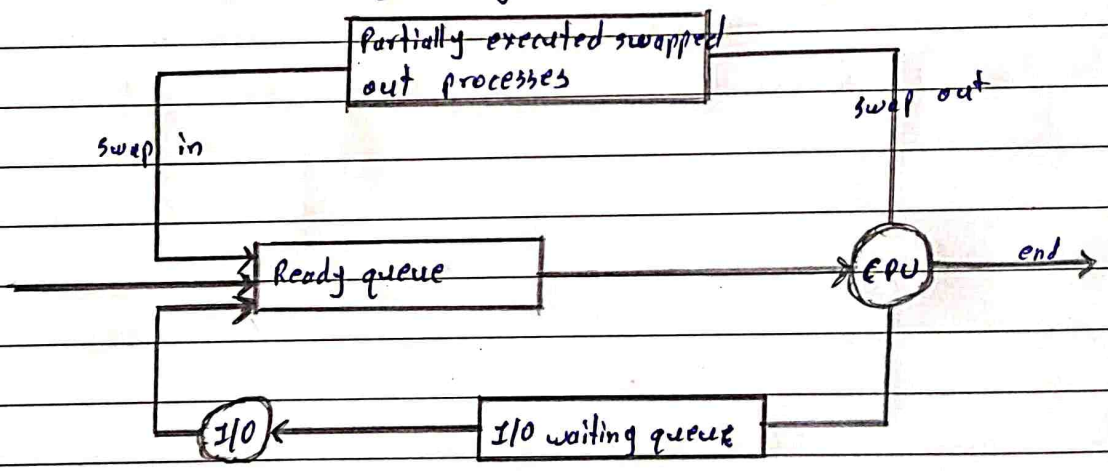
- Dispatcher - the module of OS that gives control of CPU to a process selected by STS

L11

Swapping / context-switching / Orphan process / Zombie process

Swapping -

- Time-sharing system may have medium term scheduler (MTS)
- remove processes from memory to reduce degree of multi-programming
- these removed process can be reintroduced into memory, and it's execution can be continued where it left off, this is called swapping
- swap-out and swap-in is done by MTS
- swapping is necessary to improve process mix or because a change in memory requirements has overcommited available memory, requiring memory to be freed up.
- swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make the memory available to other processes. at some later time, the system swaps back the process from the secondary storage to main memory



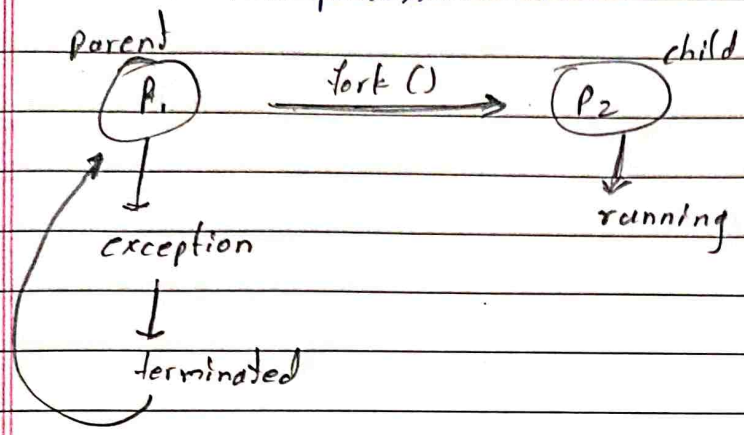
Context-Switching

- switching the CPU to another process requires performing a state of the current process and a state restore of a different process.
- when this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run
- it is pure overhead, because the system does no useful work while switching
- speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

Page No.
Date

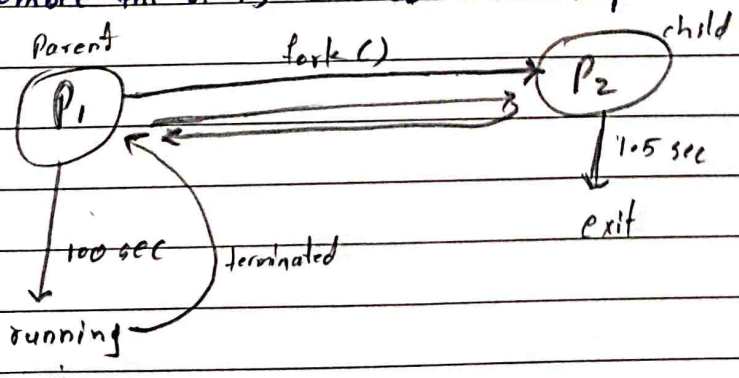
☐ orphan process

- the process whos parent process has been terminated and it is still running
- orphan process is adopted by init process
- init is the first process of OS



☐ Zombie process / Defunct process

- a zombie process is a process whose execution is completed but it still has an entry in the process table
- zombie process usually occur for child process, as the parent process still needs to read its child's exit status. once it is done using the wait system call, the zombie process is eliminated from the process table. this is known as reaping the zombie proce
- it is because parent process may call wait() on child process for a longer time duration and child process got terminated much earlier.
- as entry in the process table can only be removed, after the parent process reads the exit status of child process. hence. the child process remains a zombie till it is removed from the process table



L12 Intro to process scheduling / FCFS / convoy effect

Page No.
Date

Process scheduling
- basis of multi-programming OS
- by switching the CPU among processes, that OS can make the computer more productive
- many processes are kept in memory at a time. when process must wait or time quantom expires, the OS takes the CPU away from that process & gives the CPU to another process & this pattern continues

CPU scheduler
- whenever CPU become ideal, OS must select the process from the ready queue to be executed
- done by STS

Non-Preemptive scheduling
- terminate
- wait

- once CPU has been alocated to a process, the process keeps the CPU untill it releases CPU either by terminating or by switching to wait state

Preemptive scheduling
- terminate
- wait
- quantom

- CPU is taken away from a process after time quantom expires along with terminating or switching to wait-state
- less starvation
- high CPU utilization

Goals of CPU scheduling
- maximum CPU utilization
- minimum turnaround time (TAT)
- minimum wait time
- minimum response time
- maximum throughput of system.

Page No.
Date

* Throughput - No. of processes completed per unit time
* Arrival time (AT) - time when process is arrived at the ready queue
* Burst time (BT) - the time required for the process for it's execution
* Turn Around time (TAT) - time taken from first time process enters ready state till it terminates (CT - AT)
* Wait time (WT) - time process spends waiting for CPU (TAT - BT)
* Response time - time duration between process getting into ready queue and processes getting CPU for the first time
* Completion time (CT) - time taken till process gets terminated

FCFS - first come first serve

- whichever process comes first in the ready queue will be given CPU first.
- in this, if one process has longer BT, it will have major effect on average WT of different processes called convoy effect
- convoy effect is a situation where many processes, who need to use a resource for a short time, are blocked by one process holding that resource for a long time
- this can poor resource management

Convoy effect

P.no	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
P_1	0	20	20	20	0
P_2	1	2	22	21	19
P_3	2	2	24	22	20

Avg. WT = 13

P_1	P_2	P_3

t = 0, 20, 22, 24

P.no	AT	BT	CT	TAT	WT
P_2	0	2	2	0	0
P_3	1	2	4	3	1
P_1	2	20	24	22	2

Avg. WT = 1

P_2	P_3	P_1

t = 0, 2, 4, 24

drawback

L13

CPU Scheduling / SJF / Priority / RR

Page No.
Date

Shortest job first (SJF) - non-preemptive

- process with least BT will be dispatched to CPU first
- must do estimation for BT for each process in ready queue beforehand, correct estimation of BT is an impossible task (ideally)
- run lowest time process for all time then, choose job having lowest BT at that instance
- this will suffer from convoy effect as if the very first process which came is ready state is having a large BT
- process starvation might happen
- criteria for SJF algos, AT + BT

P	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
P_1	0	8	8	8	0
P_2	1	4	12	11	7
P_3	2	9	26	24	15
P_4	3	5	17	14	9

Avg. WT = 7.75

P_1	P_2	P_4	P_3

t=0, 8, 12, 17, 26

SJF - premetive

- less starvation
- no convoy effect
- gives average WT less for a given set of processes as scheduling short job before a long one decreases the WT of short job more than it increases the WT of the long process

P	AT	BT	CT	TAT	WT
P_1	0	~~8~~ 7	17	17	9
P_2	1	4	5	4	0
P_3	2	9	26	24	15
P_4	3	5	10	7	2
					6.5

P_1	P_2	P_4	P_1	P_3

t=0, 1, 5, 10, 17, 26

Page No
Date

* Priority scheduling - non-preemptive
- priority is assigned to a process when it is created
- SJF is a special case of general priority scheduling with priority inversely proportional to BT

P	Priority	AT	BT	CT	TAT	WT
P_1	2	0	4	4	4	0
P_2	4	1	2	25	24	22
P_3	6	2	3	25	22	19
P_4	10	3	5	9	6	1
P_5	8	4	1	20	16	15
P_6	12	5	4	13	8	4
P_7	9	6	6	19	13	7
						9.71

P_1	P_4	P_6	P_7	P_5	P_3	P_2

t=0 4 9 13 19 20 23 25

* Priority scheduling - preemptive
- current run state job will be preempted if next job has higher priority
- may cause indefinite waiting (starvation) for lower priority jobs (possibility is they won't get executed ever). true for both preemptive & non-preemptive version

solution — Ageing is the solution
- gradually increase priority of process that wait so long... e.g.- increase priority by 1 every 15 minutes

P	Priority	AT	BT
P_1	2	0	~~4~~ ~~3~~
P_2	4	1	~~2~~ ~~1~~
P_3	6	2	~~3~~ ~~2~~
P_4	10	3	~~5~~ ~~3~~
P_5	8	4	~~1~~
P_6	12	5	~~4~~
P_7	9	6	~~6~~

indefinite waiting or extream starvation
↓
drawback

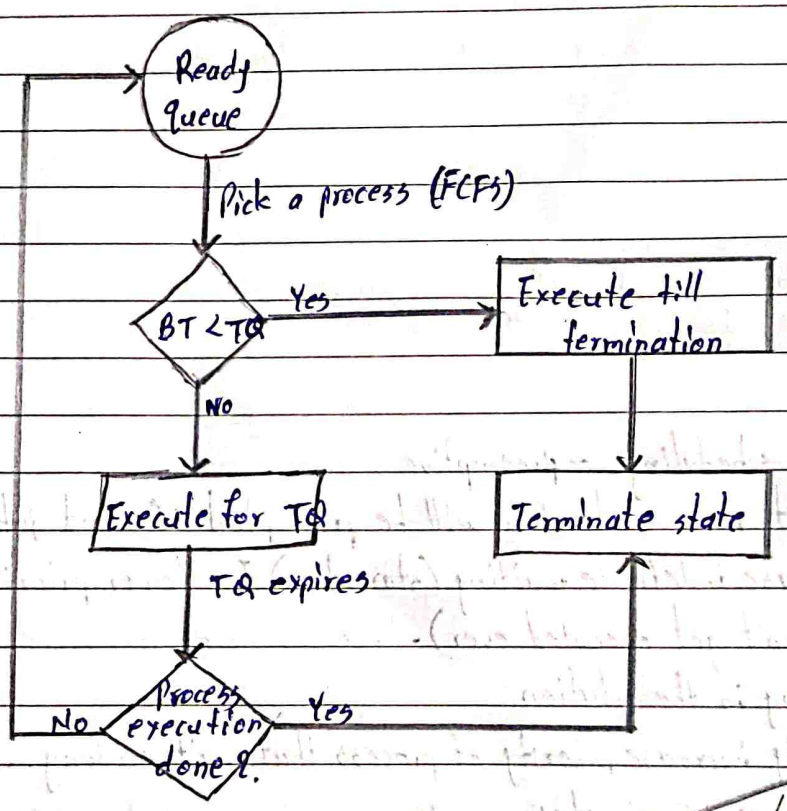
P_1	P_2	P_3	P_4	P_6	P_4	P_7	P_5	P_3	P_2	P_1

t=0 1 2 3 5 9 12 18 19 21 22 25

Page No.
Date

Round robin scheduling (RR)

- most popular
- like FCFS but preemptive
- designed for time sharing system
- criteria = AT + TQ . doesn't depend on BT -TQ- Time quantom
- no process is going to wait forever . hence very low starvation . & no convoy effect
- easy to implement
- if TQ is small . more will be the context switch (more overhead)



P	AT	BT
P_1	0	~~4~~ ~~2~~
P_2	1	~~5~~ ~~3~~ ~~1~~
P_3	2	~~2~~
P_4	3	~~1~~
P_5	4	~~6~~ ~~4~~ ~~2~~
P_6	5	~~3~~ ~~1~~

Quantom = 2 second

P_1	P_2	P_3	P_1	P_4	P_5	P_2	P_6	P_2	P_6	P_5

P_1	P_2	P_3	P_1	P_4	P_5	P_2	P_6	P_5	P_2	P_6	P_5

t=0 2 4 6 8 9 11 13 15 17 18 19 21

L14

MLQ / MLFQ

- multi-level queue scheduling
  - ready queue is divided into multiple queues depending upon priority.
  - a process is permanently assigned to one of the queues (inflexible) based on some property of process, memory size, process priority or process type
  - each queue has its own scheduling algorithm. Eg. RR → SP, IP
    FCFS → BP

ect

highest priority

→ System process (SP) queue

→ Interactive process (IP) queue

lowest priority → Batch process (SP) queue

- system process - created by OS (high priority)
- Interactive process - forground process - needs user I/O
- Batch process - background process - runs silently, no user input require

- scheduling among different sub-queues is implemented as fixed priority preemptive
  - if interactive process comes & batch process is currently executing then, batch process will be preempted

draw back
- only after complition of all the processes from the top-level ready queue, the further level ready queues will be scheduled
- this came starvation for lower priority process
- convoy effect is present.

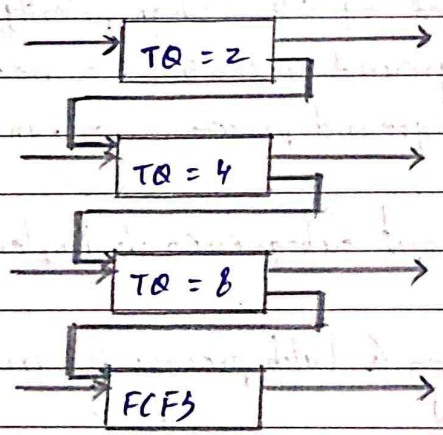
Page No.
Date

Multi-level feedback queue scheduling

- multiple sub-queues are present
- allows the process to move between queues. the idea is to separate processes according to the characteristics of their BT. if a process uses too much CPU time, it will be moved to lower priority queue. this scheme leaves I/O bound and interractive processes in the higher-priority queue.

  In addition, a process that waits too much in a lower-priority queue may be moved to a higher priority queue. this form of ageing prevents starvation.
- less starvation then MLQ
- it is flexible
- can be configured to much a specific system design requirement.

Sample MLFQ design



Comparison.

	FCFS	SJF	PSJF (Preemptive SJF)	Priority	P-priority (Preemptive priority)	RR	MLQ	MLFQ
design	simple	complex	complex	complex	complex	simple	complex	complex
preemption	no	no	yes	no	yes	yes	yes	yes
lonvoy effect	yes	yes	no	yes	yes	no	yes	yes
overhead.	no	no	yes	no	yes	yes	yes	yes

L15

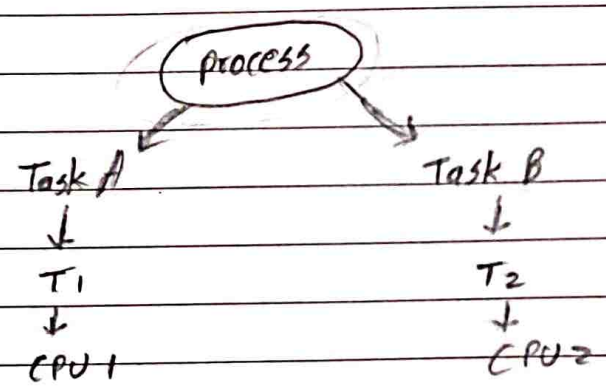
Introduction to Concurrency

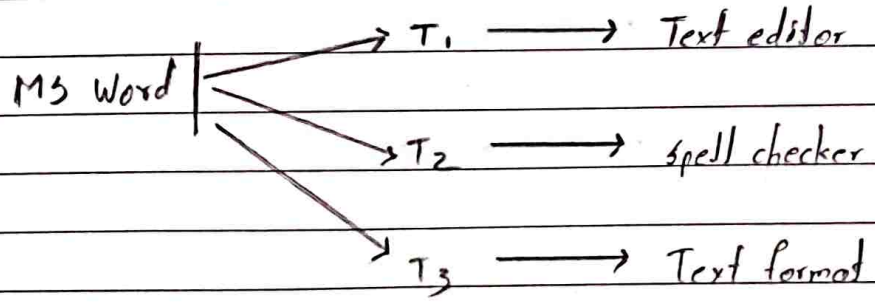
Page No.
Date

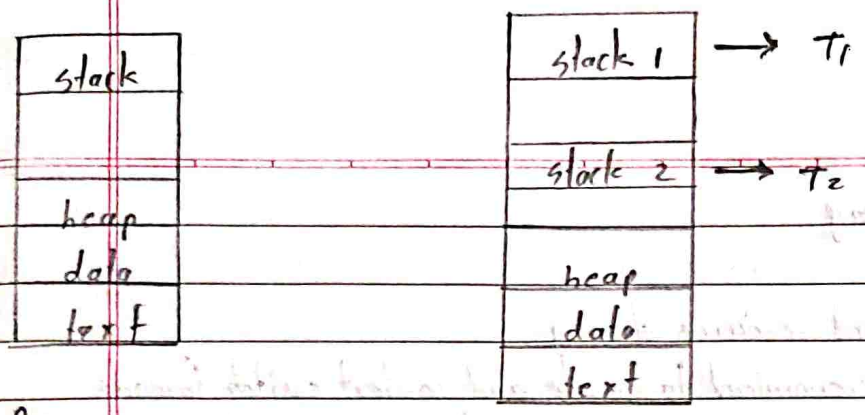
- **Concurrency** - is the execution of multiple instruction sequence at the same time. it happens in the OS when there are several process threads running in parallel

- **Thread** - • single sequence stream within a process
  • an independent path of execution in process
  • light weight process
  • used to active parallelism by dividing a process's task which are independent path of execution

  eg - • multiple tabs in browser, text editor (when you typing in an editor, spell checking, formating of text and saving the text are done concurrently by multiple threads)







Process

Threads

- Thread scheduling - threads are scheduled for execution based on their priority. even though threads are executing within the runtime, all threads are assigned processor time slices by the OS

- Threads context switching
  - OS saves current status of threads & switches to another thread of same process
  - doesn't include switching of memory address space (but PC, registers, stack switches)
  - fast switching as compare to process switching
  - CPU's cache state is preserved

- How each thread get access to CPU
  - each thread has its own program counter
  - depending upon the thread scheduling algorithm, OS schedule these threads
  - OS will fetch instruction corresponding to PC of that thread & execute instruction

- I/O or TQ based context switching is done here as well
  - We have TCB (Thread control block) like PCB for state storage management while performing context switching.

- Will single CPU system would gain by multi-threading technique?
  - never
  - as two threads have to context switch for that single CPU.
  - this won't give any gain.

Page No.
Date

Benefits of multi-threading

- responsiveness
- resource sharing - efficient resource sharing
- Economy - it is more economical to create and context switch threads
  - also allocating memory and resources for process creation is costly, so better to divide tasks into threads of same process
- Threads allow utilization of multiprocessor architecture to a greater scale and efficiency

Code for two threads.

```
# include <iostream>
# include <threads>
# include <unistd.h>
using namespace std;

int main(){
        thread t1(task A);
        thread t2(task B);
        t1.join();
        t2.join();
        return 0;
}

void taskA(){
        for(int i=0; i<10; i++){
            sleep(1);
            printf("taskA : %d \n", i);
            fflush(stdout);
        }
}
```

```
void taskB (){
        for (int i=0 ; i<10 ; i++){
            sleep(1);
            printf("TaskB : %d \n", i);
            fflush(stdout);
        }
}
```

1 → T_1
1 → T_2
2 → T_1
2 → T_2
3 → T_1
3 → T_2
4 → T_1
4 → T_2

L16

Critical section problem and how to address it

- process synchronization technique play a key role in maintaining the consistency of shared data.

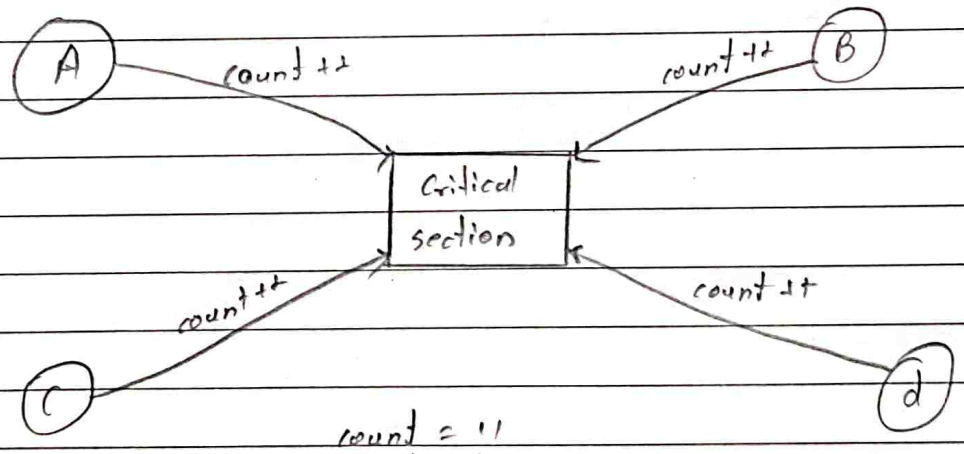
Critical section

- the critical section refers to the segment of code where processes / threads access shared resources, such as common variable and files, and perform write operations on them. since processes / threads execute concurrently, any process can be interrupted mid-execution.

Major thread scheduling issue

a) Race condition

a race condition occurs when two or more threads can access shared data and they try to change it at the same time. because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared ~~memory~~ data. Therefore, the result of the change in data is dependent on the shared scheduling algorithm, i.e both threads are "racing" to access / change the data.



count = 11
count ++
↳ temp = count + 1
temp = 2
count = temp

when temp = count + 1 that time if thread is switched the count will can't increase that happens when more threads performs operation.

Page No.
Date

Q Single flag ?

T_1	T_2
while(1){	while(1){
while(turn != 0);	while(turn != 1);
C.S	C.S
turn = 1	turn = 0
R.S	R.S
}	}

turn = 0 → T_1 → T_2

turn = 1 → T_2 → T_1

it's not suitable because it depending of state of turn

☐ Solution to race condition

a. Atomic operations : make critical code section an atomic operation, i.e. executed in one CPU cycle

b. mutual exclusion using locks.

c. semaphores

★ Can we use a simple flag variable to solve the problem of race condition ?

→ no

☐ Peterson's solution can be used to avoid condition but holds good for only 2 process/threads

flag[2]
turn

flag[2] → indicate if a thread is ready to enter the C.S,
flag[i] = true implies that i is ready

turn → indicates whoes turn is to enter the C.S

Page No.
Date

T_1

```
while(1){
    flag[0] = T
    turn = 1
    while(turn==1 && flag[1] = T);
        C.S
        flag[0] = F
}
```

T_2

```
while(1){
    flag[1] = T
    turn = 0
    while(turn==0 && flag[0] = T);
        C.S
        flag[1] = F
}
```

- mutex / locks
  - locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread / process to access critical section

- Disadvantages -
- Contention - one thread has acquired the lock, other threads will be busy waiting, what if thread that ackwired the lock dies, then all other threads will be in infinite waiting
- Deadlocks - • debugging
  • starvation of high priority threads

L17 Conditional variable and semaphores for Threads synchronization

Page No.

Conditional variable

- the conditional variable is a synchronization primitive that lets the thread wait until a certain condition occurs
- works with a lock
- thread can enter a wait state only when it has acquired a lock. when thread enters the wait state, it will release the lock and wait untill another thread notifies that the event has occured. once the waiting thread enters the running state, it again acquires the lock immidiately and starts executing.
- why to use conditional variable ?
→ to avoid busy waiting
- contention is not here

Semaphores

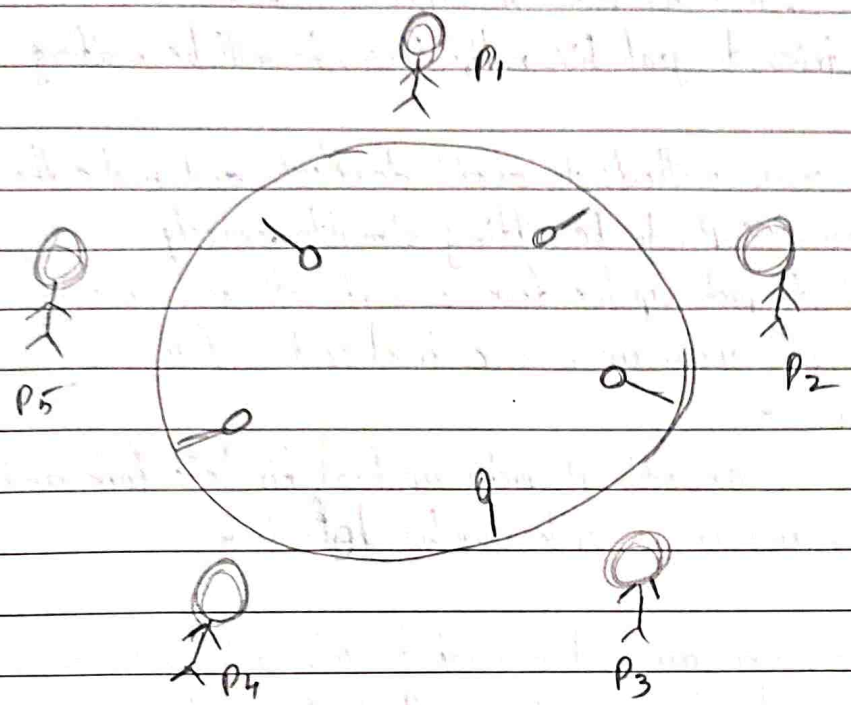
- synchronization method
- an integer that is equal to number of resources
- multiple threads can go and execute C.S. concurrently
- allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource one at a time
- binary semaphore value can be 0 or 1
  - Aka, mutex locks

- counting semaphore
  - can range over an unristricted domain
  - can be used to control access to a given resource consisting of a finite number of instances

Page No.
Date

- to overcome the need for busy waiting, we can modify the defination of the wait() and signal() semaphore operations. when a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. the block-operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. then control is transfered to the CPU scheduler, which selects another process to execute
- a process that is blocked, waiting on a semaphore S, should berestarted when some other process executes a signal() operation. the process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. the process is then placed in the ready queue

The dining philosophers problem.



- we have 5 philosophers
- they sit on circular table surrounded by 5 chairs (1 each), in the center of table is a bowl of noodels, and the table is a loaded with 5 single forks
- Thinking state → when a P. gets hungry, he try's to pick up 2 forks adjacent to him he can pick 1 fork at a time.
- one can't pick up a fork if it is already taken
- when P. has both forks at the same time, he eats without releasing forks

→ solution can be given using semaphore
- each fork is a binary semaphores
- a P. calls wait() operation to acquire a fork
- release fork by calling signal()
- semaphore fork [5] {1}

- although the semaphore solution makes sure that no two neighbors are eating simultaneously but it culd still create deadlock.

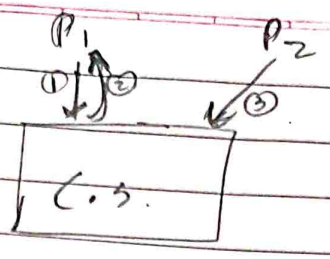
- suppose that all 5 p. become hungry at the same time and each picks up their left fork, then all fork semaphores would be 0
- when each p. tries to grab his right fork, he will be waiting for ever (deadlock)

- we must use some methods to avoid deadlock and make the solution work
  - allow at most 4 p. to be sitting simultaneously
  - allow a p. to pick up his fork only if both forks are available and to do this, he must pick them up in a critical section (atomically)
  - Odd-even rule -

    an odd p. picks up first his left fork and then his right fork, whereas an even p. picks up his left fork

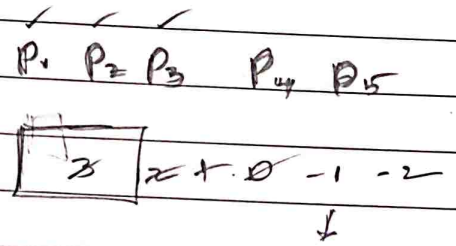
- hence only semaphore are not enough to solve this problem we must add some enhancement rules to make deadlock free solution.

mutual exclusion means when P_1 get access to C.S. it locks it till it redeases resources, when P_1 get completed then it anlock the resources



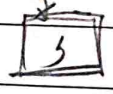
- Semaphore
  1) Counting semaphore
  2) binary semaphore.

* Counting semaphore
- counting semaphore means predefined n numbers of process in C.S at a time.



when any running Process get executed and terminated then P_4 get access try once more and get the access of C.S

Process



going to in then run down code

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

```
Up(semaphore){
    S.value = S.value + 1;
    if(S.value <= 0){
    // select a process from suspended
    // list, wake();
    }
}
```

L21

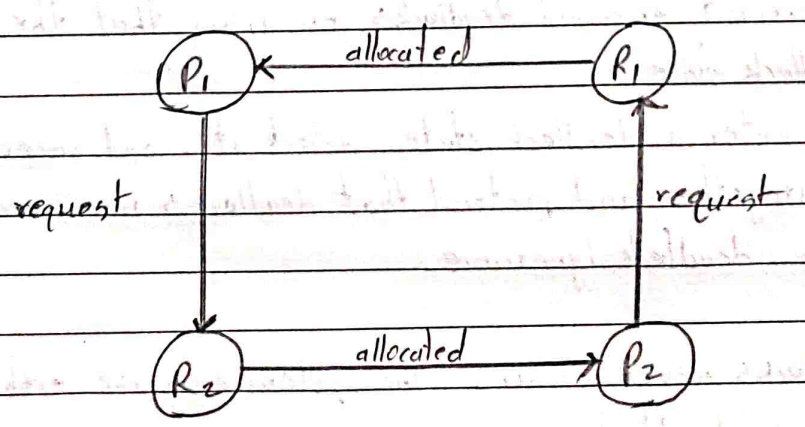
Deadlock

- In multiprogramming environment, we have several processes competing for finite number resources
- process requests a resource (R), if R is not available (taken by other process), process entern in a waiting state, some time that waiting process is never able to change it's state because the resource it has requested is busy (forever), called deadlock (DL)
- two or more processes are waiting on some resource's availability, which will never be available as it is also busy with some other process. the processes are said to be in deadlock
- DL is a bug present in the process / thread synchronization method
- in DL, processes never finish executing and the system resources are tied up, preventing other job from starting
- resources Ex → memory space, CPU cycles, files, locks, sockets, I/O devices. etc.

- single resource can have multiple instructions of that Eg → CPU is resource, and system can have 2 CPUs

■ How a process / thread utilize resource ?

1) Request request the R, if R is free lock it, else wait till it is available
2) Use
3) Release release resource instance and make it available for other processes



Page No.
Date

Deadlock necessary condition — 4 conditions should hold simultaneously

① mutual exclusion -
- only one process at a time can use the resource, if another process requests that resource, the requesting process must wait untill the resource has been released

② Held and wait
- a process must be holding at least one resource and waiting to aquire additional resources that are currently being held by other process.

③ no-preemption
- resource must be voluntarily released by the process after complition of execution - (no resource preemption)

④ circular wait
- a set ($P_0, P_1 \ldots P_n$) of waiting process must exists such that P_0 is waiting for a resource held by P_1, P_1 is waiting for resource held by P_2 and so on.

methods for handling deadlocks

- use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state
- Allow the system to enter a deadlock state, detect it, and recover
- ignore the problem altogether and pretend that deadlocks never occure in system (ostrich algorithm) aka, deadlock ignorance

To ensure that deadlocks never occurs, the system can use either a deadlock prevention or deadlock avoidance scheme.

Page No.
Date

Deadlock Prevention

by ensuring at least one of the necessary conditions cannot hold

① mutual exclusion -

- use locks (mutual exclusion) only for non-sharable resources
- sharable resources like read-only files can be accessed by multiple processes / threads
- however, we can't prevent DL's by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable

② hold and wait

- to ensure H&W condition never occurs in the system, we must guarantee that, whenever process requests a resource, it doesn't hold any other resource

* A - protocol (A) can be, each process has to request and be allocated all it's resources before it's execution.

* B - protocol (B) can be, allow a process to request resources only when it has none, it can request any additional resources after it must have released all the resources that it is currently allocated

③ no-preemption

- if a process is holding some resources and request another resource that cannot be immediately allocated to it, then all the resources the process is currently holding on are preempted, the process will restart only when it can regain it's old resources, as well as the new one that it is requesting (live lock may occure)
- if a process requests some resources, we first check whether they are available, if yes, we allocate them. if not, we check whether they are allocated to some other process that is waiting for additional resources. if so, preempt the desired resource from waiting process and allocate them to the requesting process.

④ Circular wait

- to ensure that this condition never holds is to impose a proper ordering of resource allocation
- P_1 & P_2 both require R_1 & R_2, locking on these resources should be like, both try to lock R_2 & R_1 the it is circular state

$P_1 \rightarrow R_1$ $\quad$ $P_2 \rightarrow R_2$

L22 Deadlock avoidance

Page No.
Date

idea is, the kernel be given in advance info concerning which resources will use in its lifetime.

- by this, system can decide for each request whether the process should wait to decide whether the current request can be satisfied or delayed, the system must consider the resources currently available, resources currently allocated to each process in the system and the future requests and releases of each process

- schedule process and it's resource allocation in such a way that the DL never occurs
- safe state : a state is safe if the system can allocate resources to request ing process up to max process in some order and still avoid DL
- a system is in safe state only if there exists a safe sequence
- in an useful state, the operating system cannot prevent process from requesti ng resource in such a way that any deadlock occurs, it is not necessary that all unsafe states are deadlocks, an unsafe state may lead to deadlock
- the main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the resulting state is safe state
- in a case, if the system is unable to fulfill the request of all process, then the state of the system is called unsafe.
- scheduling algorithm using which DL can be avoided by finding safe state (banker's algorithm)

☑ Banker algorithm

- when a process request a set of resources, the system must determine whether allocating these resources will leave the system in safe state, if yes, then the resources may be allocated to the process. if no, then the process must wait till other process release enough resources

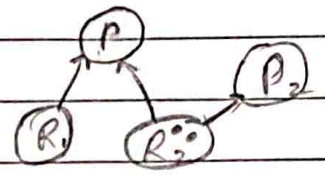
Deadlock Detection

- systems haven't implemented deadlock-prevention or a deadlock avoidance technique, then they may employ DL detection then, recovery technique.

* Single instance of each resource type wait-for graph method
- a deadlock exists in the system if and only if there is a cycle in the wait-for graph. in order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the wait-for graph

wait for graph is nothing but block representation of

* multiple instances for each resource type resource and process
- banker algorithm.



Recovery from deadlock

* Process termination
  - abort all DL processes
  - abort one process at a time until DL cycle is eliminated

* Resource preemption
  - to eleminate deadlock, we sucessively preempt some resources from process and give these resources to other processes untill DL cycle is broken.

Bankers Algorithm

deadlock avoidance
deadlock detection

Total A = 10
B = 5
C = 7

process	Allocation A	Allocation B	Allocation C	Max need A	Max need B	Max need C	Available A	Available B	Available C	Remaining need A	Remaining need B	Remaining need C
P_1	0	1	0	7	5	3	3	3	2	7	4	3
P_2	2	0	0	3	2	2	5	3	2	1	2	2
P_3	3	0	2	9	0	2	10	5	7	6	0	0
P_4	2	1	1	4	2	2	7	4	3	2	1	1
P_5	0	0	2	5	3	3	7	4	5	5	3	1

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

L24 Memory management techniques | Contiguous memory allocation

1. In multi-programming environment, we have multiple processes in the main memory (Ready Queue) to keep the CPU utilization high and to make computer responsive to users.
2. to realize this increase in performance, however, we must keep several processes in the memory, this is, we must share the main memory. as a result, we must manage main memory for all different processes.

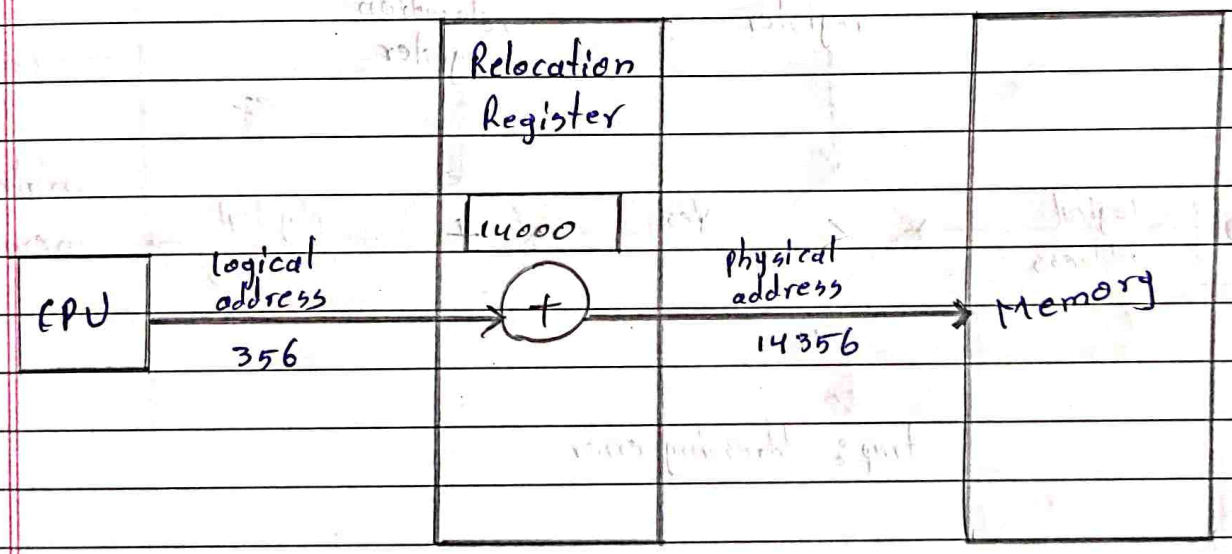
3. Logical versus Physical Address Space

* Logical Address
- an address generated by CPU
- the logical address is basically the address of instruction or data used by a process.
- user can access logical address of the process.
- user has indirect access to the physical address through logical address
- logical address does not exist physically. Hence, aka. virtual address.
- the set of all logical addresses that are generated by any program is referred to as logical address space.
- Range → 0 to max

Page No.
Date

* Physical Address
- an address loaded into the memory-address register of the physical memory
- user can never access the physical address of the program
- the physical address is the memory unit. it's a location in the main memory physically
- A physical address can be accessed by a user indirectly but not directly.
- the set of all physical addresses corresponding to the logical addresses is commonly known as physical address space.
- it is computed by the memory management Unit (MMU)
- Range → (R+0) to (R+max). for a base value R

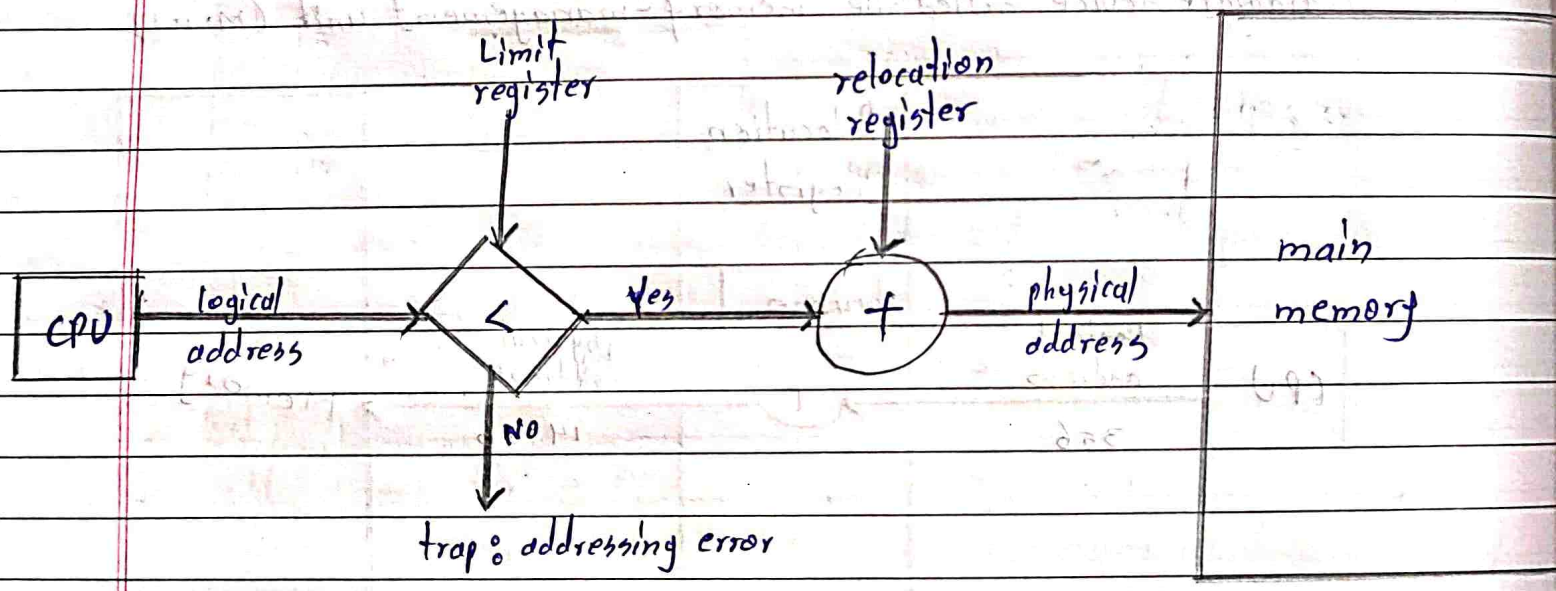
- The runtime maping from virtual to physical address is done by a hardware device called the memory-management unit (MMU)



- How OS manages the isolation and protect (memory maping & protection)
- OS provides this Virtual address Space (VAS) concept
- to seperate memory space, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses
- the reallocation register contains value of smallest physical address (base address (RJ). the limit register contains the range of logical addresses (e.g, relocation = 100040 & limit = 74600)

- each logical address must be less than the limit register.
- MMU maps the logical address dynamically by adding the value in the relocation register
- when CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit register with the correct values as part of the context switch. since every address generated by the programs and data ↳ the CPU (logical address) is the from being modified by running process
- any attempt by a program executing in user mode to access the OS memory or other uses' memory results in a trap in the OS, which treat the attempt as a fatal error

Address Translations



Allocation method on physical memory

- contiguous allocation
- Non-contiguous allocation

Contiguous memory Allocation

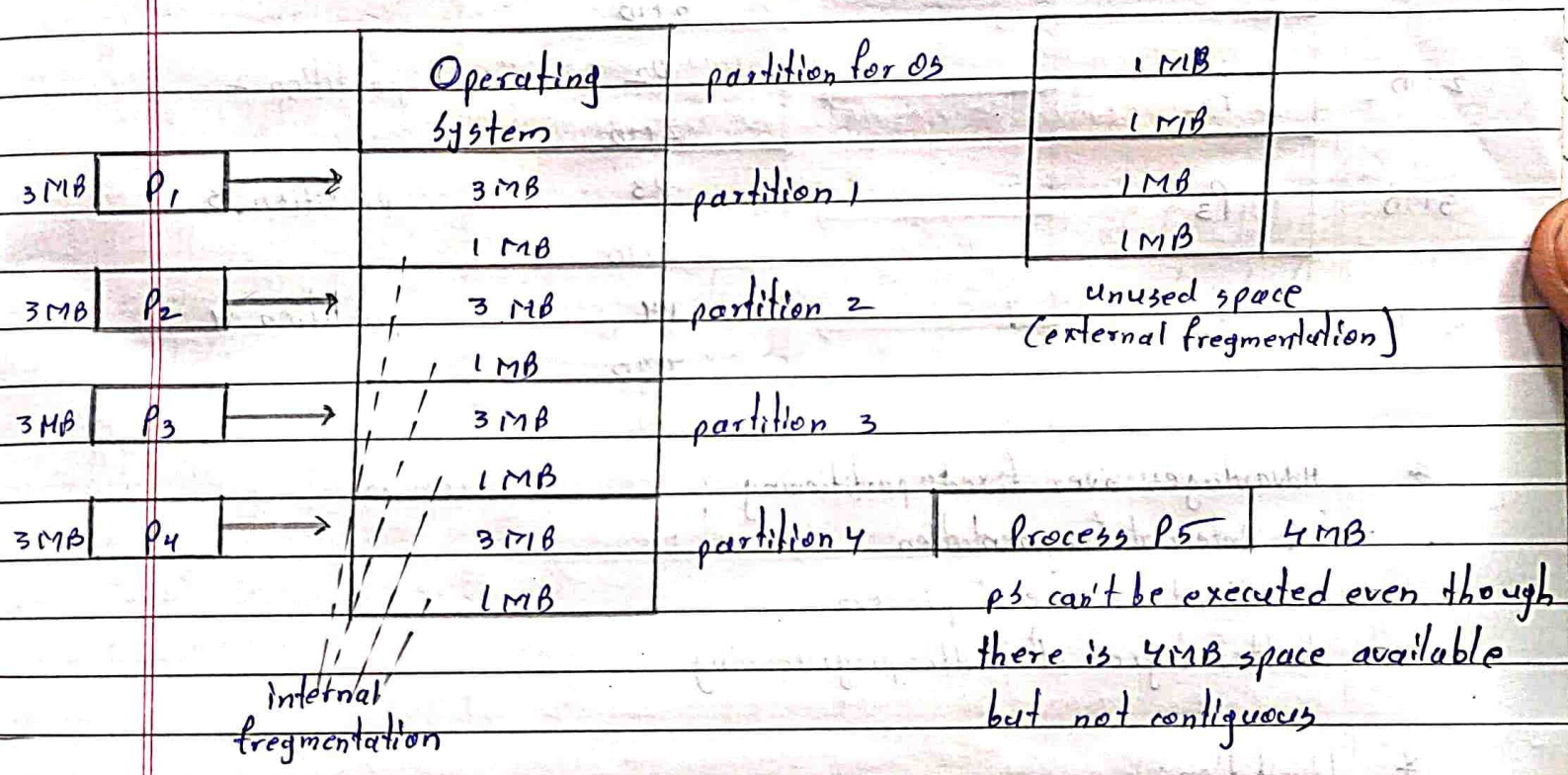
- in this scheme, each process in contained in single contiguous block of memory.

Page No.
Date

① Fixed Partitioning

- the main memory is divided into partitions of equal or different size

ed against these registers, we can protect both OS and other users



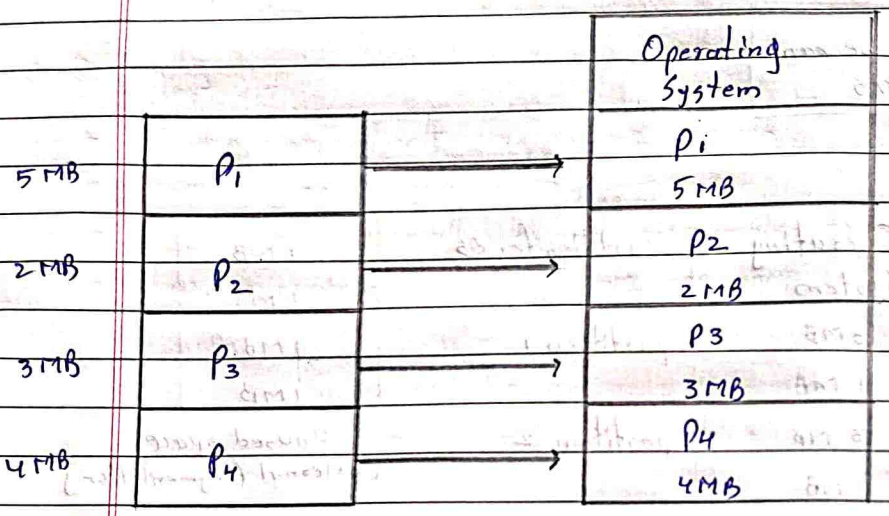
Limitations

- Internal fregmentation
- external fregmentation
- Limitation on process size → if the process size is larger than the size of maximum sized partition than that process cannot be loaded into the memory. therefore, a limitation can be imposed on the process size that it is cannot be larger than the size of the largest partition
- low degree of multiprogramming → in fixed partitioning the degree of multiprogramming is fixed and very less because the size of the partition

cannot be varied according to the size of process.

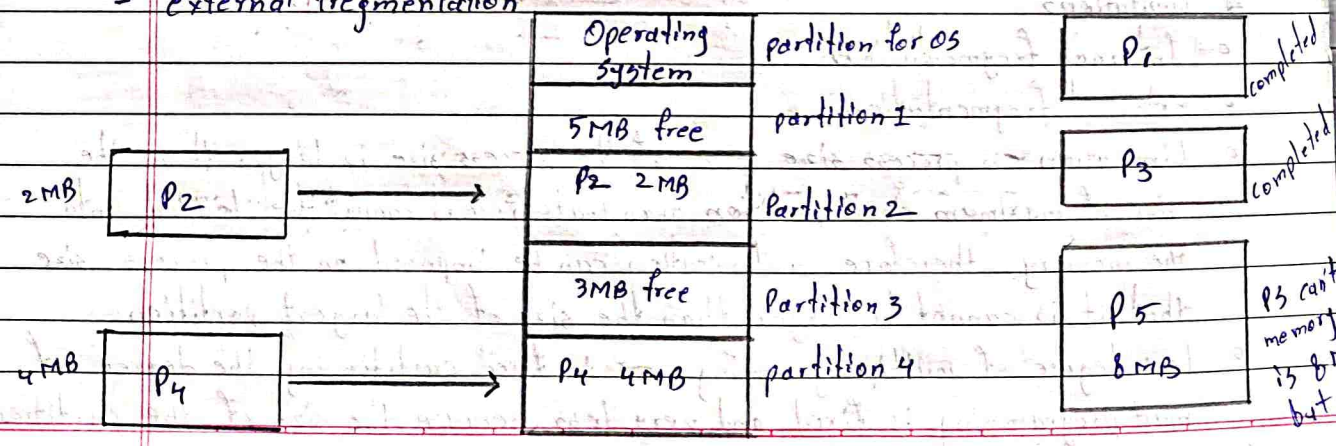
Dynamic Partitioning

- in this technique. the partition size is not declared initially. it is delared at the time of process loading



* Advantages over fixed partitioning
  - no internal fregmentation
  - no limit on size of process
  - better degree of multi-programming

* Limitations
  - external fregmentation



L 25

Free Space management.

Page No.
Date

Defregmentation / compaction

- dynamic partitioning suffers from external fragmentation
- compaction to minimize the probability of external fragmentation
- all the free partitions are made contiguous, and all the loaded partitions are brought together
- by applying this technique, we can store the bigger process in the memory. the free partitions are merged which can now be allocated according to the needs of new process. this technique is also called defregmentation.
- the efficiency of the system is decresed in the case of compaction since all the free spaces will be transferred from several places to a single place.

How free space is stored / represented in OS ?

- free holes in the memory are represented by a free list (linked-list data structure)

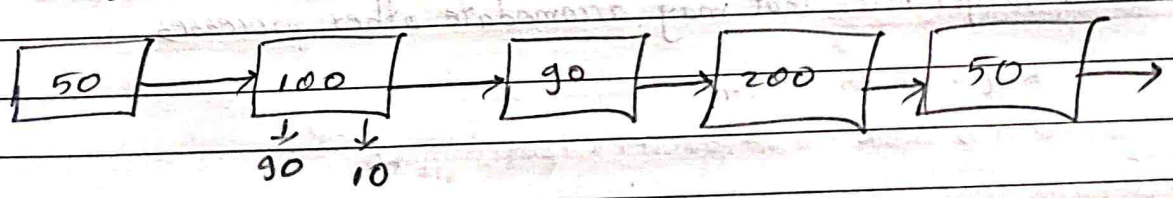
How to satisfy a request of a of n size from a list of free holes ?

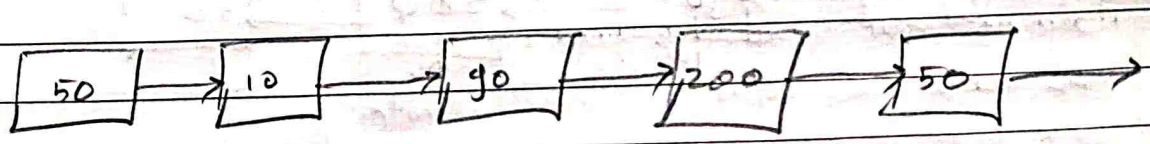
- various algorithm which are implemented by the OS in order to find out the holes in the linked list and allocate them to the processes

① first fit

- allocate the first hole that is big enough
- simple and easy to implement
- fast / less time complexity

Req - 90kB - P1



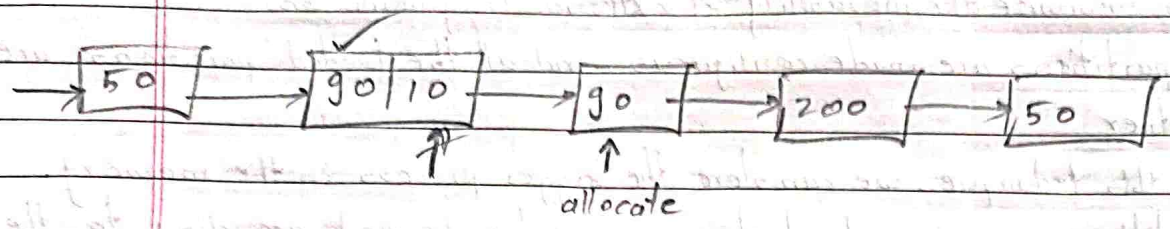


P1 be loaded into even though there is no space available not contiguous

② Next fit

- enhancement on first fit but starts search always from last allocated hole
- same advantages of first fit

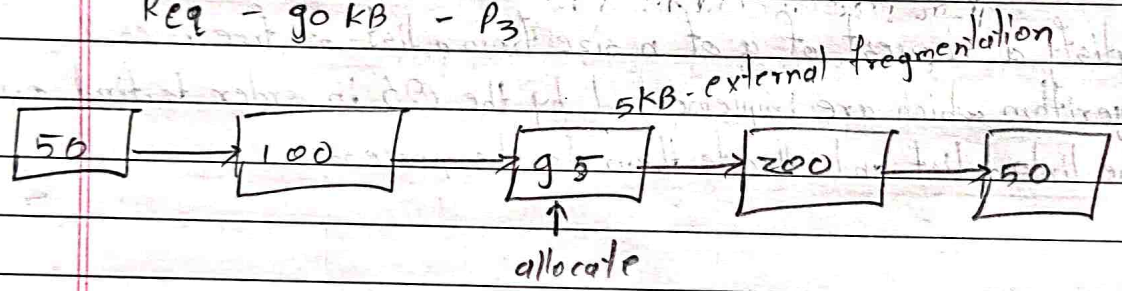
Req - 90 KB - P_2



③ Best fit

- allocate smallest hole that is big enough
- lesser internal fragmentation
- may create many small holes and cause major external fragmentation
- slow, as required to iterate whole free holes list

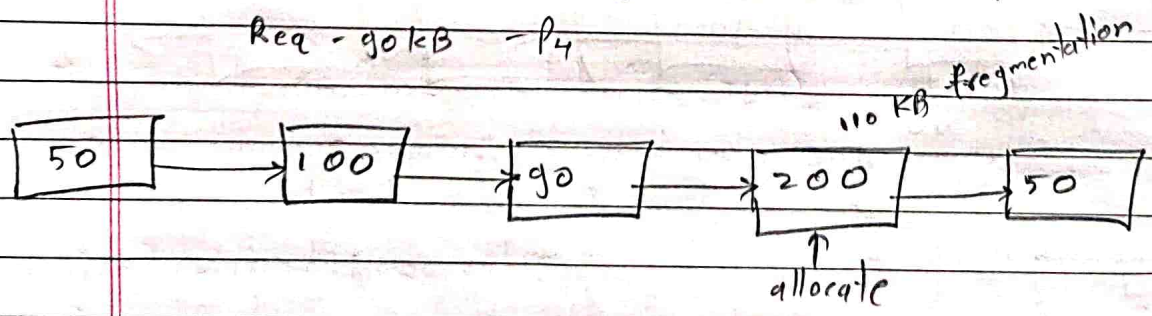
Req - 90 KB - P_3



④ Worst fit

- allocate the largest hole that is big enough
- slow, as required to iterate whole free holes list
- leaves larger holes that may accomodate other processes

Req - 90 KB - P_4



L26 Paging / Non-contiguous memory allocation

Page No.
Date

- The main disadvantage of dynamic partitioning is external fragmentation
  - can be removed by compaction, but with overhead
  - we need more dynamic / flexible / optimal mechanism, to load processes in the partitions

- Idea behind paging
  - if we have only two small non-contiguous free holes in the memory, say 1 KB each
  - if OS wants to allocate RAM to a process of 2KB, in contiguous allocation, it is not possible, as we must have contiguous memory space available of 2KB (external fragmentation)
  - what if we divide the process into 1KB - 1KB blocks.

- Paging Page size = Frame size
  - paging is memory management scheme that permits the physical address space of a process to be non-contiguous.
  - it avoids external fragmentations and the need of compaction
  - idea is to divide the physical memory into fixed-sized blocks called frames along with divide logical memory into blocks of same size called pages
  - page size is usually determined by the process architecture. traditionally, pages in system had uniform size, such as 4096 bytes, However, processor design often allow two or more, some times simultaneous, page size due to it's benefits

- Page table
  - a data structure stores which page is mapped to which frame.
  - the page Table contains the base address of each page in the physical memory
  - every address generated by CPU (logical address) is divided into two parts a page number (P), a page offset (d). the P is used as an index into a page table to get base address the corresponding frame in physical memory.

Page No.
Date

Logical memory
Process P_1

page 0
page 1
page 2
page 3

page Table	
0	1
1	4
2	3
3	7

physical memory

0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

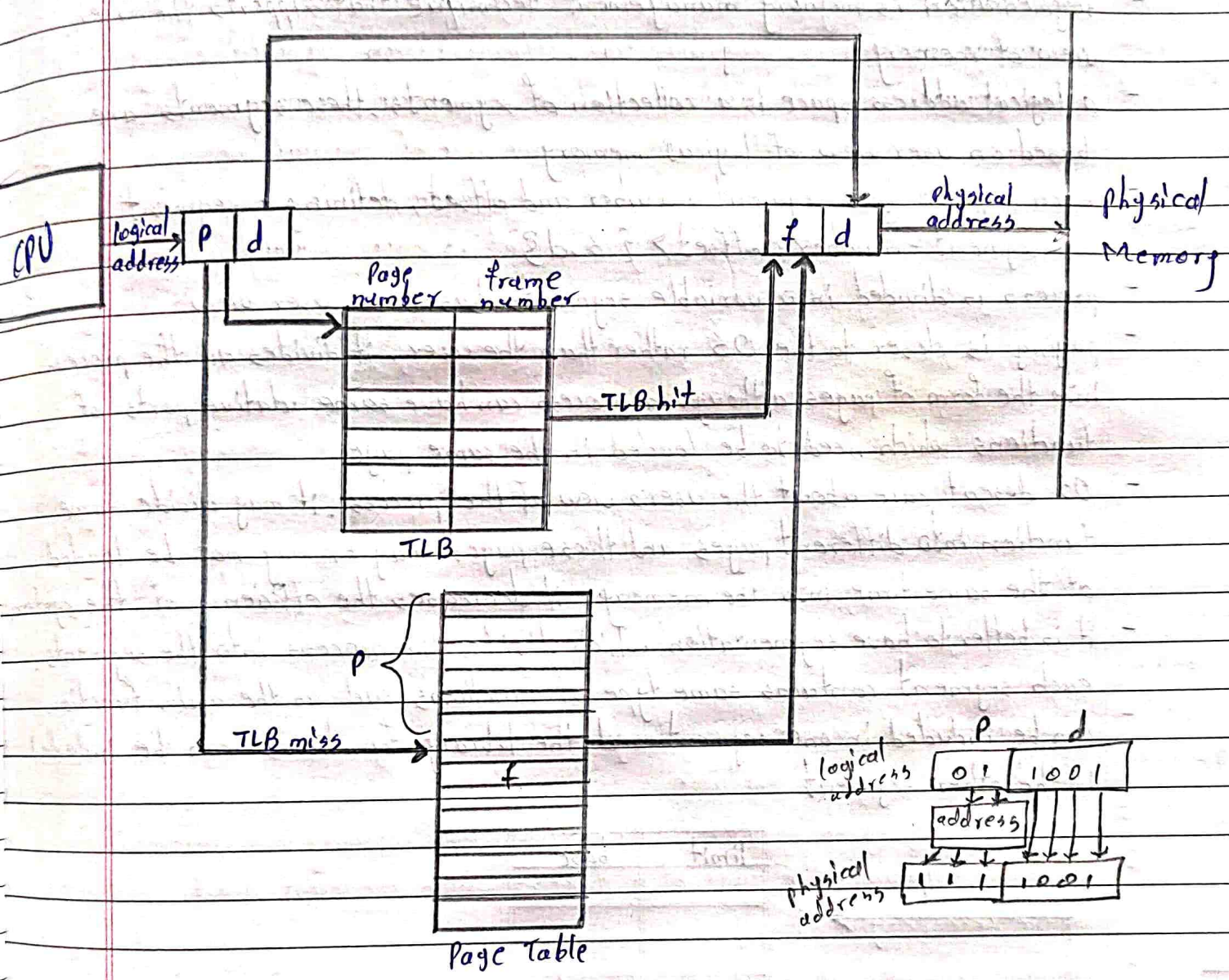
- page table is stored in main memory at the time of process creation and it's base address is stored in process control block (PCB)
- a page table base register (PTBR) is present in the system that points to the current page table. changing page tables requires only this one register, at the time of context-switching.

■ How paging avoids external fragmentation?
- non-contiguous allocation of the pages of the process is allowed in the random free frames of the physical memory

■ Why paging is slow and how do we make it fast?
- there are too many memory references to access the desired location in physical memory

■ Translation look-asid buffer (TLB)
- a hardware support to speed-up paging process
- it's a hardware cache, high speed memory
- TBL has key and value
- page table is stores in main memory and because of this when the memory references is made the translation is slow
- when we are retriving physical address using page table. after getting frame address corresponding to the page number, we put an entry of

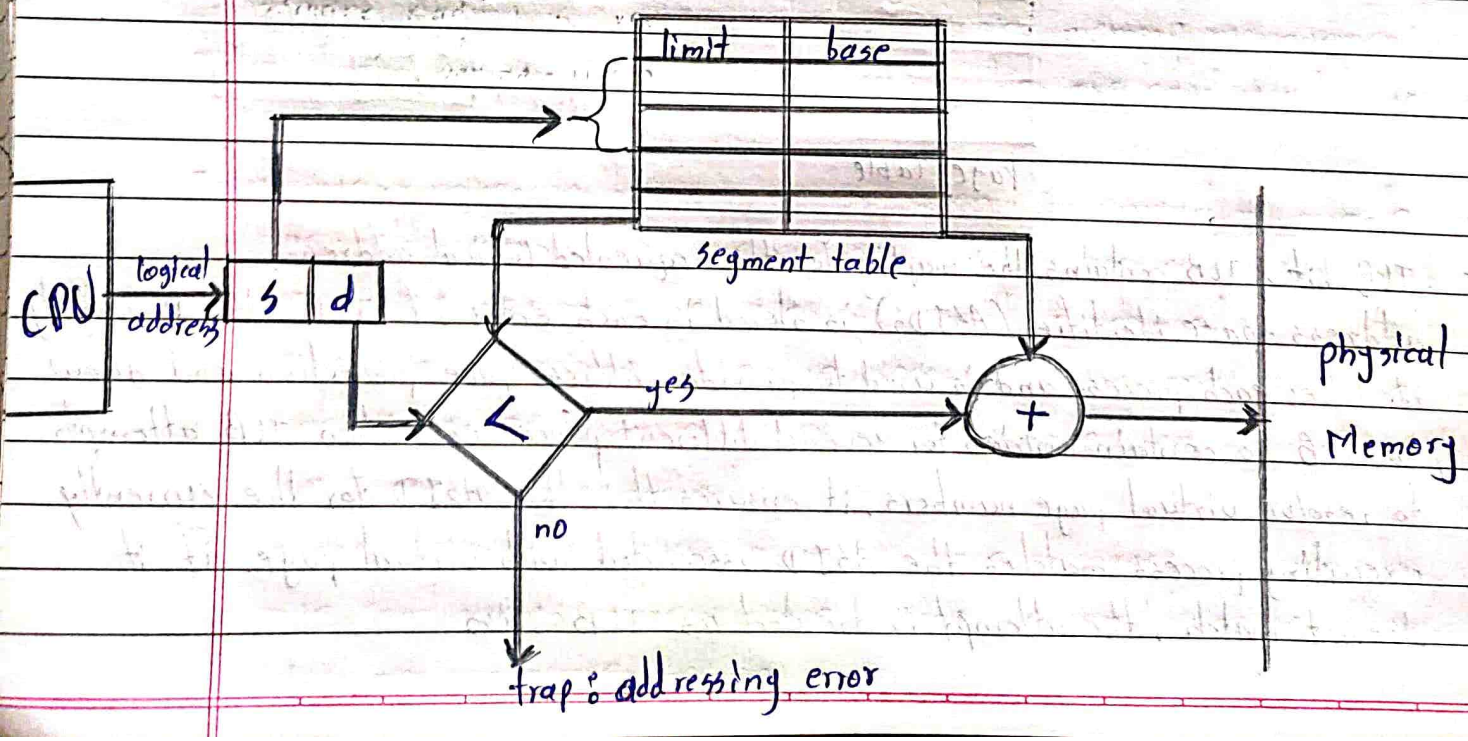
the into the TLB, so that next time, we can get the values from TLB directly without referencing actual page table. Hence, make paging process faster



- TLB hit, TLB contains the maping for the requested logical address.
- address space identifier (ASIDs) is stored in each entry of TLB. ASID uniquely identifies each process and is used to provide address space protection and allow to TLB to contain entries for several different processes. when TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently executing process matches the ASID associated with virtual page. if it doesn't match, the attempt is treated as TLB miss

L27 Segmentation / Non-contiguous memory allocation

- an important aspect of memory management that become unavoidable with paging is separation of user's view of memory from the actual physical memory
- segmentation is memory management technique that supports the user view of memory
- a logical address space is a collection of segments, these segments are based on user view of logical memory
- each segment has segment number and offset, defining a segment
  <segment-number, offset> {s, d}
- process is divided into variable segments based on user view
- paging is closer to the OS rather than the user. it divides all the process into the form of pages although a process can have some relative parts of functions which need to be loaded in the same page
- OS doesn't care about the user's view of the process. it may divide same function into different pages and those pages may or may not be loaded at the same time into the memory. it decreases the efficiency of the system
- it is better to have segmentation which divides the process into the segments. each segment contains same type of functions such as the main function can be included in one segment and the library functions can be included in the other segment



Page No.
Date

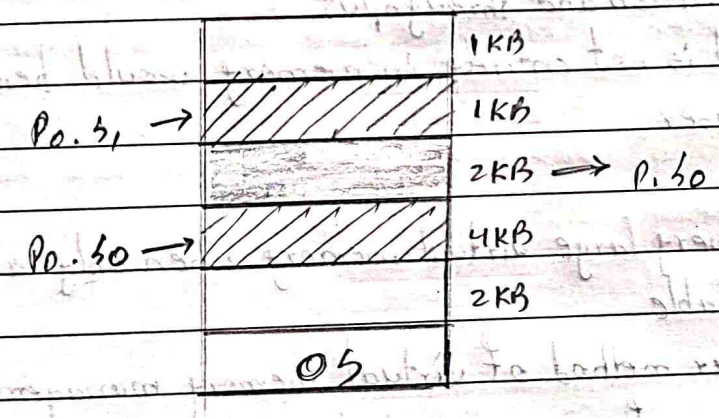
Advantages

- no internal fragmentation
- one segment has a contiguous allocation, hence efficient working within segment
- the size of segment table is generally less than the size of page table
- it results in a more efficient system because the compiler keeps the same type of functions in one segment

Dis-advantages

- External fragmentations
- the different size of segment is not good that the time of swaping
- moders system architecture provides both segmentation and paging implemented in some hybrid approach

External Fragmentation.



request P_1 S_0 → 2KB
S_1 → 3KB

3KB present in RAM but not useful for S_1 of 3KB becaus it not in contiguous form

L28 what is virtual memory / demand paging / page faults

Page No.
Date

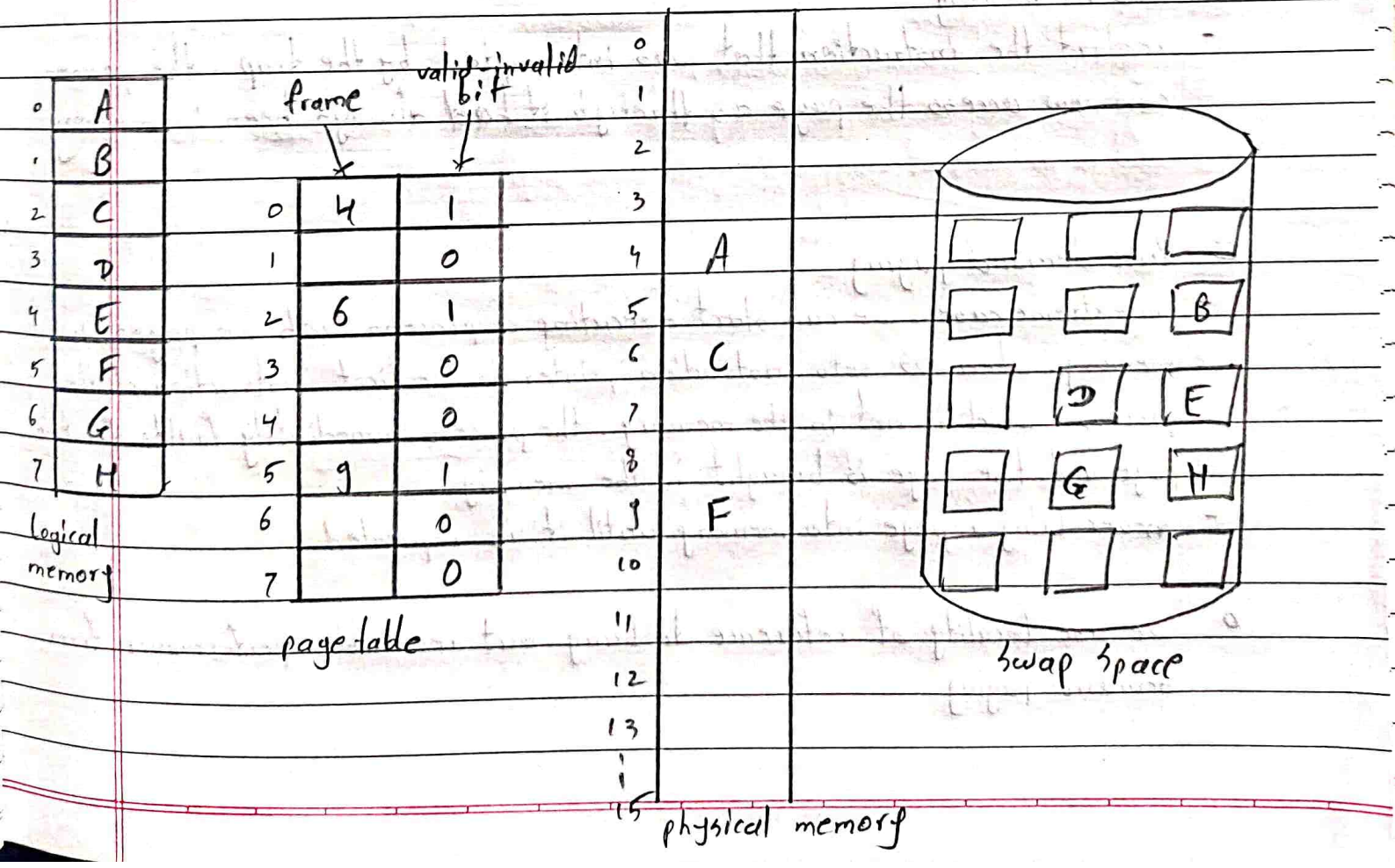
- Virtual memory is a technique that allows the execution of processes that are not completely in the memory. it provides user an illusion of having a very big memory (main). this is done by treating a part of secondary memory as the main memory. (swap space)
- Advantage of this is. programs can be larger than physical memory
- it is required that instructions must be in physical memory to be executed but it limits the size of a program to the size of physical memory. in fact, in many cases, the entire program is not needed at the same time so, we want an ability to executed program that is only partially in memory would give many benefits
  1) a program would no longer be constrained by the amount of physical memory that is available
  2) because each user program could take less physical memory, more programs could be run at the same time, with the corresponding increase in CPU utilization and throughput
  3) running a program that is not entirely in memory would benifit both the system and the user

- programmer is provided very large virtual memory when only a smaller physical memory is available
- demand paging is popular method of virtual memory management.
- in demand paging. the page of a process which are least used, get stored in the secondary memory
- a page is copied to the main memory when it's demand is made. or page fault occurs. there are various page replacement algorithms which are used to determine the pages which will be replaced.
- rather than swapping the entire process into memory, we use lazy swapper. a lazy swapper never swaps a page into memory unless that page will be needed.
- we are viewing a process as a sequence of pages, rather than only large contiguous address space, using the term swapper ~~replacement algorithm which are used to~~

is technically incorrect. a swapper manipulates entire process, where as a page is concerned which individual pages of process

How demand paging work?

- when a process is to be swapped-in, the pager guesses which pages will be used
- instead of swapping in whole process, the pager only brings only those pages into memory. This, it avoides reading into memory pages that will not be used anyway.
- above way, OS decreases the swap time and the amount of physical memory needed.
- The valid-invalid bit scheme in the page table is used to distinguish between pages that are in memory and that are on the disk
- valid-invalid bit 1 means, that associated page is both legal and in memory
- valid-invalid bit 0 means, the page either is not valid (not in the LAS of the process) or is valid but is currently on the disc



- if a process never attempt to access some invalid bit page. the process will be executed sucessfully without even the need pages present in the swap space
- what happens if the process tries to access a page that was not brought into memory, access to a page marked invalid causes page fault. paging hardware noticing invalid bit for a demanded page will cause a trap to the OS

□ The procedure to handle the page fault
- check an internal table (in PCB of the process) to determine whether the reference was valid or an invalid memory access.
- if referance was invalid process throws exception
- if referance is valid, pager will swap in the page
- we find a free frame (from free frame list)
- schedule a disk operation to read the desired page into the newly allocate
- when disc read is complete, we modifi the page table that, the page is now in memory
- restart the instruction that was interrupted by the trap. the process can now access the page as through it had always been in memory

□ Pure demand paging
- in extreme case, we can start executing a process with no pages in memory. when OS sets instruction pointer to the first instruction of the process, which is not in the memory. the process immediately faults for the page and the page is brought in the memory
- never bring a page into memory until it is requested

o we use locality of reference to bring out resonable performance from demand paging

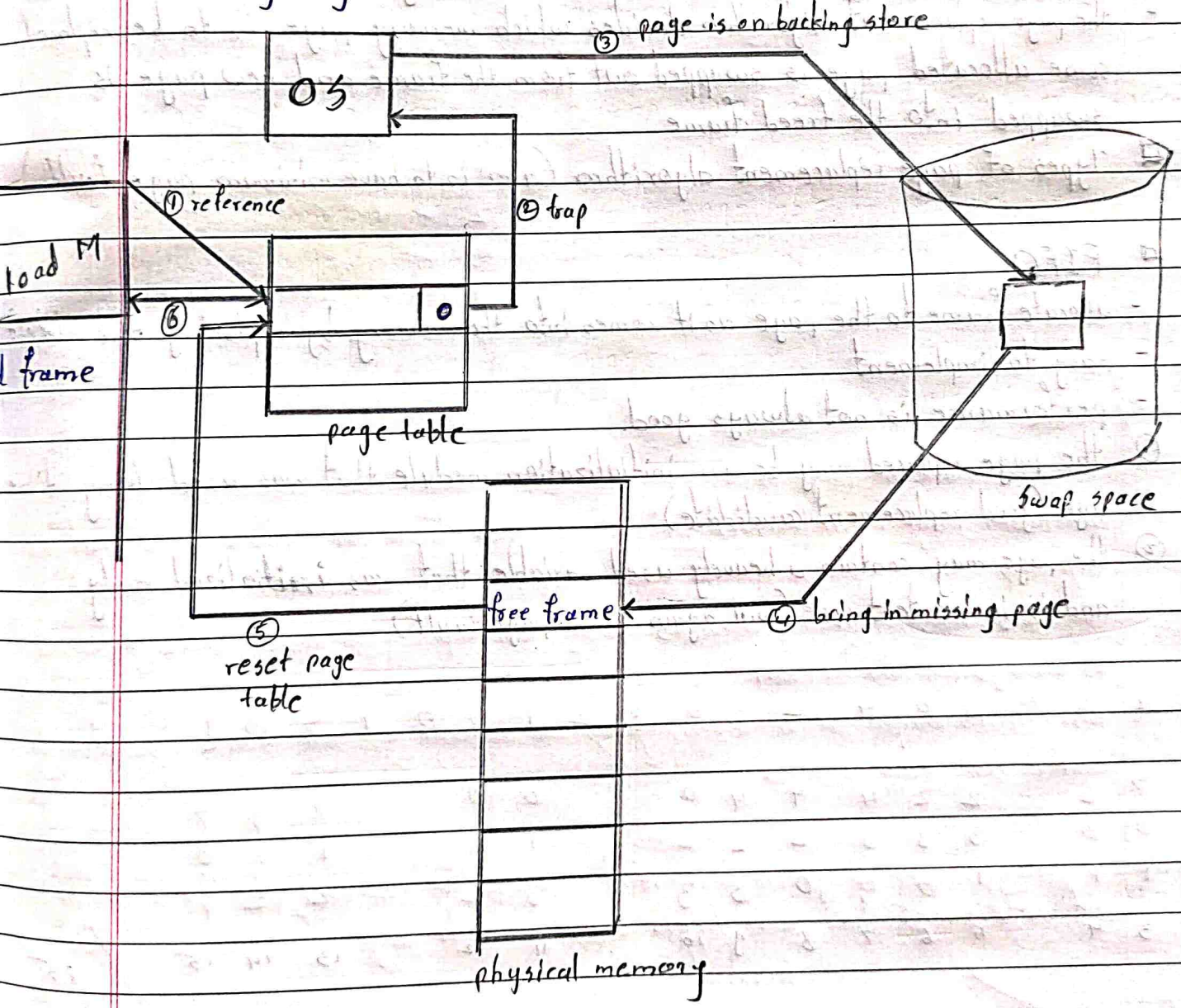
Page No.
Date

Advantages of virtual memory

- the degree of multi programming will be increased
- user can run large apps with less real physical memory

Dis advantages of virtual memory

- the system can become slower as swapping takes time
- Thrashing may occur



⑥ → restart instruction

L29 page replacement algorithms

- whenever page fault occurs, that is, a process tries to access a page which is not currently present in a frame and OS must bring the page from swap-space to a frame.
- OS must do page replacement to acommodate new page into a free frame, but there might be a possibility the system is working in high utilization and all the frames are busy, in that case OS must replace one of the pages allocated into some frame with the new page
- the page replacement algorithm decides which memory page is to be replaced some allocated page is swapped out from the frame and new page is swapped into the freed frame.

□ Types of page replacement algorithm (aim is to have minimum page faults)

□ FIFO
- allocate frame to the page as it comes into the memory by replacing the oldest page
- easy to implement
- performance is not always good

① the page replaced may be an initialization module that was used long time ago (good replacement candidate)

② the page may contain a heavely used variable that was initialized early and is in content use (will again cause page fault)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1
1	2	3	4		5	6	7	8	9	10			11	12			13	14	15

15 frames

Page No.
Date

Belady's anomaly is present

- in the case of LRU and optimal page replacement algorithms, it is seen that the number of page faults will be reduced if we increase the number of frames. However, belady found that, in FIFO page replacement algorithm, the number of page faults will get increased with the increment in number of frames
- this is the strange behaviour shown by FIFO algorithm in some of the cases

Optimal page replacement

- find if the page that is never referenced in future. if such a page exists, replace this page with new page
- if no such page exists, find the page that is referenced farthest in future. replace this page with new page
- lowest page fault rate among any algorithm
- Difficult to implement as OS requires future knowledge of reference string which is kind of impossible. (similar to SJF scheduling)

page

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		
1	2	3	4		5		6			7			8				9		

9 frames

least-recently used (LRU)

- we can use recent past as an approximation of the near future then we can replace the page that has not been used for the longest period

- Can be implemented by two ways
1) Counters
2) Stack

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		
1	2	3	4		5		6	7	8	9			10		11		12		

12 frames

1) Counters
- associate time field with each page table entry
- replace the page with smallest time value

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

count → universal

0 – ~~1 2 3 4~~ 5
1 – ~~1 2~~ 0
2 – ~~1 2 3~~ 4
3 – 1
4 –
7 – ~~1 2 3~~ 0

2) Stack
- keep a stack of page number
- whenever page is referenced, it is removed from the stack and put on the top
- by this, most recently used is always on top, and least recently used is always on the bottom
- as entries might be removed from the middle of the stack, so doubly linked list can be used

- counting based page replacement - keep the counter of the number of references that have been made to each page. (reference counting)

7	0	1	2	0	3	0	4	2	3	- -	—
7	7	7	2		2						
	0	0	0		3						
		1	1		1						

1	2	3
0	1	2
~~7~~	~~0~~	1

top → MPU
Bottom → LRU

Page No.
Date

* least frequently used (LFU)
  - actively used pages should have a large reference count
  - replace page with the smallest count

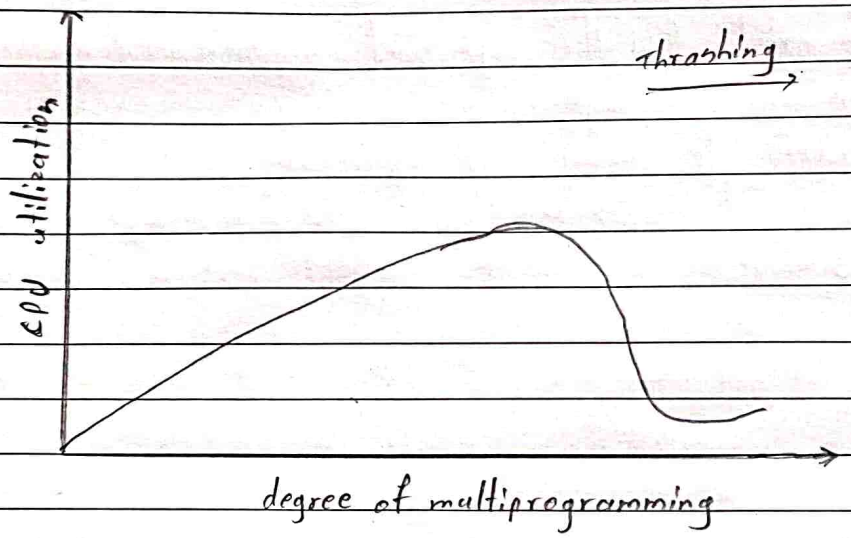
* Most frequently used (MFU)
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used
  -

* Neither MFU nor LFU replacement is common.

L30

Thrashing

- if the process doesn't have the number of frames it need to support pages in active use, it quickly page-faults. at this point, it must replace some page. However, since all it's pages are in active use, it must replace a page that will be needed again right away. consequently it quickly faults again, and again, and again, replacing pages that it must bring back in immediately
- This high paging activity is called thrashing
- A system is thrashing when it spend more time servicing the page faults than executing process



- Technique to Handle thrashing
  - Working set model
- this concept is based on the concept of locality model
- this basic principle states that if we allocate enough frames to a process to accomodate its current locality, it will only fault whenever it moves to some new locality. but the allocated frames are lesser than the size of the current locality, the process is bound to trash.

Page No.
Date

- page fault frequency
  - thrashing has high page fault rate
  - we want to control page fault rate
  - when it is too high, the process needs more frames. conversly, if the page-fault rate is too low, then the process may have too many frames
  - we establish upper and lower bounds on the desired page fault rate
  - if pf-rate exceeds the upper limit, allocate the process another frame. if pf-rate fails falls below the lower limit, remove a frame from the process
  - by controlling pf rate, thrashing can be prevented.