INTRODUCTION

Q) what is an operating system?

↳ acts as an intermediary between users of the computer and the computer hardware.

↳ operating systems goals :

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use computer hardware in an efficient manner

* COMPUTER SYSTEM STRUCTORE :

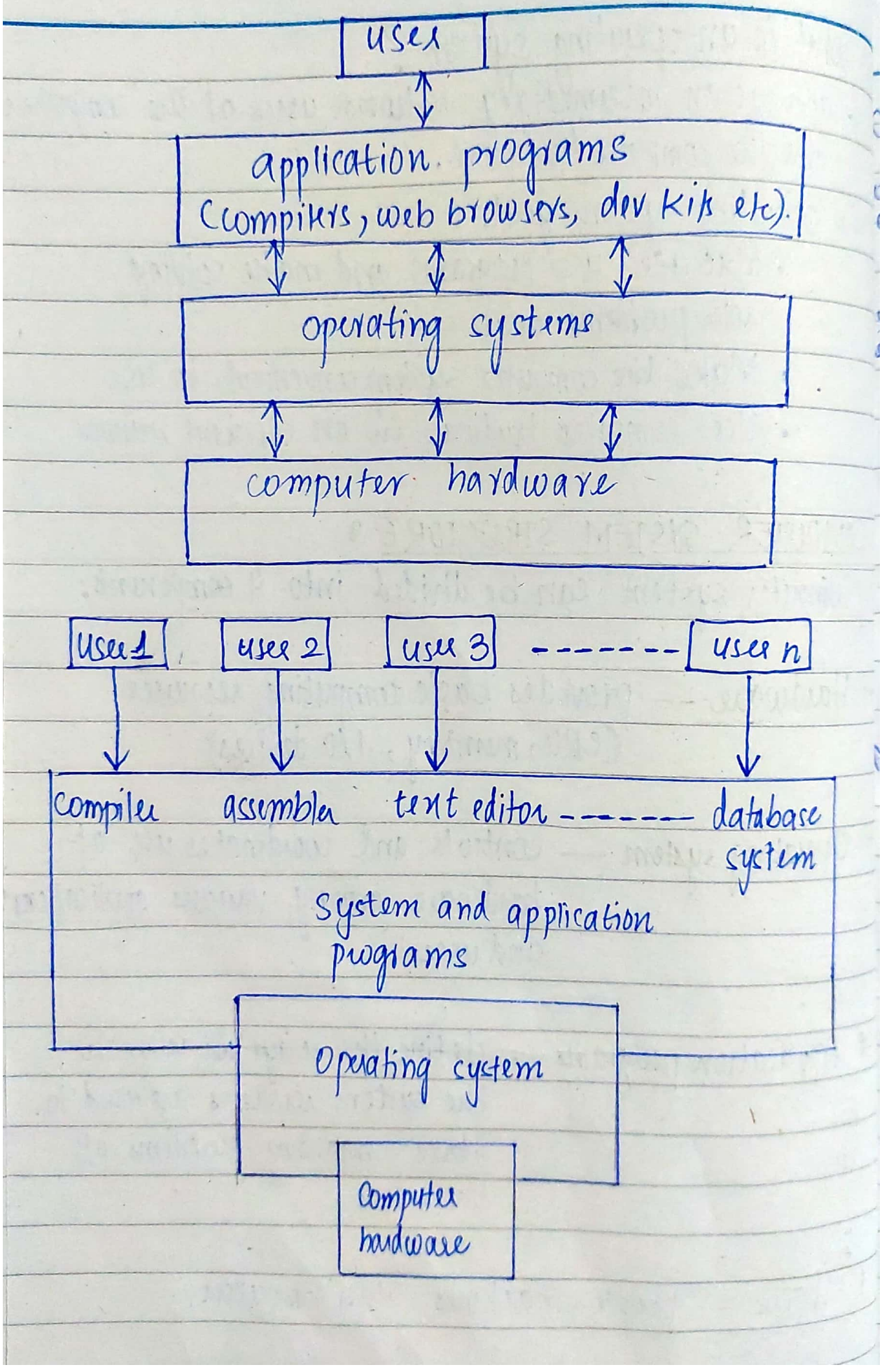
Computer system can be divided into 4 components:

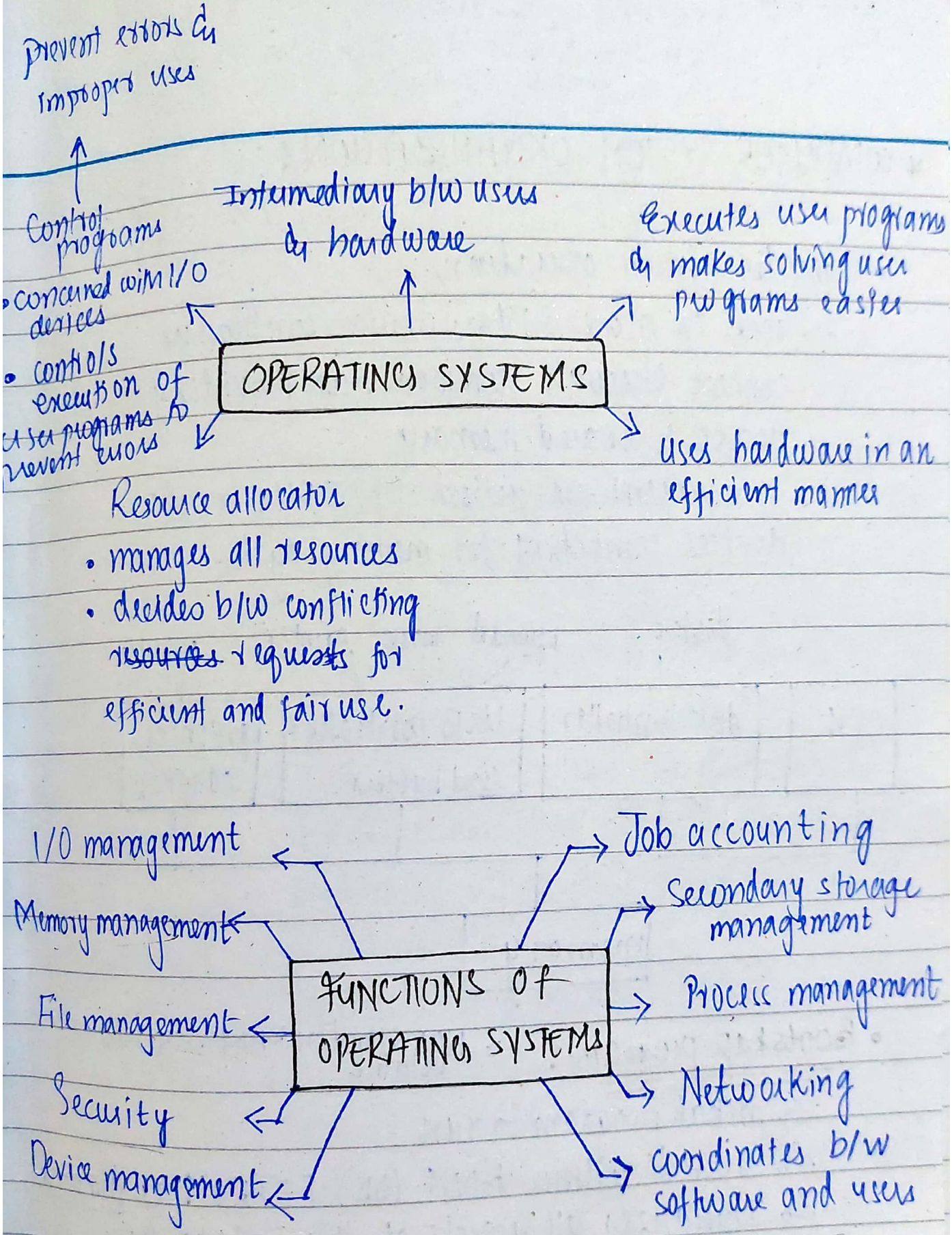
* Hardware — provides basic computing resources (CPU, memory, I/O devices)

* Operating system — controls and coordinates use of hardware among various applications and users

* Application programs — define the ways in which the system resources are used to solve computing problems of the users

* Users — people, machines, other computers

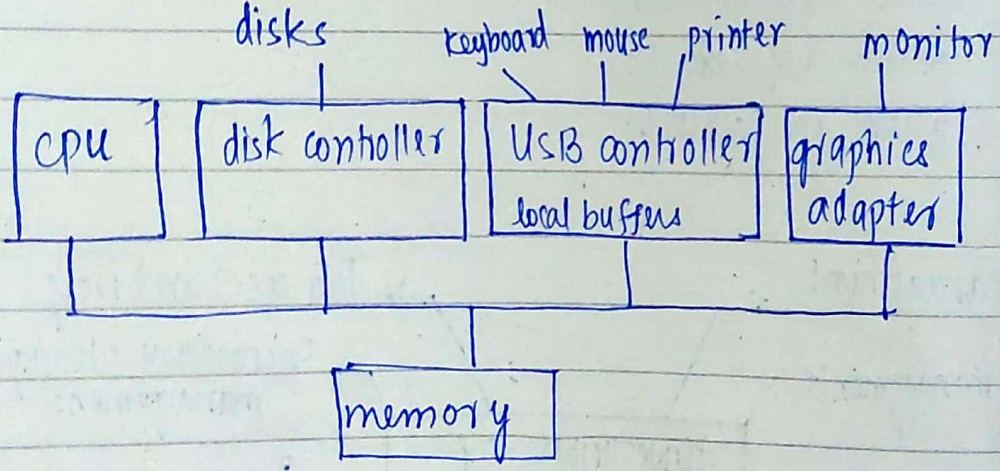

user
application programs
(compilers, web browsers, dev kits etc).
operating systems
computer hardware
User 1
User 2
User 3
User n
compiler
assembler
text editor
database system
System and application programs
Operating system
Computer hardware


prevent errors & improper uses
Control programs
• concerned with I/O devices
• control's execution of user programs to prevent errors
Intermediary b/w users & hardware
Executes user programs & makes solving user programs easier
OPERATING SYSTEMS
Uses hardware in an efficient manner
Resource allocator
• manages all resources
• decides b/w conflicting resources requests for efficient and fair use.
I/O management
Memory management
File management
Security
Device management
FUNCTIONS OF OPERATING SYSTEMS
Job accounting
Secondary storage management
Process management
Networking
coordinates b/w software and users

• Firmware: Software that provides basic machine instructions that allow the hardware to function and communicate with other softwares.

COMPUTER SYSTEM ORGANIZATION:

- Computer system operation
  - → one or more CPUs, device controllers connect through common bus providing access to shared memory
  - → concurrent execution of CPUs and devices competing for memory cycles



- Bootstrap program: → NOT stored in RAM bcz RAM is volatile
  - ↳ initial program to run
  - ↳ stored within ROM (OR) EEPROM as firmware
  - ↳ Initializes all aspects of the system, from CPU registers to device controllers to memory contents
  - ↳ loads Kernel into memory
  - ↳ then Kernel provides services to the system and its users and starts execution.

- **Kernel:**
  - Core of operating system stored in RAM
  - Provides the interface

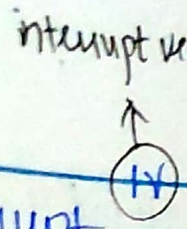
- OS is stored in secondary storage i.e. hard disk. Therefore bootstrap program loads Kernel to RAM.

- Software update programs are stored in virtual memory, then loaded to RAM.

- **Interrupt:**
  - A hardware signal from a device to the CPU.
  - Tells the CPU that the device needs attention and that the CPU should stop performing what it is doing and respond to the device.
  - Consists of a service routine
    - Software routine that hardware invokes in response to an interrupt

  - hardware
  - software
    - executes interrupt by "system call"

* If interrupt 0 is generated, CPU goes to interrupt vector, where interrupt service routine of corresponding address is stored (in RAM). Then this routine is sent for execution of the interrupts and the CPU resumes its previous task.



Interrupt → Interrupt service routine → Interrupt specific handler

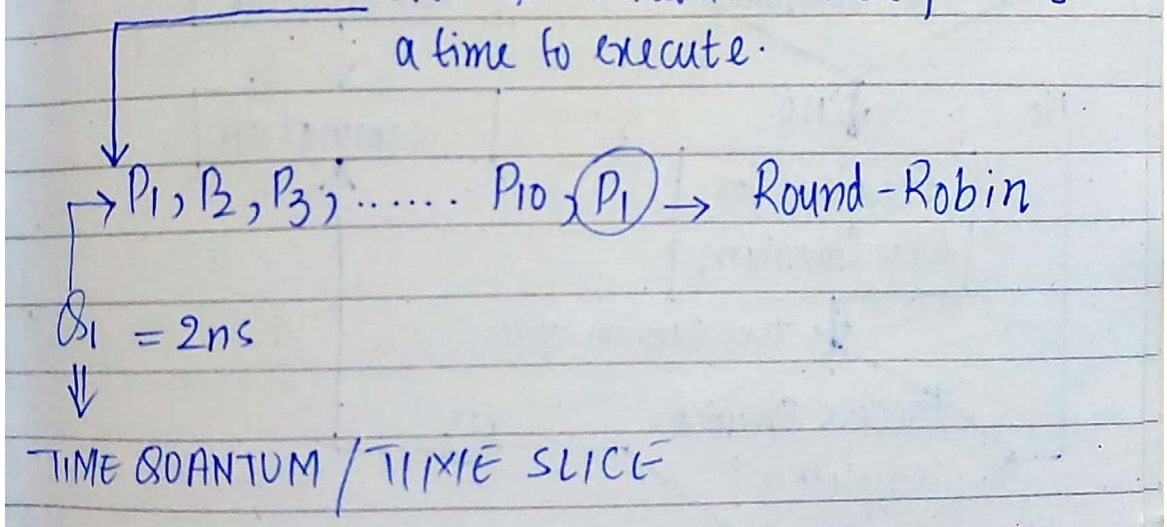
- local buffer : • Buffer used for I/O of data
  - region of memory used to temporarily store data while it is being moved from one place to another
  - allocated in process private memory

* COMPUTER SYSTEM OPERATION :

- I/O devices and CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers.
- I/O is from the device to local buffers of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt.

Key_stroke → Local buffer → Memory

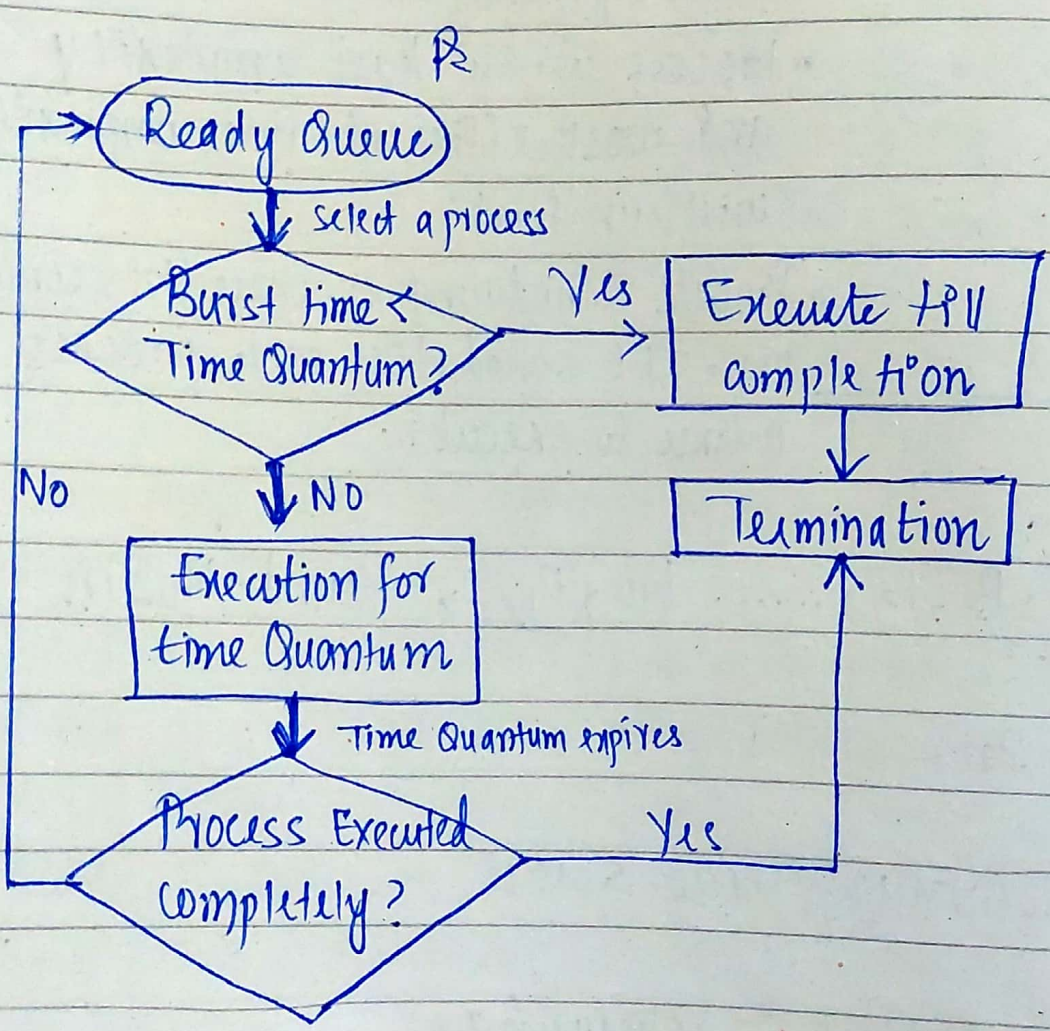
- **Preemption:** • Ability of the operating system to preempt (that is, stop or pause) a currently scheduled task in favour of a higher priority task.
  - Process with least complexity and most efficient time complexity will run first
  - Process scheduling is where this comes in. It schedules each process a time to execute.



- **Round-Robin Scheduling:**
  - ↳ CPU is assigned to the process on the basis of First come first Serve (FCFS) for a fixed amount of time.
  - ↳ This fixed amount of time is called time quantum or time slice
  - ↳ After time quantum expires, the running process is preempted (stopped / paused) and sent to the ready queue

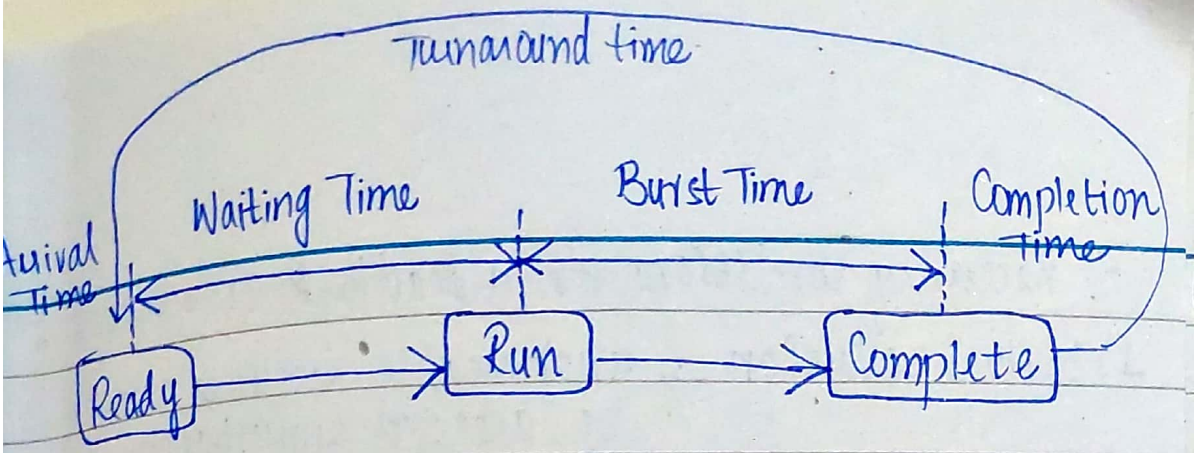
→ Then the processor is assigned to the next arrived process

→ It is always preemptive in nature



ROUND ROBIN SCHEDULING

- Burst Time: Total amount of time required by the CPU to execute the whole process.

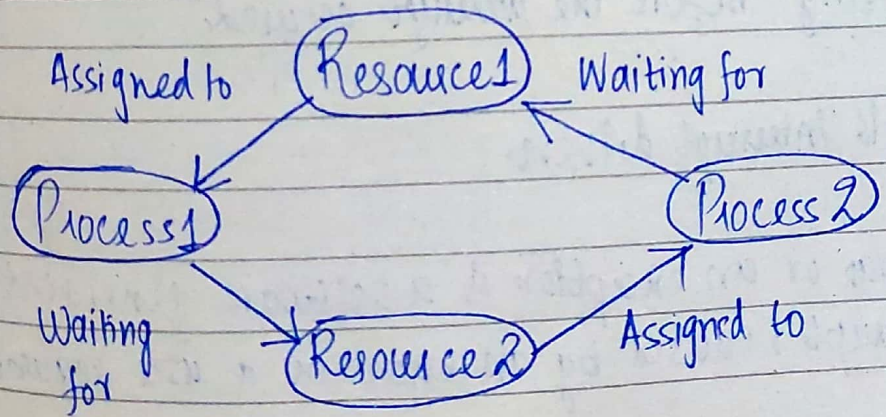


* Completion Time (CT) − Arrival Time (AT) = Waiting Time (WT) + Burst Time (BT)

* Turnaround Time (TAT) = Completion Time (CT) − Arrival Time (AT)

* Waiting Time (WT) = Turnaround Time (TAT) − Burst Time (BT)

* DEADLOCK :



- a situation in which more than 1 process is blocked because it is holding a resource and also requires some resource that is acquired by some other processes.

4 necessary conditions for a deadlock to occur:

1) Mutual exclusion → unshareable resources cannot be accessed simultaneously by processes / only 1 process can use resources at a time
2) Hold and wait
3) No preemption
4) Circular ~~set~~ wait → a set of processes are waiting for each other in a circular fashion

* Interrupt Handler:

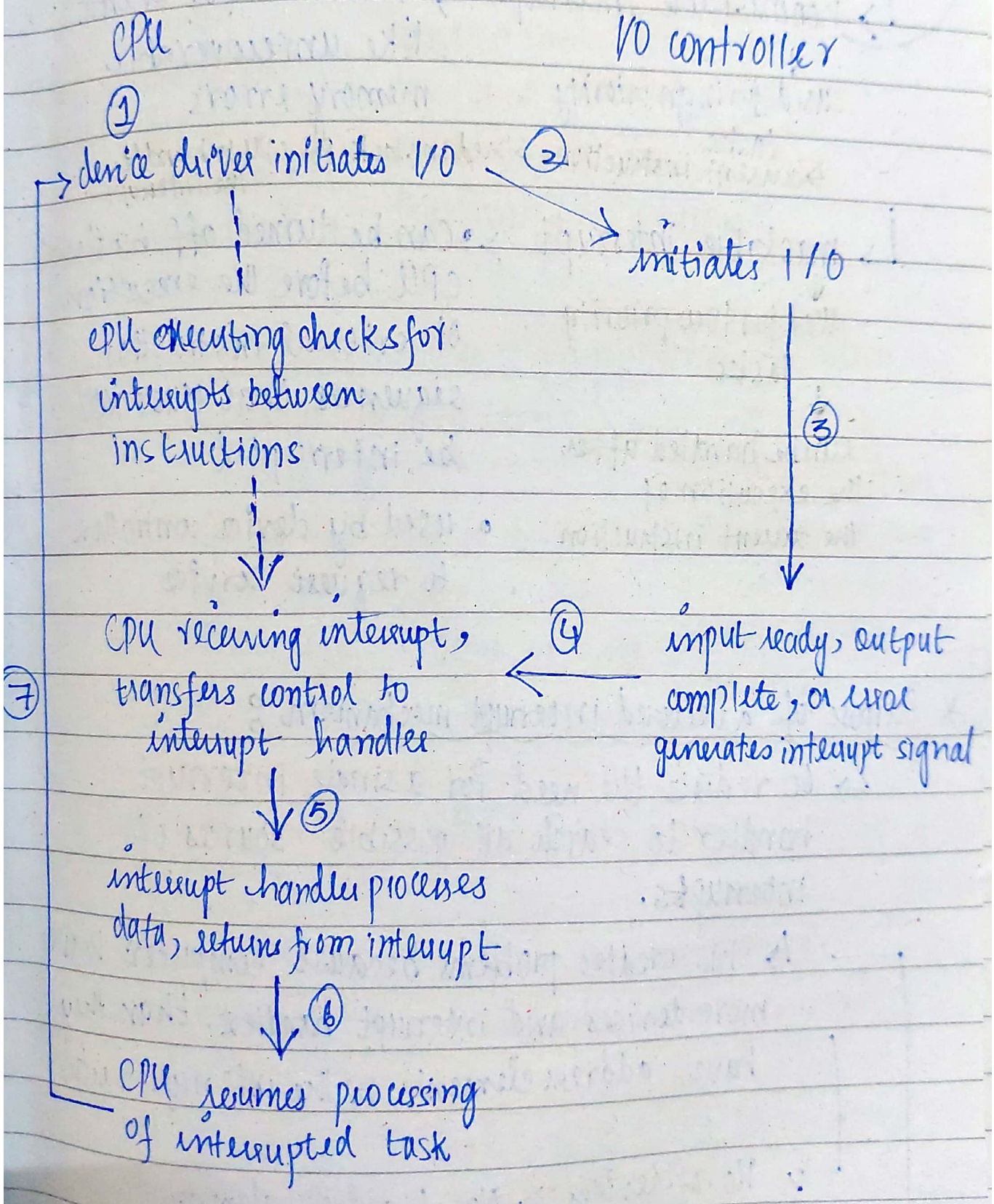
Job of interrupt handler is to service the device and stop it from interrupting. Once the handler returns, the CPU resumes what it was doing before the interrupt occured.

* OS is interrupt driven.

* A trap or an exception is a software generated interrupt, caused by an error or a user request.

* Unit of execution is known as thread.

* INTERRUPT-DRIVE I/O CYCLE :



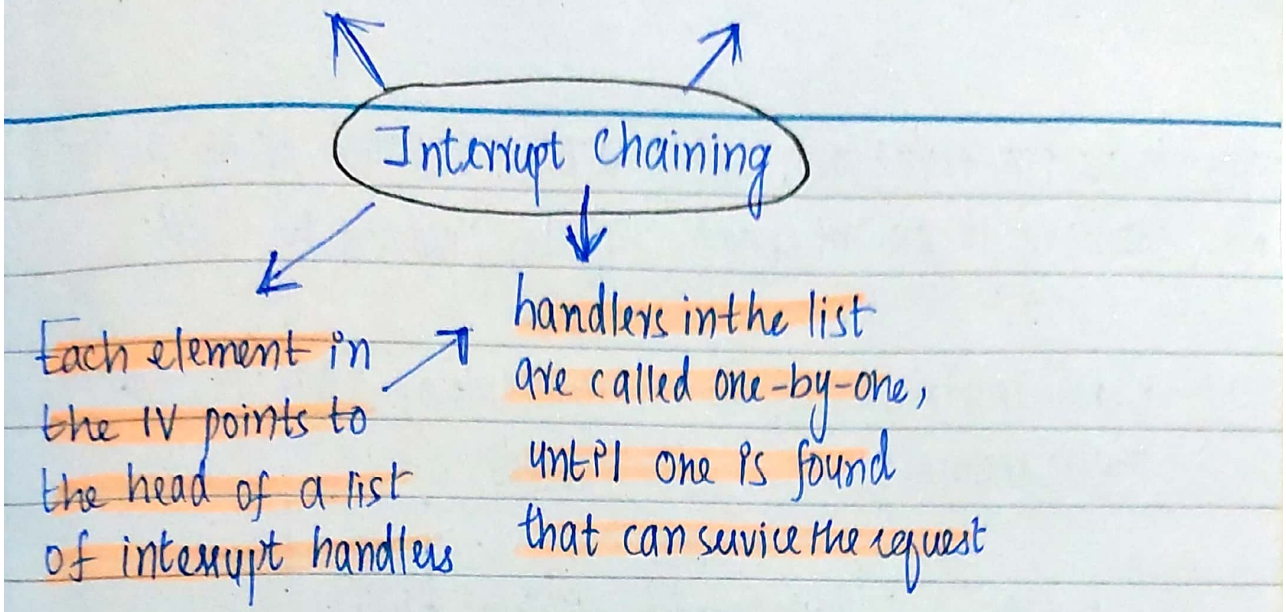
cannot be ignored

* Most CPUs have two interrupt request lines.
  - nonmaskable interrupt → reserved for events like unrecoverable memory errors
    - used for high priority tasks
    - current instructions stored in stack for CPU to handle the interrupt
  - maskable interrupt → • can be turned off by the CPU before the execution of critical instruction sequences that MUST NOT be interrupted
    - used for low priority tasks
    - can be handled after the execution of the current instruction
    - used by device controllers to request service

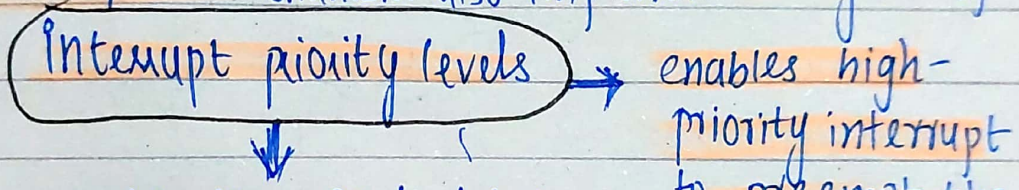
* Purpose of a vectored interrupt mechanism:
  - to reduce the need for a single interrupt handler to search all possible sources of interrupts.
    - this creates problems because computers have more devices and interrupt handlers than they have address elements in the interrupt vector
      - No. of devices > No. of address elements in IV
    - Solution: Interrupt Chaining

Software : polling

hardware : daisy chaining



* The interrupt mechanism also implements a system of



* STORAGE STRUCTURE :

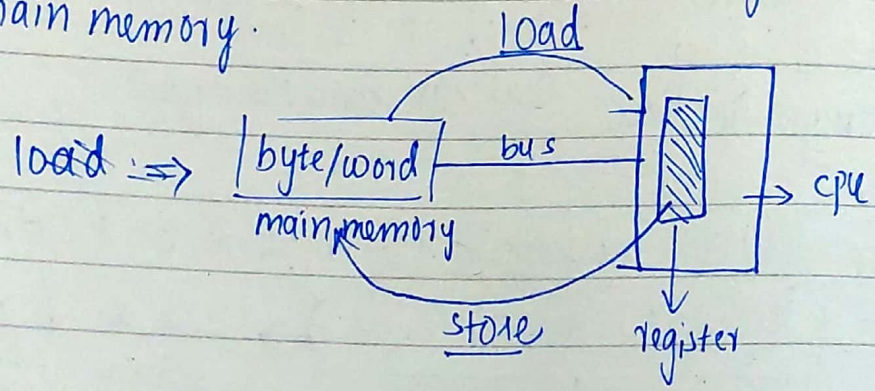
The CPU can only load instructions from the memory, so any programs must first be loaded into memory to run.

→ All forms of memory provide an array of bytes. Each byte has its own address.

Interaction is achieved through a sequence of load and store instructions to specific memory addresses.

→ load instruction moves a byte or word from main memory to an internal register within the CPU.

→ store instruction moves contents of register to main memory.



* MAIN MEMORY :

↳ only large storage media that the CPU can access directly
→ rewritable (RAM)
→ implemented in a semiconductor technology called DRAM.

(IMP)
* Programs and data cannot be stored in main memory because :

1. Main memory is too small to store all needed programs and data permanently
2. Main memory is volatile.

↳ solution : Secondary storage

* Registers are fastest because they can access data directly by the CPU. They are closest to the CPU since they are hardwired but they are expensive.

Speed α

* SECONDARY STORAGE :

↳ extension of main memory that provides large non-volatile storage capacity

eg. Magnetic disk

→ holds large quantities of data permanently

• hard-disk drives (HDDs)

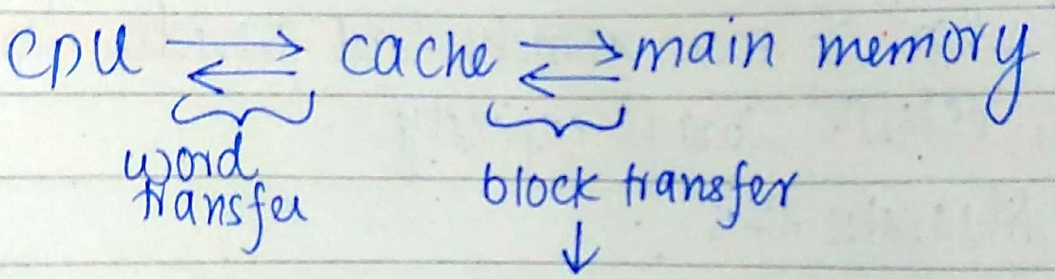
↳ stores the operating system, applications and data files permanently

→ ~~to~~ non-volatile

* slower than main memory. Most programs are stored in secondary storage until they are loaded into memory.

↳ used as both the source & destination of processing.

- Cache memory
  - ↳ software or hardware used to temporarily store information.
  - → uses caching – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
  - → located on a CPU chip or module



↓

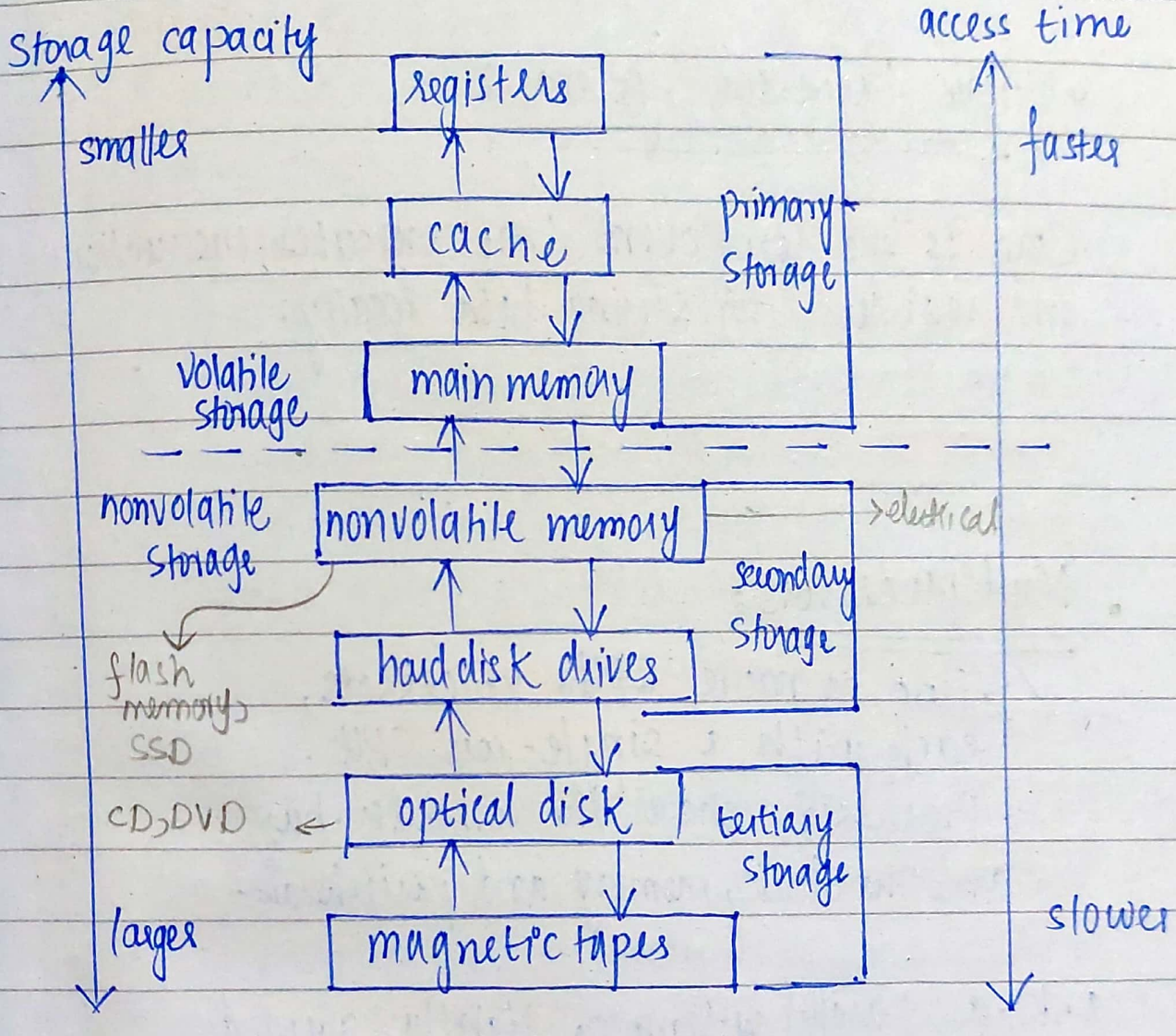
moves multiword structures from a source address to a destination address

* Cache is checked first to determine if information is in there.
  - ↳ if it is, then information is used directly from cache
  - ↳ if not, data is copied to cach and used there

NVS ⇒ non-volatile storage

NVM ⇒ non-volatile memory

* HIERARCHY OF STORAGE DEVICES :



NVS

Mechanical

- HDDs, optical disks, holographic storage, magnetic tape.
- larger and cheap per byte than

Electrical

flash memory, FRAM, NRAM, SSD.
aka NVM

- costly, smaller, but faster

*COMPUTER SYSTEM ARCHITECTURE:

• Single Processor Systems:

Core is the component that executes instructions and registers for storing data locally.

• Multiprocessors:

↳ Two or more ~~CPUs~~ processors, each with a single-core CPU.

↳ Processors share the computer bus and the clock, memory and peripherals

a.k.a parallel systems or tightly-coupled systems.

shared memory ↙ Single memory multiple CPUs

Fault tolerance – one processor fails, other becomes available (like star topology)

Graceful Degradation – w.r.t time, processors increased and CPU becomes process-wise degraded

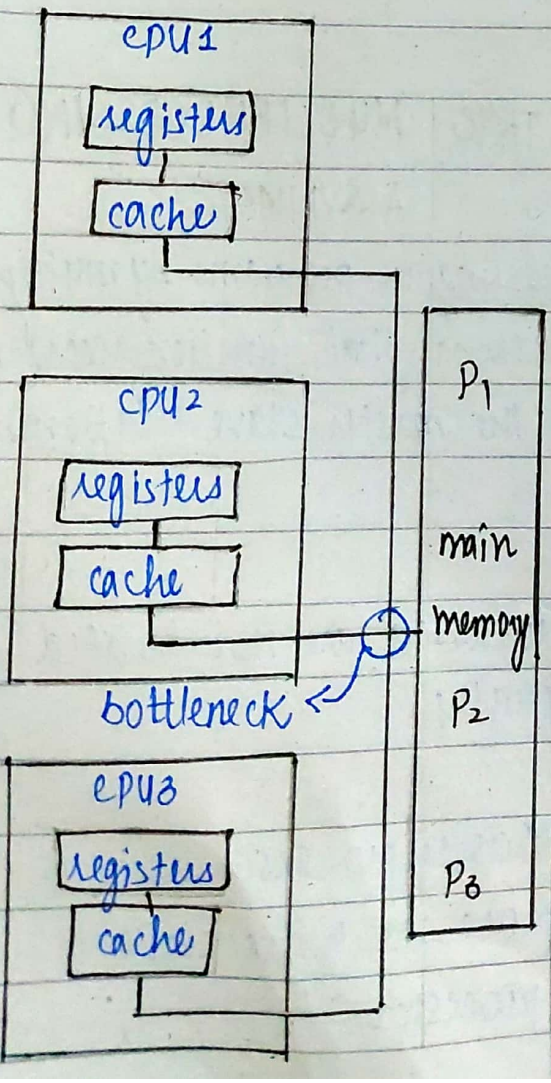
* Ready Queue – • Queue in which processes wait for CPU time.
• a ready queue has a set of processes in the main memory that is ready and waiting to execute

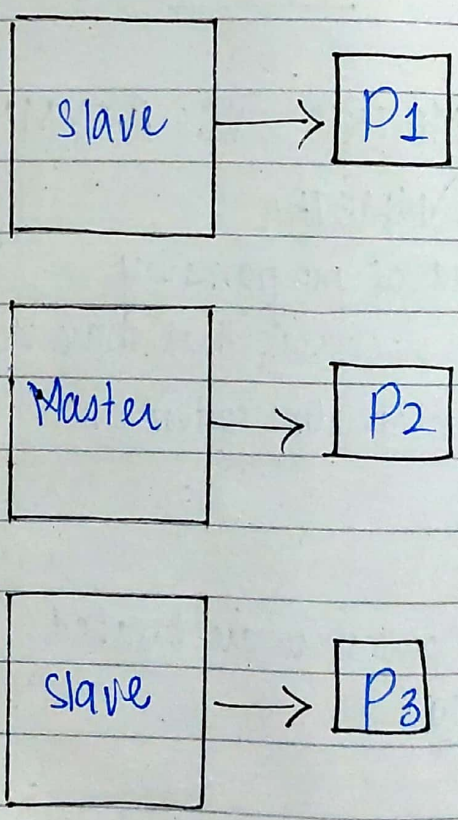
→ Primary advantage of multiprocessor systems is increased throughput i.e. by increasing no. of processors, we expect to get more work done in less time.
↳ Speed-up ratio with N processors is not N, however; it is less than N.
↳ when multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all parts working correctly. Due to this, expected gain from additional processors is lowered.

* SYMMETRIC VS ASYMMETRIC MULTIPROCESSING:

SYMMETRIC	ASSYMMETRIC
Processing of programs by multiple processors that share a common operating system and memory	Processing of programs by multiple processors that function according to the master-slave relationship
All the processors are treated equally	Processors are not treated equally
Processors take processes from the ready queue – each processor can have separate ready queues	Master processor assigns processes to the slave processors

Processors communicate with each other by shared memory	Processors communicate with the master processor
All processors have the same architecture	Architecture can be different for each processor
Not as easy to design or handle	Easier to design and handle
Comparatively costly	Cheaper .





* Bottleneck occurs because of increase in number of processors leading to increased traffic since memory is shared by all processors.
  ↳ assymmetric multiprocessing is the solution

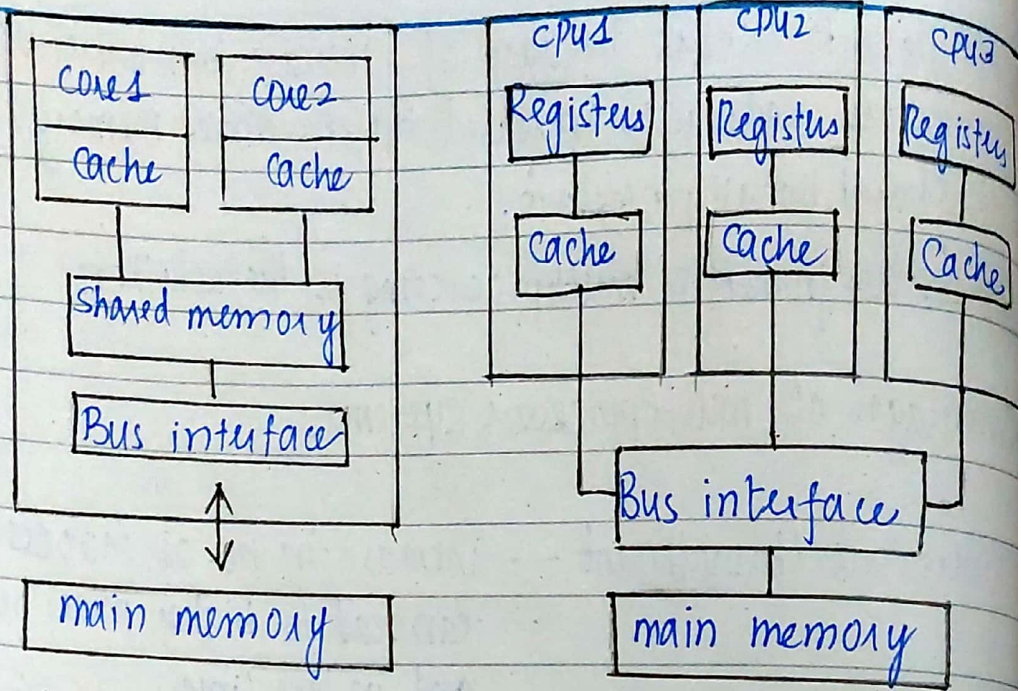
Advantages of multiprocessor systems:

① Increased throughput — increase in no. of processors can lead to higher power output and in less time

② Economy of scale — they can cost less than multiple single processor systems

③ Increased Reliability — failure of one process will not halt the system, only slow it down.

* MULTICORE VS MULTIPROCESSOR:
  ↳ multicore operates a single CPU while multiprocessor has multiple CPUs
  ↳ multicore has less traffic
  ↳ multicore system is more efficient for executing a single program whereas multiprocessor is more efficient for executing multiple programs simultaneously



Multicore processor

Multiprocessor

* UMA architecture :

Each processor has uniform access to memory a.k.a symmetric multiprocessors.

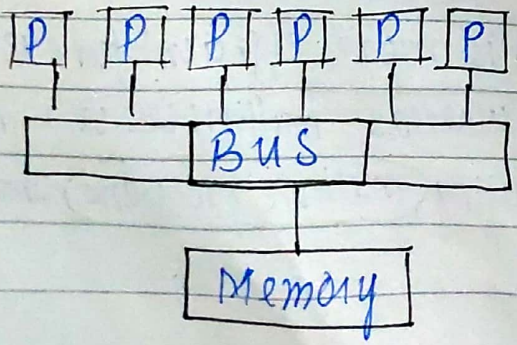
→ They are multichip and multicore.

→ ~~All processors can access~~

→ Offer limited bandwidth

→ Slower than NUMA since it uses a single memory controller

→ Used in time sharing and other applications



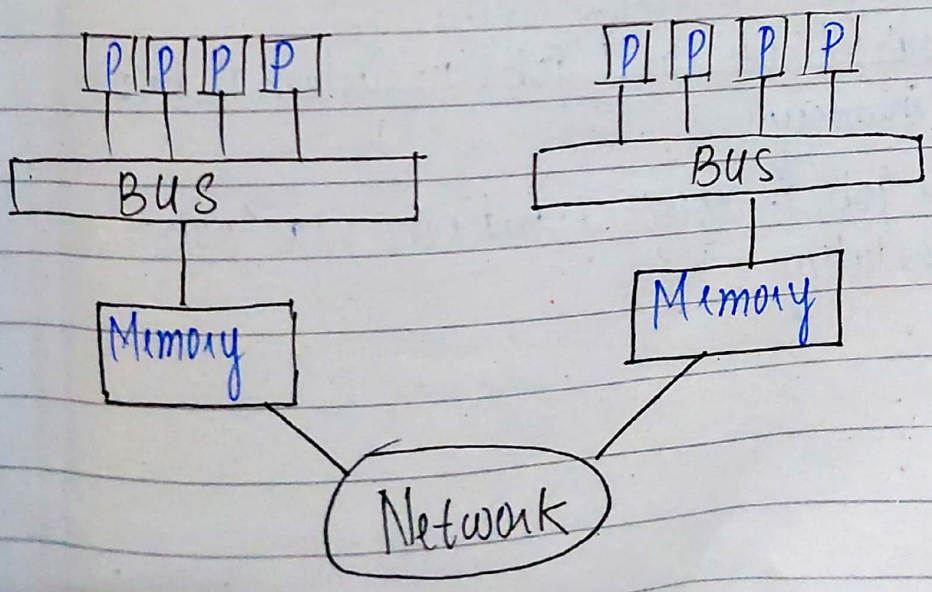
* NUMA Architecture:

↳ Time for memory access depends on location of data. Local access is faster than non-local access. Easier to scale than SMPs.

~~P P P P~~

→ Offer more bandwidth than UMA
→ uses multiple memory controls to speed up operations compared to UMA
→ memory access is unequal
→ Offers better performance and speed, suitable for real-time applications

→ A potential drawback is increased latency when a CPU must access remote memory across the system interconnect, creating a performance penalty.
→ Time is non-uniform since we leave the set.



* OPERATING SYSTEM OPERATIONS:

Q) What is a system call? Give 2 examples of system calls.

- It is a routine of operating system used to transit from kernel mode to user mode.
- It is a code of OS allocating memory for OS code
- It provides the means for a user program to ask the OS to puform tasks reserved for OS on the program's behalf.

→ It is invoked in a variety of ways depending on the functionality provided by the processor.

- MULTIPROGRAMMING:

↳ Required for efficiency
→ organizes jobs (code and data) so CPU always has one to execute
→ increases CPU utilization.
→ a subset of total jobs in system is kept in memory
→ one job is selected and run via job scheduling.

← Ready Queue : Simplified version of kernel data structure
↳ consists of a queue with one entry per priority
→ Queue in which process wait for CPU.

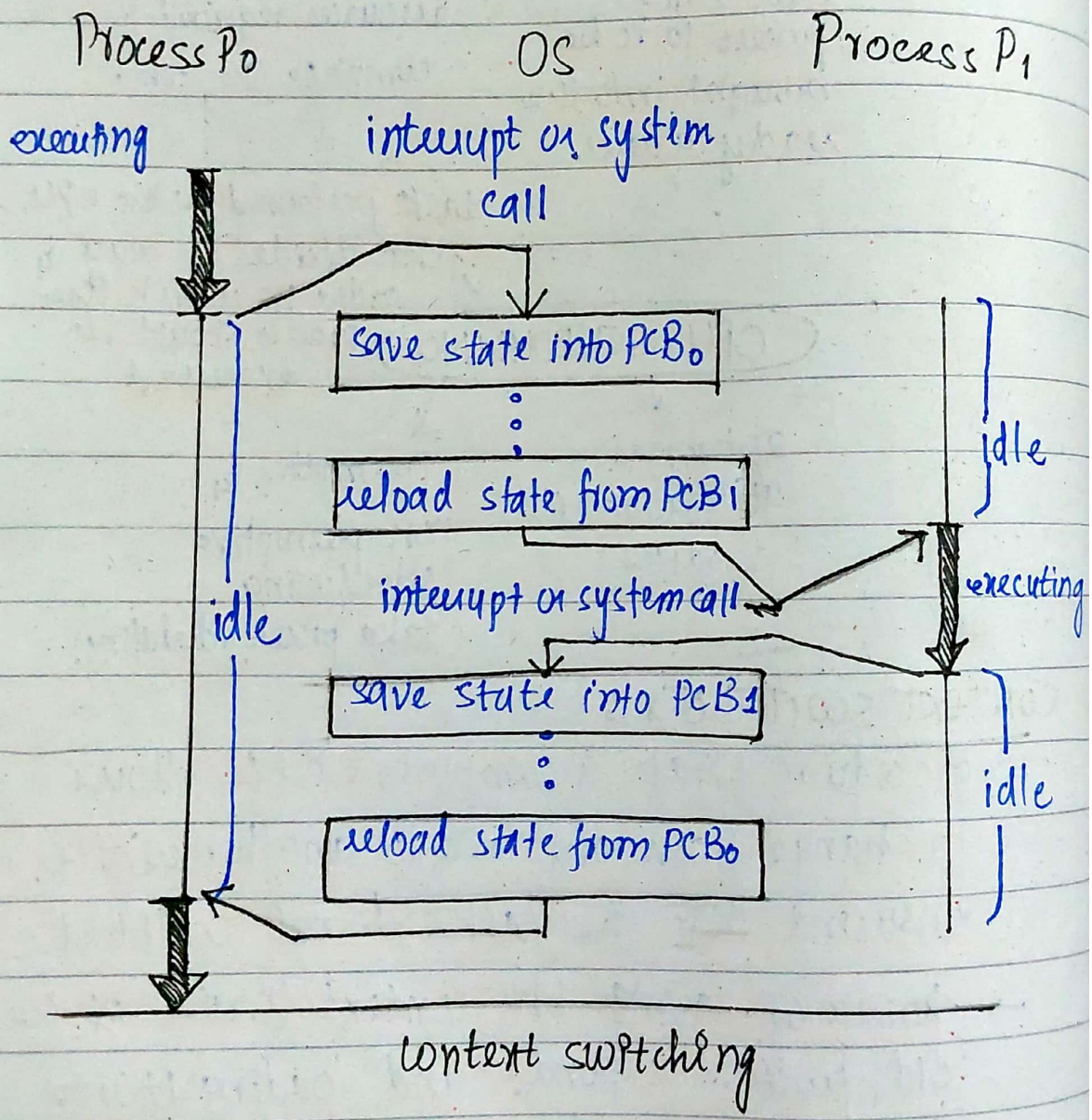
JOB SCHEDULING
↑ computer application for controlling unattended background program execution of jobs
↓ mechanism to select which process to to be brought into the ready queue.
→ job scheduler decides which jobs to execute at which time & the CPU resources required to complete the job.

CPU SCHEDULING
↑ task performed by the CPU that decides the way & order in which the process should be executed
↓ optimizes utilization of resources
↓ preemptive & non-preemptive scheduling (aka process scheduling)

* CONTEXT SWITCHING :
↳ procedure that a computer's CPU follows to change from one task to another while ensuring that the tasks do not conflict
→ process of saving the context (state) of the old process (suspend) and loading it into the new process (resume)
→ occurs when CPU switches b/w one process and another
→ saved states of currently executing process means to copy live registers to Process Control Block (PCB)
→ helps with utilization of OS

Restoring

→ ~~Restore~~ the state of the processor to run or execute next, which means copying live registers values from PCBs to registers.



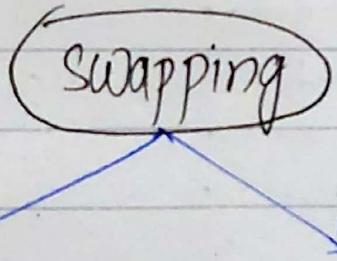
* SWAPPING:

↳ process by which a process is temporarily swapped (moved) from the main memory (RAM) to the secondary memory (disk).

→ may required bcz main memory has less storage space than secondary memory

→ System swaps from secondary to main memory later on

amount of memory swapped $\propto$ Total time



Swap-in

→ removing a program from a hard disk and moving it to RAM

Swap-out

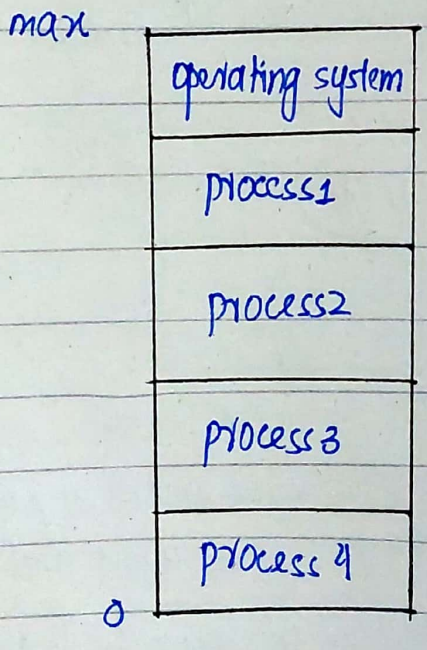
→ removing a program from RAM and storing it in secondary memory

→ helps with better utilization of memory

non multiprogrammed system – CPU sits idle

multiprogrammed system – OS switches to, and executes another process. When P1 is finished a system call is made and CPU returns back to that task.

- memory layout for multiprogramming system:



- MULTI-TASKING:
  - ↳ CPU executes mutiple processes by switching among them
  - → switches occus frequently, providing the user with a fast response time.

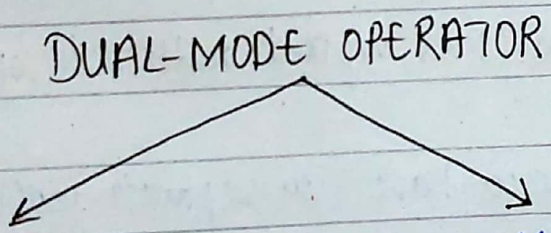
* OPERATING SYSTEM OPERATIONS: (contd.)

• DUAL-MODE OPERATION:

↳ allows the operating system to protect itself and other system components

↳ since OS and users share hardware and software resources, a properly designed OS must ensure security of the system

→ To ensure proper execution of the system, we must be able to distinguish b/w the execution of OS code and user-defined code.



- system boots, hardware starts in Kernel mode & starts user applications in user mode
- when a user application requests a service from the operating system, system transitions from user mode to Kernel mode

when computer system is run by user applications like creating a text document

NOTE: If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

→ Mode bit is added to the hardware of the computer to indicate current mode:
Kernel (0)
user (1).

↳ provides ability to distinguish when the system is running in user or kernel mode

↳ some instructions designed as privileged - only executable in kernel mode

→ system call changes mode to kernel, return from call resets it to user

• MULTI-MODE OPERATION:

↳ Increase in CPUs support multimode operation

↓

Intel processors have four separate protection rings, where 0 is kernel mode and 3 is user mode.

CPUs that support virtualization have a separate mode to indicate when the virtual machine manager (VMM) is in control of the system.

↳ handles virtual machines by changing CPU states

* TRANSITION FROM USER MODE TO KERNEL MODE:

↳ Timer to prevent infinite loop / process hogging resources

* Set interrupt after a specific period
* Operating system decrements counter
* When count = 0, interrupt is generated
* Set up before scheduling process to regain control or terminate program that exceeds allotted time.

→ System boot time → hardware starts in kernel mode

→ whenever trap or interrupt occurs, hardware switches from user mode → Kernel mode (reset mode bit 1 to 0).

via a system call:

↳ asks OS to perform tasks reserved for OS on the program's behalf.

↳ control passes through interrupt vector to a service routine corresponding to the address of the interrupt to execute the interrupt.

↳ additional information needed for the request maybe passed in registers, on the stack, or in memory (via pointers to memory locations passed in registers)

→ Once hardware protection is in place, it detects errors that violate modes

→ If a user program fails – executing an illegal instruction (privileged instruction in user mode), – hardware traps to the operating system

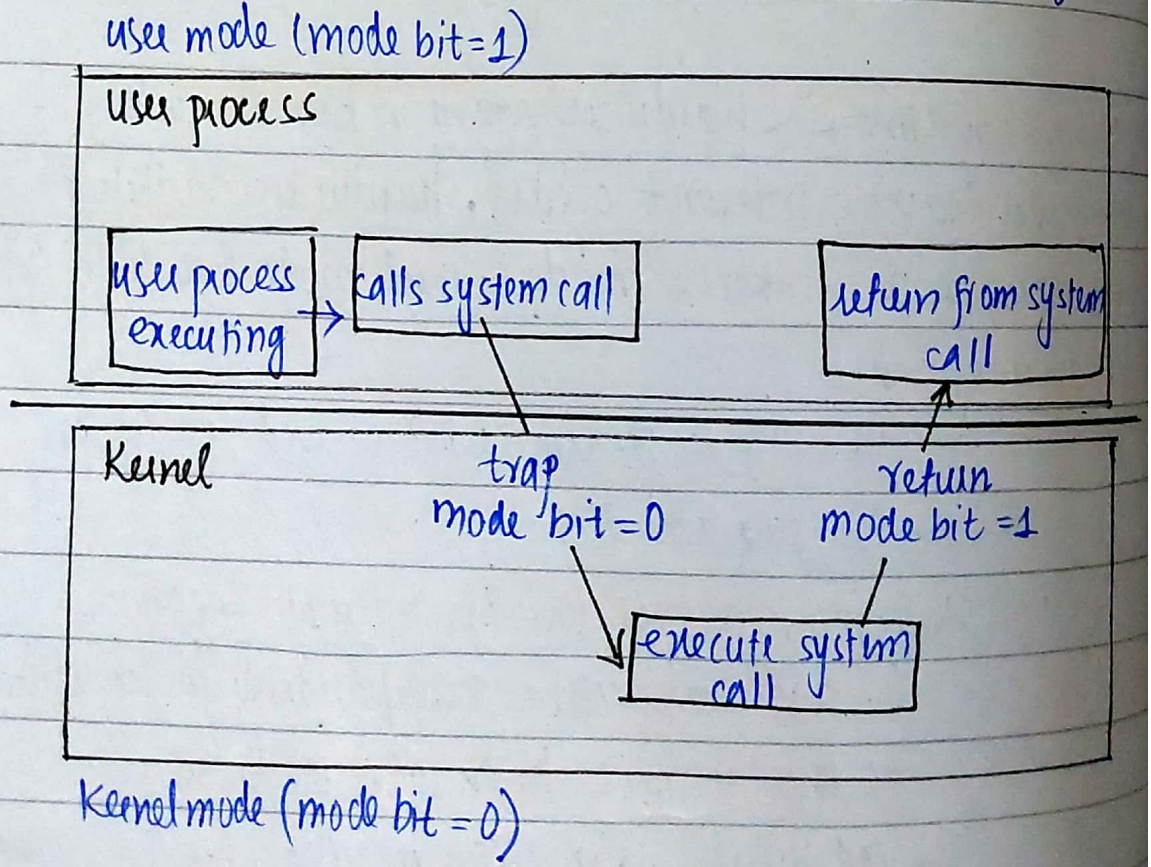
↳ Trap transfer control through IV to OS

↳ Program error is terminated abnormally by OS

↳ appropriate msg given

↳ memory is dumped

↳ written to a file so that programmer can examine it and restart the program.



mass-storage

RESOURCE MANAGEMENT & → memory, I/O, cache

↳ refer to diagram on 3rd page

* PROCESS MANAGEMENT :

Process → program in execution. It is a unit of work within the system.

Program → passive entity → content of a file stored on disk
Process → active entity →

* It is possible to provide system calls that allow processes to create sub-processes to execute concurrently.

* Process requires resources to accomplish its task namely - CPU, memory, files and I/O devices.
↳ Initialization data (input) is passed along.

* A single-threaded process has one program counter specifying the next instruction to execute. The execution of such a process must be sequential.

* A multi-threaded process has multiple program counters, each pointing to the next instruction to execute for a given thread

Program counter contains address for the next instruction to be fetched.

* The OS is responsible for the following activities in connection with process management:
  * Creating and deleting both users and system processes
  * Scheduling processes and threads on the CPUs
  * Suspending and resuming processes
  * Providing mechanisms for process synchronization
  * Providing mechanisms for process communication.

* MEMORY MANAGEMENT:

- main memory is the only large storage device that the CPU is able to address and access directly

The operating system is responsible for the following activities in connection with memory management:

* Keeping track of which parts of memory are currently being used and by whom
* Deciding which processes and data to move into and out of memory
* Allocating and deallocating memory space as needed

* FILE-SYSTEM MANAGEMENT :

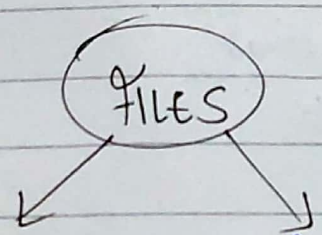
File – collection of related information defined by the creator.

↓

represents both programs and data

↓

.txt, .bin



* OS & maps files onto physical media and accesses these files via the storage devices

* Files are normally organized into directories to make them easier to use
  ↳ When multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (read, write, append).
* OS activities include :
  * Creating and deleting files and directories
  * Primitives to manipulate files and directories
  * Mapping files onto secondary storage
  * Backup files onto NVS media.

* MASS-STORAGE MANAGEMENT :

↳ computer must provide secondary storage to back up the main memory.

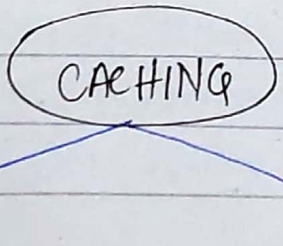
* OS is responsible for the following activities in connection with secondary storage management
  * Mounting and unmounting (~~main memory~~ → secondary (main-memory ↔ secondary storage)
  * Free-space management
  * Storage allocation
  * Disk scheduling
  * Partitioning
  * Protection

* Some storage need not be fast
  → Tertiary Storage → optical storage, magnetic tape
  ↓
  slow

  → Varies b/w WORM (write-once, read-many times) and RW (read-write)

* CACHE MANAGEMENT:

- Cache is a software or hardware used to temporarily store information



Software

1) cache checked first to determine if information is in there
   * If yes, info is used directly
   * If no, data is copied to cache and used

2) Internal programmable registers provide high-speed cache from main memory
   → Programmer/compiler implements register allocation and register replacement algorithms to decide relevance of information
   - main memory
   - registers

→ Data transfer from disk → memory controlled by OS

hardware

* Instruction cache to hold instructions expected to be executed next.
  ↳ ~~If this~~ w/o this cache, CPU would have to wait several cycles while an ins. was fetched from main memory
  ↳ Data transfer from cache → CPU and register (no OS intervention)

Local cache – technique used to speed network access data to files
- Caching data on clients rather than on servers
- allows multiple operations (write) combined into one across the network

• Sequential caching (single processor):

Magnetic disk → Main memory → cache → Hardware register

• Multiprocessors:

→ Each of the processes should obtain most recently updated values (cache coherence)
→ Each CPU contains a local cache
→ Cache coherence – uniformity of shared resource data that ends up stored in multiple local caches
aka data inconsistency
multiple caches may possess different copies of the same memory block

* I/O SYSTEM MANAGEMENT:

<> hides the peculiarities of hardware devices from the user
→ responsible for
- Memory management of I/O including
  1) Buffering (storing data temporarily while it is being transferred)
  2) Caching (storing parts of info in faster storage for performance
- General device driver interface
- Spooling (overlapping of output of 1 job with input

* PROTECTION AND SECURITY :

Protection – any mechanism for controlling access of processors or users to resources defined by the OS
- concerned with internal threats
- prevents unauthorized users from interfering with user's applications and data
- Authorization mechanism

Security – Defense of the system against external attacks
- Grants system access to appropriate users only
- Antiviruses
- Enayption and authentication mechanisms

Protection and security requires the system to be able to distinguish among all its users.

↳ OSs maintain a list of usernames and associated user identifier (user IDs).
↳ UNIQUE

→ User ID is associated with all files, processes of that user to determine access control

→ Group identifier (Group ID) allows sets of users to be defined and controls ~~managed~~ operations

↓

Owner of a file → all operations
user of that file → read only
} Discard server

Swapping → whole process transfered to the disk
Paging → parts of the process transfered to the disk

→ Privilege escalation allows user to change
  - to effective ID with more rights → extra permissi
  - setuid → causes that program to run with the user ID of the owner of the file rather than the current user's ID.
    (effective user ID)

Additional notes

PHYSICAL MEMORY	VIRTUAL MEMORY
• Actual RAM and a form of computer data storage that stores currently executing programs	• A memory management technique that creates an illusion to user a larger physical memory
• Actual memory	• Physical memory
• Faster	• Slower
• Uses swapping technique	• Uses paging (helps implement virtual memory)
• Limited to the size of RAM chip	Limited by the size of hard di
• Can directly access the CPU	• Cannot directly access the CPU

* COMPUTER ARCHITECTURE:

→ It is a blueprint for design and implementation of a computer system.
→ It provides the functional details and behaviour of a computer system and comes before the computer organization.
→ Computer architecture deals with 'what to do'.

* COMPUTER ORGANIZATION:

→ Computer organization is how operational parts of a computer system are linked together.
→ Computer organization deals with 'how to do'

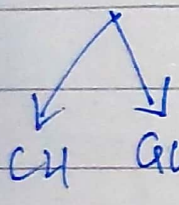
Computer Architecture	Computer Organization
1. Architecture describes (what) the computer does	1. The Organization describes (how) a computer performs operations
2. Computer architecture deals with the functional behavior of computer systems	2. Computer Organization deals with structural relationships

3. Architecture indicates the computer's hardware	Organization indicates the computer's performance
4. Architecture can be viewed as a series of instructions, addressing modes and registers	The implementation of the architecture is called organization.
5. For designing a computer, the architecture is fixed first	For designing a computer, organization is decided after its architecture
6. Computer architecture comprises logical functions such as instruction sets, registers, data types and addressing modes	Computer Organization consists of physical units like circuit designs, peripherals and adders
7. Architecture coordinates the hardware and software of the system (acts as an interface)	Computer Organization handles the segments of the network in a system
8. Architecture deals with high-level design issues	Organization deals with low-level design issues
9. Examples - Intel and AMD x86 processors.	invisible hardware elements such as interfacing of peripherals, memory techs and control signals.

CHP # 2 : OPERATING SYSTEM STRUCTURES

→ OS provides the environment within which programs are executed.
↳ focuses on savices that the system provides
→ provides interface that makes available to users
→ provides components and their interconnections.

User-interface - provides functions that are helpful to the user (user perspective)

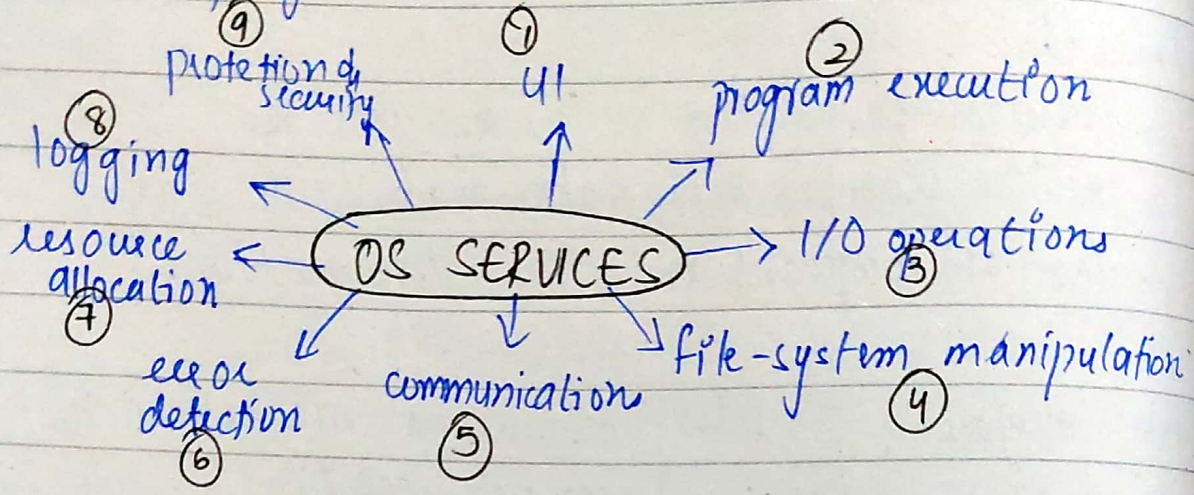


Program execution - System must be able to ~~read~~ load a program into memory and to run that program, end execution, either normally or abnormally

I/O operations - OS is responsible for transfering data to and from I/O devices

* OPERATING SYSTEM SERVICES:

↳ OS provides environment for the execution of programe



(1) User-interface:

↳ a space where human-computer interactions occur

- CLI → uses text commands and a method for entering them
- GUI → window system with a mouse serving as a pointing device to direct I/O, choose from menus and make selections, and a keyboard to enter text

(2) Progam-execution:

↳ the system must be able to load a program into memory and to run that program, end execution, either normally or abnormally

③ I/O operations:

→ OS is responsible for transfering data to and from I/O devices

→ a running program may require I/O, which may involve an I/O device.

→ For efficiency and protection, users cannot directly control I/O devices, therefore OS must provide means to ~~cont~~ perform I/O operations

④ File-system manipulation:

→ programs need to read and write files and directories

→ creation and deletion by name, searching for a given file and list information is also required

→ OS includes permissions management to allow or deny access to files or directories based on file ownership

→ Privilege escalation is also provided by the OS for extra permissions

⑤ Communications:

→ circumstances occur in which one process needs to exchange information with another process. Such communication may occur b/w processes that are executing on the same computer or b/w processes that are executing on different computer systems tied together by a network.

→ Communications may be implemented via

↳ Shared memory → Two or more processes read and write to a shared selection of memory

→ Used for communication b/w processes on a single processor or multiprocessor systems where the communicating processes reside on the same machine as the communicating processes share a common address space

→ Message passing → Packets of information in predefined formats are moved b/w processes by the OS

→ Typically used in a distributed environment where communicating process reside on remote machines connected through a network

⑥ Error detection:

↳ OS needs to be constantly aware of possible errors

→ Errors may occur in the CPU and memory hardware (memory error or power failure)

→ Errors may occur in I/O devices (parity error on disk, connection failure to a network, lack of paper in printer, printer cartridge empty)

→ Errors may also occur in user programs (arithmetic overflow, accessing illegal memory location)

→ For each type of error, OS should take appropriate action (compiler error etc.)

⑦ Resource allocation:

↳ OS must manage all resources since multiple processes are running at the same time

→ Many types of resources need to be allocated for fair use → CPU cycles, main memory, file storage, I/O devices → special allocation code

→ Some managed with generic systems eg. CPU scheduling routines.

⑧ Logging :

→ To keep track of which programs use how much and what kinds of computer resources

→ Record keeping used for accounting or simply for accumulating usage statistics

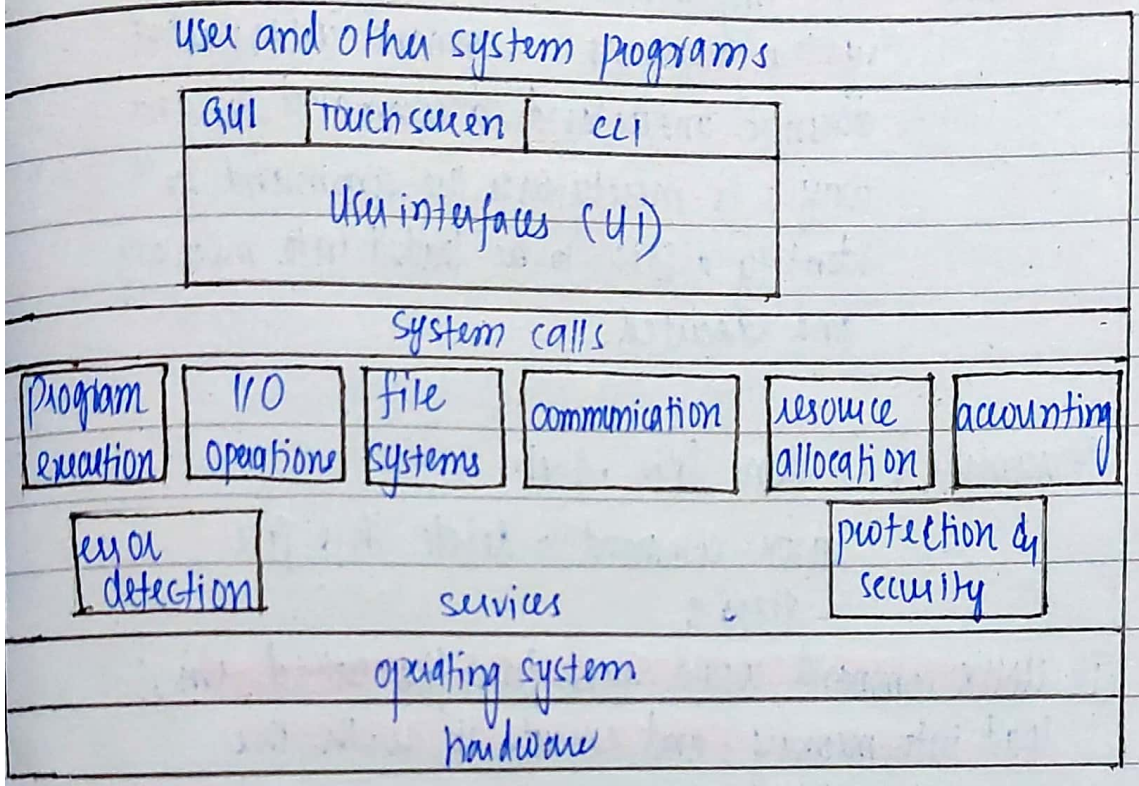
↓

Valuable tool for system administrators who wish to reconfigure the system to improve computing services

⑨ Protection and Security :

Protection – Prevents unauthorized users from interfering with user's applications and data
- Authorization mechanism
- Deals with internal threats

Security – Defense of the system against external threats
- Grants system access to appropriate users only
- Encryption and authentication mechanism



* all services are requested by the user program via system calls.

* USER AND OPERATING-SYSTEM INTERFACE :

① Command-Line Interface (CLI)/ Command Interpreter :

→ CLI allows direct command entry

→ gets and executes the next user-specified command
  ↳ file creation, deletion, list, print, copy, execute etc.

Approach ① : CLI itself contains the code to execute the command (command interpreter jumps to a section of a commands code that sets up the parameters and makes the appropriate system call)

Approach ②: CLI implements most commands through system programs. Command interpreter doesnot understand a command in any way; it merely uses the command to identify a file to be loaded into memory and executed.

Example: rm file.txt

UNIX command to delete this file

steps:

① UNIX command would search for a file called rm, load into memory, and execute it with the parameter file.txt.

→ Logic associated with "rm" command would be defined completely by the code in the file rm.

② Graphical User Interface (GUI):

↳ user friendly desktop metaphor interface

→ Mouse, keyboard, and monitor

→ Icons represent files, programs, actions etc.

→ Mouse actions ~~provide~~ invoke information, program, select a file or directory (folder) or pull down a menu that contains commands

KDE & GNOME desktops run on Linux both CLI & GUI

③ Touch-Screen Interface:

↳ Touch-screen devices require new interfaces since CLI and GUI don't work on touch screens

→ Makes use of gestures

→ Apple devices use Springboard touch-screen interface

→ Virtual keyboard for text entry

→ Voice commands

* SYSTEM CALLS :

↳ a system call is a routine of the OS used to transit from user mode to kernel mode via a mode bit (from mode bit = 1 to mode bit = 0). It provides the means for the user program to request the OS to perform tasks reserved for the OS on the program's behalf.

Example : Writing a simple program to read data from one file and copy them to another file.

→ First input : Names of input and output file
(cp in.txt out.txt)

→ This command copies the input file in.txt to the output file out.txt

→ Another approach is for the program to ask user for the names ; this will require a sequence of system calls

↳ First to write a prompting message on the screen

→ read from the keyboard the characters that defines two files / select file names via mouse-clicking

→ requires many I/O system calls

→ Once the two file names have been obtained, program opens the input file and creates and opens the output file, requires a system call for each process.

⇒ ↳ Error conditions to be handled ; output error message (sequence of system calls) and terminate the program abnormally (another system call)

↳ Output file of the same name might be deleted (system call) and ~~exist~~ a new one might be created (another system call).

↳ User sent a message whether to replace the existing file or about the program i.e. dialogue box (system call)

→ After both files have been set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call).

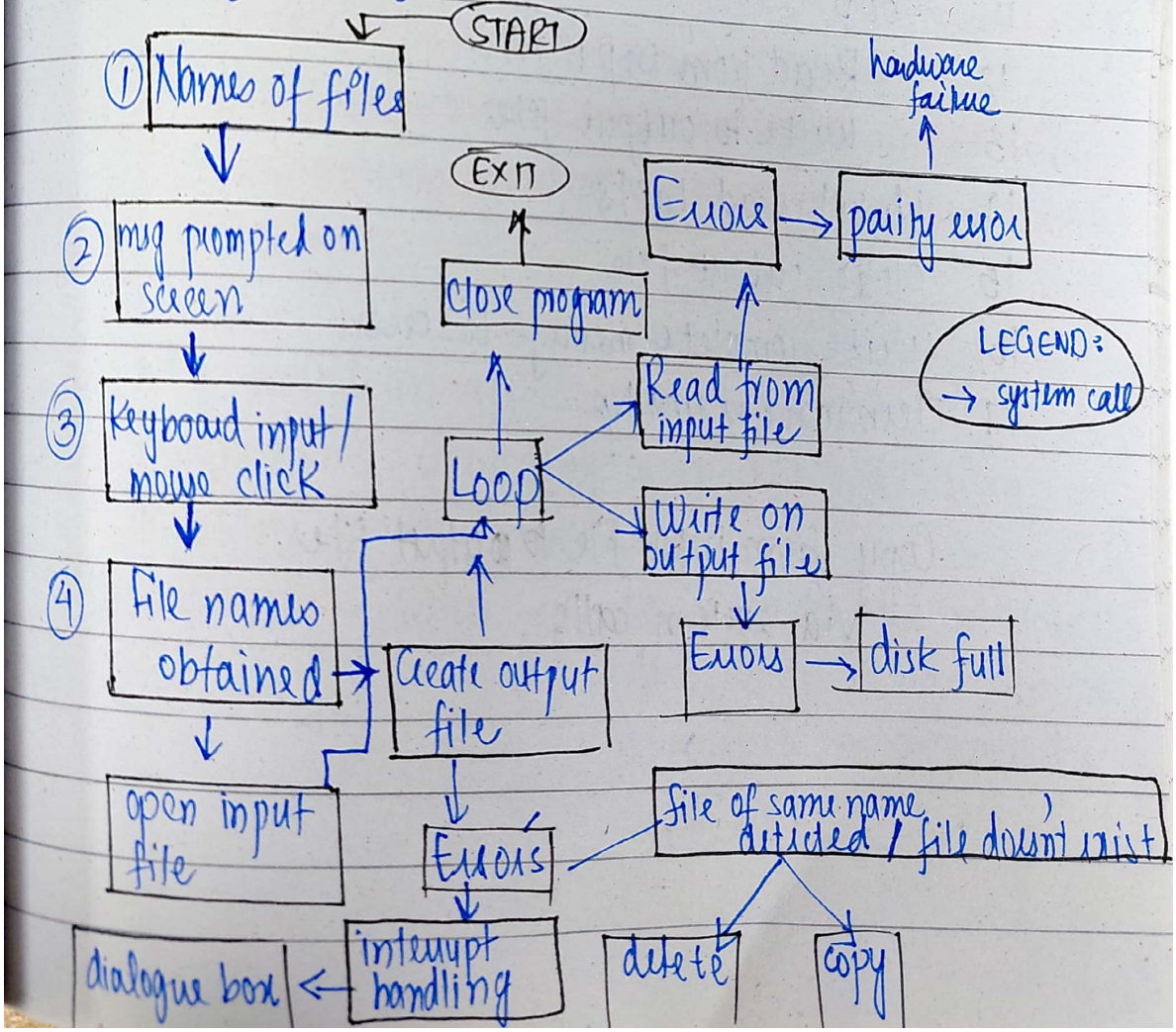
↳ Each read & write status information must be returned regarding error conditions

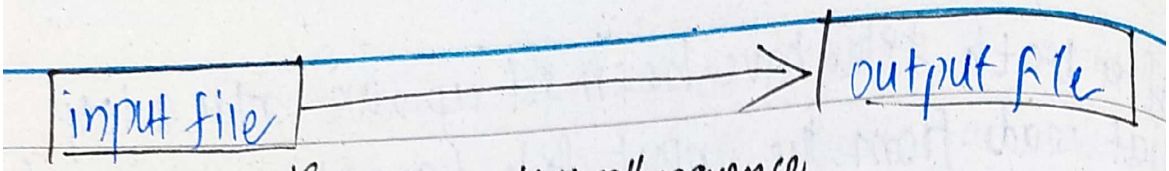
↳ parity error or no more available disk space

hardware failure (input file)

write operation error

→ After copying the contents of the input file, program may close both files (two system calls), write a message to the console (system call) and terminate the program (system call)





Example system call sequence

1 Acquire input file name
2 Write prompt to screen
3 Accept input
4 Acquire output file name
5 Write prompt to screen
6 Accept input
7 Open the input file
8 if (file != exist) { abort ; }
9 Create output file
10 if (file == exist) { abort ; }
11 Loop
12 Read from input file
13 Write to output file
14 Until read fails
15 close output file
16 Write complete message to screen
17 Terminate normally

Copy from input file to output file
via system calls

* APPLICATION PROGRAMMING INTERFACE (API) :

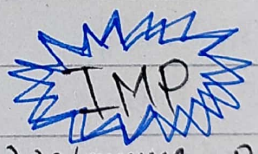
↳ collection of communication protocols and subroutines used by various programs to communicate between them

→ Software intermediary that allows two applications to talk to each other

→ Acts as intermediary layer that processes data transfers between systems, letting companies open their application data and functionality to external third-party developers and internal departments

→ Facilitates programmers with an efficient way to develop their software programs.

→ Provides necessary tools and functions to help two programs or applications to communicate with each other.



Q) Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

① APIs are portable → programs designed on APIs can be compiled and run on any system that supports the same API.

(2) APIs are way easier to work with than actual system calls

* functions that make up an API typically invoke the actual system calls on behalf of the programmer

* Another factor in handling system calls is the run-time environment (RTE) – the full suite of software needed to execute applications written in a given programming language.

→ RTE provides a "system-call interface" IMP

↙ intercepts function calls in the API and invokes the necessary system calls within the OS (eg. NTCreateProcess())

↓ serves as the link to system calls made available by the OS

↓ maintains a table indexed according to a number associated to each system call

↓ invokes the intended system call in the OS Kernel and returns the status of system call.

* To invoke a system call via an API requires parameters. Therefore, three general methods are used to pass parameters to the operating system.

→ Passing of parameters

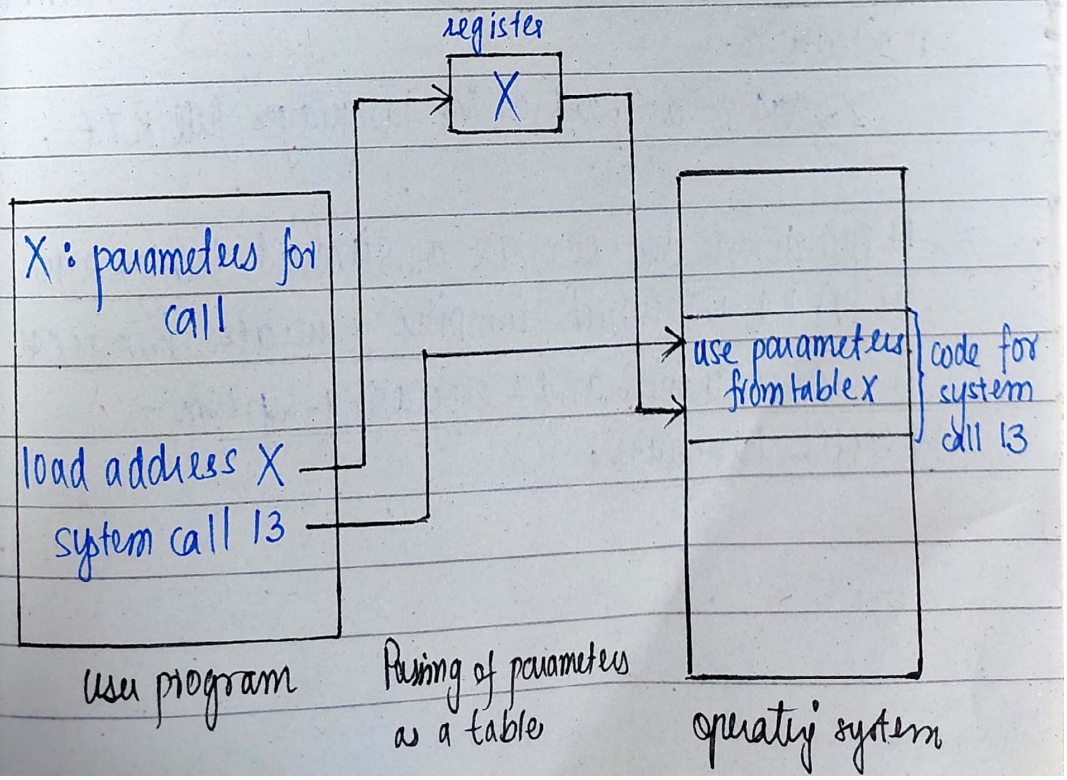
① → pass parameters in registers

② → Create a block for multiple parameters and pass the address of the block in a register

if (no. of parameters <= 5) {
use registers;
}

if (no. of parameters > 5) {
use block method;
}

③ → Push into and pop out of stack



Q Why are applications Operating system specific?

→ applications compiled on one operating system are not executable on other operating systems
→ Each operating system provides its own unique system calls

★ An application can be made available to run on multiple operating systems in one of the three way:

1. Application can be written in an interpreted language that has an interpreter available for multiple operating systems.

2. Application can be written in a language that includes a virtual machine containing the running application.
   ↳ VM is a part of the languages' full RTE.

3. Application dev can use a standard language or API in which compiler generates binaries in a machine-and-operating-system-specific language.

OPERATING SYSTEM STRUCTURE :

① MONOLITHIC STRUCTURE :

↳ Place all of the functionality of the kernel into a single, static binary file that runs in a single address space

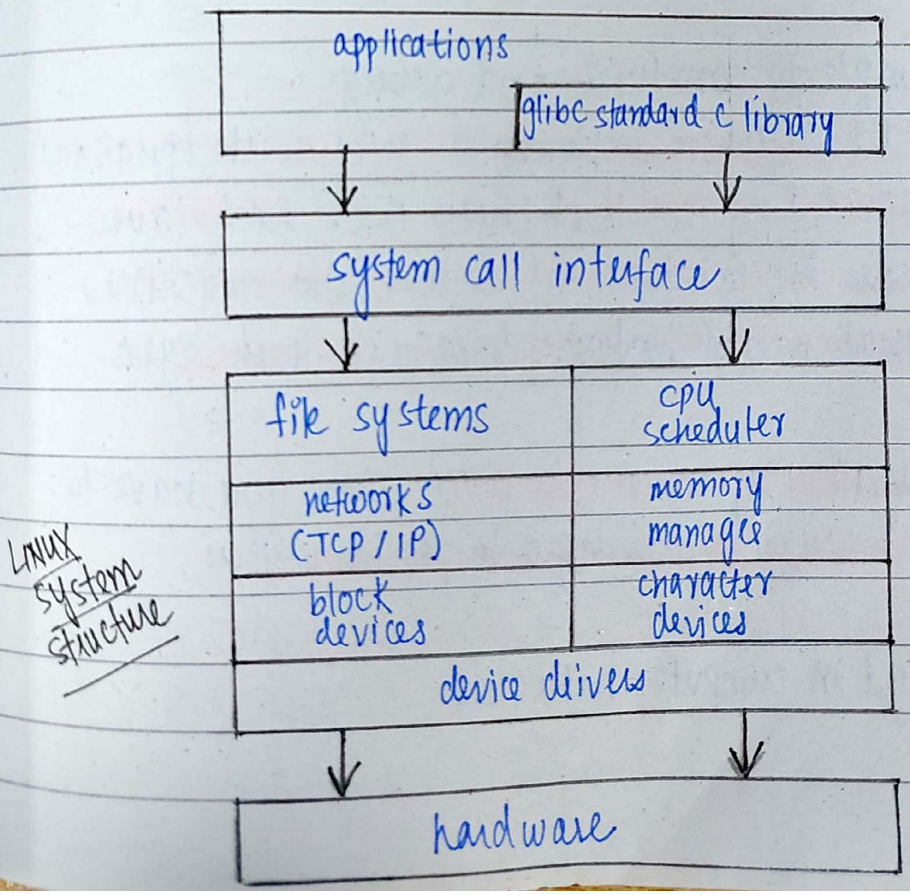
→ UNIX operating system ← Kernel
                          ← system programs

→ Fast and efficient (little overhead in syscall interface)

→ Difficult to implement and extract data

→ Tightly coupled — change in one part will affect all parts

1 file



Linux system structure

(2) LAYERED APPROACH :

↳ loosely coupled – change in one layer will not affect other layers

→ Layer → separate small component that has a specific functionality

↓

abstract object → data + set of functions

→ OS is broken into a number of layers (levels), each built on top of lower layer. The bottom layer (layer 0), is the hardware; the highest layer (layer N), is the interface.

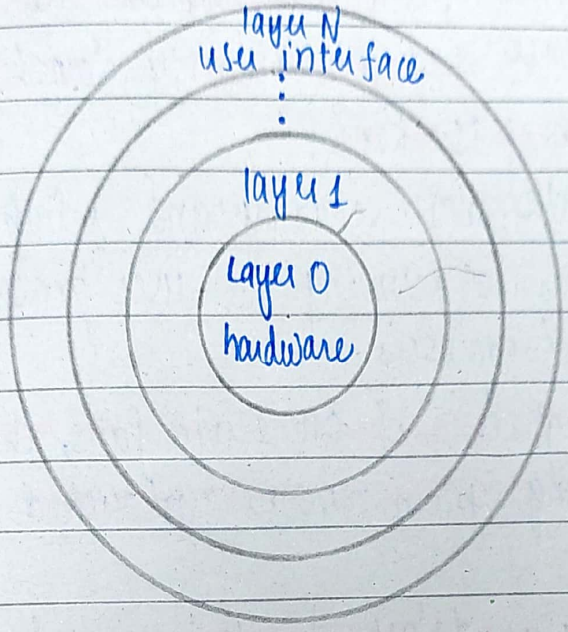
→ Lower layers are invoked by higher-level layers.

→ Simple to construct and debug

→ Each layer is implemented only with operations provided by low-level layers i.e. each layer hides the existence of certain data structures, operations and hardware from high-level layers

→ Overall performance is poor since you have to go through each layer to provide services

→ Used in computer networks



③ MICROKERNEL SYSTEM STRUCTURE :

↳ Most essential components and features of a kernel

→ Removing all non-essential components from kernel and implementing them as user level programs that reside in separate address spaces, hence the result is a small kernel

→ MicroKernels provide minimal process and memory management, in addition to communication facility

→ MicroKernels provide communication between the client program and operating system services via message-passing

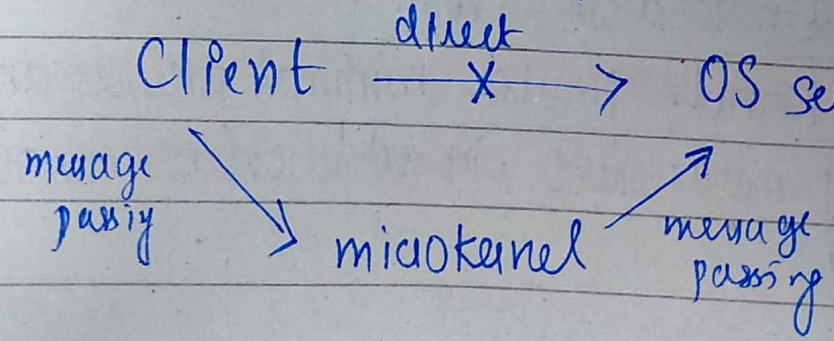
→ Microkernels make extending the operating system easier ; all new services are added to user space and consequently do not require modification of the kernel

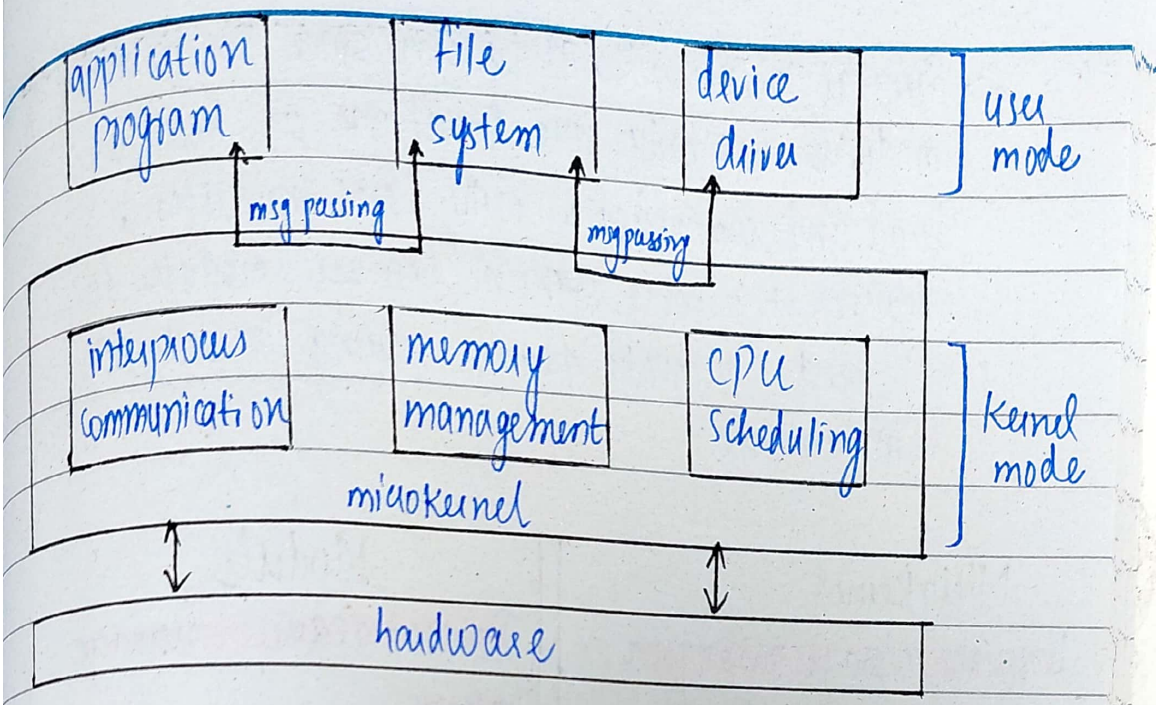
→ Also provide more security and reliability since most services are running as user processes rather than kernel processes.

→ Loosely coupled — if one service fails, the rest of the operating system remains untouched

→ Communication delays due to increased system-function overhead. When two user-level services must communicate, messages must be copied b/w services, which reside in separate address spaces.

— ↳ Processes switch from one process to another to exchange messages





④ MODULES:

↳ Loadable Kernel Module (LKM)

↳ whenever services from user program are required, the module is loaded into the main memory

→ Kernel provides core services while other services are implemented dynamically

→ Similar to layered approach; each kernel section has defined, protect interfaces but it is more flexible than layered system; any module can call any other module

Layered approach	Modules
Lower layers are invoked by higher layers only	any module can invoke any other module

→ Similar to microkernel system since the primary module has only core functions of how to load and communicate with other modules; however it is more efficient because modules do not need to invoke message passing in order to communicate

MicroKernel	Module
Requires message passing in order to communicate	Doesnot require message passing

→ LINUX uses LKMs.

(5) HYBRID SYSTEMS :

↳ most modern operating systems do not adopt a single, strictly defined structure

→ They combine different structures in order to fill the gaps of performance, efficiency and stability left by ~~single structures~~ well-defined structures

→ LINUX → monolithic (efficient)
→ modular (functions dynamically added to kernel)

→ Windows → monolithic
→ microkernel

CHP 03 : PROCESSES

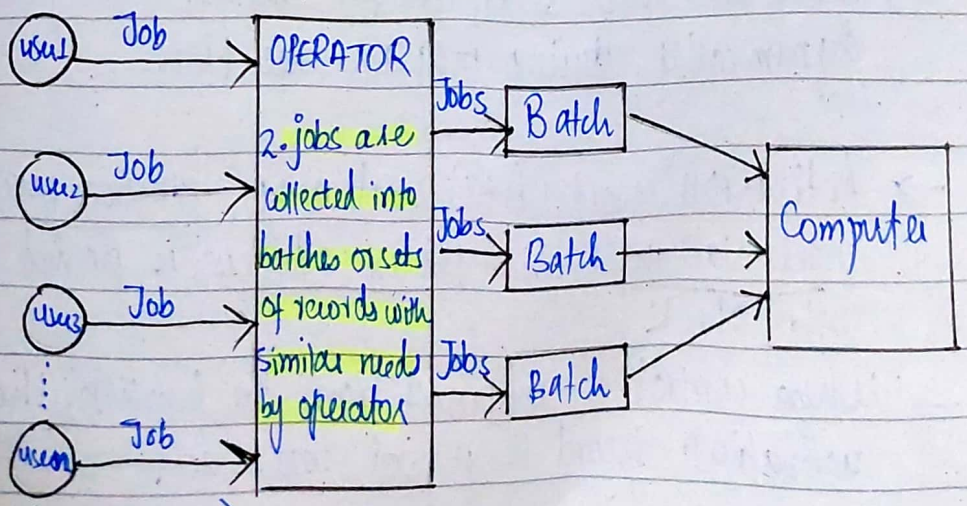
Process → Program in execution

→ Active entity

→ The status of the current activity of a process is represented by the value of the program counter and contents of the processor's registers.

An operating system executes a variety of programs:

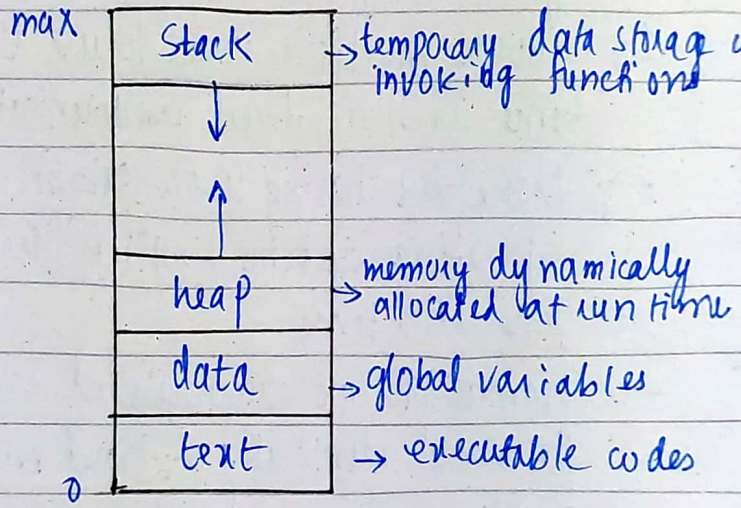
- Batch systems – jobs
  - ↳ allows multiple users to use the OS at the same time without direct communication b/w them.
    - → Done by having users submit their jobs to the operating systems, which then processes them one at a time
    - → Transactions are accumulated over time and processed identically (Travel expenses calculations of employees on a monthly basis



User prepared a job (input data) and submit

- Time shared systems - user programs or tasks
  → Technique which enables many people, located at various terminals, to use a particular computer system at the same time
  → uses CPU scheduling and multi-programming to provide each user with a small portion of a shared computer at once

* MEMORY LAYOUT OF PROCESS:



→ Stack and heap sections can shrink and grow dynamically during program execution

→ Activation record (AR) containing function parameters, local variables and return address is pushed onto stack

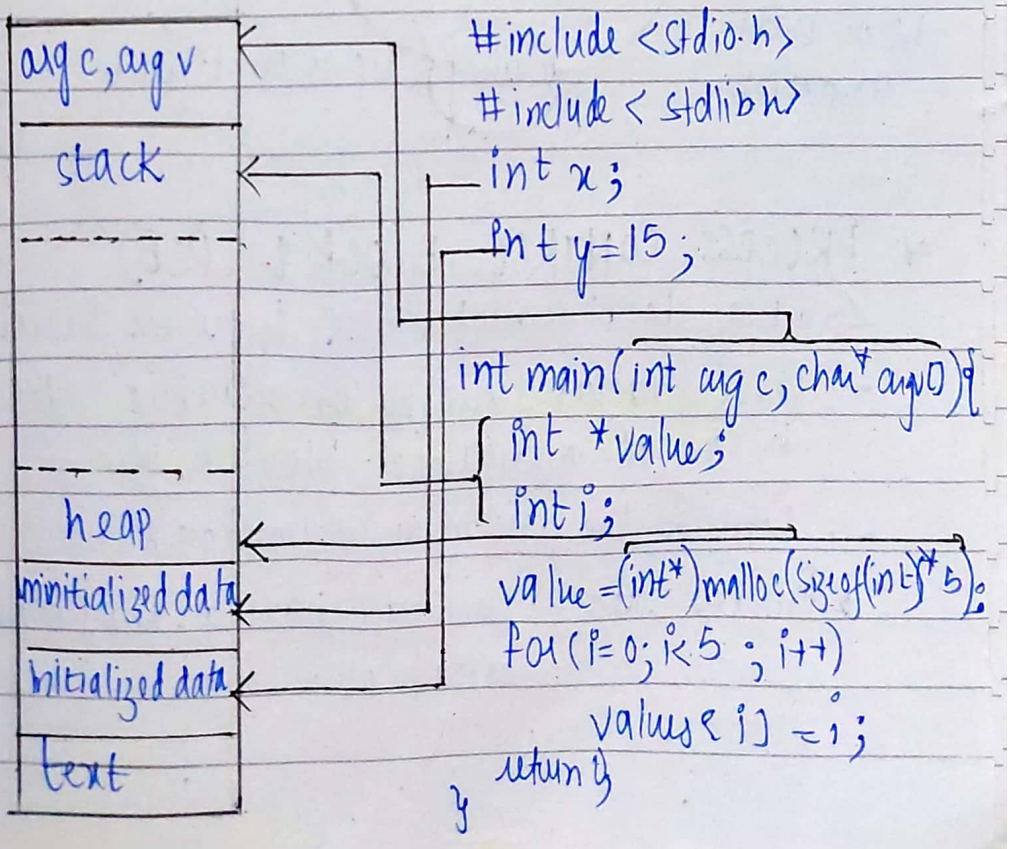
→ When control is returned from the function, the activation record is popped onto stack

Activation Record (AR) → private block of memory associated with an invocation of a procedure

→ used to manage a procedure call at runtime

→ contiguous block of storage that manages information required by a single execution of a procedure

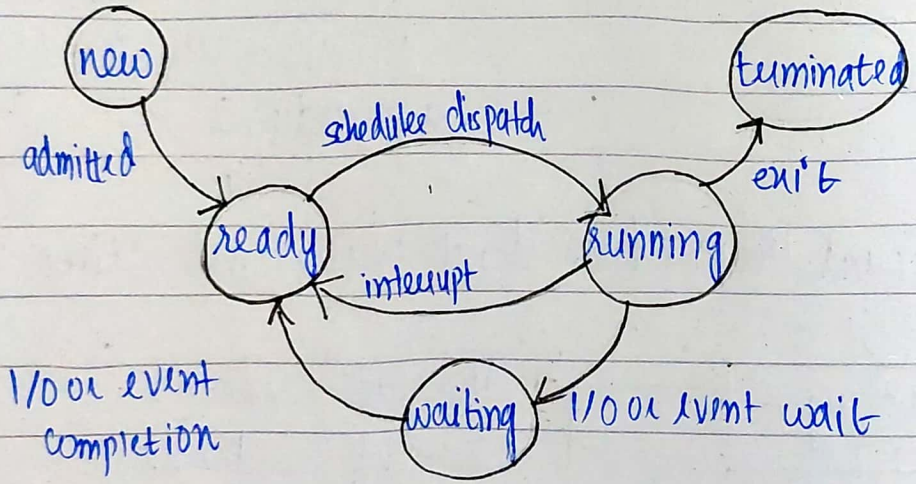
→ Stack and Heap sections donot overlap each other



* PROCESS STATE :

→ State is the current activity of a process

- new : process is being created
- running : Instructions are being executed
- waiting : Process is waiting for some event to occur
- ready : Process is waiting to be assigned to a processor
- terminated : Process has finished execution



* PROCESS CONTROL BLOCK : (PCB)

↳ a.k.a task control block & process descriptor

→ Data structure used by the operating system to store all information about a process

→ Contains the following information :

- Process state - running, waiting, new, ready, halted etc.

- Program counter - Indicates address of next instruction to be executed for this process

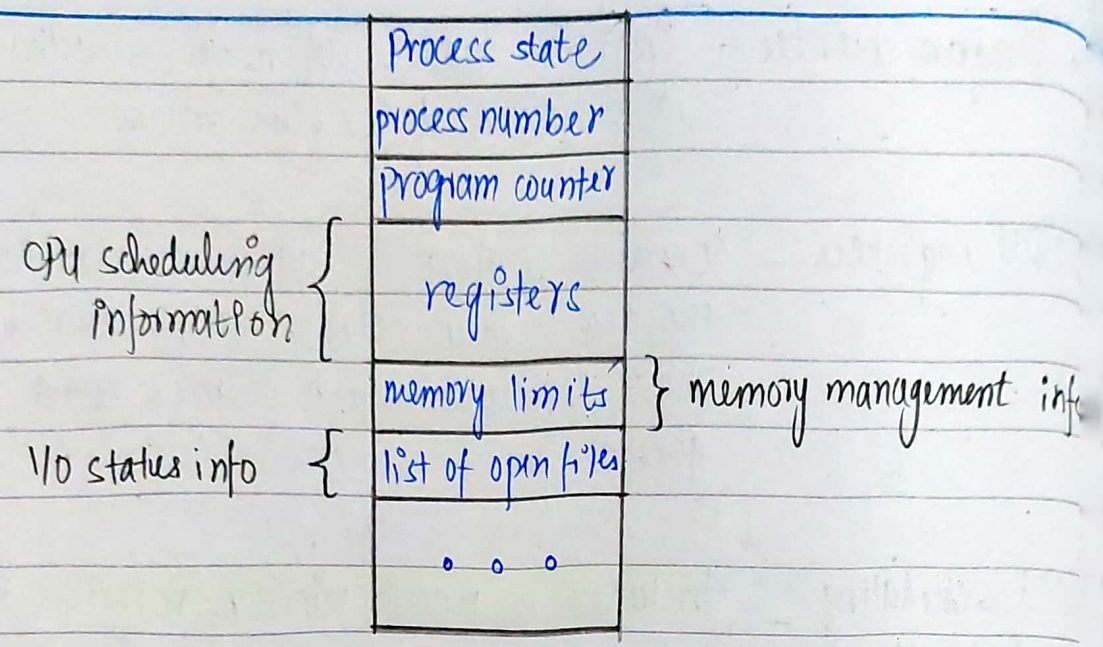
- CPU registers - contains content of all process-centric registers. They include accumulators, index registers, stack pointers and general purpose registers

- CPU scheduling information - Includes a process priority, pointers to scheduling queues and other information eg. process scheduling

- Memory management information - includes
  → values of base and limit registers
  → page/segment tables

- Accounting information - Includes
  → amount of CPU time and real time used
  → time limits
  → account numbers
  → jobs/process numbers

- I/O status information - Includes
  → list of I/O devices allocated to the process
  → list of open files



* PROCESS SCHEDULING :

↳ action of assigning resources to perform tasks

→ handles removal of running process from the CPU and selection of another process on the basis of a particular strategy

→ maximize CPU utilization

→ Objective of time-sharing - switch a CPU core among processes so frequently that users can interact with each program while it is running

- Degree of multiprogramming - Number of processes currently in memory
  - Number of processes in ready state

→ In general most processes can be described as
- I/O bound
- CPU bound

- I/O bound - Process that spends more time doing I/O than it spends doing computations

- CPU bound - Process generates I/O requests infrequently, spends more time in computations

* There are three types of process schedulers:
1. Long term / job scheduler
2. Short term / CPU scheduler
3. Medium term scheduler

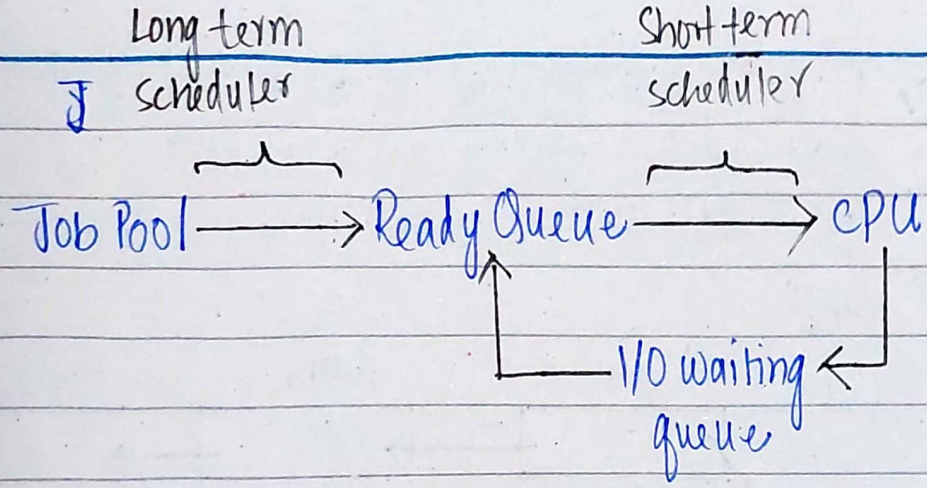
1. Long term / job scheduler:
   - → responsible for bringing processes from the Job queue (secondary memory) into the Ready queue (main memory)
   - → determines which programs to be entered in RAM
   - → controls the Degree of Multiprogramming (no. of processes in ready state at any point in time)
   - → makes careful selection of both CPU bound & I/O bound processes; increases efficiency by maintaining a balance b/w the two

② Short term / CPU scheduler:

→ responsible for selecting one process from ready state for scheduling it on running state
→ only selects process to schedule; doesn't load the process on running
→ ensures there is no starvation owing to high burst-time processes
→ Dispatcher is responsible for loading the process selected by Short term scheduler on the CPU (ready to running state) via
  - Context switching
  - Switching to user mode
  - Jumping to proper location in newly loaded program

③ Medium term scheduler:

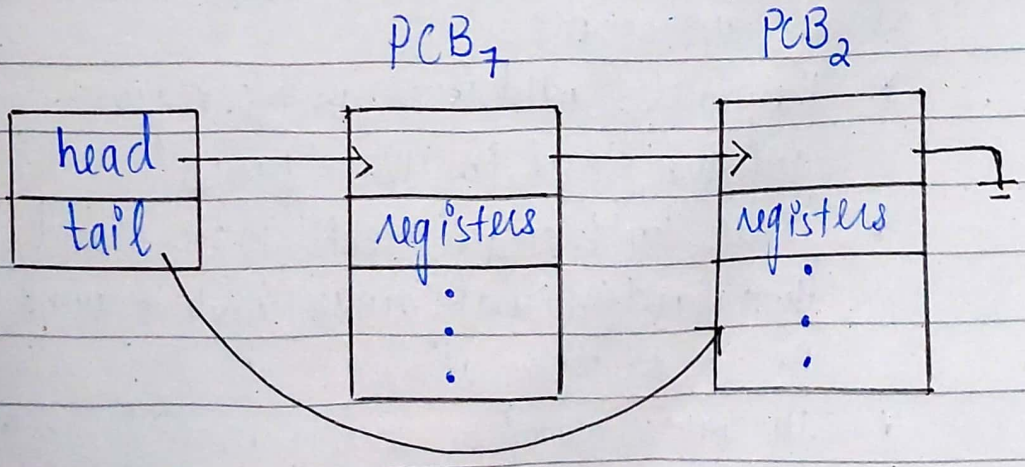
→ responsible for suspending and resuming the process
→ performs swapping to improve process mix or because a change in memory requirements has over committed available memory, requiring memory to be freed up
→ helpful in maintaining balance b/w I/O bound and CPU bound processes
→ Reduces degree of multiprogramming



* SCHEDULING QUEUES:

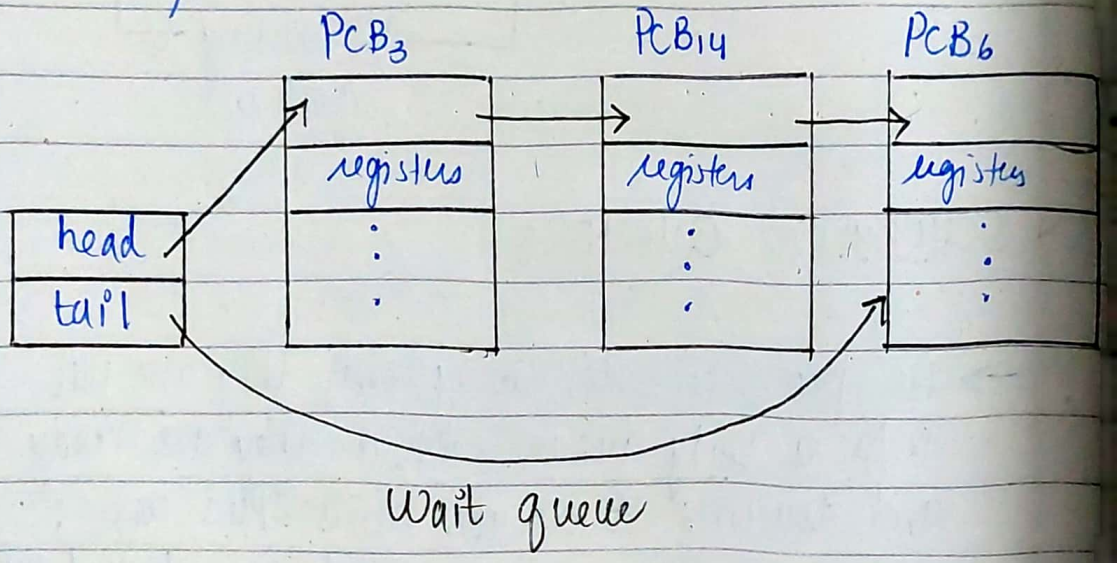
→ As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU's core; this queue is generally stored as a linked list

→ A ready queue header contains pointers to the first PCB in the list, and each PCB contains a pointer that points to the next PCB in the ready queue



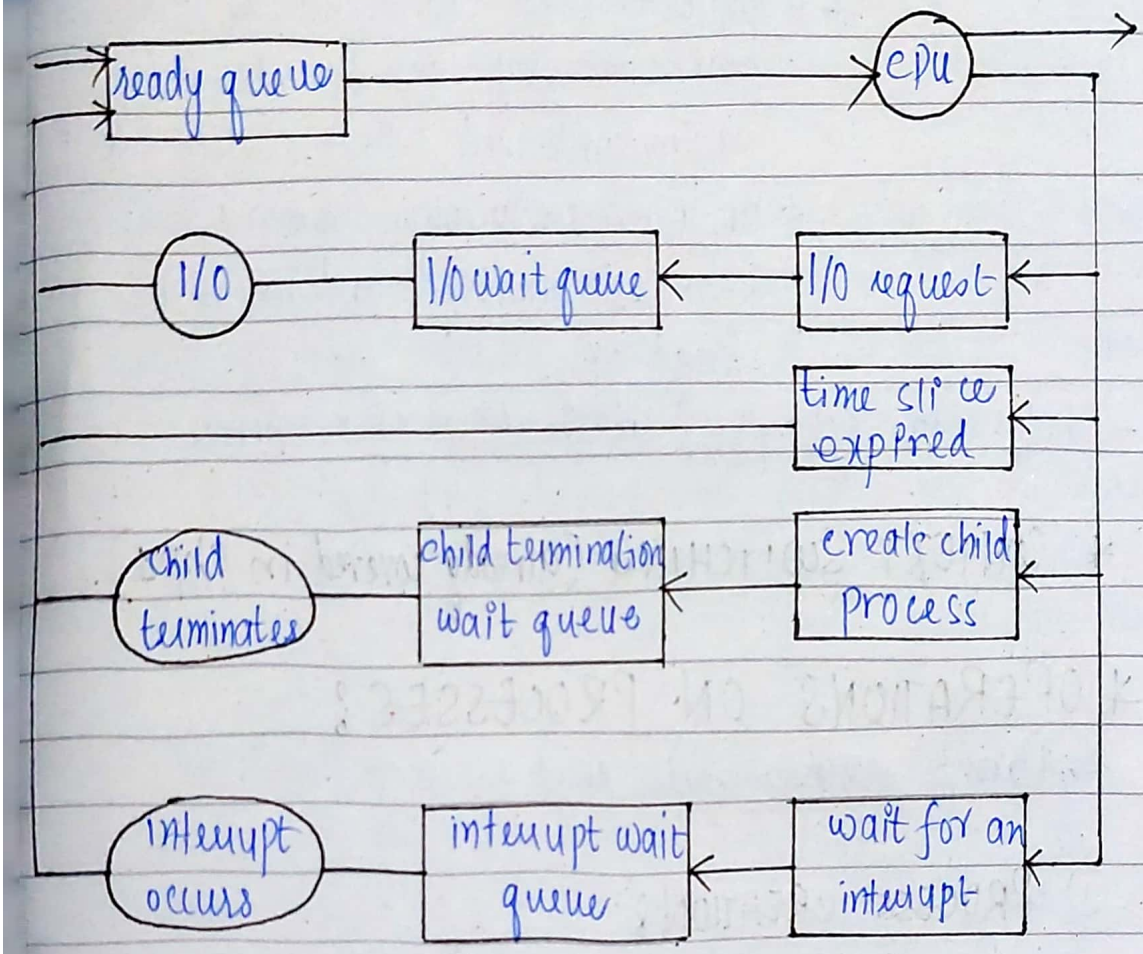
Ready queue

→ Processes waiting for I/O are placed in a wait queue which contains PCBs for jobs that need resource allocation or user input



Wait queue

→ A new process is initially put in ready queue.
→ This process waits there until it is selected for execution, or dispatched
→ Once the process is allocated a CPU core and is undergoing execution, one of the several events could occur:
* The process could issue an I/O request and then placed in wait queue
* Process could create a child process and then placed in wait queue while it awaits the child's termination
* The process could be forcibly removed from the core, as a result of time slice expire or interrupt and be put back in ready queue



Queing diagram of process scheduling

* CPU SCHEDULING :
  - → CPU core selects among the processes that are in ready queue and allocates a CPU core to one of the processes
  - → scheduler is unlikely to grant the core to a process for an extended period of time
    - ↳ forcibly removes the CPU from a process and schedules another process to run
  - → Supports swapping → reduces degree of programming by removing a process from memory

→ Process later reintroduced into main memory, and its execution can be continued where it left off

→ only necessary when memory has been overcommitted and must be freed up

→ ~~helps with~~ optimizes utilization of resources

* CONTEXT SWITCHING (already covered in chp1)

* OPERATIONS ON PROCESSES:

→ Parent processes create child processes

① PROCESS CREATION:

→ Parent processes create child processes

→ Processes identified according to a unique process identifier or pid

↳ provides unique value for each process in the system and used as index to various attributes of a process

→ systemd (pid=1) is the root parent process for all processes

→ systemd creates several processes which provide services

→ loginid is responsible for managing clients that directly log onto the system

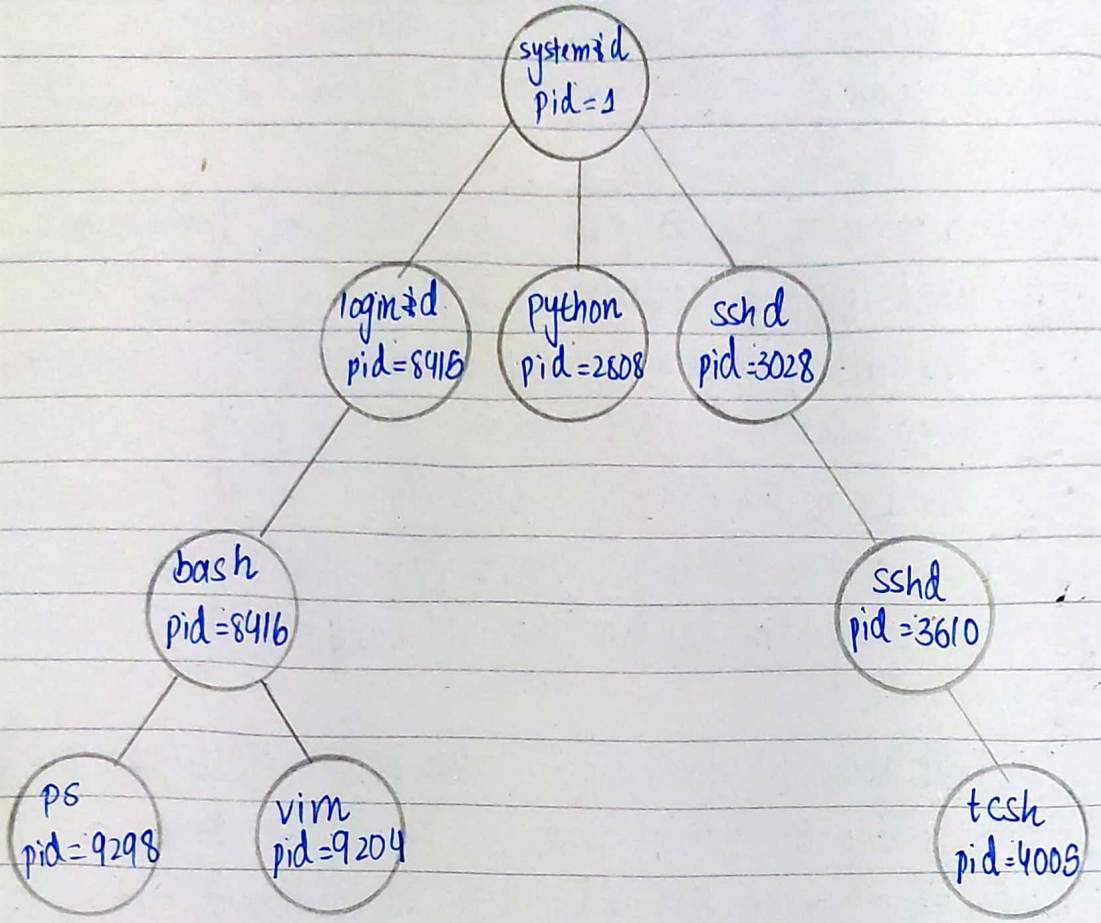
→ sshd process is responsible for managing clients that connect to the system using ssh (secure shell)

→ When a process creates a child process, that child process will need certain resources to accomplish its task
  ↳ obtains resources directly from the OS or constrained to a subset of resources of parent process
    ↳ Restricting child process to a subset of resources prevents any processes from overloading the system by creating too many child processes

→ Initialization of data i.e. input passed to the child process by the parent process eg. file name, in order to open the file and write the contents out.

→ Two possibilities of execution exist when a process creates a new process:
1. The parent continues to execute concurrently with its children
2. The parent waits until some or all of its children have terminated

→ Two address-space possibilities for the new process:
1. Child process is a duplicate of the parent process
2. Child process has a new program loaded into it



→ a new process is created by fork() system call

→ New process consists of a copy of the address space of the original process

  2→ allows parent process to communicate easily with its child process

→ Return code for fork() is zero for the child process whereas the pid (non-zero) of child is returned to the parent

→ exec() syscall is used by either of the processes to

↑ Because exec() syscall overlays the process address space with a new program, exec() doesnot return control unless an error occurs.

---

replace the process's memory space with a new program
↳ exec() syscall loads a .bin file into memory and starts execution

→ wait() syscall issued by parent process to move itself off the ready queue until termination of the child.

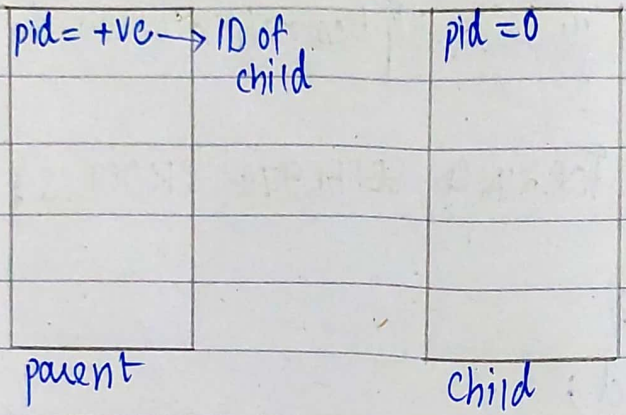
↑

• C-PROGRAM FORKING SEPARATE PROCESS:

```
int main() {
    pid_t pid;
    pid = fork(); /* fork a child process
                     fork() creates a copy of the
                     address space of the defined proces */

    if (pid < 0) {   /* error occured */
        fprintf (stderr, "Fork failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp ("/bin/ls", "ls", NULL);
                  ↳ name of the program to be loaded in child
    }
    else {   /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child complete");
    }
```

→ 2 different processes running copies of same program, except that the value of pid of child process = 0 whereas pid of parent process > 0, which is actual pid of child process.

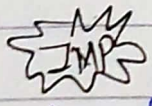


→ Child process then overlays its address space using execlp() command/system call

→ Parent process waits for child process to complete with wait() system call

→ When child process finished, parent process calls exit() system call

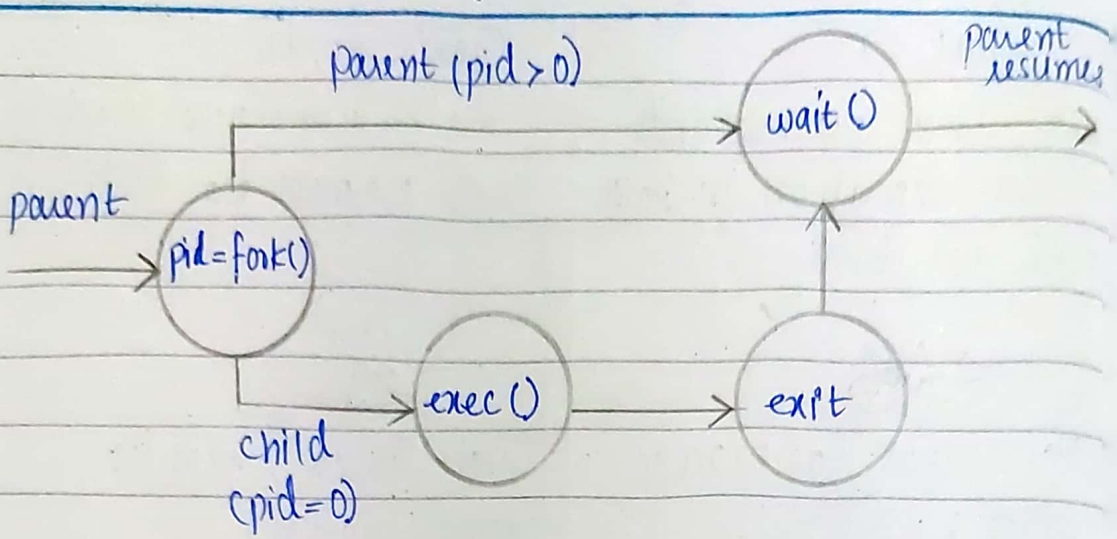
fork()	Create_Process()
• fork() has a child process inheriting address space of its parent	Create_Process() requires loading a specified program into address space of child process
• no parameters passed	• atleast 10 parameters passed

fork()	Create_Process()
• Create a new process and continues execution in both parent and child	• Creates new process and loads program from disk
	
fork() steps for process creation:	Create_Process() steps for process creation:
① Create and initialize PCB in the Kernel	① Create and initialize PCB in the Kernel
② Create new address space	② Create and initialize new address space
③ Initialize address space with a copy of entire contents of address space of parent	③ Copy arguments args into memory in address space
④ Inherit execution context of the parent (eg openfile())	④ Initialize hardware context to start execution at "start"
⑤ Inform the scheduler that the new process is ready to run	⑤ Inform the scheduler that the new process is ready to run

Q How is it efficient to make a complete copy?

→ Copy-on-write – fork() syscall copies the virtual memory map. All of the segments in the segment table are "read-only". If the parent or child edits data in a segment, an exception is thrown and Kernel creates a full memory copy of that

Process creation using fork() system call



② PROCESS TERMINATION:

→ a process terminates when it finishes executing the final statement and asks the OS to delete it by using exit() system call

↳ returns status value (int) to its waiting parent process via wait() system call.

→ all resources are deallocated and reclaimed by the operating system

→ a system call can be invoked only by the parent of the process that is to be terminated

→ A parent may terminate the execution of one of its children for a variety of reasons:

1. Child has exceeded its usage of some of its resource that it has been allocated
2. Task assigned to the child is no longer required
3. Parent is exiting, and the OS doesnot allow a child to continue if its parent terminates

* aka "dead" process. the process' table entry should be removed but it doesnot happen in case of zombie process
* wait() syscall is used for removal of zombie

Some systems follow cascading termination procedure i.e. termination of all child processes if parent process terminates normally or abnormally

→ A process that has terminated, but whose parent has not yet called wait(), is known as "zombie" process *

↳ all processes transition to this state when they terminate, but generally exist as zombies briefly

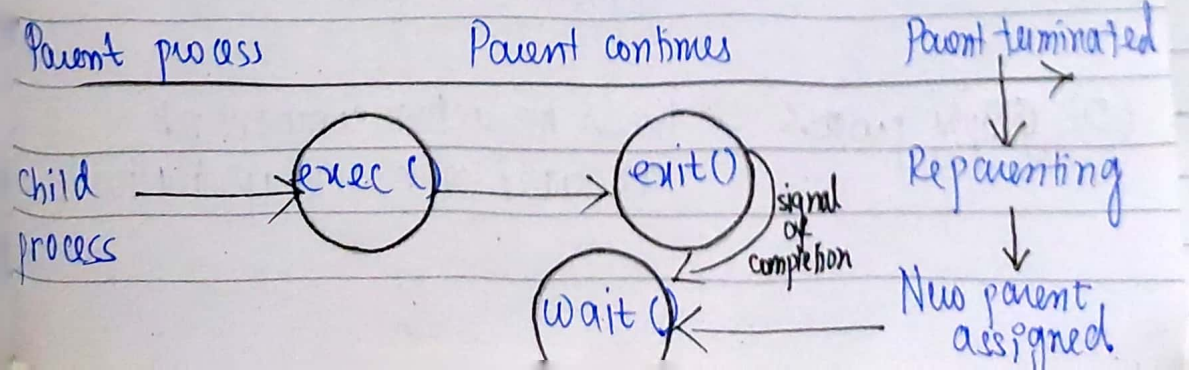
Orphan process — one which is executing but its parent process has terminated

↓

Solution: init process

↓

periodically invokes wait() allowing the exit status of any orphaned process to be collected and releasing the orphans pid and table-entry



* ANDROID PROCESS HIERARCHY :

→ Resource constraints such as limited memory

↓

mobile operating systems terminate existing processes to reclaim memory

↓

Android has defined an "importance hierarchy" of processes → most to least important

↓

① Foreground process – process visible on the screen

② Visible process – Not running on foreground but one that is performing an activity that the foreground process is refering to (status displayed on foreground process)

③ Service process – Apparent to the user

↓

spotify music running in background

④ Background process – Performing an activity but not apparent to user

⑤ Empty process – Holds no active components associated with any application

* Rendering – process that turns website code into interactive pages users see when they visit a website

---

Android terminates empty processes first

. Multi-process architecture – Chrome Browser:

→ Many browsers adopted the idea of opening several websites at once, each with a separate tab.

↳ If web application in any tab crashes, entire process crashes with it

↓ solution

Google Chrome Web browser used multi-process architecture

↓

Chrome checks for 3 types of processes:

- Browser – responsible for managing the user interface as well as disk and network I/O.

- Renderer – contains logic for rendering web pages – logic for handling HTML, JS and many more.
  A new renderer process is created for each website opened in a new tab so several renderer processes may run at the same time

* Plug-ins are web browser extensions
↓
software module for customizing a web browser

- Plug-in - process created for each type of plug in (web browser extensions used to manage Internet content that a browser is not designed to process)

→ Plug-in processes contain code for plug-in and code used to communicate with associated render processes and browser processes.

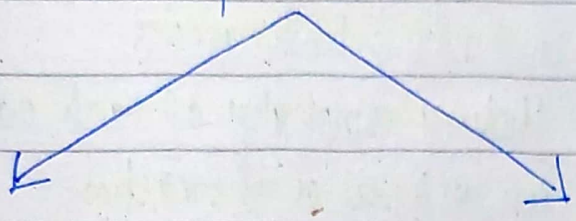
→ Advantage of multiprocess architecture is that websites run in isolation from one another. If website crashes, only its renderer process is affected; all other processes remain unharmed.

→ Renderer processes run in a "sandbox", which means that access to disk and network I/O is restricted, minimizing effects of security exploits.

Sandbox - Isolated testing environment that enables users to run programs or open files without affecting the application, system or platform on which they run
- Effort to mitigate/reduce system failures and software vulnerabilities from spreading.
- Prevents zero-day attacks
- Enables hybrid solutions
- Gain access to advanced networking and support

* INTERPROCESS COMMUNICATION:

Concurrency of executing processes



Independent
- process doesnot share data with any other process executing in system
- Data can't be shared

Cooperating
- process that can or can be affected by other processes executing in system
- Data sharing b/w processes

- Reasons for providing cooperating IPC environment:

① Information sharing – • Several apps interested in same type of data (ctrl+C, ctrl+V)
• Use of APIs to connect two apps for effective communication

② Computation speedup – • Tasks broken down into subtasks each of which executed in parallel with other for faster computational speeds
• Can be achieved if computer has multiple processing cores

IMP

③ Modularity - Property that describes how replaceable the components or modules of a system are
- Reduce complexity by breaking a system into varying degrees of interdependence and independence across
- Hiding complexity of each part behind an interface or abstraction
- System divided into separate processes or threads

→ Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data — send & receive data.

* There are 2 fundamental models of IPC:
① Shared Memory
② Message passing

① Shared Memory - a region of memory that is shared by the cooperating processes is established
- Two or more processes read and write to a shared selection of memory
- Used for communication between processes where communicating processes reside on the same machine
Communicating processes share a common address space

2) Message-Passing - Communication takes place by means of messages exchanged between cooperating processes

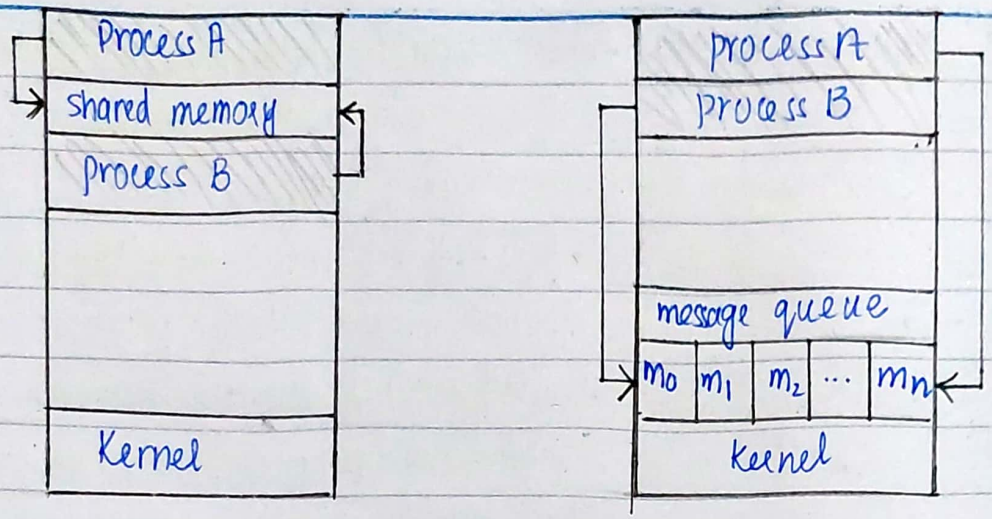
- Packets of information in predefined formats moved between processes
- Used in a distributed environment where communicating processes reside on remote machines connected through a network

→ Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

→ It is easier to implement in distributed system than shared memory

→ Shared memory can be faster than message-passing, since message-passing systems are implemented using system calls and consumes more time

→ In shared-memory systems, system calls are required only to establish shared-memory regions.



shared memory model

message passing model

* IPC IN SHARED-MEMORY SYSTEMS :

↳ communicating processes establish a region of shared memory

→ Shared memory resides in address space of process creating the shared-memory segment

• Producer-Consumer Problem :

→ a.k.a bounded-buffer problem

→ trying to achieve synchronization between more than one process

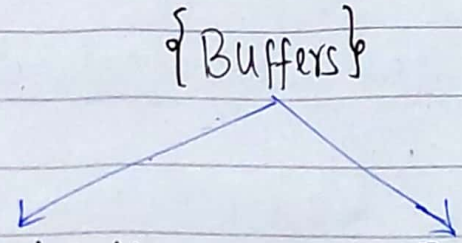
→ Producer produces information that is consumed by consumer process

→ Same memory buffer is shared by both producers and consumers

→ Producer produces an item, puts into a memory buffer, and a consumer consumes it from the memory buffer

→ provides a useful metaphor for client-server system

→ Producer and consumer must be synchronized so that consumer doesnot try to consume an item that has not yet been produced



Unbounded buffer

- No practical limit on the size of buffer
- Producer continuously produces new items

Bounded buffer

- Assumes a fix buffer size
- Consumer will wait until buffer is empty, and the producer must wait until buffer is full

. Implementation of Producer and Consumer :

```
#define BUFFER_SIZE 10
typedef struct {
   ...
} item
item buffer[BUFFER SIZE];
int in = 0;
int out = 0;
```

→ Shared buffer is implemented as a circular array with two logical pointers : in and out.
→ 'in' points to the next free position in the buffer
→ 'out' points to the first full position in the buffer

Producer Implementation:

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in+1) % BUFFER_SIZE == out) ;
        /* do nothing */        // buffer full)

    buffer[in] = next_produced; // put new item in next slot of buffer
    in = (in+1) % BUFFER_SIZE;
```

→ buffer is ~~empty~~ full when ((in+1 % BUFFER_SIZE == out))
→ buffer is empty when in == out

Consumer Implementation:

```
item next_produced;
while (true) {
    while (in == out) ;        // buffer empty
        ; /* do nothing */

    next_consumed = buffer[out]; // take out item from buffer
    out = (out+1) % BUFFER_SIZE;
    /* consumes the item in next_consumed */
}
```