OPERATING SYSTEM

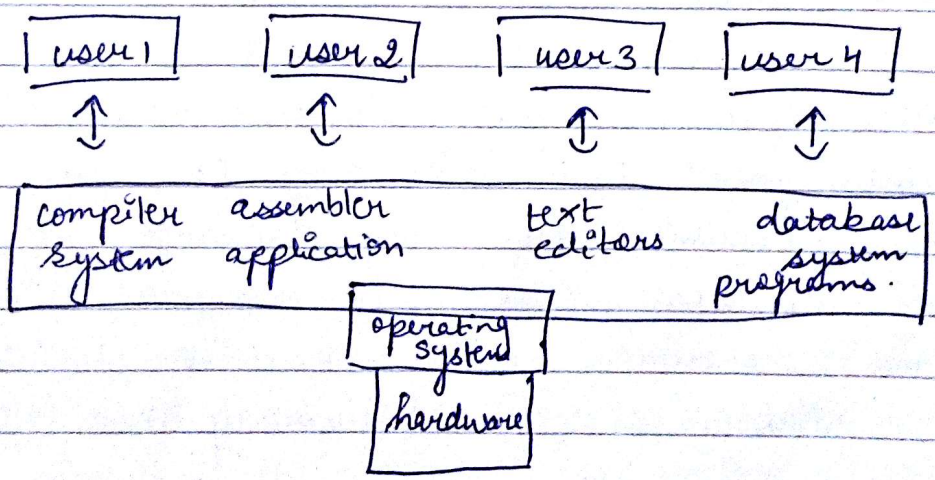
* OS acts as a intermediary b/w the ~~users~~ users of a computer and the computer hardware.

* It is a program that manages the computer hardware.

* OS are designed to optimize the utilization of hardware.

Role of operating system in computer system.

1) A computer system can be divided into 4 components
   a) hardware
   b) operating system
   c) application programs
   d) users.



The hardware - CPU, memory, input/output devices provides the basic computing resources for the system.

The application programs such as word processor, spread sheet, compilers, web browsers define

Scanned by CamScanner

the way in which these resources are used to solve users computing programs.

→ The operating system ~~com~~ controls the hardware and coordinates its use among various applications for the users.

OS from user point of view :-

The users view varies / differs acc to the interface being used. Many users sit in front of a PC such as a system is designed for one user. The goal is to maximise the work that the user is performing, in this case the OS is designed for ease of use with little attention paid to performance and least to resource utilization.

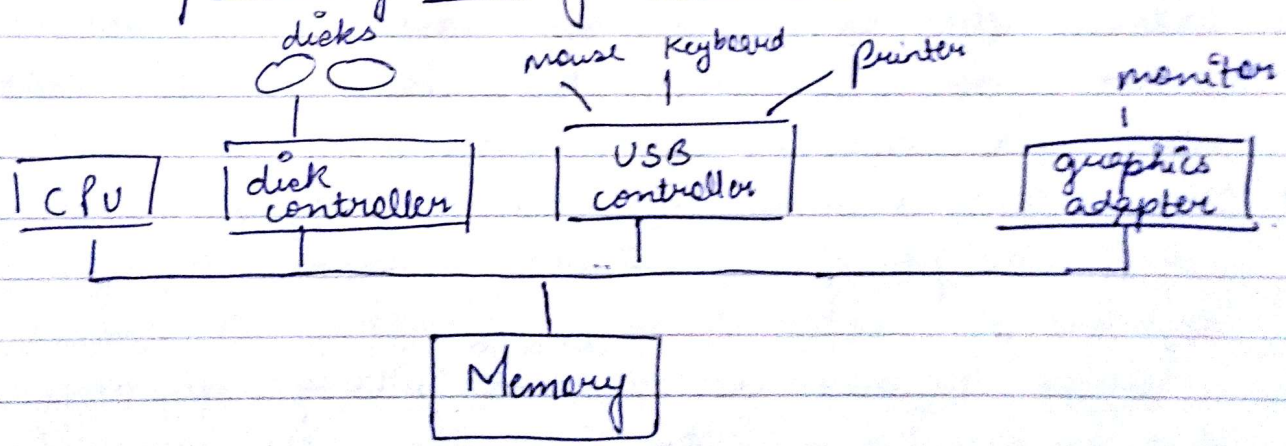
In other cases a user sit at a terminal connected to a main frame or a mini computer. These users share resources and may exchange information the OS in these cases is designed to maximise resource utilization i.e to ensure all available CPU time, memory and I/O devices are used efficiently.

Scanned by CamScanner

OS from system point of view :-

From system point of view, OS is a program involved with the hardware it acts as a resource allocator as well as resource manager.
The OS decides how to allocate the available resources to specific programs and users so that it can operate the computer system efficiently.
~~OS also acts as re~~

Computer System Organisation



- Modern computer system consists of one or more CPU's and a no. of device controllers connected through a common bus that provides access to shared memory.

- The CPU and the device controller can execute in parellel competing for memory cycles.

Scanned by CamScanner

- When a computer is powered up, it needs boots strap program ~~str~~ stored within ROM which initialises all aspects of the system from CPU registers to device controllers to memory contents.
- Boot strap program must know how to load the OS and how to start executing the machine.
- To accomplish this goal the boot strap program must locate the operating system kernel and loaded into memory. Once the kernel is initiated it can start providing services to the system and its user.
- init is the first system process that starts other daemons (mini threads). Once this is done the system waits for some event to occur.
- The ~~occur~~ occurence of event is signalled by the interrupt from either hardware or software.
- Hardware ~~w~~ may trigger an interrupt at any time by sending a signal to the CPU using the system bus and software may trigger an interrupt by a special operation called system call.

When CPU is interrupted it stops whatever it was doing and ~~imdiat~~ ~~imm~~ immediately ~~immediately~~ transfer ~~transfer~~ the execution to a fixed location where service routine for the interrupt is located.

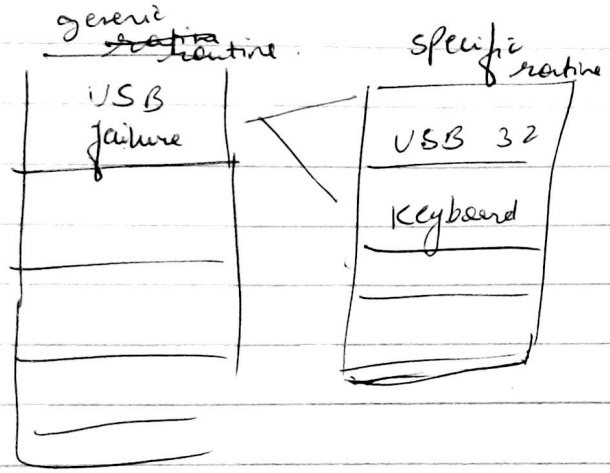
Scanned by CamScanner

Each computer design has its own ~~interrupt~~ interrupt mechanism.

Interrupt → Interrupt service routine

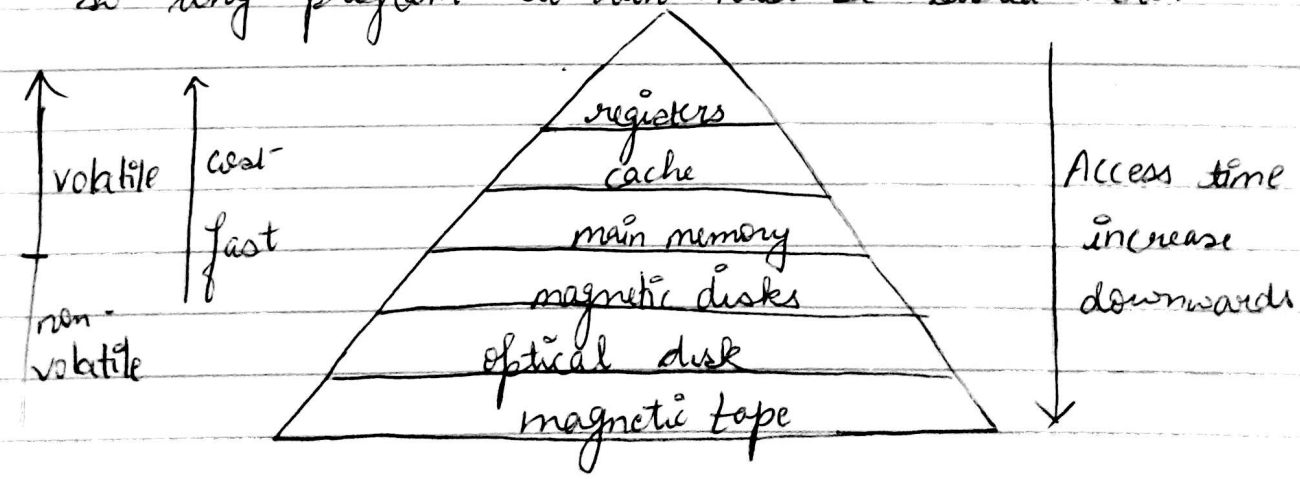
↓

generic routine → Interrupt specific routine.



Rough.

→ The CPU can load instructions only from memory, so any program to run must be stored there.



- Each byte has its own address and the interaction is achived through a sequence of load and store instructions.

Scanned by CamScanner

Load - moves the word or byte from main memory to an internal register within the CPU.

Store - move the content of a register to main memory.

INPUT/OUTPUT STRUCTURE

- A large portion of OS code is dedicated to managing input output because of the varying nature of devices & for the importance of reliability and performance of the system.

- General purpose computer system consists CPU's and multiple devices controler that are connected through a common bus.

The device controller is responsible for moving the data b/w the peripheral devices that it controlled and its local buffer storage.

OS have a device driver for each device controller which provides an OS a uniform interface to the device.

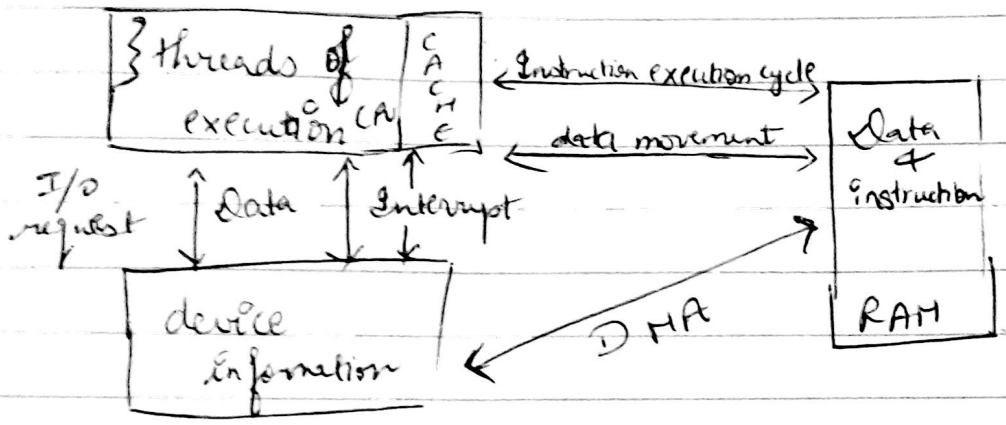
To store I/O operation, the device driver loads appropriate registers within device controller. The device controller then examines the contents of these registers to determine what action to take.

The controller starts the transfer of data from device to its local buffer.

Scanned by CamScanner

Once the transfer complete the device controller informs the device via interrupt that it has finished its operation.

The device driver then returns the control to the O.S.



CSA

Categorisation acc to the no of processors used

- Single processor System : There is only one main CPU capable of executing a general purpose inst. set including inst. from users processes.
  All single processor systems may have other special purpose processors in the form of I/O or device specific processor. These special processors are managed by the OS, it sends them info about next tasks and monitors their status.

Scanned by CamScanner

ADVANTAGES:-

Faster, more reliable, sharing a data, less wastage, increased throughput, economy of scale.

2) Multiprocessor systems:-

Also called parallel systems or multicore system. Such systems have 2 or more processors sharing the computer bus, clock, memory & ~~pheri~~ peripheral devices.

Advantage - 1) economy of scale (Saving memory)
2) throughput increase
3) Reliability increase

Graceful degradations: The ability to provide service proportional to the level of surviving hardware.

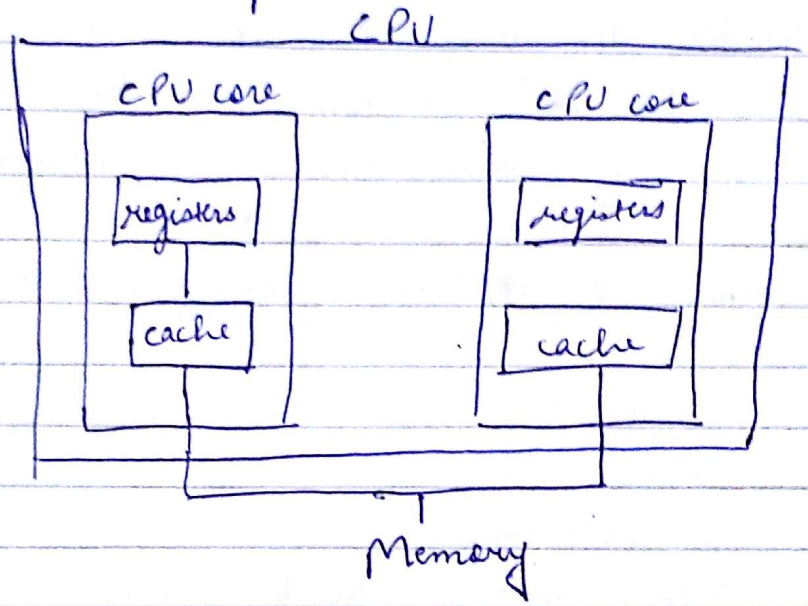
MSS are of 2 types

1) Asymetric multi processors

In this, each processor is assigned a specific task. A boss processor controls the system whereas other processors follow the boss or have predefined task.

Scanned by CamScanner

2) A ~~decent~~ recent ~~friend~~ trend in CPU design is to include multiple computing cores in a single chip. Therefore, they are known as multicore system. These are more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication, even power consumption is also less.



dual core

3) clustered systems

It is another type of multiprocessor system which gathers together multiple CPU's they consists of 2 or more individual systems. They are loosely coupled

Each node may be a single processor or multicore processor. These systems share memory and are closely linked via LAN.

The clustering is used to provide high availability service. It also increases reliability.

Scanned by CamScanner

A clustered software layer runs on cluster nodes each node can monitor one or more nodes if the monitered node fails the monitoring machine takes the job of its storage and restart its applications.

Clusters can be used to provide high performance computing.
Parallelisation divides the program into seperate components that run on individual computers parallelly.

Operating System Structure

OS provides an environment within which programs are executed. ~~Ability to~~

1) Ability to multiprogram - A single program cannot keep the CPU or i/o devices ~~no~~ busy at all times. Single users have multiple programs running. Multiprogramming increases CPU utilisation by organising jobs so that CPU always has one job to execute. OS keeps several jobs in memory simultaneously. Main memory is too small to accomodate all jobs so jobs are initially kept in disks in the job pool. This pool contains all the processes waiting for the allocation of the main memory. OS fix and begins to execute one of the jobs in memory. The jobs may have to wait for some tasks such as i/o operation to complete.

Scanned by CamScanner

In non-multiprogram system CPU remains idle ~~whereas~~ whereas in multiprogram system the OS switches to ~~an~~ execute another job.
Multiprogramming systems provides an environment in which various system resources are utilized effectively.

2) Time sharing and multitasking.

In this the CPU executes multiple jobs by switching among them but the switches occur so frequently that the users can interact with each program while it is running. In this user gives instructions to the program directly by using input devices and waits for immediate result by output device
Time shared OS ~~users~~ uses CPU scheduling and multi-programming.

A time shared OS allows many users to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given the impression that the entire system is dedicated to his job.
Time sharing & multiprogramming req that several job be kept simultaneously in memory.
If several jobs are ready to be brought into memory & if there is not enough room for all of them.

Scanned by CamScanner

Operating system operations

If there is nothing to do the OS will sit quietly and wait for ~~its~~ some event to occur. Events are signalled by interrupts or trap or exception. A trap is a software generated interrupt caused either by an error (division error or invalid memory access) or a request from a user program that an OS service may be performed.

For each type of interrupt a code segment is written in the OS that determines what action is to be taken.
We need to make sure that an error in a user program could cause problem only for that particular user without hampering other user programs.

Dual mode of multi-mode operation.

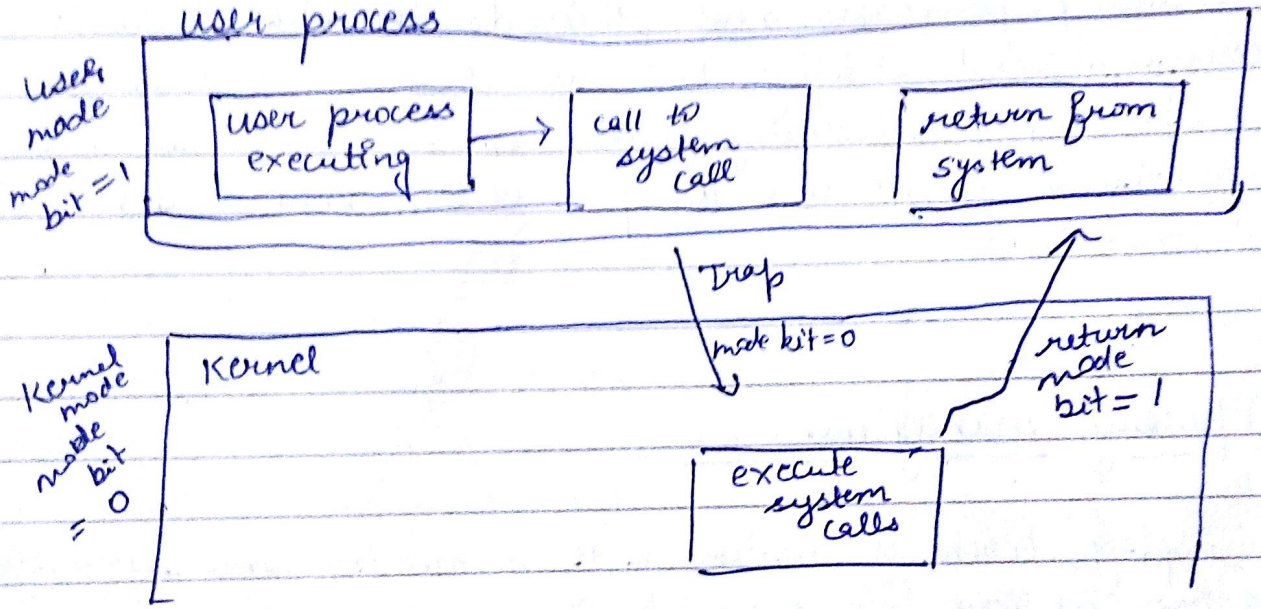
There are 2 modes of operation.
1) user mode 2) kernel mode. (priviledged mode)

A bit called the mode bit is added to the hardware of the computer to indicate the current mode, for kernel it's 0 and for user it is 1.

With mode bit we can distinguish b/w a task that is ~~exe~~ executed on ~~behalf~~ behalf of OS or on user

When a user application requests a service from the OS a system call is raised and the mode switches to kernel.

The dual mode of operation provides us with the means for protecting the OS from user errors.



Timer

OS maintains a control over the CPU. We cannot allow a user program to stuck in a infinite loop or to fail to call system services and never return the control to the OS.

Timer can be fixed or variable. A variable timer can be implemented by a fixed rate clock and a counter. The OS sets a counter and every time the clock ticks the counter is decremented & when the counter reaches

Scanned by CamScanner

0, a interrupt is raised.

Ch:- Functions of OS.

• Process Management

The OS is responsible for the following activities :-
1) scheduling processes and threads on the CPU
2) creating and deleting both user & OS processes
3) suspending & resuming processes.
4) providing mechanisms for process syncronization & communication.

• Memory management

1) Keeping track of which parts of memory are currently being utilised and who is using it.
2) deciding which processes and data to move into and out-of-memory.
3) Allocating & deallocating memory space as needed.

• Storage management

1) File system management -
   a) creating and deleting file.
   b) supporting primitives for manipulating files & directories.
   c) mapping files onto secondary storage.

Scanned by CamScanner

d) backing up files on stable storage ~~s~~ media

2) Mass-Storage management -
Free space management storage allocation disk scheduling

Caching
~~Information~~ is ~~kep~~ Info is kept in storage system. As it is used it is popped into faster storage system i.e the cache memory on temporary basis when we need a particular data or info we first check whether it is available in cache. If it is present then we use the info directly from it and if not we bring it to the cache and use it.

Input/Output Systems

OS hides the peculiarities of specific hardware devices from the user with the help of i/o sub-system which consists of following components. a) First in memory management component that includes buffering, caching and spooling.
b) A general device driver interface
c) device driver for specific hardware

• Protection & security
In multi-user environment several processes execute ~~concur~~ concurrently & access the same data. The access to data is regulated by ensuring that files, memory segments, CPU and other resources can be

Scanned by CamScanner

operated by only those processes that have gained ~~only~~ proper authorisation from the OS

Protection is a mechanism for controlling the access of processes or users to the system resources. It is the job of security to defend a system from internal & external attacks. Such attacks include viruses, worms, denial of services (DOS), identity theft & theft of service.

Computing environments

1) i) Mobile computing

It refers to computing on hand held smart phones & tablet computers. Today mobile systems are used not only for e-mail and web-browsing but also for playing music & videos, reading digital book, taking photos, recording HD videos, & using GPS

excelerometers (sensor) ~~accelerometers~~, ~~Vio~~ giarometers. (used to change the direction) and the (speed)

The memory capacity & processing speed of mobile devices are limited. Power consumption is a concern because mobile devices used processors that are smaller, slower and use few processing codes.

ii) Distributed system

A distributed system is a collection of physically seperate heterogeneous computer systems that are networked to provide users with access to the

Scanned by CamScanner

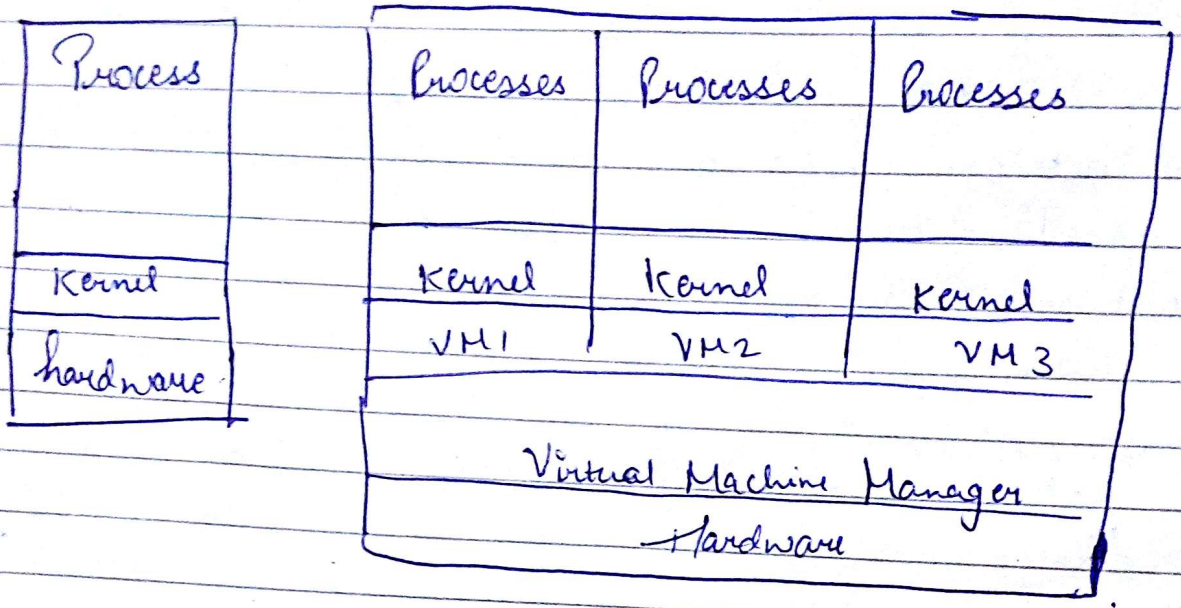
various resources that the system maintains.
Access to a shared resource increases computation speed functionality, data availability & reliability.

A network is a communciation path ~~which~~ b/w two or more parts which is responsible for their functionality
Networks may vary by protocols used, the distance b/w 2 nodes and the transport media
in networked OS is an OS that provide features such as file sharing across the network along with comm. scheme which allows different processes on different computers to exchange messages.

3) Client-server computing
Todays systems act as a server system to satisfy requests generated by client systems
Servers systems can be categorised as compute server system & file server system.
Compute server system provides an interface to which a client can send a request to perform an action in response the server executes the action and sends the result to the client.
File server system provide a file system interface where clients ~~can~~ create update read and delete file.

Scanned by CamScanner

Virtualisation



Chapter 2 Operating System Structures

System programs are also known as system utilities. It provides a convenient environment ~~from~~ for program development and execution. They are divided into following ~~4~~ categories —

1) file management - These programs create delete copy, print, rename, list & manipulate files and directories.
2) Status information - some programs provide info like date, time, amount of available memory, no. of users, performance, logging & debugging info. In some systems registory is created to store and retreat configuration information.
3) file modification - some text editors create and modify the contents of file stored on disk and storage devices.

Scanned by CamScanner

Nidhi

4) Programming language support - compilers, assemblers, debuggers and interpreters for programming languages are available.

5) Program loading & execution - the system provides absolute loaders, relocatable loaders, linking editors, overlay loaders to load the assembly code into memory for execution.

6) Communication - These programs provide the mechanism for creating virtual connections among processes, users and computer systems to send messages to one other streams, to browse web pages, to send e-mails, to login remotely or to transfer files from one machine to another.

7) Background services - some of the processes terminate after completing their tasks while others run until the system is haulted.

Operating system design & implementation

Problems faced in designing and implementing of OS. —

1) To define goals and specifications

At the highest level the system design will be affected by the choice of hardware and the type of systems - batch system, real-time system, multiuser, single user, time sharing, distributed system etc.

2) Mechanisms & policies - Mechanisms determine how to do something. Policies determine what will be done. Policies are likely to change overtime. Each change in policy would require

Scanned by CamScanner

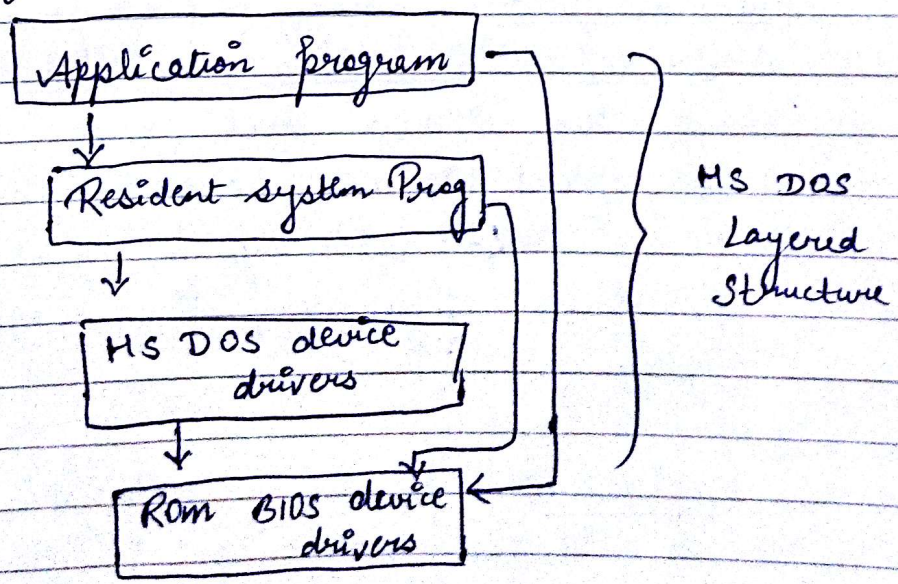
Windows introduced
- when
- which
- ~~linux~~ Unix, Linux

a change in the mechanism.

3) Implementation - Once an OS is designed it must be implemented. Early OS were written in assembly languages but nowdays OS is written in high level languages like C & C++. An OS can be written in more than one language eg: kernel can be written in assembly and other high level routines in C and system programs in C & C++, perl, python, shell scripts ~~scripts~~ etc. Linux and windows OS are written in C & some sections of code in assembly.

OS structure

A common approach is to partition the task into small components or modules rather than having a monolithic system. Each of these modules should be well defined.



Scanned by CamScanner

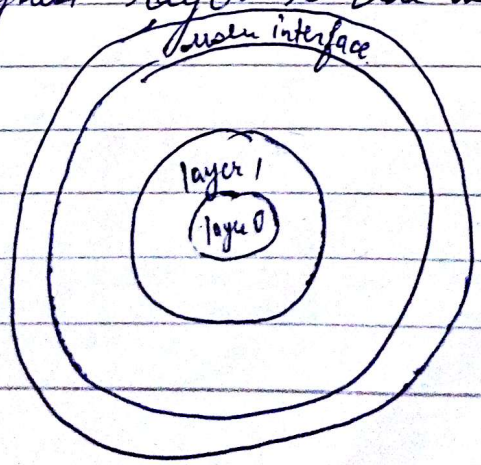
- Unix system consists of 2 seperate parts, the kernel & the system programs. The kernel is further seperated into a series of interfaces and device drivers

	Users		
	Shells & commands		
	Compilers & interpreters		
	System libraries		
	System call interface to the kernel		
Kernel	Signals terminal handling	file system	CPU scheduling
	character I/O system	swapping block I/O system	page replacement demand paging
	terminal drivers	disk & tape drivers	virtual memory
	Kernel interface to the hardware		
	terminal controllers	device controllers disks & tapes	memory controller physical memory

Layered approach

A system can be made moduler in 2 ways —

1) layered approach — where OS is broken into no. of levels or layers. The bottom layer is the hardware and the highest layer is the user interface



Scanned by CamScanner

A typical OS say layer M consists of data structures and a set of routines that can be invoked by higher level layers. Main advantage of layer architecture is the simplicity of construction & debugging. Each layer is implemented only with operations provided by lower level layers. A layer does not need to know how these operations are implemented it only needs to know what these operations do. ~~Diffi~~

Difficulty with layered architecture is appropriately defining the various layers.

Scanned by CamScanner

2-3 pgs missing in

KA

→ Microkernels
→ Solaris module
→ VMware
→ Simulation

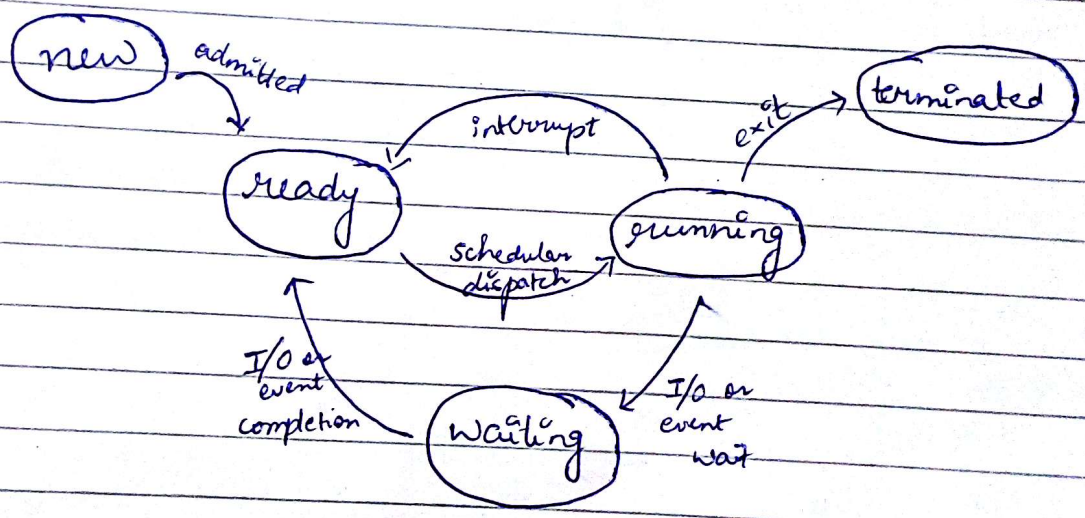
ye topics

Scanned by CamScanner

Process state

As process executes it changes its state. A process maybe in one of the following states -

1) New - The process is being created.
2) running - Instructions are being executed.
3) waiting - The process is waiting for some event to occur.
4) ready - The process is waiting to be assigned to a processor ~~term~~
5) terminated - The process has finished execution.



Process control block (PCB)

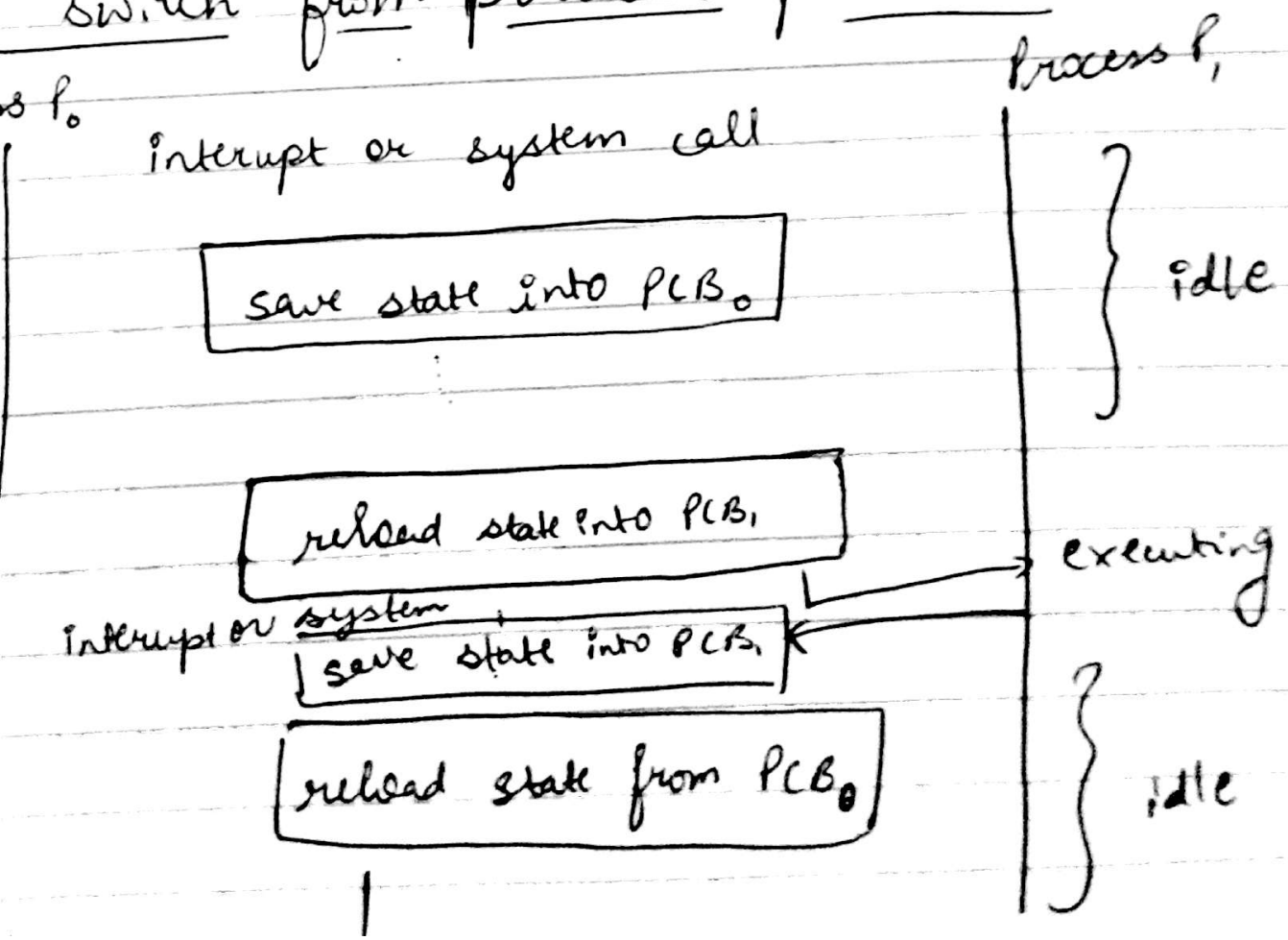
Each process is represented in the OS by a process control block. It contains the following information-

Process state
Process number
Program counter
registers
⋮

I/O status information
memory management

ss state – The state maybe new, ready, running
iting or terminated.
am counter – The counter indicates the address of
xt instruction to be executed for this particular process.
ss number – A unique identification no. is allocated
every process which is used to distinguish it
ong different processes brought into memory.
sters – It includes accumulators, index registers,
tk stack pointers, general purpose registers.
mory limits – Starting address of instruction &
limit means address of last instruction.

switch from process to process



Scanned by CamScanner

Process scheduling

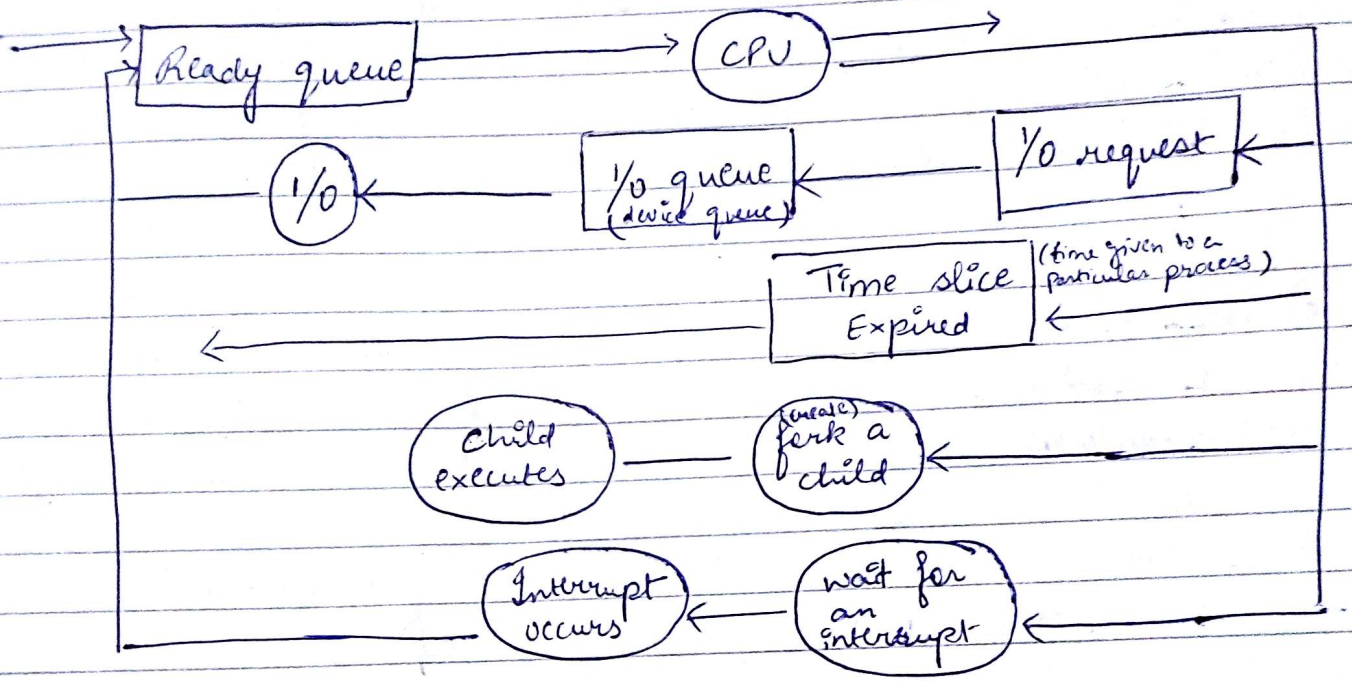
The objective of multiprogramming is to switch the CPU among processes and have some processes running at all times, to maximise the CPU utilisation.

Scheduling queues

- As processes enter the system, they are put into a job queue which consists of all processes in the system.
- The processes that are residing in main memory and are ready ~~an~~ and waiting to execute are kept in a ready queue.
- This queue is stored as a linked lists.
- A ready queue header contains pointers to the first and last PCB.
  Each PCB includes a pointer field to point to next PCB.
- When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted or waits for the occurrence of a particular event such as I/O completion. Suppose a process makes I/O request. Since there are many processes in the system the device maybe ~~bus~~ busy with some other process request. The process therefore have to wait for their device. The list of these processes waiting for a particular device are kept in a device queue.

Scanned by CamScanner

Def A common representation of process scheduling is a queuing diagram.



Schedulers

The selection process is carried out by the scheduler. In a batch system many processes are submitted than can be executed. These processes are spooled to a mass storage device where they are kept for later execution. The long term scheduler / job scheduler select processes from this spool & loads them into memory.

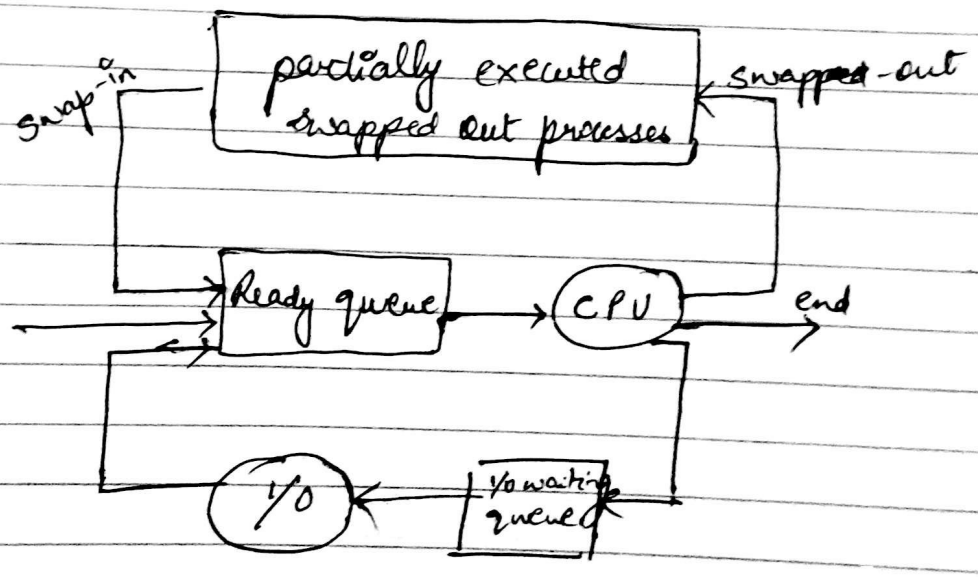
(STS) The short term scheduler - CPU scheduler selects from the processes that are ready to execute & allocates the CPU to one of them.

The short term scheduler must select a new process for the CPU frequently but the long term scheduler (LTS) executes less frequently.

Scanned by CamScanner

- LTS ~~STS~~ controls the degree of multiprogramming. It is important for LTS to make careful selection among the ~~processes~~. CPU & I/O bound processes. If all processes are I/O bound the ready queue will always be empty & short term scheduler will have a little work to do. If all processes are CPU bound the I/O waiting queue will always be empty & devices will be unused.

- On time sharing systems, an intermediate level of scheduling is done with the help of middle term scheduler (MTS)



Context Switch (considered as pure overhead)

- Interrupts calls the OS to change the CPU from its current task and to run a kernel routine. When an interrupt occurs the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done.

Scanned by CamScanner

[Ps-l] It shows all the processes of ~~to~~ linux.

The context is represented in the PCB of the process. It includes the value of CPU registers, process state, memory management info etc.
Switching the CPU to another process requires performing a state save of a current process & a state restore of a different process. This task is known as a context switch.

Operations ~~of~~ on the processes

1) Process creation

- During execution a process may create a several new processes. The creating process is called a the parent process and the new process is called child process.
- All the processes are identified with the help of ~~unit~~ unique process identifier (Pid) which is a integer value. The Pid gives unique value and is used as index to access various attributes of a process within a kernel.
- The Init process is the ~~root~~ or parent process for all user processes.
- Kthread process is responsible for creating additional processes that perform the task on behalf of the kernel.
- sshd process is responsible for managing clients that connect to system by using secure shell (sshd).
- login process is responsible for managing clients or authenticating clients.

When a process creates a child process, that child process will need resources like CPU time,

Scanned by CamScanner

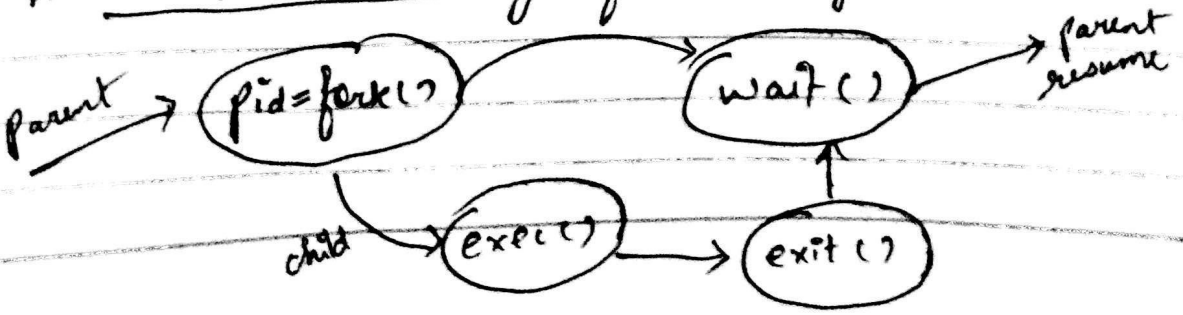
memory, files, I/O devices to accomplish its task.

- Restricting a child process to a subset of parents resources prevents any processes to overloading the system by creating too many child processes.
- When a process creates new process, 2 possibilities for execution exists –
  1) The parent continues to execute concurrently with its children.
  2) The parent waits ~~until~~ until, some or all of the children have terminate.

- There are 2 address space possibility for the new process –
  1) The child is duplicate of the parent process.
  2) The child process has a new program loaded ~~into~~ into it.

- In ~~U*~~ Unix a new process is created by fork system call the new process consists of a copy ~~a~~ of a address space of original process. This mechanism allows parent to communicate easily with the child. After a fork system call one of the 2 processes executes system call to replace the process memory with new program.

<u>Process creation using fork() system call</u>



Scanned by CamScanner

Process termination

- Process terminates when it finishes executing its final statement & ask the OS to delete its using exit() system call.
- Process return a status value to its parent process, as a result all the resources allocated to the process are deallocated by the OS.
- A parent may terminate the execution of one of its children for variety of reasons.
  1) The child has exceeded the usage of some of the space that it has been allocated.
  2) The task assigned to the child is no longer required.
  3) The parent is exiting & OS doesn't allow child to continue.
- Parent process may wait for the termination of child process by using wait() system call.
- The wait() system call is passed as a parameter that allows the parent to obtain exit status of a child.
- The system call also returns the process identifier of the terminated child so that parents knows which of his children has terminated.

Scanned by CamScanner

Interprocess Communication (IPC)

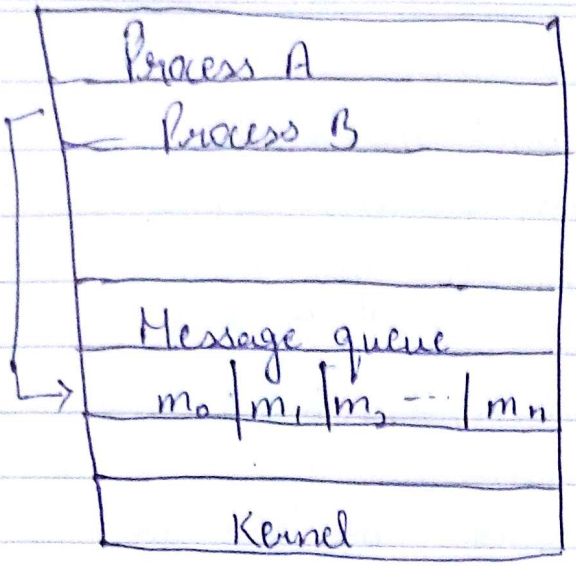
- A process may be independent or cooperating.
- An independent process can not affect or be affected by other processes.
- A co-operating process can affect or be affected by other processes.
- There are several reasons for providing an env that allows process cooperation.
  1) Info sharing
  2) computation speedup
  3) Modularity
  4) convineance.

- Cooperating processes IPC mechanism that allows them to exchange data & info.
  There are 2 ~~modules~~ models for IPC
  1) shared memory
  2) Message passing.

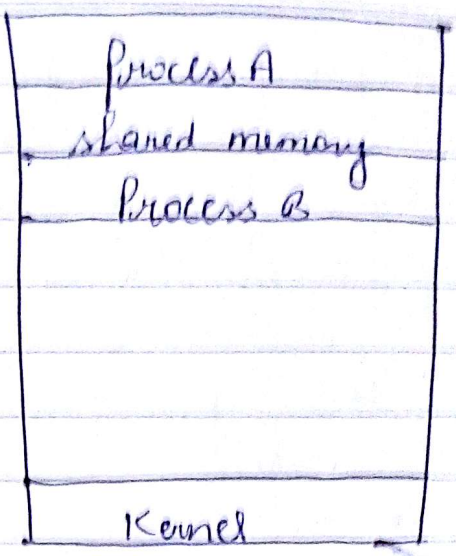
- In shared model, a region of memory that is shared by co-operating processes is established. Processes can exchange info by sending & writing data to the shared region.

- In message passing system, comm takes place by means of messages b/w cooperating processes.

Scanned by CamScanner



message passing



shared model

- Shared memory region resides in the adress space of the process creating the shared memory address.
- The processes are responsible to ensure that they are not writing to the same location simultaneously.
- IPC using shared memory requires comm to establish a region of shared memory. Shared memory region becomes the part of process, creating processes exchange info by reading & writing in shared eg producer & consumer problem.
  A producer produces info that is consumed by a consumer process. One Sol^n to this problem, uses shared memory.
- To allow producer & consumer processes to run concurrently we must have a buffer of item that can be filled by producer & emptied by consumer.

Scanned by CamScanner

- A producer can produce one item, while the consumer ~~can~~ is consuming another item. Both the processes must be synchronised so that the consumer does not try to consume an item that has not yet produced.

- Two types of buffers can be used, bounded & unbounded buffer.
- The consumer have to wait for new items, but the producer can always produce new items (unbounded buffer case)
- In the case of bounded buffer, the consumer must wait if the buffer is empty & producer must wait if buffer is full.

Bounded Buffer

The following variables resides in a region of memory shared by producer and consumer processes.

```
#define BUFFER_SIZE 10
typedef struct
{
  } item;
int BUFFER[BUFFER_SIZE]:
  int in = 10;              // producer
  int out = 0;              // consumer
```

- The shared buffer is implemented as a circular array with two logical pointers (int & out).

Scanned by CamScanner

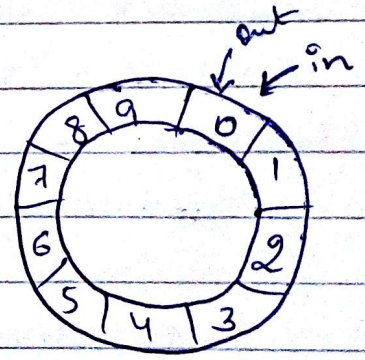
'In' pointer points ~~the~~ to next free position in the buffer whereas 'out' variable points to first full position in the buffer.

The buffer is empty when in = out. The buffer is full when [(in + 1) % BUFFER_SIZE = out]

Producer

```
while (true)
{ // produce an item in next produced

    while ((in+1) % BUFFER_SIZE) == out )
      ;            // do nothing
    buffer[in] = next_produced;
    in = (in+1) % BUFFER_SIZE ;
}
```




```
item next consumed;
while (true)
{ while ( in == out)
      ;   // do nothing
    next consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    // consume the item in next consumed
}
```

Scanned by CamScanner

Messaging passing system

It provides a mechanism to allow processes to communicate & to syncronise their actions without sharing the same addresses space

It is useful in distributed env. A message passing facility provides atleast 2 operations –
1) send 2) receive.

- message can be fixed or variable size. If process P & Q wants to communicate, they must send messages to & receive messages from each other via comm layer.

This link can be implemented in the following ways
1) Direct or indirect comm
2) syncronous or asynchronous comm.
3) Automatic or explicit comm.

→ Direct comm.

naming tech – Processes must have a way to refer to each other in direct comm. Each process that wants to communicate must explicitly name the ~~recip~~ recipient or sender of comm

– send & receive primitives are defined as
- send (P, message) : send msg to process P
- Receive (Q, message) : Receive msg for process Q

Scanned by CamScanner

- The comm has following properties -
1) A link is established automatically b/w query of processes that wants to comm.
2) A link is associated with exactly 2 processes
3) B/w each pair of processes, their exist one layer

- In asymmetry in addressing only the sender process names the recepient does not name the sender.

Receive (id, msg)

Id is set to the name of the process with which comm has taken place.

Indirect comm -

In this, messages are send to & receive from mail boxes. Each mail box has a unique id. A process can communicate with another process via in no. of diff mail boxes But process can communicate only if they have a shared mail box.

Send (A. message) : Send the message to mail box A
Receive (A. message) : Receive the message from " " A.

- In this scheme the comm. link has following properties :-
1) A link is established b/w pair of processes only if both member of pair have a shared mail box.
2) A link may be associated with more than 2 processes
3) B/w each pair of comm. processes a no of diff links may exist, with each link corresponding to one mail box.

Scanned by CamScanner

Q) Suppose P_0, P_1, P_2, P_3 all shared the mail box A. P_1 sends the msg to A while P_2 & P_3 execute receive from A. Which process will receive message send by P_1?

1) Allow a link to be associated with two processes almost
2) ~~Allo~~ Allow almost one process at a time to execute receive operation.
3) Allow the system to select ~~arbrit~~ arbitrily which process will receive the message.

Synchronization

Message passing can be either blocking or non-blocking (synchronous / asynchronous)

Blocking send: The sending process is blocked until the message is received by the receiving process or by the mail box.

Non-blocking send: The sending process sends the message & resumes operation.

Blocking receive: The receiver blocks, until the message is available.

Non-blocking receive: The receiver retrieves either a valid message or NULL.

Scanned by CamScanner

Producer

```
message next_produced ;
while (true)
 { /* produce an item in next produced ;
    send (next_produced);
 }
```

Consumer

```
message next_consumed ;
while(true)
 { receive (next_consumed);
    /* consume the item in next consumed ;
 }
```

Buffering

Messages exchange by comm processes reside in a temp queue such queues can be implemented in 3 ways -

1) Zero capacity - The queue length is zero, thus the link cannot have any messages waiting in it.
   - In this case, the sender must block until the recepient receive the message.

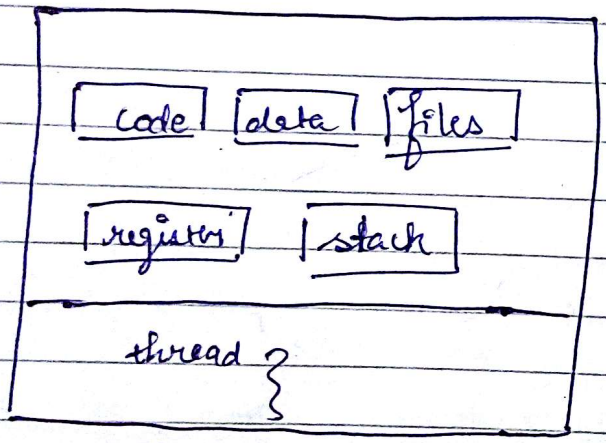
2) Bounded capacity - The queue has a finite length n, thus atmost n messages can reside in it.

3) Unbounded capacity - The queues length is potentially infinite.

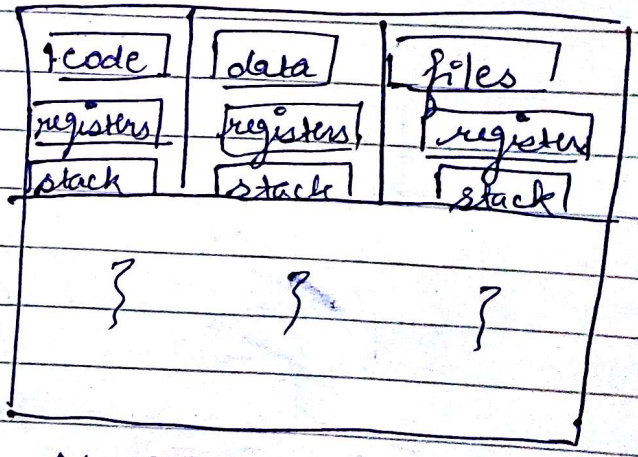
Scanned by CamScanner

Ch 4 Multi-threaded Programming

- A process was an executing program with a single thread of control.
- A thread is a basic unit of CPU utilisation, it comprises a thread ID, a program counter, a register set and a stack.
- A thread shares code section, data section and OS resources with other threads belonging to the same process
- If a process has multithreads of control it can perform more than one task at a time.

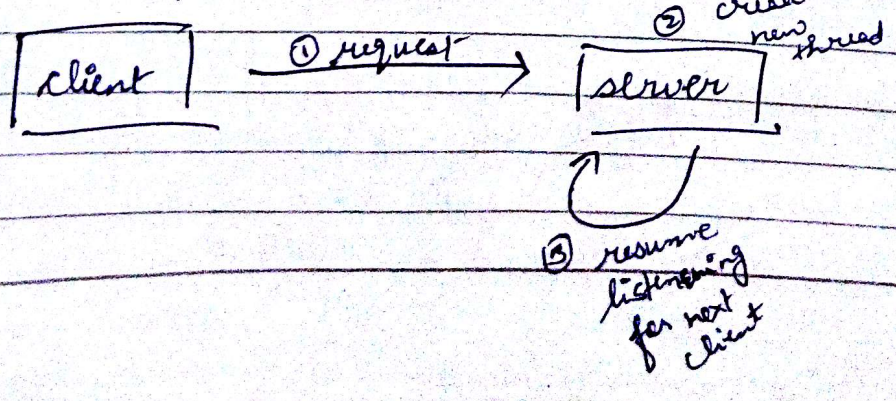


Single threaded process



Multithreaded process

- Web server process can be multithreaded, the server will create a separate thread that listens for clients request



Scanned by CamScanner

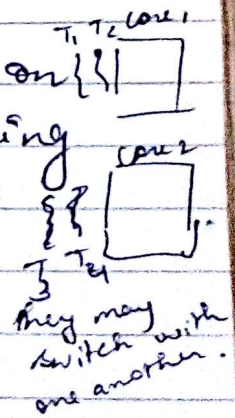
Benifits of multithreaded Programming

1) responsiveness - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.

2) resource sharing - Threads share the memory and the resources of the process to which they belong. The benifit of sharing code and data is that it allows an application to have different threads of activity within the same address space.

3) economy - Threads share the resources of the process to which they belong, it is more economical to create and context switch thread.

4) scalability - Threads may run in parallel on different processing parts in multi processing architecture



Multi-core programming

- Multithreaded programming provides a mechanism for more efficient use of multiple computing cores and improved concurrency.

- Consider an application with 4 threads on a system with a single computing core. Concurrency here means that the exeution of the threads

Scanned by CamScanner

will be interleaved over time.

single core

T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4

1 sec

time.

- In a system with multiple cores, concurrency means that the threads can run in parallel because the system can assign a separate thread to each core.

core 1 | T_1 | T_3 | T_1 | T_3 | T_1 . . . . .

core 2 | T_2 | T_4 | T_2 | T_4 | T_2 | . . . . .

Programming challenges

- 5 areas present challenges in programming for multicore systems

i) Identifying tasks - This involves ~~examining~~ examining applications to find areas that can be divided into separate concurrent tasks so that they can run in parallel on individual cores.

ii) Balance - For the tasks that can run in parallel, programmers must ensure that the tasks perform equal work of equal value.

iii) data splitting - The data accessed & manipulated by the tasks must be divided to run on separate cores.

Scanned by CamScanner

iv) data dependency - The data accessed by the tasks must be examined for dependencies b/w 2 or more tasks. when one task ~~depend~~ depends on the data from another then programmers must ensure the execution of tasks must be ~~syn~~ synchronised.

v) testing and debugging - when a program is running in parallel on multiple cores, many different execution paths are possible therefore testing & debugging such concurrent programs is difficult.

Type of parallelism

There are 2 types of parallelism
1) data parallelism
2) task parallelism

1) It focuses on distributing subsets of the same data across multiple computing cores

2) It involves distributing tasks (threads) across multiple computing cores. Each thread performs a unique operation.

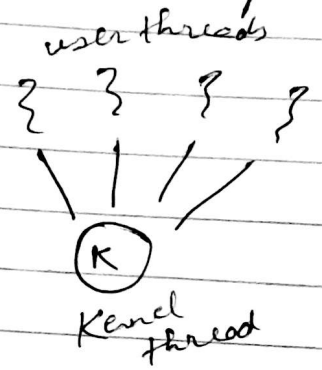
Multithreading models

- It provides support for user threads & kernel threads.
- There are 3 ways to establish a relationship b/w user & kernel threads.

Scanned by CamScanner

1) <u>many-to-one model</u>

This model maps many user level threads to one kernel thread. Thread management is done by the thread library in the user space. Entire process will be blocked if a thread makes a blocking system call. Only one thread can access the kernel at a time so multiple threads are unable to run parallely on multicore systems.

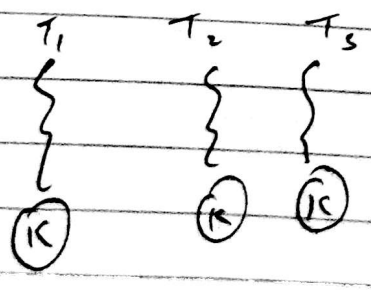


2) <u>one-to-one model</u>

This model maps each user thread to a kernel thread. It provides more concurrency than many to one model by allowing another thread to run when a makes a blocking system call. It ~~allo~~ also allows multiple threads to run in parallel.

~~3) many to~~

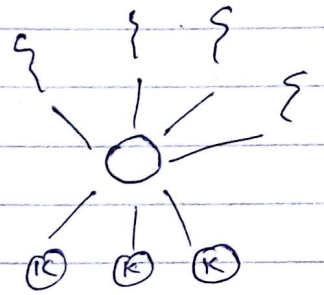
3)



Scanned by CamScanner

3) many to many model

This model multiplexes many user level threads to a smaller or equal no. of kernel threads. The no. of kernel threads may be specific to a machine. In this model the developer can create as many user threads as necessary and corresponding kernel thread can run in parallel on multiprocessor.



Scanned by CamScanner

Ch- Process scheduling

→ CPU scheduling is the basis of multiprogram OS by switching the CPU among processes. the OS can make the computer more productive

→ In a single processor system, only one process can run at a time and others must wait until the CPU is free.

→ In multiprogramming we keep several processes in the memory at one time when one process has to wait the OS takes the CPU away from that process and gives it to another process.

CPU I/O burst cycle

Process execution consists of a cycle of CPU execution and I/O wait. Process alternate b/w these 2 states Process execution begins with a CPU burst and is followed by I/O burst which is followed by CPU burst and so on.

this work is done by processor

load store add store read from file	CPU burst
wait for I/O	I/O burst
store increment index wait to a file	CPU burst
wait for a I/O	I/O burst

Scanned by CamScanner

CPU scheduler

→ whenever the CPU becomes ideal the OS must select one of the processes in the ready queue to be executed this selection is carried out by short-term scheduler or CPU scheduler.

→ The scheduler selects a process from the processes in memory that are ready to be executed and allocates the CPU to that process. A ready queue can be implemented as FIFO queue or a priority queue or in a tree form or unordered linked list.

Numericals Imp. **Preemptive scheduling**

CPU scheduling decisions may take place under certain circumstances.

1. when a process switches from the running state to the waiting state. (I/O request comes)
2. when a process switches from the running state to the ready state. (when interrupt occurs)
3. when a process switches from waiting state to the ready state. (when I/O activity is completed)
4. when a process terminates.

In case 1 & 4 the scheduling scheme is non-preemptive or co-operative.

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or switching. (switching to a waiting job)

Scanned by CamScanner

→ Premptive scheduling results in race condition when data are shared among several processes. Consider the 2 processes that share the data. While one process is updating the data it is prempted so that the second process can run. The second process then tries to read the data which leads to an inconsistent state.

Dispatcher

Dispatcher is the module that gives the control of the CPU to a process selected by the short term scheduler. This involves the following steps :-

1) switching context
2) switching to a user mode
3) jumping to the proper location in the user program to restart that program.

Scheduling criteria

Criterias for comparing CPU scheduling algorithms are

1. CPU utilisation - We want to keep the CPU as busy as possible.

2. through-put - One measure of the work is the no. of processes that are completed per time unit is called throughput.

3. turn around time - From the point of view of a particular process the important criteria is how long it takes to execute the process.

Scanned by CamScanner

The interval from the time of submission of the process to the time of completion is known as turn around time.

4. Waiting time - The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O. Waiting time is the sum of the units spend waiting in the ready queue.

5. Response time - The time from the submission of a request until the first response is produced is the response time.

Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated to the CPU.

I 1). FCFS scheduling (first come first serve)

In this the process that requests the CPU first is allocated the CPU first. (it can be implemented using FIFO queue). When the process enters the ready queue its PCB is linked onto the tail of the queue. When the CPU is free it is allocated to the process at the head of the queue.

eg:-

Processes	CPU burst time
P_1	24
P_2	3
P_3	3

Scanned by CamScanner

We are assuming the processes are arrived in order P_1, P_2 & P_3 then P_3.

Gantt chart

P_1	P_2	P_3

0 24 27 30

waiting time

P_1	0
P_2	24
P_3	27

Avg waiting time

$$\frac{0+24+27}{3}$$

$= 17$ sec.

If order will be P_2, P_3 & P_1 the avg time $= 9$ sec

P_2	P_3	P_1

0 3 6 30

II Shortest job first scheduling (SJFS)

This algorithm associates with each process. The length of the process of next CPU burst. When the CPU is available it is assign to the process that has the smallest next CPU burst. If the CPU burst of 2 processes are same FCFS scheduling is used to break the ~~time~~ tie.

Processes	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

P_4	P_1	P_3	P_2

0 3 9 16 24

waiting time

P_4	0
P_1	3
P_3	9
P_2	16

Avg time

$$= \frac{0+3+9+16}{4}$$

$= 7$ sec

8/7

Scanned by CamScanner

SJFS algorithm is optimal because it gives the minimum avg. waiting time for a given set of processes.

The drawback is knowing the length of next CPU burst. SJFS can be either premptive or non-premptive. A premptive SJS will prempt the currently executing process. It is also known as shortest remaining time first algorithm.

SJFS (Premptive)

Process	Arrival time	Burst time
P_1	0	~~8~~ 7
P_2	1	4
P_3	2	9
P_4	3	5

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

Gantt Chart

Waiting time

P_1 $10 - 1 = 9$

P_2 0 (coz arrival time is 1 so no waiting)

P_4 $5 - 3 = 2$

P_3 $17 - 2 = 15$

$$\text{Avg} = \frac{9+0+2+15}{4} = 6.5 \text{ ns}$$

Scanned by CamScanner

Priority scheduling

A priority is associated with each process and the CPU is allocated to the process with the highest priority. (lowest number)
Equal priority processes are serviced by FCFS.
Priorities are indicated by a range of numbers beginning by 0 to 10.

processes	Burst time ~~Priority~~	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Gantt chart

P_2	P_5	P_1	P_3	P_4

0 1 6 16 18 19

	waiting time
P_2	0
P_5	1
P_1	6
P_3	16
P_4	18

$\frac{41}{5}$ 8.2

Avg time = 8.2

A major problem with Priority scheduling is indefinite blocking or starvation
A solution to this problem is ageing which means gradually increasing the priority of the processes that wait in the system for long time.

Scanned by CamScanner

Round robin scheduling

It is designed for time sharing systems premption is added to enable the system to switch b/w processes.
A small unit of time called a time quantum or time slice is defined. A ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue allocating the CPU to each processe for a time interval of upto 1 time quantum.

time slice = 4 sec

Processes	
P_1	~~24~~ ~~12~~ 12
P_2	3
P_3	3

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

waiting time

P_1	$10-4=6$
P_2	4
P_3	7

$$\frac{6+4+7}{3} = \frac{17}{3} = 5.6 \text{ sec}$$

OS work in CN →
till paging

Scanned by CamScanner

OS

Ch Memory management:
1 Pg missing

Scanned by CamScanner

- As process enters the system and are put in the input queue the operating system takes into account the memory requirements of each process and the amount of available memory space. At any given time we have a list of available block sizes and an input queue.
- If there is no large enough block available then OS waits or picks up another process from the input queue
- Memory blocks available comprise of a set of holes of various sizes scattered throughout the memory.
- When process arrives and needs memory the system searches for a ~~ss~~ hole large enough if the hole is too large then it splits up it into 2 parts.
- When a process terminates it releases its memory which is placed back in the set of free holes.
- If the new hole is adjacent to other holes these are merged to form a larger hole.
- The system may need to check whether there are processes waiting for memory and whether this newly freed recombined memory could satisfy the demands of any of the waiting processes. This problem is called dynamic storage allocation problem.

Solutions are :-

1) First - fit
2) Best fit
3) worst fit

First fit algo allocates the first hole i.e big enough to the requesting process.

Scanned by CamScanner

→ Best fit it allocates the smallest hole i.e big enough. In this we search the entire list. This method produces the smallest left over hole.

→ Worst fit allocates the largest hole to the requesting process for this we must search the entire list.

Fragmentation

Both first & best fit strategies suffer from external fragmentation. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous i.e storage is fragmented into large no of ~~so~~ ~~ca~~ small holes.

~~Another factor~~ Memory fragmentation can be internal or external.

Consider a multiple partition allocation scheme with a hole of 18464 bytes. Suppose a process requests 18462 bytes. If we allocate exactly the requested block we are left with a hole of 2 bytes. The overhead to keep the track of this 2 bytes hole will be larger than the hole itself.

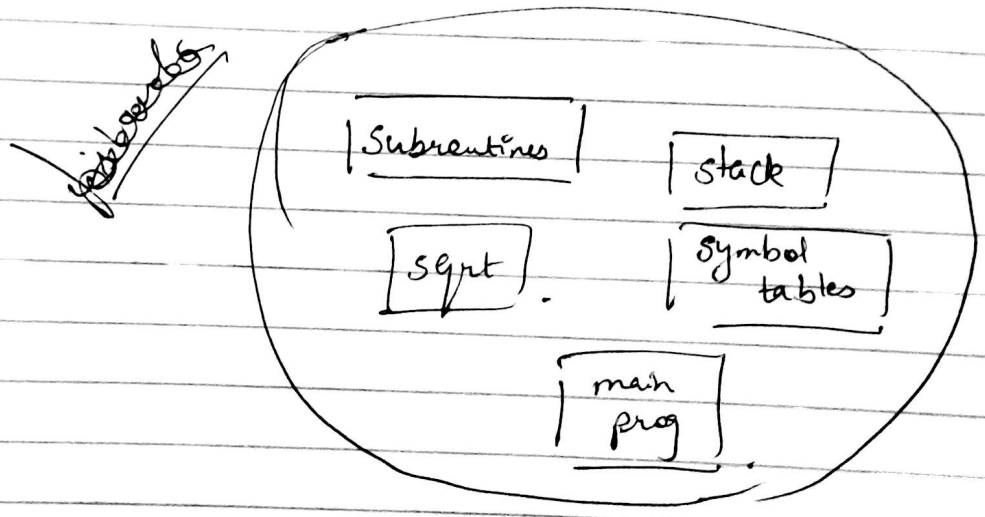
- The general solⁿ is to divide the memory into fixed size block and allocate the processes based on the block size
- One solⁿ to the problem of external fragmentation is compaction the goal is to shuffle the memory contents so as to place all free memory together in one place the simplest algorithm is to move all processes towards one end of memory and all holes move in other direction.

Scanned by CamScanner

Another solⁿ to external fragmentation is to permit the process space to be non-contiguous i.e allowing a process to be allocated physical memory wherever such memory is available

Segmentation

It provides a memory mechanism that map the programmers view to the actual physical memory. Programmers view memory as a ~~co~~ collection of variables size segments
A logical address space is a collection of segment.



<segment no, offset value>

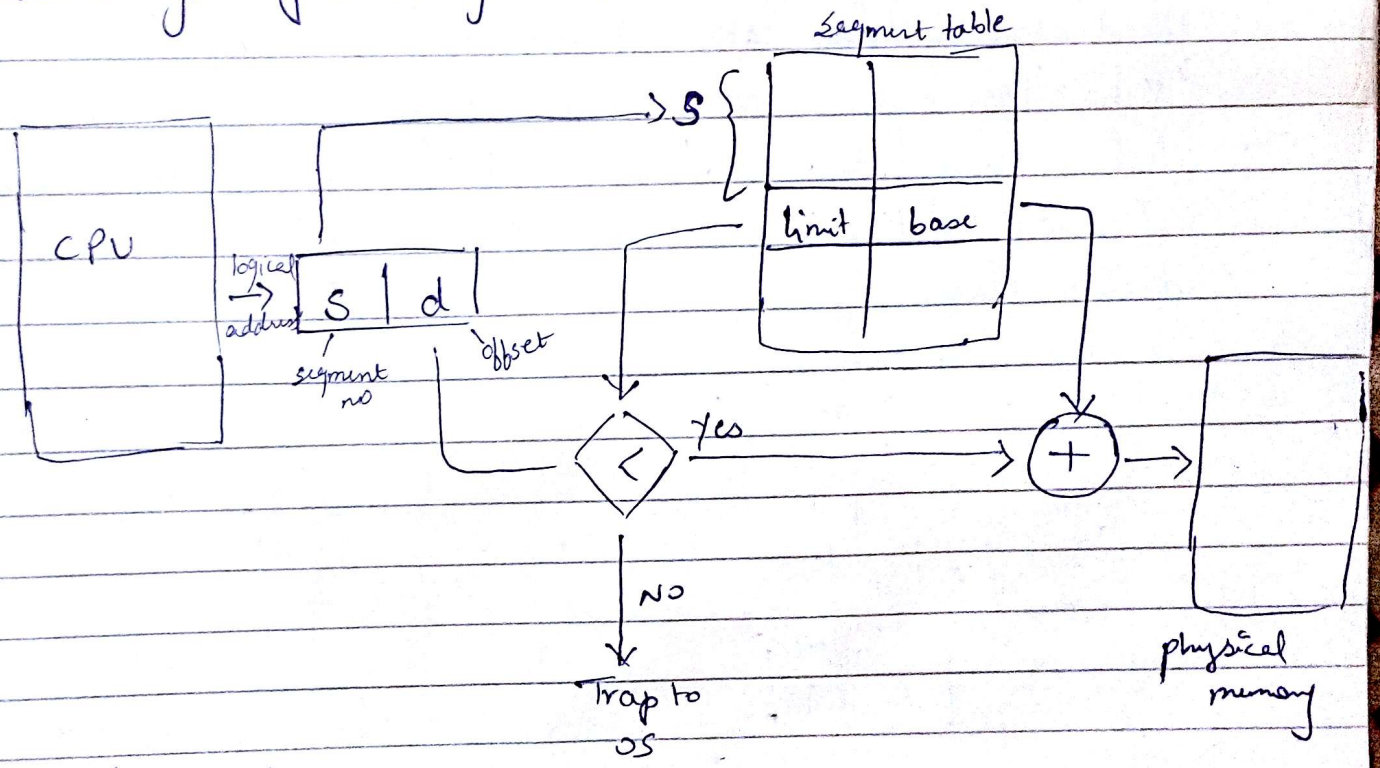
A C ~~compli~~ compiler might contain separate segments for code, global variables, ~~a~~ heap, C library, stack used by each thread.

~~Segmentation~~

Scanned by CamScanner

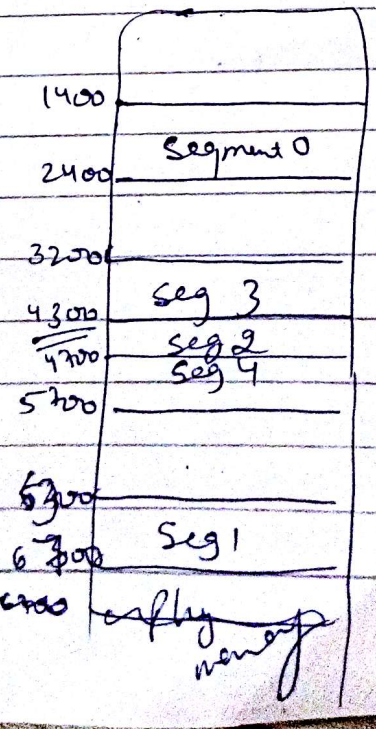
Segmentation hardware

The programmer refer to objects in the programes by a 2D address but the actual physical memory is the 1D sequence of bytes thus a method to map 2D address to 1D is required this mapping is done by segment table. Each entry in the segment table has a segment base and a segment limit the segment base is the starting physical address & the segment limit is the length of the segment.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Seg Table



logical address 2D

2	53

seg no | offset

$$\begin{array}{r} 4300 \\ +\ 53 \\ \hline 4353 \end{array}$$

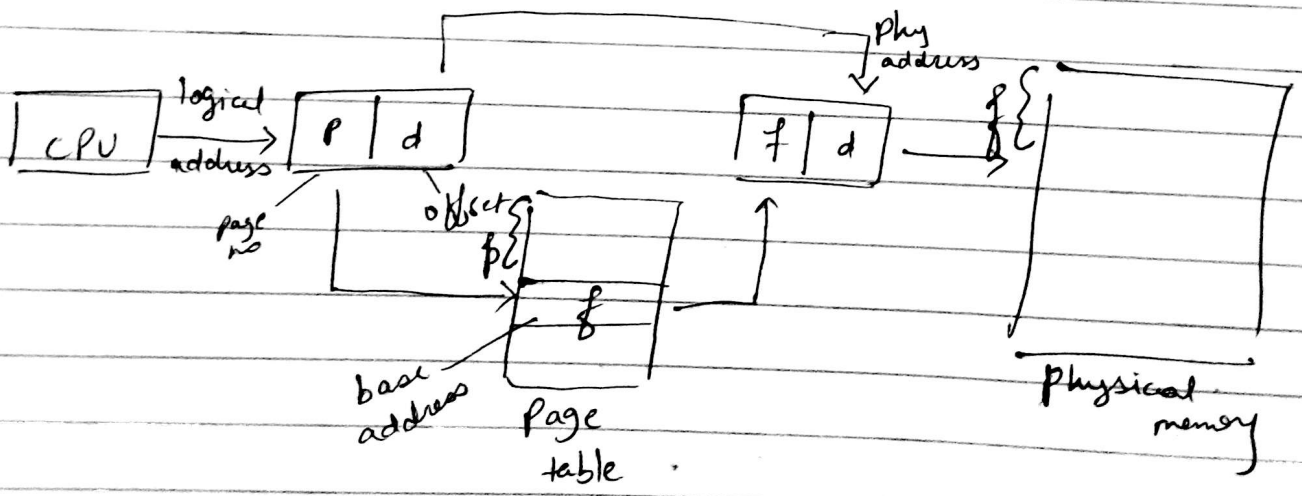
1D physical address of seg 2

Scanned by CamScanner

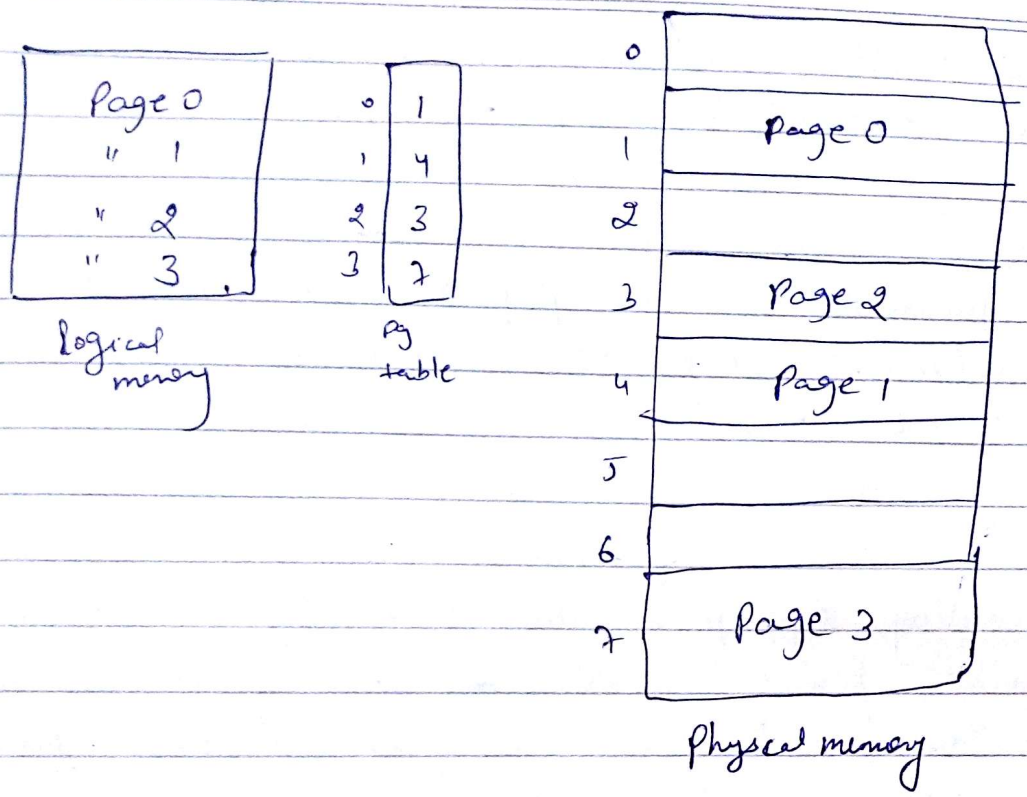
Pageing

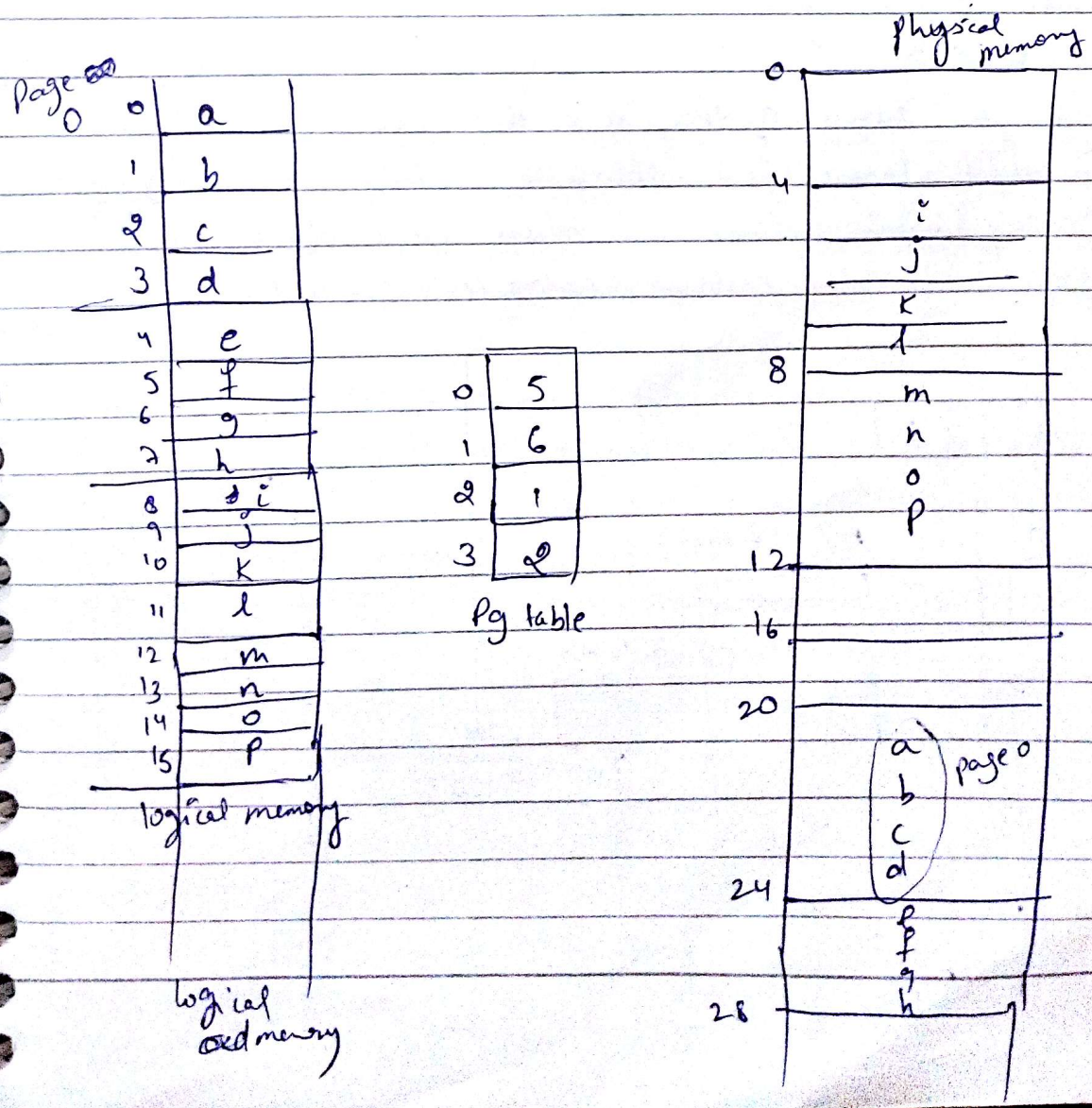
Segmentation permits the physical address space of a process to be non-contiguous.

- Pageing also provides this advantage It also avoids external fragmentation & need for compaction.
- It solves the problem of fitting memory chunks of varying sizes onto backing store.
- In this technique we break the physical memory into fixed sized blocks called frames & breaking logical memory into blocks of same size called pages.
- When the process is to be executed its pages are loaded into any available memory frame from their source.



Scanned by CamScanner





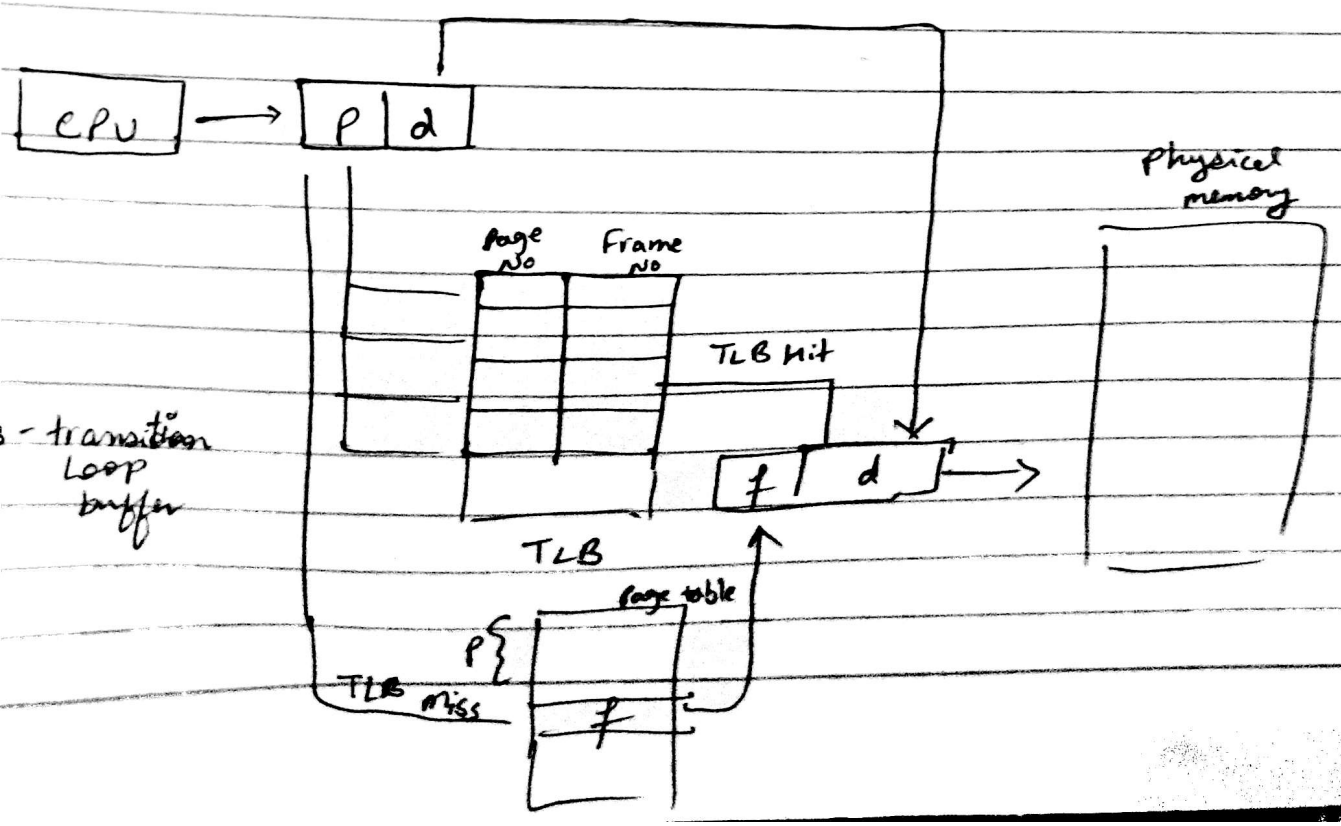
Scanned by CamScanner

Page size is 4 bytes. physical memory is 32 bytes.
given logical address (suppose) is 0. ⇒ pg no = 0
offset = 0

looking into pg table we see page 0 is in frame 5
∴ logical address 0 maps to physical address
$\underbrace{5}_{\text{frame}} \times \underbrace{4}_{\text{bytes}} + \underbrace{0}_{\text{offset}} = 20.$

Since operating system is managing physical memory so when a process arrives in the system to be executed its size in pages is examined each page of process needs one frame thus n pages requires n frames if n frames are available they are allocated.

OS must be aware of the allocation detail of physical memory - which frames are allocated, which are available, how many total frames are there etc. This info is kept in a data structure called frame table.



Scanned by CamScanner

VIRTUAL MEMORY UTILISATATION

Virtual memory is a technique that allows the execution of processes that are not completely in memory.

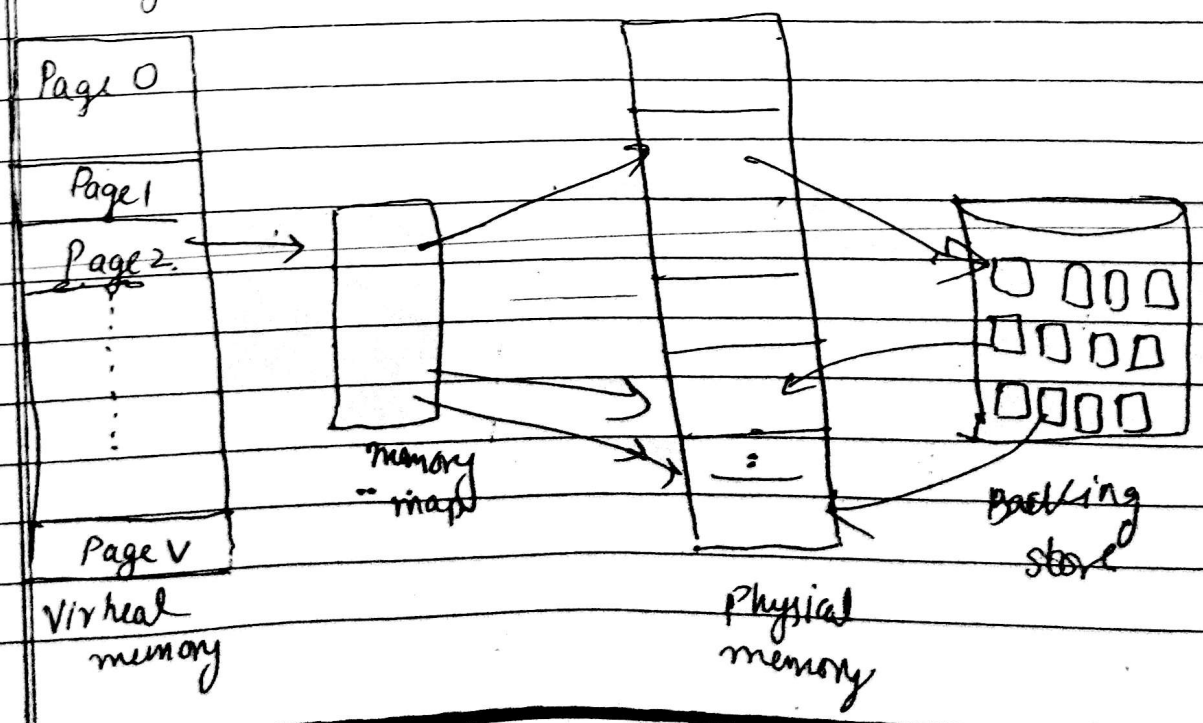
The instructions being executed must be in physical memory ∴ it limits the size of a program to the size of physical memory.

Even in these cases, the entire program is needed, but it may not all be needed at the same time.

The ability to execute a program i.e only partially in memory has many benefits:-

i) a program is not constrained by the amount of physical memory available.

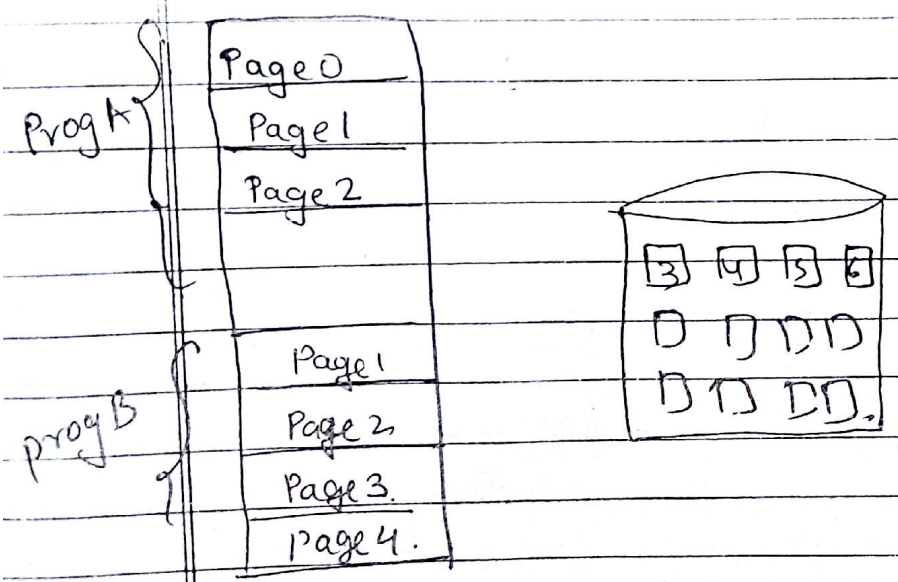
ii) lesser amount of physical memory is consumed by each user ∴ more programs can be run at the same time, thus increase the CPU utilisation and throughput.



The virtual address space of a process refers to the logical view of how process is stored in memory.
Virtual memory allows files and memory to be shared by 2 or more processes by page sharing.

Demand paging -

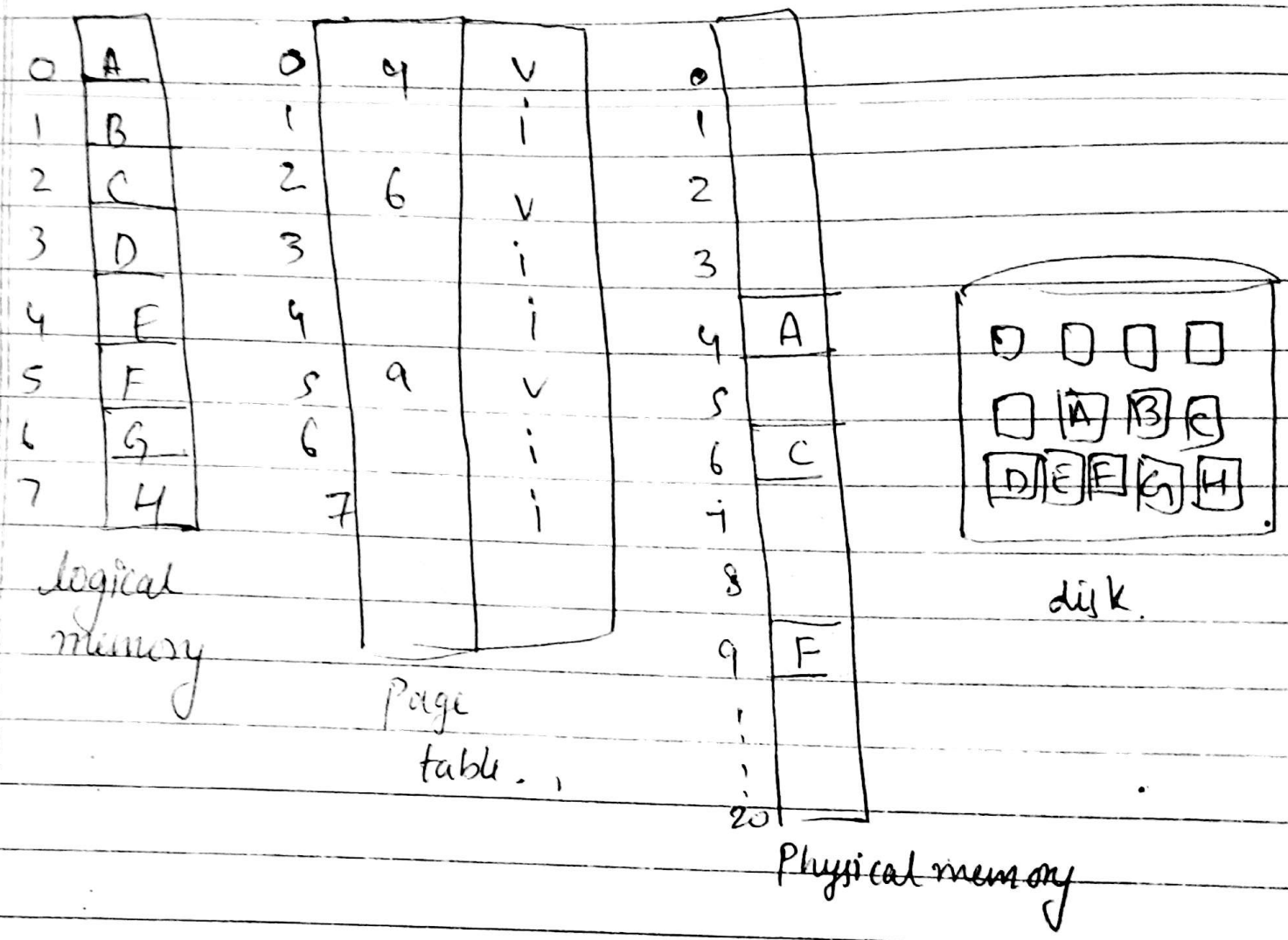
In demand page virtual memory pages are loaded, when they are demanded ~~tt~~ during program execution.
pages that are never accessed are never loaded into physical memory.
In this processes reside in secondary memory when we want to execute a process, we swap it into memory.
A lazy swapper is used which never swaps a page into memory, unless that page will be needed.



When a process is to be swapped in, the pager process guesses, which process will be used, before the process is swapped out again.

The valid invalid bit is used to distinguish the pages that are in memory and the pages that are on disk. When this bit is valid, the associat

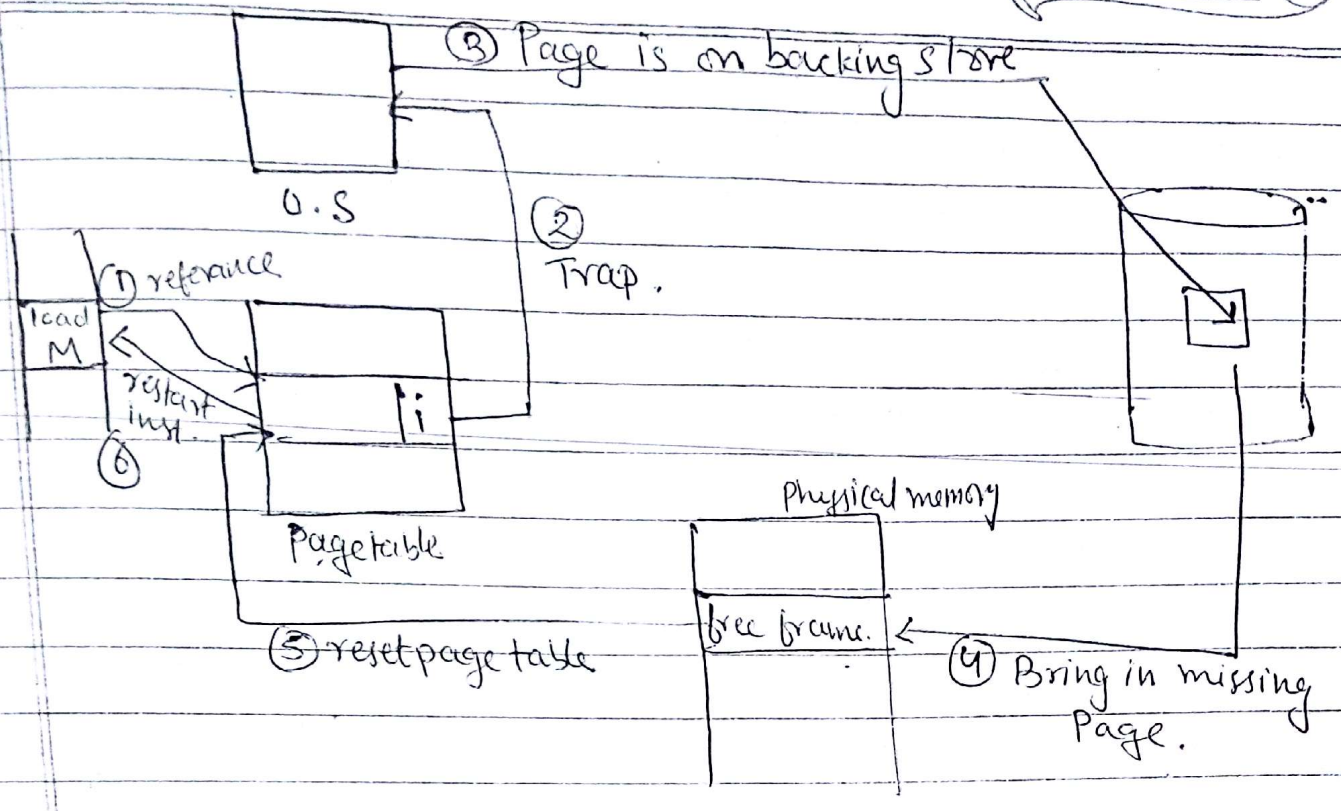
page is legal and is in memory. if it is invalid, it shows the page is either not valid or is valid but is currently on the disk.



What happens if the process tries to access a page, that was not brought into the memory

access to a page marked as invalid causes a page fault. The paging hardware in translating the address through the page table notice that invaild bit is set, causing a trap to OS. This trap is the result of the OS failure to bring the desired page into the memory.

Scanned by CamScanner



- The O.S checks internal table, i.e PCB for the current process to find out whether, refrence is valid or invalid.
- If invalid, it terminates the process or if valid and not yet brought it page its in.
- finds a free frame
- shedule disk operation, to read a desired page into, newly allocated frame and page table.
- modify the internal table indicating presence in memory.
- restart the instruction.

pure demand paging -
when we start executing a process, with no pages in memory, this concept is known as pure demand paging.

When the OS sets the instruction pointer to the first instruction of the process which is present in non-resident memory page: The process immediately faults for that page.

Scanned by CamScanner

- A crucial requirement for demand paging is the ability to restart any instruction after a page fault occurs.

performance of demand paging:-
The effective access time (EAT) for demand page memory is computed as.

$$EAT = (1-p)ma) + p \times \text{fault time}$$

ma - memory access time
p - probability of page fault

To compute EAT, we must know, how much time is needed to service a page fault.

A page fault causes a following sequence to occur.
1) Trap to the OS
2) Save the user register and process state.
3) determine that interrupt was a page fault.
4) check that the page reference was legal and determine location of page on disk.
5) issue a read from a disk to a free frame
a) wait in a queue for this device until the read request is serviced
b) wait for the device or seek latency time
c) began the transfer of the page to a free frame
(optional) 6) While waiting allocate the CPU to some other user.
7) recieve an interrupt from disk input output sub system (I/O completion)
8) save the register and process state for other user (only if step 6 is executed)

9. Determine that the interrupt was from the disk
10. Correct the page table and other table to show that the desired page is in memory now.
11. Wait for the CPU, to be allocated to this process again.
12. restore the register, process state and new page table, and resume the interrupted instructions.

Page replacement ~~algro~~ algorithms.

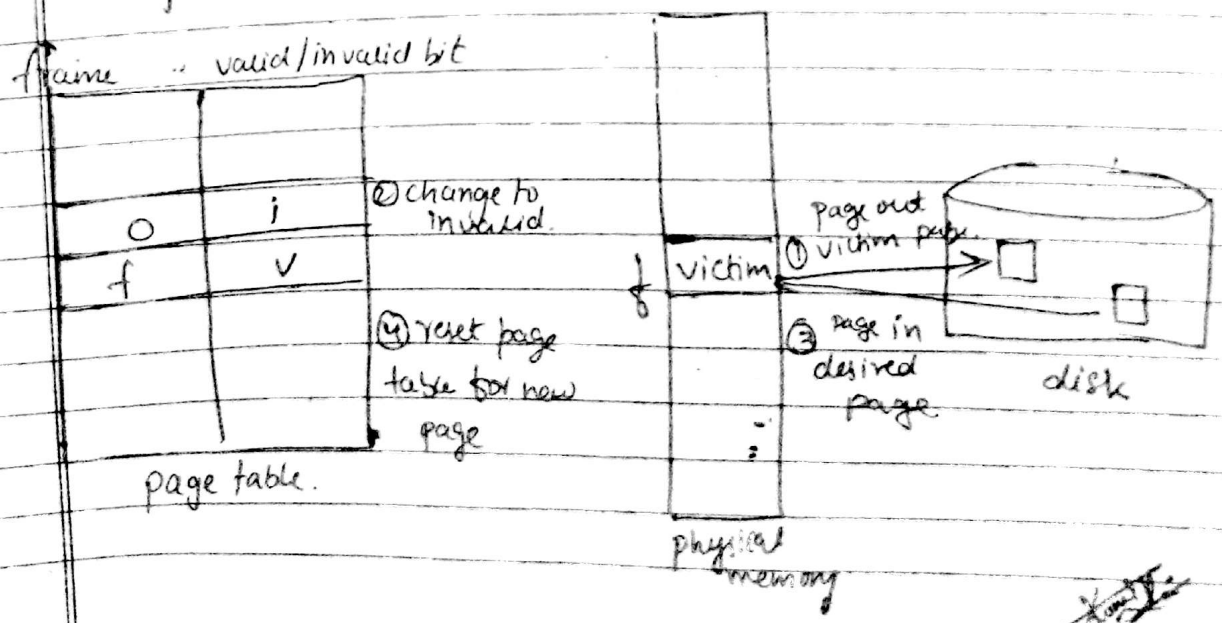
→ need for page replacement -

if we increase the degree of multiprogram, we over allocate the memory.

While a user process is executing a page fault occurs. The O.S determines where the desired page is residing on disk but then finds that there are no free frames. Thus there is a need for page replacement.

If no frame is free, we find the one that is not currently being used and freeze it.

We can free a frame by writing its content to the swap space / disk and changing the page table entry.



FIFO page replacement algorithm.

This algorithm associates, with each page, the time when that page was brought into memory, when a page must be replaced, The oldest page is choosen.

frame = 3

Q ~~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
	0	0	0	3	3	3	2	2	2	1	1
		1	1	1	0	0	0	3	3	3	2
P	P	P	P	P	P	P	P	P	P	P	P

7	7	7
1	0	0
2	2	1
P	P	P

15

frame 4, 3

~~1, 2, 3, 4, 1, 2, 5~~, 1, 2, ~~3~~, 4, 5.

1	1	1	4	4	4	5	5	5	
2	2	2	2	1	1	1	3	4	
		3	3	3	2	2	2	3	
			P	P	P	P	P	P	

9

1	1	1	1	5	5	5	5	4	4
	2	2	2	2	1	1	1	1	5
		3	3	3	3	2	2	2	2
			4	4	4	4	3	3	3
				P	P	P	P	P	P

10

Belady's Problem.

Scanned by CamScanner

Optimal page replacement algorithm.
it will give you lowest page fault rate, never suffers from belady's problem. It replaces the page that is not used for longest period of time.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
frame 3.

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

9 page faults.

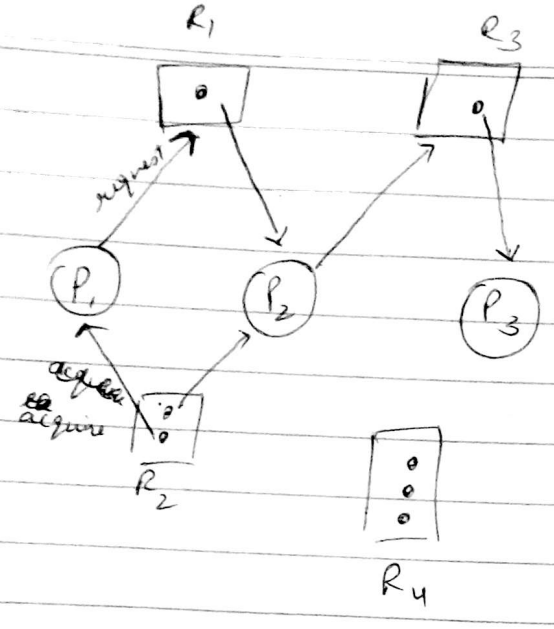
LRU (least recently used) algorithm.
In this we use the recent past as an approximation of the near future and then we can replace the page, that has not been used for longest period of time.

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						

12 faults.

Ch. Deadlocks (ppt)



No deadlock state coz there is no circular queue.

Ch Synchronization

- A creating process is one that can affect or be affected by other processes executing in the system.
- Processes can execute concurrently. This means that one process may partially complete execution before another process schedule.
- A process may be interupted in any point in its instruction stream and the processing core may be assigned to be execute instructions of another process. In parallel execution 2 instruction streams execute simultaneously on separate processing cores.

Bounded buffer program allowed at most BUFFER_SIZE - 1 in the buffer.
Solution is to add an integer variable counter initialise to 0 counter is incremented every time

we add a new counter and decremented every time we remove an item from buffer

Producer

```
while (true)
{ /* produce an item in next produced */
    while (counter == BUFFER_SIZE)
        ; /* do Nothing */
    Buffer[in] = next produced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

because buffer is circular buffer that why we have used % sign

Consumer

```
while(true)
{ while (counter == 0)
        ; /* do nothing */
    next_consumed = Buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter--;
    /* consume an item from next consumed */
}
```

Both the above codes are correct separately but incorrect if executed concurrently.

Eg:- if counter = 5 producer & consumer processes execute the statement concurrently counter++ & counter--.

The value of the counter may be 4, 5 or 6.

Scanned by CamScanner

counter ++

~~register = counter~~

register 1 = counter

register 1 = register 1 + 1

counter = register 1

counter --

register 2 = counter

register 2 = register 2 - 1

counter = register 2.

Concurrent execution of counter++ & counter -- is equivalent to sequential execution in which the statements are interleaved in some order for eg:-

T_0 : producer execute register 1 = counter
[register 1 = 5]

T_1 : producer execute register 1 = register 1 + 1
[register 1 = 6]

T_2 : consumer execute register 2 = counter
[register 2 = 5]

T_3 : consumer execute register 2 = register 2 - 1
[register 2 = 4]

T_4 : producer execute counter = register 1
[counter = 6]

T_5 : consumer execute counter = register 2
[counter = 4]

2 diff value (problem occurs)

We may arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation where several processes access and manipulate the same data concurrently & the outcome of the execution depends on the particular order in which access takes place is called a race

condition.

The solution is to ensure that only one process at a time can be manipulating the variable counter. for this the processes must be syncronized.

Critical section problem

→ Consider a system with n processes $P_0, P_1, P_2, P_3 \ldots P_{n-1}$. Each process has a segment of code called a critical section in which the process maybe changing common variables or update a table, write a file etc.

→ when one process is executing in its critical section no other process is allowed to execute in its critical section.

→ Critical section problem is to design a protocol that the processes can use to co-operate.

→ Each process must request permission to enter its critical section.

→ The section of code implementing this request is the <u>entry section</u>.

→ The critical section may be followed by an <u>exit section</u>.

The remaining code is in the <u>remainder section</u>

```
do
  { entry section
        critical section
    Exit section
        Remainder section
  } while (true);
```

Scanned by CamScanner

~~Ig~~ A solution to critical section problem must ~~satif~~ satisfy 3 requirements :-

1) mutual exclusion - If process P_i is executing in its critical section then no other processes can be executing in their critical sections.

2) progress - If no process is executing in its critical section and some processes wishes to enter their critical ~~ct~~ section then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next & this selection cannot be postponed indefinitely.

3) Bounded waiting :- There exists a bound or a limit on the no. of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section & before that request is granted.

- 2 general approaches are used to handle critical section in OS.

i) premptive kernel
ii) non-premptive kernel

i) A premptive kernel allows a process to be premptive while it is running in kernel mode.

ii) A non-premptive kernel does not allow process running in kernel mode to be prempted.

Scanned by CamScanner

Peterson's solution

- It is a software based sol^n and restricted to 2 processes that alternate execution b/w there critical sections and remainder sections.
- The processes are P_0 & P_1 which shares the 2 data items
  ~~int t~~ int turn;
  boolean flag[2].
- The variable turn indicates whos turn it is to enter its critical section. If turn = i then P_i process is allowed to execute its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section

```
do
{   flag[i] = true;   - It means ki ye i process aana chahta h
    turn = j;         - It means turn j ki h.
    while (flag[j] && turn == j)
        critical section
    flag[i] = false;
}
```

To prove that the sol^n is correct we need to show
a) mutual exclusion is preserved
b) progress requirement is satisfied
c) bounding waiting requirement is met.

a) For mutual exclusion - Each process P_i enters its critical section only if either flag[j] = false or turn = i
If both processes execute in their critical section at the same time flag[0] = flg[1] == True, the value of turn can be either 0 or 1 but it cannot be

Scanned by CamScanner

both. Only one of the process successfully executes the while statement hence mutual exclusion is preserved.

b&c) For progress & bounded waiting - We note that process P_i can be prevented from entering the critical section only if it is stuck up in the while statement with condition flag[j] = true && turn == j if P_j has set flag[j] = true and is also executing in its while loop then either turn == i or turn == j.
If turn == i then P_i will enter else P_j will enter. Once P_j exits its critical section then it has to reset its flag to false.
If P_j ~~resets~~ resets the flag then it must also set the turn to i. Thus P_i does not change the value of turn while executing the while loop therefore P_i will enter its critical section after atmost one entry by P_j.

Mutex Locks

→ It is a software solⁿ to critical section problem which is used to protect critical regions and thus prevents the race condition.

→ A process must acquire the lock before entering a critical section and on exit it releases the lock. The acquire fⁿ acquires the lock & the release fⁿ releases the lock.

```
acquire ()
{ while( !available)
        ; /* busy wait */
```

Scanned by CamScanner

```
        available = false;
    }
    release ()
        {   available = true;
        }
```

→ <u>Available is a boolean variable</u> which indicates if lock is available or not.

→ The <u>disadvantage</u> of this method is that it requires busy waiting. While a process in its critical section any other process that tries to enter critical section must loop continuously in the call to acquire fn. This type of mutex lock is also called <u>spinlock</u>.

Semaphores

A Semaphores S is an integer ~~vat~~ variable that is accessed by ~~to~~ 2 atomic operations wait & signal

wait(S)	signal(S)

{ while (S <= 0)
; // busy wait
S--;
}
	{ S++;
	}

→ Wait operation is used to test ~~and~~ and signal operation is used to test the value of semaphores

→ When one process modifies the semaphore value, no other process can simultaneously modify that.

Scanned by CamScanner

* 2 types of semaphore exist
  1) counting
  2) binary

- Counting semaphore can range over unrestricted domain whereas binary semaphore ~~only~~ can range only b/w 0 & 1
- The semaphore is initialized to the no. of resources available each process that wishes to use a resource performs wait operation on the semaphore. When a process releases a resource, it performs a signal operation on the semaphore.

Semaphore implementation

→ When a process executes a wait operation and finds that the semaphore value is -ve or 0 it must wait i.e it blocks itself. A process that is blocked waiting on a semaphore S should be restarted when some other process executes the signal operation.

→ The wake-up operation is used that changes the process from waiting state to ready state and places it in the ready queue.

```
typedef struct
{ int value;
    Struct process *list;
} semaphore;
wait ( Semaphore *S)
    { S -> value --;
        if ( S -> value < 0)
```

Scanned by CamScanner

```
    { add this process to S->list;
      block();
    }
}

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

Deadlock and Starvation

Semaphore implementation with a waiting queue results in a situation where 2 or more processes may wait indefinitely for an event. Event could be execution of signal operation. When such a state is reached these processes are said to be deadlocked.

For eg:- 2 processes P_0 & P_1 each accessing 2 semaphores S and Q which are set to value 1

P_0	P_1
wait (S)	wait (Q)
wait (Q)	wait (S)
⋮	⋮
signal (S)	signal (Q);
signal (Q)	signal (S);

Suppose P_0 executes wait (S) operation and then P_1 executes wait (Q) operation. When P_0 ~~execution~~

Scanned by CamScanner

executes wait(Q), it must wait until P, executes signal(Q)
The set of processes is in deadlock state when every process in the set is waiting for an event that can be caused by only another process in the list.

Classic problems of synchronization

1) Bounded buffer problem -
The producer, consumer processes share the following data structures int n;

```
semaphore mutex = 1;
Semaphore empty = n;   - all buffers are empty
Semaphore full = 0;    - non is full
```

- We assume the pool consists of n buffers each capable of holding one item.
- The mutex semphore provides mutual exclusion for accesses to the buffer pool and is ~~Init~~ initialised to 1.
- The empty & full semaphore count the no. of empty and full buffers.

producer process

```
do
{ /* produce an item in next_produced; */
  wait(empty);
  wait(mutex);   - so that no 2 buffers work at the same time.
  ---

  /* add next produced to the buffer */
  - -

  signal(mutex)
  signal(full)
} while (true);
```

Scanned by CamScanner

consumer

```
do
{ wait (full);
  wait (mutex);
  - -

  /* remove an item from buffer to next consumed */
  - - -

  signal (mutex);
  signal (empty);
  - -

  /* consumed an item in next consumed */
} while (true);
```

2) Reader - writer problem

→ A database is shared among concurrent processes. Some processes wants to read the database and some wants to update the database.

→ If 2 ~~data~~ readers access the share data simutaneously no adverse effect will result.

→ If a writer and reader/writer access a database simultaneously, a problem may occur.

→ Solution is writers should have exclusive access to the shared database. ~~This problem is~~

→ No readers be kept waiting for other readers to finish.

→ If a writer is waiting to access the object, no new readers start for reading.

Above solution may result in a problem starvation to the

Scanned by CamScanner

writers and starvation to the readers.

```
Semaphore rw_mutex = 1;
          mutex = 1;
int read_count = 0;
```

→ Mutex semphore ensures mutual exclusion when variable read count is updated (read_count keep tracks of how many processes are currently reading the object

→ rw_mutex ensures mutual exclusion for writers.

writer process

```
do
  { wait (rw_mutex);
    - -
    /* writing is performed */
    - -
    signal (rw_mutex);
  } while (true);
```

Reader process

```
do
  { wait (mutex);   — (mutual exclusion b/w reader & writer)
    read_count ++;
    if (read_count >=1)   (one reader) then stop the writer
        wait (rw_mutex);
    signal (mutex);
    - - -
  /* reading is performed */
    - -
  wait (mutex);
  read_count --;
```

Scanned by CamScanner

```
if( read_count == 0)
    signal (rw_mutex);
    signal (mutex);
} while(true);
```

— X —

Scanned by CamScanner