Introduction :-

→ If a user wants to use run any application / program like a browser, powerpoint or a text editor on a computer system, he might use the resources of the computer system like the memory, CPU, Monitor, Keyboard, hard disk etc

- memory → where app will be loaded
- CPU → execution
- Monitor → display the output
- Keyboard → to give input
- hard disk → to store

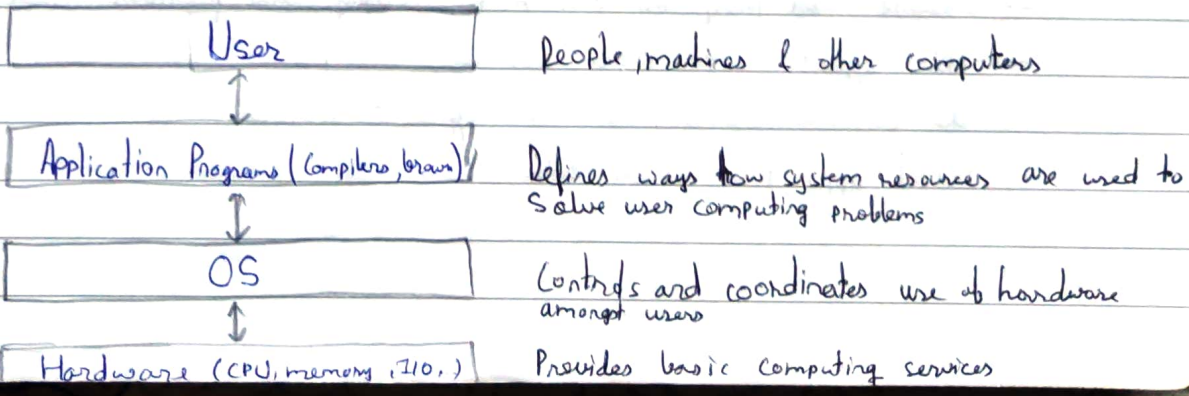
Ex printf("Hello World"); → How is the string displayed on monitor screen?

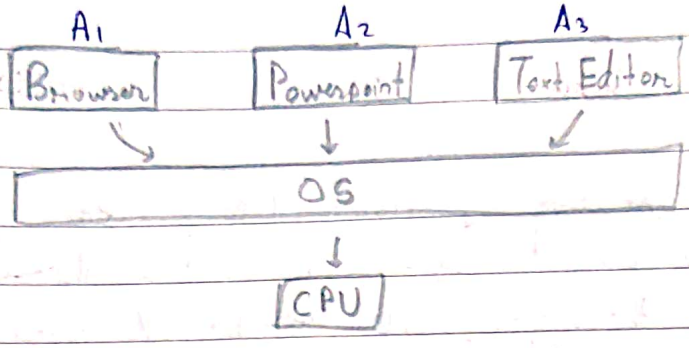
Main Memory → Processor → Graphics Card → Monitor

"Hello World" (Main Memory) → "Hello World" + attributes (font, color...) (Processor)

Now, this is a complex and hardware dependent process, user has just simply written one command printf which is taking care of this complex process.
Now, Suppose the user was asked to take care of the display part as well then instead of writing this printf statement, he would have to write a complete program to handle the output device.
But, this printf statement can take care of this entire process because of manager of the resources Operating System.

Thus, the OS is providing an abstraction to the user so that the user is not bothered with the handling of the various devices / resources of the system.





→ Any application which requires to use any hardware resource would have to send a request to the OS and the OS would manage the resources on behalf of that particular application.

→ Now, suppose if the user had direct access to the hardware resources of the system i.e. OS is not needed / present. Now, this user could take control of the processor and would not release it with the result that other user / applications would be waiting in queue for this particular application to release the processor.

→ With the availability of the OS, it makes sure that all the users get a fair usage of the system resources.

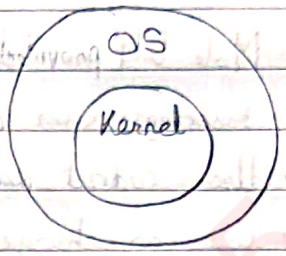
→ OS also makes sure that address space of application A1 is not accessed by A2 or A3, thus providing privacy and security between the applications and it runs the applications in isolation.

⇒ Operating System: A program that acts as an interface between user of a computer and computer hardware.

⇒ Goals of Operating System:

→ Execute user programs and make solving user problems easier.
→ Allow applications to share resources.
→ Use computer resources in an efficient manner.
→ Protect apps from each other.

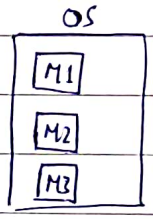
Dual Mode of Operating System :-



⇒ Kernel :

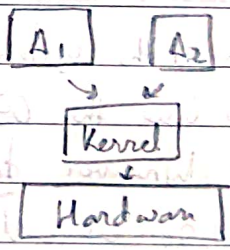
→ It is the core component of the OS.
→ OS is a software and Kernel is that part of the software which takes care of the imp necessary tasks of the OS.
→ It is the first program of OS to be loaded.
→ It is always residing in the memory.
→ It has direct access to hardware.

⇒ Modular approach :



→ Software as a one large program contains all modules for different tasks.
→ An update in one part requires update in the whole.
→ Difficult to maintain
→ Trust issues

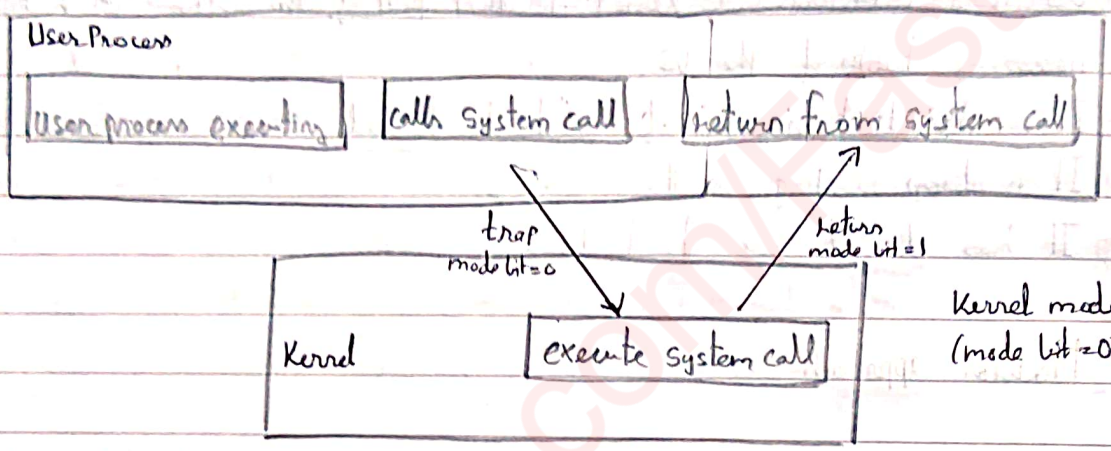
So, we use Modular approach
→ It can be implemented & maintained independently.
→ Interface with OS through device drivers.



⇒ Dual-mode Operation :

→ Users and OS share hardware and software resources.
→ We must ensure that any program cannot cause other programs or OS to execute incorrectly.
→ So, How to distinguish between execution of operating-system code & user-defined code Using Dual-mode operation (User mode & Kernel mode)

→ Mode bit provided by hardware provides ability to distinguish when system is running user code or Kernel code.
→ User cannot access the mode bit.
→ OS can change mode to 'Kernel' and change it back to 'user'.
→ Some instructions designated as privileged are only executable in Kernel mode.



→ Suppose a user process is executing. If in any case it wants to use the resources of the system, it will have to request the OS.
→ Any request the user process makes to the Kernel is referred to as the system call.

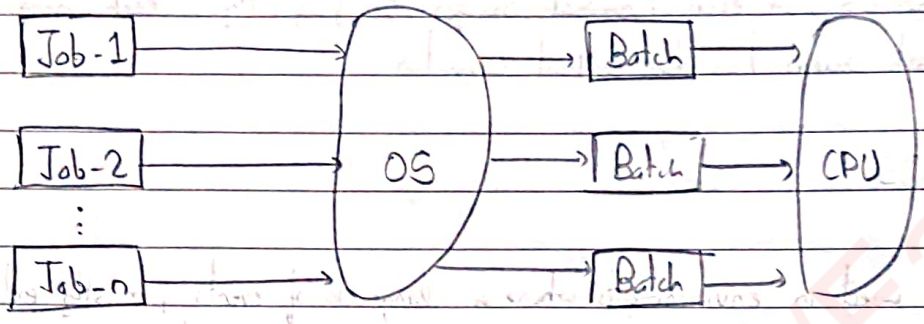
→ At the system boot time, hardware starts in 'Kernel' mode (mode bit=0)
→ Once the OS is loaded it will start the user applications in 'user' mode (mode bit=1).
→ Whenever trap / interrupt occurs, hardware switches from 'user' to 'Kernel' mode so OS will gain control of computer.
→ System always switches to 'user' mode before passing control to a user program.

★ Functions of the OS:-
★ Types of OS:-

1) Batch:

→ This type of OS does not interact with computer directly.
→ There is an operator which takes similar jobs having the same requirement

and group them into batches.
→ It is the responsibility of the operator to sort jobs with similar needs.



2) Multiprogrammed:

→ Multiprogramming is a variation of batch processing in which the CPU is kept busy at all times.
→ When a process completes its I/O in a multiprogramming environment, the CPU can begin the execution of other processes. As a result, multiprogramming helps in improving the system's efficiency.
→ When software is run, it is referred to as "Task", "Process" or "Job".
→ When compared to serial or batch processing systems, concurrent program execution reduce system resource usage and increase throughput.

a) Multitasking Operating System:

→ A multitasking OS allows two or more programmes to run simultaneously.
→ This is accomplished by the OS transferring each program into or out of memory one by one. When a program is switched ~~off~~ out of the memory, it is saved on the disc temporarily until it is needed again.

b) Multiuser Operating System:

→ A multiuser OS allows multiple users from various terminals to share the processing time on a certain powerful central machine.

→ The OS achieves this by frequently switching between terminals, each of which is allotted a certain amount of processor time on a central computer.
→ Because the OS on each terminal changes so frequently, each users appears to have constant access to the central computer.

3) Real Time OS:

→ RTOS are used in environments where a large no. of events, mostly external to the computer system, must be accepted & processed in a short time or within certain deadlines. This system is time-bound & has a fixed deadline.
→ Ex: Airline Traffic control systems, Network multimedia systems etc

a) Hard RTOS

→ These OS guarantee that critical tasks be completed within a range of time.
→ Ex: Scientific experiments, weapon systems, ATCS etc.

b) Soft RTOS

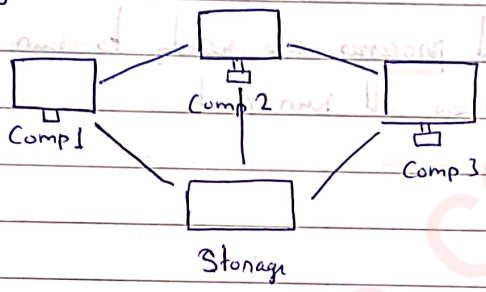
→ This OS provides some relaxation in time limit.
→ Ex: Multimedia systems, digital audio systems etc

4) Distributed:

→ A DOS uses many central processors to support various real-time applications & users.
→ As a result, jobs for data processing are split across the processors.
→ It uses a single communication channel in order to connect several computers.
→ These systems have their own processor & memory.
→ The individual systems that communicate over a single channel are considered as one entity also known as loosely connected systems.

5) Clustered :

→ Clustered OS are a combination of software & hardware clusters.
→ Hardware clusters aid in sharing of high-performance disks among all computer systems, while software clusters give a better environment for all systems to operate.
→ A cluster system consists of various nodes, each of which contains its cluster software.
→ If one of the clustered system's node fails, the other nodes take over its storage & resources & try to restart.



Clustered System

6) Embedded :

→ An embedded OS is designed for use in embedded computer systems. It has limited features.
→ The main goal of designing an embedded OS is to perform specified tasks for non-computer devices.
→ An embedded OS is a combination of hardware & software.
→ It produces an easily understable results to humans in many formats such as images, text & voice.

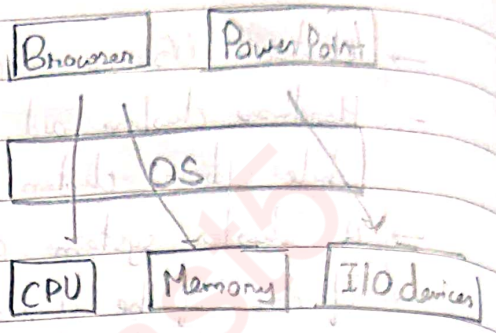
★ Functions of an OS :-

→ When we store our programs, they are stored on non-volatile storage (disk) & wherever, we want to execute them, they will be loaded by the OS into the main memory (RAM). Volatile
→ This program in execution is now referred to as a process.
↓ when in disk
↓ when in RAM

Functions of OS:

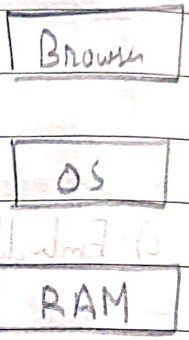
1) Process Management

→ Creating & deleting processes
→ Scheduling processes on CPUs
→ Suspending and resuming processes
→ Providing mechanisms for process synchronization
→ Providing mechanisms for process communication



CPU Scheduling: If several processes are ready to run, system must choose which process will run next.

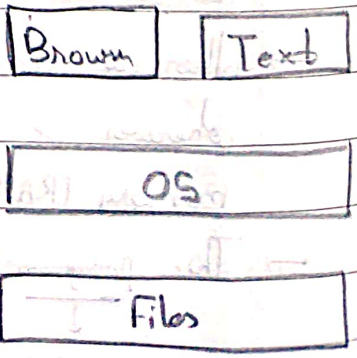
2) Memory Management



→ Keeping record of memory blocks being used & process using them.
→ Allocating & deallocating memory space.
→ Deciding which processes & data to move into & out of memory.

3) Filesystem Management

→ OS provides a uniform, logical view of information storage & this storage unit is referred to as a file
→ So, the OS abstracts all the physical properties of its storage devices (file) from the user.
→ OS also maps the files onto physical media & helps in accessing them via storage devices.



Responsibilities:
→ Creating & deleting files
→ Creating & deleting directories to organize files
→ Supporting primitives (change location / change access) for manipulating files & directories.

→ Mapping & backing files on mass storage.

4) Storage Management (Disc Management)

→ Mouting & Unmounting → when we want to dissassociate any particular storage from the system.
↳ now, the disk is available to the OS, & the OS can access the file system on disk & it can read and write files which are available on the disk.

→ Free-space management.
→ storage allocation
→ Disk scheduling - if there are a no. of processes which want to access the disk & have sent in request, for certain files, then its the job of OS to take care of disk scheduling.

→ Partitioning
→ Protection

5) I/O Management

→ OS hides the distinctness of hardware devices (I - Keyboard, Joystic, O - monitor, printr) from user.
→ Each device has a device driver which knows the uniqueness of that particular device.
→ OS interacts with the device driver to access that particular device, so, the OS is providing the device driver interface to the users for accessing the devices.

6) Protection & Security

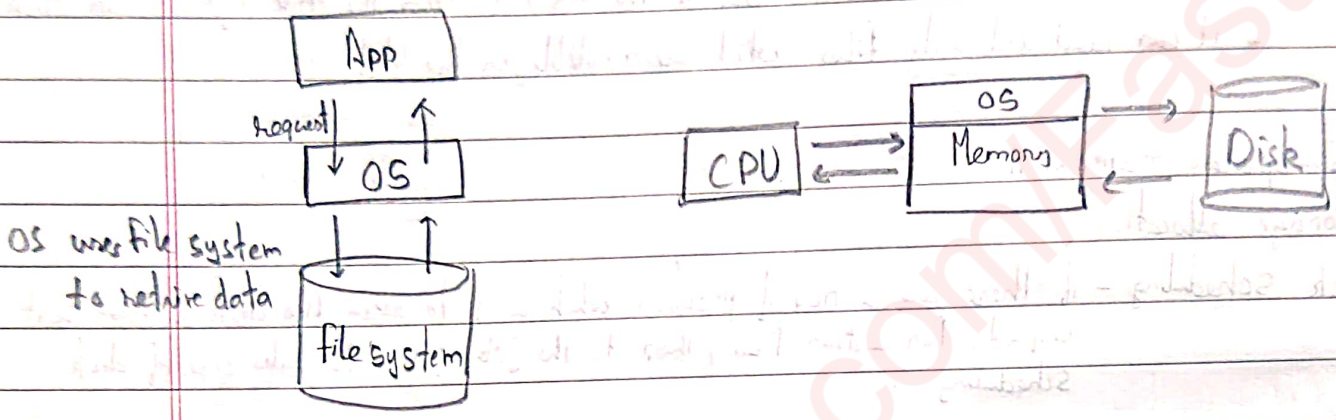
→ It ensure resources can be operated on by only those processes that have proper authorization from OS.
→ Protection - controlling access of processes or users to resources defined by a computer system.

→ A system can have adequate protection but still be vulnerable to failure or attacks.
→ Security - to defend a system from external & internal attacks.

✱ File System : Skel

→ File System is an on-disk data structure & method used by OS to store & retreive data.

→ File system consists of the Files, file names, directory, file descriptor

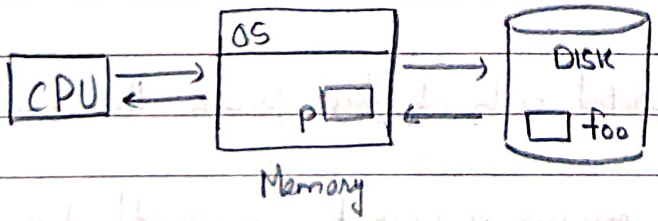


Logical View

Ex: fd = open("foo") → foo is a file available on the disk & I want to open that file, so a request is given to OS, and OS will open the file from the disk & it will return an integer known as file descriptor.

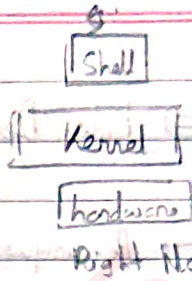
read (fd, p, 500) → Using this fd, we can read by providing fd, a pointer to the buffer in the memory & the size.

write (fd, p, 500) → Similarly write operating can be performed



✱ Shell:

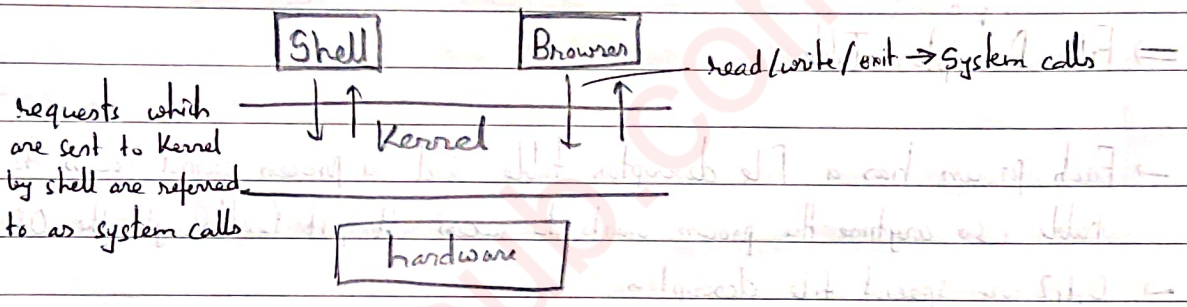
→ Shell that is provided by the unix system is interactive.



→ Shells runs as an application, earlier it was part of Kernel.

→ OS Kernel provides an interface to shell to start a new program.

→ Ex: $ browser
  → Kernel will interpret this command & it will find the file "browser" from the storage
  → Assuming "browser" is in the given directory, it will get loaded & takes control
  → Now, "browser" may open new files, read/write files, and exit when done
  → exit returns control back to shell
  → Shell closes the open files & removes "browser"



✱ System Calls:

→ System calls are the requests which are made to the OS.
→ So, it is an interface to the services which are made available by the OS.
→ These calls are generally available as functions written in C/C++ and certain low-level tasks (involving Kernel to access hardware) may be written using assembly-language instructions.

6 major categories of System calls:

1) Process Control – create, terminate, load, execute, wait, signal
2) File Management – Create, delete, open, close, read, write
3) Device Management – request, release, read, write
4) Information maintainance – get/set time/date, get/set attributes
5) Communications – create/delete connection, send, receive
6) Protection – get/set file permissions

⇒ File Descriptor:

→ A unique non-negative integer to describe an open file or I/O resource in system is known as File Descriptor.

→ This number describes the resource & how it can be accessed.

→ Whenever a process requests for any resource:
- Kernel grants access (if req is valid)
- Creates entry in the global file table → for the OS
- Provides process with the location of that entry through the fd

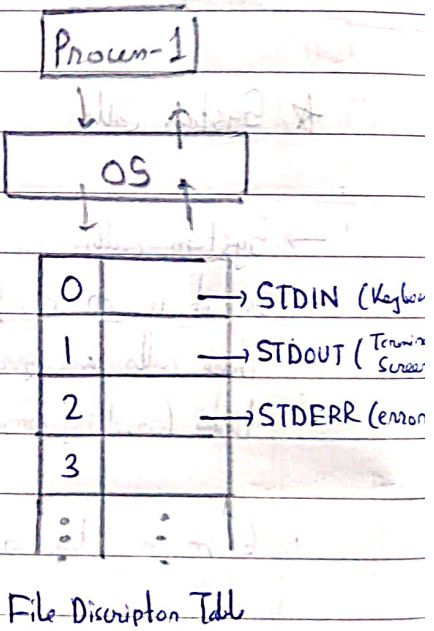
⇒ File Descriptor Table:

→ Each process has a file descriptor table and a process cannot access its file descriptor table, so anytime the process wants to access it, it is through the OS only.

→ 0,1,2 are special file descriptors.

Ex: read (0, buf, 50) → giving a system call for standard input, so the process wants to read from std input (Keyboard), buf is where it is to be put & 50 bytes have to be read

write(1, "hello", 5) → write to fd 1 i.e. process wants to write onto the terminal screen & "hello" is the string to display & 5 is the size of the string to be displayed.

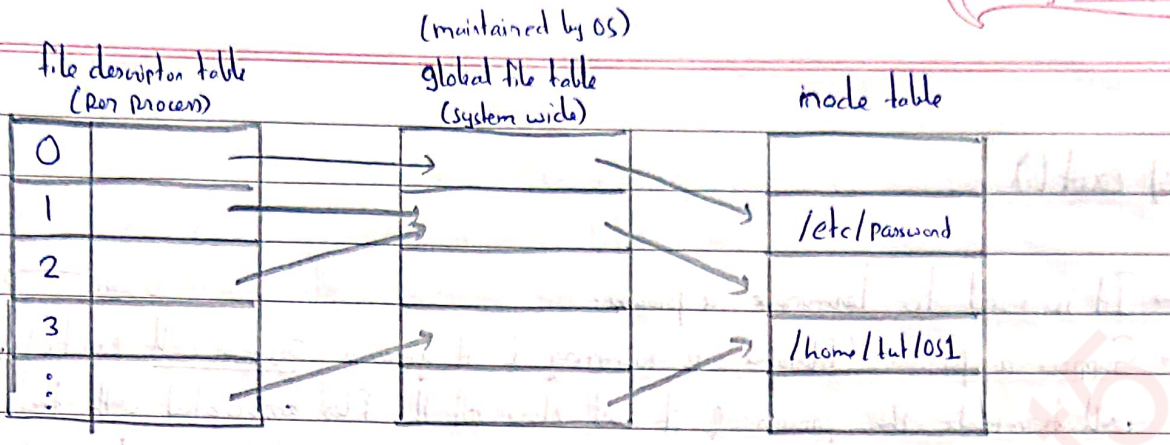


File Discripton Table

→ A process can manipulate files through open & close system calls.

→ So, wherever a process call open(): • OS looks for first available key (fd) in fd table
• Assigns it as file descripton

Ex: open("foo") = 4
↑ fd (available in table)



Global file table:

→ It will contain information about the file like the mode (read/write/append etc), inode of the file, byte offset (where is the current pointer or the offset of the file where it is being written or being read from) and the access restrictions (whether a file is read-only / write-only / has both permissions).

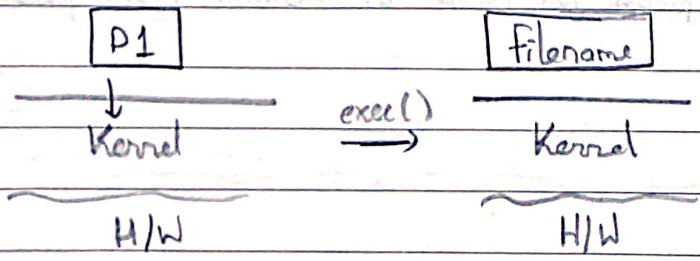
Inode table: Describes the actual underlying files.

→ Multiple file descriptors can refer to the same entry in global file table.
→ Multiple global file table entries can refer to same inode.

⇒ System Calls:

1) exec()

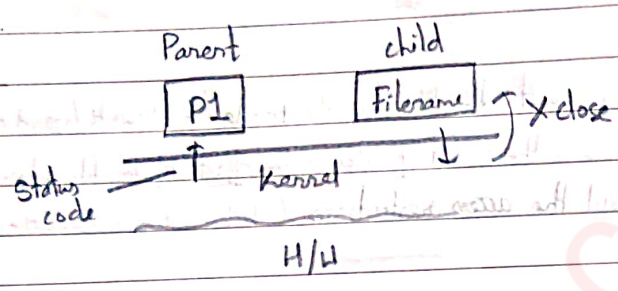
→ Suppose a process P1 is running & it calls the system call exec("filename").
→ So, what the kernel will do is that it will replace this process P1 by the file given as the argument.
→ Note that this executable program "filename" on disk is converted to process in memory.



2) exit()

→ It is used to terminate a process.

→ Suppose a process "filename" is running & it gives a system call exit(), so the OS will terminate this process & it will close all the files associated with this process & it will also return an exit status (int) to the OS which will be sent to its parent process.



Note: Suppose there was a process P1 & it called exec("filename") i.e. P1 has now been replaced by the process "filename". Now, if this process calls exit(), control will not go back to P1 because P1 is not a parent of "filename".

P1 will not get the status code

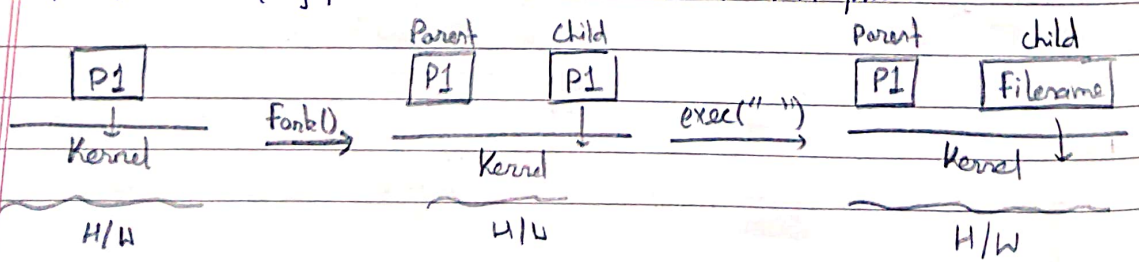
3) fork()

→ Suppose a process P1 is running & it makes a system call fork() to the Kernel, then it creates an identical copy of the process.

→ So, we say that P1 has created a copy child or has forked a child.

→ Now, if the child process calls exec("filename") then it will be replaced by the "filename".

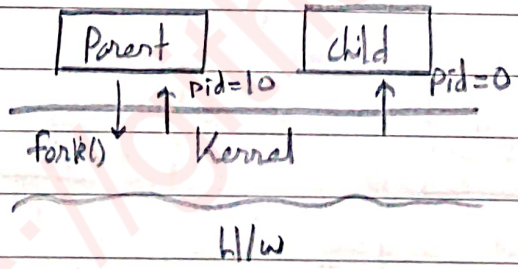
→ So, using fork(), any process can create an identical child process.



→ Now, when the "filename" calls exit(), it gets removed & control goes back to its parent process P1 & also receives a exit status code.
→ Parent then continues from where it left off

pid = fork() → this fork system call returns an integer which is of process id type

→ We know that each process has a process id (pid).
→ So, whenever the parent is giving the fork system call two processes are returning from the fork system call. one return value is sent to the parent & another value is returned to the new process (child process).
→ Now, using the return values, we can distinguish whether it is a parent process or child process.
→ After creating a child, parent is going to wait & the child process can run & call the exec() system call to replace this process with another program & the new process will run & the parent will wait for the child to terminate & once the child terminates, it will start running again.



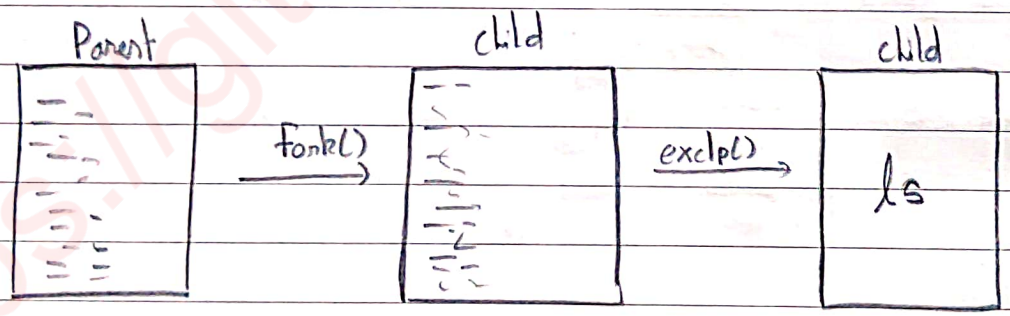
```
int main(){
    fork()
    fork()
    fork()
    fork()
    return 0;
}
```

⇒ How many process will be created?

$2^4 = 16$

```
int main(){
    pid_t pid;
    pid = fork();                          → fork child process

    if(pid < 0){
        printf("Fork Failed");             → error (no child will be created)
        return 1;
    }
    else if(pid == 0){
        execlp("/bin/ls", "ls", NULL);     → child process
    }                      I       exit status will be returned to parent
                           running this process
    else{
        wait(NULL);                        → parent waits for child to complete
        printf("Child process has completed");   → Parent process
    }
    return 0;
}
```



→ So, we have process id of type process id type & then the parent process has given a system call fork().

→ This fork() will create a child process (of exactly same code & type as parent).

→ Now, this fork system gets return values (pid), one to parent (pid > 0) & another to child (pid = 0)
of its child

→ When the fork() instruction was being executed, the program counter of the parent was pointing to the next instruction and when the child is created the PC value

would also be exactly the same for the child.

→ So, once this fork has been executed, now either the parent can run or the child can run.

→ Suppose the parent process is running.

→ Since its pid (of its child) is >0 and now it will wait for some exit status (NULL) which will be returned by the child process.

→ Suppose the child process is running.

→ Since its pid is zero, it will execute : execlp("/bin/ls", "ls", NULL);

→ Now, the original program (child) which was the copy of the parent has been replaced by this "ls" program & this program will become a process and after running the process, there will be an exit call in the process to terminate which will return a NULL value.

→ Since, the parent was waiting for the NULL value, the parent will start running & will start executing its remaining instruction & finally it will terminate.

b) open()

→ This system call is used to provide access to a file / resource.

→ So, if there is a process and it wants to open a new file / wants to access a particular resource, it will send this request to the kernel by means of this particular system call.

→ When the OS receives this request, it will allocate resources to the file & it will provide a handle to the process, which will be used by the process to refer to this file.

fd = open("filename", mode)

fd: handle (integer); mode: read / write / append

→ The process gets a return value of -1 if the system calls fails.

→ This fd can be used as argument for future system calls to perform read, write etc operations.

5) close()

→ This system call is used to terminate the access to a file
→ So, wherever a process sends this system call that means the file is no longer required by the process now.
→ So, the Kernel will flush out all the buffers, meta data associated with the file will also be updated and the resource associated with the file will be de-allocated.

close(fd) → returns 0 if file closed successfully
↳ returns -1 if any error occured

6) read()

→ This system call is used by a process to access data from a file

ret = read (fd, buf, size)
- fd: to refer the file to read
- buf: where the read data has to be stored
- size: Number of bytes to be read from the file

→ If -ve value is returned then read is not possible.
→ OS will also take care of access control (permission to read).

7) write()

→ This system call is used by a process to write data from buffer into a file.

ret = write (fd, buf, size)

→ If -ve value is returned then write is not possible.
→ OS will also take care of access control (permission to write)

8) dup()

→ This creates a copy of a file descriptor.

fd1 = dup(fd)

→ It uses lowest-numbered unused descriptor of in the descriptor table for the new descriptor.

→ If copy is successfully created:
- Original and copy file descriptors may be used interchangeably.
- Both refer to same open file
- Share file offset & file status flags

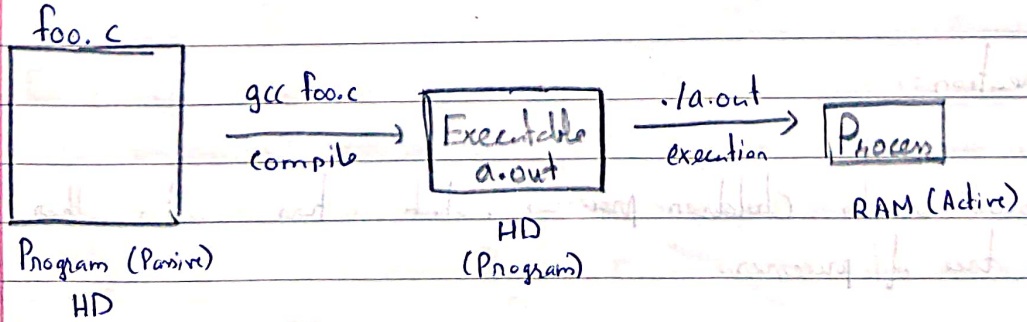
close(1) → closing standard otput
fd1 = dup(1) → anything written to stdout now goes to file pointed by fd1
↓
=1

★ Process Management :-

⇒ Introduction to Process:

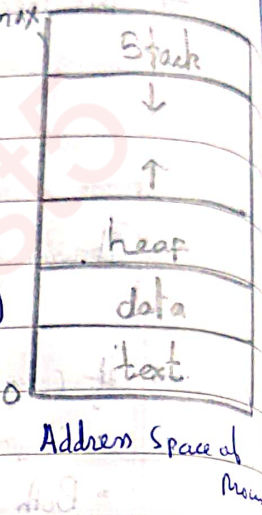
→ Any program or application is stored on a secondary storage (harddisk) & it is in passive form (high-level language program or an executable file).
→ Now, when this file is loaded into the RAM for execution, it will become active and referred as process.

↓ starts
via GUI mouse clicks, command line entry of this file



→ Process is a program in execution.

→ Now, when this process goes into the RAM then each process is assigned a memory map.



Memory Map consists of:

a) Text - Executable code

b) Data - Global variables

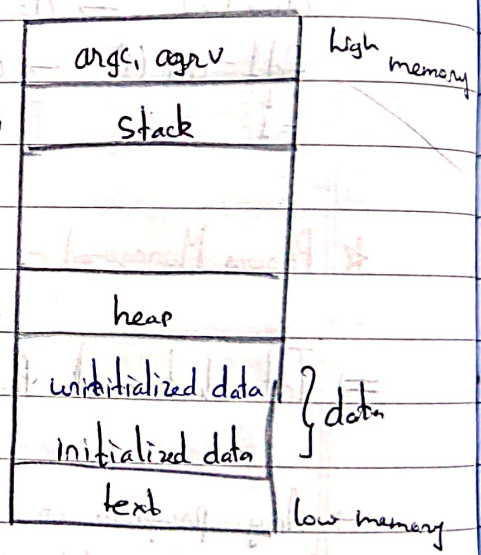
c) Stack - Temporary Data (Function parameters, return addresses, local variables)

d) Heap - memory dynamically allocated during run time

Ex:
```
#include <stdio.h>
#include <stdlib.h>

int x;
int y=25;

int main (int argc, char *argv[]){
    int *val;
    int i;
    val = (int*) malloc (sizeof(int)*5);
    for(i=0; i<5; i++){
        val[i]=i;
    }
    return 0;
}
```



⇒ Process Creation:

→ Parent process creates children processes, which in turn, creates other processes, forming a tree of processes.

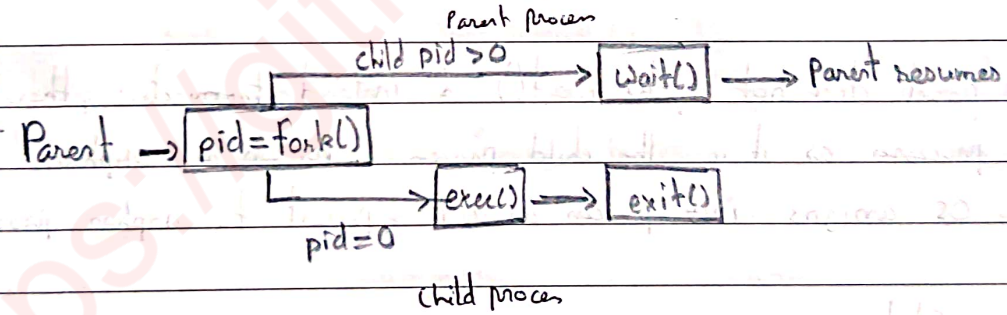
→ Each process that is created is given a unique no. by the OS which is known as Process Identifier (Pid). and process is identified & managed via this pid.

→ Address-space of child process : a) child is duplicate of parent
b) child loads a new program

→ Execution options : a) Parent & child execute concurrently
b) Parent waits until children terminate

→ Resource sharing options : a) Parent & children share all the resources
b) Children share subset of parent's resources
c) Parent & child share no resources

→ Unix system calls : a) fork() - Creates new process
b) exec() - used after a fork() to replace process's memory space with new program.
c) wait() - used by parent process to wait for child to terminate



⇒ Process Termination:

→ When process executes its last statement, it requests OS to delete it using the exit() system call.

→ After termination of a child process, OS returns a status code to parent (via wait() system call used by parent), so that parent can now continue its execution.

→ When exit() is used by a process, its resources are deallocated by the OS, however, its entry in process table must remain until parent calls wait(), because process table contains process's exit status.

→ Parent may also terminate execution of its children processes using abort() system call.

→ Some reasons for doing so:

a) child has exceeded allocated resources

b) Task assigned to a child is no longer required

c) Parent is terminating, & OS does not allow a child to continue if its parent terminates.

Cascading termination: Some OS do not allow child to exist if its parent has termin
(All children, grand-children etc are terminated)

→ Parent process may wait for termination of a child process by using wait()
pid = wait (& status) → call returns status information & pid of terminated process

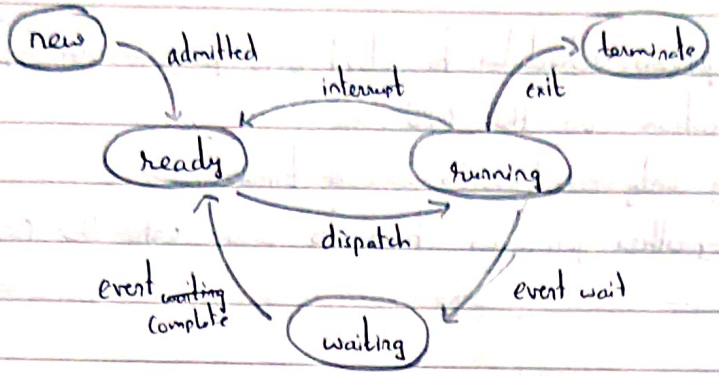
→ If a child process terminated but the parent has not yet called wait(), then the child process is a zombie process.

→ If a parent did not invoke wait() & instead terminated, thereby leaving it's child processes as it is, the child process is known as orphan. In this case, some OS assigns init process as a new parent to orphan processes.

⇒ Process State

→ When a process is executing it changes its state.

States: a) New: Process being created
b) Ready: Process is in the RAM & waiting to be assigned to a processor
c) Running: Process has been assigned to a CPU & instructions are being executed
d) Waiting: During execution, process may have to wait for some event (I/P)
e) Terminated: Process has finished execution



⇒ Procen Control Block (PCB):

→ When a procen is created, the OS has to maintain information about the procen & so all this information is maintained in a PCB.

PCB contains:

a) Procen State - state of procen at that time
b) Procen number - unique id assigned to the procen to identify it
c) Program counter - address of the next instruction to be executed
d) CPU registers - contents of registers being used by procen
   - general purpose registers, special purpose (stack pointer, flag)
e) CPU scheduling information - priority: pointer to scheduling queue & others
f) Memory Management information - memory allocated to procen
   - base & limit address (registers)
   - page tables
g) I/O status information - I/O devices allocated to procen
   - list of open files etc
h) Accounting information - CPU used, clock time elapsed since start
   - time limit assigned to process

Procen state
Procen number
Program counter
registers
scheduling
memory limits
list of open files
...

PCB

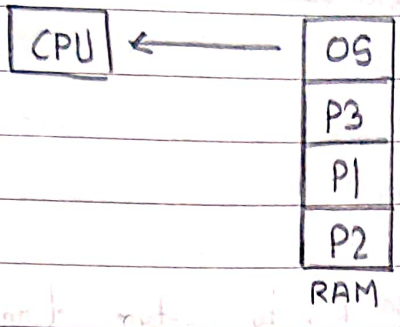
⇒ Procen Scheduling:

→ Number of procenes currently in memory is known as degree of multiprogramming.
→ The main objective of multiprogramming is to have some procen running at all times to maximize CPU utilization.

→ So, it is the job of the process scheduler to select one process amongst the available processes for the next execution on the CPU core.

→ Each CPU core can run only one process at a time.

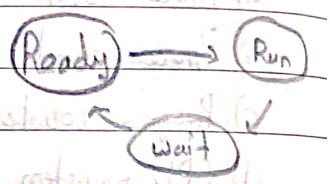
→ If there are more processes, other processes have to wait until the core is free.



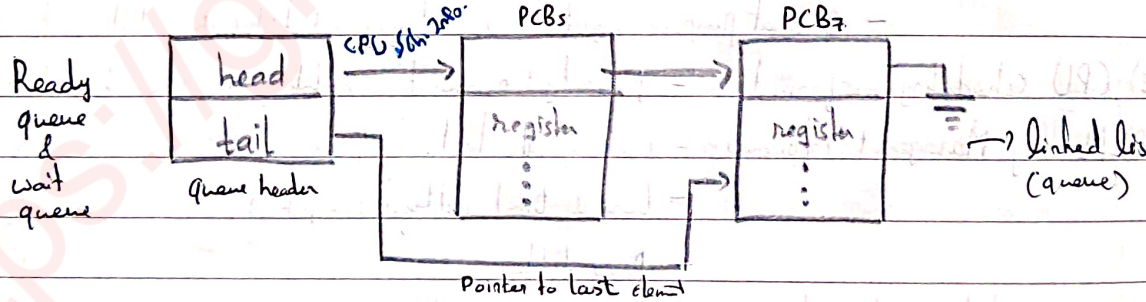
→ The scheduler maintains scheduling queues of processes:

a) Ready queue - set of all processes residing in main memory, ready & waiting to execute.

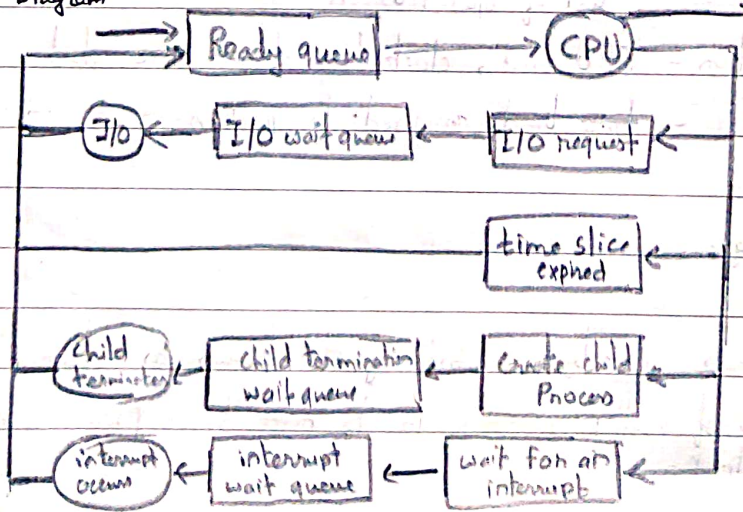
b) Wait queue - set of processes waiting for an event



→ Processes can migrate among various queues.



Queueing Diagram



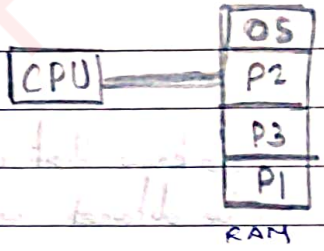
When one process goes for some other event, this CPU will be now allocated to some other process in ready queue by the scheduler

⇒ Context Switch:

→ The job of CPU scheduler is to assign the CPU to one of the processes which are available in the ready queue.

→ Suppose a process P2 was running on the CPU & note it has to go for an event / time splice of P2 is expired, so the CPU will be released & it has to be assigned to some other process by the scheduler.

→ When CPU switches from one process to another: (stored in register)
- System must save state of old process (state save)
- Load saved state for new process (state restore)
- This task is known as a context switch.



→ Context of a process is represented in the PCB. and the system is switching from one context to another context.

→ Context-switch time is pure overhead. System does no useful work while switching. so, More complex the OS & PCB, longer time the context switch will take.

→ Context-switch time is sometimes hardware dependent also. (when only one register file is used for multiple processes, so when context switching takes place, the value of registers must also be changed according to particular process).

1) Voluntary Context switch

→ When a process has given up control of CPU.

→ Because it requires a resource that is currently unavailable.

→ Ex: Waiting for I/O.

2) Non Voluntary Context switch

→ When CPU is taken away from a process.

→ Ex: when its time slice has expired or it has been preempted by a higher priority process.

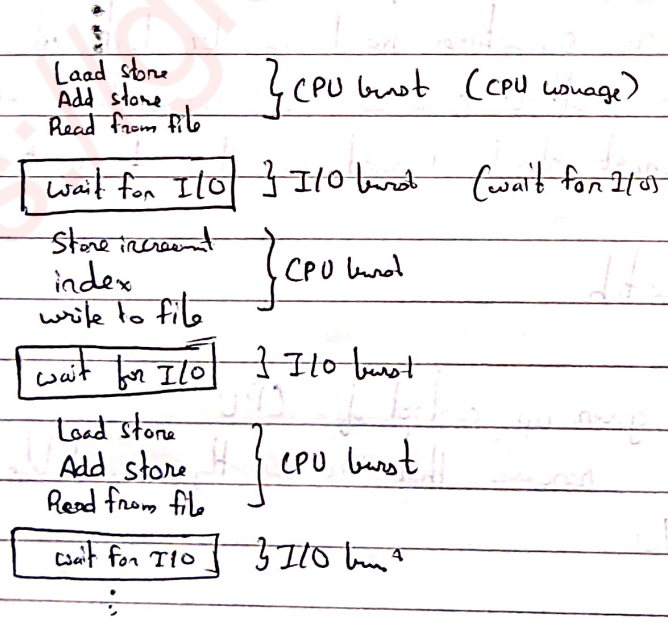
Most processes can be described as: I/O Bound & CPU Bound

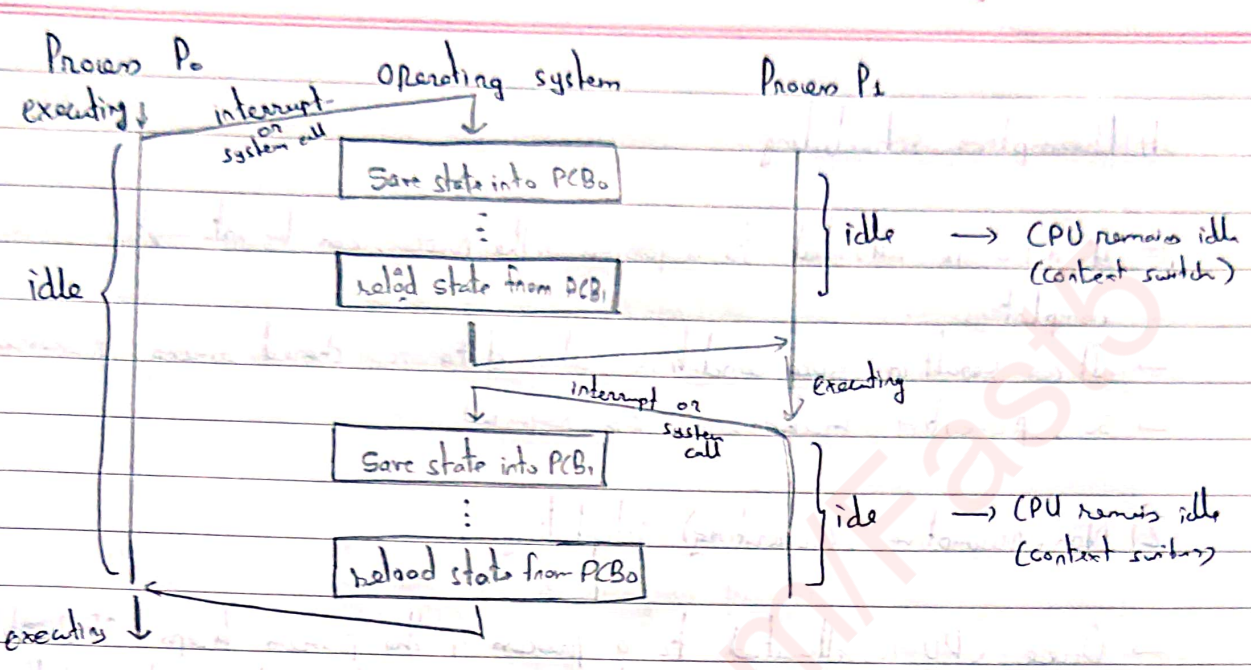
1) I/O-bound process: Spends more of its time doing I/O than doing computations.
Many short CPU bursts (short time using CPU)

2) CPU-bound process: uses most of its time doing computations & generates very few I/O requests.
Few long CPU bursts

We know that our objective is to maximize the CPU utilization which can be obtained with multiprogramming (having multiple programs in the main memory)

CPU-I/O Burst Cycle for a process:
→ Process execution - cycle of CPU execution and I/O wait.
→ CPU burst followed by I/O burst.





executing → CPU burst
interrupt / system call → I/O burst

⇒ CPU Scheduler:

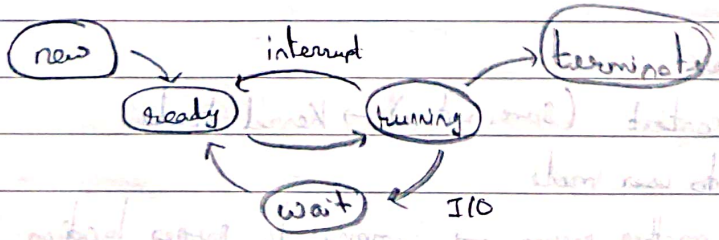
→ We know that the job of the CPU scheduler is to allocate the CPU to one of the processes available in the ready queue. So, it will select one of the process from the ready queue & allocate the CPU to it.

→ This ready queue can be ordered in various ways depending upon the scheduling algorithm.

To other waiting running by current process or select a diff one

CPU scheduling decisions may take place when a process:

→ Switches from running to waiting state
→ Switches from running to ready state.
→ Switches from waiting to ready state.
→ Terminates



A situation when several processes access & manipulate the same data concurrently & the outcome of the execution depends on the particular order in which the access takes place, is called a race condition

1) Preemptive scheduling:

→ If CPU is allocated to a process, the process can be interrupted, even before its completion.
→ It can result in race condition when data is shared among processes.
→ So, preventive mechanisms are required

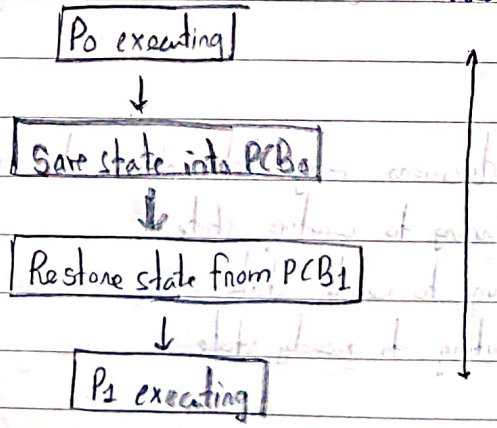
2) Non-preemptive (Cooperating) scheduling:

→ Once CPU is allocated to a process, the process keeps CPU until it releases it either by terminating or by switching to waiting state (I/O). (Voluntarily)

⇉ Dispatcher:

→ Once the CPU scheduler selects the process from the ready queue to give control of CPU; it is the job of Dispatcher to give control of the CPU to the selected process.

Ex: P_0 was executing but then CPU scheduler switches/selects process P_1.



It involves:
1) Switch context (save, restore) → Kernel Mode
2) Switching to user mode
3) executing another process → jumping to proper location in user program to restart that program

Dispatch latency: time taken by the dispatcher to stop one process & running another process.

* Dispatch latency should be as minimum as possible so that the system can quickly switch from one process to another.

⇒ CPU scheduling criteria:

1) CPU utilization (to keep CPU as busy as possible)

→ Conceptually, can range from 0-100%.
→ In a real system, range from 40% (for lightly loaded system) to 90% (for heavily loaded system).

2) Throughput

→ No. of processes that complete their execution per unit time.
→ For long processes, may complete only one process per sec (Throughput ↓)
→ For short processes, may be able to complete 10s of processes per sec (Throughput ↑)

3) Turnaround Time (TAT)

→ Amount of time to execute a particular process.
→ Interval from getting into ready queue to completion of a process.
→ Sum of periods spent waiting in ready queue, executing on CPU & doing I/O.

4) Waiting Time

→ The amount of time a process has been waiting in the ready queue.
→ Sum of all the periods spent waiting in ready queue.

5) Response Time

→ Amount of time from when request was submitted until first response is produced.
→ In an interactive system, TAT may not be best criterion.
→ A process can produce some output fairly early & can continue computing new results while previous results are being output to user.

So, for optimization of scheduling it is desirable to:

→ Maximize CPU utilization & Throughput ✓
→ Minimize turnaround time, waiting time, response time ✓
→ But in most cases, we optimize the average measure (all)
→ Under some circumstances, we prefer to optimize minimum or maximum values rather than average.
→ Ex: to guarantee that all users get good service, we can minimize the maximum response time

⇒ First Come, First Served (FCFS) scheduling:

- Non preemptive
- Dynamic (Arrival time)
- On line
- Not optimal

→ To schedule processes in the order in which they arrive in ready queue.
→ Easy to implement
→ Poor in performance due to high waiting times.
→ Primarily non-preemptive scheduling algorithm.

(CT-AT) (TAT-BT)

Ex:

Process	Burst Time	Arrival Time	Completion Time	TAT	Waiting Time
P1	6	2	17	15	9
P2	2	5	23	18	16
P3	8	1	11	10	2
P4	3	0	3	3	0
P5	4	4	21	17	13

Non-preemptive

Ready queue | P4 | P3 | P1 | P5 | P2 |
0 1 2 6 5

Gantt Chart | P4 | P3 | P1 | P5 | P2 |
0 3 11 17 21 23

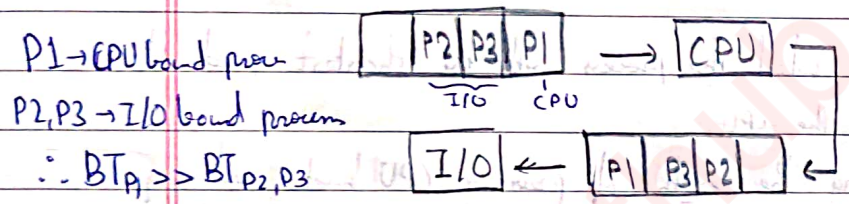
Average Waiting Time = $\frac{9+16+2+0+13}{5} = 8$

Average TAT = $\frac{15+18+10+3+17}{5} = 12.6$

Imp

Problems with FCFS:

→ Average waiting time may vary substantially if CPU burst times of processes vary greatly.
→ Assume: One CPU bound process & many I/O bound processes.



When P1 enters the CPU, its CPU burst time will be long, so P2, P3 will have to wait for a long time. Now, suppose it requires I/O, it goes into ready queue & since P2, P3 are I/O bound processes, they will also enter in ready queue (after going into I/O) again waiting for process P1 to complete. So, everytime there is high waiting time for P2 & P3.
This is known as Convoy Effect.

→ All other processes wait for one big process to get off CPU.
→ Results in lower CPU & device utilization than might be possible if shorter processes are allowed to go first.

Shortest Job First scheduling:

- Non preemptive or preemptive
- Static (Ci → constant)
- on line or off line
- Minimizes the avg response time (x optimal)

→ In this algorithm, we consider the length of the next CPU burst for each process & whichever process has the shortest next CPU burst, it will be scheduled first.

→ But the problem with this algorithm is that the length of the next CPU request is not known. So, we can either ask the user or we can estimate what will be the length of the next CPU burst.

→ If the length of the next CPU bursts of two processes are same then we can use FCFS to break the tie.

→ SJF - preemptive or non-preemptive

→ SJF is optimal - gives minimum average waiting time for a given set of processes

Determining length of next CPU burst:

→ Since we do not know the length, we can predict its value because we can expect that the next CPU burst of that process will be similiar in length to the previous ones.

→ Once we have estimated, we can select the process with the shortest predicted next CPU burst for execution on the CPU.

→ This prediction is done by using the length of previous CPU bursts and using exponential averaging.

Let t_n → length of n^{th} CPU burst - contains most recent information (process)

τ_n → predicted length of n^{th} CPU burst - contains history of predicted information

τ_0 → initial (a constant or an overall system average)

τ_{n+1} → predicted value for next CPU burst

$(0 \le \alpha \le 1)$ α → parameter - Controls relative weight of recent & past history in prediction

$$\boxed{\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n}$$

If $\alpha = 0$, $\tau_{n+1} = \tau_n$ (here most recent history cannot affect)

If $\alpha = 1$, $\tau_{n+1} = t_n$ (here most recent history only affects)

∴ Commonly $\alpha = 1/2$, so recent history & past history are equally weighted.

Non-Preemptive SJF

Ex:

Process	Burst Time	Arrival Time	Completion Time	TAT	Waiting Time
P1	6	2	9	7	1
P2	2	5	11	6	4
P3	8	1	23	22	14
P4	3	0	3	3	0
P5	4	4	15	11	7

Avg TAT = 9.4 Avg WT = 5.2

Ready Queue:

AT 1 2 4 5

P3	~~P1~~	P5	~~P2~~

BT 8 6 4 2

Gantt Chart

P4	P1	P2	P5	P3

0 3 9 11 15 23

Preemptive SJF / Shortest Remaining Time First (SRTF)

Ex:

Process	Burst Time	Arrival Time	Completion Time	TAT	Waiting Time
P1	6	2	15	13	7
P2	2	5	7	2	0
P3	8	1	23	22	14
P4	3	0	3	3	0
P5	4	4	10	6	2

Avg TAT = 9.2 Avg WT = 4.6

AT 1 2 4

~~P3~~	~~P1~~	~~P5~~	~~P2~~

BT 8 ~~5~~ ~~3~~

Gantt Chart:

P4	P1	P5	P2	P5	P1	P3

0 3 4 5 7 10 15 23

2<2
1<6
5>4
3>2
1<2
1<1
1<3

⇒ Round Robin (RR) Scheduling:

→ Here we define a small unit of time, time quantum / time slice (10-100 ms)
→ Ready queue is treated as a circular queue.

→ Scheduler goes around the ready queue, allocates CPU to process, sets timer to interrupt after 1 time quantum & dispatches process.
→ Treats ready queue as a FIFO queue of processes & new processes are added to the tail of the ready queue.
→ RR is FCFS scheduling with preemption, thus enabling system to switch between processes.

Two possibilities of process CPU burst:

i) Less than 1 time quantum
→ Process itself releases CPU voluntarily.
→ Scheduler proceeds to next process in ready queue.

ii) Longer than 1 time quantum
→ Timer goes off & causes an interrupt to operating system
→ Context switch executed
→ Process put at tail of ready queue.
→ CPU scheduler selects next process in ready queue

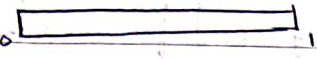
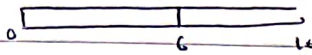
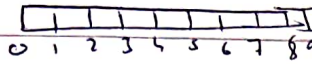
⇒ If there are 'n' processes in the ready queue and time quantum is 'q':

→ Each process gets '1/n' of CPU time in chunks of at most 'q' time units
→ No process waits more than '(n-1)q' time units.

Performance:
q → large ⇒ FIFO ×
q → small ⇒ large number of context switches ×
∴ q must be large with respect to context switch, otherwise overhead is too high.

Note: 80% of CPU bursts should be shorter than time quantum.

(a)

Process time = 10	quantum	context switches
	12	0
	6	1
	1	9

∴ q should be larged compared to context switch time.
If Time Quanta 10 ~ 100ms then Context switch < 10 µs

Ex:

Process	Burst Time	Arrival Time	Completion Time	TAT	Wait Time
P1	6 4 2	2	21	19	13
P2	2	5	13	8	6
P3	8 6 4	1	23	22	16
P4	3 1	0	7	7	4
P5	4 2	4	17	13	9

Time stamp / quatum q = 2

queue: P3, P1, P4, P5, P3, P2, P1, P5, P3, P1, P3, P1, P3 (Compl, Compl, Compl, Comp, Compl)

Gantt chart:

P4	P3	P1	P4	P5	P3	P2	P1	P5	P3	P1	P3
0	2	4	6	7	9	11	13	15	17	19	21	23

⇒ Priority Scheduling: - Preemptive
- static or dynamic
- On line

→ Here a priority number is given to each process.
→ CPU is allocated to process with highest priority.
→ Processes having same priority are scheduled according to FCFS order.
→ In SJF - priority was inverse of predicted next CPU burst time.

Priorities can be defined internally or externally.

1) Internally - use some measurable quantity to compute priority
Ex: time limits, memory requirements, no. of open files, ratio of avg I/O burst to avg CPU burst.

2) Externally - set by criteria outside OS (Administrator)
Ex: importance of process, type/amount of funds being paid for computer use, department sponsoring the work

→ Can be preemptive / non-preemptive.
→ When a process arrives at ready queue, its priority will be compared with the priority of currently running process.
→ If the scheduling is preemptive, ~~then the high~~ & priority of newly arrived process is higher than priority of currently running process then the higher priority process will get the CPU.
→ If the scheduling is non-preemptive, & Priority of newly arrived process is higher than currently running process then the new process will be put at the head of the ready queue.

Limitation:

→ Major problem of Priority scheduling is indefinite blocking, or starvation.

Suppose there is a ready queue & processes P1, P2, P3 are lined up & P1 is currently running and P2 is having slightly lower priority & P3 also has more lower priority. So, P2 & P3 are waiting to get CPU. Now, suppose when P1 was running, few more processes P4, P5 entered into the ready queue with higher priority than P2 & P3. So, after P1 finishes P4 & P5 will get the CPU respectively. If this continues P2 & P3 will be in the state of infinite waiting. This situation is known as Starvation.

Solution: Aging

→ To gradually increase the priority of processes that wait in system for a long time.
→ Ex: if priorities range from 127 to 0, periodically increase priority of a waiting process by 1.

Non-Preemptive (Assume higher priority indicated by lower numbers)

Ex:

Process	Burst Time	Arrival Time	Priority	Completion Time	TAT	Waiting Time
P1	6	2	3	9	7	1
P2	2	5	1	11	6	4
P3	8	1	4	23	22	14
P4	3	0	5	3	3	0
P5	4	4	2	15	11	7

Avg TAT = Avg WT = 5.2

Non-Preemptive

queue: ~~P4~~ ~~P1~~ P3

Gantt chart:

P4	P1	P2	P5	P3
0	3	9	11	15	23

Preemptive

5 4 3 2

queue: P4, P3, P1, P5, ~~P2~~

2 7 4 3

Gantt chart:

P4	P3	P1	P5	P2	P5	P1	P3	P5
0	1	2	4	5	7	10	14	21	23

Completion Time	TAT	WT
14	12	6
7	2	0
21	20	12
23	23	20
10	6	2

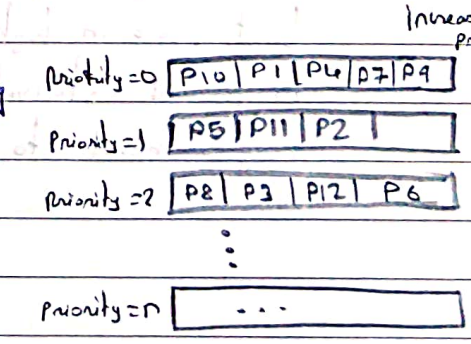
Avg WT = 8

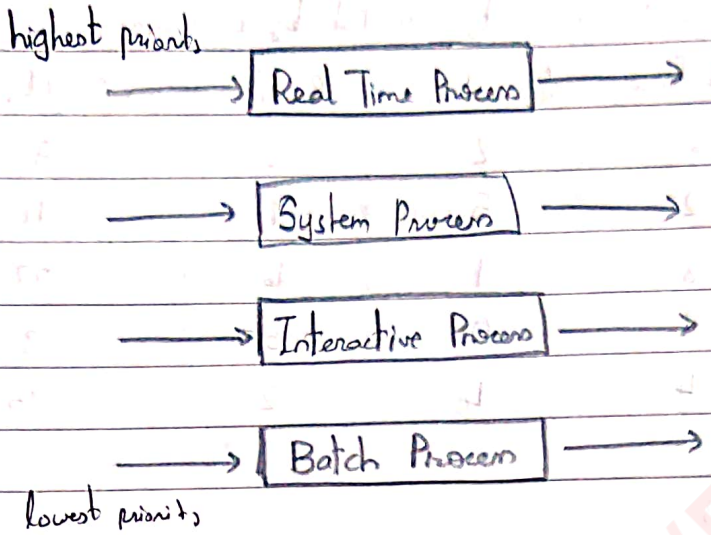
⇒ ~~Multilevel Feedback~~ Multilevel Queue Scheduling:

→ Here we have separate ready queues for each distinct priority.

→ All the processes which are in the highest priority queue they will be scheduled first & the processes which are having a lower priority will be scheduled after all the processes in the higher priority queues have been finished.

→ Each queue might have its own scheduling algorithm for its processes. (RR, SJF, ...)



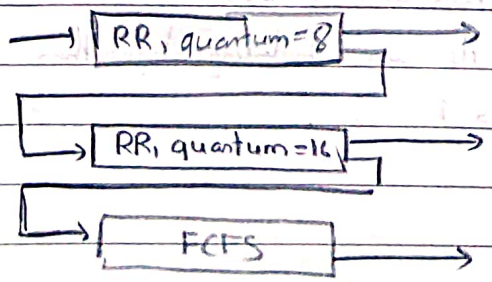


→ Each queue has absolute priority over lower-priority queues. i.e. if a higher priority process enters the ready queue while a low priority process is running, the low priority process will be preempted & higher priority process will be scheduled first.

→ Another possibility is that some kind of time-slice can be done among the queues i.e. each queue gets a certain portion of CPU time, which it can schedule among its various processes.

⇒ Multilevel Feedback Queue Scheduling: (Improvements in above)

→ Here a process can move between various queues.
→ Separate processes according to characteristics of their CPU bursts.
→ If a process uses too much CPU time, move it to a lower-priority queue.
→ I/O bound & interactive processes, typically characterized by short CPU bursts, are in higher-priority queues.
→ To prevent starvation, A process waiting long in a lower-priority queue may be moved to a higher-priority queue. This is known as Aging.



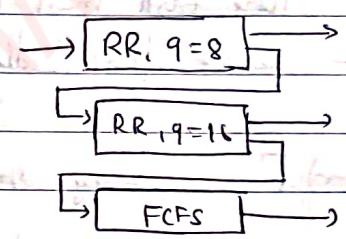
Thread Scheduling & Multiple-Processor Scheduling from pdf
Lateray

Scheduler must define the following parameters:

→ Number of queues
→ Scheduling algorithms for each queue
→ Method is used to determine when to upgrade a process (Low P_R → high P_R)
→ Method used to determine when to demote a process (high Priority → low Priority)
→ Method used to determine which queue, a process will enter when that process needs service.

Ex:

Process	Burst Time	Arrival Time
P1	~~36 24~~	0
P2	~~24 12~~	16
P3	~~12 8~~	20



68
-36
32

Q_1 : ~~P1 P2 P3~~
Q_2 : ~~P1 P2 P3~~
Q_3 : ~~P1~~

Completion Time	TAT	WT
68	68	32
60	44	24
64	45	32

Avg WT = 29.33

Gantt chart:

P3	P1	P2	P3	P1	P2	P3	P1
0	8	16	24	32	48	60	64	68

⇒ Scheduling Real Time Process:

→ Real Time Systems:

→ Event-driven nature.
→ System typically waits for an event in real time to occur.
→ Events may arise in : a) software → ex: when a timer expires
b) hardware → ex: a remote-controlled vehicle detects approaching obstruction
→ When event occurs, system must respond & service it as quickly as possible.
→ For real-time scheduling, scheduler must support preemptive & priority based scheduling

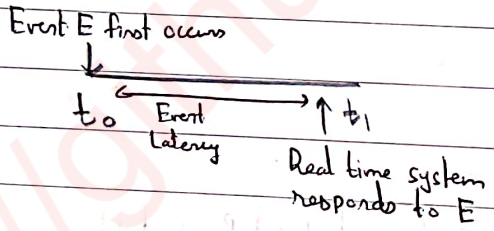
1) Soft real-time systems

→ No guarantee as to when tasks will be scheduled.
→ Guarantees only that the process will be given preference over non-critical processes.

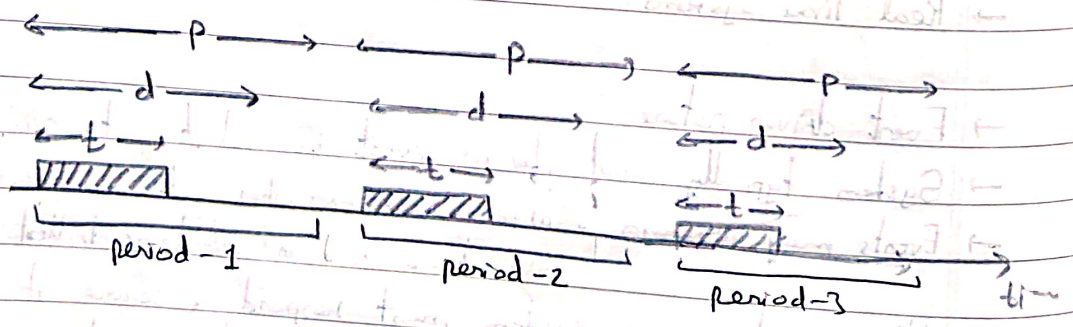
2) Hard real-time system

→ Here task must be serviced by its deadline
→ Service after deadline has expired is same as no service performed.

(should be min) Event Latency: amount of time elapse from when an event occurs to when it is serviced.
→ Different events have different latency requirements.



→ Real-time process are considered periodic.
→ They require CPU at constant intervals.
→ Suppose the process has processing time t, deadline d & period p ($0 \le t \le d \le p$)
Then Rate of periodic task is given by $\frac{1}{p}$



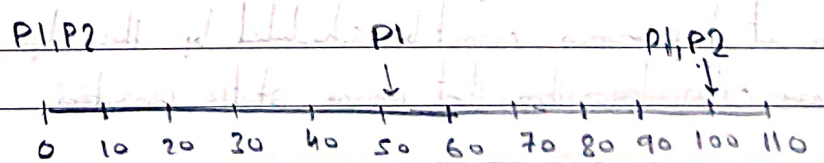
→ Process once entered into the system may have to announce deadline requirements to scheduler.
→ So, based on the deadline requirement, & using the admission-control algorithm, scheduler does one of the two things:
1) Admits process - guaranteeing that process will complete on time.
2) Rejects request - if cannot guarantee that task will be serviced by its deadline.

Real-time Process Scheduling:

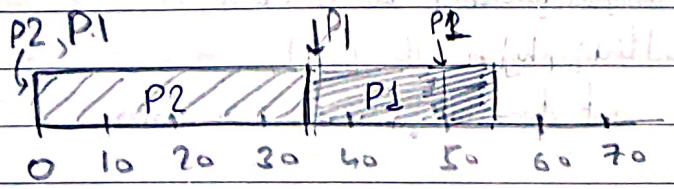
→ Schedules periodic tasks using static priority policy with preemption.
→ Priority assigned based on inverse of its period. (Rate = 1/p)
Shorter periods → higher priority
longer periods → lower priority
} assign higher priority to tasks that require CPU more often (I/O bound)
→ Assumption: processing time of a periodic process is same for each CPU burst.

Missed deadlines:

Two Processes: P1 and P2 → Periods: p1 = 50, p2 = 100
processing times: t1 = 20, t2 = 35
Deadline for each process is such that it completes its CPU burst by start of its next period.



Suppose P2 is assigned higher priority than P1.



→ So, P1 missed its deadline because P1 again arrived the system at 50 while P1 was running.

⇒ Rate Monotonic Scheduling: (Real-Time Process)

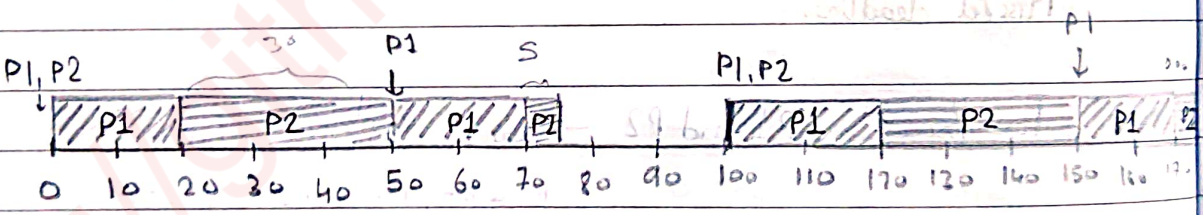
→ So, is it possible to schedule tasks so that each meets its deadlines?
→ Measure CPU utilization of a process P_i as ratio of its burst to its period - t_i/P_i

Process P1 : $20/50 = 0.40$
Process P2 : $35/100 = 0.35$
∴ Total CPU utilization = 75% ($\leq 100\%$)

→ So, Scheduler can schedule tasks in such a way that both meet their deadlines & still leave CPU with available cycles.

Ex: P1 : p1 = 50, t1 = 20
P2 : p2 = 100, t2 = 35

Period of P1 is shorter than that of P2, so P1 is assigned higher priority than P2



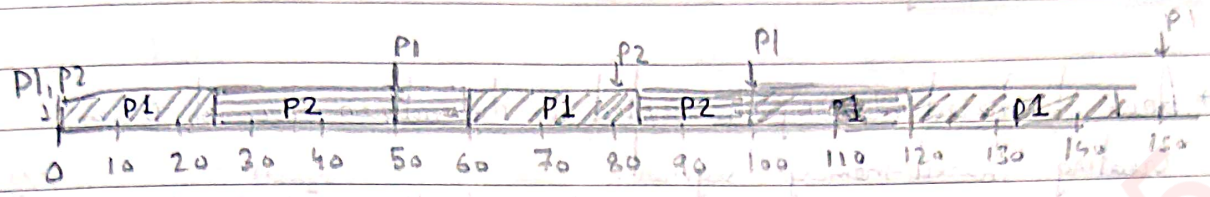
→ Rate-monotonic scheduling is considered to be optimal.
→ If a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.

⇒ Earliest Deadline First (EDF) Scheduling : (Real Time Process)

→ Priorities are assigned according to dead lines.
→ Earlier the deadline, higher the priority.
→ Preemption is not done if deadline is later.

P1 : $p_1 = 50$, $t_1 = 25$ → deadline = 50, 100, 150, ...

P2 : $p_2 = 80$, $t_2 = 35$ → deadline = 80, 160, 240, ...



→ EDF does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst.

→ The only requirement is that the process whenever it enters the system, it needs to announce its deadline to the scheduler & based on the deadline, the scheduler now decides whether the current process which is running should be preempted or not.

⇒ Interprocess Communication (IPC):

→ We know that during execution, each process is allocated some memory space and each process can access only its memory address.

→ There is no way in which a process can view or access the address space or data of any other process but suppose if two processes want to communicate then how will they do so? → using Interprocess Communication

Advantages of IPC:

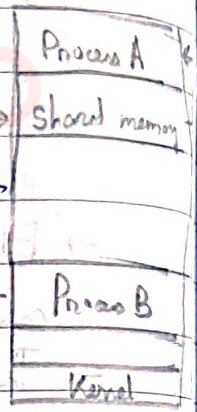
✓ → Information Sharing - like two processes wants to share files - one process write & other reads, several processes sharing the same database.

✓ → Modularity - an application can be developed as separate modules, this becomes convenient for the developers to build that application. These modules might want to communicate with each other.

✓ → Computation Speedup - when a particular task is broken down into several subtasks and then subtasks can run concurrently. These different broken tasks might need to share data or wants to communicate with each other.

Three Models of IPC are: 1) Shared memory 2) Message passing 3) Signals 4) Pipes

1) Shared Memory:



→ Area of shared memory is created in the address space of process creating shared memory segment. (segment)

→ Other process which wants to communicate must attach it to their address space.

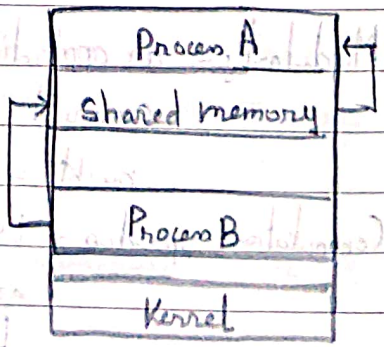
→ Now, both the process can then use this shared memory like a regular memory & can exchange information by reading, writing data in shared segment.

→ Communication over here is under the control of the user processes because these processes, the one that is creating this shared memory segment & the other process which is attaching this shared memory segment, they are both user processes.

→ Advantage - Fast

→ Limitation - To provide mechanism for user processes to synchronize when they access the shared memory i.e. when one process is accessing the shared memory, the other one should not be allowed to access & vice versa.

Code in linux:



int shmget (key, size, flags)

→ creates shared memory segment

→ returns id of the segment - shmid

→ key is unique identifier of shared memory segment

→ size: size of shared memory

→ flags: shared memory options

int shmat (shmid, addr, flags)

→ Attach shared memory (with id shmid) to address space of calling process

→ addr : pointer to memory address space of calling process.

→ flags : option

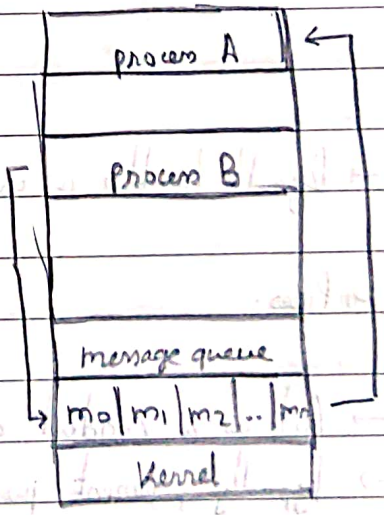
int shmdt (shmid)

→ Detach shared memory

2) Message Passing :

→ Here shared memory is not created in the address space of user but it is created in the address space of the Kernel.

→ Suppose if process B wants to send some message to process A, so process B will write into this shared memory space & it can use the system call send (A, message) to send the message to process A and now the Kernel will send some kind of signal to process A, so process A will use the system call receive (B, message) to receive the message from process B.



→ Here communication is control by Kernel.

✓ → Advantage - explicit sharing, less error prone.

✓ → Limitation - slow (as it involves system calls)

3) Signals :

→ These signals can notify a process that some event has occurred & this signal will be then delivered to the process and once a signal has been delivered to a process, the signal must be handled by the process.

Synchronous Signal: Here the signal is delivered to the same running process which caused the event (illegal memory access, divide by 0)

Asynchronous Signal: Here the signal is generated by an external event (terminating a process) & send to the running process.
Ex: a parent process wants to terminate its child process, so a parent (external) generate the signal & is received by the child process.

Unix Commands:

→ For delivering a signal: Kill (pid_t pid, int signum)
↳ Kind of signal to be delivered

→ Process handler for a signal: sighandler_t signal (signum, handler)
↳ Kind of the signal to be handled
↳ point to the memory address where that function of handling the signal is specified

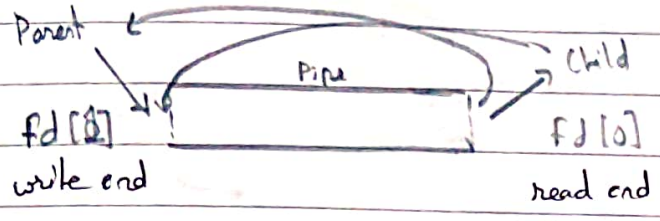
→ Default handler is used if no handler is provided.

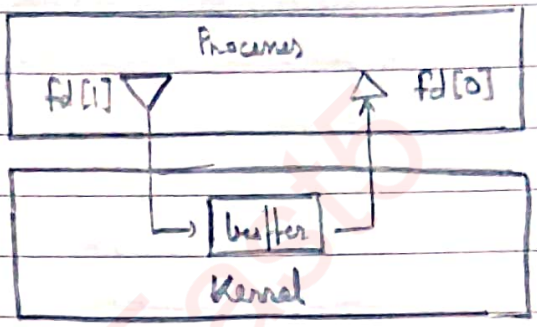
4) Pipes:

→ Acts as a conduit allowing two processes to communicate.
→ Typically, parent process creates pipe to communicate with a child process.
→ Since a pipe is a special type of file, child inherits pipe from its parent process.
→ Ordinary pipes are constructed using the function pipe (int fd[])

→ Pipe is accessed through int fd[] file descriptors: fd[0] is read end
fd[1] is write end

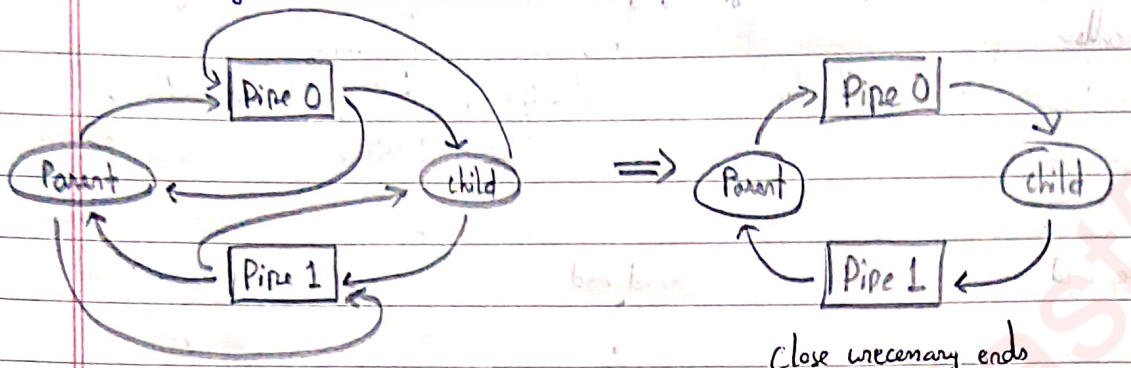
→ Unix treats pipe as a special type of file. Pipes can be accessed using read() & write() system calls.






```
#define Buffer_Size 25
int main() {
    char read_msg[Buffer_Size];
    char write_msg[Buffer_Size] = "Greetings";
    int fd[2];
    pid_t pid;
create pipe    if(pipe(fd) == -1){
        fprintf(stderr, "Pipe failed");
        return 1;
    }
fork child process    pid = fork();
    if(pid < 0){
error        fprintf(stderr, "Fork failed");
        return 1;
    }
parent proc    if(pid > 0) {
close unused (read) end    close(fd[0]);
write to pipe    fprintf(fd[1], write_msg, strlen(write_msg)+1));
close write end    close(fd[1]);
    }
child proc    else {
close unused (write) end    close(fd[1]);
read from pipe    fscanf(fd[0], read_msg, Buffer_Size);
close read end    printf("read : %s", read_msg);
    close(fd[0]);
    }
```

Two-way Communication:



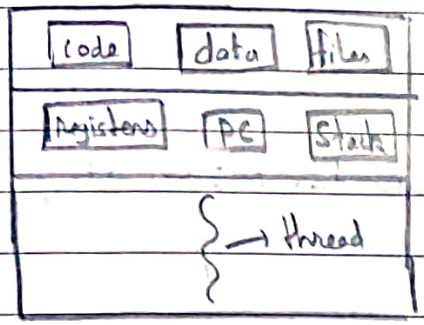
Close unnecessary ends

⇒ Threads:

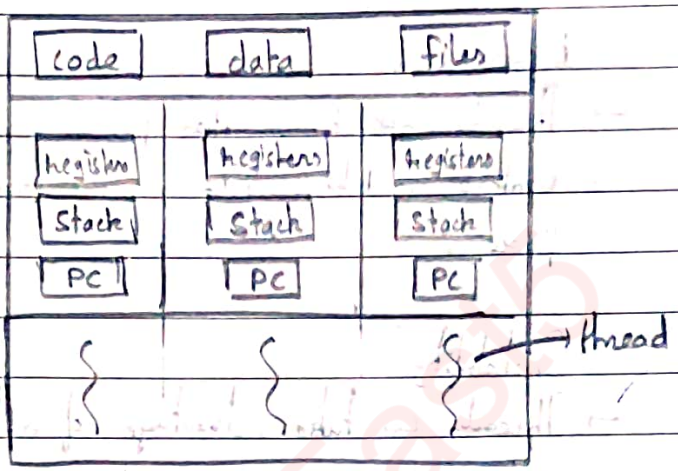
→ If we break a task into smaller tasks then that job can be finished quickly.
→ So, Parallelization can improve the performance of the applications. ie. if an application is implemented in such a way that it has multiple small computing entities, then each task can perform a small part of that job & these computing entities can run in parallel if that is supported by the hardware.
→ Ex: GPUs can perform thousands of tasks in parallel.
→ So, an application can be implemented with several threads. and these threads are the small computing entities which will perform a small part of the job.
↳ Threads are light weight processes, basic unit of CPU utilization & help in achieving parallelization.

→ A traditional process has a single thread of control.
→ If a process has multiple threads of control, it can perform more than one task at a time.
→ Ex: web browser - one thread for display, another thread for retrieving data.
Word processor - a thread for display, another thread to respond to user input, another thread to perform spell-check.

Procesn v/s Threads



Single threaded process

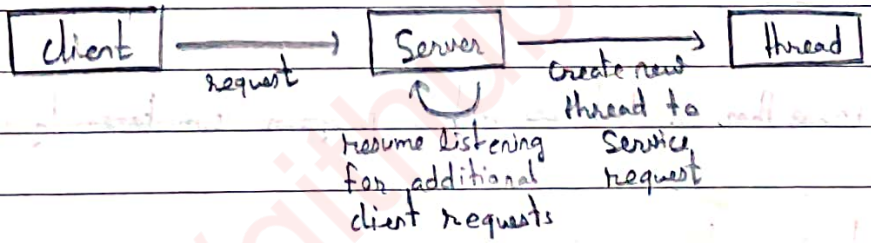


Multi threaded Process

→ Thread comprises of : 1) Thread ID 2) Program counter (PC) 3) Register set 4) stack

→ Shares among other threads in a process : 1) code section 2) data section 3) open files, signals etc

Multithreaded Server Architecture:



→ Earlier when threads were not being used, process creation was the method which was commonly used but each time we create a new process - it is time consuming & resource intensive because no resources are shared when multiple processes are created whereas threads can share resource. ~~and if a new process~~

Benefits of threads:

Responsiveness:

→ Even if a part of the process is blocked by a thread due to system call, other threads can continue doing the other part of the process.

→ Important for user interfaces.

Resource Sharing:

→ Easier communication since threads shared data compared to IPC (shared memory / message passing

Economical:

→ Threads are cheaper than process creation.

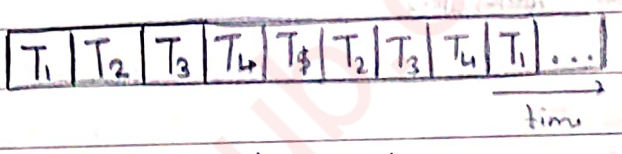
→ Context switching between the threads is faster than context switching between the processes.

Scalability:

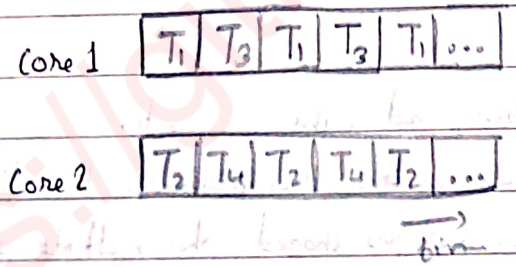
→ Threads can take advantage of multicore architectures i.e. if one thread is running on one core, another thread can be scheduled on another core.

Multicore Programming:

→ Concurrency: more than one task is supported & each tasks are making some progress

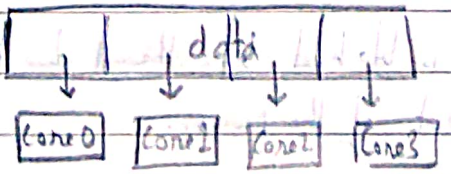


→ Parallelism: more than one task are making progress simultaneously.



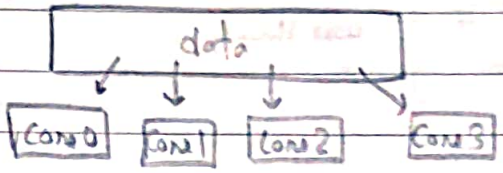
1) Data Parallelism:

→ Subsets of the same data are given to multiple cores & same operation is performed on each core to reduce time.



2) Task Parallelism:

→ Whole data is sent to different cores & different operation is performed on each core.



→ Multithreading ~~Models~~:

Support for threads: 1) User level 2) Kernel level

1) User threads:

→ Created by user.
→ managed without Kernel support
→ 3 primary user-level thread libraries: POSIX Pthreads, Windows threads, Java threads

2) Kernel threads:

→ Created & managed by Kernel
→ Ex: virtually all general purpose OS, including windows, Linux, Mac OS, ios, android.

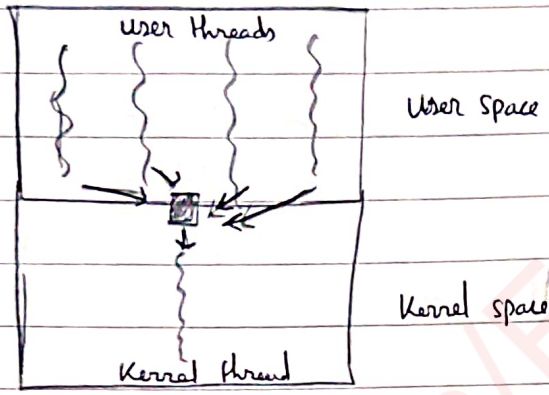
→ Multithreading Models:

→ This models define the relationship ~~better~~ between user threads & Kernel threads.

1) Many-to-One Model:

→ Here many user-level threads are mapped to a single Kernel thread.
→ Thread management is done by user thread library.
→ If one thread makes a blocking call, it ~~will~~ cause all other threads to block & the entire process will get blocked.

→ Multiple threads cannot run in parallel on multicore system because only one thread can access Kernel at a time.



2) One-to-One Model

→ Here, Each user-level threads maps to a Kernel thread, so creating a user-level thread, creates a Kernel thread.

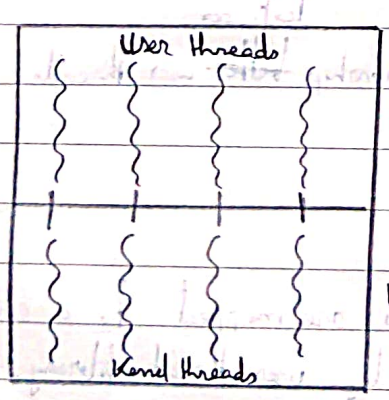
→ More concurrency than Many-to-One model.

→ Allow other threads to run even if one thread makes a blocking call, since all threads are associated with other Kernel threads.

→ It gives the advantage of multi-cores because there are multiple Kernel threads which can run simultaneously in different cores.

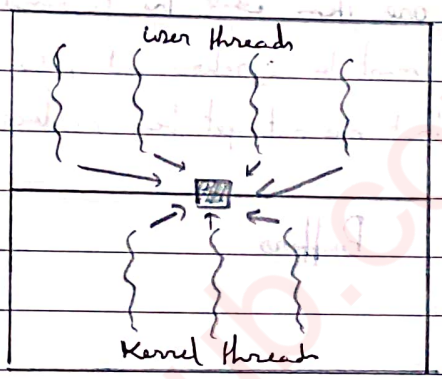
* → Because the overhead of creating Kernel threads can burden the performance of an application, most implementations of this model restricts the number of threads supported by the system.

→ Ex: Windows, Linux



3) Many-to-Many Model:

→ Many user level threads are mapped to small / equal number of Kernel threads.
→ No. of Kernel threads is specific to application or machine (cores).
→ Developer can create as many user threads as necessary & the corresponding Kernel threads can run in parallel on a multicore machine.
→ Also, when one thread performs a blocking system call, the kernel can schedule another thread for execution.



⇒ Process Synchronization:

A cooperating process is one that can affect or be affected by other processes executing in the system. (Other process is known as Independent process)

Cooperating process can either

directly share a logical address space (i.e. both code & data)

be allowed to share data only through files or messages

Concurrent access to shared data memory may result in data inconsistency.

→ Producer Consumer Problem:

→ A producer process produces information that is consumed by a consumer process.
→ One solution to the producer-consumer problem uses shared memory.
→ To allow producer & consumer processes to run concurrently, we must have a available buffer that can be filled by the producer & emptied by the consumer.
→ The buffer will reside in a region of memory that is shared by the producer & consumer processes.
→ A producer can produce one item while the consumer is consuming another item.
→ The producer & consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Buffers

Unbounded buffer ↓
The consumer may have to wait for new items, but the producer can always produce new items.

Bounded buffer (Fixed-size buffer) ↓
The consumer must wait if the buffer is empty, & the producer must wait if the buffer is full.

Suppose there is a variable counter which holds the state of size of buffer

var counter = 0

Counter is incremented everytime we add a new item to the buffer → counter ++
Counter is decremented everytime we remove a item from the buffer → counter --

Ex: Suppose counter = 5

counter++ } critical section of code
counter-- }

The producer & consumer processes execute the statements "counter ++" & "counter --" concurrently.

Following the execution of these two statements, the value of the variable counter may be 4, 5 or 6.

The only correct result, though is **counter = 5**, which is generated correctly if the producer & consume execute separately.

In Assembly language:

counter++ is implemented as:
$R_1 \leftarrow$ counter
$R_1 \leftarrow R_1 + 1$
counter $\leftarrow R_1$

counter-- is implemented as:
$R_2 \leftarrow$ counter
$R_2 \leftarrow R_2 - 1$
counter $\leftarrow R_2$

T_0:	producer	$R_1 \leftarrow$ counter	$R_1 = 5$	
T_1:	producer	$R_1 \leftarrow R_1 + 1$	$R_1 = 6$	
T_2:	consumer	$R_2 \leftarrow$ counter	$R_2 = 5$	
T_3:	consumer	$R_2 \leftarrow R_2 - 1$	$R_2 = 4$	
T_4:	producer	counter $\leftarrow R_1$	counter = 6	x
T_5:	consumer	counter $\leftarrow R_2$	counter = 4	x

Context Switch due to blocking call or time slice

Counter value depends on which process arrives last

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, where several processes access & 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.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization i.e. orderly execution of cooperating processes & ensure only one process manipulates critical section at a time

→ Critical Section Problem:

→ Each processes of a system has a segment of code, called a critical section in which the process may be changing common variables, updating a table, writing a file & so on. (accessing shared memory)

→ When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
→ It means when one process is accessing & manipulating the shared data, no other process will be allowed to access & manipulate the shared data.
→ So, multiple process will not concurrently manipulate the shared data.

→ Each process must request permission to enter its critical section.
→ The section of code implementing this request is the entry section.
→ The process after manipulating the critical section must notify other process that it has completed its usage of critical section, so other processes which are waiting can access & manipulate the critical section, this ~~request~~ section of code implementing this request is the exit section.
→ The remaining code is the remainder section.

```
do {
    [ entry section ]
        critical section
    [ exit section ]
        remainder section
} while (True);
```

A solution to the critical-section problem must satisfy the following 3 requirements:

1) Mutual Exclusion:
→ If process Pi is executing in its critical section, then no other processes can be executing in their critical section.

2) Progress:
→ If no process is executing in its critical section & some processes wish to enter their critical sections, then only those processes that are not

executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

✓ 3) Bounded Waiting (No Starvation):

→ A bound must exist on number of times a process is allowed to enter its critical section while another process is waiting.

→ Peterson's Solution: → int turn;
→ Bool flag[2];

→ A software based solution to the critical section problem.

→ The solution is restricted to two processes that alternate ~~their~~ execution between their critical sections & remainder sections.

Suppose the processes are P_i & P_j

→ It requires two data items to be shared between the two processes:

1) int turn → indicates whose turn it is to enter its critical section (i|j)
2) bodean flag[2] → used to indicate if a process is ready to enter its critical section

Process P_i		Process P_j
do{	($P_i \rightarrow P_j$)	do{
flag[i]=true;		flag[j]=true;
turn=j; ← (although P_i wants to enter its critical section, being humble it gives first chance to process P_j)	turn = i	turn=i;
while(flag[j] && turn==j);		while(flag[i] && turn==i); wait
//critical section		//critical section
flag[i]=false;		flag[j]=false;
//remainder section		//remainder section
} while(True);		} while(true);

	flag[i]	flag[j]
	T/F	T/F

(Program)

Does this solution satisfy the 3 requirements?

1) Mutual Exclusion ✓ : when Pi was in critical section Pj was stuck in the while loop & could not enter its critical section. & vice versa

2) Progress ✓ : when both the processes were at the beginning stage, they were not in their remainder section & both wished to enter their critical section & both collectively decided which process will enter the critical section & this decision was not prolonged indefinitely.

3) Bounded Wait ✓ : Once process Pi is out of the critical section, it turns its flag to false i.e. it will not enter the critical section again & it allows the other process Pj to enter the critical section. So, there is a bound (no. of times) a process enters its critical section.

✓ Limitation: Not guaranteed to work on modern architectures because processors and/or compilers may reorder operations that have no dependencies, to improve performance.

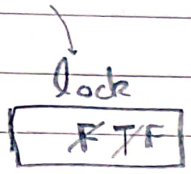
→ Test and Set lock:

→ A (hardware) solution to the synchronization problem.
→ There is a shared lock variable which can take either of the two values, 0 or 1.
→ Before entering into the critical section, a process inquires about the lock.
→ If it is locked, it keeps on waiting till it becomes free.
→ If it is not locked, it takes the lock & executes the critical section.

TestAndSet() Instruction :

```
boolean TestAndSet (boolean *Lock){
    boolean prev = *lock;
    *lock = true;
    return prev;
}
```

Atomic Operation: Everything will happen as a single operation which cannot be interrupted by anyone.



Process P_1

```
do{
    while (TestAndSet(&lock));   // F
    // Critical section
    Lock = false;
    // remainder section
} while (true);
```

Initially set to 0

Process P_2

```
do{
    while (TestAndSet(&lock));   // T  wait
    // Critical section
    Lock = false;
    // remainder section
} while (true);
```

Satisfy requirements?

1) Mutual Exclusion ✓:
2) Progress ✓
3) Bounded wait ✗ : When process 2 wishes to enter the critical section, there might come other process P3, P4, P5, ... which might take the lock variable & enter the critical section. so, there is a chance of starvation.

→ It can be used to build other constructs: spin lock, mutex, semaphores

→ Mutex lock (Binary Semaphores):

→ A (software) solution to the critical section problem.
→ Mutex means Mutual Exclusion.

Process 1 | lock | Process 2

lock: 0/1

```
do{                                   do{
    acquire(&lock);                       acquire(&lock);
    // critical section                   // critical section
    release(&lock);                       release(&lock)
    // remainder section                  // remainder section
} while(true);                        while(true);
```

How is acquire() & release() implemented ?

1)

```
Process 1                             Process 2
do{                                   do{
    while(lock≠0);                        while(lock≠0);
    lock=1                                lock=1;
    //Critical section                    //Critical section
    lock=0                                lock=0;
    // remainder section                  // remainder section
} while(true);                        } while(true);
```

Is Mutual Exclusion guaranteed ?

lock=0

Context switch →

P1: while(lock≠0);

P2: while(lock≠0);

lock=1

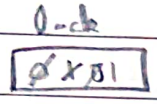
P1: lock=1

Both processes enter critical section!!

Since the "checking of the lock & modifying it" was not atomic in natur, now both the processes can enter the critical section. That means both the operations of acquire() must be atomic, only then mutual exclusion can be maintained.

2)
```
acquire(int *Lock){
    while(*Lock ≠ 0);
    *Lock = 1;
}
```
← atomic

```
release(int *Lock){
    *Lock = 0;
}
```
✓



Process 1 → Lock ← Process 2

```
do{
    acquire(& lock);
    // critical section
    release(& lock);
    // remainder section
} while(true);
```

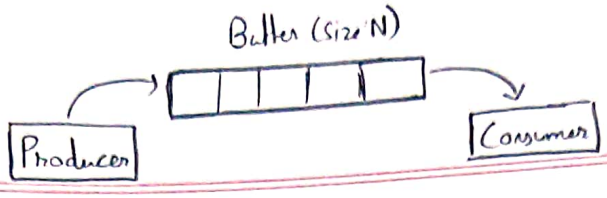
```
do{
    acquire(& lock);   wait
    // critical section
    release(& lock);
    // remainder section
} while(true);
```

Limitations:
→ Requires busy waiting (Spin Lock).
→ wastes CPU cycles

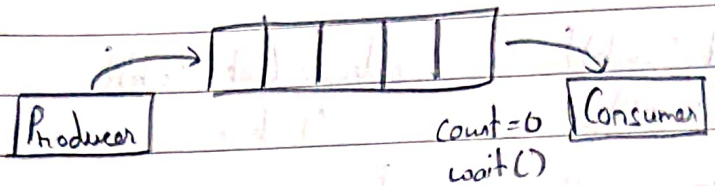
Advantages (if lock held for shorter duration):
→ Context switch not required instead it keeps spinning.
→ If we have a multicore system, one thread can spin on one processing core while other thread uses critical section on another core.

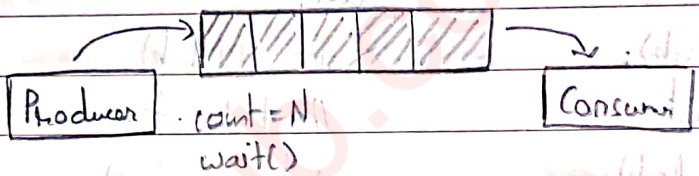
→ Recall Producer Consumer Problem:

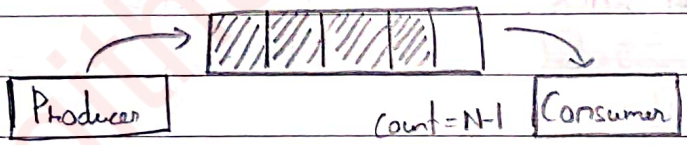
→ It is also referred to as Bounded Buffer Problem.



→ What would happen when: • producer keeps producing even if buffer is full
• Consumer keeps consuming even if buffer is empty
. . To address this issues we require Synchronization







after semaphore

→ Semaphores:

→ Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.

→ A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() & signal().
↓ P() ↓ V()

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

```
Signal(S){
  S++;
}
```

P0	P1
✓ wait(S)	wait(S) ✓
× wait(S)	wait(S) ×
⋮	⋮

ever if buffer is full
ever if buffer is empty
we require Synchronization

onsumer

after semaphore

to manage concurrent
as a semaphore.
initialization, is accessed
& signal().
↓
V()

P0	P1
✓ wait(S)	wait(S) ✓
× wait(S)	wait(S) ×
⋮	⋮

Types of Semaphores:

1) Counting Semaphore:

→ Its integer value can range over an unrestricted domain.
→ It is used to control access to a resource having finite no. of instances (Ex: S=5)
→ Process waiting to use resource must perform the wait() operation on semaphore.
→ In order to release resource, process must perform the signal() operation on semaphore.
→ When count of semaphore is 0, all resources are being used. So, Process that wish to use resource will block until count becomes greater than 0

2) Binary Semaphore:

→ The value of a binary semaphore can range only between 0 & 1.
→ Same as Mutex lock.

Disadvantages:
→ The main disadvantage is busy waiting.
→ While a process is in its critical section, any other process that wishes to enter its critical section must loop continuously in the entry section.
→ Busy waiting wastes CPU cycles that some other process might be able to use productively
→ This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

→ To overcome the need for busy waiting, we can modify the definition of the wait() & signal() semaphore operations.
→ Rather than engaging in busy waiting, the process can block itself, then control is transferred to the CPU scheduler, which selects another process to execute.

→ But then we have problems of Deadlocks & starvation associated with semaphores.

→ Solution to Bounded Buffer Problem using Semaphores:

→

We will make use of 3 semaphores: 1) mutex 2) empty (counting semaphore) 3) full

- mutex: 0 / 1 → used to acquire & release lock → To avoid race condition
- empty: ↓ whose initial value is the no. of slots in the buffer
- full: ↓ whose initial value is 0 as all slots are empty initially

Producer	Consumer
do{	do{
wait until 'empty' ← wait (empty); // the decrement	wait (full); → wait until full > 0 then decrement 'full'
acquire 'lock' ← wait (mutex);	wait (mutex); → acquire lock
// add data to buffer	// remove data from buffer
release 'lock' ← signal (mutex);	signal (mutex); → release lock
increment 'full' ← signal (full);	signal (empty); → increment 'empty'
} while (true);	} while (true);

→ The Readers-Writers Problem:

→ A database is to be shared among several concurrent processes.
→ Some of these processes may want only to read the database (Readers), whereas others may want to update (read & write) the database (Writers).
→ If two readers access the shared data simultaneously, no adverse affects will result because they are not modifying the database.
→ However, if a writer and some other thread (reader or writer) access the database simultaneously, chaos may occur.
→ To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.
→ This synchronization problem is referred to as the readers-writers problem.

Solution using Semaphores:

→ We will make use of two semaphores & an integer variable

✓ 1) mutex (inialized to 1) → used to ensure mutual exclusion when read count is updated i.e. when any reader enters / exits from the critical section.

✓ 2) wrt (initialized to 1) → common to both readers / writer process

✓ 3) readcount (initialized to 0) → keeps track of how many processes are currently reading the data.

Writer Process

```
do{
    wait(wrt);      → writer request for critical section
    // perform the write operation
    signal(wrt);    → leaves the critical section
} while(true);
```

Reader Process

First readers responsibility is to acquire wrt lock.
Last readers responsibility is to release wrt lock

```
do{
    wait(mutex);
    readcount++;        → before we need to get the lock otherwise race condition can occur
    if(readcount==1)    ← First Reader Process
        wait(wrt)       → this ensures no writer can enter if there is even one reader
    signal(mutex);      → other readers can enter while this current reader is inside the critical section
    // Current reader performs reading
    wait(mutex);
    readcount--;        → a reader wants to leave
    if(readcount==0)    ← Last Reader Process   → no reader is left in the critical section
        signal(wrt);    → writers can enter
    signal(mutex);      → reader leaves
} while(true);
```

99% chance

→ Monitors:

Problems with Semaphores:

→ Incorrect use of semaphore operations: signal (mutex) .... wait(mutex)
wait (mutex) .... wait (mutex)

→ Using them incorrectly may result in deadlocks, timing errors etc that are difficult to detect.

no problem like above

→ Monitor is a high level abstraction that provides a convenient & effective mechanism for process synchronization.

→ A monitor type presents a set of programmer-defined operations that provide mutual exclusion within the monitor.

→ The monitor type also contains the declaration of variables whose values define the shared variables, along with the bodies of the procedures or functions that operate on those variables.

Syntax:

```
monitor monitor_name
{
  // shared variable declarations
      .
  procedure P1(..){
     ...
  }
  procedure P2(..){
     ...
  }
  :
  Procedure Pn(..){
     ...
  }
  Initialization Code(...){
     ...
  }
}
```

functions that operate on the shared variables

→ A procedure defined within a monitor can access only those variables which are declared locally within the monitor & its formal ~~monitors~~ parameters.

→ Similarly, the local variables of a monitor can be accessed by only the local procedures.

→ The monitor construct ensures that only one process at a time can be active within the monitor.. ← ensures Mutual Exclusion

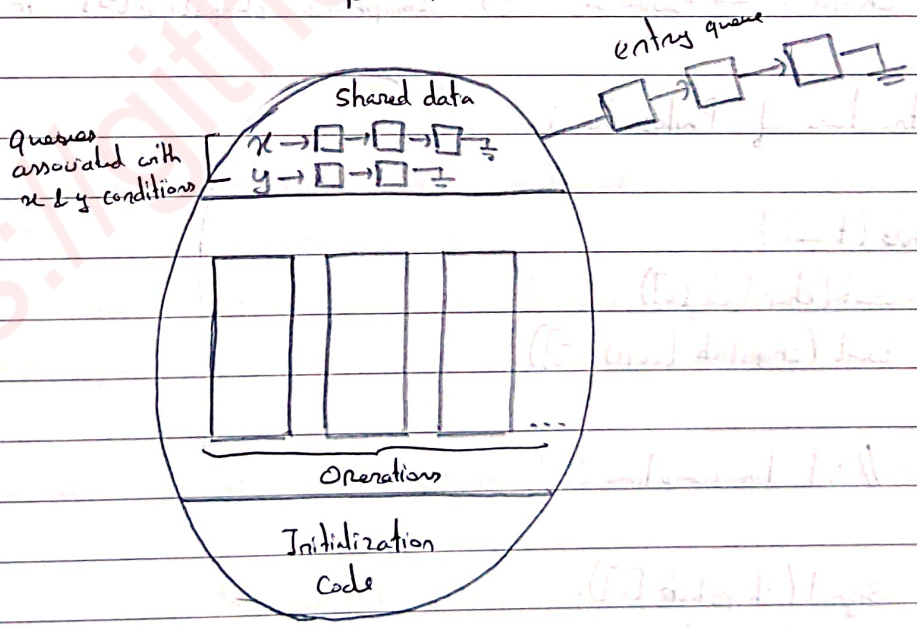
→ Additional Synchronization mechanism is provided by Condition Construct.

condition x, y;

→ The only operations that can be invoked on a condition variables are wait() & signal().

→ The operation "x.wait();" means that the process invoking this operation is suspended until another process invokes "x.signal()".

↓
resumes exactly one suspended process



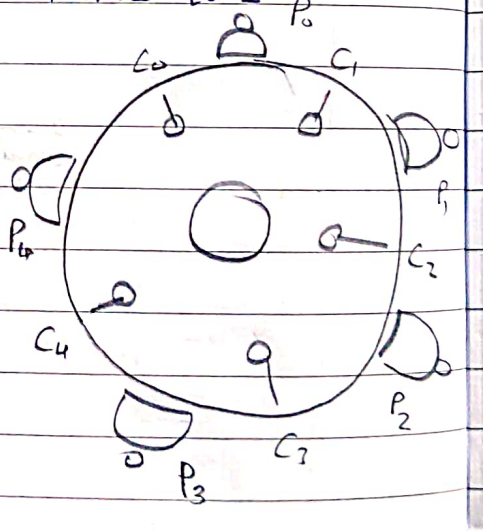
→ The ~~Dining~~-Philosophers Problem.

99% chance

→ Dining Philosophers Problem:

→ The dining philosophers problem is a classic synchronization problem.
→ Here, we have a set of 5 philosophers who are seated around a circular table.
→ There philosophers spent their lives thinking or eating & they do not interact with their neighbour.
→ There is a shared bowl of rice & there are N chopstics for N philosophers
→ A philosophers need to have both his adjacent chopstics to eat.
→ Occasionaly wherever a philosopher wants to eat, he or she will try to pickup one chopstick at a time & then eat from the bowl.
→ Once a philosopher finishes eating, he must release both of his chopstics

Using Semaphores:
∴ Bowl of Rice → shared dataset (Critical section)
∴ Philosopher → Processes
∴ Chopstics → Resources ⇒ Semaphore chopstick[5] initialized to 1



Structure of Philosopher i

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

    // eat for some time

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

    // think for some time
}
```

Problem with Semaphores:

→ Suppose the philosopher P_0 wants to eat, so grab his first fork, then suppose that process is preempted by philosopher P_1 & it also grabs his first fork & so on.
→ so, every philosopher grabs their first fork, so it introduces a deadlock situation where each & every philosopher is waiting for their second fork

Solution:

1) → Allow a philosopher to pickup chopstick only if both chopstics are available, to do this, philosophers must pick their chopstics in a critical section.

2) Asymmetric Solution
→ Odd-numbered philosopher picks up first left chopstick & then right chopstick.
→ Even-numbered philosopher picks up right chopstick first & then left chopstick.

Using Monitors:

Condition (Restriction): a philosopher may pick up chopstics only if both of them are available

So, Need to distinguish among 3 states (Thinking, Hungry, Eating) in which we may find a philosopher.

→ So, we introduce a data structure: enum {Thinking, Hungry, Eating} state[5];

→ Philosopher i can set variable state[i] = Eating, only if his two adjacent neighbours are not eating i.e. (state[(i+1)%5] != Eating and state[(i+4)%5] != Eating)

→ Declared condition self[5];
It allow philosopher i to delay when he is hungry but is unable to obtain chopsticks needed.

→ Before starting to eat:
- Each philosopher must invoke the pickup() operation.
- If the philosopher who is invoking this pickup() operation, is unable to get the chopsticks, then it will result in the suspension of the philosopher process.
- If successful, the philosopher may eat.

→ After eating, philosopher must invoke the putdown() operation.

Implementation:

```
monitor DiningPhilosophers
{
  enum {Thinking, Hungry, Eating} state[5];
  condition self[5];

  initialization_code(){
    for(int i=0 ; i<5 ; i++)
        state[i] = Thinking;
  }

  void pickup(int i){
      state[i] = Hungry;
      test(i);
      if(state[i] != Eating)
        self[i].wait;
  }

  void putdown(int i){
      state[i] = Thinking;
      test((i+4)%5);
      test((i+1)%5);
  }

  void test(int i){
    if((state[(i+4)%5] != Eating) &&
       (state[i] == Hungry) &&
       (state[(i+1)%5] != Eating)){
        state[i] = Eating;
        self[i].signal();
    }
  }
}
```

giving permission to left & right to eat

test left & right neighbours

⇒ Deadlocks:

→ We know that a computing system consists of resources & these resources may be partitioned into several types (or classes) like $R_1, R_2, \ldots R_m$.

→ These resources can be CPU cycles, memory space, I/O devices, semaphores, locks etc

→ Each resource of type R_i can have W_i identical instances.

→ So, if any thread or process requests an instance of a resource type, then allocation of any instance should satisfy the requests.

→ Each process wherever wants to utilize a resource, it has to follow 3 steps:
1) request
2) use
3) release

→ A thread can request as many resources as it requires to carry out its designated task. Number of resources requested may not exceed total no. of resources available in the system.

→ If a process (or thread) request a resource that is managed by the Kernel then the OS will check whether the thread that has requested the resource, is has been allocated the resource or not. For that, OS will maintain a system table, where it will record for each resource, whether that resource is free or is allocated & if it is allocated, to which thread/process that particular resource has been allocated.

→ If a thread requests resource currently allocated to another thread, then the requesting thread is added to the queue of threads waiting for that resource

→ A set of threads is in deadlocked state when:
→ Every thread in the set is waiting for an event & that event can be caused by only another thread in the set.

I
Resource Acquition & Release

Example: Semaphores S_1 & S_2 is initized to 1

P1:	P2:
wait(S_1)	wait(S_2)
wait(S_2) — Infinite waiting	wait(S_1) — Infinite waiting
⋮	⋮
Signal(S_1)	Signal(S_2)
Signal(S_2)	Signal(S_1)

Imp

Deadlocks can arise only if the follow 4 conditions hold simultaneously:

1) Mutual Exclusion: only one process / thread can use an instance of a resource

2) Hold and wait: a process holding atleast one resource is waiting to acquire additional resources held by other processes.

3) No preemption: a resource can be released only voluntarily by process holding it.

4) Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource held by P_1,
P_1 is waiting for a resource held by P_2,
⋮
P_n is waiting for a resource held by P_0.

→ Resource Allocation Graph:

→ Deadlocks can be described in terms of a directed graph called a system resource-allocation graph of set of vertices V & a set of edges E.

V is partitioned into 2 types: $P = \{P_1, P_2, \ldots, P_n\}$ set of all processes in system
$R = \{R_1, R_2, \ldots, R_n\}$ set of all resource types in system

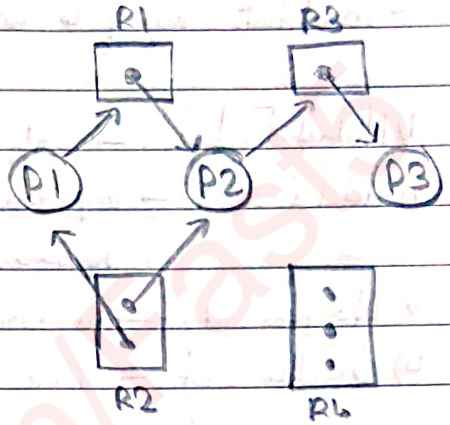
Edge E 2 types: 1) Requested edge : directed edge $P_i \rightarrow R_j$
2) Assignment edge : directed edge $R_j \rightarrow P_i$

Ex: $P = \{P1, P2, P3\}$

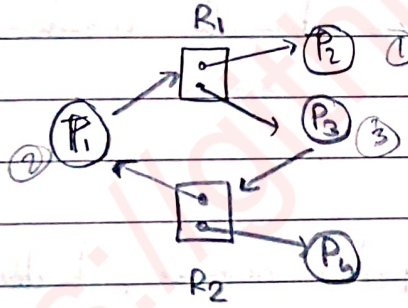
$R = \{R1, R2, R3, R4\}$

$E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$

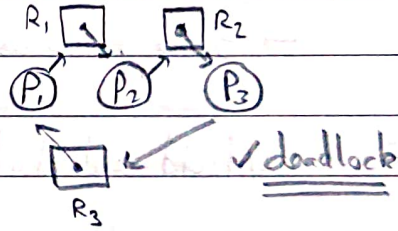
Instances (dots): $R1 \rightarrow 1$, $R2 \rightarrow 2$, $R3 \rightarrow 1$, $R4 \rightarrow 3$



If graph contains no cycles : No thread deadlocked.
If graph contains a cycle : Deadlock may exist
If each resource type has exactly one instance & cycle is present : Deadlock exists



Cycle exists but no deadlock



→ Deadlock Handling:

1) Ostrich Strategy : Ignore the problem & pretend that deadlocks never occur in system. Most modern OSs use this strategy.

2) Ensure system will never enter a deadlock state : (failing any one of 4 conditions)
a) Deadlock prevention – take care of necessary conditions leading to deadlock.
b) Deadlock avoidance – check system state before allocation of resource

3) Allow system to enter a deadlock state & then recover

→ Deadlock Prevention:

Invalidate any one of the necessary conditions.

1) Mutual Exclusion → only one process at a time can use a resource
2) Hold & wait → a process holding at least one resource is waiting to acquire additional resources held by other processes
3) No preemption → a resource can be released only voluntarily by process holding it
4) Circular wait → there exists a set $\{P_0, P_1, P_2, \ldots, P_n\}$ of waiting processes such that P_i is holding a resource & waiting for a resource held by P_{i+1}.

✗ 1) Mutual Exclusion

→ Not required for sharable resources (eg, read-only files)
→ Must hold for non-sharable resources (otherwise inconsistency, race conditions may occur)
→ Some resources like semaphores, mutex are intrinsically non-sharable.

✓ 2) Hold and Wait

→ Must guarantee whenever a process requests a resource, it does not hold any other resources.
✗ → For that, we request the process that it announces, how many resources it needs "or" before it begins execution & all that resources are allocated to it.
→ Allow process to request resources only when process has none allocated to it.
↓
Starvation possible

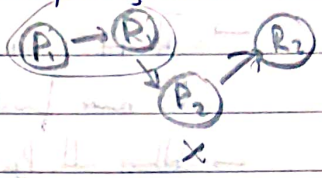
✓ 3) No Preemption

→ If process holding resources requests another resource that cannot be immediately allocated, all resources currently held by the process are released.

→ Process can restart only when it can regain all required resources.

Alternative:

→ Check whether the resources held that have been allocated to some other thread, that thread is also waiting for some additional resources.
→ So, Preempt that waiting thread & allocate desired resource to requesting thread.
→ Cannot generally be applied to resources like mutex locks & semaphores.



✓ 4) Circular Wait

→ Impose a total ordering of all resource types.
→ Require each process requests resources in an increasing order of enumeration.
→ Invalidating circular wait condition is most common.
→ Assign each resource a unique number.
→ Resources must be acquired in order.

Ex: P_0 – R1, R3, R5
P_1 – R2, R6

→ Deadlock Avoidance:

→ Avoidance algorithm requires that system has a priori information i.e. Each process must declare maximum number of resources of each type it may need.
→ Algorithm examines resource-allocation state (↓ defined by number of available & allocated resources & max. demand of process) to ensure that there can never be a circular-wait condition.
→ When a process requests an available resource, system must decide if immediate allocation leaves system in a safe state.

Safe State:

→ System is in safe state if there exists a sequence $\langle P_1, P_2, \ldots, P_j, \ldots, P_i, \ldots P_n \rangle$ of all processes in a system such that for each P_i:

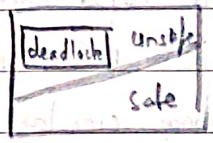
→ Resources, P_i can still request can be satisfied by currently available resources + resources held by all P_j, with $j<i$.

→ If P_i's resource needs are not immediately available, P_i can wait until all P_j have finished.

→ When all P_j has finished, P_i can obtain the needed resources, execute, & return allocated resources, and terminate.

→ When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

System → Safe state → no deadlock
↳ unsafe state → possibility of deadlock



Avoidance Algorithms → ensure that a system will never enter an unsafe state.

Ex: A system with 12 instances of a resource & there are 3 processes: P0, P1, P2
P0 requires 10 resources, P1 may need 4 & P2 may need 9 resources

At time t_0: P0 is holding 5, P1 is holding 2 & P2 is holding 2 resources
(prior info) Free resources = 3

	Maximum	Current Allocation	Needed (Max-Cur)
P0	10	5	5
P1	4	2	2
P2	9	2	7

Can be fulfilled
→ Free resource = 5+5=10
⇒ Free resource = 3+2=5

∴ At time t_0, system is in safe state & the sequence $\langle P1, P0, P2 \rangle$ satisfies safety condition.

But system can go from a safe state to unsafe state.

Ex: At time t_1, P2 requests & is allocated one more resource. (Free resource = 2)

	Maximum	Current Allocation	Needed
P0	10	5	5
P1	4	2	2
P2	9	3	6

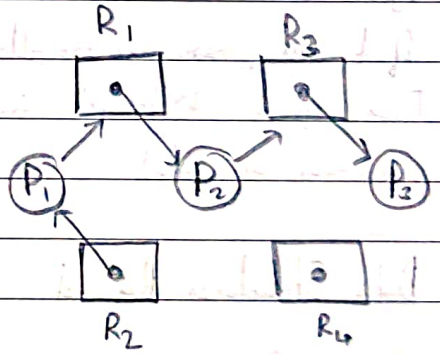
⇒ Free resource = 2+2 = 4

∴ Results in a deadlock, could have been avoided by not allocating resource to P2 by examing the previous safe sequence.

Deadlock Avoidance using Resource Allocation Graph:

→ Suitable for systems with only one instance of each resource type.

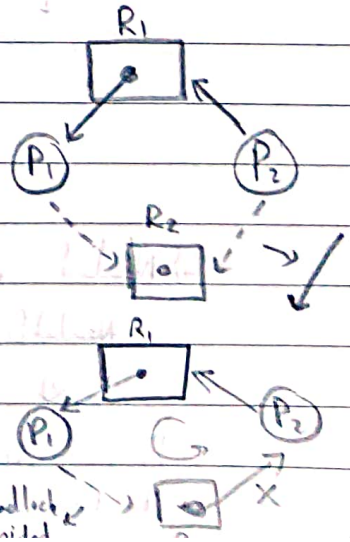
Graph with set of vertices V & set of edges E



Vertices : 1) set of process $P = \{P_1, P_2, \ldots, P_n\}$
2) set of resources $R = \{R_1, R_2, \ldots, R_m\}$

Edges : 1) Request edge : directed edge $P_i \rightarrow R_j$
2) Assignment edge : directed edge $R_j \rightarrow P_i$

→ We also know that the avoidance algorithms require that each process must declare the resources that they need prior to execution.



3) claim edge : directed edge $P_i \rightarrow R_j$ (dashed line)
(indicates process P_i may request resource R_j in future)

When a process requests a resource ⇒ claim edge → request edge

When a resource is released by a process ⇒ Assignment edge → claim edge

Now, when request edge is converted to an assignment edge & if a resource is allocated to a process, check if it leads to deadlock.

→ Banker's Algorithm:

→ We have seen that a resource allocation graph can be used for deadlock avoidance if there are single instances of resource types but if there are multiple instances of the resource types, then Banker's Algorithm can be used for deadlock avoidance.

→ Also avoidance algorithms, require that each process must a priori claim the maximum no. of resource instances, it will be using.
→ However this no. may not exceed the total no. of resources which are available in the system.

So, Wherever a process requests a set of resources:
→ System must determine whether allocation of resources will leave system in a safe state.
→ If it will, resources are allocated.
→ Else, process (or thread) must wait until some other process (or thread) releases enough resources as needed by the process.

Data Structures Required:

Ex: n=3 (no. of process), m=4 (no. of resources types)

1) Available : vector of length 'm'
Available[j]=K where K → available instances of resource type R_j

2) Maximum : matrix (n×m)
Max[i,j]=K
⇓
process P_i may request at most K instances of resource type R_j

3) Allocation: matrix (n×m)
Allocation[i,j]=K
⇓
process P_i is currently allocated K instances of R_j

4) Needed : matrix (n×m)
Needed[i,j]=K
⇓
P_i may need K more instances of R_j to complete its task.

Safety Algorithm:

Example: 5 processes ($P_0, P_1, \ldots, P_4$), 3 resource types: A (10 instances)
B (5 instances)
C (7 instances)

At time t_0,

	Allocated (Priori info)			Max			Need (MAX - Allocated)			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	2	0	0	3	2	2	1	2	2			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

Initialize: Work = [3 3 2], Finish = [0 0 0 0 0]

$\Rightarrow$ Work = [5 3 2], Finish = [0 1 0 0 0] ($\leq$)

$\Rightarrow$ Work = [7 4 3], Finish = [0 1 0 1 0] ($\leq$)

$\Rightarrow$ Work = [7 4 5], Finish = [0 1 0 1 1] ($\leq$)

$\Downarrow$

Work = [7 5 5], Finish = [0 1 0 1 1]

$\Downarrow$

Work = [10 5 7], Finish = [1 1 1 1 1]

$\therefore$ Safe sequence: $< P_1, P_3, P_4, P_0, P_2 >$ at time t_0

At time t_1, P_1 requests for (1,0,2)

1) check req $\leq$ need of P_1 ✓
2) check req $\leq$ available of P_1 ✓
3) To again find the safe sequence after allocating P_1 (1,0,2).

$\Downarrow$

$P_1 \rightarrow$ allocated $\rightarrow$ 302 ✓
Need $\rightarrow$ 020 ✓
Available $\rightarrow$ 230 ✓

$\therefore$ Work = [230]
Finish = [0 0 0 0 0]

Now, again find the safe sequence & it is $< P_1, P_3, P_4, P_0, P_2 >$, so we can grant the resources requested by P_1.

Now, at time t_2, P_4 requests for (3,3,0)

1) ✓
2) ✗ (3 3 0) > (2 3 0)

∴ Request is rejected

→ Deadlock Recovery:

1) Process Termination

a) Abort all the deadlocked processes
→ But the problem is that these processes may have computed for a long time & the results of all these partial computations will have to be discarded.

b) Abort one process at a time, until deadlock cycle is eliminated.
→ So, out of all the deadlocked processes, one process will be chosen to be aborted & after it has been aborted, then again a deadlock detection algorithm will be invoked to check whether the remaining processes are deadlocked or not.
→ If the deadlock is still present, then again another process will be chosen.
→ This incurs considerable overhead because every time a process is being aborted, a detection algorithm has to be invoked to check for deadlock.

Situation, a process might be in while terminating it:

→ If a process was in the middle of updating a file & now terminating that process may leave the file in an incorrect state.
→ If a process was in the middle of updating a shared data while holding a mutex lock & now terminating that process, the system must restore the status of the lock as now available & there is no guarantee on the integrity of the shared data.

→ Now, if we use the method of partial termination that means we are terminating only one process at a time. So, out of all the deadlocked processes, we have to determine that which deadlocked process should be terminated & the strategy would be to abort those processes first whose termination will incur the minimum cost.

Factors to consider when choosing a process to abort:

✓ 1) Priority of process
✓ 2) How long has the process computed, how much it is near to completion
✓ 3) How many resource a process is using
✓ 4) How many resources a process needs to complete.
✓ 5) Is process interactive or batch?

Priority
Real time > System > Interactive > Batch

Higher priority — Lower priority

2) Resource Preemption:

Preempt resources from a process and allocate it to another process until deadlock is not there.

Selecting a victim:

→ From which process, the resources should be preempted? Obviously that process which will incur minimum cost.
→ Above factors can be used.

Rollback the victim:

→ After preempting the resource from a victim process, the process must be returned to some previous safe state & then wherever it starts computation next, it will start from that particular state.

Problem of Starvation:

→ Now, when we do resource preemption, there is a possibility that the same process

may always be picked up as a victim, so that process will never be executed.

→ So, to avoid starvation, we include no. of roll backs as the cost factor.

✱ Memory Management Unit :-

→ We have seen earlier, wherever a program has to run, it will brought from the disk into the main memory (RAM).

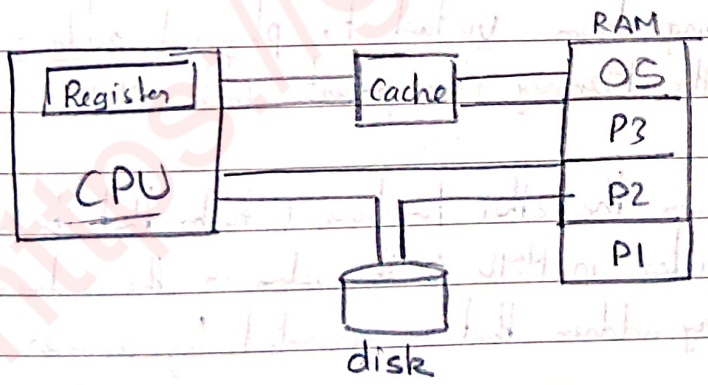
→ This memory, it sees a stream of addresses coming from the central processing unit (processor), so the processor sends read or write requests.

→ Now, the storage possibilities in the computer system also includes the registers which are present inside the processor & acess to the register can be done by the processor in less than 1 clock cycle. but the main meory acess can take many clock cycles, which may cause a stall.

→ So, instead of using the main memory which requires a longer access time, there is a cache which is a high speed memory & it sits between the main memory & CPU registers.

→ So, for easy & quick acess, some data can be trasferred to (RAM) main memory and from RAM to the cache.

→ Also, this RAM, it has multiple process loaded into it & we know that the protection of this memory is required to ensure correct operation i.e. no proces should be able to acess the address space of another proces.



⇒ Logical & Physical Address Space :

→ The concept of Logical & Physical Address space is very important for proper memory management.

Logical Addres Space – Usually seen by the programs
Addres that is generated by CPU
Also refered as Virtual Addres (of a process)

Physical Addres Space – Address seen by the memory unit

So, CPU will generate the logical addres which will be mapped into a physical addres which then can be seen by the main memory.

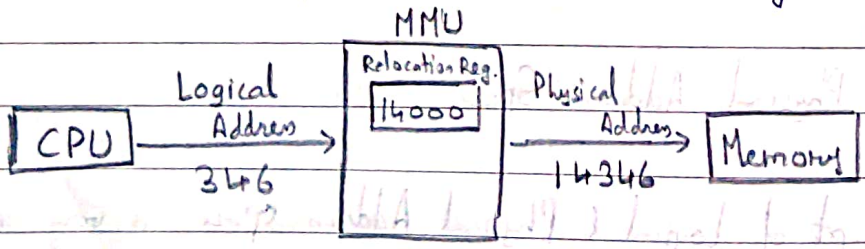
→ The binding of the instructions & data to memory addresses (i.e. mapping of logical to physical addres) can be done at any stage:

✓ 1) Compile / Load time : Generated logical addres same as Physical addres.

✓ 2) Execution time : Here Logical addres different than Physical addres (when program is being executed)

→ So, this run-time mapping from virtual to physical addres is done by a hardware device called the Memory Maragement Unit (MMU).

→ In this MMU, apart from the other functions & tasks that it performs, there is one relocation register in MMU & the value in the relocation register will be added to every addres that is generated by a user process.



→ Here, the logical addres space of the program, this addres 346 is actually mapping to a particular addres 14346 in the RAM.

⇒ Non-Contiguous Memory Allocation: 1) Segmentation
2) Paging
3) Segmented Paging

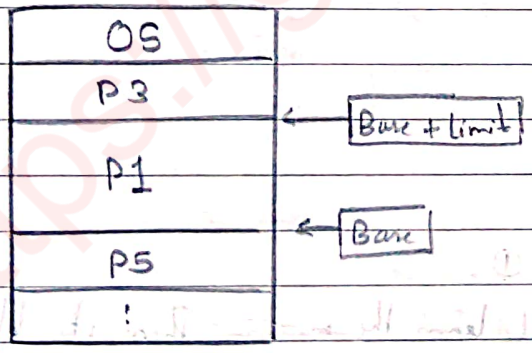
→ So, this mapping is done by the MMU depending on where the process has to be given space in the RAM.
→ Since, this value of relocation register will be different for every process depending on where the space is given in RAM, MMU ensures that no user process enters or accesses the spaces of any other process.

⇒ Contiguous Memory Allocation:

→ Memory is usually divided into 2 partitions: one for OS & other for user process
↓
can be placed in low memory address or high memory address

Contiguous Memory Allocation: each process is contained in a single section of memory that is contiguous to section containing the next process.

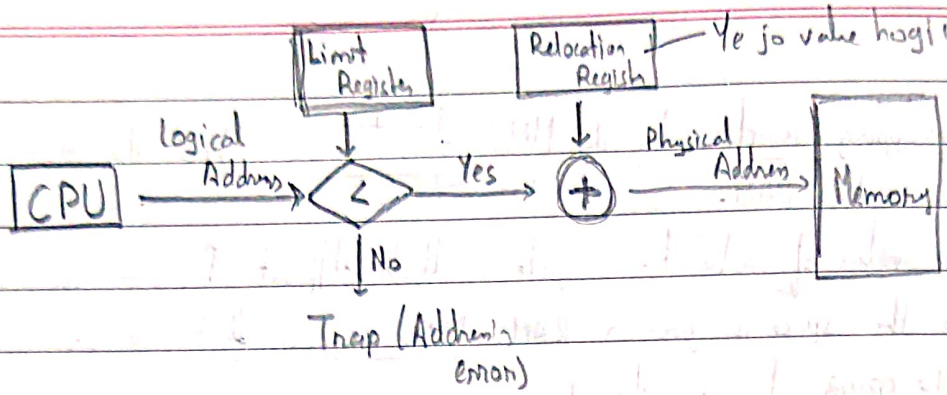
Base Register: It keeps the smallest address from which that process is starting & there would be a maximum address till which a particular process can access which would be specified by the limit.
+
Base Register (So no process can access the address space of another process)



Each process has a base & limit address specified.

→ We know that to protect the user processes from accessing the space of each other, relocation registers are used.
→ Base Registers contains the value of smallest physical address. limit register contains the range of logical addresses.
→ Limit Registers contains the range of logical addresses. (So each logical address generated by the CPU which belongs to a particular process, it must be less than the value contained in the limit register).

→ Fixed Partition Memory Allocation: → Internal fragmentation



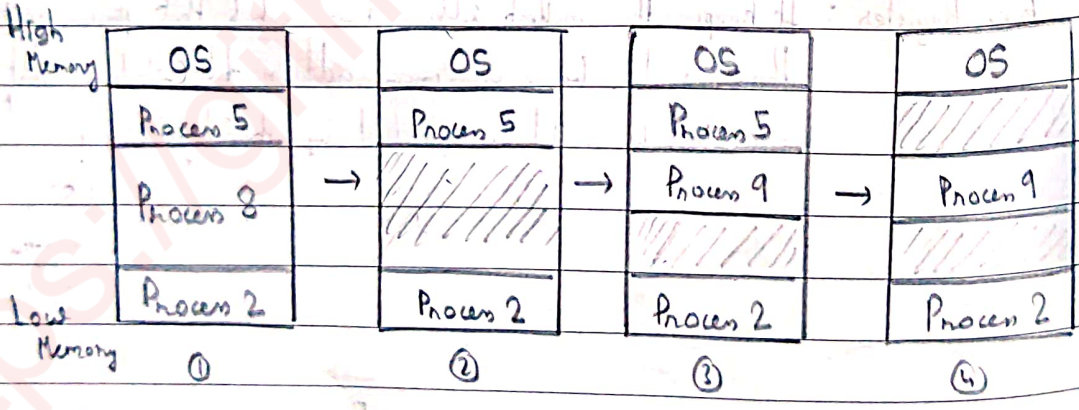
→ Thus, this way MMU maps logical address dynamically.

→ Variable Partition Memory Allocation:

→ Here each partition or section is not of same size, so if there are different partition sizes available, then the OS will have to keep the information of the parts of the memory which available & occupied.

→ Initially, all the memory is available for user process.

Hole: Block of available memory.



→ Suppose the status of the RAM be ①.

→ Now, if the Process 8 completes & leaves the memory then its block of memory which is now available is referred to as a Hole.

→ If a new process has to be brought into the memory then it can be brought & placed into this hole as long as the requirement of that process is less than the size of this hole.

→ So, there may be many holes of various sizes scattered throughout memory.

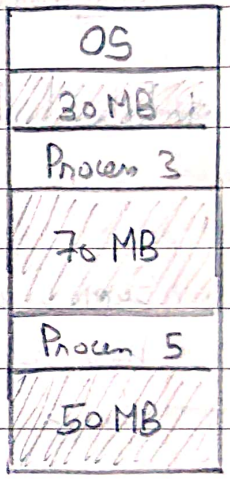
→ Wherever a process arrives, we allocate memory from that hole which is large enough to accomodate it.

→ Also, wherever a process exits, it frees this partition or the section of the memory that was allocated to it & if the partitions (free) are adjacent to each other then they can be combined also.

Dynamic Storage Allocation Problem:

If there is a list of free holes & there is a request of a process which requires 'n' size of memory, then how do we take care of this allocation? What ~~are~~ is the algorithm that can be followed to allocate 'n' size from a list of free holes?

1) First-fit: Allocate first hole that is big enough



Suppose a process comes & it requires 40MB of memory. Now, the OS has information about all the available memory. (OS will check each hole & find the first that fits)

If ~~the~~ the OS use the First-fit algorithm, the process will get the span of hole which has size 70MB & the updated hole size will become 30MB.

2) Best-fit: Allocate smallest hole that is big enough

disadvantage

Here, the OS will search the entire list of holes (unless ordered by size), & choose the 50MB hole for the process which requires 40MB of memory. Also, 10MB will be the updated size of the hole.

Advantage: It produces smallest leftover hole.

3) Worst-fit: Allocate largest hole → disadvantage

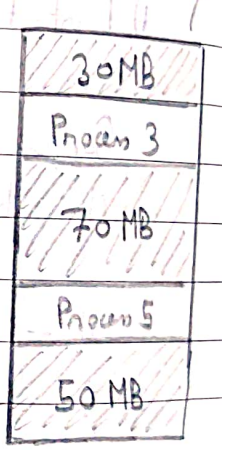
Here, OS will have to search the entire list of holes, allocate the max size hole to the process requesting the memory. Here in the above example, process with 40MB requirement will be given a part of 70MB hole & OS will update the size of hole to 30MB.

Advantage: It produces the largest leftover hole.

→ It has been seen empirically, that the First-fit & Best-fit are better than Worst-fit in terms of speed & storage utilization.

Fragmentation: Problem:

1) External Fragmentation: total memory space exists to satisfy a request, but is not contiguous.



Suppose a process comes & requests 80MB of memory space. Here OS will not be able to allocate any space to this process though total available memory space is 150MB (>80MB)

2) Internal Fragmentation: allocated memory slightly larger than requested memory.

Suppose a process comes & requests 28MB of memory space. Here the OS will allocate the required memory space & update the hole size to 2MB. This 2MB hole will never be going to used as leftover space is very small. This leads to Internal Fragmentation.

→ First-fit analysis reveals that given N blocks are allocated, 0.5N blocks are lost to fragmentation.

→ It mean 1/3 of total memory available blocks may be unusable → 50-percent rule

$\frac{(0.5N)}{(0.5N+N)} = \frac{1}{3}$

→ We can reduce external fragmentation by using (compaction).

→ It says, shuffle the memory contents (holes, process etc) to place all free memory together in one large block.

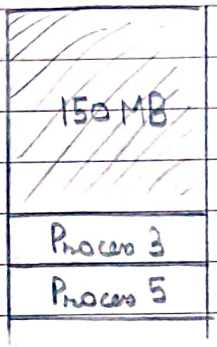
→ Compaction is possible only if relocation is done at execution time.

Compaction Algorithm:

→ Move all the processes towards one end of the memory.

→ All holes move in other direction, producing one large hole of available memory.

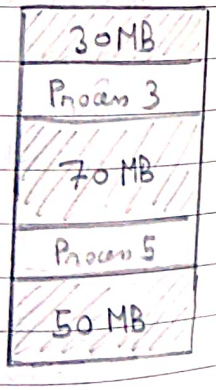
→ It can be expensive.



→ Another possible solution to external fragmentation problem is to permit logical address spaces of processes to be non-contiguous.

→ Process is allocated physical memory wherever memory is available. (in chunks).

→ This strategy is used in paging, most common memory management technique.



⇒ Paging:

→ In paging, the physical address space of the process can be non-contiguous, so its not necessary that the complete process will be put in one section of the memory only, it can be broken up into parts & put in different sections of the memory.

→ So, the process will be allocated physical memory wherever it is available.

→ Thus, this technique will avoid the problem of external fragmentation and also avoid the problem of varying sized memory chunks.

↳ dynamic storage allocation problem

→ Here, the physical memory is divided into fixed size blocks called frames.

→ Size of frames is generally in power of 2, between 512 bytes & 16 M bytes.

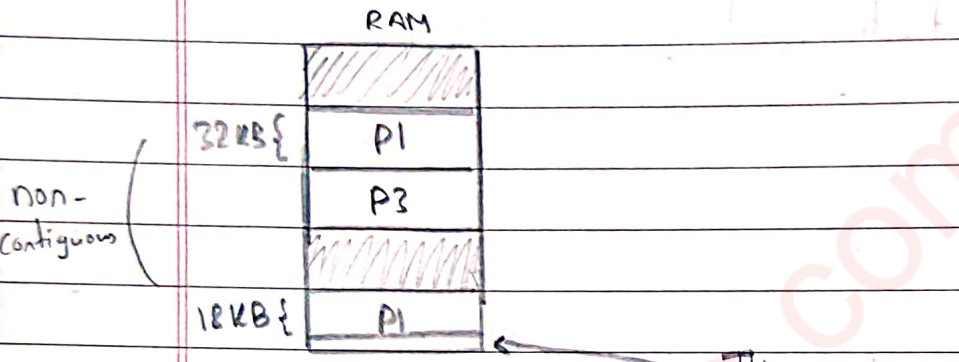
→ Also, the logical memory is divided into fixed size blocks called pages.

(Size same as that of frames)

→ So, the OS will keep track of all the free frames.

→ So, to the run a program which is having N pages in its logical address space, the OS needs to find N free frames in memory to load the program completely.

Ex: A process P1 comes in & it requests 50 KB of available memory. & Suppose there are N frames of 32 KB.

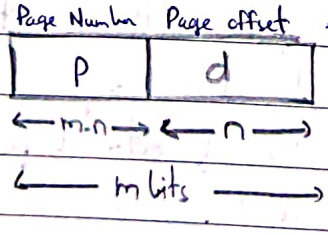


→ The backing store which stores the logical address space of process can be likewise splitted into pages.

→ Still it may result in Internal Fragmentation.

→ Address Translation (LA → PA):

Suppose the total size of logical address of a process be 2^m & size of each page be 2^n.

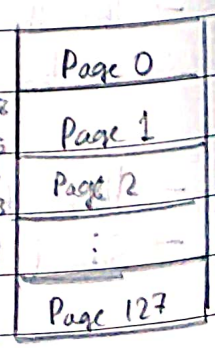


→ base address of above process to the considering process's address

Ex: Given logical address space of size 2^{10} & page size is 2^3

Here m=10, n=3

∴ No. of Pages = $\frac{2^{10}}{2^3} = 27$



Where is the logical address 22 located? → Page-2 (16-23)

binary → 0 0 0 0 0 1 0 1 1 1 0

check

p (7-bits) | d (3-bits)

⇓ ⇓

Page Number = 7 | Page offset = 6 (16 + 6 = 22) ✓

Base = 22 − 6 = 16

→ Now, we have to set up a page table to translate the logical address to physical address.

→ We know that the logical address is generated by the CPU and this logical address while in the logical address space, all of these addresses & pages are contiguous but when it will mapped to the physical memory (RAM), & since we are using paging (to allocate space wherever available), so that means there needs to be some mechanism to keep track of where a particular page is being put in the memory. So, Page Table helps here.

→ Page Table contains the base address of each page (frame) in the physical memory.

LA → 128 Pages, PA → 128 frames

Page Table

	Frame No.
0	5
1	7
2	
⋮	
127	

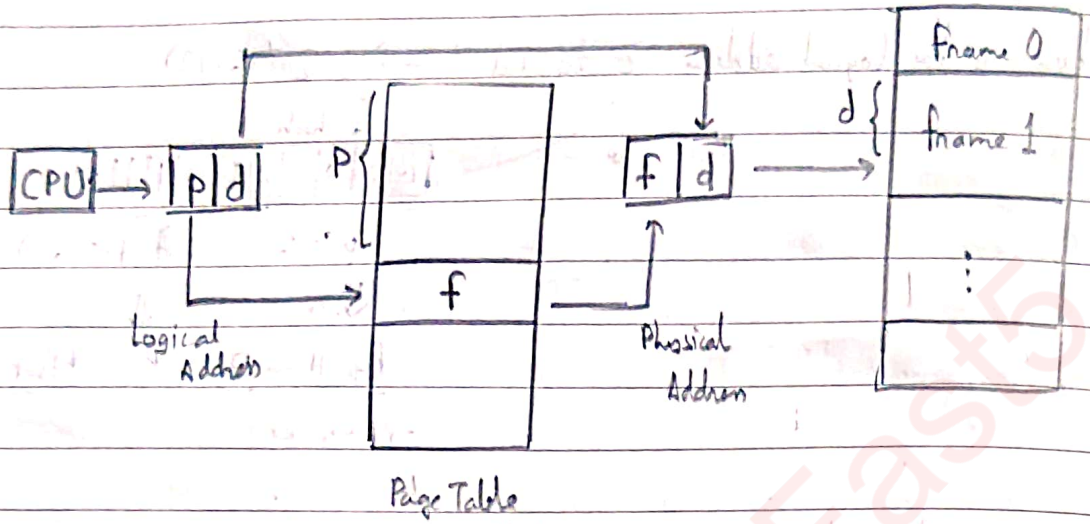
Page No (Index)

RAM: Page 0 → frame 5

→ This frame information along with the page offset information will be combined to give the physical memory address.

(Page offset = frame offset as Page Size = frame size)

Paging Hardware



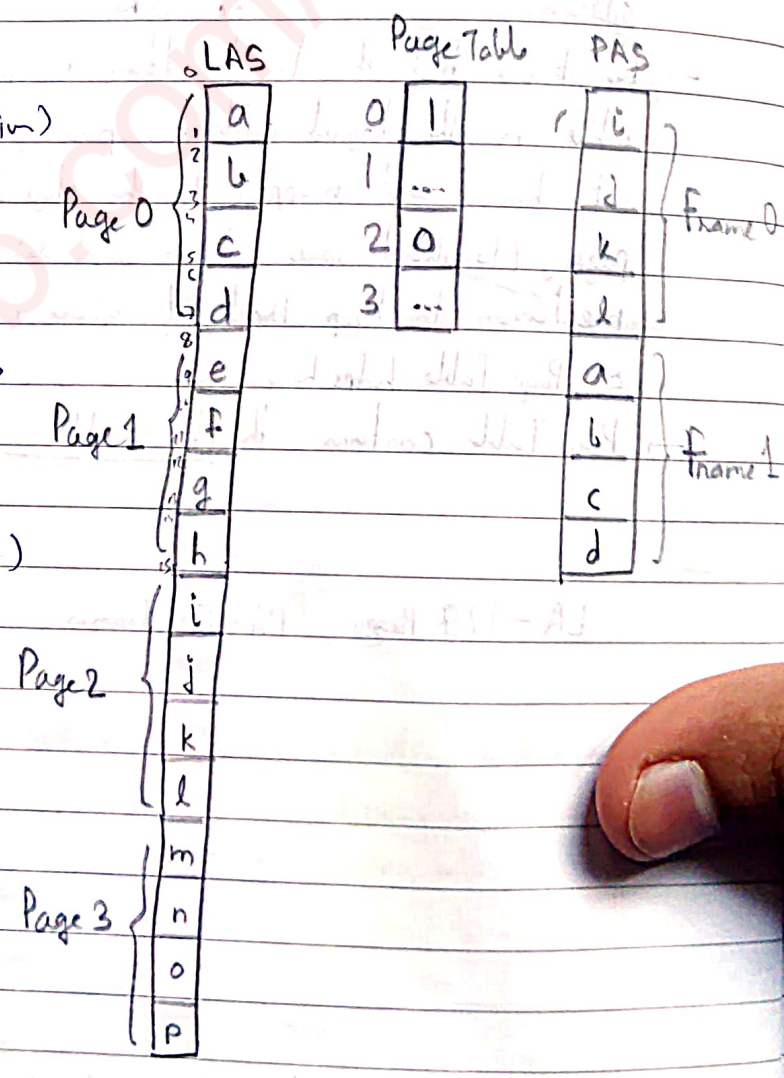
Example

Logical Address : m=4, n=2 (given)

$\therefore$ No. of Pages $= \frac{2^4}{2^{4-2}} = 2^2 = 4$

Page Size = ~~4 bytes (given)~~ $\frac{2^4}{4} = 4$ bytes

Physical Memory = 8 bytes (2 frame)



→ OS is aware of allocation details of physical memory – which frames are allocated, available, total frames etc.

→ Information is generally kept in a single, system-wide data structure called a Frame table.

Valid bit: It is 1 when a page exist on the frame that is allocated.
Invalid bit: It is 0 when a page is no mapping of page with frame or some other page has overwritten the frame.

when a page does not exist in any of the frame

Page 0
Page 1
:
Page 7

LAS

	Frame no.	Valid / Invalid
0	2	1
1	6	1
2	...	1
3	...	1
7	6	0

Page Table
frame

0	Page 2
1	Page 3
2	Page 0
3	Page 5
4	Page 1
5	Page 6
6	Page 4

PAS

→ Calculating Internal Fragmentation:

Page Size = 2048 bytes
Process Size = 72,766 byte

∴ Process requests for 35 pages + 1068 bytes
(71,680 bytes)

∴ Internal Fragmentation = 2048 − 1068 = 962 bytes

Worst case fragmentation = 1 Page (1 frame) − 1 (in bytes)
Size

On avg fragmentation = 1/2 Page Size (frame)

So, to reduce internation fragmentation, should we reduce the frame size (Page Size)?
↓ No
Because it eventually will increase the no. of entries in page table, so page table will require a lot of memory now.

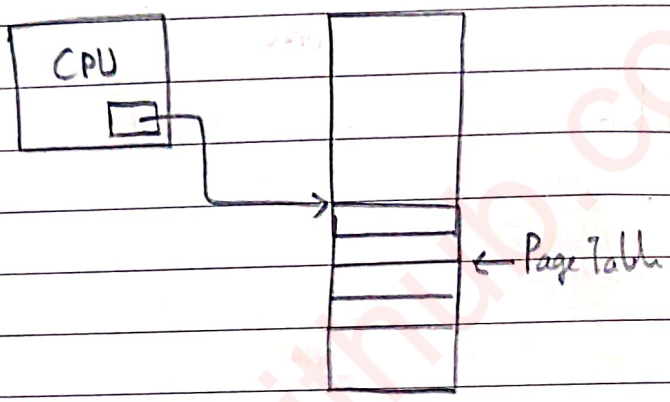
→ Implementation of Page Table:

→ We know that page table is kept in the main memory.

→ Now, everytime a CPU generates the logical address, it has to access the page table to find out where that address is available, in which frame no..

→ So, how does it access the page table, how does it know where the page table is starting from, what is the address of this page table

→ So, the address of the page table is kept in a register which is called the Page-table base register (PTBR) which lies in the CPU.



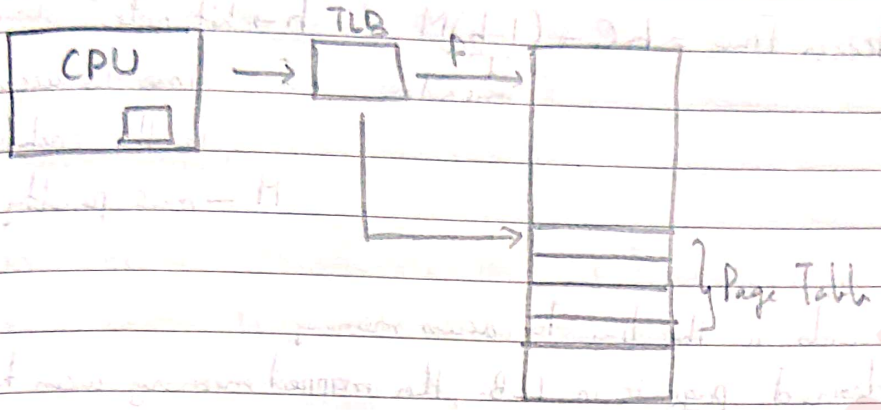
→ So, every logical address that is generated requires two memory accesses, one you will have to access the page table to get the frame no. & other to access the actual physical memory address (based on frame no.).

→ This problem of accessing the memory twice (takes time) for every logical address generated by CPU is referred to as the two-memory access problem.

→ The solution of this problem is to have a fast-lookup hardware cache & it is called the Translation look-aside buffer (TLBs).

→ So, a part of the page table can be kept in the TLB, so when the logical address is generated by the CPU, first the part of the page table which is available in the TLB will be checked, if this page no. for this process was available in the TLB, then the frame no. can be got, the physical address will be computed & then directly we can go to the main memory. If it is a miss then it means

This page number was not found in the TLB, then the page table of main memory needs to be accessed.

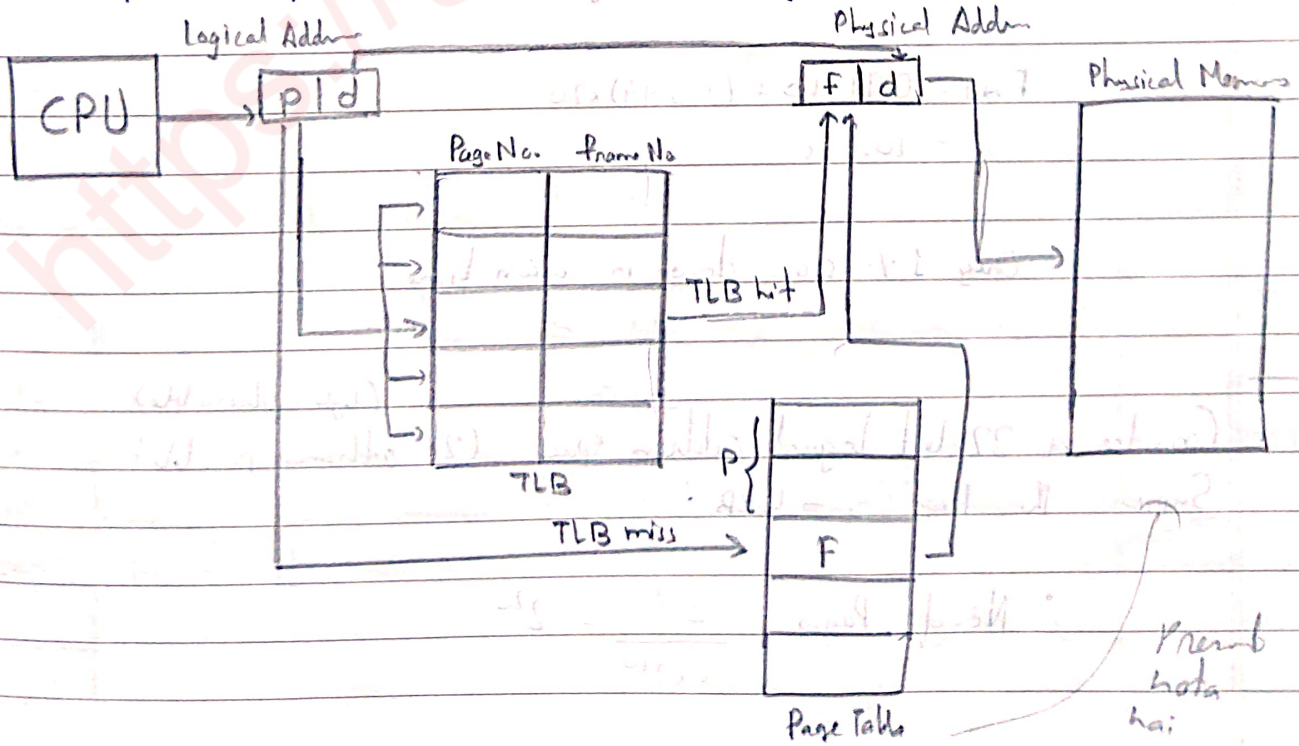


→ Some TLBs, they also use the address space identifiers (ASIDs) i.e. suppose multiple processes are running, so parts of page table of each process will be maintained in the TLBs, so to see that this particular entry is for which particular process, ASIDs are used.

→ TLBs are typically small caches (64 - 1024 entries)

→ When there is a TLB miss, along with getting the frame no. of page table entry in the RAM, the value is also loaded into TLB for faster access next time.

→ Some replacement policies are also there for loading new entries in TLB.



→ Effective Accen Time:

Avg Effectin Access Time = $hC + (1-h)M$ (miss rate)

h → hit rate = $\frac{\text{No. of hits}}{\text{attempted accens}}$

C → time to accen informati- in the cache

M → miss penalty

Ex: → 10 nanoseconds is the tim to accen memory

→ If the desired page is in TLB, then mapped memory accen takes 10 ns

→ otherwin, two memory accen takes 20 ns

→ Assum 80% is the hit rate of TLB

$EAT = 0.80 \times 10 + (1-0.80) \times 20$

$= 12$ ns

% slowdown = $\frac{12 - 10}{10}$ (12: not found in TLB, 10: found in TLB) $\times 100$ = 20% slow down

If the hit ratio is slightly more realistic i.e. 99%.

$EAT = 0.99 \times 10 + (1-0.99) \times 20$

$= 10.1$ ns

Only 1% slow down in accen time

Consider a 32-bit logical addren space (2^{32} addrens possible) (byte addressable)

Suppose the Page Size = 4KB:

$\therefore$ No. of Pages = $\frac{2^{32}}{4 \times 2^{10}} = 2^{20}$

LAS

Page 0	4KB
Page 1	4KB
Page 2	
⋮	
Page $2^{20}-1$	

(2^{12} each)

So, the Page table will also have 2^{20} entries.

Now, if each entry is of 4 bytes (frame no. + valid bits), then the size of page table will be $2^{20} \times 4$ bytes = 4 MB
(for each process)

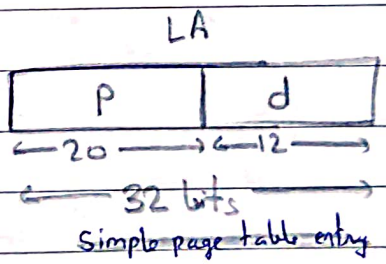
We do not want to allocate this much large space contiguously in main memory as finding the correct frame will become tedious.

Solutions: 1) Hierarchical Paging ✓
2) Hashed Page Tables ✓
3) Inverted Page Tables ✓

⇒ Hierarchical Page Tables (Two-level Page Table):

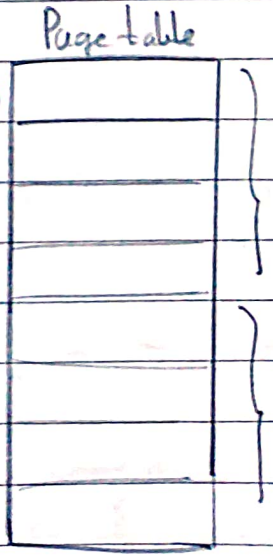
→ Page the page table, as page table size may be large → Multi level page table

32-bit logical address, 4KB page size
No. of Page = $\frac{2^{32}}{2^{12}} = 2^{20}$



Page size = 4 KB & size of each entry in page table = 4 bytes

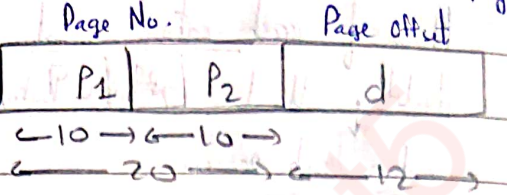
∴ No. of entries of page table in one page = $\frac{4KB}{4B} = 2^{10}$



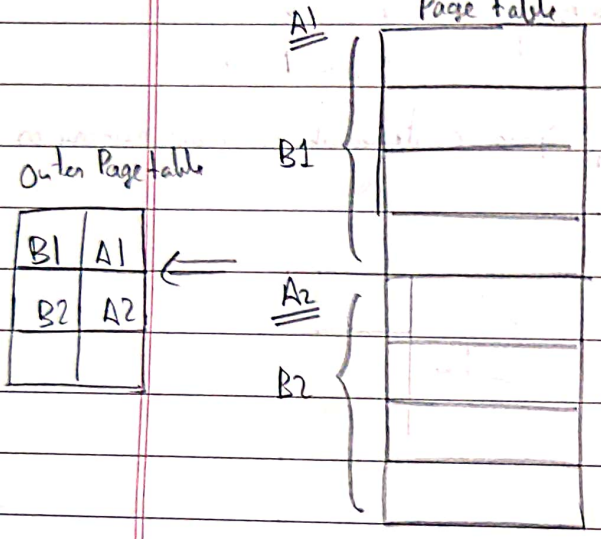
Now, there are two possibilities: one to keep all of this page table pages contiguous in main memory or to keep each page in a separate block of memory

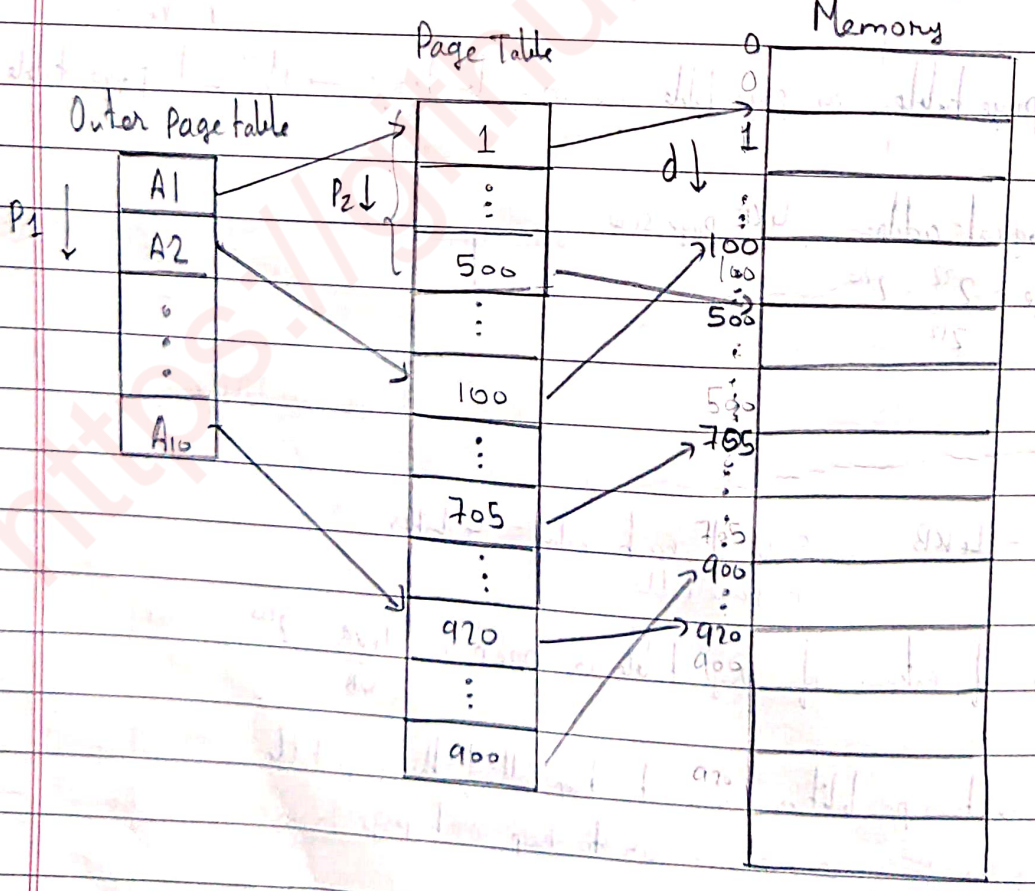
To page the page table, page number is further divided into : 1) 10-bit page no
2) 10-bit page offset

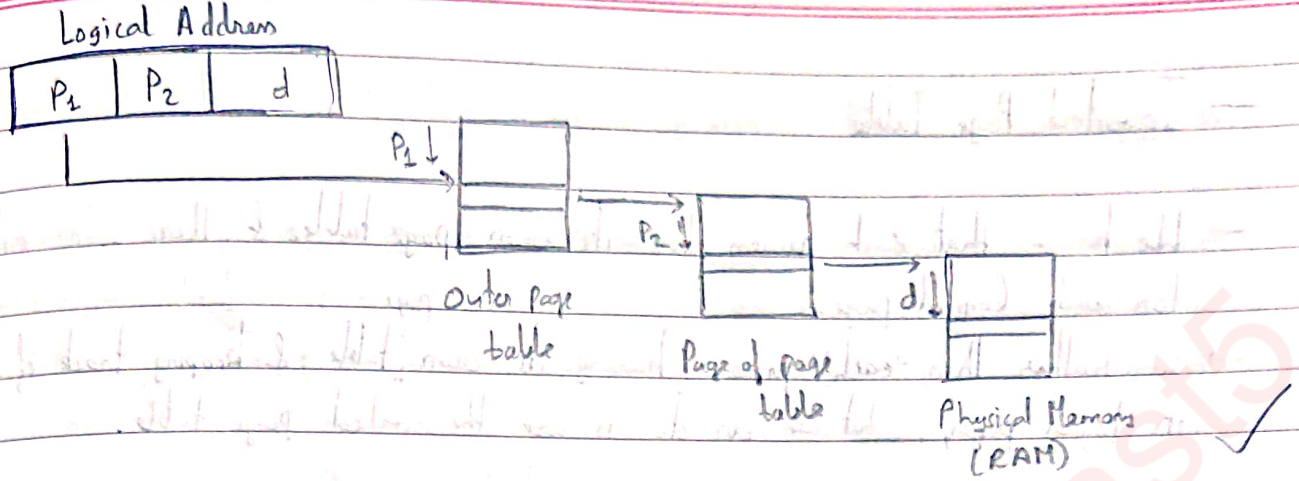
Thus, the logical address now looks like:



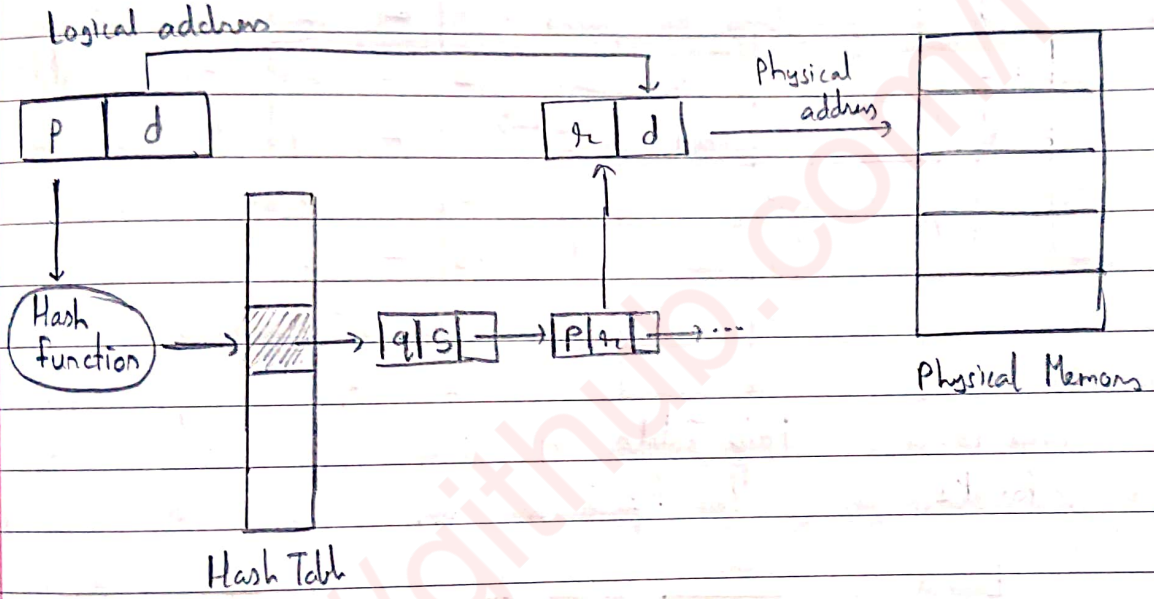
p1 → index of (into) outer page table
p2 → displacement within page of inner page table







⇒ Hashed Page Tables:



(Logical)

→ Here, the virtual page no. is hashed into the page table.
→ This page number value is used with a hash function to hash into the hash table.
→ This hash table has many elements & each elements contains a linked list as many logical addresses may hash into the same location.
→ Each elements contain : 1) Virtual page number
2) Value of mapped page frame (frame no. which the virtual page number has been mapped to)
3) pointer to the next element

→ So, the page number of the logical address will be compared ~~to~~ with all the page nos one by one as the list is traversed.
→ Suppose a match is found, it mean r is the value of frame no. where this virtual page is available, so this frame no. (r) will be taken & the offset which remains the same will be taken from the logical address to now compute the physical address.

→ Inverted Page Table:

→ We know that each process has its own page table & there is an entry for each logical page.

→ So, rather than each process having its own page table & keeping track of all its logical pages, what we can do is use the inverted page table.

	frame no.
0	4
1	0
2	1
3	
4	
5	
6	
7	

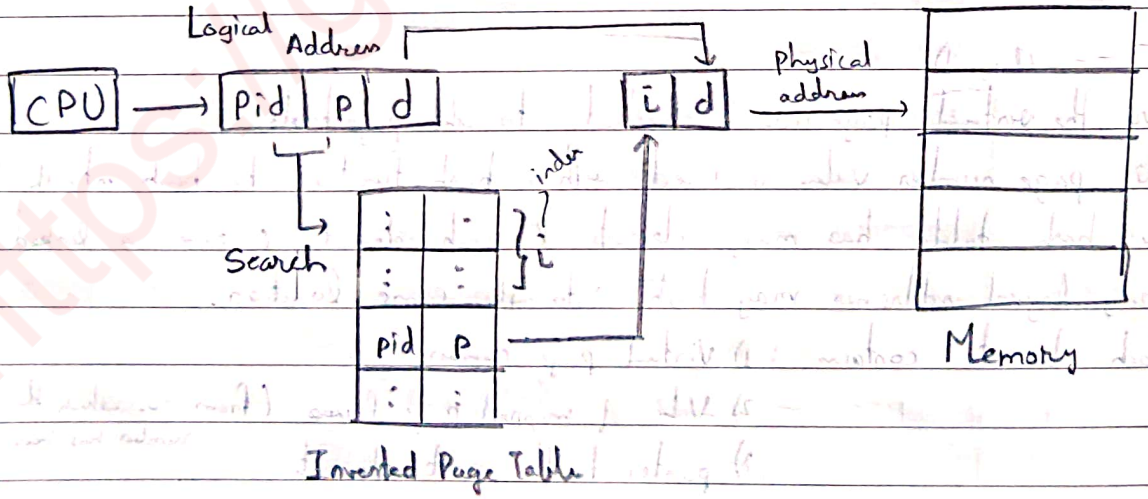
Page Table for P1

	frame no.
0	2
1	3
2	5
3	
4	
5	
6	
7	

Page Table for P2

	frame no.	pid	Page no.
0	1	1	
1	1	2	
2	2	0	
⟷ 3	2	1	
4	1	0	
5	2	2	

Inverted Page Table



→ So, the CPU is generating the logical address which contains pid (Process id), Virtual page no & page offset.

→ Now, this pid will be used to map into the inverted page table, so there will be a search to check where this pid is available.

Linking → COA
Loading → COA

→ Once that pid is found, there could be multiple entries, then the page no. is to be checked.
→ Once the entry is found in the inverted page table, the value of i from the box gives the frame no.
→ So, thus the value of i & offset (same as logical addres) will combine to give the physical address, through which it can access the physical memory.

→ Thus, we can see that this decreases the memory that is required to store page table for each process but it increases the time which is needed to search the table wherever there is a page reference.
→ One solution for this can be to use a hash table to limit the search to one or atmost a few page table entries & for using the hash table also, the TLB can also be used to store a part of the hash table & the inverted page table.

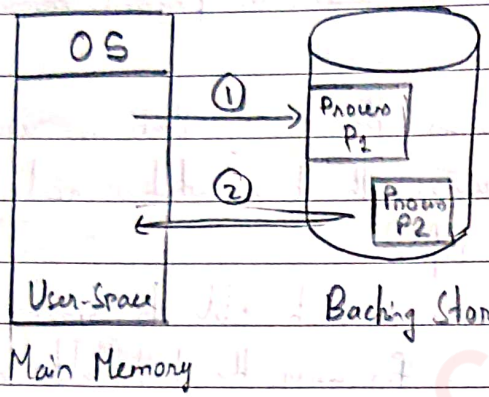
⇒ Swapping:

→ We know that other processes are being executed, they loaded into the main (or physical) memory. ~~but~~ (a new process can enter into the memory)
→ A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. This improves main-memory utilization.
→ These swapped-out processes are stored in swap-space also referred to as backing store.

Backing Store:
→ It a fast secondary storage disk that is large enough to accomodate copies of all memory images for all users.
↓
Process available in main memory
→ Backing store must also provide direct access to these memory images when they are to be retrieved back into the main memory for execution.

Swap out, Swap in (Roll out, Roll in):

→ This swapping variant is used for priority-based scheduling algorithms i.e. lower-priority process ~~are~~ is swapped out, so high-priority process can be loaded & executed.



→ We know that the physical address space which is available in the memory of the system is limited in natur.

→ So, swapping makes possible to use this limited space so that it kind of feels that we can exceed the real physical memory of the syster.

→ So, this way it increases the degree of multiprogramming in a system. because now new process & more processes can be accomodated in the main memory by swapping out the existing processes onto the backing store.

→ Now, when these processes which are in the swap space are brought back into the main memory, is it necessary that they come back to the same physical addres?

→ This depends on the address binding method

Compile/Load time	→	run time
not possible to change the physical addres		possible to change the physical address

→ The system maintains a ready queue of ready-to-run process which have memory images on the disk, so that they can be put back into the

main memory when required.

→ The major part of swap time is the transfer time & this transfer time is directly proportional to the amount of memory which is swapped. i.e. size of the process which is swapped into the swap space will determine the transfer time.
→ Now, if the next process which is to be put on the CPU is not in the memory then we need to swap out a process & swap in the target process.
→ so, context switch time in this case can be very high.

→ Suppose there was a 100 MB process which was to be swapped to the hard disk with the transfer rate of 50MB/s.

∴ Swap-out time = 2s

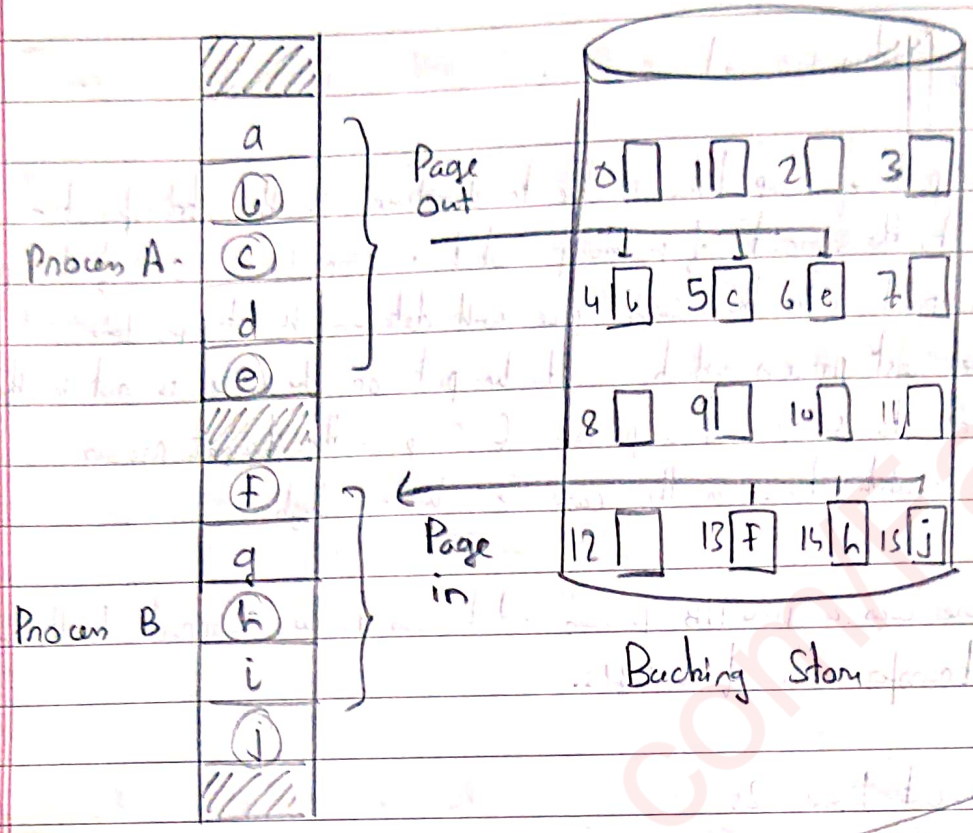
If the process which is being brought into the main memory is also of same-size then,

Swap-in time = 2s

∴ Total context switch swapping time = 4s (which is large)

→ so, standard swapping is not used in Modern Operating Systems unless if the free memory becomes extremely low.

→ Nowadays, Most systems use a variation of swapping where rather than swapping out the entire process, only certain pages of the process will be swapped out.
→ Here, a page out operation moves a page from the memory to backing store & if it is brought from the backing store to memory, it is refered to as a page in operation.



⇒ Virtual Memory :

→ We know that if instructions have to be executed then they must be in the physical memory, so any program which is lying on the disk, it has to be taken to the RAM (physical memory) so that it can be executed by the processor

→ If the size of main memory is limited that means the size of the program will also be limited because now you can fit only that size of the program which can be fitted into the RAM.

→ But we have seen in many cases that the entire program is not needed like there are certain codes which handle error conditions, then there are also static memory allocations to arrays & other certain features of programs which are not needed so they are not taken to RAM, when the program is loaded into the main memory.

→ Also, the entire program may not be needed in the memory at the same time

Refer to diagram of 1st page of MMU

(Rest is kept in Hard disk only)

→ Ability to execute a program only partially in the memory can have many benefits:

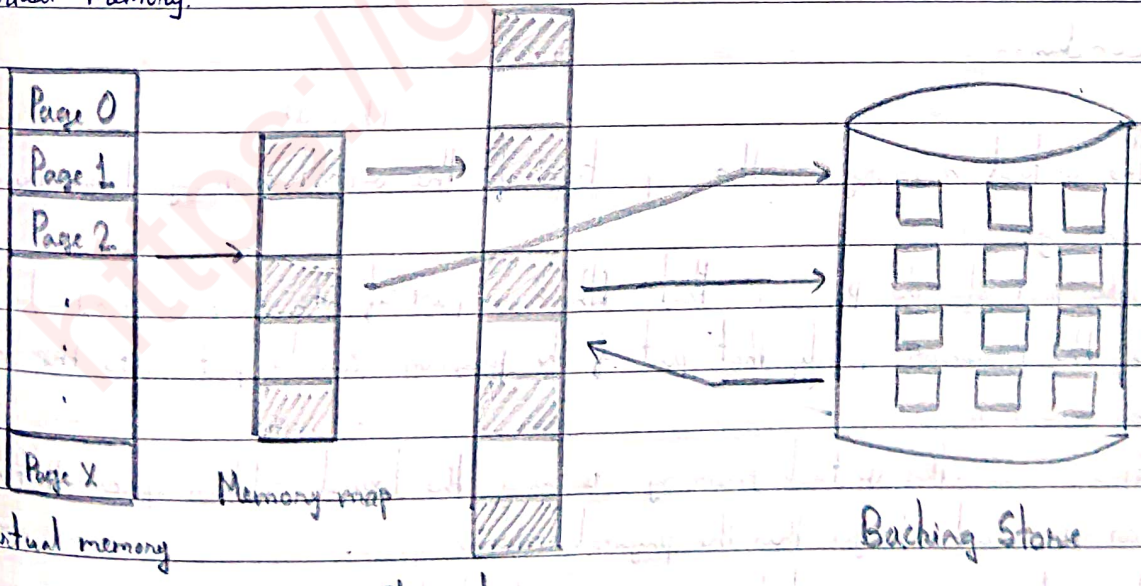
1) Program size is no longer constrained by the amount of the physical memory available.
2) If each program takes less physical memory, more programs can run at the same time, thus increasing CPU utilization & Throughput.
3) Also, since we are taking only a part of the program into the main memory, that means less I/O will be needed to load or swap portions of programs into memory.

→ This concept can be implemented using Virtual Memory.

→ Virtual memory involves separation of the logical memory from the physical memory.
(perceived by developers) (actually available)

→ So, this concept allows, a large virtual memory to be provided for the programmers even if a smaller physical memory is available.

→ So, now the developers or the programmers are not limited by the size of the physical memory that ~~will~~ is be available in the system, they use the concept of a large virtual memory.



Explanation
PTO

Above diagram:

→ There is a virtual memory where the whole program is divided into pages and this is what is seen by the developers as a consecutive or sequential set of addresses (set of addresses divided into pages).

→ Then there is a memory map that says which page belongs to or is kept in which part of the physical memory.

→ Now, in the physical memory, we will take only that page which is actually needed, so the page of that virtual memory program will be put according to the memory map into a particular part of the physical memory & this memory map will also keep track of where these pages are available on the backing store.

→ From the backing store, the pages can be taken into the RAM & can be taken out of the RAM also.

→ So, this virtual memory which is in the form of pages is put in the backing store & from the backing store, the pages will be taken to the physical memory.

→ so, this virtual memory is only a concept of consecutive addresses which is seen by the developers.

Conclusion:

→ The virtual (logical) memory is the separation of the virtual memory from the physical memory.

→ ~~Only part of the program that~~ Whichever part of the program (instructions) are required by the processor, only that part of the program can be put into the memory for execution.

→ Also, since this virtual memory is using the logical address space that means it can be much larger than the physical address space.

→ Now, this program will be kept on the hard disk which is much larger in size i.e. now the program is not limited by size of the physical memory.

→ So, the physical address space in the physical memory can be shared by several processes.

→ Also if the space in physical memory is being used by a part of process & when it is taken out, the same space can be used by another part of another program.
→ So, Since more programs can be put or loaded into the memory, you can have multiple parts of multiple programs loaded into the main memory & those programs can run concurrently by the CPU as per some scheduling algorithms.

→ So, the logical view of the processes that are stored in the memory is consecutive addresses which are consisting of pages. & the physical memory is organized in page frames.
→ MMU translates the logical addres to the physical addres

→ Demand Paging:

→ We know that we can bring the entire process into the memory at the load time but we may not require the entire program in memory initially.
→ So, what we can do is that we can bring a page from logical addres space to memory only when it is needed (demanded).
→ The benefits of this are:

on i/o

- less I/O is needed & no unnecessary I/O is there in order to stop wasting time
- less memory is needed
- faster response
- more users can be entertained

→ Wherever a page is needed by the CPU, there is a reference to it i.e. its particular addres which will be first checked whether it is a valid reference or not that means if the process is referring to a particular addres in addres space or trying to access addres space of some other process.
→ If it is invalid, the process is aborted or terminated.
→ If it valid, page is not currently in memory, then it is brought into memory i.e. the from swap space or the secondary storage.

→ So, the basic concept of demand paging is to load a page in memory only when it is needed.

→ When a process is executing, there is a possibility that some pages will be in memory & some page will be there in the secondary storage.

→ So, how does the system distinguish, whether the page is in the memory or it is in the secondary storage?

→ So, for this we have valid-invalid bit which is also put in the page table.

- 1 (v) (valid) — Page present in the main memory
- 0 (i) (Invalid) — Page not present in the memory

→ Initially, the valid-invalid bit is set to i for all entries which shows that ~~show~~ no previous data which is available in the page table is valid.

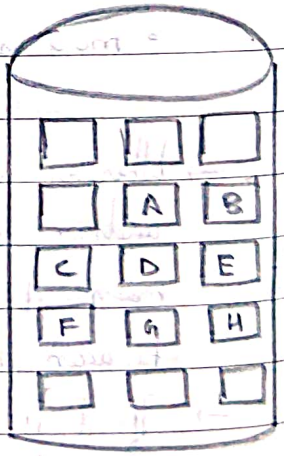
→ Now, during MMU address translation, if the valid-invalid bit is i that means the page is not in the memory, so this results in a page fault. which means that the processor is referring to a page which is not in the memory & now it has to be brought from the secondary storage to the main memory.

Page		Frame	V/i bit
0	0	4	v
1	1		i
2	2	6	v
3	3		i
4	4		i
5	5	9	v
6	6		i
7	7		i

Logical Memory — Page Table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
...	

Physical Memory



Backing store

→ Handling Page Faults:

→ If there is a reference to a page by the CPU, first the MMU will check whether the reference is valid or an invalid memory access.
→ If invalid, the OS will terminate the process. Otherwise the reference is valid & is referring to its own address space & not to any other address space of another process, & if it is not in memory, we are having a page fault.

→ Now, the OS will be send a trap signal & following things the OS needs to do:
1) find a free frame in the main memory
2) It has to swap the page which is required into that free frame via a scheduled disk operation
3) It has to also update the page table to indicate that the page is now in the main memory (valid-invalid bit → V)
4) Restart the instruction that caused the page fault.

Free-Frame List:

→ Whenever there is a page fault, we know that the OS must bring the desired page from the secondary storage into the main memory.
→ Most of the OSs maintains a free frame list which is a pool of free frames for satisfying such requests.
→ Typically, the free frames are allocated using zero-fill-on-demand technique, that means whenever a frame is allocated to another page, the content of the frame is erased before it is allocated.
→ When the system boots up, all the frames of memory are placed on the free frame list.
→ Now, as the processes starts running & as the free frames are requested, the size of free frame list gradually shrinks.

Performance of Demand Paging:

If no page faults ⇒ Effective Access Time = Memory Access Time

If there is a page fault ⇒ now, that page has to be read from the secondary storage & then brought to the main memory & then only the desired address can be accessed.

Suppose: Memory access time → ma
Probability of page fault → p $(0 \le p \le 1)$

∴ Effective Access Time = $(1-p)ma + p \times$ (page fault service time)

Ex: ma = 200ns, page fault service time = 8ms

$EAT = (1-p)200 + p \times 8000000 = 200 + 7999800p$

If 1 out of 1000 accesses causes page fault, EAT = 8.2 μs

↓

slowing down of system by a factor of 40

→ Copy-on-Write:

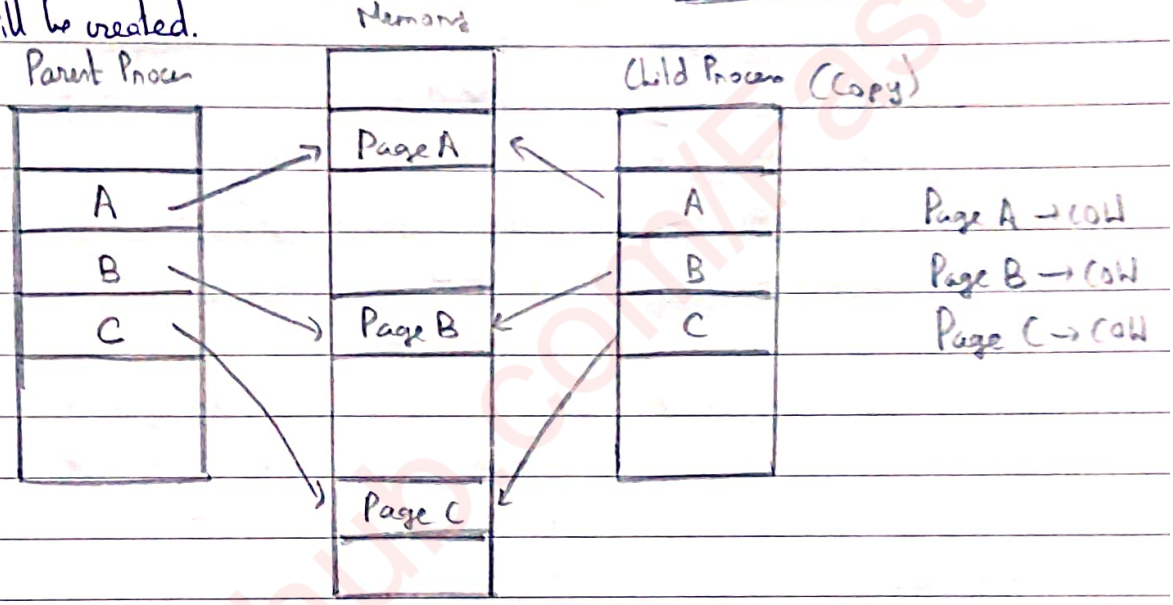
→ We know that the fork() system call is used to create a child process. So, a parent process will call this fork() system call to create a child process which will be the duplicate of parent process.
→ so, a copy of the parent's address space will be created for the child.
→ so, whatever pages were belonging to the parent, similiar pages will be created in the child process also.
→ Now, once a child process is created, it will usually invoke the exec() system call

So, that it could load a new program in its address space.

→ So, the copying of the parent's address space becomes unnecessary or not required.

→ To avoid this unnecessary overhead, we used a technique known as Copy-on-write.

→ This technique allows parent & child processes initially to share same pages. & no duplicate copies will be created.



→ Shared pages will be marked as copy-on-write pages.

→ Only those pages that can be modified will be marked as copy-on-write.

→ The pages that cannot be modified like executable files, can still be continued to be shared by the parent & child.

→ If either of the process, parent or child wants to write to a shared page. then a copy of only that shared page will be created.

→ So, whichever process wants to write on that page, now will be able to write on the new page that is created. & lose the pointer to previous copy.

→ So, the ~~read~~ unmodified pages, they will still be shared by the parent & child process.

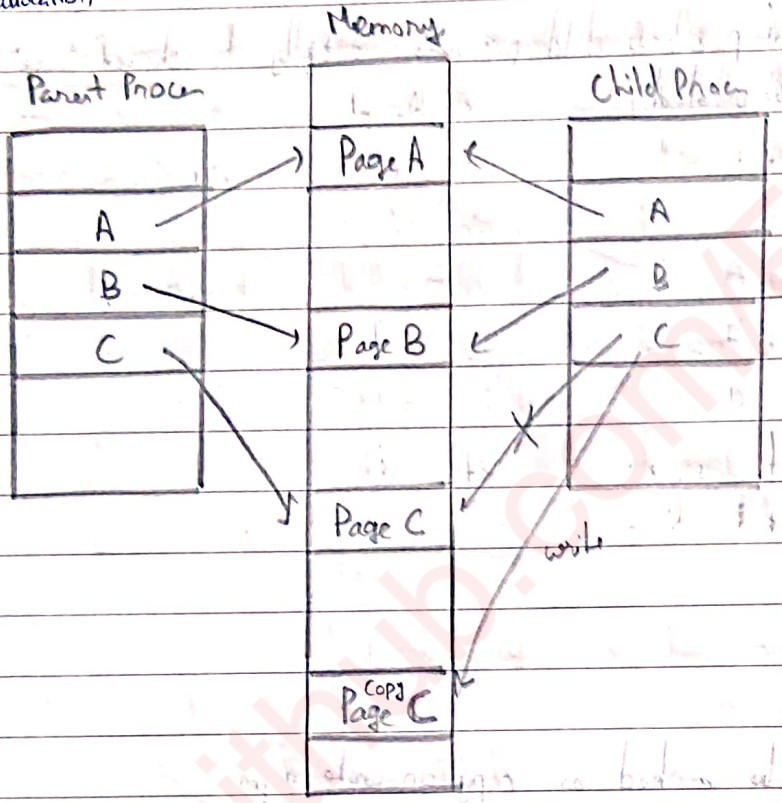
→ This allows a more efficient process creation because now at process creation, all the pages of the parent do not have to duplicated for the child. So, at the initial stage both parent & child can share the same pages & as and when the either process which wants to modify a page, then only the modified pages are copied.

→ Wherever a process wants to create a new page or a new page has to be copied, it has to be allocated a new frame.

→ So, for the new frame, the free ~~free~~ frames are allocated from a pool of zero-fill-on-demand frames.

zero-fill: frame is erased before allocation

on-demand: when required



→ Page Replacement:

→ We know that if the processor requests for a particular page & if that page is not available in the main memory (or the physical memory) then there will be a page fault.

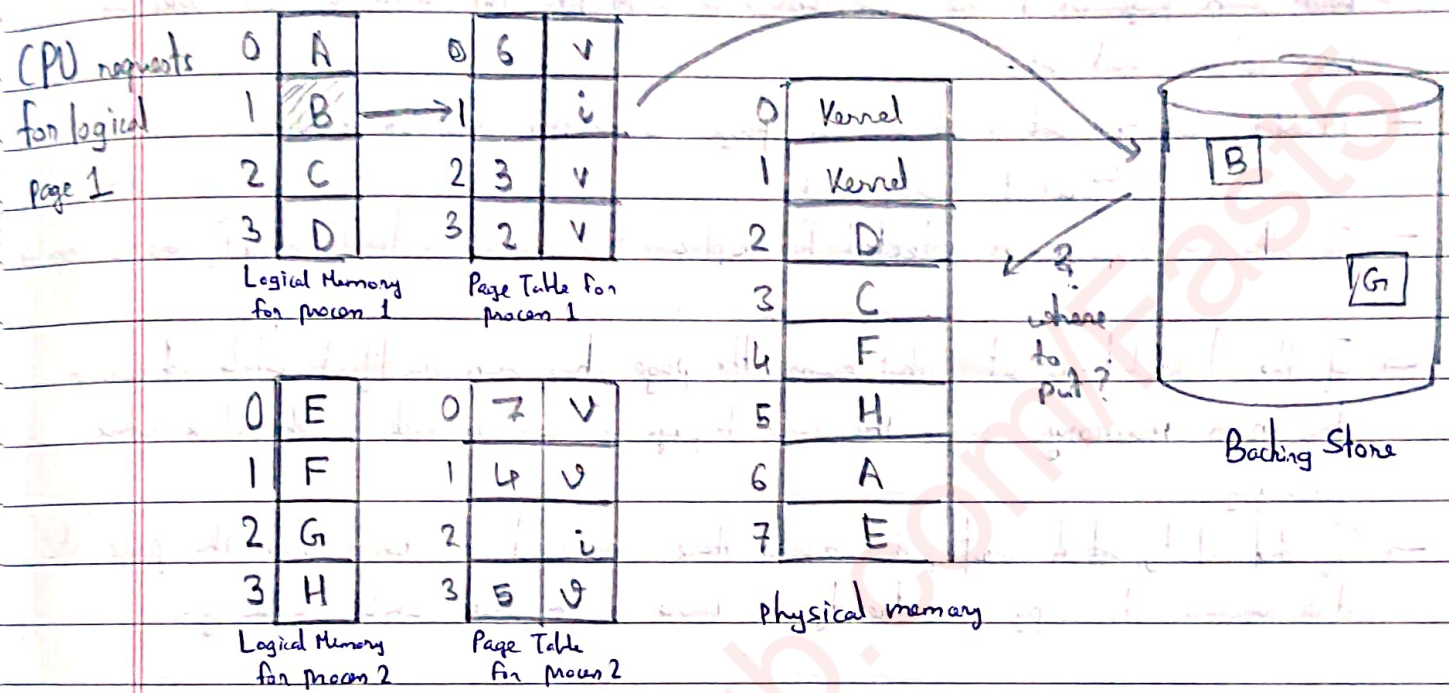
→ When this page fault occurs, the OS must bring the desired page from the secondary storage & bring it to the main memory.

→ Now, when the page has to be brought into the main memory, a frame should be available in the main memory, so that this page can be brought in over there.

→ But what happens if there is no free frame because all the frames might be used either by the OS or it might be used by the various processes.

→ So, in this case, we need a page replacement i.e. we will find some page

in the main memory which is currently not in use because of may that process is in suspended state, so will find such frame in the memory & then free it.



Page Fault Service Routine:

→ Find location of desired page on disk.
→ Find a free frame in the memory to put the desired page.
  ↳ free frame available → use it
  ↳ not available → use page replacement algorithm to select victim frame.
→ Now, we can bring the desired page into (newly created) free frame & then also update the page & frame tables.
→ Now, the CPU can continue that process by restarting that instruction that caused the trap.

If free frame is available → one page transfer from Secondary to main memory. (desired page) (disk)

If no free frame is available → two page transfers: 1) Victim frame → MM to HD
2) Desired from → HD to MM

Thus, it increases the effective access time.
This overhead can be reduced by using a dirty (or modify) bit.

Dirty (Modify) Bit:

→ With each page or frame, we associate a modify bit with it just like we have valid-invalid bit.

→ Modify bit → set : whenever a page is written into or modified
→ unset : only read

→ So, Whenever a page is selected for replacement (to become victim), modify bit is exami

→ If the dirty bit is set, that means the page has been modified while it was in the main memory, so in this case, the page is written back to the disk while replacing.

→ If the dirty bit is unset, that means there is no need to write back the page into disk because its copy is already available in the disk. while replacing.

→ This technique also applies to read-only pages like pages of binary code, executable files which cannot be modified. So, such pages can be discarded whenever desired because their copy is already available in the disk.

→ So, this technique significantly reduces the time which is required to service a page fault.

→ This also reduces the I/O time by half if the page has not been modified because now only the new page is brought into the main memory & the victim page is not written back.

There are many different page replacement algorithms which can help us identify a victim frame that means which particular frame will be freed but how do we select the best page replacement algorithm?

→ We can evaluate an algorithm by running it on a particular string of memory references & computing the no. of page faults.

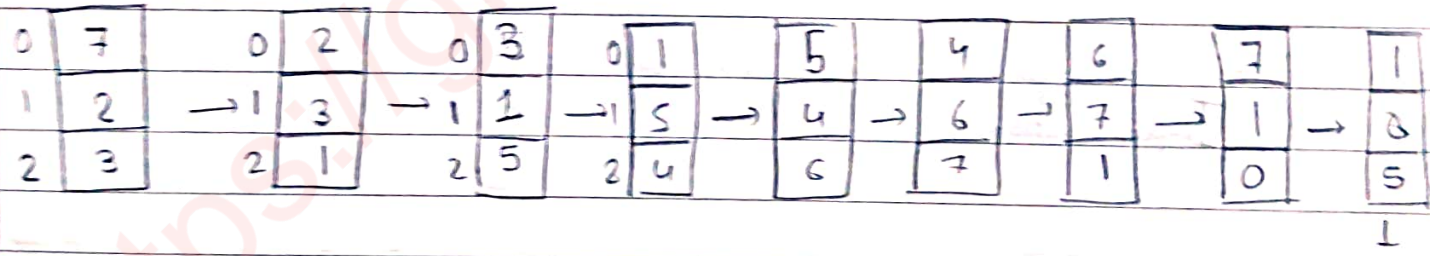
→ This string of memory references is called a reference string.

→ Whichever algorithm has the least page faults for a reference string, the algorithm is considered to be a good replacement algorithm.

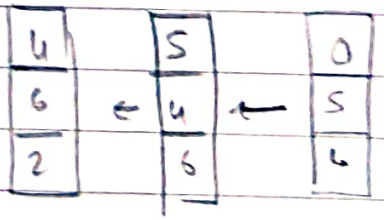
1) FIFO Page Replacement Algorithm:

→ It is the simplest page-replacement algorithm.
→ With each page that is brought into the memory from the disk, a time unit is associated with it which is maintained.
→ Now, whenever a page has to be replaced to bring a new page when free frames are not available, then the time with the page which is the oldest is chosen as the victim frame.
→ Rather than keeping track of time, we can also create queue to hold all the pages in the memory & we will replace the page which is in the front of the queue & insert the new page at the back of the queue.

Ex: Reference String: 7(F), 2(F), 3(F), 1(F), 2(P), 5(F), 3(F), 4(F), 6(F), 7(F), 7(P), 1(F), 0(F), 5(F), 4(F), 6(F), 2(F)
F - Page Fault
Memory with 3 frames



∴ No. of Page faults = 14



→ Easy to implement, however performance not always good.

Does increasing the memory size, increases the performance of FIFO? No
This is known as Belady's anomaly. i.e. for some page-replacement algorithms, page fault rate may increase as the no. of allocated frames increases.
Ex: 1,2,3,4,1,2,5,1,2,3,4,5
i) Size: 3 Frames. → 9
ii) Size: 4 Frames → 10

2) Least Recently Used (LRU) Page Replacement Algorithm:

→ In this algorithm, we will replace the page that has not been used in the most amount of time.

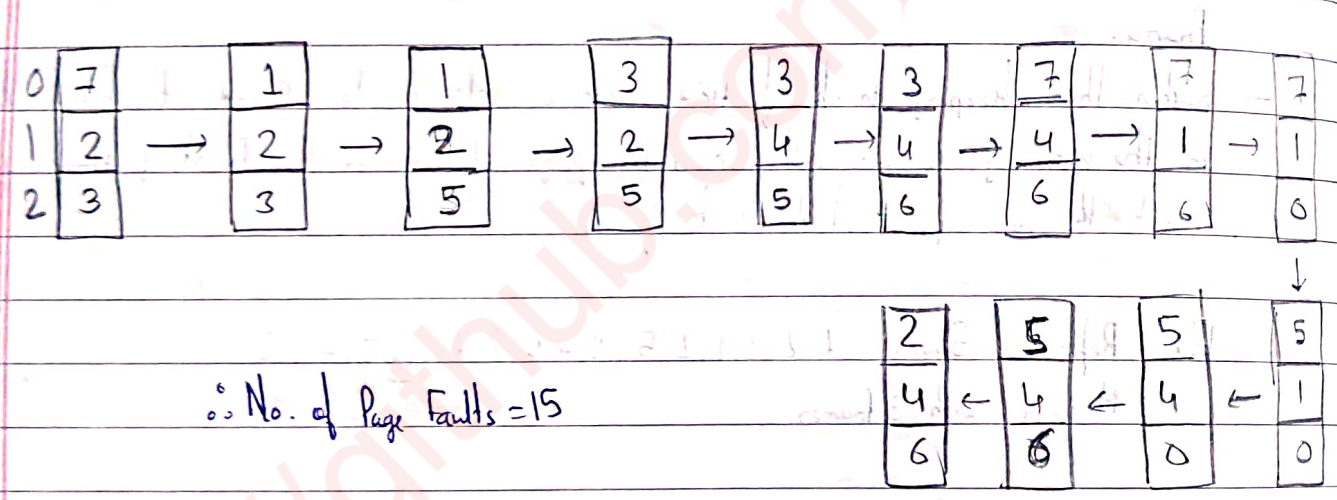
→ So, that means with each page, now we will have to associate the time of last use.

→ Considered a good algorithm & as it has less no. of page faults.
(& frequently used)

addrss is lying on page 7 ↓ ← Look towards left

Ex: Reference String : 7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2
P P P P P P P P P P P P P P P

Memory with 3 frames:



∴ No. of Page Faults = 15

Implementation:

3) Optimal Page Replacement Algorithm:

→ This algorithm says that if a page has to be replaced in the main memory, then replace that page which will not be used for the longest period of time in future.

→ So, if we know that a particular page is not going to be used in future, then we can remove that page from the main memory to bring in a new page from the hard disk but how do we know which page will not be used in future because we do not know the future memory references of the reference string.

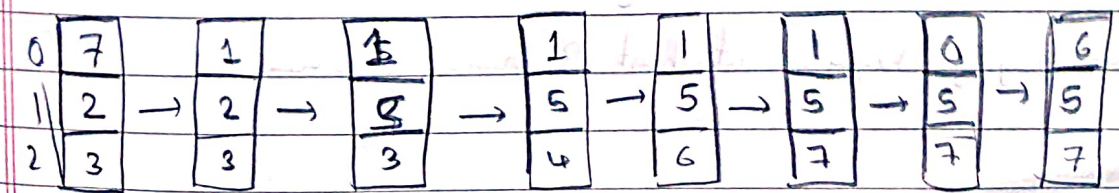
→ So, this if it could be implemented then this would be the ideal page replacement algorithm.

** → So, this page replacement algorithm is actually used for evaluating how well the other algorithms are performing.

Ex: Reference String : 7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 6, 7
(P P P P P P P P P P)

Memory with 3 frames.

∴ No. of Page faults = 10



→ Frame Allocation:

→ We have seen that any process which is being executed, the pages of that process are brought into the main memory & put in the memory frames and each process needs a minimum no. of frames.

→ The reason for allocating this minimum no. is due to the performance because if the no. of frames which are allocated to each process decrease then the page fault rate increase & thus the process execution will slow down.

→ Also, once a page-fault occurs, that particular page has to be brought from the disk to the main memory into a free frame & that instruction which caused the page fault will have to be restarted.

→ so, all of this leads to degradation in performance.

→ so, that means that any process which is running, should have enough frames to hold all the different pages that any single instruction can reference.

Minimum Number of Frames:

Ex: Consider a machine where all memory-reference instructions may reference only one address.

↓

At least one frame would be required for the instruction & one frame would be required for memory reference.

Ex: If one-level indirect addressing is allowed: for ex, a load instruction on frame X can refer to an address on frame Y, which is an indirect reference to frame Z

↓

Then paging requires at least 3 frames per process.

Minimum number of frames per process → defined by architecture
Maximum number of frames per process → defined by amount of available physical memory

How to allocate fixed amount of free memory among various processes?

If there are 'm' free frames & 'n' processes, how may frames does each process get?

↓

Two major allocation schems:
1) Equal allocation 2) Proportional allocation.

→ Thrashing:

→ We have seen that there should be a minimum no. of frames which should be allocated to a process.
→ If the process does not have enough frames, then the page fault rate will become high because it is not having the required no. of frames which are needed for its execution.
→ Now, suppose a process P is there which required a page P_1 but it is not present in the memory, so it will page fault by swapping the page say P_2.
→ But now suppose the process which required Page P_2, now again requires P_2, again page fault will occur.
→ So, this high paging activity is known as Thrashing.

→ That means continuously or very frequently, a process is facing page faults & it needs to bring pages which are not there in the main memory from the hard disk.
→ So, the process is spending more time in paging i.e. bringing pages from the disk to the memory, rather than executing.
→ So, the process is busy swapping the pages in & out.

→ The OS monitors the utilization of CPU.
→ If the utilization of the CPU is too low, the OS thinks the CPU does not have too much work, the processes do not require the CPU, so the OS will increase the degree of multi-programming i.e. the OS thinks the ~~OS~~ CPU is lying idle & it will feel that it needs to bring another process to the main memory from the disk by taking the frames which have been allocated to the existing process.

→ can take frames from any process (even itself)

→ Suppose a global page replacement algorithm is being used.
→ Now, if a new process enters & it needs more frames, then the new process will start faulting and then it will take frames away from all the other existing processes.

→ Now, the existing processes need those pages & now when the new process takes those frames away from the existing process, the existing processes will also start to fault & they will take frames from other processes.

→ Let's say initially we had P_1, P_2 & P_3, three processes in the memory & then processes did not have enough frames, so they were faulting.
→ So, these processes were spending more time in input-output, the CPU utilization went low, so the OS brought in a new process P_4.
→ Now, P_4 need free frames & ∵ global page replacement is being used, so P_4 will take away frames from P_3 or P_2 or P_1.
→ Already these processes did not have enough pages & now P_4 has also taken their frames, so these will now start faulting again & P_4 will also fault.
→ So, finally the faulting processes will queue up for paging & are waiting to get a page (frame) to bring its req. page to memory from the disk.
→ Now, when all of these processes have lined up for I/O or paging then again the CPU is sitting idle.
→ So, CPU utilization will decrease, so again the OS will increase the degree of multiprogramming resulting in a deadly cycle.
→ So, this is called thrashing, when the processes are having multiple page faults due to lack of frames.

Prevention :

→ To prevent thrashing, what we can do is to provide a process with as many as frames as it needs. But how do we know how many frames, the process needs?

Strategy 1 :→ We start looking at how many frames, the process is using.
→ So, a locality model of process execution is defined.
It is the set of pages which are actively being used
→ As a process executes, it moves from locality to locality.
→ A running program is composed of several different localities which may overlap.

→ So, the technique is that to allocate enough frames to a process to accomodate its current locality.

→ So, ~~the~~ the process will fault for pages which are in the locality till all these pages are available in the memory.