Operating System

Operating System:- It is an software which manages the resources of hardware and software and provide user such a environment that all the software can run efficiently without interfering others.

why OS?

What if there is no OS?

a) Bulky and complex app (Hardware interaction code must be in app's code base).

b) Resource exploitation by 1 APP.

c) No memory protection.

Features of OS :

1. Resource Management.
2. Interface between user & computer hardware.
3. Access to the computer hardware.
4. Hides the underlying complexities of memory allocation & management.
5. Isolation & Protection to different apps.

Types of Operating System :-

Goals which need to be achieved.

1. Maximum utilization of CPU.
2. Process Starvation.
3. High Priority execution.

① Single Process O.S :- It does not jump to next job until it finishes first. All the goals are not fulfilled.

Ex:- MS DOS

② Batch Processing O.S :- Jobs are collected in batches by operators to run similar jobs in one batch. This was done by using punch cards or magnetic tape.

Ex:- ATLAS

③ Multi Programming O.S :- If jobs went for some I/O work, then CPU will take next job to work on for removing idleness and maximise CPU utilization.

J_1	J_2	J_3	J_4	J_5

Ready Queue
(Ready to be executed)

By context switching jobs are switch from J_1 to J_2. J_1 process is stored in a data structure PCB (Process control Block) and J_2 job is restored. Ex:- THE.

④ Multi Tasking O.S :-

→ High Priority Job possible.

→ single CPU.
→ context switching.
→ Time sharing.
→ No Process starvation.

Time sharing is there, lets say 1000ms to all the jobs one by one. This increase responiveness.

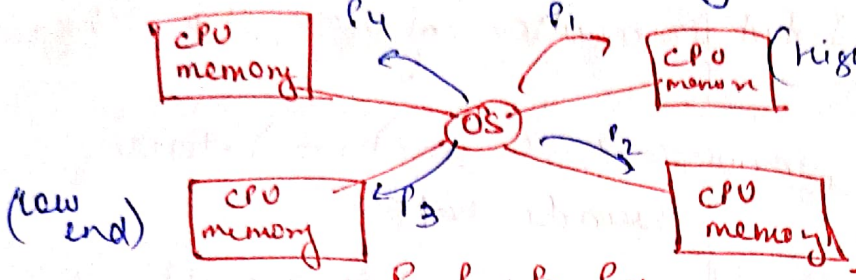
if (time > 1000 ms)
~~next job~~. context switching
next job.
time = 0
}.

Examples → CTSS

⑤ Multi processing O.S:- → Context switching → Time sharing
→ CPU. ≥ 1 (multiple).

→ More CPU utilization. Ex:- Windows.
→ Increased reliability (विश्वसनीयता) 1 CPU fails, other works.

⑥ Distributed O.S:- (Loosely coupled O.S):



→ May have many users.
→ Loosely coupled autonomous computer system.
→ One fail do not affect other.

Ex:- AIX, Solaris, Locus.

⑦ RTOS (Real Time O.S.):- → Error free
→ Real time (very fast)

Ex:- ATC (Air Traffic control), Industrial (Nuclear plant, defence, missiles).

Program :- An executable file containing certain set of instructions written to complete a specific job or operation on your computer.
→ Compiled code → Ready to be executed
→ Stored in Disk.

Process :- Program under execution. Resides in computer main memory (RAM)

Thread :- → An independent path of execution in a process.
→ Light weighted process (part of process)
→ Used to achieve parallelism by dividing process's task which are independent path of execution.

→ Example → Windows, Chrome Tabs, text editor (writing + spell check + formatting + save all are happening at same time cause each one is thread & working independently.

Multi Tasking	Multi Threading
① Execution of more than 1 task as process	① Execution of more than 1 thread
② Context switching b/w process is there.	② Context switching between threads happen.
③ Can work in 1 CPU	③ To achieve its benefit we need ≥1 CPU.
④ Isolation & memory protection (Cause different process needs isolated & different memory)	④ No isolation & memory protection. (Since being part of same process, they use same memory and resources.

Thread Context switching	Process Context Switching
① OS save current state of thread & switches to another thread of same process.	① OS save current state of process and switches to another by restoring its state.
② No switching of memory space (Cause of same process)	② Switching b/w memory space.
③ ∴ Fast switching	③ ∴ Slow switching
④ CPU's cache is preserved.	④ CPU cache is flushed.

Thread Scheduling:- Threads are schedule for execution based on their priority. Even though threads are executing within runtime, all threads are assigned process time slices by the operating system.

Components of OS

① **Kernel** :- Part of OS which directly interacts with hardware and performs crucial tasks.

a → Heart of OS / core component

→ Very first part to load on start-up.

② **User-space** → ~~Place~~ Part of OS where application software runs, apps don't have privileged-access to the underlying hardware. It interacts with kernel.

a) GUI (Graphical User Interface)

b) CLI (Command line interface)

Shell :- (Command interpreter) where user writes codes to perform certain task.

Function of Kernel :-

① **Process Management** :- a) scheduling process and threads on CPU.

b). Creating and deleting system and user process.

c) Suspending and resuming process.

d) Provide mechanism for process synchronization & process communication.

② **Memory Management** :- a) Allocating and deallocating memory space as per need.

b). Keeping track of which part of memory being used and by which process.

③ **File Management** :- ① Creating and deleting files and directories

② Mapping files in the storage (secondary)

③ Backup support into a stable storage media.

④ I/O mangement :- To management and control I/O operation and I/O devices.

→ Buffering, spooling, Caching.

Types of Kernels ① Monolithic.

Adv.
① All func. all in kernel itself.
② High performance as communica. is fast (less user mode, kernel mode overhead)
③ Eg - Linux, Unix, MS-DOS

Disadv.
① Bulky in size
② Memory required to run is high
③ Less stable, reliable → 1 module crashes - whole kernel is down,

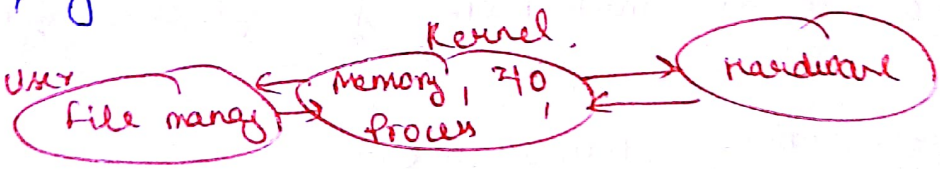
② Micro Kernel :

Adv.
① Major function in kernel.
① Memory & Procen Manag.
② Be smaller in size.
③ More reliable and stable.

Disadv
① Performance is slow.
② Overhead switching b/w user space and kernel mode.

Eg → L4 Linux, Symbian OS, MINIX, etc.

③ Hybrid Kernel.

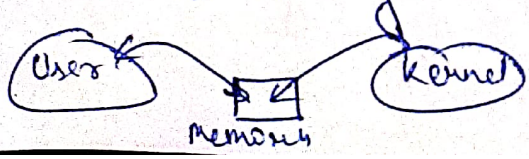


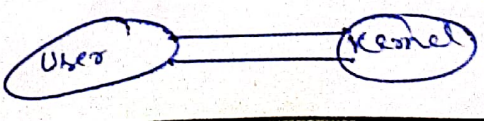
① Combined approach.
② speed & design of mono & stability & modularity of micro.

Eg:- macOS, Windows NT 7++,

③ IPC also happen but lesser over heads.

Communication b/w user mode & Kernels happens through IPC (Inter Procen Communication) in 2 ways

① Memory Sharing.


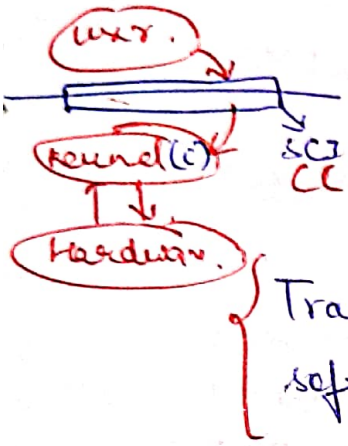
② Massage Passing


④ Exo Kernel → Works on end to end principle. It has fewest hardware abstraction as possible. It allocates physical resources to application.

⑤ Nano Kernel → Offers abstraction without system services like micro kernel ∴ becomes analogous.

System calls

How do apps interact → System Calls using.



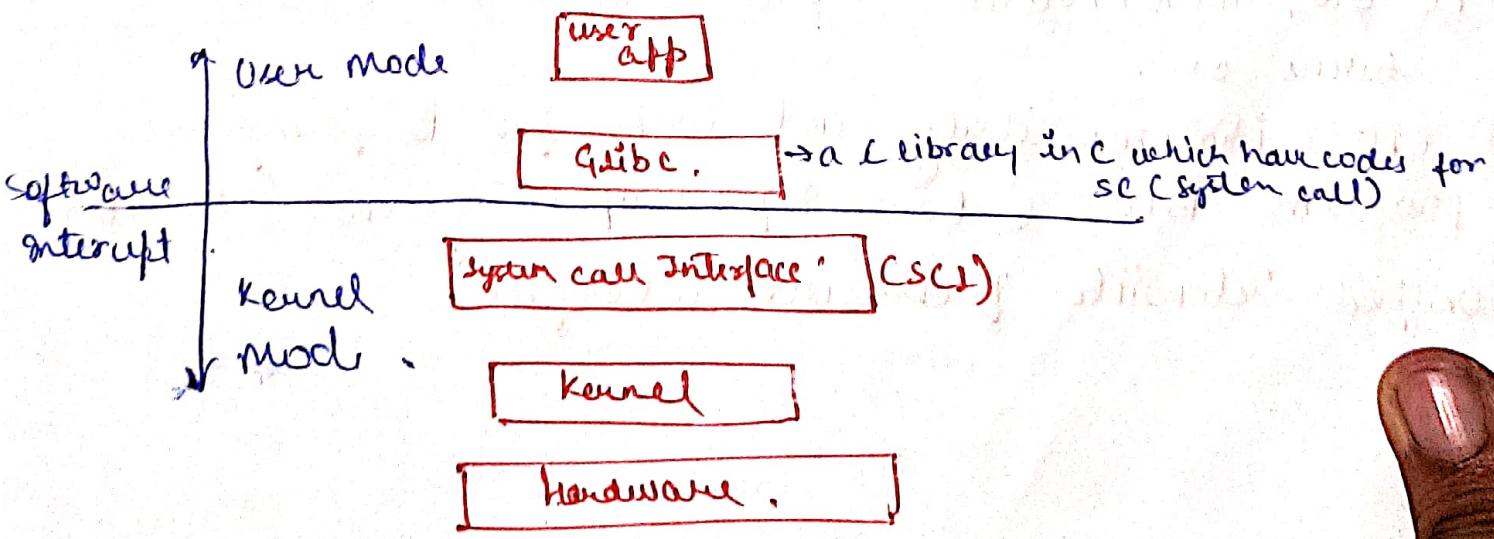
{ CLI commands are wrapper of actual system calls and these system calls then interact with kernel. }

{ Transitions from user (US) to KS is done by software interrupt.. }

→ System calls and kernel space implementation is written in C.

→ System calls :- System calls is a set of mechanism which ~~helps~~ ~~the~~ a US program can request using a call from the kernel. for which it does not have permision to perform

S.C. are the only way through which a process can go into kernel mode from user mode.



Category.	Windows.	Unix.
① Process control	CreateProcess() ExitProcess() WaitforsingleObject()	fork() exit() wait().
② File Management	Createfile() Readfile() Writefile() Closehandle() Setfilesecurity()	open() read() write(). close() chmod().
③ Device Management	SetConsolemode() Readconsole() Writeconsole()	ioctl() read() write()
④ Information Management.	GetCurrentProcessID() setTimer() Sleep()	getpid() alarm() sleep().
⑤ Communication.	CreatePipe() CreatefileMapping() MapViewOfFile()	pipe(). shmget() mmap()

What happens when you turn on your computer?

① PC ON, motherboard, harddisk or other devices external turns on.

② CPU Initializes itself and looks for the firmware program BIOS (Basic Input Output system) (UEFI → Unified Extensible firmware interface)

(3) BIOS loads its configurational settings.

(4) Bios looks for memory area backed up by CMOS (Complementary Metal Oxide semiconductor). Along with it CPU runs BIOS, which tests and initializes system hardware through POST (Power ON self test) process. means it tests for various components required for proper turning on. Ex:- RAM (a) It check if RAM is present if not error pops up.

(b) It check RAM is working or not, if not error pops up.

(5) BIOS handsoff responsibility to OS Boot loader for booting our PC up.

→ BIOS checks which boot device has OS.

→ BIOS looks for MBR or EFI. that loads the (Master Boot Record)

rest of OS known as BOOTLOADER. BIOS checks for program either in MBR or EFI (seperate drive for OS).

(6) Bootloader is a small program that has largest task of loading OS. → Windows bootloaders Windows Boot Manager (Bootmgr.exe)

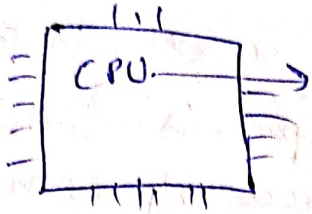
→ Macos → boot.efi

→ Linux → GRUB.

32-bit and 64-bit

→ 32 bit can access 2^{32} unique memory address. ($0 - 2^{32}-1$) (4GB) while 64-bit → 2^{64}. ($0-2^{64}-1$) (Billions GB very large)

→ 64 bit ~~bit~~ OS has 64-bit register while 32-bit has 32-bit registers.



contains billions of transistor & they combine to form register.

means physical size of register determines the type of OS needed in system. (First hardware then software)

→ 64 bit & 32-bit can process 64 bits & 32-bits data resp.

Advantages of 64-bit over 32-bit

① Addressable Memory :- 32-bit → 2^{32} unique address
64 bit → 2^{64} " "

② Resource usage :- 32-bit can use upto 4GB RAM at a time while 64 bit can access large amount of RAM.

③ Performance :- 32 bit can execute 4 byte of data in 1 instruction cycle while 64 bit can execute 8 byte.
~~1 instruction cycle~~ / sec → thousands to billion of instruction cycle run.

④ Compatibility :- 32-bit → 32-bit OS run while 64-bit can run 32-bit or 64-bit OS.

⑤ Better graphics performance ; 8 bytes graphic calculation make graphic intensive apps run faster.

Storages

① **Register** :- Smallest unit of storage.
→ Data passes through register before processing in CPU. It is used to quickly accept, store and transfer data & instructions that are being used immediately CPU.

② **Cache** :- Additional memory system that temporarily stores data which are frequently used for quicker processing

③ **Main memory** → RAM.

④ **Secondary storage** :-

Comparision factors : ① Cost ② Speed ③ Volatile ④ Storage size.

Process

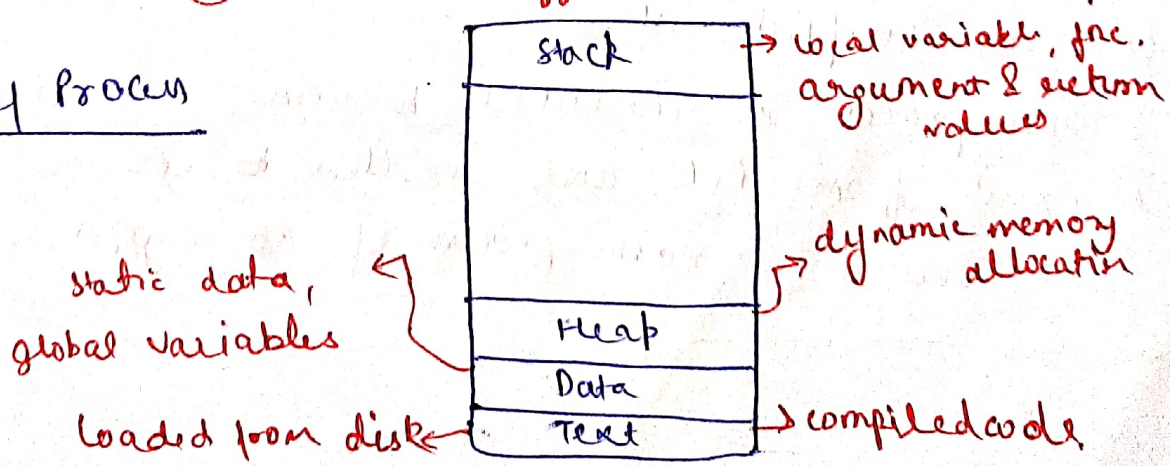
Program → compiled code ready to execute.

Process → Program under execution.

OS creates process - (By converting program into process)

Steps → ① load the program and static data into memory.
② Allocate runtime stack.
③ Heap ~~mem~~ memory allocation
④ I/O tasks ⑤ OS handsoff control to main()

Architecture of Process



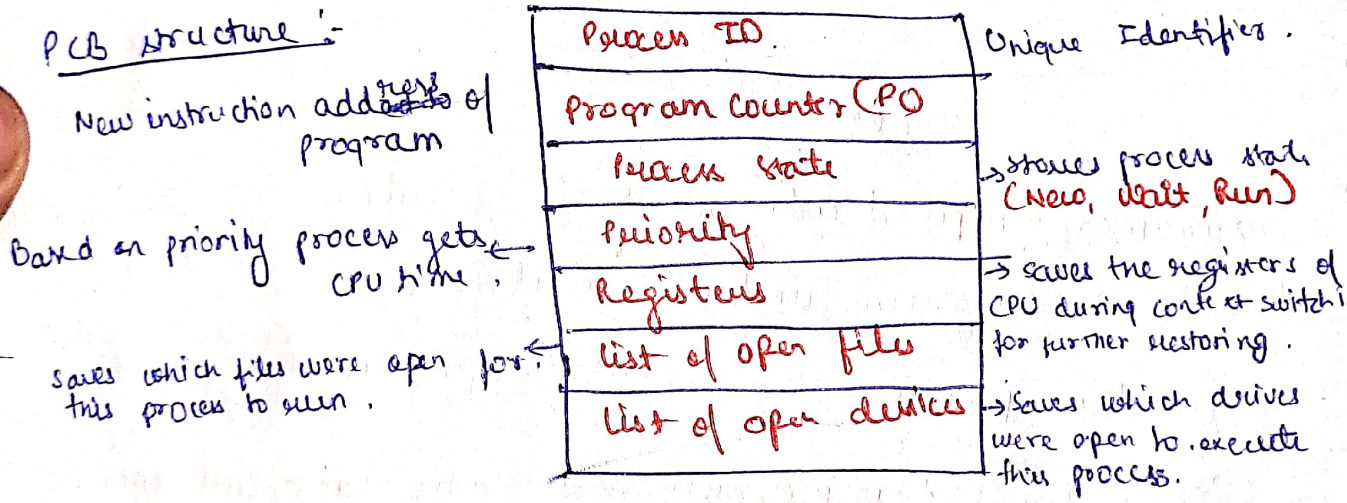
Insufficient memory → Error occurs when heap gets filled up and touches the stack.

Attributes of Process: a) It allows identifying the process uniquely.

b) Process Table (i) All process are tracked by OS. using a table like datastructure.
(ii) Each entry in the table is ~~PCB~~ PCB (Process Control Block).

c) PCB stores infor of process → It stores (PCB) process id, program counter, process ~~table~~ state, priority etc.

PCB structure :-



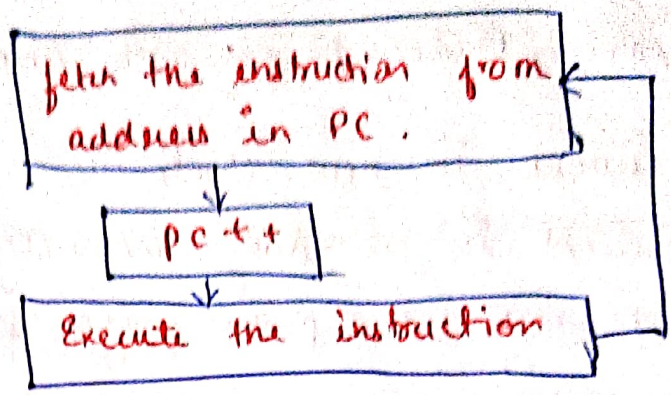
Registers in PCB :- is a data structure. When process is running and its time slice expires; the current value of process specific registers is get stored in PCB. and process can be swapped out.

→ When process is scheduled to run, the register value of is read from PCB and written to CPU registers.

This is the main procedure of CPU registers in PCB.

Program Counter :

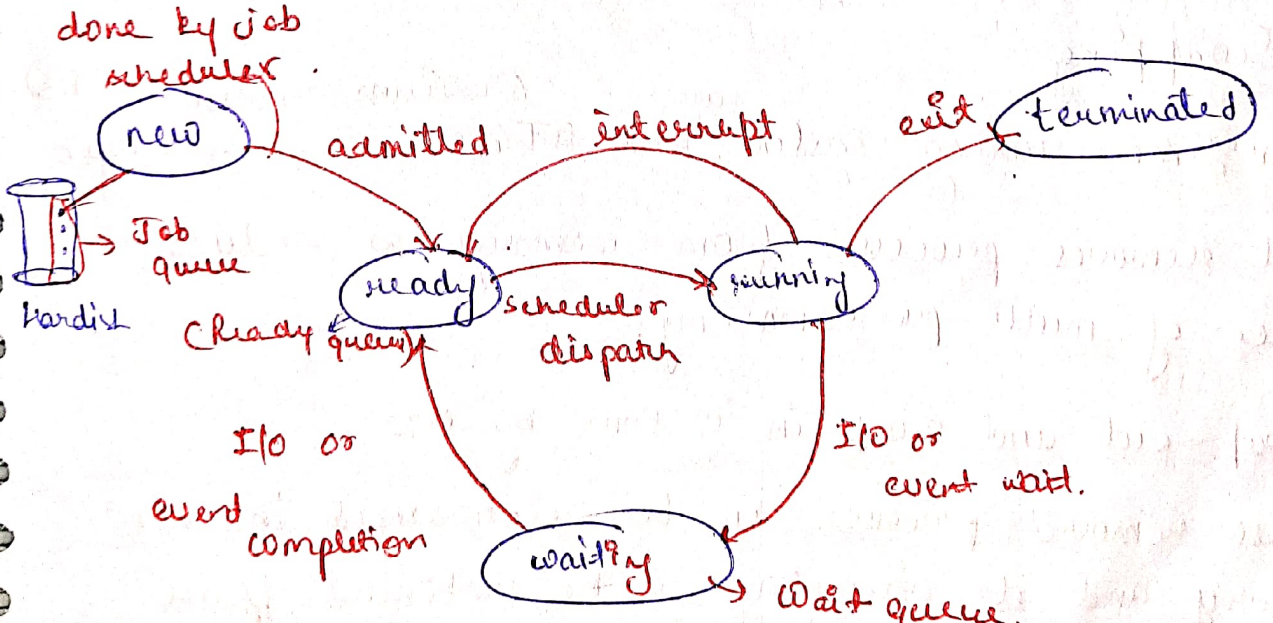
It runs untill program ends.



Process States | Process Queues

Process state : As process executes, it goes through various states in its life time.

1. **New :-** OS is about to pick program and convert it in process.
2. **Run :-** Instruction are being executed, CPU is allocated
3. **Ready :-** Process is in memory waiting to be assigned to the processor.
4. **Wait :-** Waiting for IO
5. **Terminated :-** Process has finished execution. PCB entry is removed from Process Table



① Process Queues → ① Job queue. { LTS → long Term scheduler, huge idle time }

① Process in new state
② Present in secondary memory.
③ Job scheduler picks process from pools & load it in memory (LTS)

② Ready Queue ① Process in ready state
① Present in main memory.
③ CPU scheduler (STS) picks the process from main memory (ready queue) according to priority and hands it to the CPU.

③ Waiting Queue:- ① Process in wait state.

④ Degree of multi-programming:- The no. of processes in the memory.
LTS controls the degree of multi-programming.

⑤ Dispatcher: The module of OS which hands control of CPU to a process selected by STS.

Swapping.

a) Time sharing system may have Medium Term (MTS) scheduler

b) It removes processes from memory to reduce degree of multi programming.

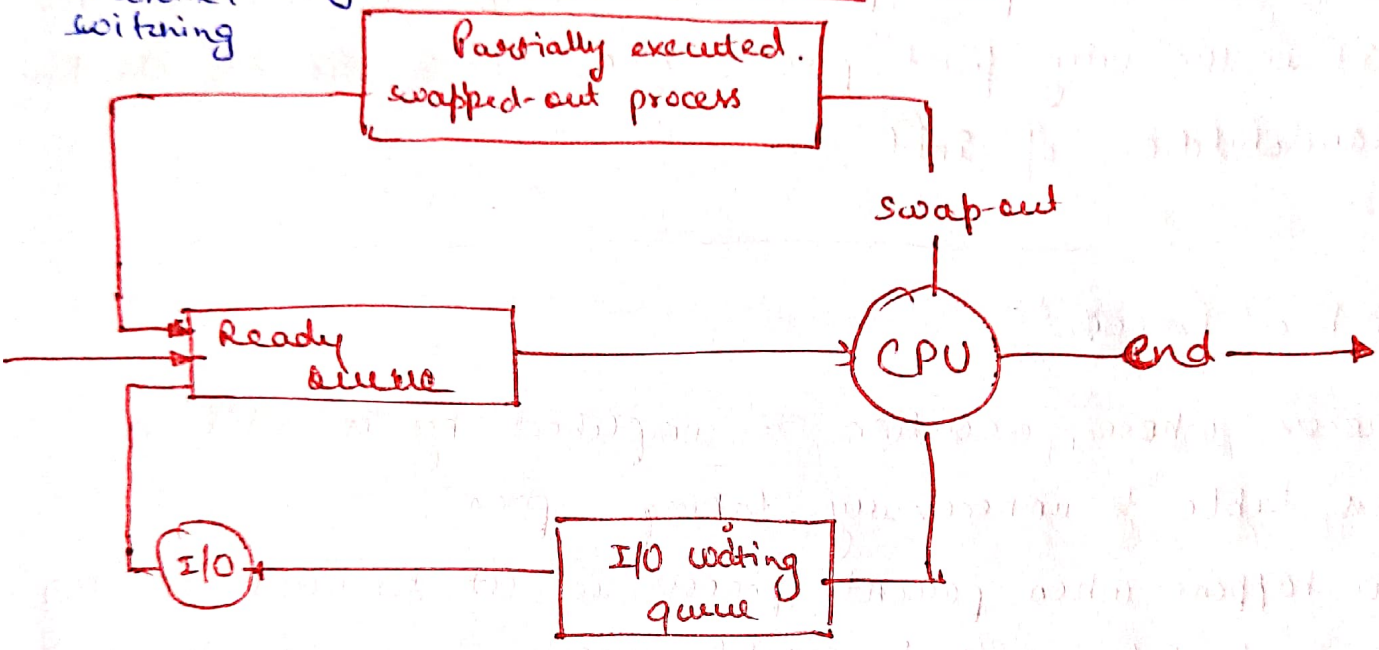
c) Swap-out and swap-in is done by MTS.

d) These removed processes can be reintroduced in the memory and its execution can be continued from where it left off is called Swapping.

e) It is necessary to remove load from the ready queue. ai

f) Swapping is mechanism, where a processes is temporarily swapped out from main memory and stored in secondary storage until it is again swapped-in in the main memory.

This process goes through CPU via context switching, so at that moment CPU is doing nothing useful to us, that's why ~~swapping~~ context switching is pure overhead.



Context-Switching :- ① When the CPU switches to another process by state save of current and state restore of different process, it is called context switching.

② When this occurs, the kernel copies the current data from CPU register to corresponding register in PCB and loads the data from another PCB register to CPU.

③ Since it is not doing anything helpful to user, it is a pure overhead.

④ Speed may vary from machine to machine due to depending on memory speed and the number of registers.

Orphan Process

a) A process whose parent is already terminated before child and now child became orphan is known as orphan process.

b) In this case orphan process is adopted by init process.

c) Init is the very first process of OS. Rest are the children, or grandchildren of init.

Zombie Process:

1) Process whose execution is completed by is still in process table & unnecessary taking space.

2) This happens when parent process is set to wait for long time and before that child process is executed.

3) Until parent process does not receive the exit status of child process, child process retains in process table, but it has already freed the resources to OS.

4) When the wait system call, the zombie process is eliminated from process Table. This is known as reaping of zombie process.

Intro to Process Scheduling

Process Scheduling :- ① Part of multi-programming OS.

② By switching process among CPU, we can make system more productive

③ Processes in the memory, dispatched goes to CPU and then again after time quantum dies or wait in I/O, it goes back to ready queue.

CPU Scheduler :- ① whenever CPU is idle, OS has to select a process and dispatch it to CPU. Done by STS.

2 types of Process Scheduling

① **Non-Preemptive Scheduling:** @Once process is dispatched to CPU, CPU keeps the process until the process is terminated or it goes for wait.

ⓑ. starvation increases. ⓒ Low CPU utilization.

② **Pre-Emptive Scheduling :-** ⓐ Once the process is dispatched to CPU, it keeps process until the process is terminated, or it switches to wait or time quantum dies.

ⓑ Starvation decreases ⓒ High CPU utilization

ⓓ High overhead than non-preemptive.

Goals of CPU scheduling :- ⓐ Maximum CPU utilization

ⓑ Low starvation ⓒ Minimum Turn-around Time.

ⓓ Minimum wait time ⓔ Min. response time.

ⓕ Max. throughput of system.

Intro to Process scheduling

Process Scheduling :- ① Basis of multi-programming OS.

② By switching process among CPU, we can make system more productive.

③ Processes in the memory, dispatched goes to CPU and then again after time quantum dies or wait is hit, it goes back to ready queue.

CPU Scheduler :- ① whenever CPU is idle, OS has to select a process and dispatch it to CPU. Done by STS.

Types of Process Scheduling

① **Non-Preemptive scheduling:** ⓐ once process is dispatched to CPU, CPU keeps the process until the process is terminated or it goes for wait.

ⓑ. starvation increases. ⓒ Low CPU utilization.

② **Pre-emptive Scheduling :-** ⓐ Once the process is dispatched to CPU, it keeps process until the process is terminated, or it switches to wait or time quantum dies.

ⓑ Starvation decreases ⓒ High CPU utilization

ⓓ High overhead than non-preemptive.

Goals of CPU scheduling :- ⓐ Maximum CPU utilization

ⓑ Low starvation ⓒ Minimum Turn-around Time.

ⓓ Minimum wait time ⓔ Min. response time.

ⓕ Max. throughput of system.

Throughput :- No. of process completed in 1 unit time

Arrival-Time (AT) :- Time when process first reaches to ready queue.

Burst-Time (BT) :- Time required by process for its execution.

Turnaround Time (TAT) :- Time taken from first time process enters ready state till it terminates (CT - AT)

Completion Time (CT) Time taken till process gets terminated.

Wait Time (WT) Time spends in wating for CPU (TAT - BT)

Response-Time :- Time gap when process first dispatched to CPU from when it arrived in ready queue.

Process Scheduling Algorithm

① FCFS (First-Come-First-Serve) :-

ⓐ Whichever process comes first in ready queue gets CPU first.

ⓑ In this, if one process has longer BT, it will have adverse affect on average WT of diff. process. It is known as Convoy Effect.

ⓒ Convoy effect is a situation in which those processes who needed short time for completion are blocked by processes who need large time to execute.

(i) This causes poor resource management.

② Shortest Job first (SJF) (Non-Preemptive)

ⓐ Process with the least BT will be dispatched first.

ⓑ Must do estimation of BT for each process and accuracy in doing so is not gaurenteed.

ⓒ Run lowest time process till it terminates and then dispatch next lowest one.

ⓓ Convoy effect is there if the very first process has large BT and has waited for too long.

ⓔ Process starvation might happen.

ⓕ Criteria for SJF Algo's AT+BT.

SJF (Preemptive) ⓐ Less starvation

ⓑ No convoy effect.

ⓒ Gives average WT less for given set of process as scheduling sort job before a long one decreases the WT of short jobs more than it increase WT of long process.

Priority scheduling (Non-Preemptive)

ⓐ Priority is assigned to a process when it is created.

ⓑ SJF is general case of priority scheduling with priority inversely proportion to BT.

② Priority Scheduling (Preemptive)

ⓐ Current state job will be preempted if the next job has higher priority.

ⓑ May cause indefinite waiting for low priority jobs (extreme starvation)

Possibility they may never execute (Preemptive and Non Preemp)

Solution: ⓐ Ageing is the solution.

ⓑ Gradually increase priority of process that wait so long.
Ex: ↑ Priority by 1 in 15 minutes.

Round Robin Scheduling: ⓐ Most popular.

ⓑ like FCFS (but preemptive) ⓒ Designed for time sharing system.

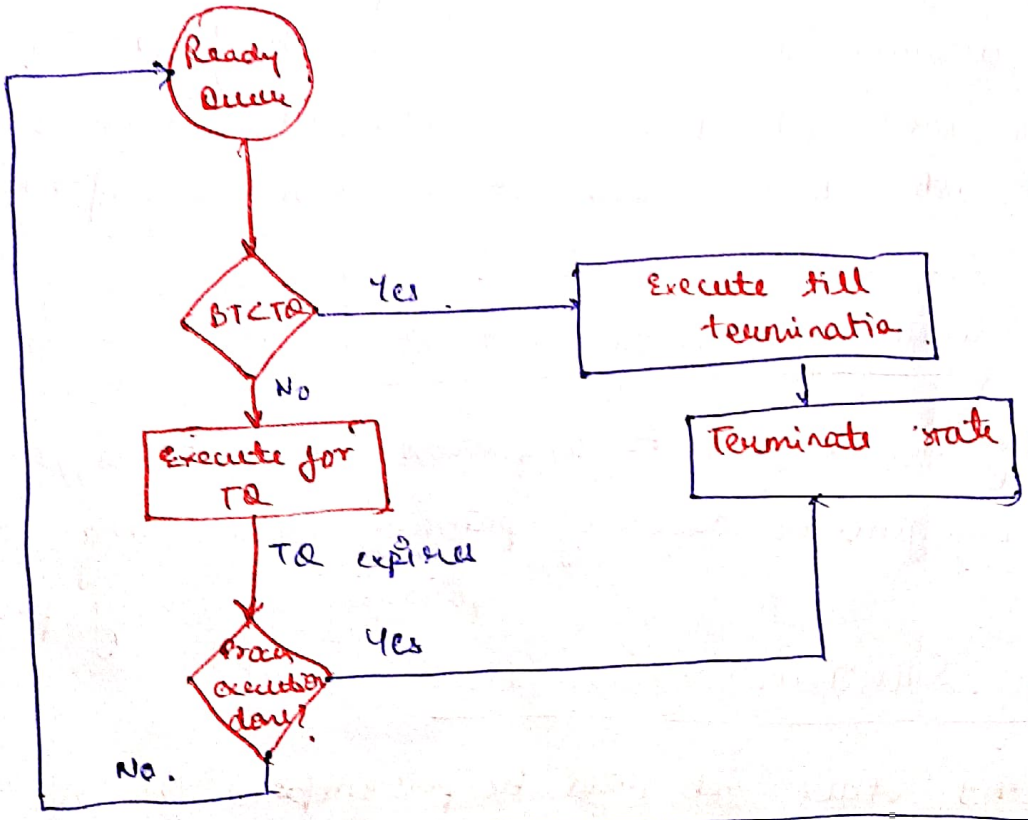
ⓓ Criteria: AT + TQ (but doesn't depend on BT)

ⓔ No process is going to wait forever so minimal starvation.

ⓕ easy to implement.

ⓖ Overhead is high overhead $\propto \frac{1}{TQ}$; overhead → context switching

---



Process scheduling between different queues

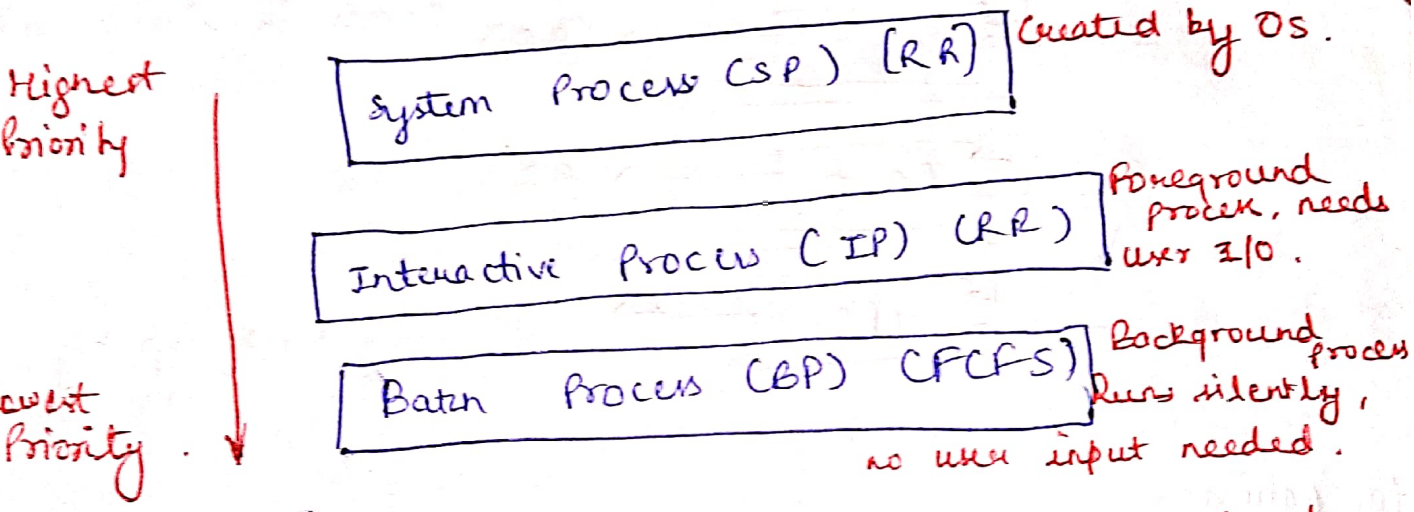
① MLQ (Multi-level queue scheduling).

(a) Ready queue is divided into multiple queues depending on the priority.

(b) A process is permanently assigned to one queue based on the kind of process, size, complexity, memory.

(c) Each queue has its own scheduling algorithm.

SP → RR, IP → RR, BP → FCFS.



→ scheduling among these queue is implemented as fixed priority preemptive.

→ High priority queue has absolute priority over lower priority queue.

→ This cause huge starvation, as preemption is also there

→ Convoy effect is present.

② Multi-level feedback Queue scheduling :-

(a) Multiple sub queues are present.

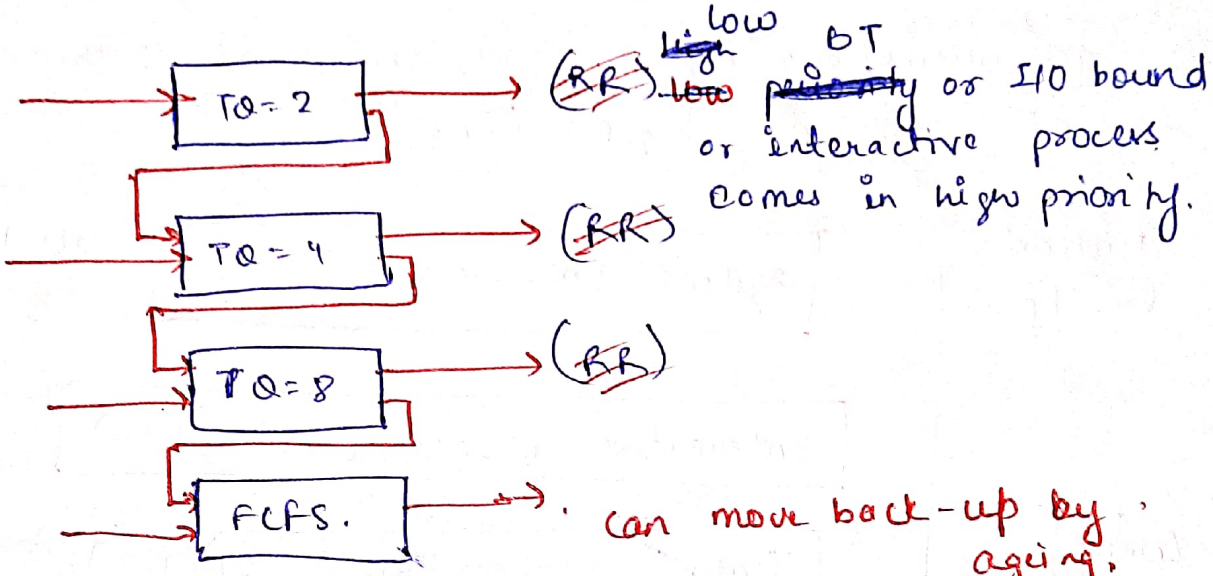
(b) Allows the process to move b/w queues. Separate processes on basis of BT. Long BT process goes to lower priority, but I/O bound & interactive process has higher priority.

In addition, the process waiting for long time in lower priority may be moved to higher priority queue. This form of ageing prevent starvation.

© less starvation than MLQ.

(d) It is flexible.

(e) Can be configured to match specific system designs.



Comparision.

	FCFS	SJF	PSJF	Priority	P-Priority	RR	MLQ	MLFQ
Design	Simple	Complex	Complex	Complex	Complex	Simple	Complex	Complex
Preemption	No	No	Yes	NO	Yes	Yes	Yes	Yes
Convoy effect	Yes	Yes	NO	Yes	Yes	No	Yes	Yes
Overhead	No	No	Yes.	NO	Yes	Yes	Yes	Yes

Concurrency

Ability of OS to execute multiple instruction sequences at a same time. It happen in OS when there are several process threads running in parallel.

Thread, Thread scheduling, Thread context switching is discussed before.

How each thread get access to CPU?

① Each thread has its own program counter.

② Depending on thread scheduling algo, OS schedule it.

③ OS will fetch instruction in PC corresponding to that thread and execute it.

I/O or TQ based context switching is done here as well.

→ We have TCB (Thread Control Block) like PCB for store management during context switching.

Single CPU will be gain benfit of single CPU as they have to context switch in same CPU, so no parallelism.

Benefits of Multithreading

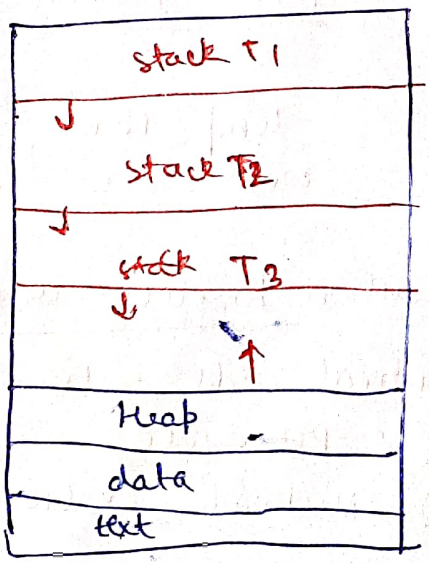
① Responsiveness

② Resource sharing (within same Process)

③ Economy :- more economical to create and ~~sw~~ context switch.

Thread space is divided within same process as ~~create~~ allocating new memory & resource is highly uneconomical

④ Threads allow utilization of multicore processor to greater scale & efficiency.



A process → 3 threads

Critical section problem

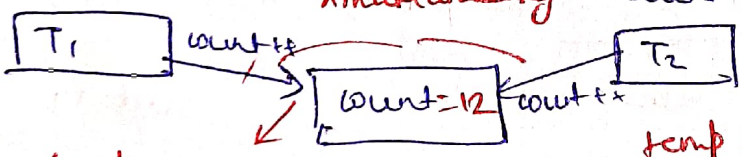
Process synchronization techniques plays a key role in maintaining the consistency of shared data.

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

Major Thread scheduling issue

a) **Race condition :-** A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Since thread scheduling algo. can swap between threads at any time, we dont know the order in which the threads will attempt to access data. ∴ the result of change is dependent on thread scheduling algo. ie. both ~~are~~ threads are "racing" to access/change the data.

Eg:- count++ is done by temp = count + 1 } inside computer/memory
count = temp



temp = 12 + 1
count = 13

temp = 12 + 1
count = 13

∴ Instead of 14 ans = 13.

Solution to race condition :- ⓐ **Atomic operations** make critical code section an ~~op~~ atomic operation. ie. executed in 1 CPU cycle.

ⓑ Mutual exclusion using locks.

ⓒ semaphores.

Can we use simple flag variable to solve the problem of race condition :- No.

Solution of C.S. should have 3 condition

① Mutual exclusion :- 1 thread at a time in C.S.

② Progress :- so special preference at any thread in going to C.S. (Jisko kaam aaye wo jaye)

③ Bonded waiting :- limit waiting time for a thread.

① Solution - 1 simple flag turn

```
T1
while (1) {
    while (turn != 0);
        C.S.
    turn = 1;
        R.S
}
```

```
T2
while (1) {
    while (turn != 1);
        C.S.
    turn = 0;
        R.S
}
```

Mutual exclusion achieved, but no progress. Order is fixed, depending on value of turn.

② Solution - 2 Peterson's solution :- can be used to avoid race condition but holds good for only 2 threads.

```
T1
while (1)
{
    flag[0] = T
    turn = 1
    while (turn == 1 and flag[1] == T);
        C.S.
    flag[0] = f;
}
```

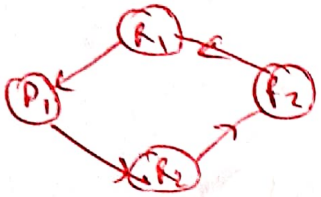
```
T2
while (1)
{
    flag[1] = T
    turn = 0
    while (turn == 0 and flag[0] == T);
        C.S.
    flag[1] = F
}
```

Mutex / locks :-

a) Locks can be used to implement mutual exclusion and avoid race condition by allowing only 1 thread/process to access shared — critical section.

Disadvantages :- ① Contention (heated disagreements) :- one thread has acquired lock and other thread will be kept waiting. what if that thread had dies being acquire lock, then all other threads will be in infinite waiting.

② Deadlocks. ③ Debugging ④ Starvating of high priority threads.



P_1, P_2 → threads

$R_1 R_2$ → resources.

P_1 has R_1 and need R_2 to complete process.

P_2 " R_2 " " R_1 to " " .

∴ System is ranged and deadlock is created.

Conditional variable and Smaphores for Thread Synchronization

i) Conditional variable :- a) The conditional variable is an synchronization primitive that lets the thread wait until a certain condition occurs.

b). Works with a lock.

c). Thread can enter a wait state only when it has acquired a lock. when thread enters the wait state, it immediately releases the lock until another thread notifies. ~~for~~ ~~for~~ ~~did~~ Then thread enters critical section, and immediately acquires lock and start executing.

c) Why to use conditional variable?

(i) To avoid busy waiting.

c) Contention is not here.

Semaphores: a) Synchronization method.

b) An integer eqn = no. of resources.

c) Multiple threads can go and execute C.S. concurrently.

d) Allow the multiple program threads to access the finite instance of resources. whereas mutex allows multiple threads to access the single shared resource at a time.

e) Binary semaphore a.k.a mutex locks. (0 or 1)

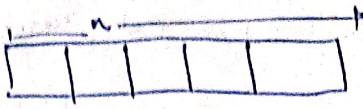
f) Counting semaphore:- (i) Can range over an unrestricted domain.

(ii) can be used to control access to a given resource consisting of a finite no. of instances.

g) To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations. when the process execute the wait operation and finds that semaphore variable is not positive, it must wait. however rather than engaging in busy waiting, the process can block itself. The block-operation places a process into a waiting queue associated with the semaphore and the state of the operation is switched to sleep state. Then control is transferred to the CPU scheduler, which selects another process to execute.

h) A process that is blocked waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from waiting state to ready ~~state~~ queue.

Producer - Consumer Problem



Buffer space (shared memory for producer & consumer thread)

Problem :- ① sync b/w producer & consumer.

② Producer must not insert data when buffer is full

③ Consumer must not pick " " " " empty.

Solution :- (semaphores) ① m, mutex :- Binary semaphore used to acquire lock on buffer

② empty → a counting semaphore, initial value n.

③ fill → tracks filled slots, intial value = 0.

Producer

```
do{ wait(empty) // wait until
        empty>0 then, empty=value--;
wait(mutex);
// C-S. add data to buffer.
signal(mutex);
signal(full); //increment full->value++;
} while(1)
```

Consumer

```
do{ wait(full) // wait until full>0,
        then full--;
wait(mutex);
// remove data from buffer.
signal(mutex);
signal(empty); // increment empty++
} while(1)
```

Mutex :- It is a locking mechanism used for synchronization access of resource. Only 1 task can acquire the mutex, means there is ownership associated with the mutex, and only the owner can release the lock.

Signaling mechanism :- Semaphore is a signaling mechanism, which signals the other thread when C.S. is empty or job is executed & completed by one thread

Reader-

① Reader

② Writer

③ if >1

④ if >1

comes

Semaphore

① mutex

→ No. 2

② Wait → To ent

Writer's

Reader's

Reader- Writer Problem

(1) Reader thread → Read
(2) Writer thread → write (update).
(3) if >1 readers are reader → no issue.
(4) if >1 writer. or 1 writer & some other thread (R/W), runs parallelly. → Race condition occurs or data inconsistency

Semaphores :-

(1) mutex :- ~~Binary semaphore~~ → To ensure ⊥ mutual exclusion when rc is updated
→ No. 2 threads can modify rc at a same time.

(2) wrt → To ensure no. reader & writer thread approaches C-S at a same time.

~~(1)~~ (3) read count (rc) → integer intial = 0, tracks how many readers are reading in C.S.

writer's solution :-

```
do {  wait (wrt);
   // do some write operation.
      signal (wrt);
   } while (true);
```

Reader's solution :-

```
do { wait (mutex); // for read count variable
     rc++;
     if (rc == 1)
        wait (wrt); // ensures no writer can enter if there is even 1 reader.
     signal (mutex);
  // C.S. Reader is reading
     wait (mutex);
     rc--; // a reader leaves.
     if (rc == 0) // no reader is left in C.S.
        signal (wrt) // writer can enter
     signal mutex); // reader leaves.
  } while (true);
```

Swap left and Right Pages

Deadlock Part - 1

, Multiple ~~resources~~ process requesting finite number of resources.

→ When the process request a resource (R) and when it is not available process goes to the waiting state, and some times. that resource is busy forever, this state is known as Deadlock.

③ DL is a bug in process | thread synchronization.

④ In DL, process never finish executing, and system resources are tied up preventing other processes to start.

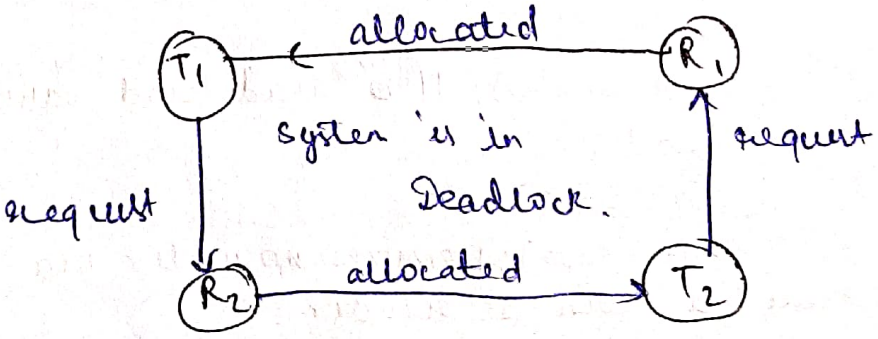
Example of resources → memory space, CPU cycles, files, locks, sockets, I/O devices, etc.

⑤ Single resources can have multiple instances (4-core CPU) [R ...]

How process | thread utilize resource?

① Request :- Request the R, if R is free then lock it.

② Use :- ③ Release :- Release resource instance to make it available to other processes.

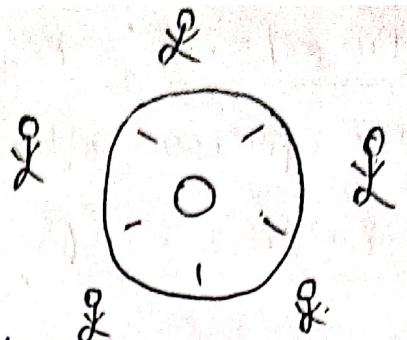


4 Necessary condition for deadlock to happen.

① Mutual exclusion :- Only 1 process can use the resource at a time and other processes requesting for it must wait until.

② Hold & wait :- Processes is holding one resource and waiting for other to complete over all execution,

Dining Problem (Philosphers)



→ 5 philospher (think or eat)
→ eat → acquire 2 forks.
→ think → do nothing.

Problem:- 5 ph. and 1 ph needs 2 forks to eat.
in worst condition we need 5×2 = 10 forks but we have only 5 forks.

Solution: (using semaphores)

a). Each fork is binary semaphore.
b) A ph. calls wait() operation to acquire a fork.
c) Release fork by calling signal()
d) Semaphore fork [5] ↓ 1 ↑.

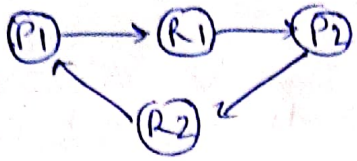
Although semaphore solution makes sure, no 2 neighbours are eating simultaneously but it could still create Deadlock.

Methods to avoid Deadlock:-

a) ~~Allo~~ Allow at most 4 ph. to be sitting simultaneously
b) Allow the ph. to pick up his fork only if both forks are available and he must do it in critical section (atomically)
c) Odd even rule:-
an odd ph. ~~first~~ picks up his left fork and then his right fork, whereas an even ph. picks up his right fork then his left fork.

∴ Only semaphore can't alone solve this problem. We must need some other methods like above to solve.

Non-Preemption :- Resource should be voluntarily released by the process after completion of execution.

Cycle-Detection:- 

cycle should be present.

Methods for handling deadlock.

(a) Use a protocol to prevent or avoid deadlock, ensuring system will never enter into it.

(b) Allow the system to enter deadlock state, detect it and recover it.

(c) Ignore the problem altogather and pretend the deadlock never happened in system. (Ostrich Algo) or (Deadlock Ignorance)

To ensure deadlock never occurs system can use either deadlock prevention or deadlock avoidance scheme.

Deadlock prevention :- by ensuring at least one of the necessary condition cannot holds.

(a) Mutual exclusion:- Use locks for non-shareable resources.

(1) Shareable resources like Read-Only files can be accessed by multiple processes.

(2) We cannot prevent DL wholly cause there are some non-shareable process.

(b) Hold and Wait :- Whenever a process request a resource it must not have resource beforehand.

(1) Protocal (A) Allocate all resource to a process before its execution.

(2) Protocal (B) allow request only when it has no resources. means process must free all resources before requesting.

(c) Non-preemptive :- If the process is holding a resource and requesting for another one and it took some.

significant time in waiting then process should preempt its current resources and again request for it all well as new one.

② If the process with some higher priority is waiting for some resources. then the resource should be preempt. by the process which itself is in waiting.

Circular wait :- ① To ensure, this condition never holds is to ensure proper ordering of resource allocation.

② (P_1 and P_2) can only move to (R_2) when it has (R_1) allocated.

Dead lock avoidance :- Kernel is given advance info. about.

→ Resources allocated to the processes (total no.)

→ No. of resources needed by process to complete execution

→ Total no. of resource available.

In this we allocate resources in such a way that deadlock is avoided at any moment.

Safe state :- A safe state is that in which system can allocate resource to each process and still avoided DL. There has to be some safe sequence for system to be in safe state.

Unsafe state :- O.S. cannot prevent processes from requesting resource which may lead to deadlock. Unsafe state may lead to deadlock.

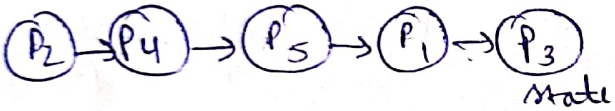
→ Request should only be allowed to make when the resulting process is safe.

→ Algorithm used here is Banker's Algo.

If a process request the set of resources, then the system must decide whether it will lead the safe state, If yes, then only it fulfill the processes request.

P	allocated A B C	max need A B C	available A B C	Remaining need A B C
P_1	0 1 0	7 5 3	3 3 2	7 4 3 (4)
P_2	2 0 0	3 2 2	5 3 2	1 2 2 (1)
P_3	3 0 2	9 0 2		6 0 0 (5)
P_4	2 1 1	4 2 2	7 4 3	2 1 1 (2)
P_5	0 0 2	5 3 3	7 4 5	5 3 3 (3)
	7 2 5		7 5 5	

A → 10 B → 5 C → 7

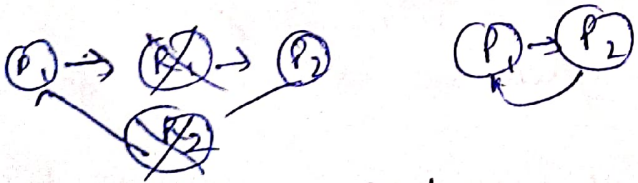


state

If P_3 needed. 8 process of A then ~~system~~ will be unsafe. and dead lock can occur.

Deadlock Detection :- If O.S. has'nt implemented deadlock prevention or avoidance, then we must proceed for deadlock detection.

① For single instance resource → from RAG (Resource Allocated Graph) to ~~WFG~~ (wait-for graph). Remove the ~~res~~ resources and then check for cycle.



cycle present. DL → detected

" not " " not ".

② For multiple instance → Bankers algo, will tell ~~DL~~ cycle is presented. If it reaches to unsafe state cycle may be present.

Recovery from Deadlock:-

(a) Process Termination :- (1) abort all process.

(2) abort 1 process at a time until DL is eliminated.

(b) Resource preemption :- To eliminate DL, we successively preempt some resources from processes and give these resources to other processes until DL cycle is broken.

Memory Management Techniques

Keeping processes in a main memory (ready queue) in such a way, we can keep more number of processes in an efficient manner.

Logical Address space :- (1) Address generated by CPU which user can access.

(2) Logical address does not exist physically ∴ virtual address space and user access actual address via logical address.

(3) The set of all the logical addresses generated by a program is called as logical Address space.

(4) Range → 0 to max.

Physical Address :- (1) Addresses loaded into the memory address registers of the loaded memory.

(2) user can never get access of physical memory.

(3) The set of all the physical address corresponding to the virtual addresses is known as physical address space.

(4) Computed by Memory Management Unit (MMU).

(5) Range → (R + 0) to (R + max)

R → Base address

max → offset of the process.

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

How OS manages the isolation & protection?
(Memory Mapping & Protection)

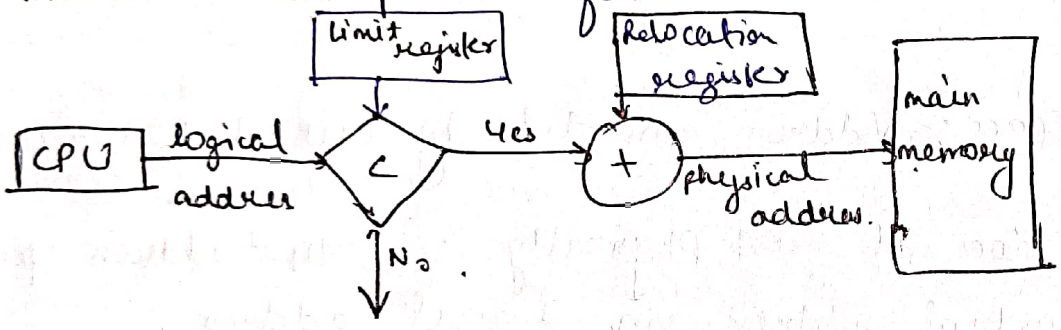
① OS provide virtual space concept.

② The relocation register contain ① base address ② limit register has range of logical addresses.

③ logical address must be less than limit address.

④ Relocating the values before program execution is also a part of context switching.

③ Any attempt for physical address will trap in OS and treat the attempt as fatal error.



Allocation Method ① Contiguous Allocation
② Non-Contiguous allocation.

① Contiguous Allocation (fixed partition): main memory is divide either in equal partitions or unequal with different sizes. and process are allocated in each partition (empty).

Limitation :- ① Internal fragmentation
② External fragmentation.

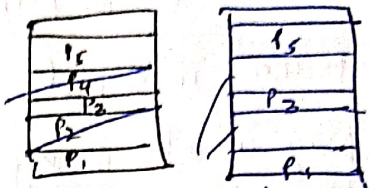
③. Limitation on the process size. (P_1 > 4KB and partition is of 3KB.).

④ Low degree of multi programming :- No. of process at a time in a memory.

Dynamic Partitioning :- the partition size is not declared initially. It is declared at the time of process loading.

Advantages over fixed partioning : ① No internal fragmentation
② No limit on size of process.
③ High degree of multi-programming.

Limitation :- External fragmentation.



External fragmentation

Free space Management

Defragmentation / Compaction :- ① Dynamic partitioning suffers from external fragmentation.

② Compaction to minimize the probability of external fragmentation.

③ The free partitions are merged which can be allocated according to the needs of new process. This process is called Defragmentation.

④ The efficiency of the system is decreased in this case since all the free space is transferred to a single place. This takes lot of CPU cycles. thus overhead.

How free space is stored / represented in OS?

It is represented by a free list (Linked-list D.S.) containing ~~add~~ starting address and length of free space.

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

Various algo are there to find out the suitable hole in free list and allocate it.

① First fit :- ⓐ Allocate the first hole that is big enough.
ⓑ Simple and easy to implement

① Fast, less time complexity.

① Next Fit:- @ Enhancement of first fit, but starts search from the last hole allocated.
① Same advantages as first fit.

② Best Fit:- @ Allocate smallest hole that is big enough.
① Lower internal fragmentation.
② May create many small holes and, cause major external fragmentation.
③ Slow, as required to iterate whole free list.

④ Worst fit:- @ Allocate largest hole, big enough.
① Lower external. Slow, as required to iterate whole free list.
② leaves larger holes that may allocate other processes.

Paging / Non-Contiguous Memory Allocation

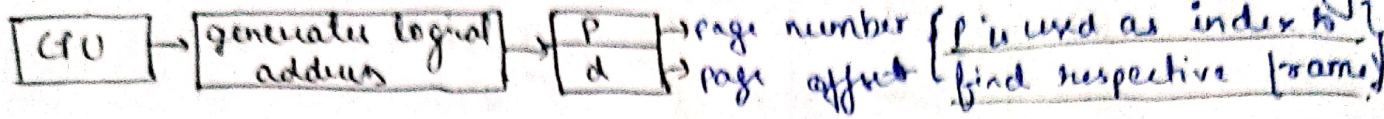
Main demerit of dynamic partitional is external fragmentation. This can be removed by compaction but overhead.
We need more dynamic / flexible / optimal mechanism to load process in partitions.

Idea behind Paging:- a) Not able to allocate process due to external fragmentation. Even though memory is available.

Paging:-
① Paging is a memory managment scheme that allows physical address allocation to be non contiguous.
② Avoid external fragmentation and need of compaction.
③ Dividing physical memory into fixed-parts called frames and logical memory space in fixed partition called pages. (# Page size = Frame size).
④ Page size is determined by processor architecture. Traditionally it was of fixed size of 4096 bytes.

(5) Page Table: (a) D.S. storing which page is mapped to which frame

(b) A page table contain base address of each page in the physical memory.



(c) Page Table is stored in main memory at the time of process creation. whose base address is stored in PCB.

(d) PTBR (Page Table Base Register):- holds the base address for the page table of the current process. It is a processor register that is managed by operating system.

How paging avoids external fragmentation? Non contiguous allocation of pages are allowed at random free space in main memory.

Why paging is slow and how to make it fast?

Too many memory references to reach desired address.

Translations Look-Aside Buffer (TLB).

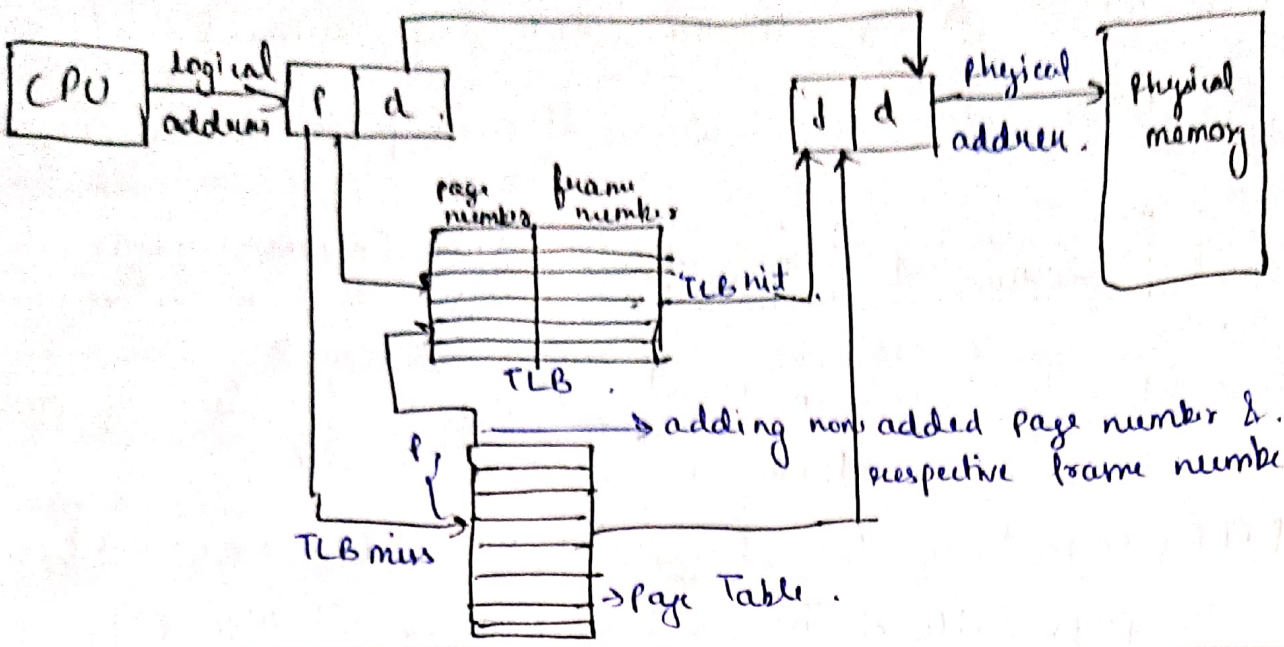
(a) Hardware support to speed-up paging process.

(b) Hardware cache, high speed memory

(c) Key and value pairs, independent hardware.

(d) After retreiving physical address from page table we store it in TLB for later approach, which makes it fast.

(e) And each process has its own (ASID) (Address space Identifier) in TLB. to avoid every time deleting and loading new process references.



Segmentation | Non Contiguous Memory Allocation.

An important aspect of MMU that becomes unavoidable with paging is seperation of user's view of memory from actual physical memory.

~~This~~ Segmentation is memory management technique which supports user's view of memory.

⇒ Rather than fixed sized pages, process is divided into variable size segments based on user view of logical memory.

segment → number → segment number. {s, d}.
offset → offset

⇒ O.S. doesn't care of user's views. ∴ a function can be divided in 2pages and stored in different frames. Due to which, executing ~~the~~ one function takes much time than needed, thats why segment came in picture. containing function of same type, ~~to~~ for faster evaluation.

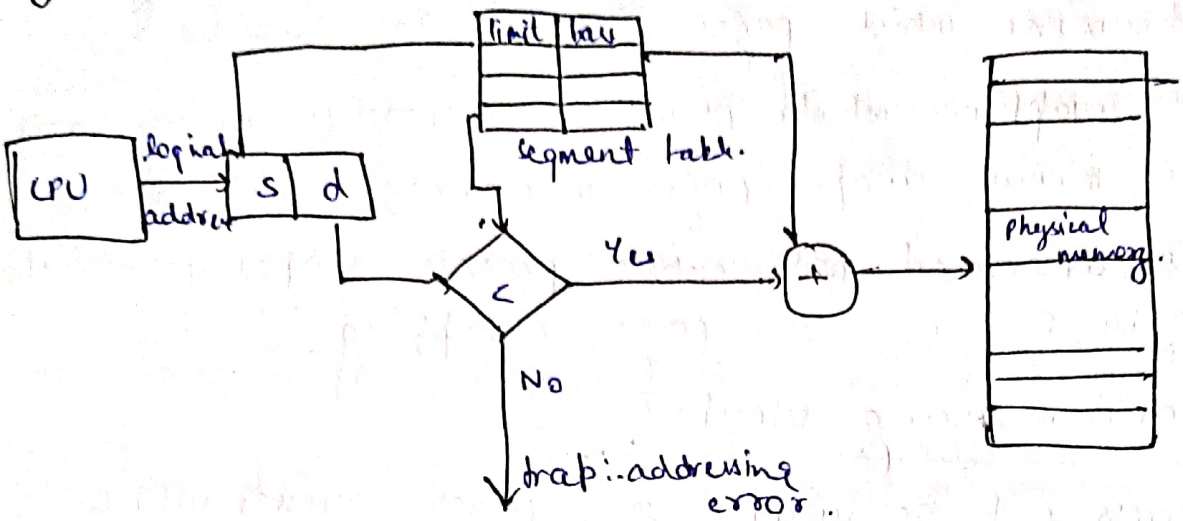
Advantages: ① No internal fragmentation unlike Paging.
② One segment has contiguous allocation, hence efficient working within segment
③ Size of segment table is generally less than page table
④ more efficient., cause compiler keeps same type function at

one segment.

Disadvantages: ① external fragmentation (MTS)

② Different size of segment is not good at time of swapping

Modern system architecture provides both segmentation & paging in some hybrid approach.



What is Virtual Memory? | Demand Paging | Page Faults

Virtual Memory: is a technique that allows the execution of processes that are not completely in the memory. It provides an illusion of having a very big main memory. This is done by treating a part of secondary storage (swap space) as a part of main memory.

Programs larger than main memory can also run.

What can we achieve by this?

a) A program is no longer be constrained by the amount of physical memory that is available.

b) : each user program will take less memory, more programs can run at a time max. utilization of CPU.

c) Running a program that is not entirely in memory would benefit both system and user.

→ Programmer is provided very large virtual memory when physical memory is very less.

Demand Paging: a popular method of VMM, where the pages of process in least demand are stored in secondary storage.

→ A page is copied to the main memory where its demand is made, on page fault occurs. Page replacement algo are then used to determine which pages are to be replaced.

→ Instead of swapping whole process in memory, we use Lazy swapper. which never swaps page in memory until needed.

→ Swapper is concerned with entire process swapping while Pager is " " page swapping.

How Demand Paging Works?

① when process is to be swapped in, pager guesses which pages are to be used

② Pager only swaps those pages which are needed by CPU for execution, and rest stays.

③ The valid-Invalid bit scheme is used to distinguish b/w pages that are in memory and that are on the disk.

(i) Valid-Invalid bit 1 means, associated page is legal & in memory.

(ii) " " " 0 " a " " is not in main memory may be valid or invalid.

④ If the process never attempts for invalid pages, whole process may execute without need of paging.

⑤ If the process is not in memory and process asked for it then page fault occurs. and a trap to the OS is send.

Procedure to handle page fault:

① Check an internal table (in PCB of process) to determine whether the reference was valid or an invalid memory access.

② If ref is invalid through exception, otherwise. pager will swap in the page.

③ We find a free frame (from free frame list).

④ Schedule a disk operation to read the desired page into the newly allocated frame.

⑤ When disk read is complete, we modify the page table, valid-invalid bit as 1.

⑥ Restart the instruction that was interrupted by trap. The process can now access the page as it was always in memory.

Pure Demand Paging. In extreme case, we start the process without a single page in memory and when os ask for it process immediately faults for page and page is brought in the memory.

Never bring a page in memory until required.

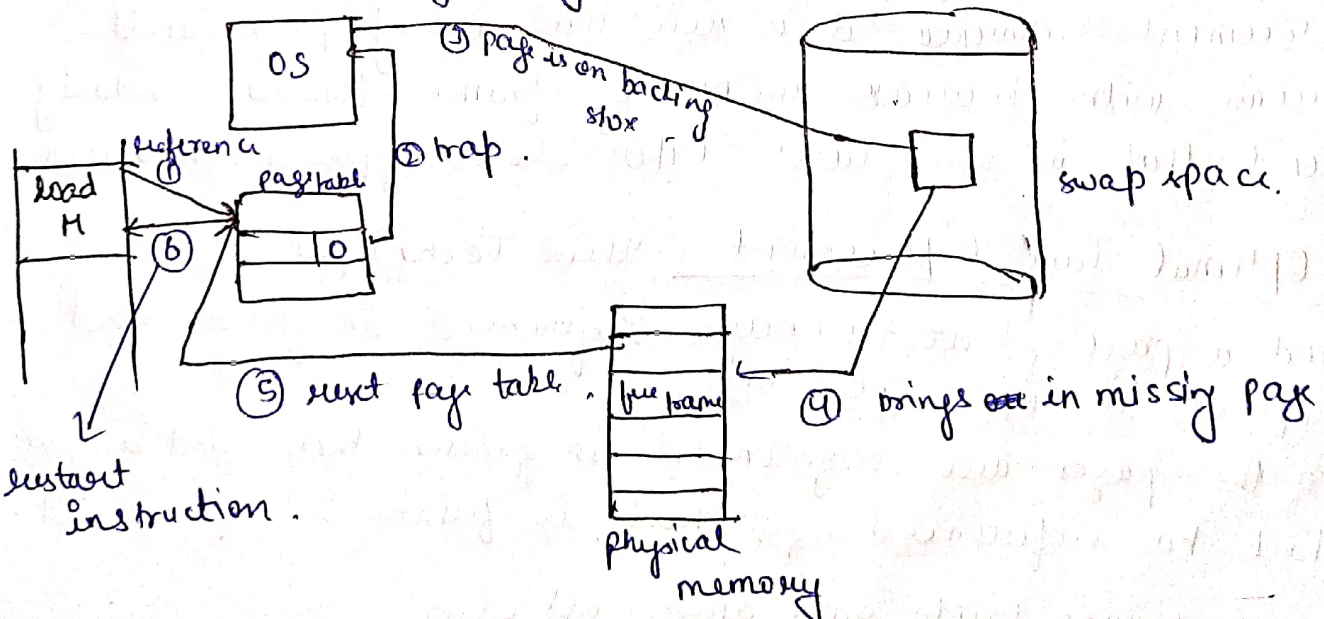
We use locality of reference to bring out reasonable performance from demand paging.

Advantages of Virtual memory → The degree of multi-programming will be increased.

User can run large apps with less physical memory.

Disadvantages :- ① System will become slower as swapping takes time.

② Thrashing may occur.



Page Replacement Algorithm

① Whenever a page fault occurs, means OS is accessing for page which is not present in memory, so OS has to bring it from virtual memory.

② In case physical memory has no free frame to put our asked page, then a page replacement algorithm decides which memory page is to be replaced. Some allocated pages are swapped out and new pages are swapped into freed frame.

Types of Page Replacement Algo. (Aim to have min Page fault)

① FIFO (First in first out) :- Allocate frame to the page as it comes ~~oldest~~ in memory by replacing the oldest.

② Easy to implement.

③ Performance is not always good.

(i) Page replaced by initialization module that was used long time ago. (Good Replacement candidate)

(ii) Page may contain a heavily used variable that was initialized early and is in constant use (will again cause Page fault).

Belady's anamoly :- In case of LRU and optimal page replacement technique, it is seen that no. of page fault decrease with increase in no. of frame. However, belady found that in some cases FIFO shows opposite behaviour.

② Optimal Page Replacement (Ideal Technique)

ⓐ Find a page that is never referenced in future and replace it with new one.

ⓑ If the pages are referenced in future, then find a page which has referenced farthest in future & replace it.

ⓒ Lowest page fault rate among all algo.

⑤ Difficult to implement as OS requires future knowledge of reference string, which is kind of impossible (similar to SJF scheduling).

② Least-Recently-Used (LRU)

① We can use recent past as the approximation of near future. then we replace new page with page not being used in longest period.

② Can be implemented in 2 ways.

ⓐ Counters

(i) Associate time field with each page table entry.
(ii) Replace the page with smallest time value.

ⓑ Stacks (i) Keep a stack of page number.
(ii) Whenever the page is referenced, it is removed from stack and put on top.
(iii) By the way most recently used are always on top, and least recently used on bottom
(iv) Since we have to remove from the middle of stack, we can use doubly linked list.

④ Counting based page replacement :- Keep a counter of no. of reference made to each page.

(i) Least frequently used :- (a) Actively used pages should have large reference count.

(b) Replace page with smallest count.

(ii) Most frequently used :- @ least count page may have recently brought and yet to be used.

Neither LFU nor MFU are common.

LRU cache Code implementation.

```
struct dl {  // doubley linked list)
 pair<int,int> data;
 struct dl* next;
struct dl* prev;
  dl( int key, int val) {
  data.first = key ; data.second = val ;
   next = NULL ; prev = NULL };
};
 int n ;  //size of list or capacity.
map<int, dl*> mp ;
struct dl *head = NULL; struct dl *tail = NULL;

LRUcache (int capacity) {
    n = capacity;
   head -> next = tail ; }

int get (int key) {.
  if (mp.find (key) == mp.end ) return -1;
  dl * temp = mp[key];
     temp -> prev -> next = temp -> next ;  } removing node from
     temp -> next -> prev = temp -> prev ;  } original position.

     head -> nxt = temp -> next ;
     temp -> next -> prev = temp ;  } adding to new position
     head -> next = temp ;          } closet to head.
      temp -> prev = head ;
     mp[key] = temp;
  return temp -> data.second ;
 }
```

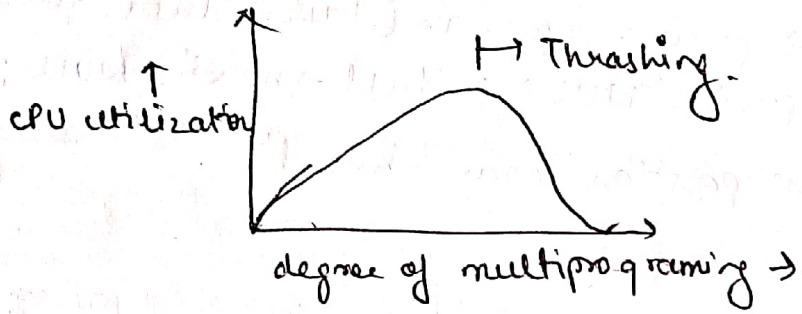
```
void put (int key, int val ){
 // if node is already present.
 if (mp.find(key) != mp.end() ) {
   dl * temp = mp[key];
→ removing from old position
→ adding to new position near head :
    mp[key] = temp;   temp → data → second = value;
 }.
else if ( mp.size() == n )
 {. dl * temp = new dl ( {key, value} );
   tail = tail → prev ; mp.erase(tail → data.first);
   tail → next → prev = NULL; tail → next = NULL;
→ Adding to new position near head.
   mp..mp[key] = temp;
 }
 else {
    dl * temp = new dl ( {key, value} );
 → adding to new position near head.
    mp[key] = temp;
 }.
}.
```

Thrashing

→ If the process doesn't have no. of frames it needs to support pages in active use, it will quickly page fault. At this point, it must replace some other page. However, since all its pages are in active use we replace a page that will be needed right away. Consequently, it quickly faults again & again, replacing pages that it must bring back immediately.

b) This high paging activity is called as Thrashing.

c) A system is thrashing when it spends more time servicing the page faults than executing process



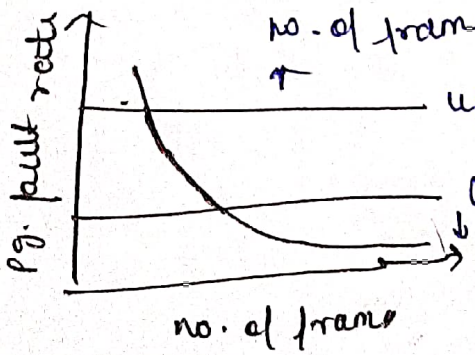
Techniques to Handle Thrashing :-

ⓐ **Working set Model** :- (i) It is based on principle of locality.

(ii) It simply say if we allocate enough frames to a process to accomodate its current locality, it will only fault when it moves new locality. But if allocated pages are less, system is bound to thrash.

ⓑ **Page fault frequency** :- (i) Thrashing has high page-fault rate and we want to control it.

(ii) If pg fault exceeds upper limit, allocate process another frame, if pg-rate falls below the ~~upper limit~~ lower limit, remove a frame from the process.



Disk Scheduling Algorithm and Disk scheduling

Disk scheduling is done by operating system to schedule I/O request arriving for a disk. It is also known as I/O scheduling.

Seek Time:- It is the time taken to locate the disk arm to a specified track where data is to be read or written.

Rotational Latency:- It is the time taken by the desired sector of the disk to rotate into positions so that it can access read/write heads.

Transfer Time:- Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and the no. of bytes to be transferred.

Disk Access Time:- Seek Time + Rotational Latency + Transfer time.

Disk Response Time:- It is the average time spent by the request waiting to perform its I/O operations. Average Response Time is the response time of all requests.

Disk scheduling Algo

① FCFS:- FCFS is the simplest of the disk scheduling algo. In FCFS, the request are addressed in a order they arrive in the disk.

② SSTF:- In Shortest seek time first, requests having shortest seek time are executed first. So seek time of every request is calculated in advance in the queue and then they are scheduled according to the calculated seek time. As a result, request near disk arm is scheduled first.

(3.) SCAN :- In the SCAN algo, disk arm moves into a particular direction and services the requests coming into its path and after reaching the end of the disk, it reverses its direction and again services the request arriving in its path. As, this algo works like a elevator, thus also called elevator algo.

(4) CSLAN :- Circular scan is same as scan, only difference is it do not service any request while returning from the other end.

(5) LOOK :- It is also like SCAN algo ~~despite~~ except for the difference that the disk arm despite going to the disk end, goes only to the last request to be serviced in front of the head and then reverse its direction from there only. Thus, it prevents extra delay which occured due to unnecessary traversal to the end of the disk.

(6) CLOOK :- (Circular look) same as look, only difference is it does not service any request in the returning path, from the other end.