OS

Date / /
Page

* Textbooks:
1. Operating system Concepts – 8th edition by → Silberschatz, Galvin, Gagne
2. Modern Operating System – 3rd edition by → Andrew S Tanenbaum

* Why to Study OS:-
1. Carrer → Subject + GATE (10 Mrks)
2. To understand how computer works.
3. To learn how to write concurrent code
4. Resource Management
5. Experience of Dealing with Big Slw.
6. Everything together : Lang + Hlwt DS + Algo + money + art + luck + etc...
7. Computer System Design.

* "Fail early, Fail Fast & Learn how to make things work."

* Syllabus:-

Unit 1: Introduction to OS, OS structure & strategies, Process Thread, Scheduling algorithms,

Unit 2: Memory mngt, how to allocate memory, how to replace page in main memory, Virtual Mem,.
File System: File concept, directory, implementations.

Unit 3: Process Synchronization & Deadlocks.

Unit 4: Device Mangt: Disk, RAID, IO system

Unit 5: Case Studies: Linux & Windows XP.

Prerequisites: 1. DS, 2. Computer Organisation & Architecture
3. Programming Languages.

* OS: 1. OS is an interface blw user & hardware
2. OS is a prg that manages the computer Hlw.

(More Precisely)

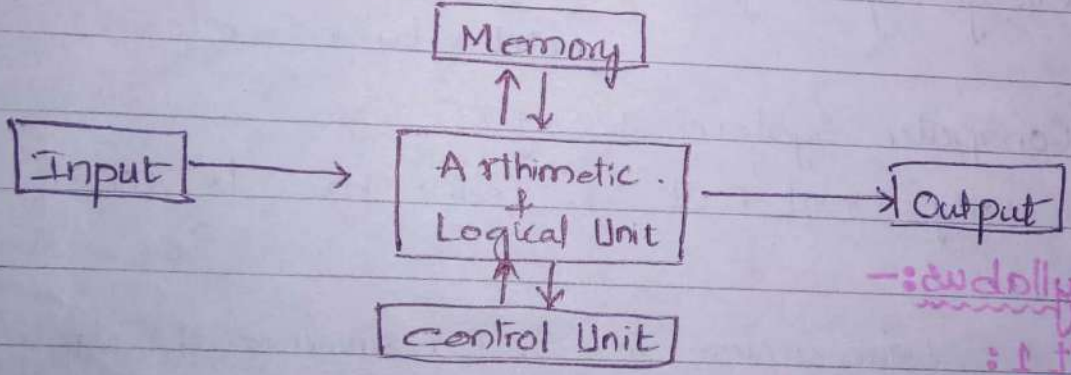
Set of prgs that manages the computer Resources.

1. (or) It provides a basic for application prgs and acts as an intermediary blw the computer user & Hlw.

* Goals: 1. To make the computer convinient to use
Primary: Laptops, Desktops (Simple to use)
Secondary: (Super Comp., Mainframes.
2. To use the computer hlw in an efficient manner

* Hardware: 1. CPU = control Unit + ALU
2. Memory — [Main Memory, Secondary Memory]
3. I/O devices.

* Computer Hardware:



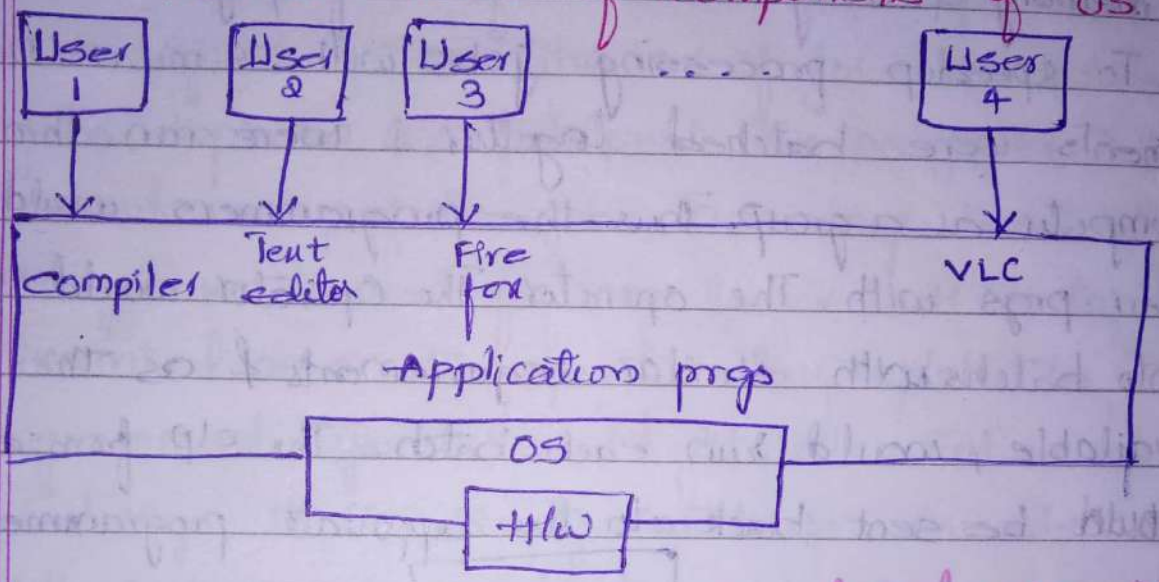
* The application prgs such as compilers, games, firefox, MS-office, Libre office, etc defines the way in which these resources (Hlw) are used to solve the computing prblms of users.

3. Resource Manager: A computer has many resources (Hlw & Slw) that may required to solve a prblm: CPU time, memory space, strage space, I/O devices, & so on OS acts as the manager of these resources & allocates

bookkeeping (how is required, how is available...)

them to specific prgs & users as necessary for tasks.

Abstract view of components of OS



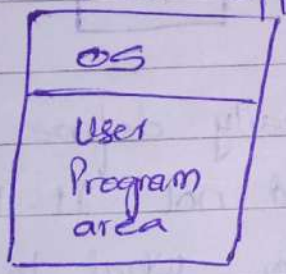
* We can't clearly define what shld be included in OS & what shld not. It basically depends on the OS developer/vendor what to include & what not to.
eg: Ubuntu includes Libre office, where as MS-Windo does not include Ms office.

* Types of Operating Systems: OS's are there from the very 1st computer generation. OS's keep evolving over the period of time. Following are the few of the OS's which are most commonly used.

1) Simple Batch Systems: The users of batch OS ~~did not~~ can't interact directly with the computer systems. Rather, the user prepared a job, which consists of the program, data & some control information about the nature of the job & submitted it to the computer operator. At some later time (perhaps mins, hrs, or days) the o/p will be appeared.

This consisted of result of prg as well as a dump of memory & registers in case of prg error.
To speedup processing jobs with similar needs/requirements were batched together & were ran through the computer as a group. Thus the programmers would leave their prgs with the operator. The Operator would sort prgs into batchs with similar requirements & as the comp. is available, would run each batch. The o/p from each job would be sent back to the appropriate programmer.

Memory layout:



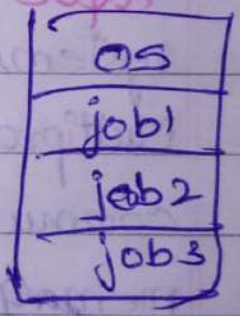
Disadvantages:
1. Lack of interaction b/w the user & the job.
2. CPU is often idle, because the speed of mechanical I/o devices is slower than CPU. (I/o Interrupts)
3. Difficult to provide desired priority.

2) Multiprogrammed batch Systems: All the jobs submitted by users will be placed in a job pool. A pool of jobs on disk allows the OS to select which job to run next. When jobs come in directly on cards or even on magetic tape, it is not possible to run jobs in a diff order. Jobs must be run sequentially on FCFS basis. However, when several jobs are on a direct access device (disk), job Scheduling becomes possible. The important aspect of job Scheduling

is the ability to multiprogram.

The idea is as follows: The OS keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The OS picks & begins to execute one of the jobs in memory. Eventually the job may have to wait for some task, such as tape to be mounted or an I/O operation to complete. In a non-multiprogrammed sys CPU would sit idle. But in a multiprogrammed sys the OS simply switches to execute another job. When that job needs to wait, the CPU is switched to another job & so on. Eventually the 1st job finishes waiting & gets the CPU back. As long as there is always some job to execute, the CPU will never be idle. This increases the performance of computer.

Memory Layout:



3) Time Sharing or Multitasking Systems: Time sharing is a logical extension of multiprogramming. It is a technique which enables many ppl, located at various terminals, to use a particular comp. sys. at same time. Processor's time which is shared among multiple users simultaneously is termed as time sharing. The main diff b/w MPS & MTS is that in case of MPS's objective is to maximize processor use, whereas in MTS's objective is to minimize response time.

+Multiple jobs are executed by the CPU by switching blw them, but the switches occurs so frequently that the user can receive an immediate response.

4. Personal computer systems: As hlw costs decreased, it has become feasible to have a computer dedicated to a single user. These types of systems are usually referred to be personal computers. PC OS's were neither multitasking nor multiuser. The goals of these OS's have changed with time. Instead of maximizing CPU & peripheral utilization, the systems opt for maximizing user convinience & responsiveness.

Eg: MS windows, Apple Macintosh, MS-DOS

5) Multiprocessor Systems (OR) Parallel systems / Tightly Coupled Systems: Most systems to date are single processor systems; that is they have only one main CPU. Multiprocessor sys's have more than 1 processor in close communication sharing the computer bus, clock & sometimes memory & peripheral devices.

Adv: 1) Enhanced performance.

2) Increased Reliability: If funs can be distributed properly among several processors, then the failure of 1 processor will not halt the system.

3) Several tasks can be executed concurrently or 1 task can be divided into several sub-tasks & executed concurrently.

6) Distributed System: A recent trend in computer sys's

is to distribute computation among several processors. In contrast to the highly coupled Sys's, the processors donot share memory or clock. Instead each processor has its own local memory. The processors communicate with one another through various communication lines. Such as high speed buses (or) Telephone lines. These systems are usally referred to as loosely coupled systems or distributed ~~comp~~ Sys.

Adv: 1) Resource Sharing: A user at one site may be able to seeluse the resources available at another.

2) Computation Speedup: Compilation can be partitioned into a no. of subcomputations & run concurrently. In addition if a particular site is overloaded with jobs, some of them may be moved to other lightly loaded sites → load Sharing

3) Reliability is Increased: If one fails, others can operate

4) Communication: Since they are connected, they can exchange the data.

7) Real-Time Systems: It is defined as a data processing system in which the time interval required to process & respond to ilps is so small that it controls the environment. The time taken by the System to respond to an ilp & display required updated information is termed as response time. So in this method response time is very less as compared to the processing.

Real time Operating Sys's are used when there are rigid

time requirements on the operation of processor or the flow of data. They have well-defined, fixed time constraints. Otherwise the sys will fail. For eg: Scientific experiments, medical imaging syss, weapon sys's, robots, etc..

There are 2 types of RTOS:

i) Hard RTOS: They gurantee that critical tasks complete on time.

ii) Soft RTOS: They are less restrictive. Critical real-time tasks gets priority over other tasks & retain the priority until it completes.

8) Handheld Systems: Include PDA's (Personal Digital Assistants), such as palm & pocket PC's, & cellular telephones, many of which use special purpose embedded OS's. Developing of handheld sys's & applications may face many challenges, most of which are due to the limited size of such devices, common characteristics are: i) Slower processing speed, ii) Less I/O devices can be connected, iii) Less battery power

OS Structure

* OS components: We can create a sys as large & complex as an OS only by partitioning it into small pieces.

i) Process Management: A process is a program in execution. Management of processes includes i) Process Scheduling, ii) creation/termination, iii) Block/Unblock/Suspend/Resume iv) Synchronization, v) communication, vi) Deadlock Handling vii) Debugging.

2) Main Memory Mngt: i) Allocation / De-allocation for processes, Ilo, Files, ii) Maintainance of several processes at a time, iii) Keep track of who's using what memory, iv) Movement of processes mem. to/from secondary storage.

3) File Mngt: A file is a collection of related information i) File creation & deletion, ii) Directory creation & deletion, iii) Mapping files onto secondary storage, iv) Files backup

4) Ilo Mngt: It consists of i) Mem. mngt component including buffering, caching, & Spooling. ii) General device driver-interface, iii) Drivers for speeific blw devices

5) Secondary storage Mngt: Since main mem. is to small to accommodate all data & prgs, & its data are lost when power is lost, the computer sys must provide secondary storage to backup main memory.
The OS is responsible for: i) Free space mngt, ii) Storage Allocation & iii) Disk scheduling.

6) Networking: OS is responsible for: i) Communication blw distributed processors. ii) Getting inf abt files, processes etc on a remote machine. iii) Can use either a message passing or shared memory model, etc

7) Protection System: If a computer sys has multiple users & allows concurrent execution of multiple processors, then various processes must be protected from 1 another's activities. Protection refers to a mechanism for controlling the access of prgs, processes or users to

the resources defined by a computer sys. This mechanism must provide a means for specification of the controls to be imposed, together with means of enforcement.

8) Command-Interpreter sys: It is the interface blw user & OS. Many commands are given to the OS by control statements. When a new job is started in a batch sys, or when a user logs into a time shared sys, a prg that reach & interprets control statements is executed automatically. This prg is called as cmd-line interpreter & is often known as Shell. It's fun is quite simple, get the next cmd & execute it.

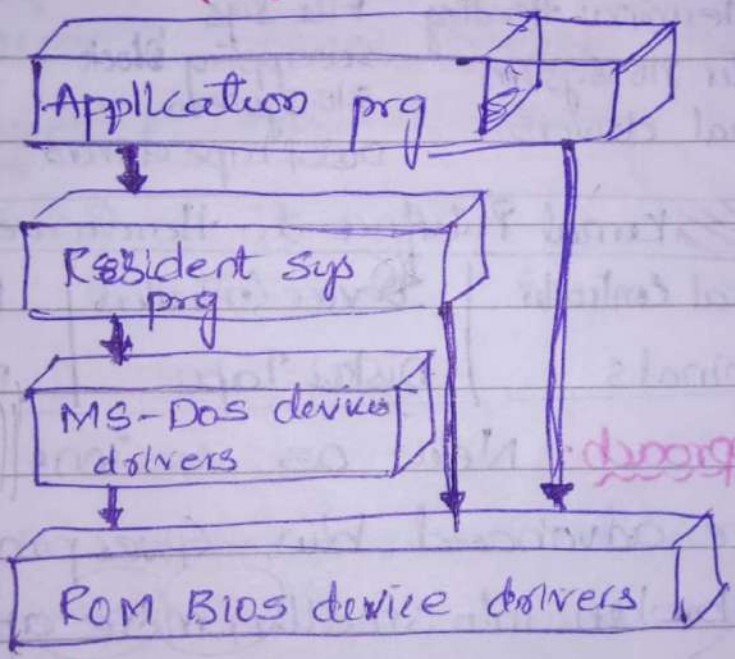
OS Structure

* OS must be engineered carefully if it is to function properly & to be modified easily. A common approach is to partition the task into small components, rather than having 1 monolithic system.
* A monolithic kernel is an OS architecture where entire OS is working in kernel space & is alone in supervior mode. A set of primitive or sys calls implement all OS services such as process mngt, concurrency, Mem Mngt, Device drivers can be added to the kernel as Modules.
* The structure defines how the OS components are interconnected & melded into kernel.

) Simple Structure: (Due to Hw limits/simple Hw).
MS-DOS (Microsoft-Disk OS). In MS-DOS, the interface &

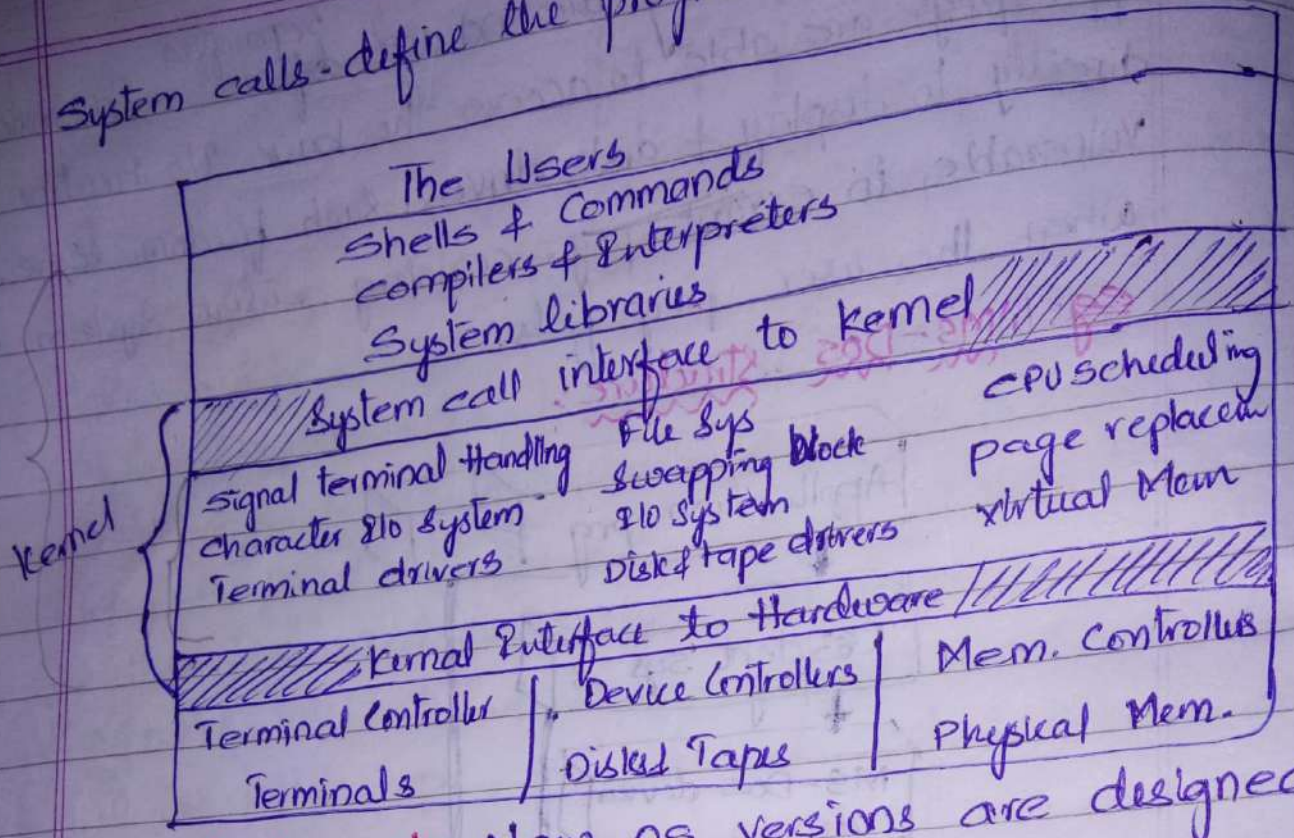
levels of functionality are not well separated. For instance, appli. prgs are able to access the basic I/O routine to write directly to display & disk drives. Such freedom leave, MS-DOS vulnerable to errant (malicious) prg. causing entire system crashes when the user prg fails.

eg: (i) MS-DOS Structure:



UNIX Structure: It consists of 2 separable parts: The Kernel & The System prgs. The kernel is further separated into a series of interfaces & device drivers, which have been added & expanded over the yrs as UNIX has evolved. Everything below the systemcall interface & above the physical hlw is the kernel. The kernel provides file sys, CPU scheduling, Mem. Mngt & other os fns through system calls. Taken in sum, that is enormous amt of funtionality to be combined into 1 level. Sys prgs use the kernel supported system calls to provide useful fns, such as compilation & file manipulation.

System calls - define the programmers interface to UNIX



2) Layered Approach: New os versions are designed to use more advanced hw. Gives proper hw support, os may be broken into smaller, more appropriate pieces than those allowed by the original MS-DOS or UNIX. The OS Then can retain much greater control over the computer & the applications that make use of that computer.
The modularization of the Sys can be done in many ways: The most appealing is the layered approach which consists of breaking the OS into a no of layers (levels) each built on top of lower levels. The bottom layer (level 0) is the H/w; the highest layer (level N) is the user interface.
The main adv of the layered approach is Modularity The layers are selected such that each uses funs &

services of only lower level layers.
The layers approach was used in "THE OS". It was defined in Six layers.

layer 5: User Programs
layer 4: Buffering for I/O devices
layer 3: Operator Console device driver
layer 2: Mem. Mngt
layer 1: CPU Scheduling
layer 0: Hardware

Fig. "THE OS" layer Structure

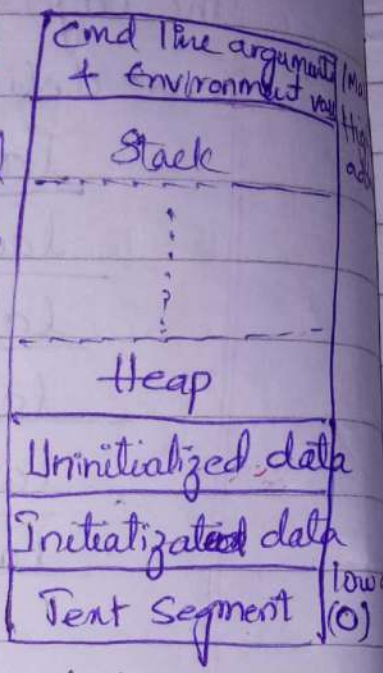
* **Process Management:**

Process Concept: Earlier Computer systems allowed only 1 prg to be executed at a time. This prg had complete control over all the sys resources. In contrast, current day computer sys's allow multiple prgs to be loaded into mem. & executed concurrently. This evolution required firmer control & more compartmentalization of various prgs; & these needs resulted in the notion of process. A process is considered as the unit of work in time-sharing sys's.

* A sys consists of collection of process. OS processes executing sys code & user processes executing user code. All these processes can execute concurrently to make the computer more productive.

Process: An executing instance of a prg is called as a process. (or) A Process is the OS's abstraction for a running prg.

* Process in Memory / Memory layout of a C prg / Process address Space:



1) **Text Segment:** Consists of the machine instructions that CPU executes. It is read only segment so as to prevent a prg from accidentally modifying its instructions.

2) **Initialized data Segment:** Contains the variables that are specially initialized in the prg, outside of any function (Global).

3) **Uninitialized data Segment / Block separated by Symbols:** All uninitialized variables are stored in this segment. By default variables are ~~stored~~ initialized by the kernel to arithmetic zero or null pointers before the prg starts executing.

4) **Stack:** Each fun. is allocated with a stack when it is called. The stack is used to store local variables of a fun, fun return values. And the stack is freedup when the fun. completes its execution.

5) **Heap:** Used for dynamic memory allocation & managed via calls to malloc, realloc, calloc, free, etc...

* The stack & Heap starts in opposite direction and grow towards each other. If they shld ever meet, then either a stack overflow will occur or else a call to malloc will fail due to insufficient memory available.

6) **Command line arguments:** int main(int argc, char *argv[])

argc: No of args passed

argv: a ptr array which ... to prg

* A process by itself is not a process: A prg is a passive entity, such as a file containing a list of instructions stored on disk. (often called as executable file), whereas a process is an active entity, with prg counter specifying the next instruction to execute & set of associated resources.
* A prg becomes a process when a executable file loaded into memory.

* **Process states:-** As a process executes, it changes state. The state of a process is defined in part by the current activity of the process. Each process may be in one of the foll. states.

1) New: The process is being created.
2) Running: Instructions are being executed.
3) Waiting: The process is waiting for some events to occur (such as an I/O completion or reception of a signal)
4) Ready: The process is waiting to be assigned to a processor
5) Terminated: The Process has finished execution.

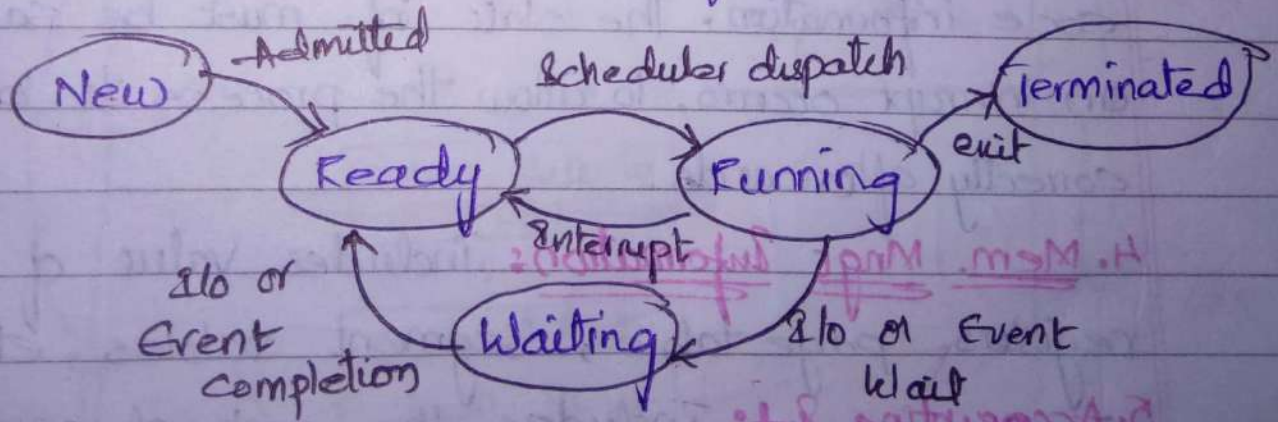


fig. Process State diagram

* Although 2 processes may be associated with the same p

They are nevertheless considered 2 separate execution sequences. For instance, several users may be running diff copies of mail prg, or the same user may invoke many copies of the web browser prg. Each of these is a separate process; & although the text segment is equal, the data, heap & stack sections vary.

*Only 1 process will be running on any processor at any instant, others may be in waiting/ready states.

* Process Control Block (PCB): Each process is represented in the OS by a process control block (PCB). also called as Task Control Block.

PCB contains the foll. information associated with a specific process.

1. Process state: The state may be New, Running, Ready, etc.
2. Program Counter: The counter indicates the address of the next instruction to be executed for this process.
3. CPU Registers: Includes accumulators, Index registers, stack ptrs, general purpose registers, plus any condition code information. The state inf. must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards.
4. Mem. Mngt Information: Includes value of base & limit registers, page table, segment tables, etc.
5. Accounting Inf: Includes the amt of CPU time & real time used, time limits, account nos., job or process no. etc.
6. Ilo status Inf: Includes list of Ilo devices allocated to the

process, a list of open files & so on.
In brief, PCB simply servers as the repository for any information that may vary from process to process.

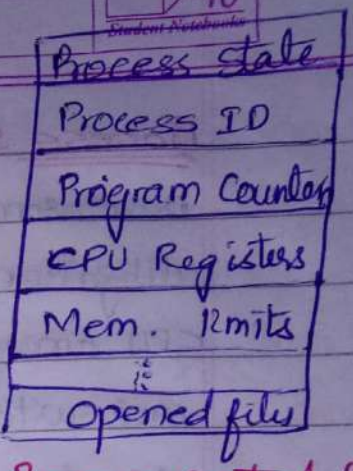


Fig: Process control Block.

* CPU switching from process to process

Whenever CPU switches from 1 process to another process, the state of 1st process is saved back into PCB1 & the state of 2nd process is loaded into OS.

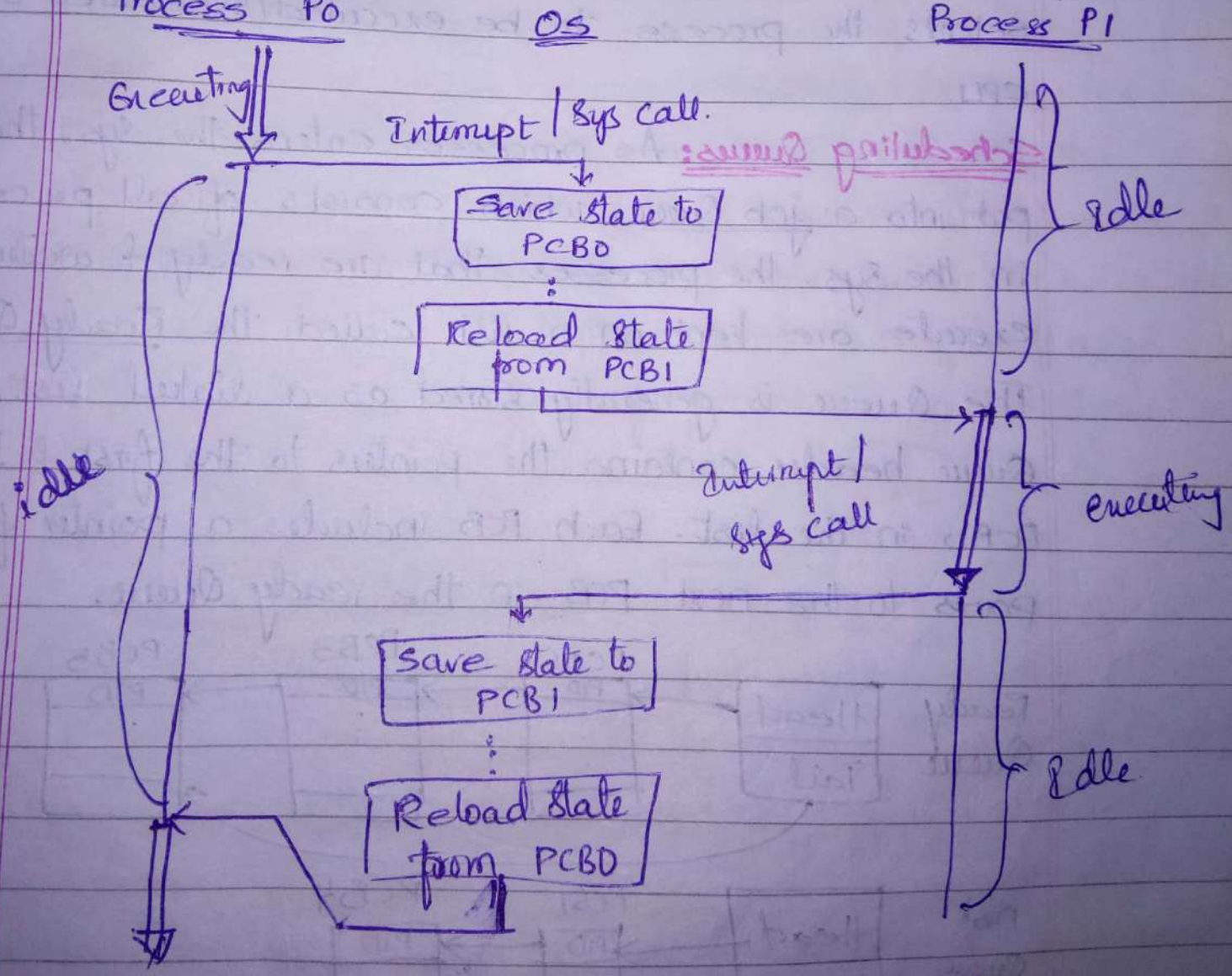


Fig. CPU Switching from process to process.

Process Scheduling: The objective of multiprogramming is to have some process running at all time, to maximize CPU utilization. The objective of time sharing is to switch the CPU among the processes. So frequently that users can interact with each prg while it is running. To meet these objectives, the process scheduler selects available process from a set of several available processes for prg execution on CPU.

* The process scheduler maintains scheduling Queues & selects the process to be executed/scheduled onto the CPU.

Scheduling Queues: As processes enters the sys, they are put into a job Queue, which consists of all processes in the sys. The processes that are ready & waiting to execute are kept on a list called the Ready Queue. The Queue is generally stored as a linked list. A ready Queue header contains the pointers to the first & last PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready Queue.

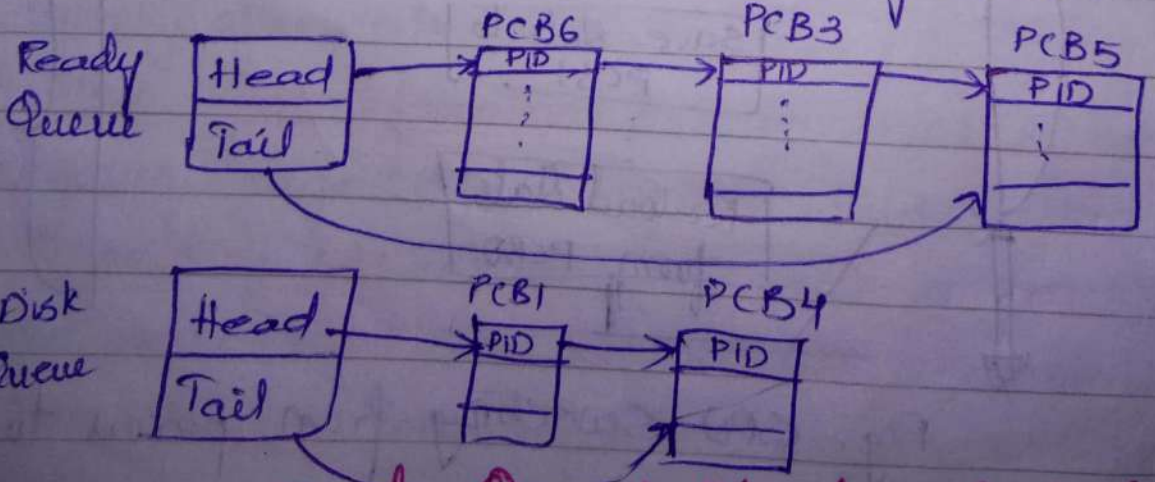


Fig: Ready Queue & I/O device (Disk) Queue

The system also includes other Queues. When a process is allocated with the CPU, it executes for a while & eventually quits, is interrupted or waits for the occurance of a particular event, such as completion of an I/O request to a shared device, such as disk. Since there are many processes in the sys, they may be busy with the I/O req. of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device Queue. Each device has its own queue.

Queueing diagram: A common representation of process scheduling is queuing diagram. Each rectangular box represents a queue. Two types of queues are present: the ready queue & a set of device queues. The circles represent the resources that serve the queues, & arrows indicates the flow of processes in the sys.

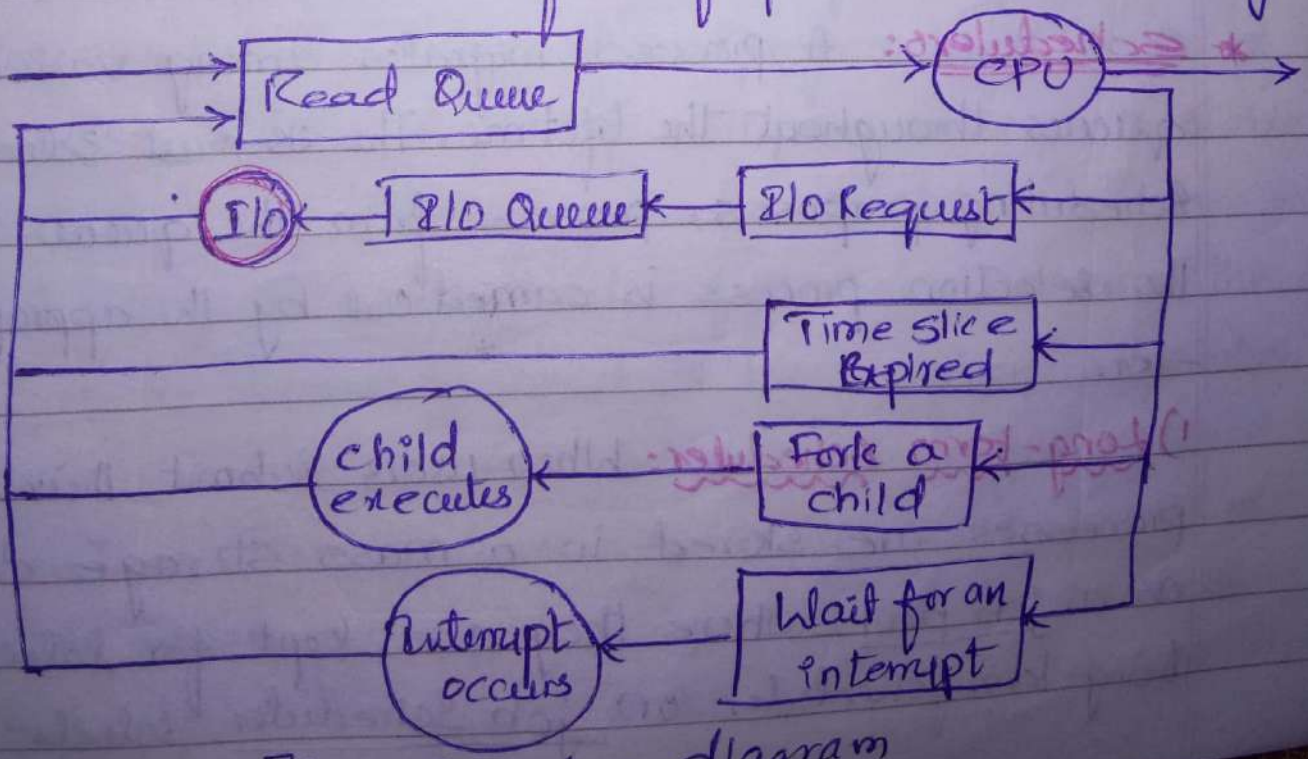


Fig. Queuing diagram

A new process is initially put in the ready Queue. It waits there until it is selected for execution, or is dispatched. Once process is allocated to the CPU & is executing, one of several events could occur.

1) The process could issue an I/O request & then be placed in an I/O Queue.

2) The process could create a new subprocess & wait for the subprocess's termination.

3) The process could be removed forcibly from the CPU, as a result of an interrupt, & be put back in the ready queue.

* In the 1st 2 cases, the process eventually switches, from waiting state to ready state & is then put back in the ready Queue. A process continues this cycle until it terminates, at which time, it is removed from all queues & has its PCB & resources deallocated.

* Schedulers: A process migrates among various scheduling queues throughout the lifetime. The OS must select, for scheduling purposes, process from the queue in some fashion. The selection process is carried out by the appropriate scheduler.

1) Long-term Scheduler: When users submit their prgs. The processes are stored in a mass storage device (disk) as a job pool, where they are kept for later execution. Long-term Scheduler (or) job Scheduler selects processes

from this pool & loads them into the main mem. for execution

2) Short-term scheduler/CPU scheduler: This selects process which are ready to execute from the main mem. & allocate the CPU to one of them.

* The primary distinction blw these 2 schedulers lies in frequency of execution. The STS must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often STS executes atleast once every 100 ms. Because of the short time blw executions, the short term scheduler must be fast.

* If it takes 10ms to decide ~~which~~ to execute a process for 100ms, then $\frac{10}{100+10} = 9\%$ of the CPU is being used (wasted) simply for scheduling work.

* The Long term scheduler executes much less frequently, minutes may seperate the creation of new process & the next. LTS controls the degree of multiprogramming (The no. of processes in mem.) If the degree of multiprogramming is stable, then the avg. rate of process creation must be equal to the avg. departure rate of process leaving the sys. Thus the LTS may need to be invoked only when a process leaves the sys. Because of longer interval blw executions, the LTS can afford to make more time to decide which process shld be selected for execution. It is imp. that the LTS make a careful selection. In general

most processes can be decided as either I/O bound or CPU bound.

I/O Bound Process: It is one that spends more of its time doing I/O than it spends doing computations.

CPU Bound Process: It generates I/O requests infrequently, using more of its time doing computations.

* It is imp that the LTS selects a good process mix of I/O bound & CPU bound.

* Medium Term Scheduler: The key idea behind medium term scheduler is that sometimes it can be advantageous to remove processes from mem. (& from active contention for the CPU) & thus reduce the degree of multiprogramming. Later the process can be re-introduced into mem, & its execution can be continued where it left off. This scheme is called swapping. The process is swapped out & later swapped in by the MTS. Swapping may be necessary to improve the process mix.

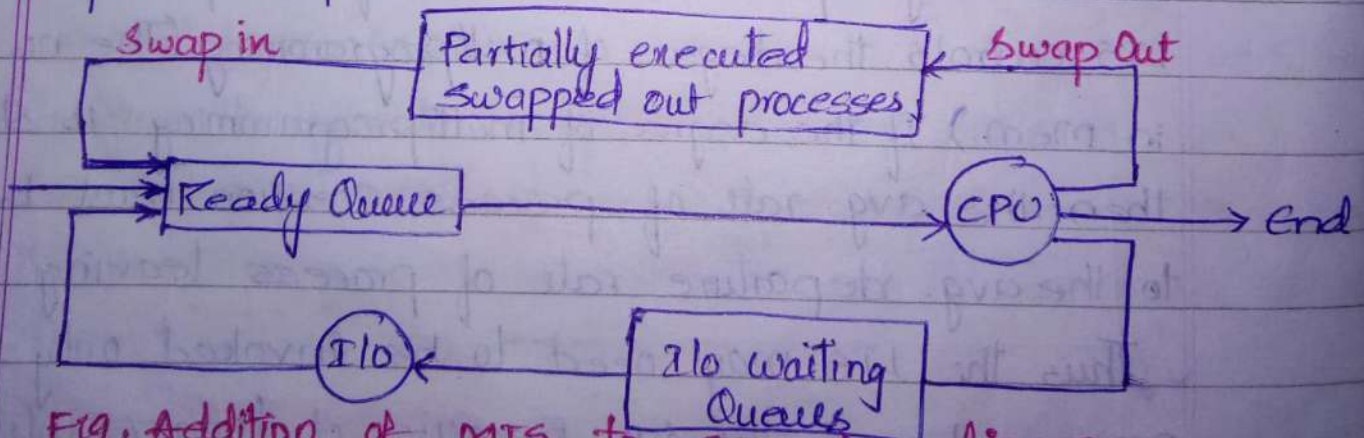


Fig. Addition of MTS to queueing diagram.

* Content Switch: When an interrupt occurs, the Sys needs to save the current content of the process running on the CPU, so that it can restore the content when its

processing is done, essentially suspending the process & resuming it.

* Switching the CPU to another process requires performing a state save of the current process & a state restore of a diff process. This task is known as Context Switch. When a context switch occurs, the kernel saves the context of old process int its PCB & loads the context of the new process scheduled to run.

* Context Switch time is pure overhead, because the sys does not do any useful work while switching.

PROCESS SCHEDULING

In a single-process sys, only 1 process can run at a time, any others must wait until the CPU is free & can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. Several processes are kept in main mem., when a process has to wait the OS takes the CPU away from that process & gives the CPU to another process. The selection process of a process is done by the Short Term Scheduler or CPU Scheduler). It selects a process from the ready Queue.

Preemptive Scheduling: CPU scheduling decisions may take place under the following four instances.

1) When a process switches from the running state to the waiting state (for example, as the result of an IO request or an invocation of wait for the termination of 1 of the child process).

2) When a process switches from running state to the ready state (for eg. when an interupt occurs).
3) When a process switches from the waiting state to ready state (for eg: Completion of I/O).
4) When a process terminates.

* For situation 1 & 4, there is no choice in terms of scheduling. A new process must be selected for execution. There is a choice, however for situations 2 & 3.
* When scheduling takes place only under circumstances 1 & 4 we say that scheduling scheme is non-preemtive or cooperative, otherwise it is preemptive.
* Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or switching to the waiting state. eg: Windows 3.x

Preemptive eg: Later versions of window 95, Mac os x, etc.

* Dispatcher: Dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. It includes the following.

i) Content switch
ii) Switching to user mode
iii) Jumping to the ~~prog~~ proper location in the user prg to restart the prg.

* This shld be as fast as possible, since it is invoked during every process switch.

* The time taken by the dispatcher to stop 1 process & start another running process is known as dispatch latency
* Scheduling criteria: Diff. CPU scheduling algs have diff properties, & the choice of a particular alg may favour 1 class of processes over the other. In choosing which algs to use in a particular situation, we must consider the properties of various algs.
* The criteria for comparing algs is as follows:

1) CPU Utilisation: CPU shld be kept as busy as possible. Conceptually CPU utilization can range from 0 to 100%. In real sys, it shld range from 40% (for lightly loaded sys) to 90% (for heavily loaded sys).

2) Throughput: The no. of process that are completed per unit time is called as Throughput. For long processes, this rate may be 1 process per hour, for short, it may be 10 processes per second.

3) Turnaround Time: The interval from the time of submission of a process to the time of completion is the TAT.

TAT = Time spent waiting to get into mem + Waiting in Ready Queue + Executing on CPU + doing IIO.

4) Waiting Time: The CPU scheduling alg doesnot affect the amount of time during which a process executes or does IIO. It affects only the amt of time that a process spends waiting in the Ready Queue.

Waiting time is the sum of the periods spent waiting in the ready Queue.

5) Response Time: In an interactive sys, TAT may not be best criterion. Often a process can produce some o/p fairly early & can continue computing new results while previous results are being o/p to the user. Thus another measure is the time from submission of request to until the 1st response is produced. This measure is called as Response Time, is the time it takes to starts responding, not the time it takes to o/p the response.

Goal of CPU scheduler
* It is desirable to maximize CPU utilization & Throughput & to minimize TAT, WT & RT.

* Process Times:

1) Arrival Time / Submission Time (AT): When the process is ready/arrived into main mem.

2) Burst Time / Service Time (BT): The CPU time required for a process to complete its execution.

3) Completion Time (CT): Time at which process execution is completed.

4) Waiting Time (WT): Can be → [I/O time, Ready Queue waiting]

$$WT = CT - BT - AT$$
$$= TAT - BT \quad [\because TAT = CT - AT]$$

5) Turnaround Time (TAT): CT − AT

6) Deadline: No. of processes completed within the deadline

* Process Example:

AT	BT	WT	WT	WT	BT	WT	WT	BT	CT
	CPU	RQ	I/O	RQ	CPU	I/O	RQ	CPU	

← TAT →

Date / /
Page

* Consider 'n' processes $P_1, P_2, P_3, \ldots P_n$

Notations: $AT = A_i$, $BT = B_i$, $CT = C_i$, Deadline $= D_i$

Formulae:

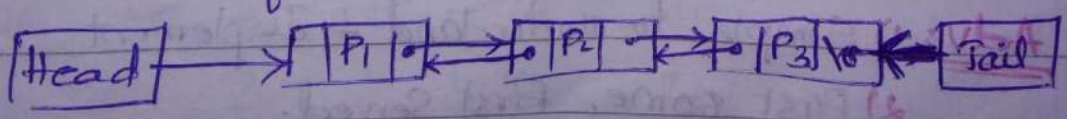
1. Turnaround Time $(TAT_i) = C_i - A_i$
2. Avg. $TAT = \sum_{i=1}^{n} (C_i - A_i)/n$
3. $WT = TAT - BT = (C_i - A_i) - B_i$
4. Avg. $WT. = \sum_{i=1}^{n} (C_i - B_i - A_i)/n$
5. Schedule length (All processes Time) $l = Max(C_i) - Min(A_i)$
6. Throughput $= n/l$ (no. of process / Time)

Scheduling Algorithms

CPU Scheduling deals with the prblm of deciding which of the processes in the Ready Queue is to be allocated to the CPU.

1) First Come, First Served (FCFS) Scheduling: With this, the process that requests the CPU first is allocated the CPU first.

* The implementation of FCFS policy is easily managed with FIFO Queue. When a process enters the ready Queue, its PCB is linked onto the tail of the queue. When CPU is free, it is allocated at the head of the queue.



Eg: BT = The Time required for a process to complete its execution on the processor.

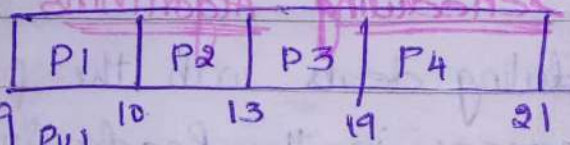
* **Gantt chart:** It's a bar chart that illustrates a particular schedule, including the start & finish times each of the participating processes.

* Consider the foll. processes, with their BT (i.e., without AT)

Process	AT	BT (ms)
P1	0	10
P2	1	3
P3	2	6
P4	3	2

The Arrival Time order is P1, P2, P3, P4

Gantt chart:

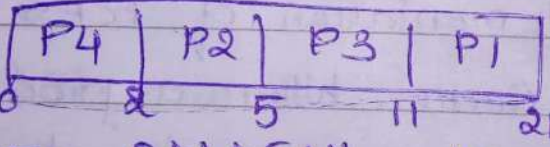


P	P1	P2	P3	P4
WT	0	10	13	19

$$\text{Avg. WT} = \frac{\text{Total WT of all processes}}{\text{No. of processes}}$$

$$= \frac{0+10+13+19}{4} = \frac{42}{4} = 10.5 \text{ ms}$$

* If the processes arrive in the order P4, P2, P3, P1 then Gantt chart



The avg. WT $= \frac{0+2+5+11}{4} = \frac{18}{4} = 4.5$ ms

Hence avg. WT depends on the order & BT's of processes.

Adv: 1) Simple to understand & Implement
2) First come, First Served.

DisAdv: 1) It is Non-preemptive (i.e., the process will run until its finished).

2) **Convoy Effect:** All the other processes wait for a big

process to get off the CPU. The effect lowers the CPU utilization.

7. Deadline overrun = $C_i - D_i$

if $(C_i - D_i) < 0$ under run

> 0 over run

$= 0$ Completed on Deadline.

Eg2: FCFS With Arrival Time:

P.NO.	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	1	2	6	5	3
3	2	3	9	7	4
4	3	5	14	11	6
5	4	6	20	16	10
6	5	2	22	17	15

Schedule length,

$l = Max(C_i) - Min(A$

$= 22 - 0$

$= 22$ //

Gantt chart:

P1	P2	P3	P4	P5	P6
0	4	6	9	14	20	22

$TAT_1 = C_1 - A_1 = 4 - 0 = 4$

$TAT_2 = 6 - 1 = 5$

$TAT_3 = 9 - 2 = 7$

$WT_1 = TAT_1 - BT_1$

$= 4 - 4 = 0$

$WT_2 = 5 - 2 = 3$

Avg WT $= \frac{0+3+4+6+10+15}{6}$

$= 38/6$

Eg3:

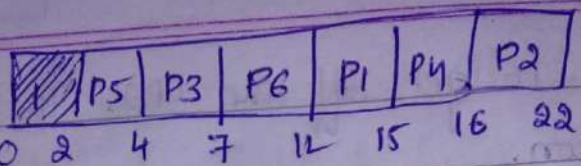
P.NO.	AT	BT	CT	TAT	WT
1	5	3	15	10	7
2	8	6	22	14	8
3	3	3	7	4	1
4	6	1	16	10	9
5	2	2	4	2	0
6	4	5	12	8	3

$l = 22 - 2$

$= 20$

Date / /
Page

Gantt chart:



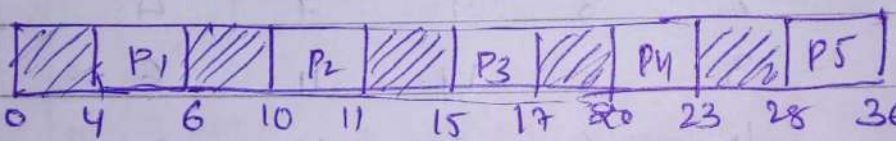
Throughput = 6/20 =

Avg WT = $\frac{7+8+1+9+0+3}{6} = \frac{28}{6}$

Eg 4

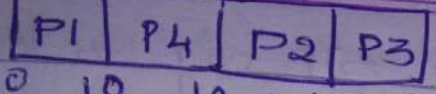
P.NO	AT	BT	CT	TAT	WT
1	4	2	6	2	0
2	10	1	11	1	0
3	15	2	17	2	0
4	20	3	23	3	0
5	28	8	36	8	0

Avg WT = 0



② Shortest job First Scheduling (SJF): When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of 2 processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the Shortest-next CPU-burst alg, because scheduling depends on the length of next CPU burst of a process, rather than its total length. Also called as "Shortest process Next" (SPN)

P.NO.	AT	BT	CT	TAT	WT
1	0	10	10	10	0
2	1	3	15	14	11
3	2	6	21	19	13
4	3	2	12	9	7



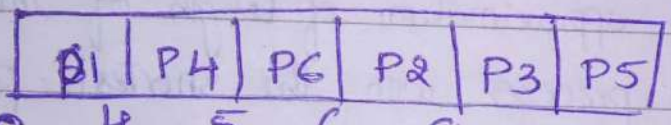
Avg WT = $\frac{0+11+13+13}{4} = \frac{37}{4} = 9.25$

eg 2)

P.NO	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	1	2	8	7	5
3	2	3	11	9	6
4	3	1	5	2	1
5	4	5	16	12	7
6	5	1	6	1	0

Avg WT = $\frac{19}{6} = 3.17$

$l = 16 - 0 = 16$



Eg 3)

P.NO.	AT	BT	CT	TAT	WT
1	8	2	13	5	3
2	3	2	11	8	6
3	6	5	21	15	10
4	2	8	29	27	19
5	5	3	16	11	8
6	4	1	9	5	4
7	2	6	8	6	0

Schedule length.

$l = 29 - 2$

$= 27$

Avg WT = $\frac{40}{7}$

$= 5.71$

* It is the optimal scheduling algorithm. i.e., It gives the min. avg. waiting time for a given set of processes

* The real difficulty with the SJF alg. is knowing the length of the next CPU request. For LTS users are motivated to estimate the process time limit accurately, when users submit the job. But for SJS, there is no way to know the length of the next CPU burst. One approach is to approximate the SJF scheduling.

✓ We may not know the length of the next CPU brust, but we may be able to predict its value. We expect that next CPU burst will be similar in length to the previous ones.

✓ By computing an approximation of length of the next CPU burst, we can pick the process with the shortest predicted CPU burst

Method*1 **Exponential Average (or) Aging Algorithm:** The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. We can define exponential avg with the foll. formula.

Let t_n be the length of n^{th} CPU burst.

- Contains most recent Information

→ T_n - Stores the past history of CPU burst.

T_{n+1} be the predicted value for our next CPU burst

Then:

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n \quad , \quad 0 \le \alpha \le 1$$

Where α - Controls the relative weight of most recent & past history in our prediction.

⇒ If $\alpha = 0$, then $T_{n+1} = T_n$ ⇒ Recent history has no effect.

⇒ If $\alpha = 1$, then $T_{n+1} = t_n$ ⇒ Only the most recent CPU burst matters.

⇒ If $\alpha = 1/2$, ⇒ Recent history & past history are equally weighted

* The initial T_0 can be defined as a constant or as an overall system avg.

* To understand the behaviour of exponential avg. we can expand the formula by substituting T_n

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n \quad \text{——— ①}$$

When $n = n-1$

$$T_n = \alpha t_{n-1} + (1-\alpha) T_{n-1} \quad \text{——— ②}$$

Substitute ② in ①

$$T_{n+1} = \alpha t_n + (1-\alpha)[\alpha t_{n-1} + (1-\alpha) T_{n-1}]$$

$$T_{n+1} = \alpha t_n + \alpha(1-\alpha) t_{n-1} + (1-\alpha)^2 T_{n-1} \quad \text{——— ③}$$

Again substitute for T_{n-1}

$$T_{n+1} = \alpha t_n + \alpha(1-\alpha) t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + (1-\alpha)^3 T_{n-2} \quad \text{——— ④}$$

$$\vdots$$

$$T_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \dots + (1-\alpha)^j \alpha t_{n-j} + \dots + (1-\alpha)^{n+1} T_0.$$

① Consider the burst times for a process as 4, 8, 6, 7 & predict the value of nxt burst. Also given $\alpha = 1/2$ & $T_1 = 10$.

A) Given $\alpha = 1/2$ & $T_1 = 10$

$t_1 = 4$, $t_2 = 8$, $t_3 = 6$ & $t_4 = 7$, $T_5 = ?$

$T_5 = 1/2 (t_4 + T_4) = 1/2 (7 + 6.75) = 6.875$

$T_4 = 1/2 (t_3 + T_3) = 1/2 (6 + 7.5) = 6.75$

$T_3 = 1/2 (t_2 + T_2) = 1/2 (8 + 7) = 7.5$

$T_2 = 1/2 (t_1 + T_1) = 1/2 (4 + 10) = 7$

Date / /
Page

⑤ Process burst Times: t_1 40, t_2 20, t_3 40, t_4 15 & $\alpha = 1/2$, $T_1 = 10$

Find T_5

A) $T_5 = 1/2(t_4 + T_4) = 1/2(15 + 31.25) = 23.125$ //

$T_4 = 1/2(t_3 + T_3) = 1/2(40 + 22.5) = 31.25$

$T_3 = 1/2(t_2 + T_2) = 1/2(20 + 25) = 22.5$

$T_2 = 1/2(t_1 + T_1) = 1/2(40 + 10) = 25$

Method 2 Simple Average formula

$$T_{n+1} = 1/n \sum_{i=1}^{n} t_i$$

Ans of Eq:2

$T_5 = 1/4 [40 + 20 + 40 + 15]$

$= 1/4 (115) = 27.75$

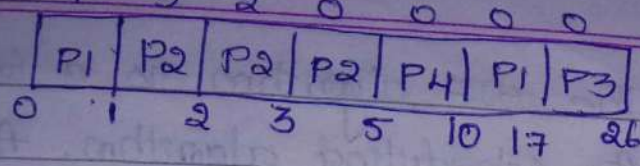
* Adv: 1) Max. Throughput
2) Min. Avg WT & TAT
3) can be implemented with prediction / estimated BT.

* Disadv: 1) Starvation for longer jobs.
2) It is not implementable because Actual BT is not known.

* **Shortest Remaining Time First (SRTF):** This is also called as preemptive SJF scheduling algorithm. The preemption of running process is based on the arrival of a new shorter process.

Eg 1.

P.NO.	AT	BT	CT	TAT	WT
1	0	8	17	17	9
2	1	4	5	04	0
3	2	9	26	24	15
4	3	5	10	07	2

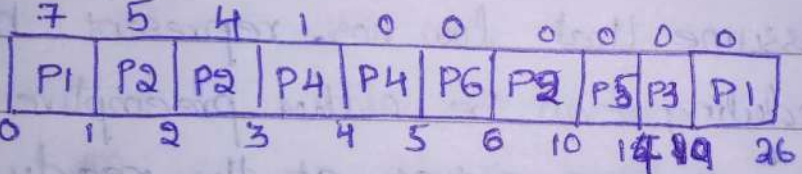


Avg. WT = $\frac{9+0+15+2}{4} = \frac{26}{4} = 6.5$

Schedule Length, $L = 26 - 0 = 26$

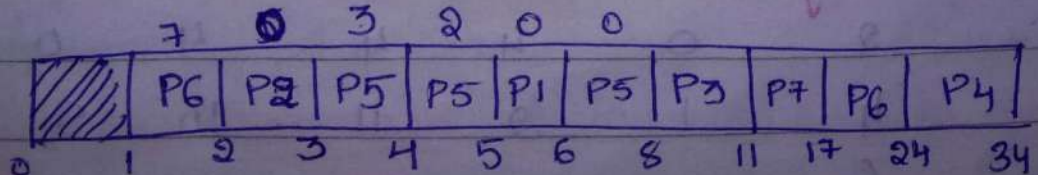
2)

P.No.	AT	BT	CT	TAT	WT
1	0	8	26	26	18
2	1	6	10	09	3
3	2	5	19	17	12
4	3	2	05	02	0
5	4	4	14	10	6
6	5	1	06	01	0



3)

P.No.	AT	BT	CT	TAT	WT
1	5	1	6	1	0
2	2	1	3	1	0
3	4	3	11	7	4
4	1	10	34	33	23
5	3	4	08	5	01
6	1	8	24	23	15
7	2	6	17	15	09



Date / /
Page

* **Priority Scheduling:** The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, & the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

✓ An SJF algorithms is simply a priority algorithm where the priority (P) is the inverse of the (predicted) next CPU burst. The longer the CPU brust, the lower the priority & vice versa.

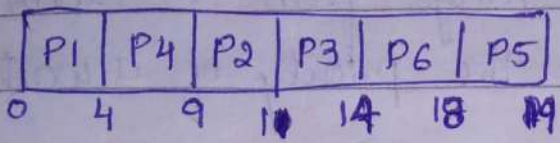
✓ Priorities are generally indicated by some fixed range of nos. such as 0 to 7 or 0 to 4095. However, there is no general agreement on whether 0 is the highest or lowest priority. Here we assume that low nos. represent high priority.

✓ Priority Scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of currently running process. A preemptive priority scheduling alg. will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling alg. will simply put the new process at the head of the ready queue.

1) **For Non-Preemptive priority Scheduling:**

P.NO.	Priority	AT	BT	CT	TAT	WT
1	2	0	4	4	4	0
2	4	1	2	11	10	8
3	6	2	3	14	12	9
4	3	3	5	9	6	1

5	8	4	1	19	15	14
6	7	5	4	18	13	9



$l = 19 - 0 = 19$ //

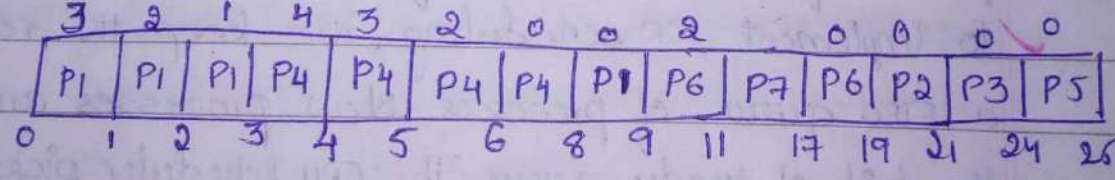
1) For Preemptive Priority Scheduling:

P. NO.	Priority	AT	BT	CT	TAT	WT
* 1	2	0	4	9	9	5
* 2	4	1	2	21	20	18
* 3	6	2	3	24	22	19
* 4	1	3	5	8	5	0
* 5	8	4	1	25	21	20
* 6	3	5	4	19	14	10
* 7	2	11	6	17	6	0

Avg WT = 10.2857

Throughput = $\frac{7}{25}$ = 0.28 //

$l = 25$



Disadvantages:-

* Indefinite blocking or Starvation: A priority Scheduling alg. can leave some low priority processes waiting indefinitely. In a heavily loaded Sys, a steady stream of higher priority processes can prevent a low priority proces from ever getting the CPU.

* Soln for Starvation: Aging: A soln to the prblm of indefinitely blocking of low priority processes is Aging.

Aging is a technique of gradually increasing the priority of processes that wait in the sys for a long time. For example, if priorities range from 127 (low) to 0 (high), we would increase the priority of a waiting process by 1 every 15 mins.

Round Robin Scheduling (RR): The RR scheduling alg. is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the sys. to switch btw processes. A small unit of time, called a time quantum or time slice is defined. A time quantum is generally from 10ms to 100ms in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of upto 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum & dispatches the process.

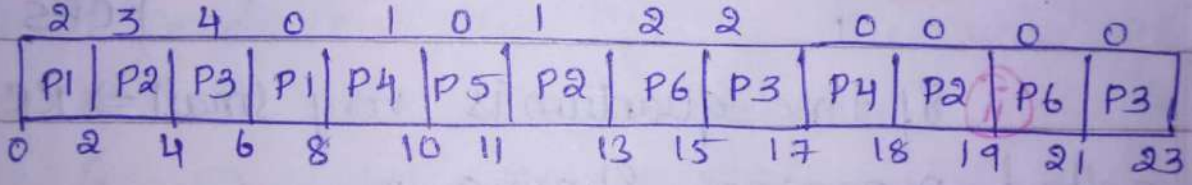
One of the 2 things will then happen. The process may have a CPU burst of less than 1 time quantum, in this case the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst time of currently running process is longer than 1 time quantum, the timer will

go off & will cause an interrupt to the OS. A context switch will be executed, & the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

1)

P.NO.	AT	BT	CT	TAT	WT
1	0	4	8	8	4
2	1	5	19	18	13
3	2	6	23	21	15
4	3	3	18	15	12
5	4	1	11	7	6
6	5	4	21	16	12

Consider Time Quant = 2.



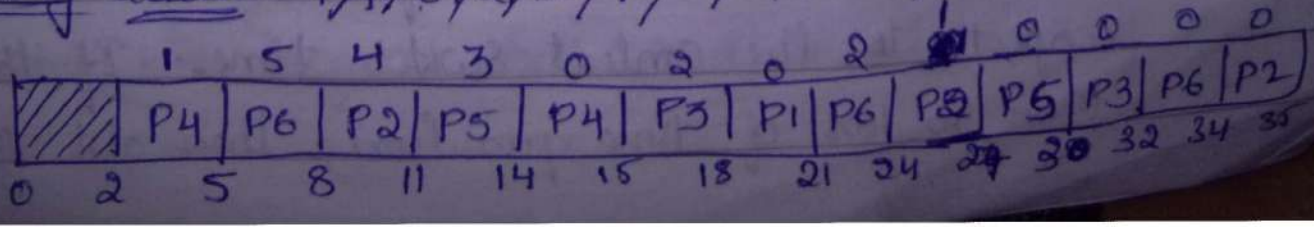
Ready Queue: P1 P2 P3 P1 P4 P5 P2 P6 P3 P4 P2 P6 P3

②

P.NO	AT	BT	CT	TAT	WT
1	7	3	21	14	11
2	4	7	35	31	24
3	6	5	32	26	21
4	2	4	15	13	9
5	5	6	30	25	19
6	3	8	34	31	23

TQ=3

Ready Queue = P4 P6 P2 P5 P4 P3 P1 P6 P2 P5 P3 P6 P2



* If there are 'n' processes in the ready queue & the time quantum is 'q', then each process gets "1/n" of the CPU time in chunks of atmost 'q' time units.
* Each process must wait no longer than (n-1)q time units until its next time quantum.

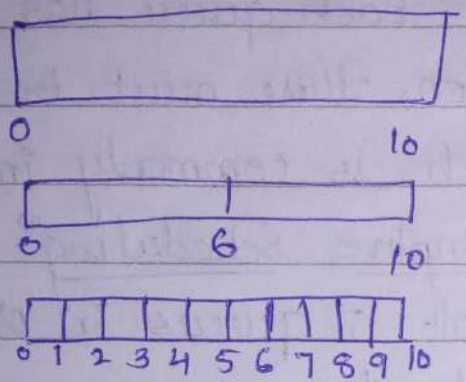
eg: If 5 processes & q = 20ms then each process will get upto 20ms every 100ms.
For next time quantum = (n-1) q
= 4 * 20 = 80ms // it has to wait

Performance of RR: Depends heavily on the size of time quantum. (i) If time quantum is larger ⇒ It is same as FCFS policy.

(ii) If time quantum is very small ⇒ RR approach is called processor sharing. i-e,
* The Process Spends more time in Context Switching ⇒ Overhead
eg: A process of 10 time units. If the quantum is 12 time units; the process finishes in less than 1 time quantum with no overhead.
* If the quantum = 6 time units, process requires 2 quantum resulting in a Context Switch.
* If the quantum = 1 time unit, then 9 context switches will occur, slowing the execution of process.
* Thus we want the time quantum to be large with respect to the context switch time. If the context switch time is 10% of time quantum, then abt 10% of the CPU

time will be wasted in context switching.

* Generally, most systems have time quantum ranging from 10ms-100ms & the time required for context switch is typically less than 10 ms.



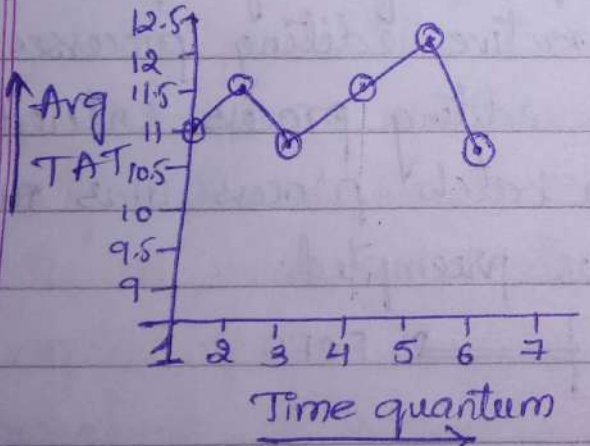
quantum	Context Switches
12	0
6	1
1	9

process. Time = 10 //

* Turnaround Time Vs Time Quantum: Turnaround time also depends on the size of the time quantum.

* eg:

P1	P2	P3	P4	→ Processes
6	3	1	7	→ BT



From the fig. the avg TAT of a set of processes does not necessarily improve as the time quantum size increases.

* eg: Consider 3 processes & each has 10 ms BT

If TQ=1 Avg. TAT = 28+29+30/3 = 29 | Improved if TQ is high.

If TQ=10 Avg. TAT = 10+20+30/3 = 20 ↓

* TQ shld be large, but it shld not be too large, which leads to FCFS.

Date / /
Page

* **Multilevel Queue Scheduling:** This alg partitions the ready queue into several separate queues. The processes are permanently assigned to 1 queue, generally based on some property of the process, such as mem. size, process priority, or process type. Each queue has its own scheduling alg. In addition, there must be scheduling among the queues, which is commonly implemented as "Fixed priority preemptive scheduling".

Eg: Lets consider the foll. 5 queues in order of priority 1) Sys processes, 2) Interactive processes, 3) Interactive editing processes, 4) Batch processes, 5) Student Process

* Each queue has absolute priority over lower priority queues. No process in the batch queue, for example, could run unless the queue for sys. processes, interactive processes & Interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

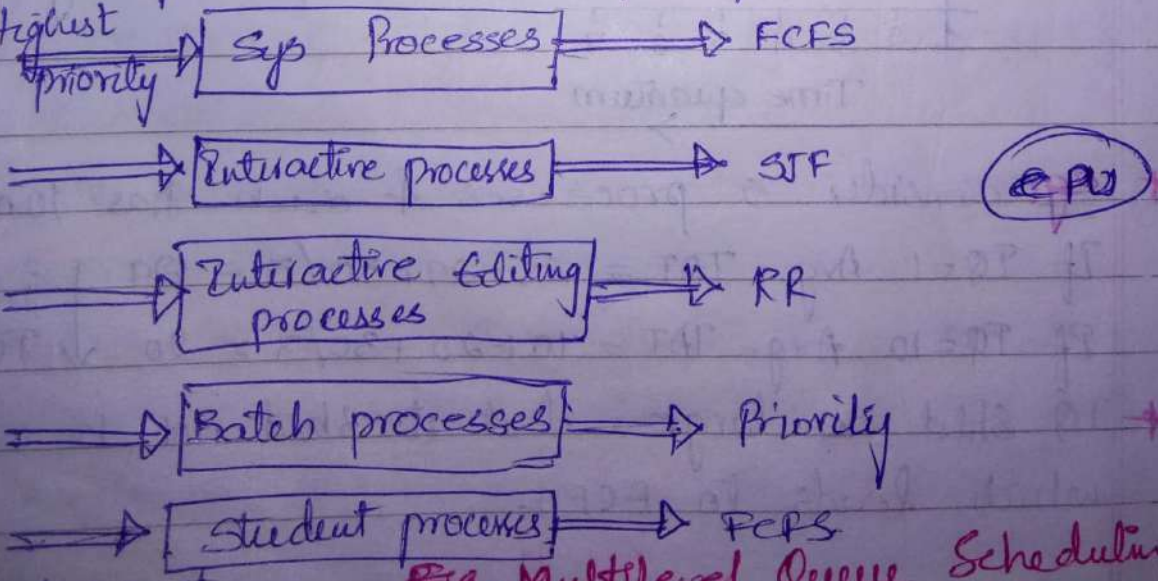


Fig. Multilevel Queue Scheduling

Date / /
Page

* Another possibility is to time slice among the queues. Here each queue gets a certain amt of CPU time, which it can schedule amg its various processes.
* Disadv: ① Starvation for lower queues.
② Lower flexibility for processes as they can't go to diff. queue.
* Multilevel Feedback Queue Scheduling:- It allows a process to move blw queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower priority queue. This scheme leaves I/O bound & interactive processes in the higher-priority queues. In addition a process that waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation
Eg: Consider a multilevel feedback queue scheduler with 3 queues, numbered from 0 to 2. The scheduler 1st executes all processes in the queue 0. Only when queue 0 is empty, it will execute processes in queue 1. Similary processes, in queue 2 will only be executed if queues 0 & 1 are empty. A process that arrives in queue 1 will preempt a process in queue 2. A process in queue 1 will in turn preempted by a process arriving for queue 0.
* A process entering the ready queue is put in queue 0.

If it does not finish within this time, it is moved to the tail of the queue 1. If queue 0 is empty, the process at the head of queue 1 is given a TQ of 16ms. If it does not complete, it is preempted & is put into queue 2. process in queue 2 are run on FCFS basis but are run only when queue 0 & 1 are empty.

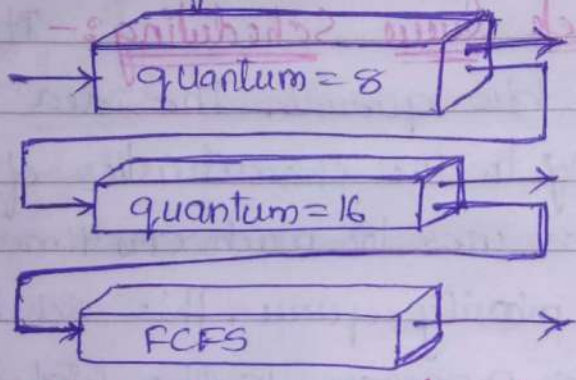


Fig. Multilevel feedback queues

* This scheduling alg. gives highest priority to any process with a CPU burst of 8ms or less. Such a process will quickly get the CPU, finish its CPU burst, & go off to its next I/O burst processes that need more than 8 but less than 24ms. are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 & are served is FCFS order with any CPU cycles left over from queues 0 & 1.

* In general, a multilevel feedback queue scheduler is defined by the foll. parameters.

1) The no. of queues.

ii) The Scheduling alg for each queue.
iii) The method used to determine when to upgrade a process to a higher priority queue.
iv) The method used to determine when to demote a process to a lower level priority queue.
v) The method used to determine which queue a process will enter when that process needs service.

Multithreaded Programming

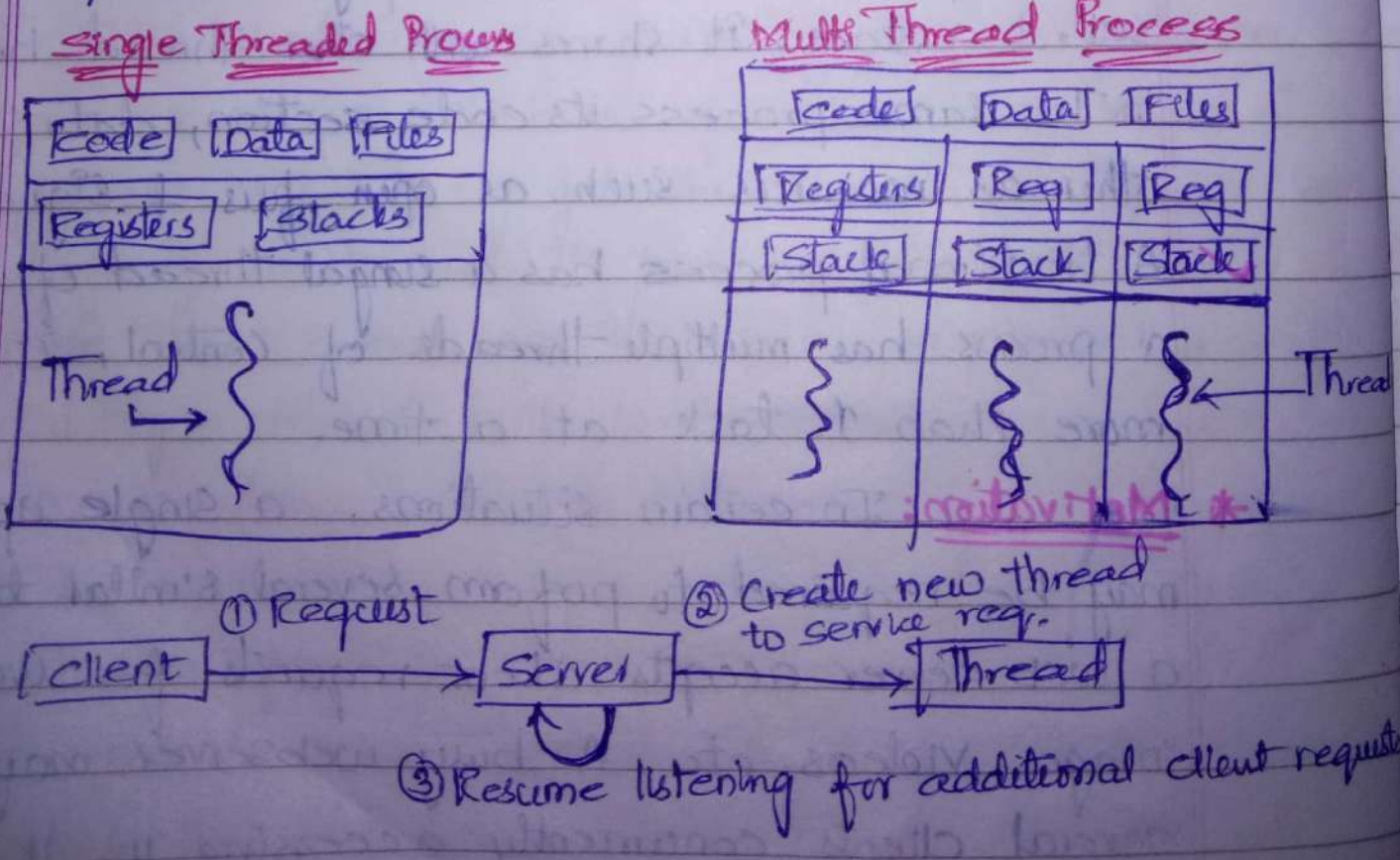
* **Thread:** Thread is a basis unit of CPU utilization (or) Its also called as a light weight process.
✓ Threads provide a way to improve application performance through parallelism.
✓ It comprises a thread ID, a prg counter, a register set, & a stack. It shares with other threads belonging to the same process its code section, data section & other as resources such as open files & signals.
✓ A traditional process has a singal thread of control. If a process has multiple threads of control, it can perform more than 1 task at a time.
* **Motivation:** In certain situations, a single application may be required to perform several similar tasks. For eg, a web server accepts client requests for web pages, images, videos, etc... A busy webserver may have several clients concurrently accessing it. If the web server is running as a traditional single threaded process

it would be able to service only one client at a time, & other clients have to wait for very long time for its requests to be serviced.

There are basically 2 soln to solve this kind of prblem:

① When the server is running & it is serving the request of 1 client & if any other client requests for service, the server can create a new process & service it. But process creation is time consuming & resource intensive. However if the new process will perform the same task as existing process, why incur all that overhead?

② Instead of creating a new process, server can create a new thread, which is rather simple & uses resources than a process.



Date / /
Page

* Difference blw a process & Thread.

Process	Thread
1) A process is a heavy weight or resource intensive	1) Thread is a light weight taking lesser resource than the process.
2) Process switching needs interaction with OS.	2) Thread switching does not need to interact with OS.
3) In multiple processesing environment, each process executes the same code but has its own resources like mem & files.	3) All threads can share same set of open files, ~~child processes~~ code segment, data segment.
4) The complete prg is blocked if the process is blocked.	4) If 1 thread is blocked, other threads can exe. a part of prg.
5) Multiple processes without using threads use more resources.	5) Multiple threaded processes use fewer resources.
6) Each process operates independently of others	6) One thread can read, write or change other thread's data.

Advs. of Multithreading: ① Responsiveness: Multithreading an interactive application may allow a prg to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsive to the user.

② Resource Sharing: Processes may only share resources through techniques such as shared mem or msg passing.

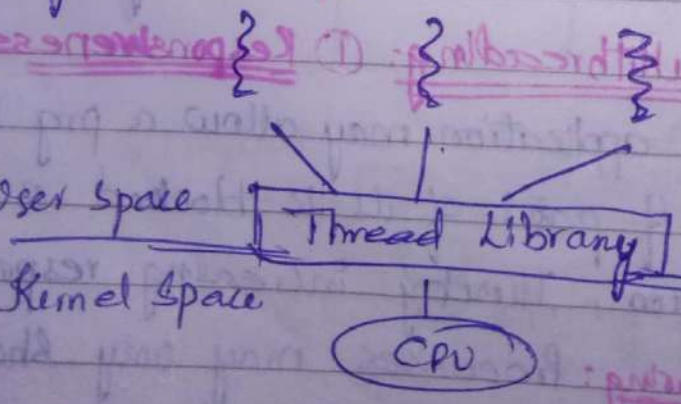
Such techniques must be explicitly arranged by the programmer. However, threads share the mem. & the resources of the process to which they belong by default. The benefit of sharing code & data is that it allows an application to have several diff. threads of activity within the same address space.

③ Economy: context-switching & requirement of resources is economical.

④ Scalability: Multithreading as a multi-cpu machine increases parallelism.

* Multithreading Models: The support for threads is provided at the user level & kernel level.

1) User Level Threads: Applications manages threads. The kernel is not aware of existence of threads. The thread library contains code for creating & destroying threads, for passing msgs & data b/w threads, for scheduling thread exe & for saving & restoring thread contexts. The application begins with the single thread & begins running in that thread.



Adv: ① Thread switching does not require kernel privileges.

② User level threads can run on any OS.

③ Fast to create & manage

Date / /
Page

Disadv: 1) Since user level threads are invisible to os, os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/o, even though the process has other threads that can run, etc.

2) User level threads requires non-blocking sys calls i.e, multithreaded kernel. Otherwise entire process will be blocked, even if there are runnable threads left in the processes. (Since user level threads do not have privilegs to handle sys calls. If 1 page fault occurs, the process ~~will~~ will be blocked).

3) Kernel Level Threads: Thread mngt is done by the Kernel. Kernel threads are directly supported by os.

✓ Kernel-threads are used to provide privileged services to applications (such as sys calls). They are also used by the kernel to keep track of what all is running on the sys, how much of which resources are allocated to what process, & to schedule them.

✓ If an application makes use of heavy sys calls, & if there are more user threads per kernel thread, then your application will run slower. This is because the kernel thread will become a bottleneck, since all sys calls will pass through it. On the otherside if your progs rarely use sys calls (or other kernel resources), you can assign a large no. of user threads to kernel thread without much performance penalty, other than overhead.

Date / /
Page

✓There are generally 3 models of establishing relationship btw user level threads & kernel level thread.

1) Many-to-One Model: This maps many user level threads to 1 kernel thread.

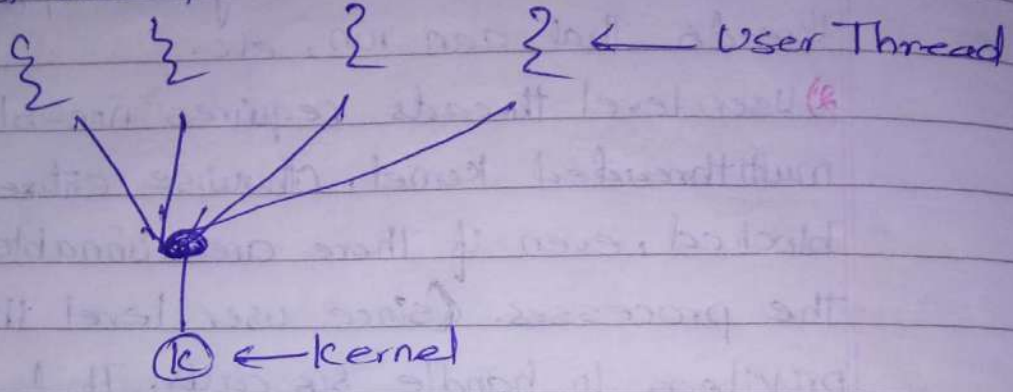


Fig. Many-to-one thread

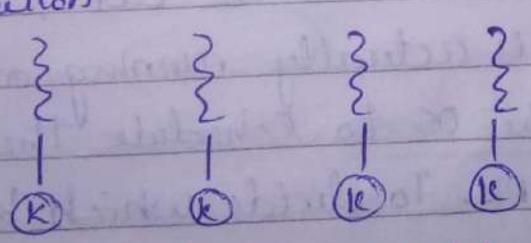
Since only 1 thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. eg: Green threads - a thread library in Solaris

2) One-to-One model: It maps each user thread to a kernel thread. It provides more concurrency than the many-to-one thread model by allowing another thread to run when a thread makes a blocking sys call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating a corresponding kernel thread. The overhead of creating kernel threads can burden the performance of an application, most implementations of the model restrict the no. of threads supported by the sys.

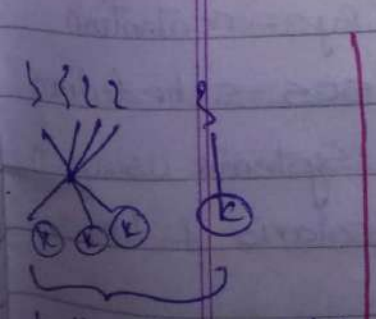
eg: Linux, Windows.

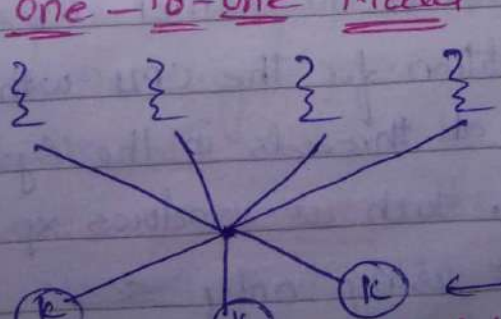
3) Many-to-Many Model: This multiplexes many user-level threads to a smaller or equal no. of kernel threads.

* Many-to-one model allow the developer to create as many user threads as he wishes, but true concurrency is not gained because the kernel can schedule only 1 thread at a time.
* The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application.
* The many-to-many model suffers from neither of these short comings: Developers can create as many user threads as necessary & the corresponding kernel threads can run in parallel on a multiprocessor. Also when a thread performs blocking sys call, the kernel can schedule another thread for execution



one-to-one Model





Many-to-Many Model

Date / /
Page

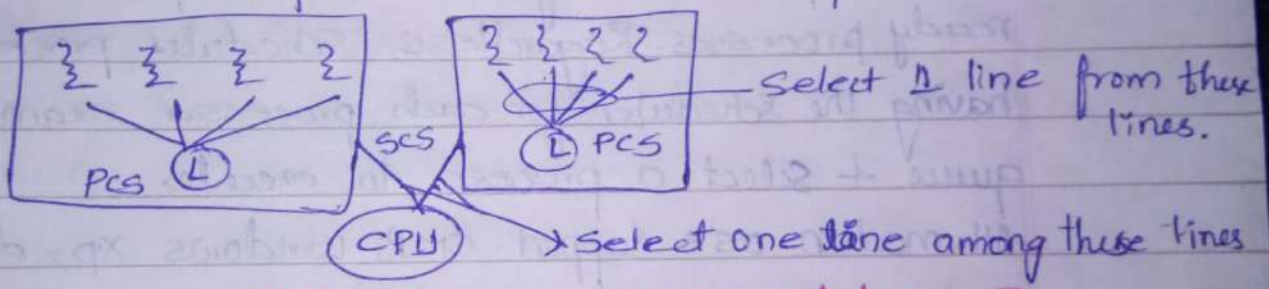
Thread Scheduling:

* On OS's that support threads, It is kernel level threads -not processes- that are being scheduled by the OS. User level threads are managed by a thread library & the kernel is unaware of them.
* To run on a CPU the user level threads must ultimately be mapped to an associated kernel-level-threads, although this mapping may be indirect & may use a ~~light~~ light weight process (LWP).
* A distinction b/w the user-level & kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one models, the thread scheduler lets user level threads to run on an available LWP, a scheme known as process-contention-scope (PCS), since competition for the CPU takes place among threads belonging to the same process. When we say the thread library schedules user threads onto available LWPS, we do not mean that the thread is actually running on a CPU, this would require the OS to schedule the kernel thread onto a physical CPU. To decide which kernel thread to schedule onto a CPU, the kernel uses sys-contention scope (SCS). Competition for the CPU with SCS scheduling takes place among all threads in the sys. Systems using the one-to-one model, such as windows xp, solaris & Linux, schedule threads using only SCS.

Typically PCS is done according to priority the scheduler selects the runnable thread with the highest priority to

run. User level threads priorities are set by the programmer & are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is imp to note that PCS will typically preempt the thread currently running in favour of a higher priority thread; however there is no guarantee of time slicing among threads for equal priority

* Many-to-many model, user threads can have either contention or process contention scope.



Multi-Processor Scheduling

Till now, we have focussed on the prblms of scheduling the CPU in a sys with a single processor. If multiple CPU's are available, load sharing becomes possible; however the scheduling prblm becomes correspondingly more complex. Many possibilities have been tried; & as we saw with single processor CPU scheduling, there is no 1 best soln.

* Homogeneous processors: The processors which are identical in their funtionality.
* Approaches to multiple-processor Scheduling: There are 2 approaches.

Date / /
Page

(AMP)

1) Assymmetric multiprocessing: All the scheduling decisions, I/O processing & other sys activities handled by a single processor the master server & other processors execute only user code.

* This is very simple because only 1 processor accesses the sys data structures, reducing the need for data sharing.

2) Symmetric Multiprocessing (SMP): Each processor is self scheduling. All processes may be in a common ready queue or each process may have its own private queue of ready processes. Regardless, scheduler proceeds by having the scheduler for each processor examine the ready queue & select a process to execute.

All modern os's support SMP, windows xp, Linux, Mac OS

Issues concerning SMP Systems:

(1) Processor Affinity: When a process is running on a specific processor, the data most recently used by/accessed by the process populate the cache for the process are often satisfied in cache mem. If the process migrates to another processor, the contents of cache mem must be invalidated for the 1st processor, & the cache for 2nd processor must be repopulated.

Because of the high cost of invalidating & repopulating caches, most SMP Syss try to avoid migration of processes from 1 processor to anther & instead attempt to keep a process running on the same processor. This is known

as processor Affinity - i.e., a process has an affinity for a processor in which it is currently running.

* Two types of processor Affinity (Forms):

i) Soft Affinity: When an OS has a policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so, is called as Soft Affinity.

ii) Hard Affinity (possible Migration): Os allows a process to specify that it is not to migrate blw processors. eg: Linux.

② Main Mem. Architecture of a System: can affect processor affinity issues. Fig. illustrates Non-Uniform Mem Architecture (NUMA) in which a CPU has a faster access to some parts of main mem. than to other parts. Typically, this occurs in a Sys. Containing combined CPU + mem boards. The CPU's on board can access the mem. on that board with less delay than they can access mem. in other boards in the Sys.

If the mem. placement algs & CPU Scheduler work together, then a process that is assigned affinity to a particular CPU can be allocated mem. on the board where CPU resides.

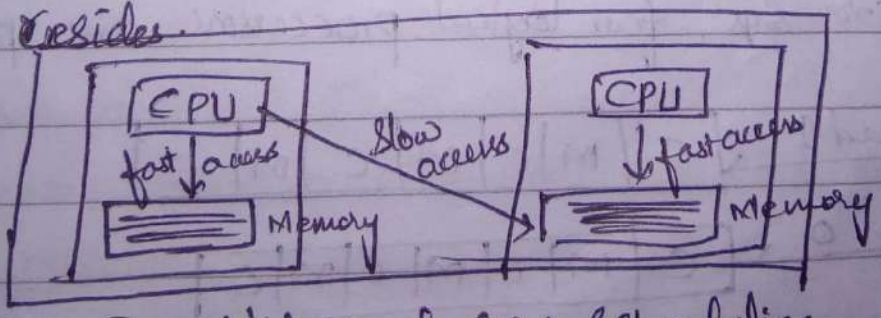


Fig. NUMA & CPU Scheduling

3) Load balancing: On SMP Sys's, it is imp to keep the workload balanced among all processors to fully utilize the

benefits of having more than 1 processor. Otherwise, 1 or more processors may sit idle while other processors have high workload, along with list of processes awaiting for the CPU.

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP Sys. It is imp to note that load balancing is typically only necessary in sys where each processor has its own private queue of eligible processes to execute. On sys's with a common run queue, loads balancing is often unnecessary, because once a processor becomes idle. * Go to Next pg *

(3) * Note: Many recent hlw designs have implemented multithreaded processor cores in which 2 (or more) hlw threads are assigned to each core. That way if one thread stalls while waiting for mem., the core can switch to another thread. Fig. represents a dual threaded processor core on which the execution of thread 0 & thread 1 are interleaved.

From an OS each hlw thread appears as "logical processor that is available to run a sw thread. Thus on a dual threaded dual core Sys, four logical processors are presented to the OS.

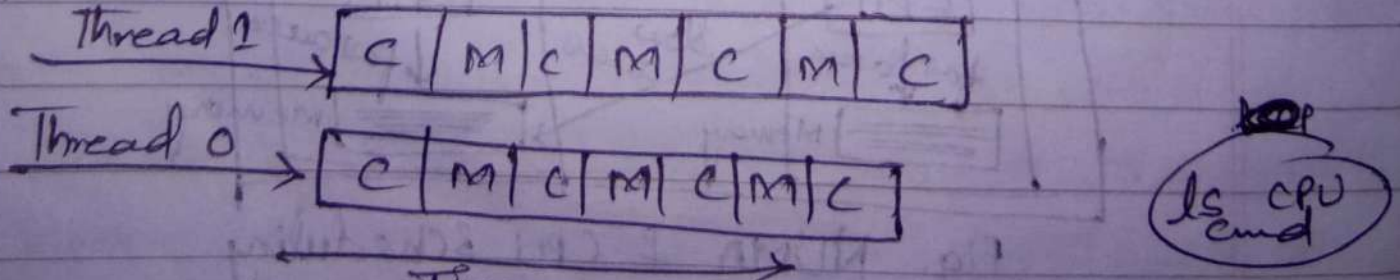


Fig. Multithreaded Multicore System (3) *

There are 2 ways to multi-Thread a processor.
i) Fine grained multithreading, ii) Coarse grained Multithreading

Coarse Grained Multithreading: A Thread executes on a processor until a long-latency event such as mem. stall occurs. Because of the delay caused by long latency event, the processor must switch to another thread to begin execution. However the cost of switching blw threads is high, as the instruction pipeline must be flushed before the other Thread can begin execution on the processor core. Once the new thread begins execution, it begins filling the pipeline with its instructions.

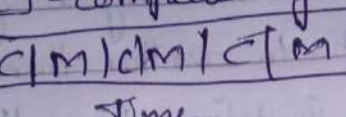
Fine Grained (Interleaved) Multithreading: Switches blw the threads at a much finer level of granularity typically at the boundary of an instruction cycle. However the architectural design of fine grained sys's include logic for threads switching. As a result the cost of switching blw threads is small.

✓ Switching is done after every cycle of CPU.

Multi-Core Processors: SMP sys's have allowed several threads to run concurrently by providing multiple physical processors. However a recent trend in computer hlw has been to place multiple processor cores on the same chip, resulting in a multi-core processor

- Each core has a register set to maintain its architectural state & thus appears to the OS to be a separate physical processor.
- SMP sys's that use multicore processors are faster & consume less power than sys's in which each processor has its own physical chip.

✓ Researchers discovered that when a processor accesses mem, it spends a significant amt of time (say 50%.) waiting for the data to be available. This situation is called as mem. stall & may occurs due to various resons like cache Miss. [C] - compute cycle [M] Mem Stall



It immediately extracts a runnable process from the common run queue. It is also imp to note that in most contemporary os; Supporting SMP, each processor does have a private queue of eligible processes.

* There are 2 general approaches to load balancing:

i) Push Migration: A specific task periodically checks the load on each processor & if it finds an imbalance - evenly distributes the load by moving (or pushing) processes from overloaded to idle or less busy processors.

ii) Pull Migration: Occurs when an idle processor pulls awaiting task from a busy processor.

Interestingly, load balancing often counteracts the benefits of processor affinity. Pushing or Pulling a process from one processor to another invalidates the benefit. Ideally there is no absolute rule concerning which policy is best. In some systems an idle processor always pulls a process from a non-idle processor; & in other systems, processes are moved only if the imbalance exceeds a certain threshold.

Interprocess Communication (IPC)

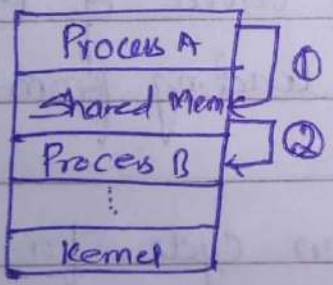
* Communication btw 2 or more processes is called as Interprocess Communication

Purpose of Interprocess Communication:

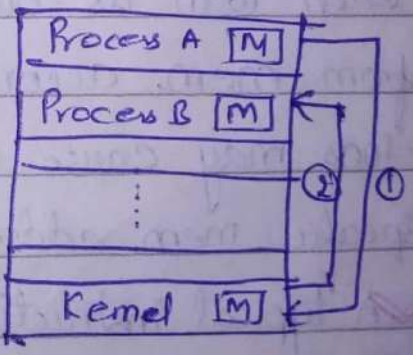
1. Data Transfer
2. Information Sharing
3. Computation Speedup
4. Modularity
5. Resource Sharing
6. Process Control
7. Synchronization
8. Convinience & etc.

* There are 2 fundamental models of Interprocess Comm.
1) Shared Mem. & 2) Message Passing

1) **Shared Mem: Model:** In this model, a region of mem. is shared by 2 or more processes. Processes can exchange inf by reading & writing data to the shared region. (It ~~immediately~~ ~~extracts~~ a ~~runnable process from the common RAM~~)



Shared Memory



Message Passing

Message Passing Model: Communication takes place by means of msgs exchanged btw the cooperating processes

✓ Msg passing is useful for exchanging smaller amounts of data & is easier to implement.

✓ Shared Mem. allows max. speed & convenience of communication. Shared Mem. is faster than msg passing.

Unit 2

Date / /
Page

MEMORY MANAGEMENT

* The main purpose of a computer sys is to execute prgs. These prgs, together with the data they access, must be at least partially in main mem. during execution.
* To improve both the utilization of the CPU & the speed of its response to users, a general purpose computer must keep several processes in memory. Many mem-mgnt schemes exist, reflecting various approaches, & the effectiveness of each alg depends on the situation. Selection of a mem-mngt scheme for a system depends on many factors, especially on the hlw design of the sys. Most algs require hlw support.
* Mem is central to the operation of a modern computer sys. Mem. consists of a large array of words (or bytes, each with its own address. The CPU fetches instructions from mem. according to the value of PC. These instructions may cause additional loading from & storing to specific mem. addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from mem. The instruction is then decoded & may cause operands to be fetched from mem. After the instruction has been executed on the operands, results may be stored back in mem. The mem unit sees only a stream of mem addresses; (it does not know how they are generated by the instruction counter, indexing, literal addresses, & so on) or what they are for (instructions or data).

Accordingly, we can ignore how a prg generates a mem address. We are interested only in the sequence of mem addresses generated by the running prg.

* **Basic Hardware:** Main mem. & the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take mem. addresses as arguments, but none that take disk addresses. Therefore, any instruction in execution, & any data being used by the instructions must be in one of these direct-access storage devices. If the data are not in mem., they must be moved there before the CPU can operate on them.

✔ Registers that are built into the CPU are generally accessible within 1 cycle of the CPU clock. Most CPU's can decode instructions & perform simple operations on register contents at the rate of 1 or more operations per clock tick. The same can't be said of main mem., which is accessed via a transaction on memory bus. Completing a mem. access may take many cycles of CPU clock. In such cases a processor normally needs to stall, since it doesnot have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of mem. accesses. The soln is to add fast mem. btw the CPU & Main mem. A mem. buffer is used to accommodate a speed differential, called cache.

Date / /

Page:

Fig: Different Addresses during compilation

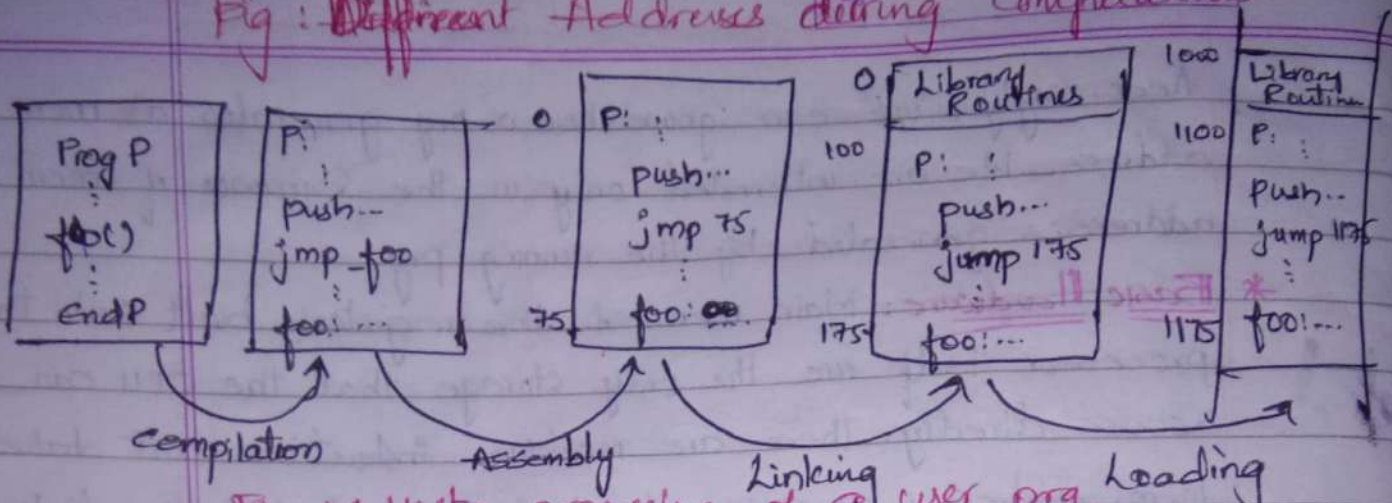
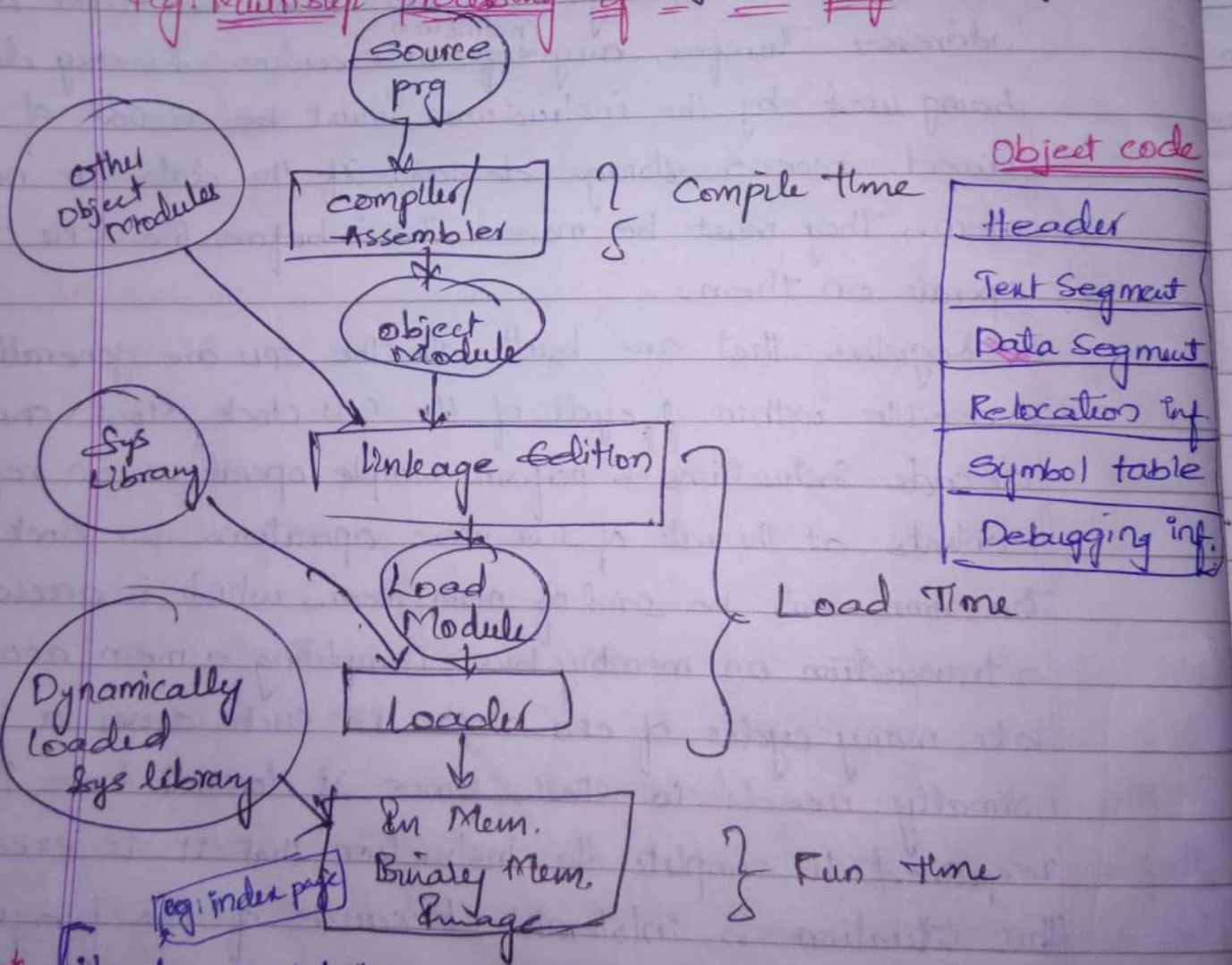


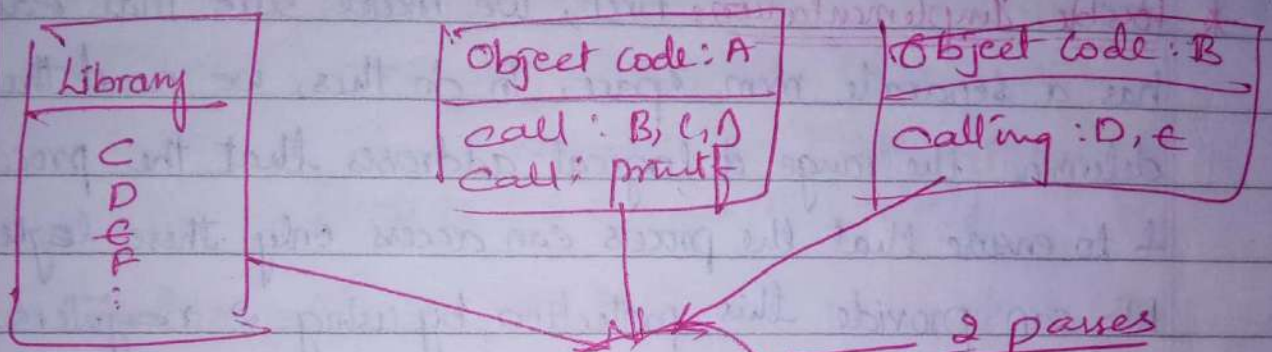
Fig: Multistep processing of a user prg.



* [Header maintains all inf abt where text segement is present, where Data Segment is present, and so on]
* Symbol Table: Wt all var. declared & wt all process/fns

that are declared, & wt all procesors been called.

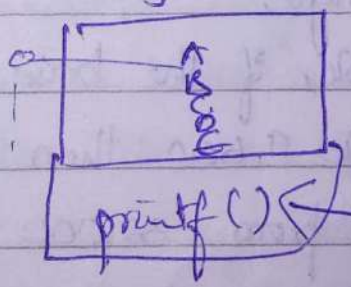
* Debugging: eg: gdb → Var change



2 passes
I ① Segment table
② Symbol table
II ① resolve unresolved symbols.
② static linking
Stub (or) glue code

linker:
Symbol resolution + Relocation

loader
Prg loading + Relocation



* compile time:
Symbolic names → Relocatable address

Link Time:
Relocat address → Reloeate address

Load Time:
Relocate addr → absolute Address.

Run Time: Process can be moved around (swap in & swap out)

* Not only we are concerned with the relative speed of accessing physical Mem., but also we must ensure correct operation to protect the OS from access by user processes & in addition, to protect user processes from 1 another. This protection must be provided by hlw. It can be implemented in several ways.

* Possible Implementation: First, we make sure that each process has a separate mem. space. To do this, we need the ability to determine the range of logical addresses that the process may access & to ensure that the process can access only these legal addresses. We can provide this protection by using 2 registers, usually a base & a limit. The base register holds the smallest legal physical mem. address; the limit register specifies the size of the range.

For example, if the base register holds 32,000 & the limit register is 9,600 then the prg can legally access all addresses from 32,000 through 41599 (inclusive).

Protection of mem. space is accomplished by having the CPU hlw compare every address generated in user mode with the registers. Any attempt by a prg executing in user mode to access OS mem. or other users/process's mem. results

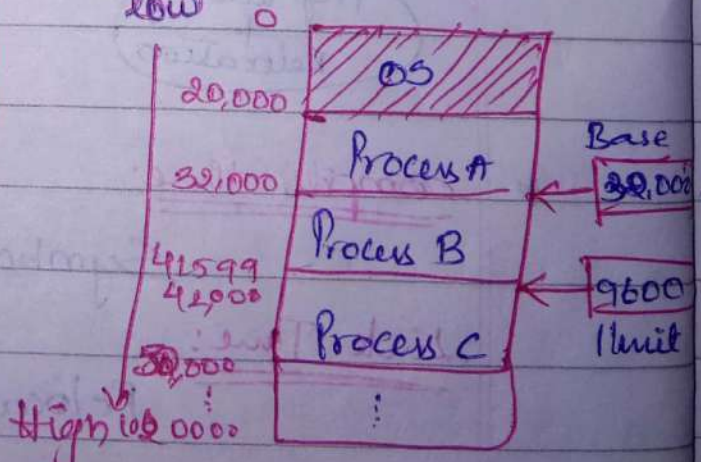


Fig: Base reg. & limit reg. define logical addr. space.

in a trap to the OS, which treats the attempt as a fatal error. This scheme prevents a user prg from accidentally or deliberately modifying the code or data structures of either the OS or other users.

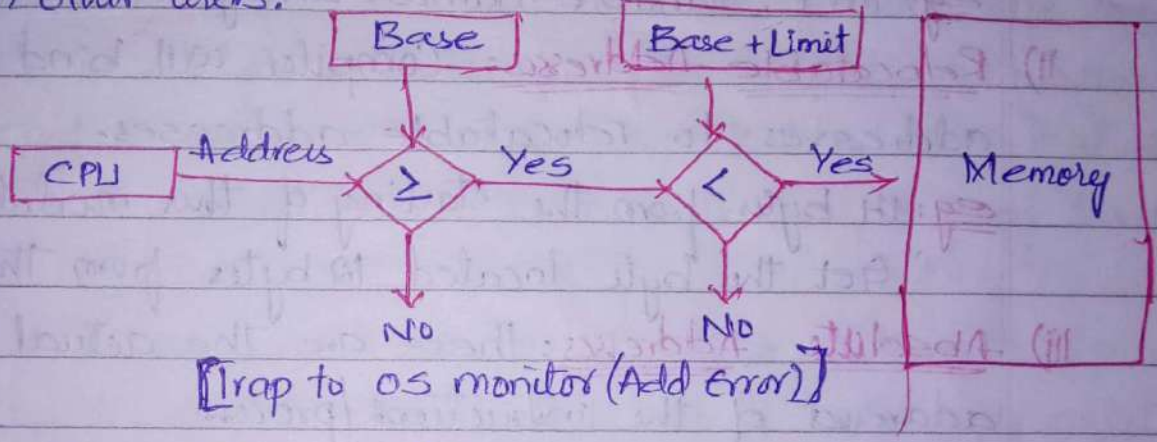


Fig: Hw address protection with base & limit registers.

The base & limit registers can be loaded only by the OS, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, & since only the OS executes in kernel mode, only the OS can load the base & limit registers. This scheme allows the OS to change the values of the registers but prevents user progs from changing the registers contents.

+ **Address Binding:** Most systems allow a user process to reside in any part of physical mem. Thus, although the addr space of computer starts at 00000, the first addr. of the user process need not be 00000. This approach affects the addresses that the user progs can use. In most cases, a user prg will go through several steps, some of which may be optional before being executed. Addresses may be represented in diff ways during these

Date / /
Page

Steps.

i) Symbolic Addresses: In assembly lang. source code it is convinient to use names for mem locations. These names are called as symbolic addresses. (Title -data, -word, .code .bss etc)
eg: In C, variable names, array names, etc.

ii) Relocatable Addresses: Compiler will bind symbolic addresses to relocatable addresses.
eg: 14 bytes from the starting of this module
"Get the byte located 12 bytes from this instruction".

iii) Absolute Addresses: These are the actual physical addresses of the instructions/process.
eg: "Get the byte located at address 255".

* The linkage editor or loader will bind the relocatable addresses to absolute addresses.
(fig (Backside) Multistep processing of a user prog.)

* The binding of instructions & data to mem. addresses is called as address binding & can be done at any step along the way.

i) Compile Time: If you know at compile time, where the process will reside in mem., then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location & extend up from there. If at later time, the starting location changes, Then it will be necessary to recompile this code.

assembly lang

Stack
Static data
int, float, double
addres

We know
e int i/
change
float
We compile

The MS-DOS .com format prgs are bound at compile time.

2) Load Time: It is not desirable to decide is advance into which region of mem, the load module must be loaded, so we now defer that decision to load time. The module does not have absolute addresses, but address that are relative to some known point such as start of the prg module.

eg. pointers in heap may run out of mem, or mem is allocated from heap. Need to generate relocatable code. Less reliable until loaded we dont know whether works or not.

If it is not known at compile time, where the process will reside in mem, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need to only reload the user code to incorporate this changed value.

3) Execution Time: If the process can be moved during its execution from one mem. segment to another, then the binding must be delayed until run time. Special hw must be available for this scheme to work.

eg: Java follows it ← Dynamically bound lang. Base & limit registers

- Until the obj gets loaded, its not get loaded till runtime
- After creating Obj, the mem for the vars is created

C - statically bounded lang.

* Logical vs Physical address space: The address generated by CPU is commonly referred as a logical address,

whereas an address seen by the memory unit - that is, the 1 loaded into the mem-address register of the mem- is commonly referred to as physical address.

* Mem. Mngt unit only can see the physical address space.

Eg: Lets say your sys has 2GB of physical mem. (RAM) divided into words of size 8 bytes each. So every word is assigned an address. This actual address is called as physical address. At this stage, we are blind to those physical addresses.

C Program:

```
#include<stdio.h>
int main()
{
  pf("location of code: %p\n", (void) main);
  pf("location of heap: %p\n", (void) malloc(1));
  int x=3;
  pf("location of stack: %p\n", (void) &x);
  return x;
}
```

The o/p could be in hexadecimal values, which are called as virtual addresses.

✓ Any address your CPU or you can see as a programmer of a user-level prg is a virtual address. It's only the OS through its tricky techniques of virtualizing mem., that knows where in the physical mem. of the machine these instructions & data lie. So if you print out an address in the prg, its a virtual one.

eg:
heap
Sta
dat
Cod

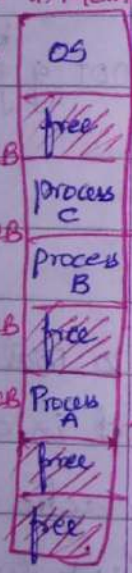
eg: 3
0KB
1000KB
2000KB
3000KB
4000KB

Date / /
Page

eg:

Why are physical addresses abstracted to user prgs. To make switching b/w the processes faster. If the entire process resides on physical mem, switching processes may take time as entire process need to be pushed to disk during switching & new process to be loaded, which may become very slow if processes are large. Generally hard disk can be used as swap space for creating virtual mem.

eg: 3 Processes in Mem.



Consider a process A, the fig. represents logical address space for process A. This space belongs to only process A. There may be similar logical address spaces for process B invisible to A. So process A thinks that it owns entire physical mem. In reality it owns only a small part of it. It's the duty of OS to map the logical address of diff processes to actual physical addresses.

The image shows 3 processes A, B, C sharing the physical address space without bumping into 1 another. Now if we look at logical address space, we see that, "Every process thinks that it has been loaded at mem. address 0 KB. But actually process A, B, C are loaded at 4000, 2000, 1000 KB respectively.

The compile time & load time address binding methods generate identical logical & physical addresses. However, the execution time address binding scheme results in different logical & physical addresses.

The set of all logical addresses generated by a prg is called logical address space. The set of all physical addresses corresponding to these logical address is called phy addr. space

Date / /
Page

* Physical & logical address:

The address generated by CPU is commonly refferred as a logical address, whereas an address seen by the mem. unit, that is, the one loaded into the mem. address register of the mem. is called as Physical address.

As we have 2 diff. addresses here & the actual mem. (RAM) is represented by physical addresses. There shld be some mapping btw logical & physical addresses.

* Physical Address Space (RAM):

Eg: Consider 1 GB RAM

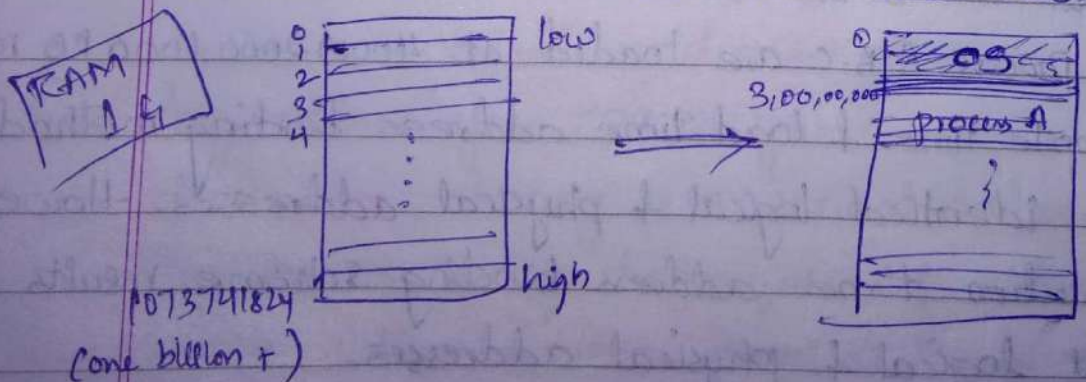
1 GB of RAM is actually 1 GiB (Gibibyte, not giga byte)

The difference is

$1\,GB = 10^9\,B = 1000\,000\,000\,B$

$1\,GiB = 2^{30}\,B = 1073\,741\,824\,B$

Every byte of mem. has its own address, no matter how big the CPU machine word is (Machine word is a amt of mem. CPU uses to hold nos. in RAM, cache or internal registers). 32 bit CPU uses 32 bits (4 bytes) to hold nos. Mem. addresses are nos. too.



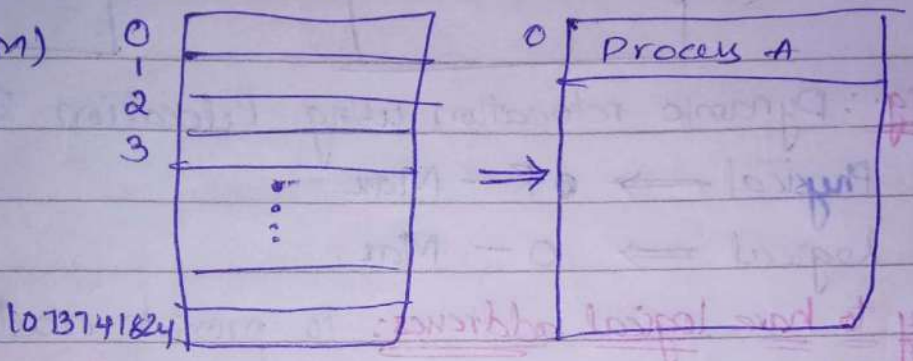
The lower addresses are occupied by the OS.

Set of all physical addresses present in the physical Mem. (or) RAM is called as Physical address Space.

Date / /
Page

* Logical address space:

An address generated by the CPU is commonly known as logical address. That means all the user progs will see only logical addresses & They can't see the actual physical addresses.



Lets consider a process A is executing on the CPU when a process executes on the CPU, it generates addresses from 0-to-max. i.e., the process feels that it is loaded at the 0th address location whereas the process is actually loaded at 3,00,00,000th addres.

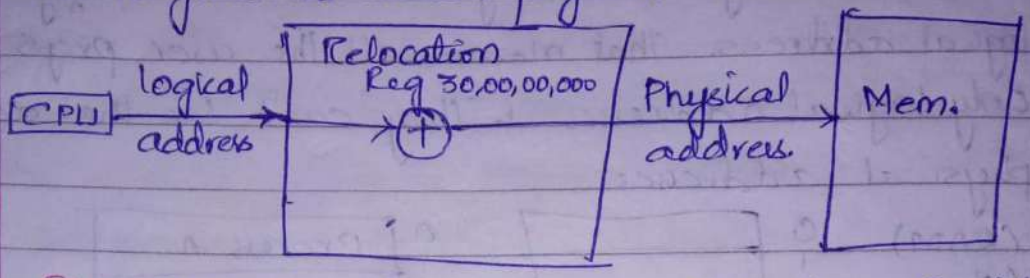
Now, as processor feels that it is loaded at 0th addr, it also have feeling that it can use all the addresses (space) from 0-1073741824 (0-Max). In the same manner each & process feels the same.

As, the process has instructions / data starting at location 3,00,00,000, but it thks that it starts at 0th location. There shld be someone to take care of this. Here comes mem. mngt unit (MMU), which maps the logical addresses to physical addres.

How to map? Just add some base address.

* If CPU generates a logical address of 346, add same base addr. which is called as Relocation Register to it.

then it gives the actual physical address.



Pg: Dynamic relocation using Relocation Register.

Physical ⟶ R – Max

Logical ⟹ 0 – Max.

Why to have logical addresses: To provide abstraction for the user progs. otherwise 1 prog can access the mem/corrupt the mem. of another prog.

* The compile time & load time address binding methods generate identical logical & physical addresses, but execution time binding scheme results in diff logical & physical addrs.

* Dynamic linking: The linking is postponed until execution time. With dynamic linking, a stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate mem. resident library routine or how to load the lib. if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in mem. If it is not, the prog loads the routine into mem. Either way, the stub replaces itself with the addr. of the routine & executes the routine.

Static linking: Sys. libraries are combined by the loader into the binary prog image before execution.

7/8/17

211: 1, 5, 6, 8, 10, 12, 13, 14, 22, 24, 30, 33, 42, 52 ✓

013: 1, 13, 14, 18, 25, 24, 32, 43, 45, 46, 52, 55, 58, 59, 60, 61, 63, 64;

210: 1, 2, 5, 6, 7, 9, 12, 13, 14, 15, 16, 17, 26, 28, 29, 37, 40, 44, 46, 54, 56, 57, 63, 65 ✓

8/8/17

210: 7, 9, 12, 16, 26, 62

013: 1, 13, 16, 24, 43, 56, 59, 60, 61, 66

209: 10, 15, 30, 36, 54, 65

9/8/17

211: 8, 9, 23, 24, 26, 32, 42, 50, 56, 57, 61

209: 59, 60, 62 ✓

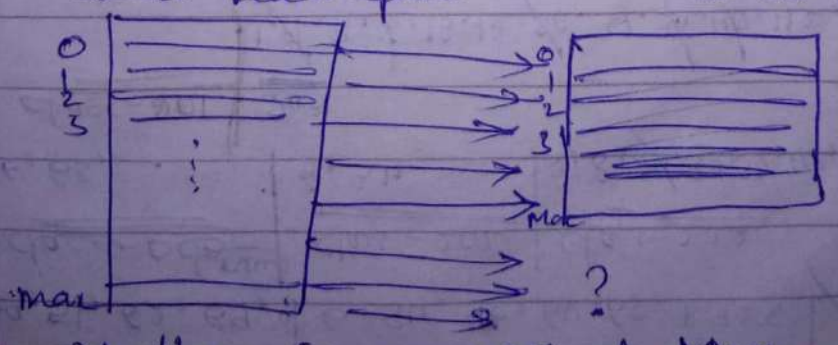
209: 10, 17, 18, 22, 24, 25, 28, 29, 31, 32, 33, 36, 46,

Shared Libraries: A library may be replaced by a new version & all prgs that reference the library will automatically use the new version. By this there could be incompatiability for older prgs. Each library & each prg stores version inf. & with dynamic linking the corresponding libraries required for the prg can be linked.

* **Dynamic loading:** It's necessary for the entire prg & all data of a process to be in physical mem. for the process to execute. The size of the process has been limited to the size of physical mem. To obtain better mem. space utilization, we can use dynamic loading a routine is not loaded until it is called. All routines are kept in disk. This main prg is loaded into mem. & is executed.

Virtual Mem.: (Not real) Without virutal mem. prg address space = RAM address space.

Eg: 4 GB process (32 bit addr space) — 1 GB RAM (30 bit addr space)



If there is no virtual Mem, Sys crashes if we try to access more RAM than we have.