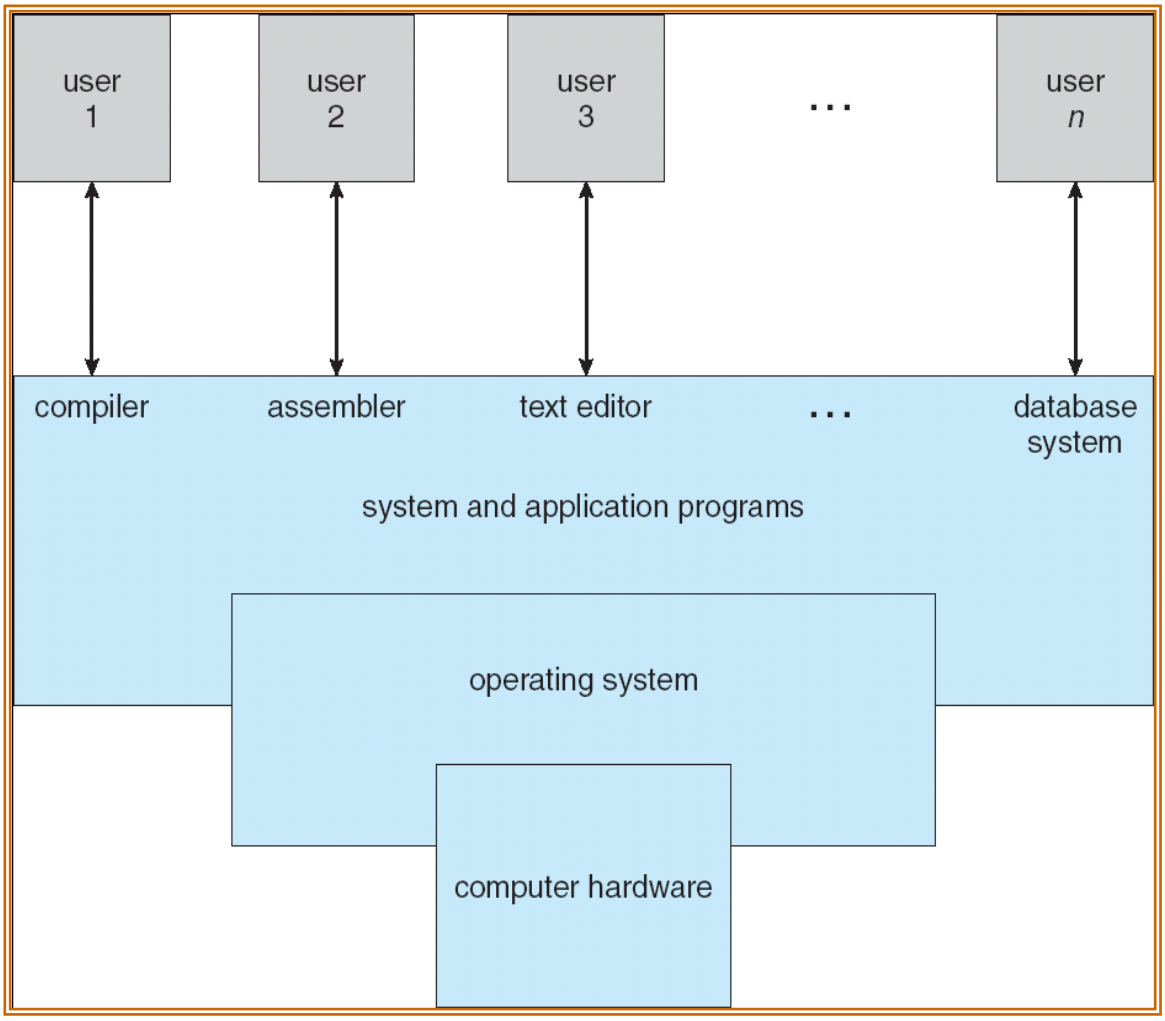
Unit - 1

An Operating system is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between a user of a computer and the computer hardware.

O.S. are designed to provide an environment in which a user can easily interface with the computer to execute programs.

Components of a computer system : Computer system can be divided roughly into four components : the hardware,the operating system,the application programs and the users.



The hardware - the central processing unit(CPU),the memory and the input/output devices - provides the basic computing resources. The application programs-such as word processor,spreadsheets,compilers and web browsers.

The O.S. controls and coordinates the use of the hardware among the various application programs for the various users.

The O.S. provides the means for the proper use of these resources in the operation of the computer system.

Each O.S. is discussed with regard to the following aspects :

I. Processor scheduling
II. Memory management
III. I/O management
IV. File management

Types of O.S. :-

(a.) Batch Operating Systems: - Batch processing generally requires the program, data and appropriate system commands to be submitted together in the form of a job.

Programs that do not require interaction and program with long execution times may be served well by a batch operating system. Ex :-payroll, forecasting, statistical analysis.

Scheduling in batch system is very simple. Jobs are typically processed in the order of submission that is first-come first-served fashion.

Memory Management in batch systems is also very simple. Memory is usually divided into two areas. One of them is permanently occupied by the resident portion of the operating system, and the other is used to load transient programs for execution. When the transient program terminates, a new program is loaded into the same area of memory.
Batch systems often provide simple form of file management. Since access to files is also serial, little protection and no concurrency control of file access is required.

(b.) Multiprogramming Operating Systems :-

A multitasking operating system is distinguished by its ability to support concurrent execution of two or more active processes.

Multitasking is usually implemented by maintaining code and data of several processes in memory simultaneously.

Multiuser O.S. provides facilities for maintenance of individual user environments, require user authentication for security and protection, and provide per-user resource usage accounting.

Multitasking operation is one of the mechanisms that a multiprogramming operating system employs in managing the totality of computer-system resources, including processor, memory and I/O devices. Multitasking operation without multiuse support can be found in operating system of some advanced personal computers and in real time system.

Multi-access O.S. allows simultaneous access to a computer system through two or more terminals. An example is provided by some dedicated transaction-processing system, such as airline ticket reservation systems.

Multiprocessing or multiprocessor operating systems manage the operation of computer systems that incorporate multiple processors. Multiprocessor O.S. is multitasking system by definition because they support simultaneous execution of multiple tasks on different processors.

Time-Sharing Systems :- Time sharing is a logical extension of multiprogramming. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

A time-shared OS allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

Time-sharing systems must also provide a file system. The file system resides on a collection of disks. Hence, disk management must concurrent execution, which requires sophisticated CPU-scheduling schemes.

Multiprocessor Systems :- multiprocessor system also known as **parallel systems** have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices.

multiprocessor systems have three main advantages :-

1. Increased throughput: By increasing the number of processors, we hope to get more work done in less time.
2. Economy of scale : multiprocessor systems can save more money than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processor share them.
3. Increased reliability: If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining processors must pick up a share of the work of the failed processor.

Distributed Operating Systems: A distributed computer system is a collection of autonomous computer systems capable of communication and cooperation via their hardware and software interconnections. Distributed computer systems evolved from computer networks in which a number of largely independent hosts are connected by communication links and protocols.

Distributed OS usually provide the means for system-wide sharing of resources, such as computational capacity, files, and I/O devices. A distributed OS may facilitate access to remote resources, communication with remote processes and distribution of computations.

Advantages:

- resource sharing
- computation speed-up
- reliability
- communication - e.g. email

Applications - digital libraries, digital multimedia

Real-Time Systems: - A real time system is used when rigid (inflexible) time requirements have been placed on the operation of a processor or the flow of data. Thus, it is often used as a control device in a dedicated application. Processing must be done within the defined constraints or the system will fail.

A primary objective of real-time system is to provide quick response times. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs.

System that control scientific experiments, flight control, medical imaging systems, industrial control system and certain display systems are real-time systems.

Operating-system provides following functions that are helpful to the user:

a. User interface - Almost all operating systems have a user interface (UI) - Varies between Command-Line (CLI), Graphics User Interface (GUI)

b. Program execution - The system must be able to load a program into memory and to run that program.
c. I/O operations - A running program may require I/O, which may involve a file or an I/O device.
d. File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them.
e. Communications – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing.
f. Error detection – OS needs to be constantly aware of possible errors.

    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

g. Resource allocation - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
h. Accounting - To keep track of which users use how much and what kinds of computer resources
i. Protection and security - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

- **Protection** involves ensuring that all access to system resources is controlled
- **Security** of the system from outsiders, requires user authentication.

TYPE OF SERVICES

User View

Operating system services are provided in many different ways. Two method of providing services are

- system calls and
- system programs.

System Calls:

System calls provide the interface between a running program and the operating system.

- Generally available as assembly-language instructions.
- Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)

Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

System Programs

- System programs provide a convenient environment for program development and execution. The can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls.

Operating System View

- The view of an operating system seen by the user is defined mainly by the system programs particularly the command interpreter.
- The interrupt driven nature of an operating system defines the general structure. When an interrupt occurs, the hardware transfers control to the operating system.
- Several different types of interrupts may occur:
  - A system call
  - An I/O device interrupt
  - A program error

Unit - 2

Process Scheduling :- A scheduler is an O.S. program that selects the next job to be admitted for execution. The main objective of scheduling is to increase CPU utilization and higher throughput.
[throughput - is the amount of work accomplished in a given time interval]
CPU scheduling is the basis of O.S. which supports multiprogramming concepts.

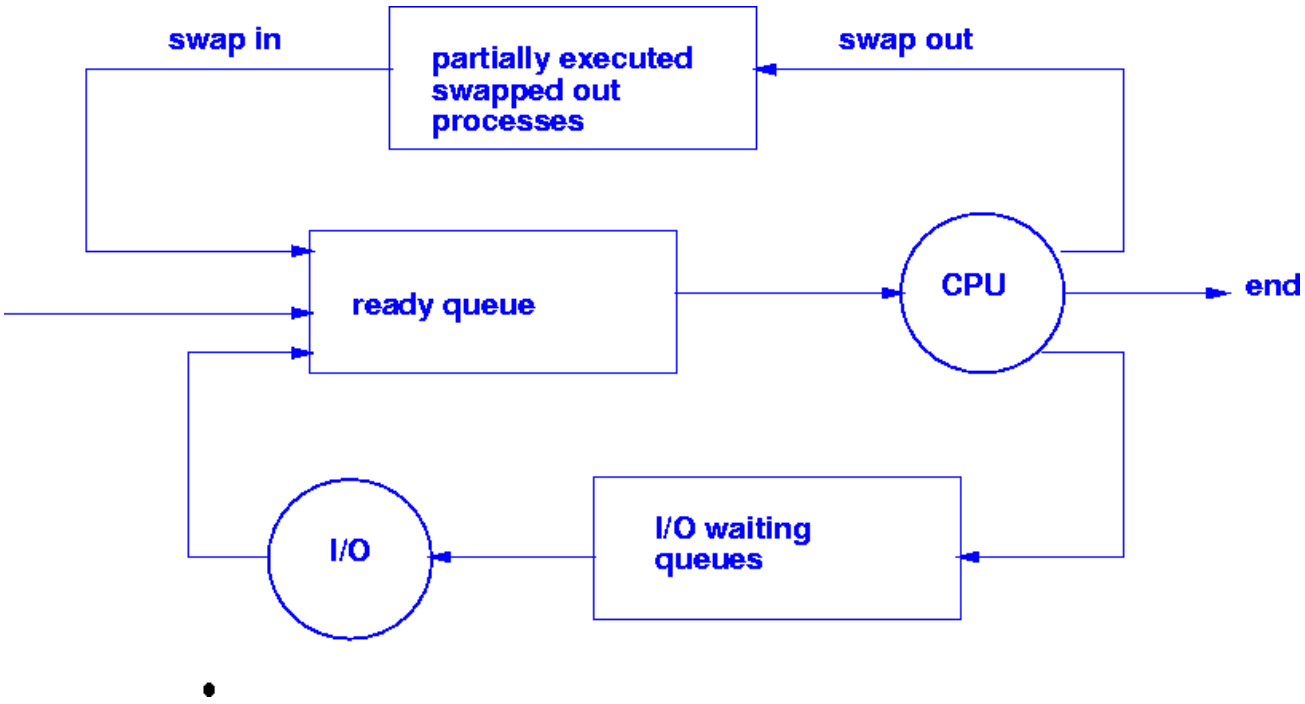
Nonpreemptive Scheduling

A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.

Types of Scheduler :-



- Long-term scheduler (or job scheduler) –
  - selects which processes should be brought into the ready queue.
  - load processes from secondary storage device into the memory.
  - invoked very infrequently (seconds, minutes); may be slow.
  - controls the "degree of multiprogramming"(the no of processes in memory).

- Short term scheduler (or CPU scheduler) -
  - selects which process should execute next and allocates CPU.
  - invoked very frequently (milliseconds) - must be very fast
  - Its main objective is maximize cpu requirement.

- Medium Term Scheduler
  - swaps out process temporarily
  - Balances load for better throughput.

Scheduling and Performance Criteria :-

- CPU utilization – keep the CPU as busy as possible
- Throughput – no of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process, It is sum of the periods spent waiting to get into memory, waiting in the ready queue, CPU time and I/O operations.
  (Turnaround Time = waiting time + processing time)
- Waiting time – amount of time a process has been waiting in the ready queue
  (waiting time = Turnaround Time - processing time)
- Response time – amount of time it takes from when a request was submitted until the first response is produced. (***not*** *output for time-sharing environment*)

Scheduling Algorithms :- CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. A major division among scheduling algorithms is that whether they support preemptive or non-preemptive scheduling discipline.

Following are some scheduling algorithms : -

- FCFS Scheduling.
- SJF Scheduling.
- SRTF or SRTN Scheduling.
- Priority Scheduling.
- Round Robin Scheduling.
- Multilevel Queue Scheduling.
- Multilevel Feedback Queue Scheduling.

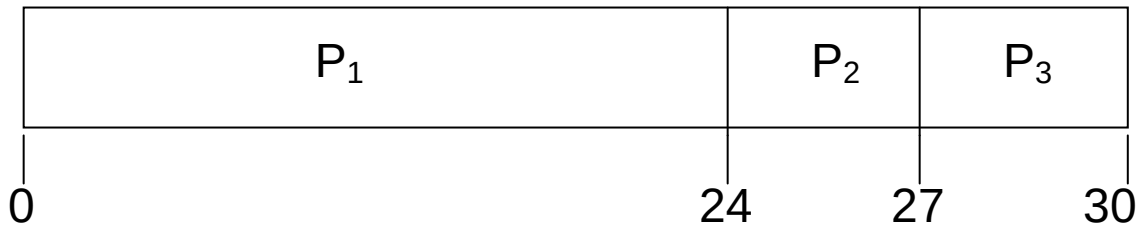
FCFS Scheduling :-

First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a non-preemptive discipline, once a process has a CPU, it runs to completion.

FCFS scheduling is non-preemptive, there is a low rate of components utilization and system throughput. Consider the following example of three processes :

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1* , *P2* , *P3* The Gantt Chart for the schedule is:



Waiting time for *P1* = 0; *P2* = 24; *P3* = 27

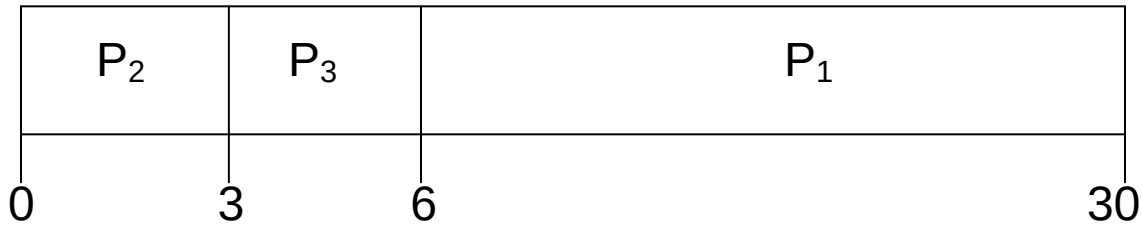
Average waiting time: (0 + 24 + 27)/3 = 17

Avg. Turn around time = [(0+24) + (24+3) + (27+3)]/3

= 81 / 3 = 27

Suppose that the processes arrive in the order : *P2* , *P3* , *P1*

The Gantt chart for the schedule is:



* Waiting time for *P1* = 6*; P2* = 0*; P3* = 3

* Average waiting time: (6 + 0 + 3)/3 = 3

Avg turn around time = [(6+24) + (0+3) + (3+3)]/3

= 39/3 = 13

* Much better than previous case

Shortest-Job-First(SJF) Scheduling :- In SJF scheduling a process is done on the basis of its having shortest execution time. If two processes have the same CPU time, FCFS is used.

Threre are two schemes:

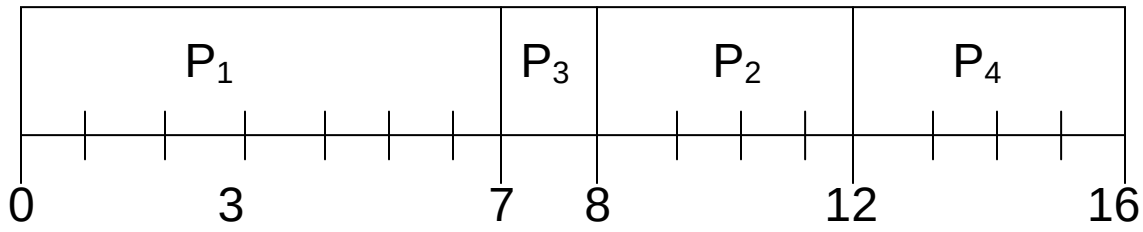
* nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst

* preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)

Example of Non-Preemptive SJF :

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4

P3	4.0	1
P4	5.0	4



Average waiting time =(0 + (8-2) + (7-4) + (12-5))/4

= (0 + 6 + 3 + 7)/4 = 4

Avg turn around time = (7+ 10+4+11) / 4 = 32/4 = 8

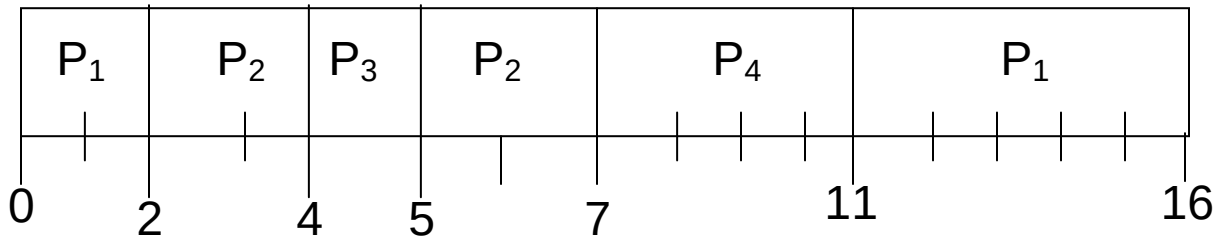
Shortest Remaining Time First(SRTF or SRTN) Scheduling :

- The SRT is the preemtive counterpart(equal) of SJF and useful in time-sharing environment.
- In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
- The algorithm SRT has higher overhead than its counterpart SJF.
- In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

Example of Preemptive SJF or SRTF :

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4

P3	4.0	1
P4	5.0	4



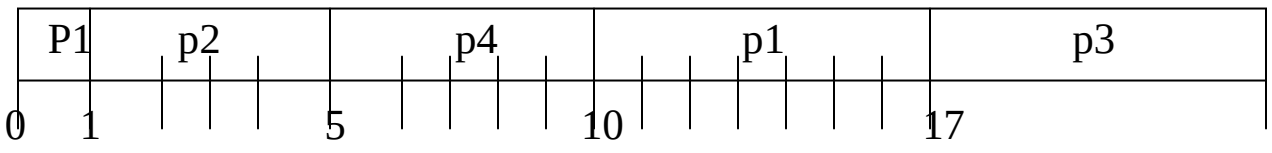
Average waiting time =((11-2) + (5-4) + 0 + (7-5))/4

= (9 + 1 + 0 +2)/4 = 3

Avg turn around time = (16+5+1+6)/4 = 28/4 = 7

Question : consider four processes with the length of the CPU burst time given in milliseconds :

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



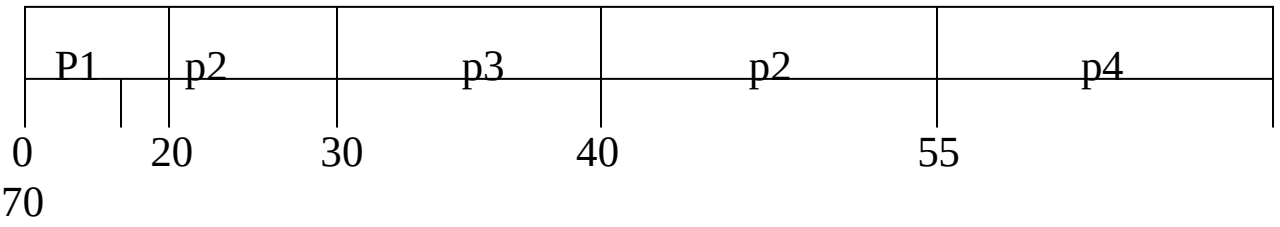
Average waiting time =((10-1) + (1-1) + (17-2) + (5-3))/4 = (9+0+15+2)/4

= 26/4 = 6.5 milliseconds

Avg turn around time = (17+4+24+7) / 4 = 52/4 = 13

Question : consider four processes with the length of the CPU burst time given in milliseconds :

Process	Arrival Time	Burst Time
P1	0	20
P2	15	25
P3	30	10
P4	45	15



waiting time for p1 = 0

waiting time for p2 = (20 – 15) + (40 – 30) = 15

waiting time for p3 = 0

waiting time for p4 = 10

Avg waiting time = (0 + 15 + 0 + 10) / 4 = 25/4 = 6.25

Priority Scheduling :- A priority is associated with each process and the scheduler always picks up the highest priority process for execution from the ready queue. Equal priority processes are scheduled FCFS.

Note that scheduling in terms of high and low priority , there is no general agreement on whether 0 is the highest or lowest priority. Here,we assume low numbers to represent high priority.

Comparison with SJF :

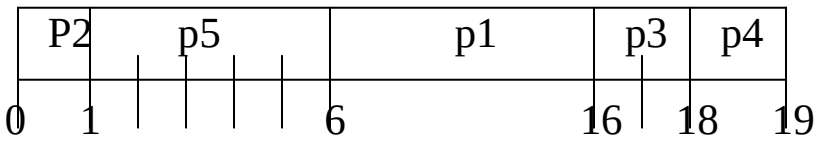
* SJF is a priority scheduling where priority is the predicted next CPU burst time

* Problem ≡ Starvation – low priority processes may never execute

* Solution ≡ Aging – as time progresses increase the priority of the process

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4

P4	1	5
P5	5	2



Average waiting time =(6 + 0 + 16 + 18 + 1)/5 = 41/5 = 8.2

Round Robin Scheduling :- This is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a "time quantum" is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as circular queue.

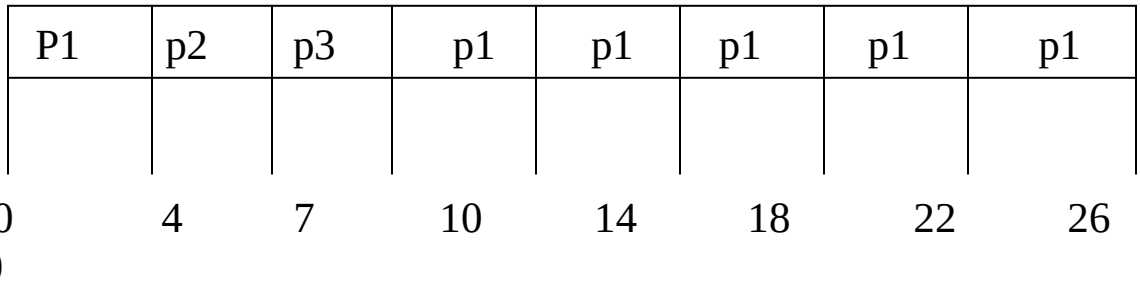
New processes are added to the tail of the ready queue. If the CPU burst of the currently running processes is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue.

The performance of the RR algorithm depends heavily on the size of the time quantum. If the time quantum is very large, the RR policy is the same as FCFS policy. If the time quantum is very small, the RR approach is called processor sharing.

Example of RR with Time Quantum = 4

Process	Burst Time
P1	24

P2	3
P3	3

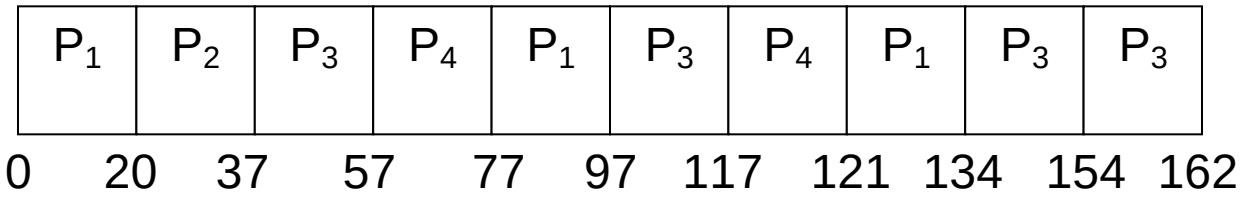


Average waiting time =(6 + 4 + 7)/3 = 5.66 milliseconds.

Question : **Example of RR with Time Quantum = 20**

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

The Gantt chart is:



waiting time for p1 = 17+20+20+20+4 = 81
waiting time for p2 = 20
waiting time for p3 = 37+20+20+4+13 = 94
waiting time for p4 = 20+17+20+20+20 = 97
So, Average waiting time = (81+20+94+97)/4 = 73]]

Multilevel Queue Scheduling :-

[[A situation where processes are easily classified into different groups . for example, a common division is made between foreground(interactive) processes and background(batch) processes. These two types of processes have different response-time requirements and might have different scheduling needs. Foreground processes may have priority over background processes

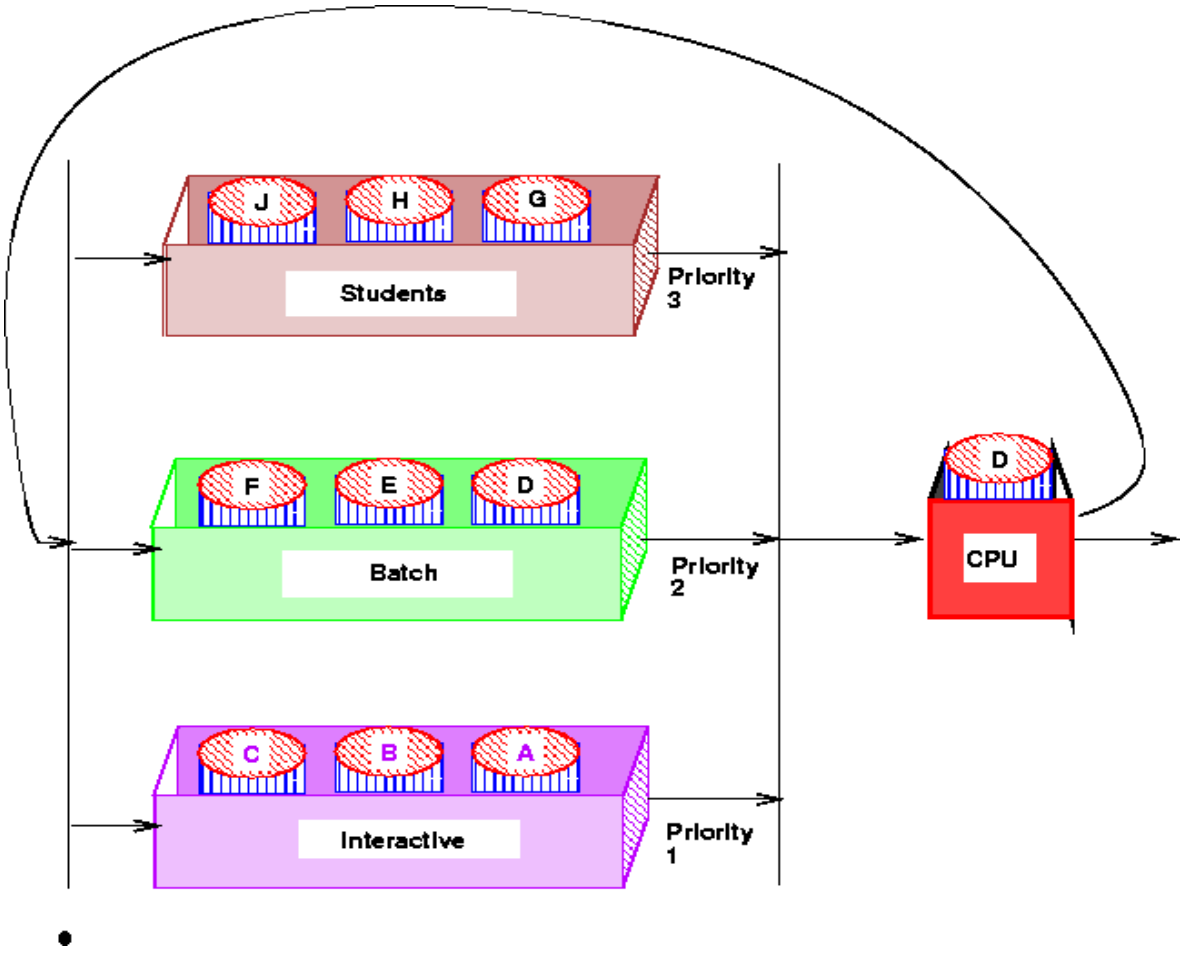
A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size,process priority, or process type. Each queue has its own scheduling algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.]]

*Ready queue is partitioned into separate queues:
>>foreground(interactive)
>>background (batch)

* Each queue has its own scheduling algorithm
>>foreground – RR
>>background – FCFS

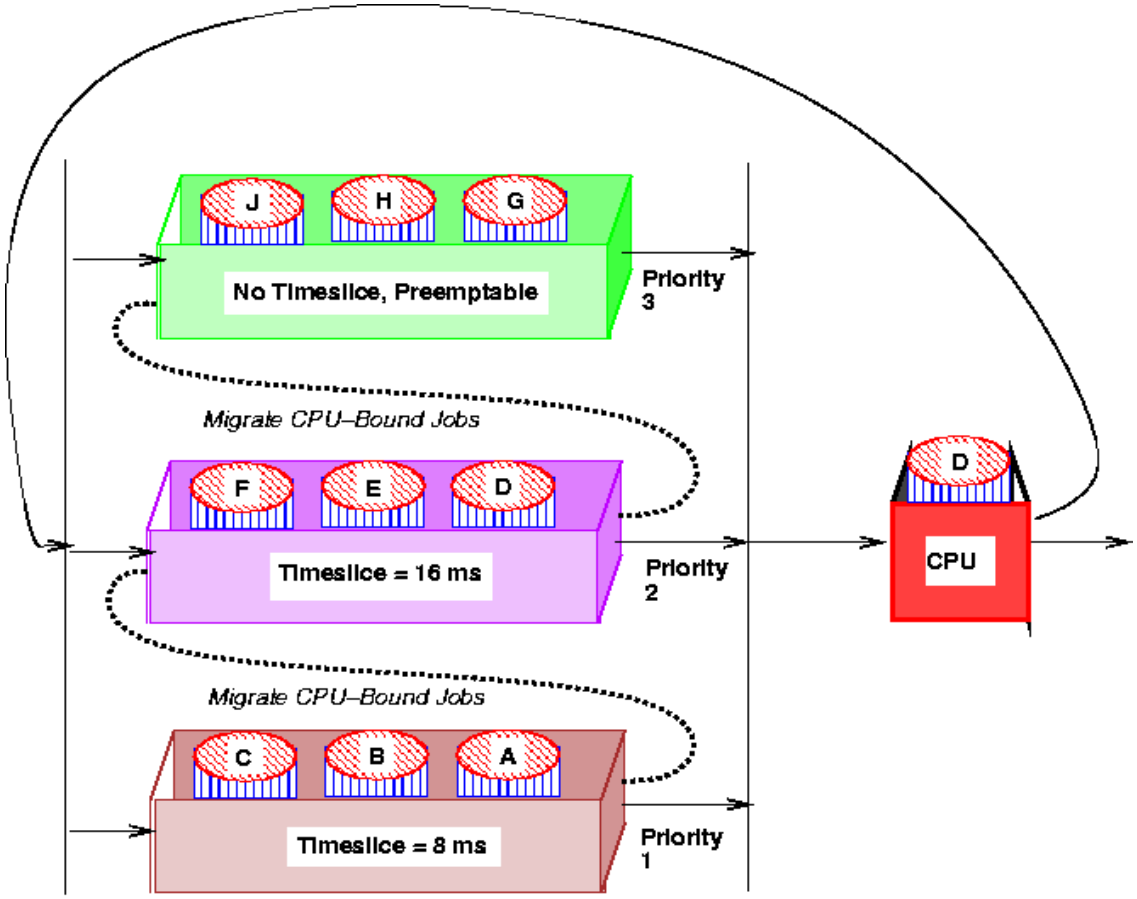
* Scheduling must be done between the queues
>>Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
>>Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
>>20% to background in FCFS



Multilevel Feedback Queue :- [[It allows a process to move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queues. Similarly, 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.]]

- Multilevel Queue with priorities
- A process can *move* between the queues.
  - Aging can be implemented this way.
- Parameters for a multilevel feedback queue scheduler:

- ❑ number of queues.
- ❑ scheduling algorithm for each queue.
- ❑ method used to determine when to upgrade a process.
- ❑ method used to determine when to demote(move down) a process.
- ❑ method used to determine which queue a process will enter when that process needs service.



Unit - 4

Memory Management

Contiguous Allocation

Contiguous Allocation means that each logical object is placed in a set of memory location with strictly consecutive addresses.

The organization and management of main memory has been one of the most important factors of O.S. design.

Memory management is concerned with allocation of main memory of united capacity to requesting processes. No process can ever run before a certain amount of memory is allocated to it.

Two important features of Memory management function are protection and sharing. An active process should never attempt to access incorrectly and destroy the contents of each other's address space.

Bare Machine :- In this scheme, the user is provided with the bare machine and has complete control over the entire memory space. It provides maximum flexibility to the user. The user can control the use of memory in whatever manner desired.

There is no need for special hardware for this approach to memory management, Nor is there a need for O.S s/w. This system has its limitation also.

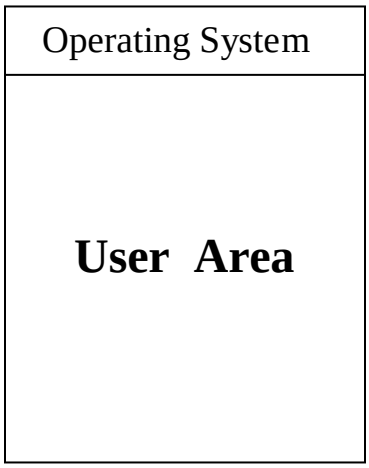
It is used only on dedicated systems where the users require flexibility and simplicity.

Resident Monitor (Single Process Monitor) :-

The next simplest scheme is to divide memory into two sections, one for the user and one for the resident monitor of the O.S.

It is more common to place the resident monitor in low memory. It is primary approach for many current microcomputer systems.

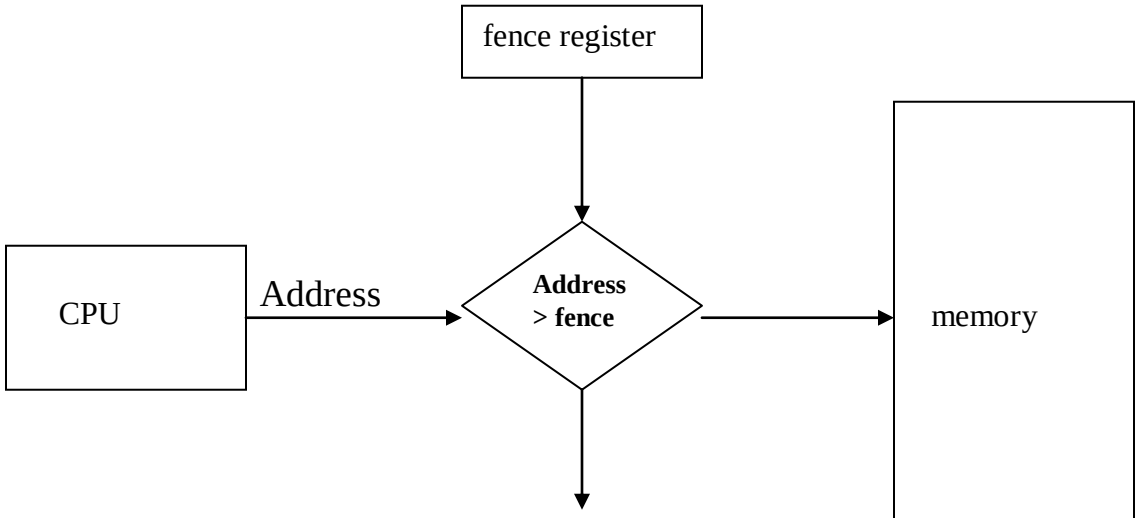
In this type of approach , O.S. only keeps the track of the first and the last location available for allocation of user program.



A new program is loaded only when the O.S. Passes a control to it. When this program is Completed or terminated, the O.S. may load Another program for execution.

Protection is hardly supported by a Single process monitor because only one process Is in memory-resident at a time. A register also Called "fence register" is set to the highest address Occupied by O.S. code. A memory address generated by user program is first compared with fence register's content. If the address is below the fence, it will be trapped and denied permission.

Limitation : less utilization of memory and CPU.



Multiprogramming with fixed partitions :-

Depending upon how and when partitions are created, there may be two types of memory partitioning

(i) static
(ii) Dynamic

Static partitioning implies that the division of memory into number of partitions and its size is made in the beginning and remain fixed there after.

In dynamic partitioning the size and the number of partitions are decided during the run time by the O.S.

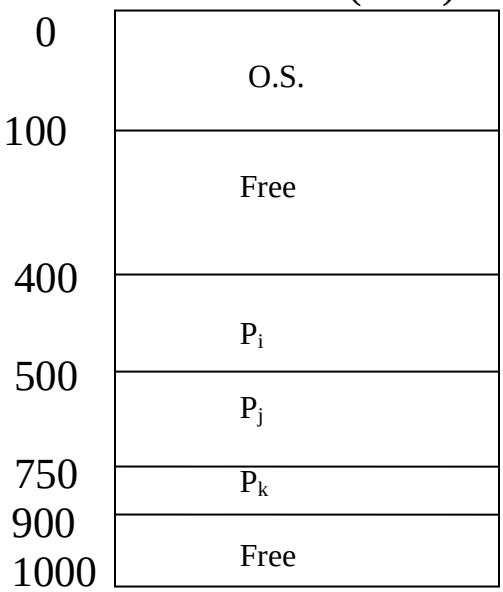
The basic approach here is to divide memory into several fixed size partitions where each partition will accommodate only one program for execution. The no. of programs (degree of

multiprogramming) residing in memory will be bound by the no. of partitions.

When a program terminates, that partition is free for another program waiting in a queue.

Once partitions are defined, an O.S. needs to keep track of their status. Such as free or in use, for allocation purpose. Current partition status and attributes are often collect in a data structure called the "partition description table". Each partition is described by its starting address, size and status.

The identity of the assigned partition may be recorded in the process control block (PCB).



Static partition

0	0 K	100 K	ALLOCATED
1	100 K	300 K	FREE
2	400 K	100 K	ALLOCATED
3	500 K	250 K	ALLOCATED
4	750 K	150 K	ALLOCATED
5	900 K	100 K	FREE

Partition Description Table

Allocation of free partition :

The two most common strategies to allocate free partitions to ready processes are :

(i) first fit

(ii) best fit

	First fit	Best fit
1	The approach followed in the first fit is to allocate the first free partition large	The best fit approach on the other hand allocates the smallest free partition that

	enough to accommodate the process.	meets the requirement of the process.
2	The first fit terminates after finding the first such partition	The best fit continues searching for the near exact size
3	First fit execute faster	The best fit achieves higher utilization of memory by searching the smallest free partition.

Swapping :-

Def : Removing suspended or preempted processes from memory and their subsequent bringing back is called swapping.

Wherever a new process is ready to be loaded into memory and if no partition is free , swapping of processes between main memory and secondary memory is done.

One important issue concerning swapping is whether the process removed temporarily from any partition should be brought back to the same partition or any other partition of adequate size. This is dependent upon partitioning policy.

The swapper is an O.S. process whose major responsibilities include

1. Selection of processes to swap out
2. Selection of processes to swap in
3. Allocation & management of swap space

Relocation :

The term program relocatability refers to the ability to load and execute a given program into an arbitrary place in memory, as opposed to a fixed set of locations specified at program – translation time.

There are two basic types of relocation

i. static relocation
ii. dynamic relocation

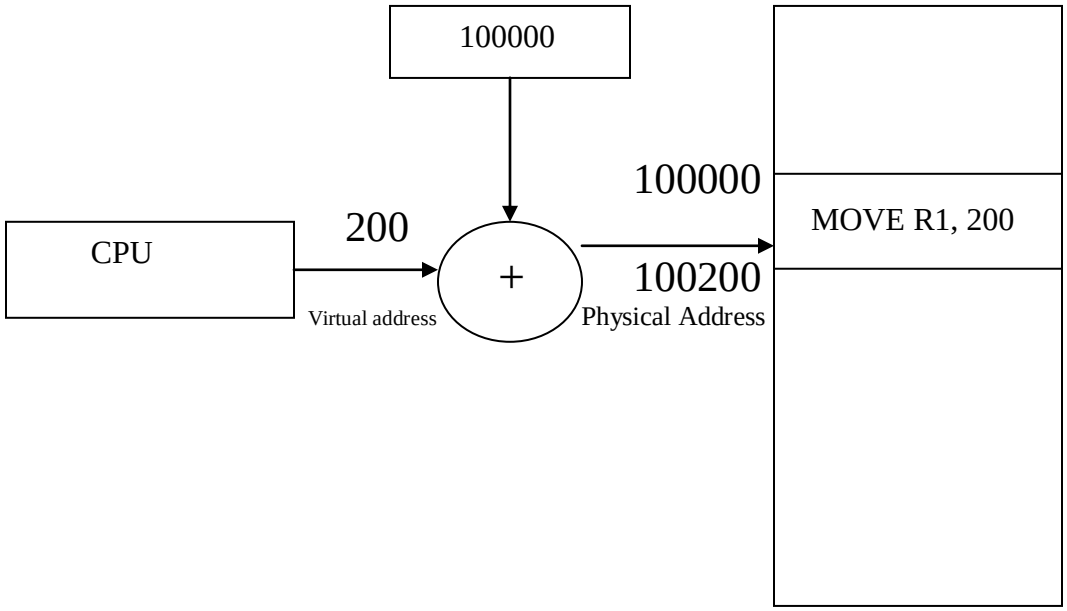
i. If the relocation is performed before or during the loading of a program into memory by a relocating linker or a relocating loader, the relocation approach is called static relocation.

In system with static relocation a swapped-out process must be swapped back into the same partition from which it was evicted

ii. Dynamic relocation refers to run-time mapping of virtual address into physical address with support of some hardware mechanism such as base registers and limit registers.

When a process is scheduled, the base register is loaded with the starting address. Every memory address generated automatically has the base register contents, added to it before being sent to main memory.

The value of base register obtained from relevant entry of the partition description table.



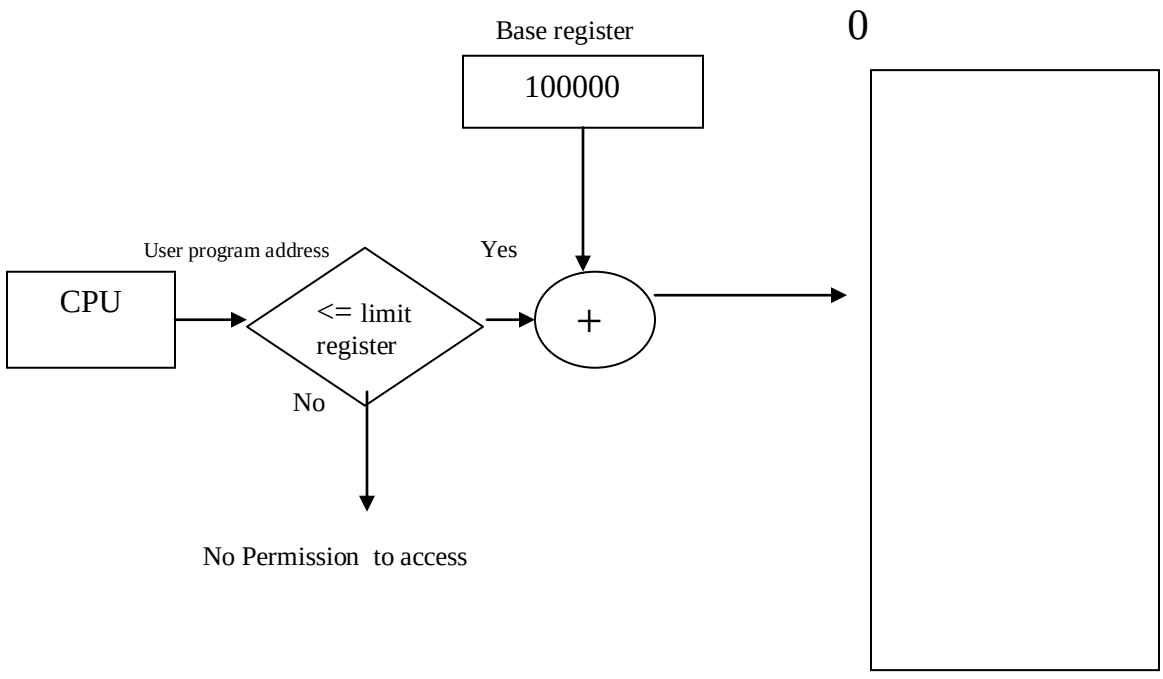
[Dynamic Relocation] Main Memory

Dynamic relocation makes it possible to move a partially executed process from one area of memory into another without affecting its ability to access instructions and data correctly in the new space.

Protection and Sharing :-

Multiprogramming introduces one essential problem of protection. Not only that the O.S. must be protected from user processes but each user process should also be protected from incorrectly accessing the areas of other processes.

In system that uses base register for relocation, a common approach is to use limit register for protection. The primary function of a limit register is to detect attempts to access memory location beyond the boundary assigned by the O.S. When a process is scheduled, the limit register is loaded with the highest virtual address in a program.



Max Memory

A good memory management mechanism must also provide for controlled sharing of data and code between cooperating processes.

There are three basic approaches to sharing in systems with fixed partitioning of memory :-

1. Entrust shared objects to O.S.
2. Maintain multiple copies, one per participating partition of shared objects
3. Use shared common memory partition.

One traditional approach to sharing is to place data and code in a dedicated common partition.

Fixed Partitioning imposes several restriction :-

1. No single process may exceed the size of the largest partition in a given system.
2. It does not support a system having dynamically data structure such as linked list, stack, queues etc.
3. It limits the degree of multiprogramming which in turn may reduce the effectiveness of short-term scheduling.

Multiprogramming with dynamic partitions :-

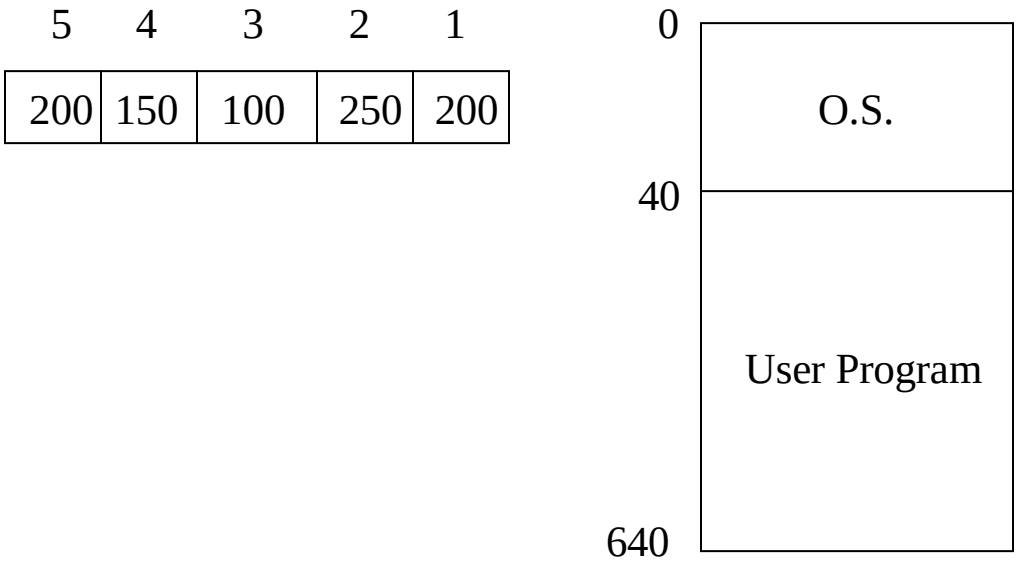
The main problem with fixed size partition is the wastage of memory by programs that are smaller than their partitions (i.e. internal fragmentation)

A different memory management approach known as dynamic partitions which creates partitions dynamically to meet the requirements of each requesting process.

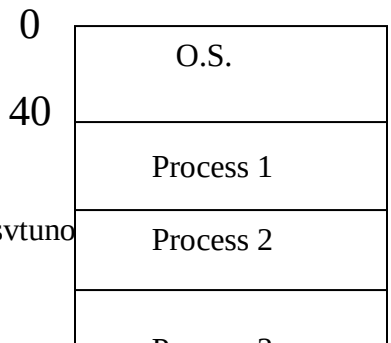
When a process terminates or becomes swapped – out, the memory manager can return the vacated space to the pool of free memory areas from which partition allocations are made.

Dynamic memory allocation improves memory utilization but it also complicates the process of allocation and de-allocation of memory.

Example :- Assume that we have 640 K main memory available in which 40 K is occupied by operating system program. There are 5 jobs waiting for memory allocation in a job queue.



Applying FCFS scheduling policy, process 1, process 2 and process 3 can be immediately allocated in memory. Process 4 can not be accommodated because there is only 600 – 550 = 50 left for it.



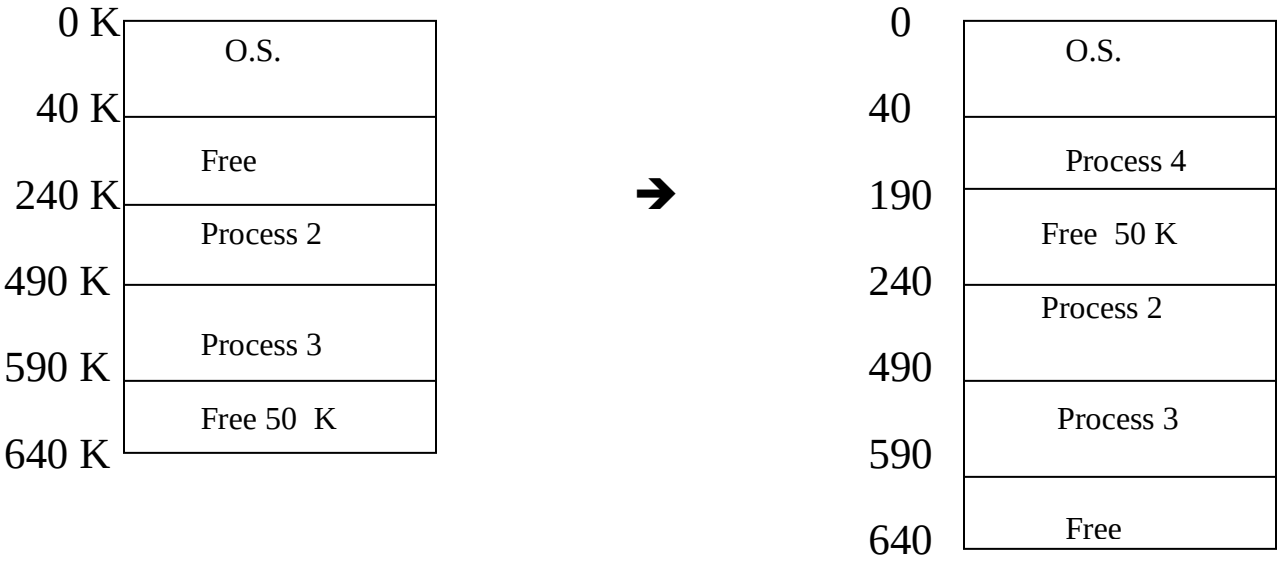
240

490

590

640

Let us assume that Process 1 is terminated, releasing 200 K memory space, next process 4 is swapped in memory.



After process 1, Process 3 gets terminated releasing 100 K Memory, but Process 5 can not be accommodated due to external fragmentation.

After the swapping out of process 2, process 5 will be loaded for execution.

40 → 40

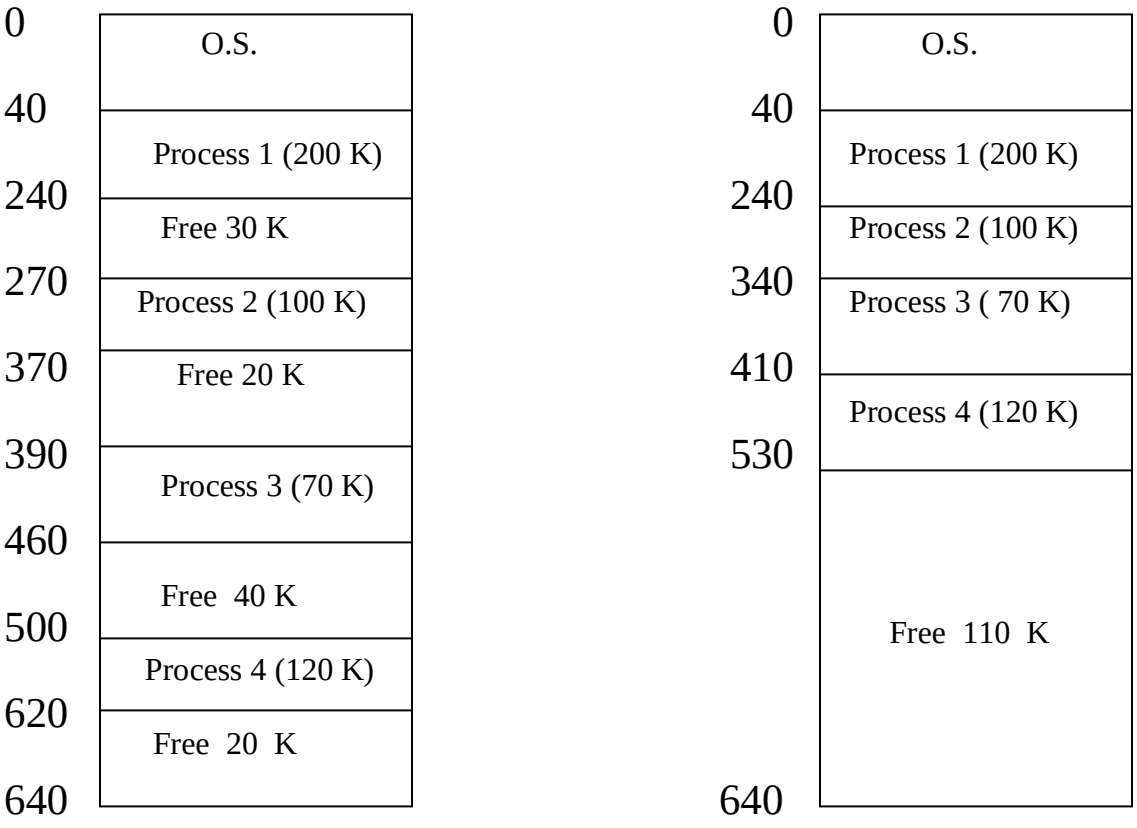
190 → 190

240 → 390

490

640 → 640

One solution to this problem is compaction. It is possible to combine all the free space into a large block by pushing all the processes upward as far as possible.



In diagram (a) there are 4 free spaces 30 K, 20 K, 40 K and 20 K which have been compacted into one large area of 110 K.

Compaction is usually not done because it consumes a lot of CPU time. It is usually done on a large machine like mainframe or super computer because they are supported with a special h/w to perform this task.

Advantages of Dynamic Memory Allocation :-

- Memory utilization is generally better than fixed size partitions are created accordingly to the size of process.
- It support processes whose memory requirement increase during their execution. In that case O.S. creates a larger partition and moves a process into it. If there is an adjacent free area it simply expands it.

Disadvantages of D.M.A. :-

- Dynamic memory management requires lots of O.S. space, time, complex memory management algorithm and bookkeeping operation.
- Compaction time is very high.

Memory – Management (Non – Contiguous Allocation)

Non – Contiguous allocation means that memory is allocated in such a way that parts of a single logical object may be placed in non contiguous areas of physical memory.

Paging :

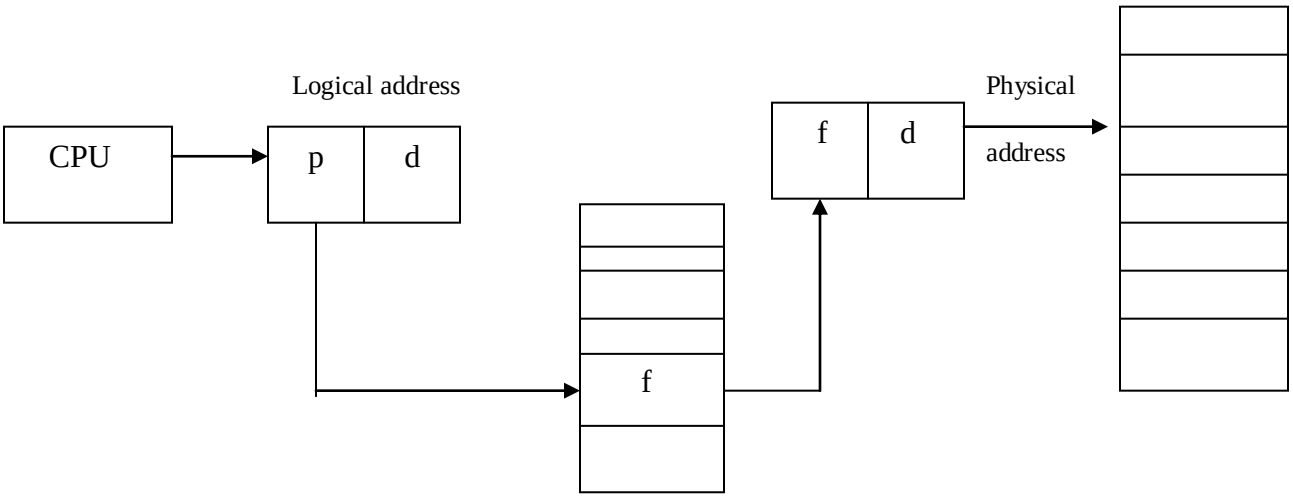
Paging is a memory management scheme that removes the requirement of contiguous allocation of physical memory.

The physical memory is conceptually divided into a number of fixed-size slats, called page frames. The virtual-address space of a process is also split into fixed-size blocks of the same size, called pages. When a program is to be run, its pages are loaded into any frame from the disk.

Each virtual address is divided into two parts : the page number (p) and offset d within that page.

In paging system, address translation is performed with the aid of a mapping table, called the page-ma table (PMT). [The PMT is constructed at process loading time in order to establish the correspondence between the virtual and physical addresses . for convenience of mapping, page sizes are usually an integer power of base 2, page sizes vary between 512 bytes and 8 K.B.]

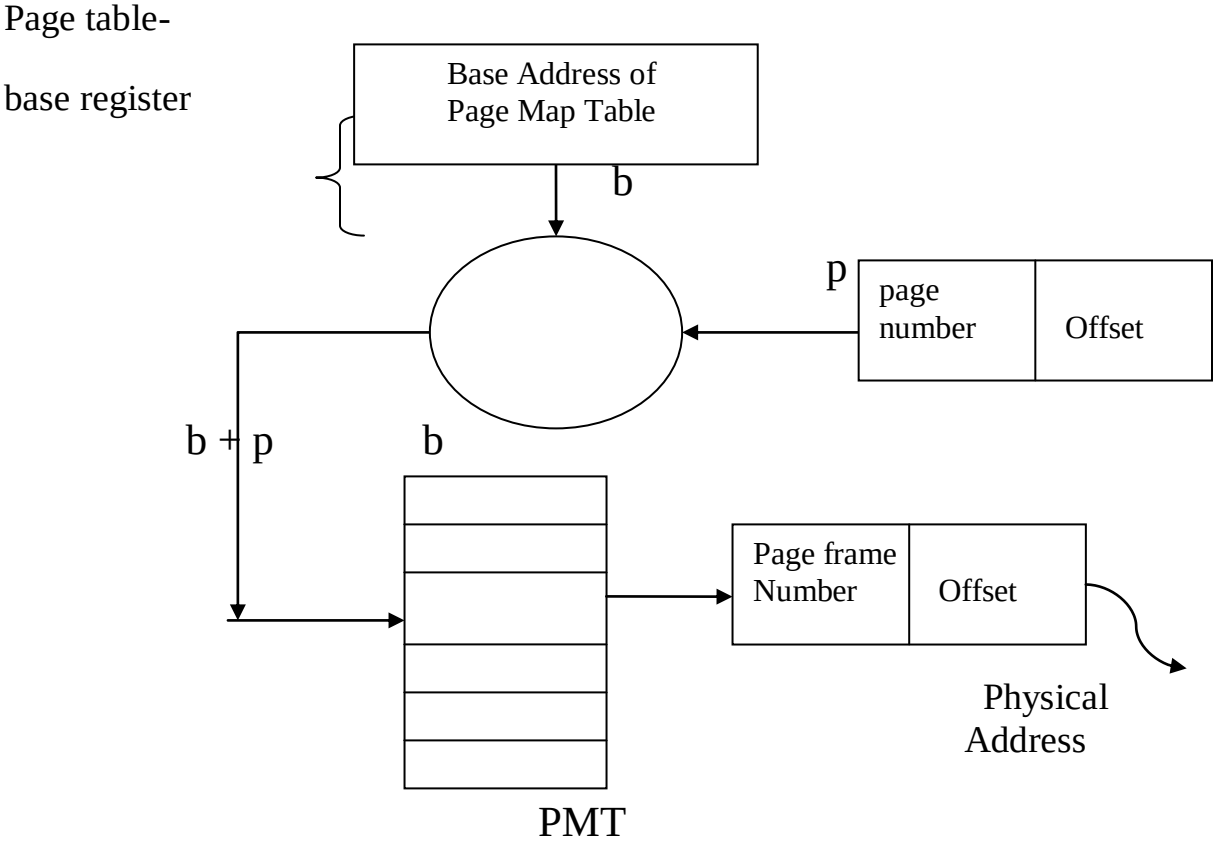
The PMT contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address.



Hardware support for Paging :-

The main objectives for h/w support for paging is to store page map table and make virtual to physical address translation. To reduce the access time, the use of registers can be considered if the no of entries in PMT is small. For keeping very large entries PMT is kept in main memory and there is a "Page Table Base Register" pointing to the beginning of PMT.

If we store PMT in Main Memory than problem with this approach is memory access time. The standard solution to this problem is to store the complete page-map-table into an associative memory also called "content addressable memory" or "look aside memory".



Sharing and Protection in a Paging system :-

In a multiprogramming environment, where several users want to execute the same s/w, keeping a separate copy of the same for individual users will cause wastage of much of primary memory.

To implement sharing each page is identified as a sharable or non-sharable. We can add "access-bits" in PMT entries.

Sharing reduces the amount of primary storage needed for several users to run efficiently and make it possible for a given system to support more users.

Protection - Memory protection is usually done by protection bits associated with each page. These bits are usually kept in the page map table. One protection bit can define a page to be read/write or a read only.

Thrashing :- The condition in which, process spends its more time in paging than in execution, called thrashing. In order to increasing CPU utilization, degree of multiprogramming is increased, but if by increasing degree of multiprogramming, CPU utilization is decreased then such a condition is called thrashing.

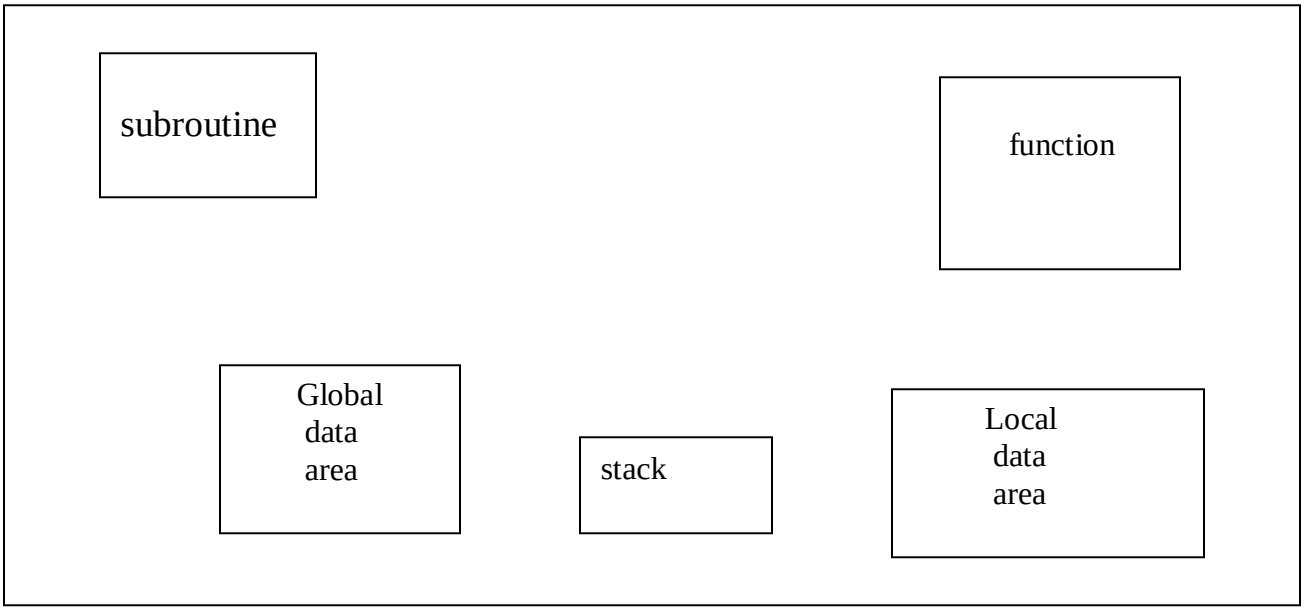
The main reason for occurring thrashing is O.S. requires CPU utilization and other is global page replacement policy.

Minimization of Thrashing :-

A local replacement algorithm is used to limit the effect of thrashing.

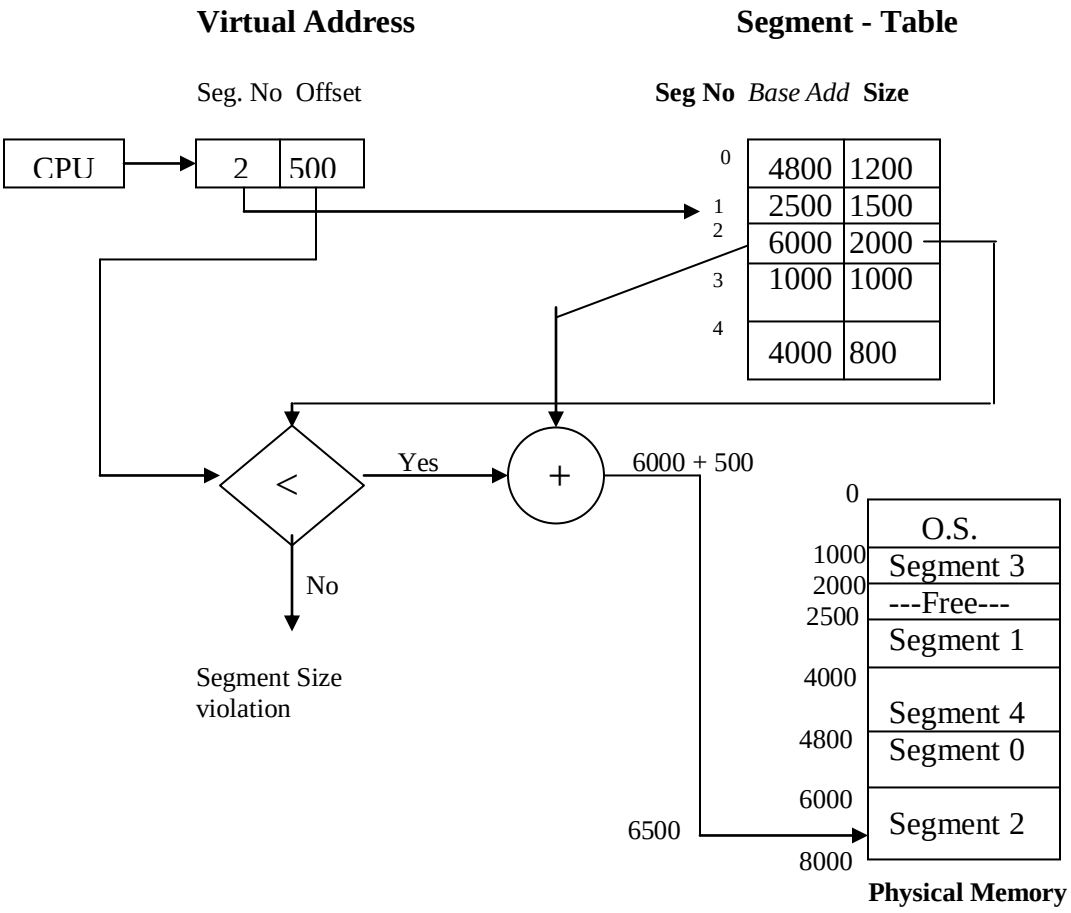
Segmentation :-

Segmentation is a memory management scheme which supports programmer's view of memory. Programmer's think of their programs as a collection of logically related entities, such as subroutines, functions, global or local data areas, stack etc.



Segments are formed at program translation time by grouping together logically related entities.

Each segment in a program is numbered and referred to by a segment number rather than a segment name.



A virtual address consist of two parts : a setment number and an offset into that setgment. Each row of the segment table contains a starting address(base address)
of segment and size of the segment.

The offset of the virtual address must be within the size of the segment.

If the offset of virtual address is not within the range, it is trapped by the O.S. otherwise the offset is added to the base address of the segment to produce physical address of the desired segment.

H/W support for segmentation :-

- Segment Table Base Register
- Segment Table Limit Register

Sharing and Protection in a Segmented System :-

One of the advantage of segmentation over paging is that segmentation over paging is that segments are allowed to be as large as they require to be.

A segment may increase and decrease in size as the data structure itself increases and decreases.

Protection of one segment from another segment is done through protection bit.

In a segmentation system once the segment is declared as shared, then the size of data structured may increase or decrease without changing the logical fact that they reside on a shared segment.

Virtual – Memory

If the size of a job is larger than the available memory than it can not be executed. Since it can not be loaded into memory entirely. In such a case, concept of virtual memory is used.

Virtual memory allows user to execute program larger than size of available main memory.

Advantage of virtual Memory :-

1. Users would be able to write programs for very large virtual address space.
2. Since each user utilizes less physical memory, more users can keep their programs simultaneously which will cause increase in CPU utilization and throughput.

3. Since a process may be loaded into a space of arbitrary size, which in turn reduce external fragmentation.

There are two major techniques of virtual memory concept –

(i) Demand Paging memory management

(ii) Demand segmentation memory management.

Demand Paging :-

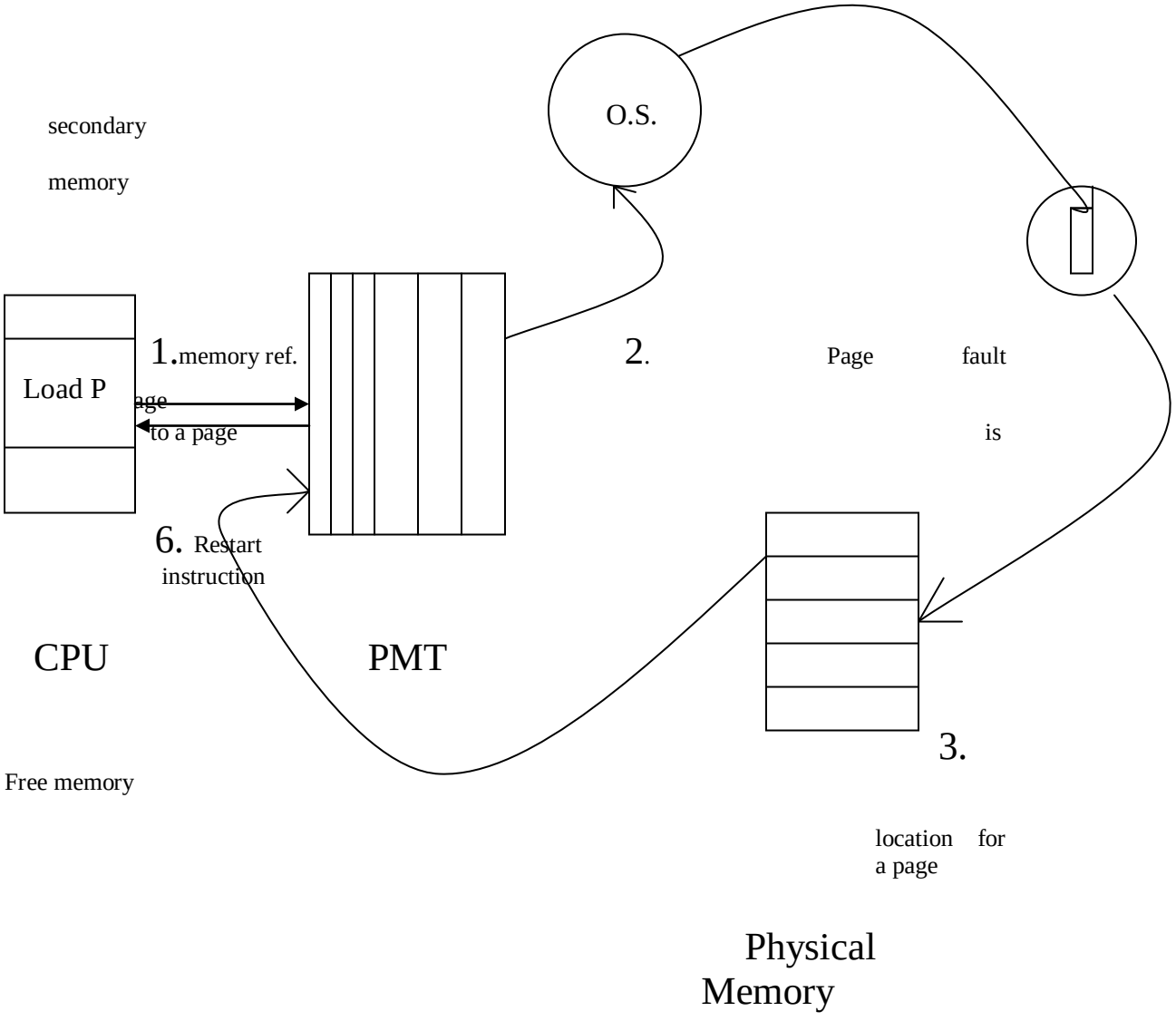
In demand paging pages are loaded only on demand, not in advance. It is similar to paging system with swapping feature. Rather than swapping the entire program in memory, only those pages are swapped which are required currently by the system.

What will happen if the program tries to access a page that was not swapped in memory ? In that case, page fault trap occurs.

List of steps O.S. follows in handling a page fault :-

1. If a process refers to a page which is not in the physical memory, then an internal table kept with a process control block is checked to verify whether a memory reference to a page was valid or invalid.
2. If page was valid, but the page is missing, the process of bringing a page into the physical memory starts.
3. Free Memory location is identified to bring a missing page
4. By reading a disk, the desired page is brought back into the free memory location.
5. once a page is in the physical memory, the internal table kept with the process and page map table is updated to indicate that the page is now in memory.
6. Restart the instruction that was interrupted due to the missing page.

Q : When do page fault occur ? describe the action taken by the O.S. when a page fault occurs.

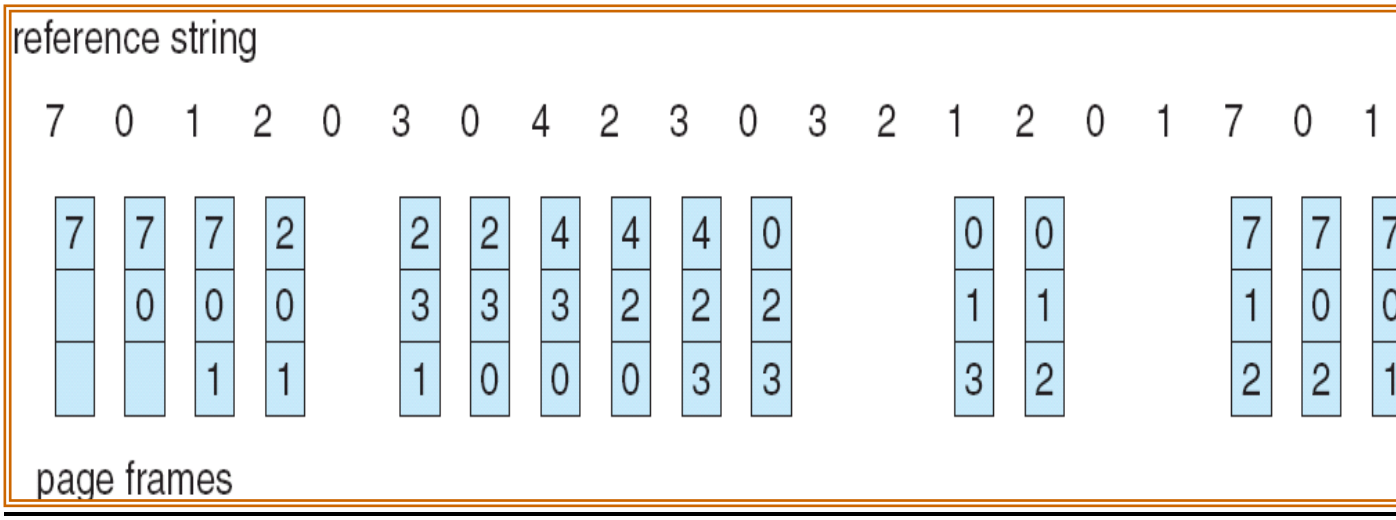


5. Update page map table

Page Replacement Algorithm

The first-in, first-out (FIFO) page replacement algorithm is low-overhead algorithm which requires little bookkeeping on the part of the operating system.

The operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the earliest arrival in front. When a page needs to be replaced, the page at the front of the queue is selected, as it will be the oldest page.



Page Faults = 15

Number of pages (P) = 20

Number of page faults F = 15

Failure frequency = F/P = 15/20 = 75 %

Belady's Anomaly:

It is observed that the no. of page fault for four frames is greater than the no of faults for three frames. This result is known as "Belady's Anomaly".

Optimal Replacement Algorithm (OPT) :

Replace that page which will not be used for the longest period of time. An optimal algorithm would never suffer from

Virtual Memory

If the size of a job is larger than the available memory than it can not be executed. Since it can not be loaded into memory entirely. In such a case, concept of virtual memory is used.

Virtual memory allows user to execute program larger than size of available main memory.

Advantage of virtual Memory :-

4. Users would be able to write programs for very large virtual address space.
5. Since each user utilizes less physical memory, more users can keep their programs simultaneously which will cause increase in CPU utilization and throughput.
6. Since a process may be loaded into a space of arbitrary size, which in turn reduce external fragmentation.

There are two major techniques of virtual memory concept –

(iii) Demand Paging memory management

(iv) Demand segmentation memory management.

Demand Paging :-

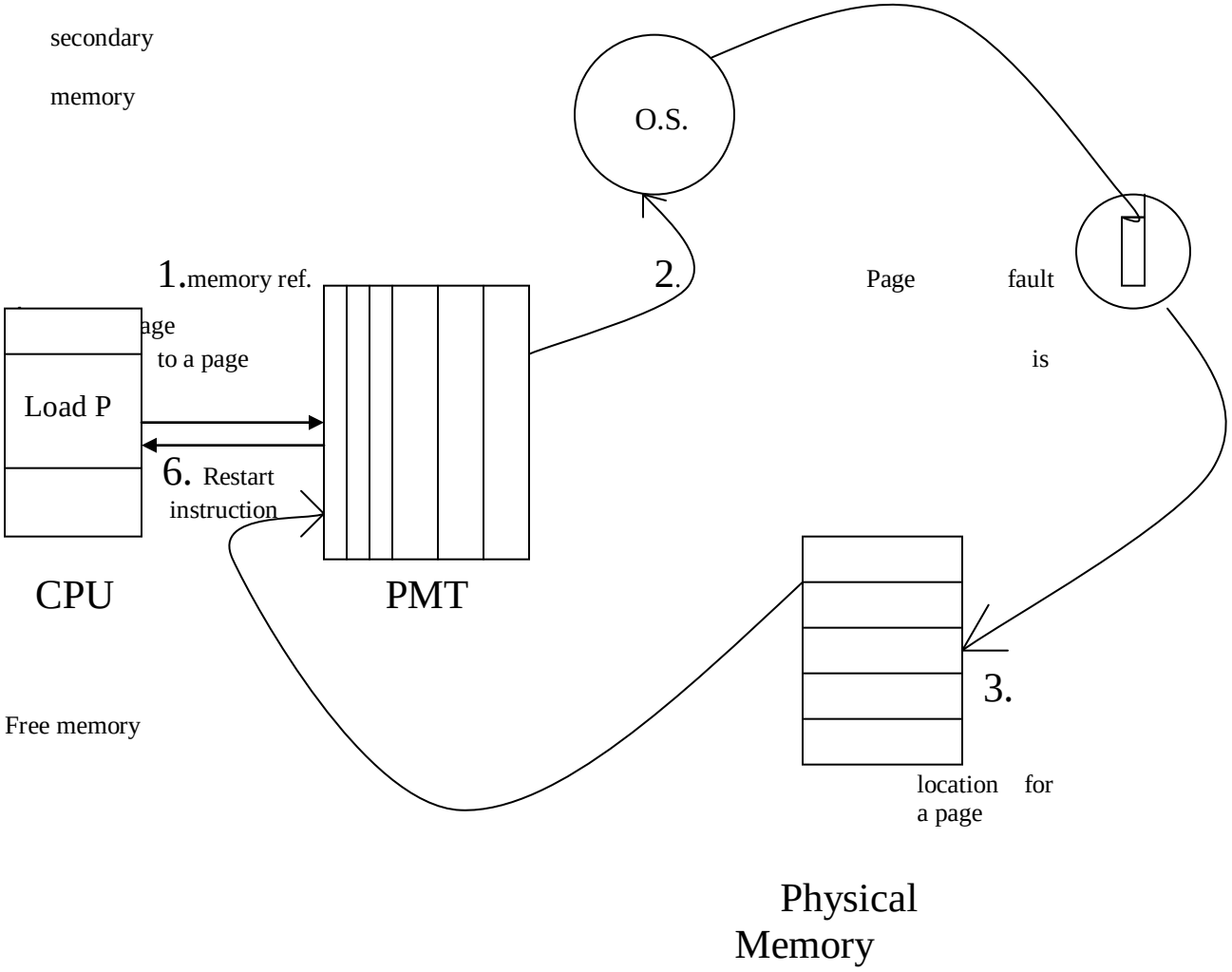
In demand paging pages are loaded only on demand, not in advance. It is similar to paging system with swapping feature. Rather than swapping the entire program in memory, only those pages are swapped which are required currently by the system.

What will happen if the program tries to access a page that was not swapped in memory ? In that case, page fault trap occurs.

List of steps O.S. follows in handling a page fault :-

7. If a process refers to a page which is not in the physical memory, then an internal table kept with a process control block is checked to verify whether a memory reference to a page was valid or invalid.
8. If page was valid, but the page is missing, the process of bringing a page into the physical memory starts.
9. Free Memory location is identified to bring a missing page
10. By reading a disk, the desired page is brought back into the free memory location.
11. once a page is in the physical memory, the internal table kept with the process and page map table is updated to indicate that the page is now in memory.
12. Restart the instruction that was interrupted due to the missing page.

Q : When do page fault occur ? describe the action taken by the O.S. when a page fault occurs.



5. Update page map table

Unit - 3

Dead Lock

A deadlock is a situation where a group of processes are permanently blocked as a result of each process having acquired a subset of the resources needed for its completion and waiting for release of the remaining resources held by others in the same group.

There are four necessary conditions for a deadlock to occur :-

1. Mutual Exclusion – only one process may use a shared resource at a time.
2. Hold and wait – each process continues to hold resources already allocated to it while waiting to acquire other resources.
3. No preemption – No resources can be forcibly removed from a process holding it.
4. Circular waiting – Deadlocked processes are involved in a circular chain such that each process holds one or more resources being requested by the next process in the chain.

All four conditions must be present for a deadlock to occur.

Most of the practical deadlock handling technique fall into one of these three categories :

A. deadlock prevention
B. deadlock avoidance and
C. deadlock detection and recovery

(A) **Deadlock prevention** :- The basic philosophy of deadlock prevention is to deny at least one of the four conditions for deadlock.

Now we examine techniques related to each of the four conditions.

(1) Mutual exclusion: - we can not prevent deadlocks by denying the mutual-exclusion condition. So we should prevent one or more of the remaining three conditions.

(2) Hold & wait: - the hold & wait condition can be eliminated by requiring a process to release all resources held by it whenever it requests a resource that is not available.

There are basically two possible implementations of this strategy :

(i) The process requests all needed resources prior to commencement of execution.

(ii) The process requests resources incrementally in the course of execution but releases all its resources (holding upon encountering a denial) held by it whenever it request a resource that is not available.

(3) No-preemption: - The no-preemption deadlock condition can obviously be denied by allowing preemption. If the process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

(4) Circular wait: - one way to prevent the circular –wait condition is by linear ordering of different types of system resources.

In this approach, System resources are divided into different classes C_j, where j = 1, 2 …n. Deadlocks are prevented by requiring all processes to request and acquire their resources in a strictly increasing order of the specified system resource classes.

Once a process acquires a resource belonging to the class Cj it can only request resources of class j+1 or higher thereafter.

Linear ordering of resource classes eliminates the possibility of circular waiting.

(B) **Deadlock Avoidance** :- A method for avoiding deadlocks is to require additional information about how resources are to be requested. The most useful model requires that each process declare the maximum number of resources of each type that it may need. So it is possible to construct an algorithm that ensures that the system will never enter a deadlock state.
The resource – allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe state : A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.

Banker's Algorithm :

The name banker's algorithm was suggested because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resources type that it may need. This number may not exceed the total no. of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will safe, the resources are allocated; otherwise the process must wait until some other process releases enough resources.

Some data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.

Available - A vector of length m indicating the number of available resources of each type. If Available[j] = k, there are k instances of resource type r_j available.

Max - An n*m matrix defined the maximum demand of each process. If Max[i,j] = k, then process p_i may request at most k instances of resource type r_j.

Allocation – An n*m matrix defining the no of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process p_i is currently allocated k instances of resource type $r_{j.}$

Need – An n*m matrix indicating the remaining resource need of each process. If Need[i,j] = k, then process p_i may need k more instances of resource type $r_{j,}$ in order to complete its task.

Note that

Need[i,j] = Max[i,j] – Allocation[i,j]

These data structure vary both in size and value as time progresses.

Resource – Request Algorithm :

Let Request $_i$ be the request vector for process p $_i$. If Request$_i$ [j] = k then process p_i wants k instances of resource type r_j. When a request for resources is made by process pi, the following actions are taken :

1. If Request$_i$ <= Need$_i$ then proceed to step 2. Otherwise we have an error, since the process has exceeded its maximum claim.

2. If $Request_i$ <= Available then proceed to step3. Otherwise the resources are not available, and pi must wait.
3. The system pretends to have allocated the requested resources to process pi by modifying the state as follows

   Available = Available - $Request_i$
   $Allocation_i$ = $Allocation_i$ – $Request_i$
   $Need_i$ = $Need_i$ – $Request_i$

If the resulting resources allocation state is safe, the transaction is completed and process pi is allocated its resources. However, if the new state is unsafe, then pi must wait for Requesti and the old resource allocation state is restored.

Safety Algorithm :

1. Let Work and Finish be vectors of length m and n respectively.
   Initialize Work = Available and
   Finish[i] = false for i = 1,2,3,……n
2. Find an I such that
   (a) Finish[i] = false, and
   (b) Needi <= Work

   If no such I exist, go to step 4
3. Work = Work + Allocationi
   Finish[i] = true
   Go to step 2

4. If Finish[i] = true for all I, then the system is in a safe state, Otherwise system is in unsafe state.

Question :

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Question :

	Allocation	Max	Available
	A B C	A B C	A B C
P0	4 1 2	1 0 2	2 2 0
P1	1 5 1	0 3 1	***2 5 1***
P2	1 2 3	1 0 2	***3 5 3***
			4 5 5

(i) What is the state of system

(a) **Safe state** (b) Unsafe state (c) can't determine

(ii) What will be the state if process pi request for one unit of B.

(a) will be allocated (b) **will not be given** (c) none

Deadlock detection

If a system does not employ some protocol that ensures that no deadlock will ever occur, then a detection & recovery scheme must be implemented.

An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If so, the system must attempt to recover from the deadlock. In order to do so the system must

(a) Maintain information about the current allocation of resources as well as any outstanding resource allocation requests.
(b) Provide an algorithm that utilizes this information to determine whether the system has entered a deadlock state.

The detection algorithm employs several time-varying data structures that are similar to those used in banker's algorithm :
Available –
Allocation –
Request –

The algorithm simply investigates every possible allocation for the processes that remain to be completed.

1. Let Work and Finish be vectors of length m and n respectively.

   Initialize Work = Available
   For I = 1,2,……n. If Allocation I =/ 0
   Then Finish[i] = false, Otherwise Finish[i] = true

2. Find an index I such that
   a. Finish [i] = false and
   b. Request I <= Work

If no such i exists go to step 4

3. Work = Work + Allocation i
   Finish [i] = true
   go to step 2.

3. If Finish [i] = false, for some i, 1<= i <= n, then the system is in a deadlock state. Moreover, if Finish[i] = false then process pi is deadlocked.

Recovery from deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. There are two options for breaking a deadlock.

One solution would be to simply true, kill one or more processes in oreder to break the circular wait.

The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination :-

In order to eliminate the deadlock by killing a process, two methods can be utilized.

4. Kill all deadlocked processes
5. Kill one process at a time until the deadlock cycle is eliminated

Combined Approached to deadlock handling :-

It has been argued that none of the presented approaches is suitable for use as an exclusive method of handling of deadlocks in a

complex system. Instead, deadlock prevention, avoidance and detection can be combined for maximum effectiveness. This can be accomplished by dividing system resources into a collection of disjoint classes, and by applying the most suitable method of handling deadlocks to resources within each particular class.

Consider a system with the following classes of resources :

1. Swapping space – an area of secondary storage designated for backing up blocks of main memory.
2. Job resources - Such as printers and drivers with removable media (tapes, cartridge disk, floppies)
3. Main Memory – assignable on a block basis, such as pages or segments
4. Internal resources – such as I/O channels and slots of the pool of dynamic memory.

Unit - 2

Principles of Concurrency :

Concurrency refer to a parallel execution of a program. A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes.
Concurrent processing is the basis of operating system which supports multiprogramming.

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Concurrency may arise in three different contexts :

- *Multiple applications*
  - Multiprogramming
- *Structured application*
  - Application can be a set of concurrent processes
- *Operating-system structure*
  - Operating system is a set of processes or threads

Some key terms related to concurrency :

- *Critical Section* – The area of a program where a resource is being used
- *Deadlock* – When two or more processes halt, unable to proceed
  - P1 is using A, needs B
  - P2 is using B, needs A

- *Mutual Exclusion* – Making sure two processes can't both have a resource
- *Race Condition* – A situation in which multiple threads or processes wants to read and write a shared data item at the same time.
- *Starvation* – A process waits indefinitely for a resource
    - P1 using A, P2 and P3 wait for A
    - P2 gets A when P1 done
    - P1 comes in, P1 and P3 wait for A

- *InterProcess Communication* – Concurrent cooperating processes must communicate to each other for such purpose as exchanging data, reporting progress etc. To prevent timing errors, concurrent processes must synchronize their accessing of shared memory.

Concurrency Requirements :-

- Mutual Exclusion must be enforced
    - Only one process at a time may be accessing the critical section
- A process can halt outside the critical section without harm
- No deadlock or starvation
- If no process is in a critical section, a process requesting entry must be allowed to enter without delay
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a limited period of time

The Critical-Section Problem :

- N processes all competing to use shared data.

- Structure of process P*i* ---- Each process has a code segment, called the critical section, in which the shared data is accessed.

```
repeat
    entry section    /* enter critical section */
      critical section  /* access shared variables */
  exit  section      /* leave critical section */
      remainder section  /* do other work */
until false
```

- Problem
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution to Critical-Section Problem :

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

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Mutual Exclusion :-

Some way of making sure that if one process is executing in its critical section the other processes will be excluded from doing the same thing.

Algorithm to support mutual exclusion, this is applicable for two process only:

Dekker's algorithm :

```
var flag : array [0..1] of bollean;
    turn : 0..1

procedure P0;
begin
    repeat
        flag[0] := true;
        while flag[1] do if turn = 1 then
            begin
                flag[0] := false;
                    while turn = 1 do {nothing};
                flag[0] := true;
            end;
        <critical section>;
        turn := 1;
        flag[0] := false;
        <remaining task>
    forever
end;


procedure P1;
begin
    repeat
        flag[1] := true;
        while flag[0] do if turn = 01 then
```

```
                    begin
                        flag[1] := false;
                            while turn = 0 do {nothing};
                        flag[1] := true;
                    end;
                <critical section>;
                turn := 0;
                flag[1] := false;
                <remaining task>
        forever
end;


// parent process

begin
        flag[0] := false;
        flag[1] := false;
        turn := 1;
        parbegin
                P0;P1
        Parend
end.
```

Peterson's algorithm for two processes :

```
var flag : array [0..1] of bollean;
      turn : 0..1

procedure P0;
begin
        repeat
                flag[0] := true;
                turn := 1;
```

```
            while flag[1] and turn = 1 do {nothing};
            <critical section>;
            flag[0] := false;
            <remaining task>
        forever
end;

procedure P1;
begin
        repeat
            flag[1] := true;
            turn := 0;
            while flag[0] and turn = 0 do {nothing};
            <critical section>;
            flag[1] := false;
            <remaining task>
        forever
end;

// parent process
begin
        flag[0] := false;
        flag[1] := false;
        turn := 1;
        parbegin
            P0;P1
        Parend
end.
```

Semaphores :

In previous topic, we discussed the mutual exclusion problem. The solution we presented did not solve all the problem of mutual exclusion. These algorithms works for two processes only and it cannot be extended beyond that number.

To overcome this problem, a synchronization tool called semaphore was proposed by Dijkstra which gained wide acceptance in several OS.

Semaphore is a variable that has an integer value It may be initialized to a nonnegative number.
A semaphore mechanism basically consists of the two primitive operations SIGNAL and WAIT (originally defined as P and V by Dijkstra), which operate on semaphore variable s.
We can implement semaphore in two ways :

1. Counting semaphore – integer value can range over an unrestricted domain
2. Binary semaphore – integer value can range only between 0 and 1; Also known as mutex locks

Definition of counting semaphore primitives :

```
type semaphore = record
                count : integer;
                queue: list of processes;
end ;
var s : semaphore;

wait(s) :
        s.count := s.count – 1;
        if s.count < 0
                then begin
                        place this process in s.queue;
                        block this process;
                end;
```

signal(s) :

```
        s.count := s.count + 1;
        if s.count <=0
                then begin
                        remove a process P from s.queue;
                        place process P on ready list
                end;
```

Definition of binary semaphore primitives :

```
type binary semaphore = record
                value : (0,1);
                queue: list of processes;
        end ;
var s : binary semaphore;

waitB(s) :

        if s.value = 1 then
                 s.value = 0
        else begin
                        place this process in s.queue;
                        block this process;
        end;

signalB(s) :

        if s.queue is empty then
                 s.value := 1
        else begin
                        remove a process P from s.queue;
                        place process P on ready list
        end;
```

Mutual Exclusion with Semaphore :

```
Module  Sem-mutex
var bsem : semaphore; {binary semaphore}
process P1;
        begin
                while true do
                        begin
                                wait(bsem)
                                Critical_section
                                signal(bsem)
                                <remaining  P1 task>
                        end
process P2;
        begin
                while true do
                        begin
                                wait(bsem)
                                Critical_section
                                signal(bsem)
                                <remaining P2 task>
                        end
process P3;
        begin
                while true do
                        begin
                                wait(bsem)
                                Critical_section
                                signal(bsem)
                                <remaining P3 task>
                        end
//parent process
        begin {Sem-mutex}
                bsem := 1  {free}
```

```
        initiate P1,P2,P3
    end;
```

The Producer/Consumer Problem (Bounded Buffer Problem) :-

The general statement is this : One or more producers are generating some type of data(records, characters) and placing these in a buffer.

A single consumer is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is only one agent (producer or consumer) may access the buffer at any one time.

On the other words, a consumer may absorb only produced items, and must wait when no items are available. Producers on the other hand, may produce items only when there are empty buffer slots to receive them.

- ❖ *N* buffers, each can hold one item
- ❖ Semaphore mutex initialized to the value 1
- ❖ Semaphore full initialized to the value 0
- ❖ Semaphore empty initialized to the value N.

The structure of the producer process

```
while (true)
   {
        //   produce an item

       wait (empty);
       wait (mutex);

        //  add the item to the  buffer (critical section)
```

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

The structure of the consumer process

```
while (true)
    {
            wait (full);
            wait (mutex);

                    //  remove an item from  buffer (critical section)

            signal (mutex);
            signal (empty);

                    //  consume the removed item
    }
```

Reader/Writer Problem

A data object is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, while others may want to update the shared object.

If two readers access the shared data object simultaneously, no adverse effect will result, however, if a writer access the shared object, problem may occur. This synchronization problem is referred to as the reader-writer problem.

We now examine two solutions to the problem.

(a)Readers have priority

Here the semaphore "wsem" is used to enforce mutual exclusion. So long as one writer is accessing the shared data area, no other writers and no readers may access it. (The reader process also makes use of wsem to enforce mutual exclusion.)

To allow multiple readers, we require that when there are no readers reading, subsequent readers need not wait before entering.

The global variable "readcount" is used to keep track of the number of readers, and the semaphore x is used to assure that readcount is updated properly.

```
program      readersandwriters

var    readcount  :  integer;
       x, wsem    :  semaphore  (:=1)

procedure  reader;

begin
     repeat
          wait(x);
          readcount = readcount + 1;
          if   readcount = 1 then wait(wsem);
          signal(x);
          READUNIT;
          wait(x);
          readcount  =  readcount  -  1;
```

```
            if readcount = 0 then signal(wsem);
            signal(x);
        forever
    end;


    procedure writer;

    begin
        repeat
            wait(wsem);
                WRITEUNIT;
            signal(wsem);
        forever
    end;

    // Parent process

    begin
        readcount = 0;
        parbegin
            reader;
            writer;
        parend;
    end;
```

"*A solution to the reader/writer problem by using semaphores*"

UNIT V

Design principle

Goals

Based on market requirements and Microsoft's development strategy, the original Microsoft NT design team established a set of prioritized goals. Note that from the outset, the priority design objectives of the Windows NT operating system were *robustness* and *extensibility*:

Robustness. The operating system must actively protect itself from internal malfunction and external damage (whether accidental or deliberate), and must respond predictably to software and hardware errors. The system must be straightforward in its architecture and coding practices, and interfaces and behavior must be well- specified.

Extensibility and maintainability. Windows NT must be designed with the future in mind. It must grow to meet the future needs of original equipment manufacturers (OEMs) and Microsoft. And the system must be designed for maintainability, it must accommodate changes and additions to the API sets it supports and the APIs should not employ flags or other devices that drastically alter their functionality.

Portability. The system architecture must be able to function on a number of platforms with minimal recoding.

Performance. Algorithms and data structures that lead to a high level of performance and that provide the flexibility needed to achieve our other goals must be incorporated into the design.

POSIX compliance and government certifiable C2 security. The POSIX standard calls for operating system vendors to implement UNIX-style interfaces so that applications can be moved easily from one system to another. U.S. government security guidelines specify certain protections, such as auditing capabilities, access detection, per-user resource quotas, and resource protection. Inclusion of these features would allow Windows NT to be used in government operations.

Mechanisms and polices

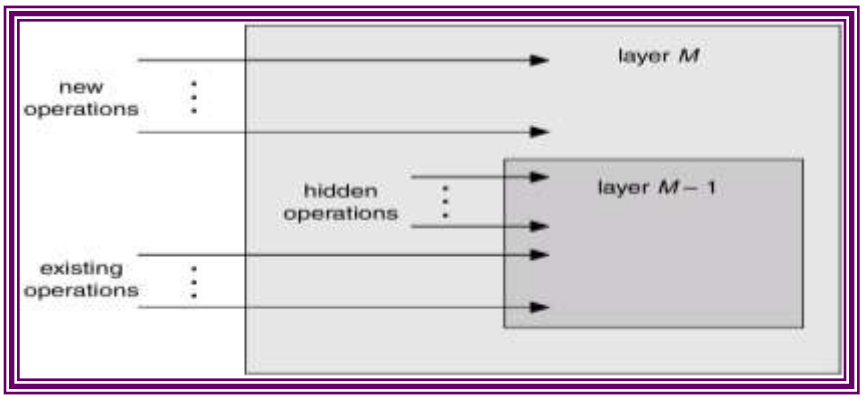
A **policy** is a plan of action to guide decisions and actions. The term may apply to government, private sector organizations and groups, and individuals. The policy process includes the identification of different alternatives, such as programs or spending priorities, and choosing among them on the basis of the impact they will have. Policies can be understood as political, management, financial, and administrative mechanisms arranged to reach explicit goals.

The separation of policy and mechanism is very important for flexibility. Policies are likely to change from place to place or time to time. A general mechanism would be more desirable.

Layered approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

An Operating System Layer

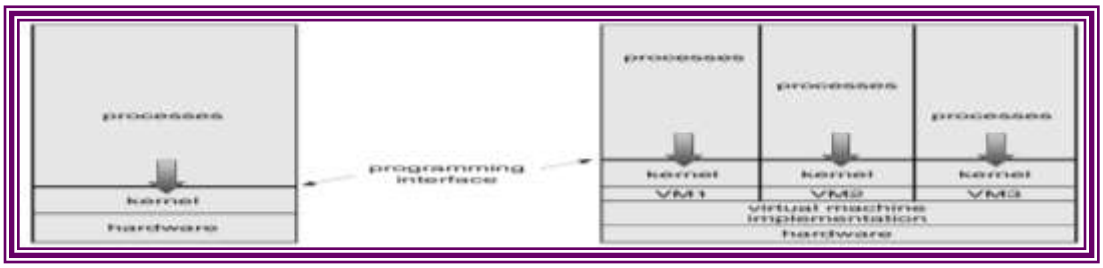


OS/2 Layer Structure

Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.
- The resources of the physical computer are shared to create the virtual machines**.**
- CPU scheduling can create the appearance that users have their own processor.
- Spooling and a file system can provide virtual card readers and virtual line printers.
- A normal user time-sharing terminal serves as the virtual machine operator's console.

System Models



Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

Multiprocessor

A **multiprocessor** computer is one with more than one CPU. The category of multiprocessor computers can be divided into the following sub-categories:

- **shared memory multiprocessors** have multiple CPUs, all with access to the same memory. Communication between the the processors is easy to implement, but care must be taken so that memory accesses are synchronized.
- **distributed memory multiprocessors** also have multiple CPUs, but each CPU has it's own associated memory. Here, memory access synchronization is not a problem, but communication between the processors is often slow and complicated.

Related to multiprocessors are the following:

- **networked systems** consist of multiple computers that are networked together, usually with a common operating system and shared resources. Users, however, are aware of the different computers that make up the system.
- **distributed systems** also consist of multiple computers but differ from networked systems in that the multiple computers are transparent to the user. Often there are redundant resources and a sharing of the workload among the different computers, but this is all transparent to the user.

System Implementation

1. Traditionally written in assembly language, operating systems can now be written in higher-level languages.
2. Code written in a high-level language:

    - can be written faster.
    - is more compact.
    - is easier to understand and debug.

3. An operating system is far easier to *port* (move to some other hardware) if it is written in a high-level language.

System Generation (SYSGEN)

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
- *Booting* – starting a computer by loading the kernel.
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.

Distributed system

Motivation

1. **Distributed system** is collection of loosely coupled processors interconnected by a communications network
2. Processors variously called *nodes, computers, machines, hosts*

    - *Site* is location of the processor

3. Reasons for distributed systems
    - Resource sharing
        - sharing and printing files at remote sites
        - processing information in a distributed database
        - using remote specialized hardware devices
    - Computation speedup – **load sharing**
    - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
    - Communication – message passing

Network Topology

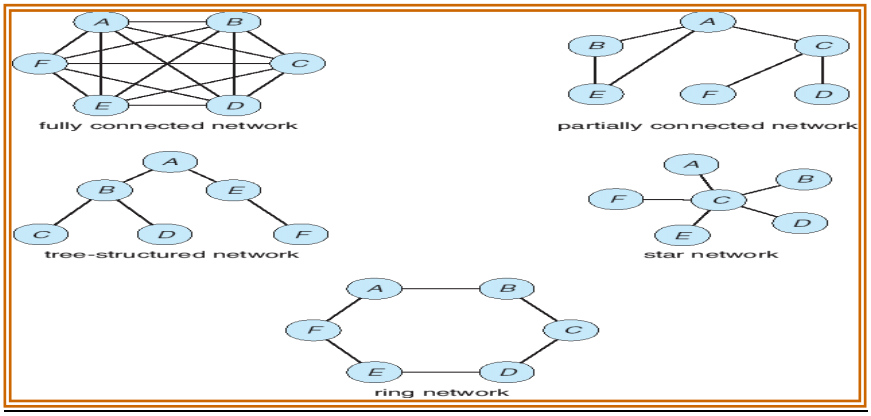
1. Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:

- **Basic cost** - How expensive is it to link the various sites in the system?
- **Communication cost** - How long does it take to send a message from site *A* to site *B*?
- **Reliability** - If a link or a site in the system fails, can the remaining sites still communicate with each other?

2. The various topologies are depicted as graphs whose nodes correspond to sites

   - An edge from node *A* to node *B* corresponds to a direct connection between the two sites

3. The following six items depict various network topologies



Communication Structure

The design of a *communication* network must address four basic issues:

- **Naming and name resolution** - How do two processes locate each other to communicate?
- **Routing strategies** - How are messages sent through the network?
- **Connection strategies** - How do two processes send a sequence of messages?
- **Contention** - The network is a shared resource, so how do we resolve conflicting demands for its use?

Naming and Name Resolution

- Name systems in the network
- Address messages with the process-id
- Identify processes on remote systems by

<host-name, identifier> pair

- *Domain name service* (DNS) – specifies the naming structure of the hosts, as well as name to address resolution (Internet)

Routing Strategies

1. **Fixed routing** - A path from *A* to *B* is specified in advance; path changes only if a hardware failure disables it

    - Since the shortest path is usually chosen, communication costs are minimized
    - Fixed routing cannot adapt to load changes
    - Ensures that messages will be delivered in the order in which they were sent

2. **Virtual circuit** - A path from *A* to *B* is fixed for the duration of one session. Different sessions involving messages from *A* to *B* may have different paths
    - Partial remedy to adapting to load changes
    - Ensures that messages will be delivered in the order in which they were sent

Dynamic routing - The path used to send a message form site *A* to site *B* is chosen only when a message is sent

- Usually a site sends a message to another site on the link least used at that particular time
- Adapts to load changes by avoiding routing messages on heavily used path
- Messages may arrive out of order
- This problem can be remedied by appending a sequence number to each message

Connection Strategies

1. **Circuit switching** - A permanent physical link is established for the duration of the communication (i.e., telephone system)
2. **Message switching** - A temporary link is established for the duration of one message transfer (i.e., post-office mailing system)
3. **Packet switching** - Messages of variable length are divided into fixed-length packets which are sent to the destination

    - Each packet may take a different path through the network
    - The packets must be reassembled into messages as they arrive

4. Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth

- Message and packet switching require less setup time, but incur more overhead per message

Contention

Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

1. **CSMA/CD** - Carrier sense with multiple access (CSMA); collision detection (CD)

    - A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting
    - When the system is very busy, many collisions may occur, and thus performance may be degraded

2. CSMA/CD is used successfully in the Ethernet system, the most common network system
3. **Token passing** - A unique message type, known as a token, continuously circulates in the system (usually a ring structure)

    - A site that wants to transmit information must wait until the token arrives
    - When the site completes its round of message passing, it retransmits the token
    - A token-passing scheme is used by some IBM and HP/Apollo systems

4. **Message slots** - A number of fixed-length message slots continuously circulate in the system (usually a ring structure)

    - Since a slot can contain only fixed-sized messages, a single logical message may have to be broken down into a number of smaller packets, each of which is sent in a separate slot
    - This scheme has been adopted in the experimental Cambridge Digital Communication Ring

Communication Protocol

The communication network is partitioned into the following multiple layers:

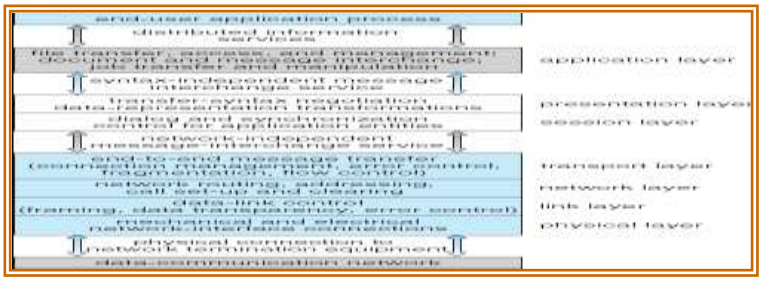
- **Physical layer** – handles the mechanical and electrical details of the physical transmission of a bit stream
- **Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer
- **Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets,

decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels

- **Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses
- **Session layer** – implements sessions, or process-to-process communications protocols
- **Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing)
- **Application layer** – interacts directly with the users' deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases

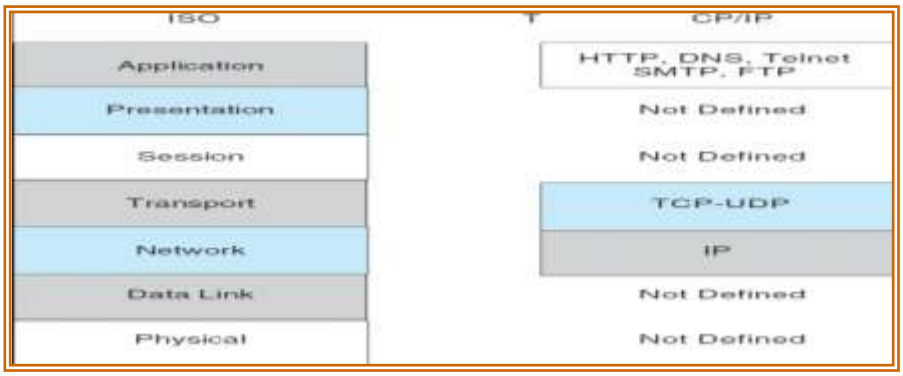
Communication Via ISO Network Model

The ISO Protocol Layer



The ISO Network Message

The TCP/IP Protocol Layers



File Concept

1. Contiguous logical address space
2. Types:
    - Data
        - numeric
        - character
        - binary
    - Program

File Structure

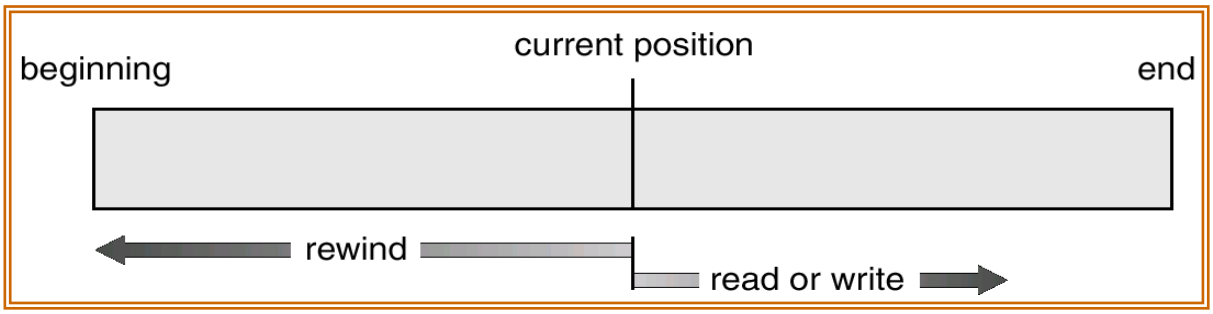
1. None - sequence of words, bytes
2. Simple record structure
    - Lines
    - Fixed length
    - Variable length
3. Complex Structures
    - Formatted document
    - Relocatable load file
4. Can simulate last two with first method by inserting appropriate control characters

5. Who decides:
    - Operating system
    - Program

Modes of computation

- Sequential Access
    - read next
    - write next
    - reset
    - no read after last write
        - (rewrite)
- Direct Access
    - read *n*
    - write *n*
    - position to *n*
        - read next
        - write next
    - rewrite *n*
- *n* = relative block number

Sequential-access File

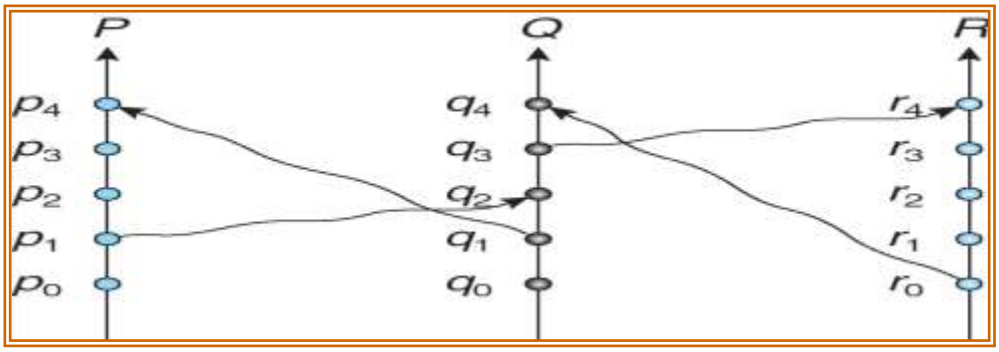


Event Ordering

1. *Happened-before* relation (denoted by $\rightarrow$)

- If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$
- If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$
- If $A \rightarrow B$ and B $\rightarrow C$ then $A \rightarrow C$

Relative Time for Three Concurrent Processes



Implementation of →

1. Associate a timestamp with each system event
    - Require that for every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B
2. Within each process Pi a logical clock, LCi is associated
    - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
        - Logical clock is monotonically increasing
3. A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock
4. If the timestamps of two events A and B are the same, then the events are concurrent
    - We may use the process identity numbers to break ties and to create a total ordering

Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Deadlock handling

Deadlock Prevention

1. Resource-ordering deadlock-prevention – define a *global* ordering among the system resources
    - Assign a unique number to all system resources
    - A process may request a resource with unique number *i* only if it is not holding a resource with a unique number grater than *i*

- Simple to implement; requires little overhead

2. Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm
    - Also implemented easily, but may require too much overhead

Timestamped Deadlock-Prevention Scheme

1. Each process *Pi* is assigned a unique priority number
2. Priority numbers are used to decide whether a process *Pi* should wait for a process *Pj*; otherwise *Pi* is rolled back
3. The scheme prevents deadlocks
    - For every edge $Pi \rightarrow Pj$ in the wait-for graph, *Pi* has a higher priority than *Pj*
    - Thus a cycle cannot exist

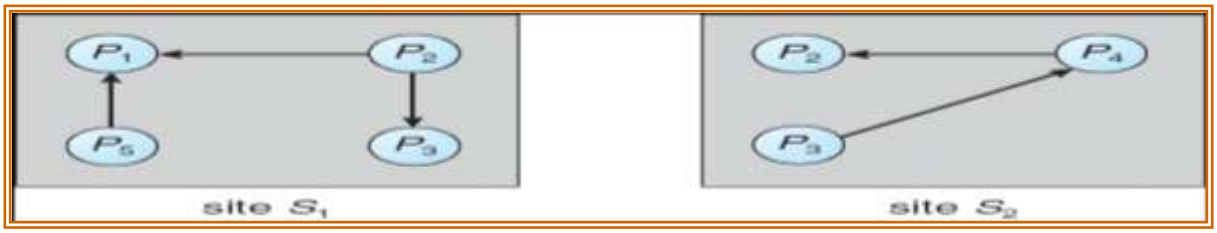
Wait-Die Scheme

1. Based on a nonpreemptive technique
2. If *Pi* requests a resource currently held by *Pj*, *Pi* is allowed to wait only if it has a smaller timestamp than does *Pj* (*Pi* is older than *Pj*)
    a. Otherwise, P*i* is rolled back (dies)
3. Example: Suppose that processes *P*1, *P*2, and *P*3 have timestamps t, 10, and 15 respectively
    a. if *P*1 request a resource held by *P*2, then *P*1 will wait
    b. If *P*3 requests a resource held by *P*2, then *P*3 will be rolled back

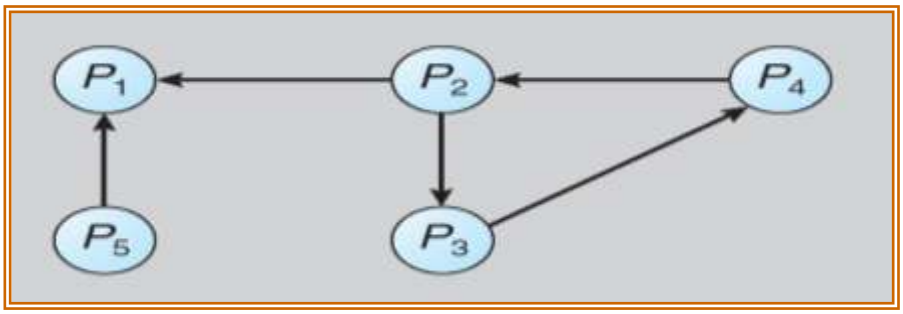
Would-Wait Scheme

1) Based on a preemptive technique; counterpart to the wait-die system
2) If *Pi* requests a resource currently held by *Pj*, *Pi* is allowed to wait only if it has a larger timestamp than does *Pj* (*Pi* is younger than *Pj*). Otherwise *Pj* is rolled back (*Pj* is wounded by *Pi*)
3) Example: Suppose that processes *P*1, *P*2, and *P*3 have timestamps 5, 10, and 15 respectively
    a) If *P*1 requests a resource held by *P*2, then the resource will be preempted from *P*2 and *P*2 will be rolled back
    b) If *P*3 requests a resource held by *P*2, then *P*3 will wait

Two Local Wait-For Graphs



Global Wait-For Graph



Deadlock Detection – Centralized Approach

1) Each site keeps a local wait-for graph
   a) The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
2) A global wait-for graph is maintained in a single coordination process; this graph is the union of all local wait-for graphs
3) There are three different options (points in time) when the wait-for graph may be constructed:

1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the coordinator needs to invoke the cycle-detection algorithm
   - Unnecessary rollbacks may occur as a result of false cycles
   - Append unique identifiers (timestamps) to requests form different sites
   - When process *Pi*, at site *A*, requests a resource from process *Pj*, at site *B*, a request message with timestamp *TS* is sent
   - The edge $Pi \rightarrow Pj$ with the label *TS* is inserted in the local wait-for of *A*. The edge is inserted in the local wait-for graph of *B* only if *B* has received the request message and cannot immediately grant the requested resource

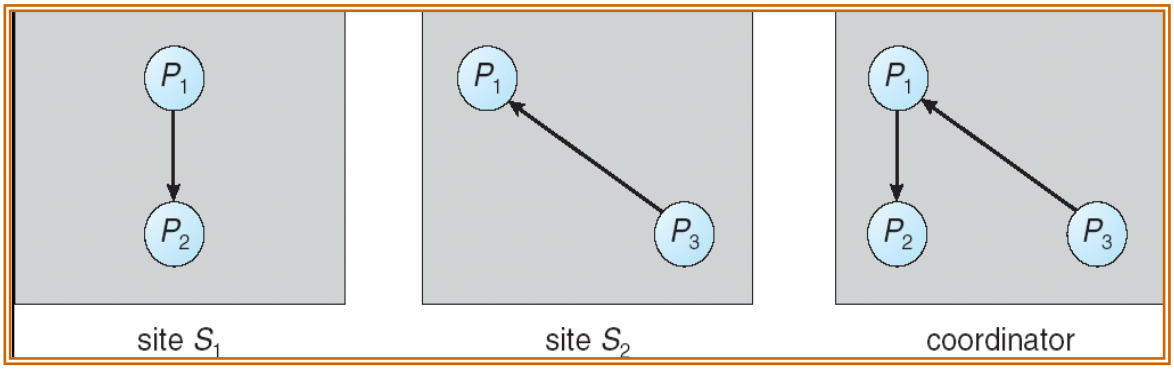
The Algorithm

The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:

(a) The constructed graph contains a vertex for every process in the system
(b) The graph has an edge Pi → Pj if and only if
(1) there is an edge Pi → Pj in one of the wait-for graphs, or
(2) an edge Pi → Pj with some label TS appears in more than one wait-for graph

If the constructed graph contains a cycle ⇒ deadlock

Local and Global Wait-For Graphs



Election Algorithms

- Determine where a new copy of the coordinator should be restarted
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process *Pi* is *i*
- Assume a one-to-one correspondence between processes and sites
- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures

Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system
- If process *Pi* sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed; *Pi* tries to elect itself as the new coordinator
- *P*i sends an election message to every process with a higher priority number, *Pi* then waits for any of these processes to answer within *T*
- If no response within *T*, assume that all processes with numbers greater than i have failed; *Pi* elects itself the new coordinator
- If answer is received, *Pi* begins time interval *T´*, waiting to receive a message that a process with a higher priority number has been elected

- If no message is sent within *T´*, assume the process with a higher number has failed; *Pi* should restart the algorithm
- If *Pi* is not the coordinator, then, at any time during execution, *Pi* may receive one of the following two messages from process *Pj*
- *Pj* is the new coordinator (*j* > *i*). *Pi*, in turn, records this information
- *Pj* started an election (*j* > *i*). *Pi*, sends a response to *Pj* and begins its own election algorithm, provided that *Pi* has not already initiated such an election
- After a failed process recovers, it immediately begins execution of the same algorithm
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number

Ring Algorithm

- Applicable to systems organized as a ring (logically or physically)
- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors
- Each process maintains an active list, consisting of all the priority numbers of all active processes in the system when the algorithm ends
- If process Pi detects a coordinator failure, I creates a new active list that is initially empty. It then sends a message elect(i) to its right neighbor, and adds the number i to its active list
- If *Pi* receives a message elect(*j*) from the process on the left, it must respond in one of three ways:
    - If this is the first *elect* message it has seen or sent, *Pi* creates a new active list with the numbers *i* and *j*
        - It then sends the message *elect(i),* followed by the message *elect(j)*
    - If $i \neq j$, then the active list for *Pi* now contains the numbers of all the active processes in the system
        - *Pi* can now determine the largest number in the active list to identify the new coordinator process
    - If *i = j*, then *Pi* receives the message *elect(i)*
        - The active list for *Pi* contains all the active processes in the system
            - *Pi* can now determine the new coordinator process.

Reaching Agreement

1) There are applications where a set of processes wish to agree on a common "value"
2) Such agreement may not take place due to:
    a) Faulty communication medium
    b) Faulty processes
        i) Processes may send garbled or incorrect messages to other processes
        ii) A subset of the processes may collaborate with each other in an attempt to defeat the scheme

Faulty Communications

1) Process *Pi* at site *A*, has sent a message to process *Pj* at site *B*; to proceed, *Pi* needs to know if *Pj* has received the message
2) Detect failures using a time-out scheme
   a) When *Pi* sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message form *Pj*
   b) When *Pj* receives the message, it immediately sends an acknowledgment to *Pi*
   c) If *Pi* receives the acknowledgment message within the specified time interval, it concludes that *Pj* has received its message
      i) If a time-out occurs, *Pj* needs to retransmit its message and wait for an acknowledgment
   d) Continue until *Pi* either receives an acknowledgment, or is notified by the system that *B* is down
3) Suppose that *Pj* also needs to know that *Pi* has received its acknowledgment message, in order to decide on how to proceed
   a) In the presence of failure, it is not possible to accomplish this task
   b) It is not possible in a distributed environment for processes *Pi* and *Pj* to agree completely on their respective states

Faulty Processes (Byzantine Generals Problem)

1) Communication medium is reliable, but processes can fail in unpredictable ways
2) Consider a system of n processes, of which no more than m are faulty
   a) Suppose that each process *Pi* has some private value of *Vi*
3) Devise an algorithm that allows each nonfaulty Pi to construct a vector $Xi = (Ai,1, Ai,2, \ldots, Ai,n)$ such that::
   a) If *Pj* is a nonfaulty process, then *Aij* = *Vj*.
   b) If *Pi* and *Pj* are both nonfaulty processes, then *X*i = *Xj*.
4) Solutions share the following properties
   a) A correct algorithm can be devised only if $n \geq 3 \times m + 1$
   b) The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays

UNIX SYSTEM

History

First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS.The third version was written in C, which was developed at BellLabs specifically to support UNIX.The most influential of the non-Bell Labs and non-AT&T UNIX development groups — University of California at Berkeley (Berkeley Software Distributions).

– 4BSD UNIX resulted from DARPA funding to develop a standard UNIX system for government use.

– Developed for the VAX, 4.3BSD is one of the most influential versions, and has been ported to many other platforms.

UNIX Design Principles

- Designed to be a time-sharing system.
- Has a simple standard user interface (shell) that can be replaced.
- File system with multilevel tree-structured directories.
- Files are supported by the kernel as unstructured sequences of bytes.
- Supports multiple processes; a process can easily create new processes.
- High priority given to making system interactive, and providing facilities for program development.

Programmer Interface

Like most computer systems, UNIX consists of two separable parts:

1) Kernel: everything below the system-call interface and above the physical hardware.
2) Provides file system, CPU scheduling, memory management, and other OS functions through system calls.
3) System programs: use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

User Interface

Programmers and users mainly deal with already existing systems programs: the needed system calls are embedded within the program and do not need to be obvious to the user.
The most common systems programs are file or directory

– Directory: mkdir, rmdir, cd, pwd
– File: ls, cp, mv, rm

Other programs relate to editors (e.g., emacs, vi) text formatters
(e.g., troff, TEX), and other activities.

File Manipulation

1) A file is a sequence of bytes; the kernel does not impose a structure on files.
2) Files are organized in tree-structured directories.
3) Directories are files that contain information on how to find other files.
4) Path name: identifies a file by specifying a path through the directory structure to the file.
5) Absolute path names start at root of file system
6) Relative path names start at the current directory
7) System calls for basic file manipulation: create, open, read, write, close, unlink, trunc.

8) The UNIX file system supports two main objects: files and directories.
9) Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

Blocks and Fragments

Mos of the file system is taken up by data blocks.
4.2BSD uses two block sized for files which have no indirect blocks:
– All the blocks of a file are of a large block size (such as 8K), except the last.
– The last block is an appropriate multiple of a smaller fragment size (i.e., 1024) to fill out the file.
– Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

The block and fragment sizes are set during file-system creation according to the ntended use of the file system:
– If many small files are expected, the fragment size should be small.
– If repeated transfers of large files are expected, the basic block size should be large.
The maximum block-to-fragment ratio is 8 : 1; the minimum block size is 4K (typical choices are 4096 : 512 and 8192 :
1024).

Process Management

- Representation of processes is a major design problem for operating system.
- UNIX is distinct from other systems in that multiple processes can be created and manipulated with ease.

These processes are represented in UNIX by various control blocks.
– Control blocks associated with a process are stored in the kernel.
– Information in these control blocks is used by the kernel for process control and CPU scheduling.

Memory Management

- The initial memory management schemes were constrained in size by the relatively small memory resources of the PDP machines on which UNIX was developed.
- Pre 3BSD system use swapping exclusively to handle memory contention among processes: If there is too much contention, processes are swapped out until enough memory is available.
- Allocation of both main memory and swap space is done first-fit.
- required for multiple processes using the same text segment.
- The scheduler process (or swapper) decides which processes to swap in or out, considering such factors as time idle, time in or out of main memory, size, etc.
- In f.3BSD, swap space is allocated in pieces that are multiples of power of 2 and minimum size, up to a maximum size determined by the size or the swap-space partition on the disk.

I/O System

The I/O system hides the peculiarities of I/O devices from the bulk of the kernel.
Consists of a buffer caching system, general device driver code, and drivers for specific hardware devices.
Only the device driver knows the peculiarities of a specific device.

Interprocess Communication

- Most UNIX systems have not permitted shared memory because the PDP-11 hardware did not encourage it.
- The pipe is the IPC mechanism most characteristic of UNIX.

– Permits a reliable unidirectional byte stream between two processes.
– A benefit of pipes small size is that pipe data are seldom written to disk; they usually are kept in memory by the normal block buffer cache.

- In 4.3BSD, pipes are implemented as a special case of the socket mechanism which provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

Linux operating system

History

- Linux is a modern, free operating system based on UNIX standards
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Many, varying Linux Distributions including the kernel, applications, and management tools

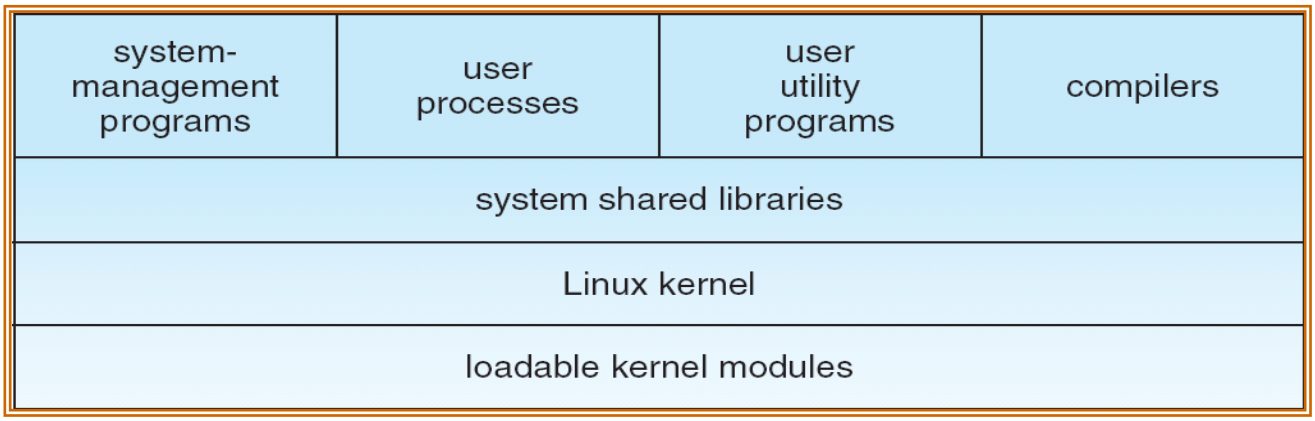
The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project
- The min system libraries were started by the GNU project, with improvements provided by the Linux community
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Components of a Linux System



- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components
- The kernel is responsible for maintaining the important abstractions of the operating system
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer
  - All kernel code and data structures are kept in the same single address space

UNIT V

Design principle

Goals

Based on market requirements and Microsoft's development strategy, the original Microsoft NT design team established a set of prioritized goals. Note that from the outset, the priority design objectives of the Windows NT operating system were *robustness* and *extensibility*:

Robustness. The operating system must actively protect itself from internal malfunction and external damage (whether accidental or deliberate), and must respond predictably to software and hardware errors. The system must be straightforward in its architecture and coding practices, and interfaces and behavior must be well- specified.

Extensibility and maintainability. Windows NT must be designed with the future in mind. It must grow to meet the future needs of original equipment manufacturers (OEMs) and Microsoft. And the system must be designed for maintainability, it must accommodate changes and additions to the API sets it supports and the APIs should not employ flags or other devices that drastically alter their functionality.

Portability. The system architecture must be able to function on a number of platforms with minimal recoding.

Performance. Algorithms and data structures that lead to a high level of performance and that provide the flexibility needed to achieve our other goals must be incorporated into the design.

POSIX compliance and government certifiable C2 security. The POSIX standard calls for operating system vendors to implement UNIX-style interfaces so that applications can be moved easily from one system to another. U.S. government security guidelines specify certain protections, such as auditing capabilities, access detection, per-user resource quotas, and resource protection. Inclusion of these features would allow Windows NT to be used in government operations.

Mechanisms and polices

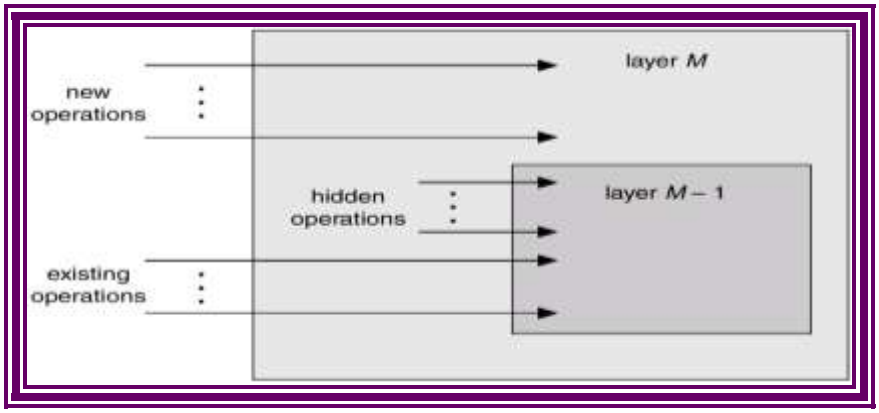
A **policy** is a plan of action to guide decisions and actions. The term may apply to government, private sector organizations and groups, and individuals. The policy process includes the identification of different alternatives, such as programs or spending priorities, and choosing among them on the basis of the impact they will have. Policies can be understood as political, management, financial, and administrative mechanisms arranged to reach explicit goals.

The separation of policy and mechanism is very important for flexibility. Policies are likely to change from place to place or time to time. A general mechanism would be more desirable.

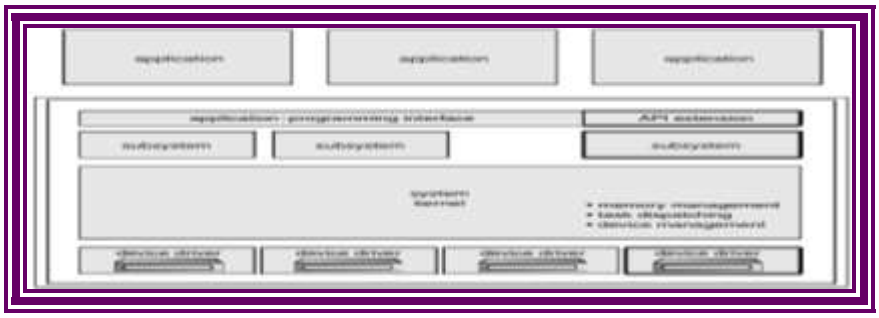
Layered approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

An Operating System Layer



OS/2 Layer Structure

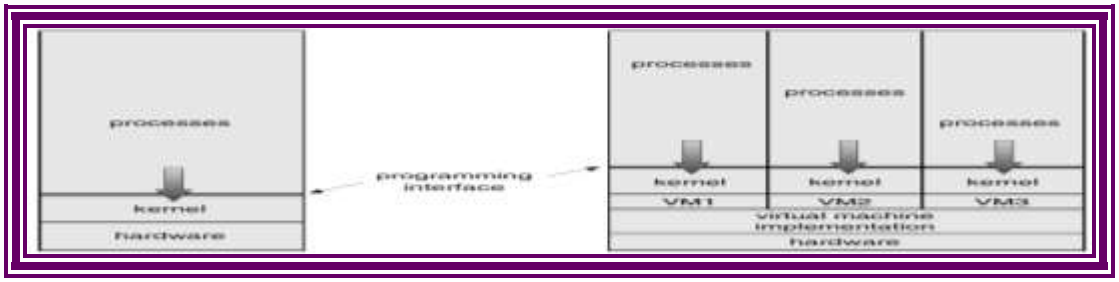


Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.
- The resources of the physical computer are shared to create the virtual machines**.**
- CPU scheduling can create the appearance that users have their own processor.

- Spooling and a file system can provide virtual card readers and virtual line printers.
- A normal user time-sharing terminal serves as the virtual machine operator's console.

System Models



Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

Multiprocessor

A **multiprocessor** computer is one with more than one CPU. The category of multiprocessor computers can be divided into the following sub-categories:

- **shared memory multiprocessors** have multiple CPUs, all with access to the same memory. Communication between the the processors is easy to implement, but care must be taken so that memory accesses are synchronized.
- **distributed memory multiprocessors** also have multiple CPUs, but each CPU has it's own associated memory. Here, memory access synchronization is not a problem, but communication between the processors is often slow and complicated.

Related to multiprocessors are the following:

- **networked systems** consist of multiple computers that are networked together, usually with a common operating system and shared resources. Users, however, are aware of the different computers that make up the system.
- **distributed systems** also consist of multiple computers but differ from networked systems in that the multiple computers are transparent to the user. Often there are

redundant resources and a sharing of the workload among the different computers, but this is all transparent to the user.

System Implementation

1. Traditionally written in assembly language, operating systems can now be written in higher-level languages.
2. Code written in a high-level language:

    - can be written faster.
    - is more compact.
    - is easier to understand and debug.

3. An operating system is far easier to *port* (move to some other hardware) if it is written in a high-level language.

System Generation (SYSGEN)

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
- *Booting* – starting a computer by loading the kernel.
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.

Distributed system

Motivation

1. **Distributed system** is collection of loosely coupled processors interconnected by a communications network
2. Processors variously called *nodes, computers, machines, hosts*

    - *Site* is location of the processor

3. Reasons for distributed systems
    - Resource sharing
        - sharing and printing files at remote sites
        - processing information in a distributed database
        - using remote specialized hardware devices
    - Computation speedup – **load sharing**
    - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
    - Communication – message passing

Network Topology

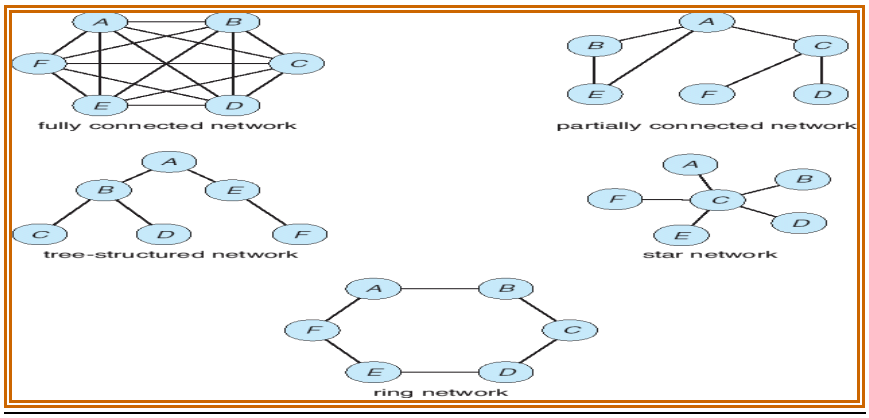
1. Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:

    - **Basic cost** - How expensive is it to link the various sites in the system?
    - **Communication cost** - How long does it take to send a message from site *A* to site *B*?
    - **Reliability** - If a link or a site in the system fails, can the remaining sites still communicate with each other?

2. The various topologies are depicted as graphs whose nodes correspond to sites

    - An edge from node *A* to node *B* corresponds to a direct connection between the two sites

3. The following six items depict various network topologies



Communication Structure

The design of a *communication* network must address four basic issues:

- **Naming and name resolution** - How do two processes locate each other to communicate?
- **Routing strategies** - How are messages sent through the network?
- **Connection strategies** - How do two processes send a sequence of messages?
- **Contention** - The network is a shared resource, so how do we resolve conflicting demands for its use?

Naming and Name Resolution

- Name systems in the network
- Address messages with the process-id
- Identify processes on remote systems by

<host-name, identifier> pair

- *Domain name service* (DNS) – specifies the naming structure of the hosts, as well as name to address resolution (Internet)

Routing Strategies

1. **Fixed routing** - A path from *A* to *B* is specified in advance; path changes only if a hardware failure disables it

    - Since the shortest path is usually chosen, communication costs are minimized
    - Fixed routing cannot adapt to load changes
    - Ensures that messages will be delivered in the order in which they were sent

2. **Virtual circuit** - A path from *A* to *B* is fixed for the duration of one session. Different sessions involving messages from *A* to *B* may have different paths
    - l Partial remedy to adapting to load changes
    - l Ensures that messages will be delivered in the order in which they were sent

Dynamic routing - The path used to send a message form site *A* to site *B* is chosen only when a message is sent

- Usually a site sends a message to another site on the link least used at that particular time
- Adapts to load changes by avoiding routing messages on heavily used path
- Messages may arrive out of order
- This problem can be remedied by appending a sequence number to each message

Connection Strategies

1. **Circuit switching** - A permanent physical link is established for the duration of the communication (i.e., telephone system)
2. **Message switching** - A temporary link is established for the duration of one message transfer (i.e., post-office mailing system)
3. **Packet switching** - Messages of variable length are divided into fixed-length packets which are sent to the destination

- Each packet may take a different path through the network
- The packets must be reassembled into messages as they arrive

4. Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth

    - Message and packet switching require less setup time, but incur more overhead per message

Contention

Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

1. **CSMA/CD** - Carrier sense with multiple access (CSMA); collision detection (CD)

    - A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting
    - When the system is very busy, many collisions may occur, and thus performance may be degraded

2. CSMA/CD is used successfully in the Ethernet system, the most common network system
3. **Token passing** - A unique message type, known as a token, continuously circulates in the system (usually a ring structure)

    - A site that wants to transmit information must wait until the token arrives
    - When the site completes its round of message passing, it retransmits the token
    - A token-passing scheme is used by some IBM and HP/Apollo systems

4. **Message slots** - A number of fixed-length message slots continuously circulate in the system (usually a ring structure)

    - Since a slot can contain only fixed-sized messages, a single logical message may have to be broken down into a number of smaller packets, each of which is sent in a separate slot
    - This scheme has been adopted in the experimental Cambridge Digital Communication Ring

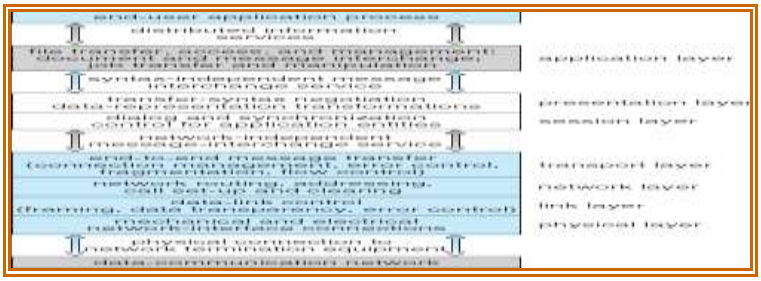
Communication Protocol

The communication network is partitioned into the following multiple layers:

- **Physical layer** – handles the mechanical and electrical details of the physical transmission of a bit stream
- **Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer
- **Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels
- **Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses
- **Session layer** – implements sessions, or process-to-process communications protocols
- **Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing)
- **Application layer** – interacts directly with the users' deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases

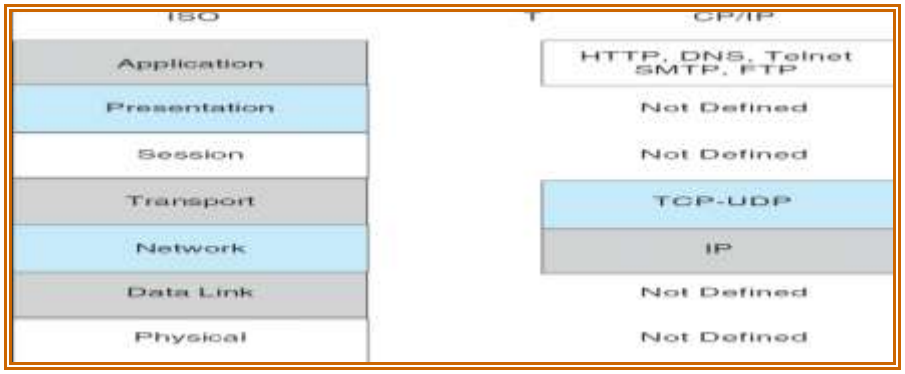
Communication Via ISO Network Model

The ISO Protocol Layer



The ISO Network Message

The TCP/IP Protocol Layers



File Concept

1. Contiguous logical address space
2. Types:
    - Data
        - numeric
        - character
        - binary
    - Program

File Structure

1. None - sequence of words, bytes
2. Simple record structure
    - Lines
    - Fixed length
    - Variable length
3. Complex Structures

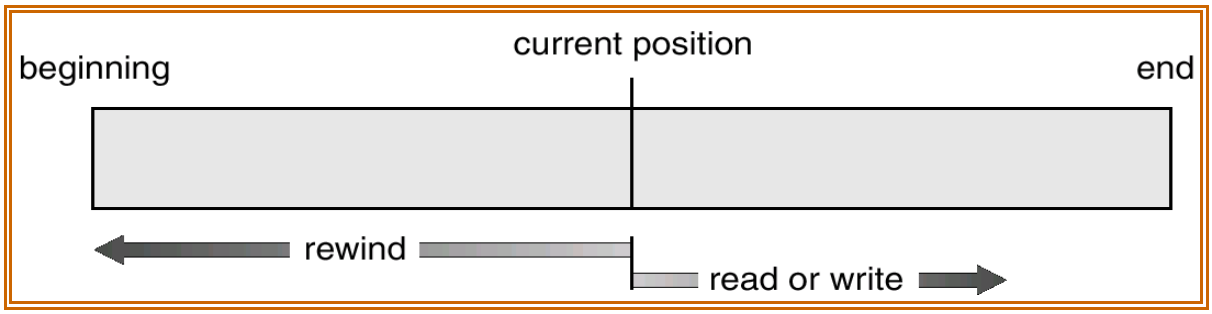
- Formatted document
- Relocatable load file

4. Can simulate last two with first method by inserting appropriate control characters
5. Who decides:
    - Operating system
    - Program

Modes of computation

- Sequential Access
    - read next
    - write next
    - reset
    - no read after last write
        - (rewrite)
- Direct Access
    - read *n*
    - write *n*
    - position to *n*
        - read next
        - write next
    - rewrite *n*
- *n* = relative block number

Sequential-access File

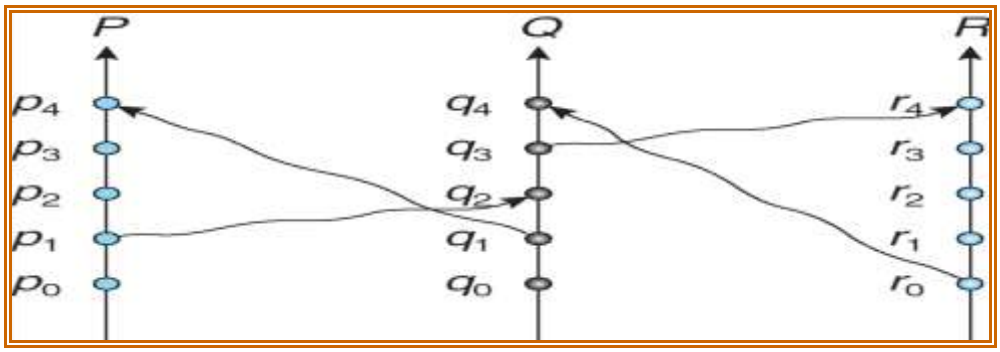


Event Ordering

1. *Happened-before* relation (denoted by $\rightarrow$)

- If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$
- If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$
- If $A \rightarrow B$ and B $\rightarrow C$ then $A \rightarrow C$

Relative Time for Three Concurrent Processes



Implementation of →

1. Associate a timestamp with each system event
    - Require that for every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B
2. Within each process Pi a logical clock, LCi is associated
    - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
        - Logical clock is monotonically increasing
3. A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock
4. If the timestamps of two events A and B are the same, then the events are concurrent
    - We may use the process identity numbers to break ties and to create a total ordering

Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Deadlock handling

Deadlock Prevention

1. Resource-ordering deadlock-prevention – define a *global* ordering among the system resources
    - Assign a unique number to all system resources

- A process may request a resource with unique number *i* only if it is not holding a resource with a unique number grater than *i*
- Simple to implement; requires little overhead

2. Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm
    - Also implemented easily, but may require too much overhead

Timestamped Deadlock-Prevention Scheme

1. Each process *Pi* is assigned a unique priority number
2. Priority numbers are used to decide whether a process *Pi* should wait for a process *Pj*; otherwise *Pi* is rolled back
3. The scheme prevents deadlocks
    - For every edge $Pi \rightarrow Pj$ in the wait-for graph, *Pi* has a higher priority than *Pj*
    - Thus a cycle cannot exist

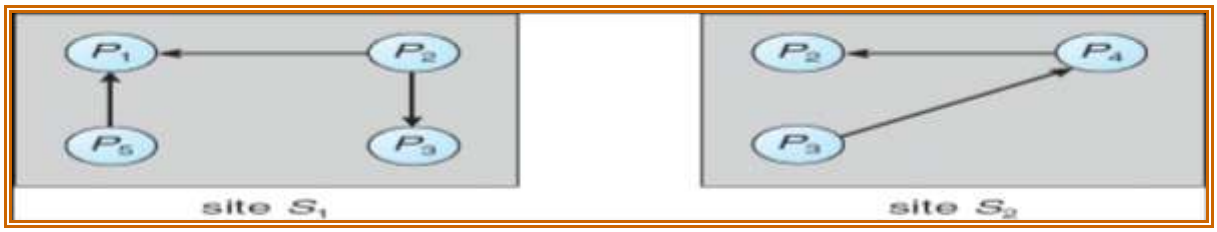
Wait-Die Scheme

1. Based on a nonpreemptive technique
2. If *Pi* requests a resource currently held by *Pj*, *Pi* is allowed to wait only if it has a smaller timestamp than does *Pj* (*Pi* is older than *Pj*)
    a. Otherwise, P*i* is rolled back (dies)
3. Example: Suppose that processes *P*1, *P*2, and *P*3 have timestamps t, 10, and 15 respectively
    a. if *P*1 request a resource held by *P*2, then *P*1 will wait
    b. If *P*3 requests a resource held by *P*2, then *P*3 will be rolled back

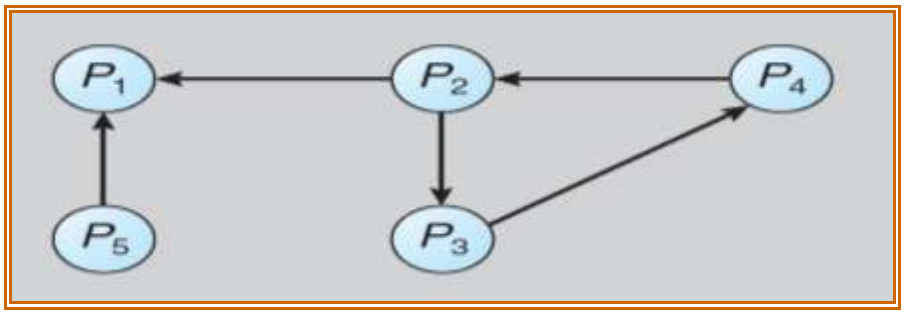
Would-Wait Scheme

1) Based on a preemptive technique; counterpart to the wait-die system
2) If *Pi* requests a resource currently held by *Pj*, *Pi* is allowed to wait only if it has a larger timestamp than does *Pj* (*Pi* is younger than *Pj*). Otherwise *Pj* is rolled back (*Pj* is wounded by *Pi*)
3) Example: Suppose that processes *P*1, *P*2, and *P*3 have timestamps 5, 10, and 15 respectively
    a) If *P*1 requests a resource held by *P*2, then the resource will be preempted from *P*2 and *P*2 will be rolled back
    b) If *P*3 requests a resource held by *P*2, then *P*3 will wait

Two Local Wait-For Graphs



Global Wait-For Graph



Deadlock Detection – Centralized Approach

1) Each site keeps a local wait-for graph
   a) The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
2) A global wait-for graph is maintained in a single coordination process; this graph is the union of all local wait-for graphs
3) There are three different options (points in time) when the wait-for graph may be constructed:

1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the coordinator needs to invoke the cycle-detection algorithm
   - Unnecessary rollbacks may occur as a result of false cycles
   - Append unique identifiers (timestamps) to requests form different sites
   - When process *Pi*, at site *A*, requests a resource from process *Pj*, at site *B*, a request message with timestamp *TS* is sent
   - The edge $Pi \rightarrow Pj$ with the label *TS* is inserted in the local wait-for of *A*. The edge is inserted in the local wait-for graph of *B* only if *B* has received the request message and cannot immediately grant the requested resource

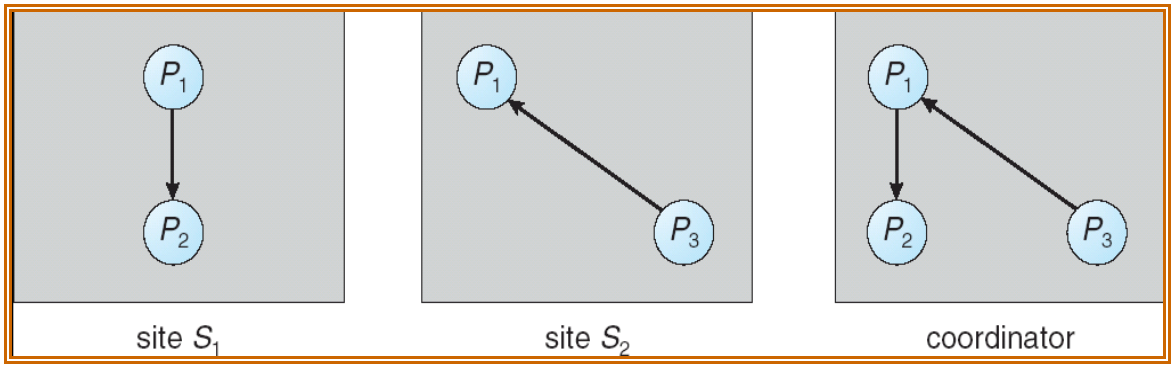
The Algorithm

The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:

(a) The constructed graph contains a vertex for every process in the system
(b) The graph has an edge Pi → Pj if and only if
(1) there is an edge Pi → Pj in one of the wait-for graphs, or
(2) an edge Pi → Pj with some label TS appears in more than one wait-for graph

If the constructed graph contains a cycle ⇒ deadlock

Local and Global Wait-For Graphs



Election Algorithms

- Determine where a new copy of the coordinator should be restarted
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process *Pi* is *i*
- Assume a one-to-one correspondence between processes and sites
- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures

Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system
- If process *Pi* sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed; *Pi* tries to elect itself as the new coordinator
- *P*i sends an election message to every process with a higher priority number, *Pi* then waits for any of these processes to answer within *T*
- If no response within *T*, assume that all processes with numbers greater than i have failed; *Pi* elects itself the new coordinator
- If answer is received, *Pi* begins time interval *T´*, waiting to receive a message that a process with a higher priority number has been elected

- If no message is sent within *T´*, assume the process with a higher number has failed; *Pi* should restart the algorithm
- If *Pi* is not the coordinator, then, at any time during execution, *Pi* may receive one of the following two messages from process *Pj*
- *Pj* is the new coordinator (*j* > *i*). *Pi*, in turn, records this information
- *Pj* started an election (*j* > *i*). *Pi*, sends a response to *Pj* and begins its own election algorithm, provided that *Pi* has not already initiated such an election
- After a failed process recovers, it immediately begins execution of the same algorithm
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number

Ring Algorithm

- Applicable to systems organized as a ring (logically or physically)
- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors
- Each process maintains an active list, consisting of all the priority numbers of all active processes in the system when the algorithm ends
- If process Pi detects a coordinator failure, I creates a new active list that is initially empty. It then sends a message elect(i) to its right neighbor, and adds the number i to its active list
- If *Pi* receives a message elect(*j*) from the process on the left, it must respond in one of three ways:
    - If this is the first *elect* message it has seen or sent, *Pi* creates a new active list with the numbers *i* and *j*
        - It then sends the message *elect(i),* followed by the message *elect(j)*
    - If $i \neq j$, then the active list for *Pi* now contains the numbers of all the active processes in the system
        - *Pi* can now determine the largest number in the active list to identify the new coordinator process
    - If *i = j*, then *Pi* receives the message *elect(i)*
        - The active list for *Pi* contains all the active processes in the system
            - *Pi* can now determine the new coordinator process.

Reaching Agreement

1) There are applications where a set of processes wish to agree on a common "value"
2) Such agreement may not take place due to:
    a) Faulty communication medium
    b) Faulty processes
        i) Processes may send garbled or incorrect messages to other processes
        ii) A subset of the processes may collaborate with each other in an attempt to defeat the scheme

Faulty Communications

1) Process *Pi* at site *A*, has sent a message to process *Pj* at site *B*; to proceed, *Pi* needs to know if *Pj* has received the message
2) Detect failures using a time-out scheme
   a) When *Pi* sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message form *Pj*
   b) When *Pj* receives the message, it immediately sends an acknowledgment to *Pi*
   c) If *Pi* receives the acknowledgment message within the specified time interval, it concludes that *Pj* has received its message
      i) If a time-out occurs, *Pj* needs to retransmit its message and wait for an acknowledgment
   d) Continue until *Pi* either receives an acknowledgment, or is notified by the system that *B* is down
3) Suppose that *Pj* also needs to know that *Pi* has received its acknowledgment message, in order to decide on how to proceed
   a) In the presence of failure, it is not possible to accomplish this task
   b) It is not possible in a distributed environment for processes *Pi* and *Pj* to agree completely on their respective states

Faulty Processes (Byzantine Generals Problem)

1) Communication medium is reliable, but processes can fail in unpredictable ways
2) Consider a system of n processes, of which no more than m are faulty
   a) Suppose that each process *Pi* has some private value of *Vi*
3) Devise an algorithm that allows each nonfaulty Pi to construct a vector $Xi = (Ai,1, Ai,2, \ldots, Ai,n)$ such that::
   a) If *Pj* is a nonfaulty process, then $Aij = Vj$.
   b) If *Pi* and *Pj* are both nonfaulty processes, then $Xi = Xj$.
4) Solutions share the following properties
   a) A correct algorithm can be devised only if $n \geq 3 \times m + 1$
   b) The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays

UNIX SYSTEM

History

First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS.The third version was written in C, which was developed at BellLabs specifically to support UNIX.The most influential of the non-Bell Labs and non-AT&T UNIX development groups — University of California at Berkeley (Berkeley Software Distributions).

– 4BSD UNIX resulted from DARPA funding to develop a standard UNIX system for government use.

– Developed for the VAX, 4.3BSD is one of the most influential versions, and has been ported to many other platforms.

UNIX Design Principles

- Designed to be a time-sharing system.
- Has a simple standard user interface (shell) that can be replaced.
- File system with multilevel tree-structured directories.
- Files are supported by the kernel as unstructured sequences of bytes.
- Supports multiple processes; a process can easily create new processes.
- High priority given to making system interactive, and providing facilities for program development.

Programmer Interface

Like most computer systems, UNIX consists of two separable parts:

1) Kernel: everything below the system-call interface and above the physical hardware.
2) Provides file system, CPU scheduling, memory management, and other OS functions through system calls.
3) System programs: use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

User Interface

Programmers and users mainly deal with already existing systems programs: the needed system calls are embedded within the program and do not need to be obvious to the user.
The most common systems programs are file or directory

– Directory: mkdir, rmdir, cd, pwd
– File: ls, cp, mv, rm

Other programs relate to editors (e.g., emacs, vi) text formatters
(e.g., troff, TEX), and other activities.

File Manipulation

1) A file is a sequence of bytes; the kernel does not impose a structure on files.
2) Files are organized in tree-structured directories.
3) Directories are files that contain information on how to find other files.
4) Path name: identifies a file by specifying a path through the directory structure to the file.
5) Absolute path names start at root of file system
6) Relative path names start at the current directory
7) System calls for basic file manipulation: create, open, read, write, close, unlink, trunc.

8) The UNIX file system supports two main objects: files and directories.
9) Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

Blocks and Fragments

Mos of the file system is taken up by data blocks.
4.2BSD uses two block sized for files which have no indirect blocks:
– All the blocks of a file are of a large block size (such as 8K), except the last.
– The last block is an appropriate multiple of a smaller fragment size (i.e., 1024) to fill out the file.
– Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

The block and fragment sizes are set during file-system creation according to the ntended use of the file system:
– If many small files are expected, the fragment size should be small.
– If repeated transfers of large files are expected, the basic block size should be large.
The maximum block-to-fragment ratio is 8 : 1; the minimum block size is 4K (typical choices are 4096 : 512 and 8192 :
1024).

Process Management

- Representation of processes is a major design problem for operating system.
- UNIX is distinct from other systems in that multiple processes can be created and manipulated with ease.

These processes are represented in UNIX by various control blocks.
– Control blocks associated with a process are stored in the kernel.
– Information in these control blocks is used by the kernel for process control and CPU scheduling.

Memory Management

- The initial memory management schemes were constrained in size by the relatively small memory resources of the PDP machines on which UNIX was developed.
- Pre 3BSD system use swapping exclusively to handle memory contention among processes: If there is too much contention, processes are swapped out until enough memory is available.
- Allocation of both main memory and swap space is done first-fit.
- required for multiple processes using the same text segment.
- The scheduler process (or swapper) decides which processes to swap in or out, considering such factors as time idle, time in or out of main memory, size, etc.
- In f.3BSD, swap space is allocated in pieces that are multiples of power of 2 and minimum size, up to a maximum size determined by the size or the swap-space partition on the disk.

I/O System

The I/O system hides the peculiarities of I/O devices from the bulk of the kernel.
Consists of a buffer caching system, general device driver code, and drivers for specific hardware devices.
Only the device driver knows the peculiarities of a specific device.

Interprocess Communication

- Most UNIX systems have not permitted shared memory because the PDP-11 hardware did not encourage it.
- The pipe is the IPC mechanism most characteristic of UNIX.

– Permits a reliable unidirectional byte stream between two processes.
– A benefit of pipes small size is that pipe data are seldom written to disk; they usually are kept in memory by the normal block buffer cache.

- In 4.3BSD, pipes are implemented as a special case of the socket mechanism which provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

Linux operating system

History

- Linux is a modern, free operating system based on UNIX standards
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Many, varying Linux Distributions including the kernel, applications, and management tools

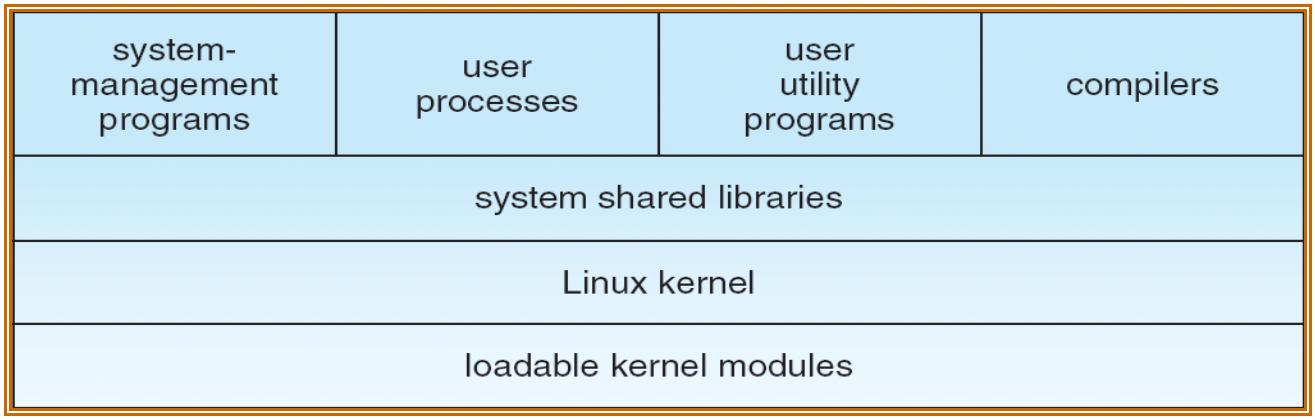
The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project
- The min system libraries were started by the GNU project, with improvements provided by the Linux community
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Components of a Linux System



- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components
- The kernel is responsible for maintaining the important abstractions of the operating system
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer
  - All kernel code and data structures are kept in the same single address space